

Software Communications Architecture Specification

APPENDIX C CORE FRAMEWORK IDL

Revision Summary

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

Change Proposals are controlled by the JTRS Change Control Board. CPs incorporated into the SCA are considered "closed" and can be seen on the JTRS web site at:
www.jtrs.saalt.army.mil/docs/documents/sca_ccb.html.

Table of Contents

APPENDIX C Core Framework IDL..... C-1
C.1 Core Framework IDL..... C-1
C.2 PortTypes Module..... C-65
C.3 LogService Module..... C-70
C.4 StandardEvent Module..... C-79

APPENDIX C CORE FRAMEWORK IDL

The CF interfaces are expressed in CORBA IDL. The IDL has been generated directly by the Rational Rose UML software modeling tool. This “forward engineering” approach ensures that the IDL accurately reflects the architecture definition as contained in the UML models. 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 *CF::Port* interface by defining basic data sequence types. The LogService CORBA Module contains the interfaces and types for a log service. Figure C-1 shows the relationship between these CORBA modules. The StandardEvent CORBA Module contains the standard event types to be passed via the event service.

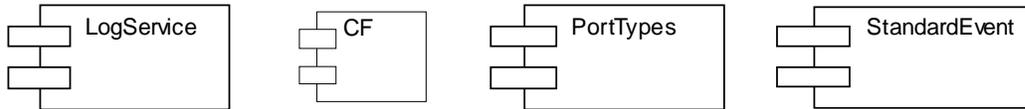


Figure C-1. Relationships Between CORBA Modules

The IDL modules are also available in electronic form.

C.1 CORE FRAMEWORK IDL.

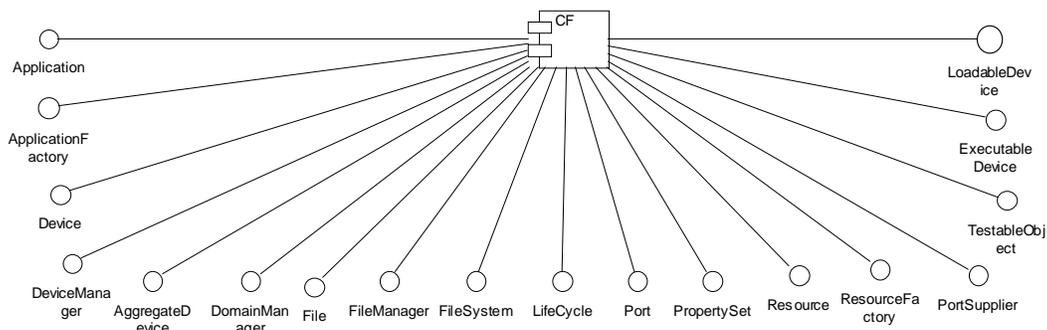


Figure C-2. CF CORBA Module

The following is the CF IDL generated from the Rational Rose model, version 2000e.

```
//Source file: CF.idl

#ifndef __CF_DEFINED
#define __CF_DEFINED

/* CmIdentification
   %X% %Q% %Z% %W% */
```

```
/* This package provides the main framework for all objects within the radio.
*/

module CF {
    interface File;
    interface Resource;
    interface Application;
    interface Device;
    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 (e.g.,
frequency, preset, etc.).
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 {
        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 "E" map the POSIX definitions, and can be
found in
IEEE Std 1003.1 1996 Edition. Those exceptions starting with CF are
defined below:
CFNOTSET is not defined in the POSIX specification. CFNOTSET 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 {
    CFNOTSET,
    E2BIG,
    EACCES,
    EAGAIN,
    EBADF,
    EBADMSG,
    EBUSY,
    ECANCELED,
    ECHILD,
    EDEADLK,
    EDOM,
    EEXIST,
    EFAULT,
    EFBIG,
    EINPROGRESS,
    EINTR,
    EINVAL,
    EIO,
    EISDIR,
    EMFILE,
    EMLINK,
    EMSGSIZE,
    ENAMETOOLONG,
    ENFILE,
    ENODEV,
    ENOENT,
    ENOEXEC,
    ENOLCK,
```

```

    ENOMEM,
    ENOSPC,
    ENOSYS,
    ENOTDIR,
    ENOTEMPTY,
    ENOTSUP,
    ENOTTY,
    ENXIO,
    EPERM,
    EPIPE,
    ERANGE,
    EROFS,
    ESPIPE,
    ESRCH,
    ETIMEDOUT,
    EXDEV
};

```

/* The InvalidFileName exception indicates an invalid file name was passed to a file service operation.

The error number indicates an ErrorNumberType value (e.g., ENAMETOOLONG). The message provides information describing why the filename was invalid. */

```

exception InvalidFileName {
    /* The error code that corresponds to the error message. */
    ErrorNumberType errorNumber;
    string msg;
};

```

/* The CF FileException indicates a file-related error occurred. The error number indicates an

ErrorNumberType value (e.g., EBADF, EEXIST, EISDIR, EMFILE, ENFILE, ENOENT, ENOSPC, ENOTDIR, ENOTEMPTY, EROFS). The message provides information describing the error. The message can be used for logging the error. */

```

exception FileException {
    /* The error code that corresponds to the error message. */
    ErrorNumberType errorNumber;
    string msg;
};

```

/* This type defines an unbounded sequence of Devices.

The IDL to Ada mapping has a problem with self referential interfaces. To get around this problem, the interface Device forward declaration has been created and this type has been moved outside of the Device interface. */

```

typedef sequence <Device> DeviceSequence;

```

/* This AggregateDevice interface provides aggregate behavior that can be used to add and remove Devices from an aggregateDevice. This interface can be

provided via inheritance or as a "provides port" for any Device that is capable of an aggregate relationship. Aggregated Devices use this interface to add or remove themselves from composite Devices when being created or torn-down. from composite Devices 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 DeviceSequence devices;

    /* The addDevice operation provides the mechanism to associate a
    Device with another Device.
    When a Device changes state or it is being torn down, its
    associated Devices are affected.
    The addDevice operation adds the input associatedDevice parameter
    to the AggregateDevice's
    devices attribute when the associatedDevice does not exist in the
    devices attribute.
    The associatedDevice is ignored when duplicated.
    The addDevice operation writes a FAILURE_ALARM log record, upon
    unsuccessful adding of an
    associatedDevice to the AggregateDevice's devices attribute.
    This operation does not return any value.
    The addDevice operation raises the CF InvalidObjectReference when
    the input associatedDevice
    is a nil CORBA object reference.
    @roseuid 3A5DAE9102D6 */
    void addDevice (
        in Device associatedDevice
    )
        raises (InvalidObjectReference);

    /* The removeDevice operation provides the mechanism to
    disassociate a Device from another Device.
    The removeDevice operation removes the input associatedDevice
    parameter from the
    AggregateDevice's devices attribute.
    The removeDevice operation writes a FAILURE_ALARM log record, upon
    unsuccessful removal of
    the associatedDevice from the AggregateDevice's devices attribute.
    This operation does not return any value.
    The removeDevice operation raises the CF InvalidObjectReference
    when the input
    associatedDevice is a nil CORBA object reference or does not exist
    in the aggregate
    Device's devices attribute.
    @roseuid 3A5DAE9102D8 */
    void removeDevice (
        in Device associatedDevice
    )

```

```

        raises (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 {
        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_SIZE = "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.
Not all the fields in the FileInformationType are applicable for
all file systems.
At a minimum, the FileSystem shall support name, kind, and size
information for a file.
Examples of other file properties that can be specified are created
time, modified time,
and last access time.

name: This field indicates the simple name of the file.
kind: This field indicates the type of the file entry.
size: This field indicates the size in octets. */

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

    typedef sequence <FileInformationType> FileInformationSequence;

```

/* The CREATED_TIME_ID is the identifier for the created time file property.

A created time property indicates the time the file was created. The value for

created time shall be long long and measured in seconds since 00:00:00 UTC, Jan. 1, 1970. */

```
const string CREATED_TIME_ID = "CREATED_TIME";
```

/* The MODIFIED_TIME_ID is the identifier for the modified time file property.

The modified time property is the time the file data was last modified. The value

for modified time property shall be long long and measured in seconds

```
since 00:00:00 UTC, Jan. 1, 1970. */
```

```
const string MODIFIED_TIME_ID = "MODIFIED_TIME";
```

/* The LAST_ACCESS_TIME_ID is the identifier for the last access time file property.

The last access time property is the time the file was last access (e.g. read).

The value for last access time property shall be long long and measured in seconds

```
since 00:00:00 UTC, Jan. 1, 1970. */
```

```
const string LAST_ACCESS_TIME_ID = "LAST_ACCESS_TIME";
```

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

that the filename is an absolute pathname of the file relative to the target FileSystem.

If an error occurs, this operation raises the appropriate exception:

CF InvalidFilename - The filename is not valid.

CF FileException - A file-related error occurred during the operation.

```
@roseuid 364B4B2E26B0 */
```

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

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

destination file with the specified destinationFileName. This operation ensures that the

sourceFileName and destinationFileName are absolute pathnames relative to the target FileSystem.

If an error occurs, this operation raises the appropriate exception:

CF InvalidFilename - The filename is not valid.

CF FileException - A file-related error occurred during the operation.

```
@roseuid 364B4B5A0640 */
```

```
void copy (
    in string sourceFileName,
```

```

        in string destinationFileName
        )
        raises (InvalidFileName, FileException);

    /* The exists operation checks to see if a file exists based on the
filename parameter.
    This operation ensures that the filename is a full pathname of the
file relative to the target
    FileSystem and raise an CF InvalidFileName exception if the name is
invalid.
    This operation shall return True if the file exists, otherwise
False shall be returned.
    @roseuid 3665751C2AA0 */
    boolean exists (
        in string fileName
        )
        raises (InvalidFileName);

    /* The list operation provides the ability to obtain a list of
files along with their
    information in the FileSystem according to a given search pattern.
    The list operation can be used to return information for one file
or for a set of files.
    The list operation returns a list of file information based upon
the search pattern given.
    The list operation supports the following wildcard characters for
base file names (i.e.,
    the part after the right-most slash):
    (1) * used to match any sequence of characters (including null).
    (2) ? used to match any single character.
    These wildcards may only be applied to the base filename in the
search pattern given.
    For example, the following are valid search patterns:
    /tmp/files/ *.* Returns all files and directories within the
/tmp/files directory.
    Directory names indicated with a "/" at the end of the name.
    /tmp/files/foo* Returns all files beginning with the letters
"foo" in the /tmp/files directory.
    /tmp/files/f?? Returns all 3 letter files beginning with the
letter f in the /tmp/files directory.

    The list operation returns a FileInformationSequence for files that
matchthe wildcard
    specification as specified in the input pattern parameter. The list
operation will
    return a zero length sequence when no file matching occurred for
the input pattern parameter.

    The list operation raises the InvalidFileName exception when the
input pattern does not start
    with a slash "/" or cannot be interpreted due to unexpected
characters.
    The list operation raises the FileException when a file-related
error occurs.
    @roseuid 36669644E5F0 */
    FileInformationSequence list (

```

```

        in string pattern
        )
        raises (FileException, InvalidFileName);

/* The create operation creates a new File based upon the provided
file name and
returns a File to the opened file. A null file is returned and a
related exception
shall be raised if an error occurs.
CF InvalidFilename - The filename is not valid.
CF FileException - File already exists or another file error
occurred.
@roseuid 36CAC30F37A8 */
File create (
    in string fileName
    )
    raises (InvalidFileName, FileException);

/* The open operation opens a file based upon the input fileName.
The read_Only parameter indicates if the file should be opened for
read access only.
When read_Only is false the file is opened for write access.
The open operation returns a File component parameter on successful
completion.
The open operation returns a null file component reference if the
open operation is unsuccessful.
If the file is opened with the read_Only flag set to true, then
writes to the file will be considered an error.
The open operation raises the CF FileException if the file does not
exist or another file error occurred.
The open operation raises the CF InvalidFilename exception when the
filename is not a
valid file name or not an absolute pathname.
@roseuid 36CAC3ECE2A0 */
File open (
    in string fileName,
    in boolean read_Only
    )
    raises (InvalidFileName, FileException);

/* The mkdir operation create a FileSystem directory based on the
directoryName given.
This operation creates all parent directories required to create
the directory path given.
If an error occurs, this operation raises the appropriate
exception.
Exceptions/Errors
CF InvalidFilename - The directory name is not valid.
CF FileException - A file-related error occurred during the
operation.
@roseuid 388F55390C58 */
void mkdir (
    in string directoryName
    )
    raises (InvalidFileName, FileException);

```

```

    /* The rmdir operation removes a FileSystem directory based on the
directoryName given.
    If an error occurs, this operation raises the appropriate
exception.
    Exceptions/Errors
    CF InvalidFilename - The directory name is not valid.
    CF FileException - Directory does not exist or another file-related
error occurred.
    @roseuid 388F554033F8 */
    void rmdir (
        in string directoryName
    )
        raises (InvalidFileName, FileException);

    /* The query operation returns file system information to the
calling client
    based upon the given fileSystemProperties' ID.
    As a minimum, the FileSystem query operation supports the following
fileSystemProperties:
    1. SIZE - an ID value of "SIZE causes query to return an unsigned
long long containing
    the file system size (in octets).
    2. AVAILABLE SPACE - an ID value of "AVAILABLE SPACE" causes the
query operation to
    return an unsigned long long containing the available space on the
file system (in octets).

    The query operation raises the UnknownFileSystemProperties
exception when
    the given file system property is not recognized.
    @roseuid 389196D696B0 */
    void query (
        inout Properties fileSystemProperties
    )
        raises (UnknownFileSystemProperties);

};

    /* The File interface provides the ability to read and write files
residing within a CF
    compliant 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.
    This filepointer points to the beginning of the file upon construction of
the file object.
    The File interface is modeled after the POSIX/C file interface. */

    interface File {
        /* The IOException exception indicates an error occurred during a
read or write
        operation to a File. The error number indicates an ErrorNumberType
value (e.g.,
        EFBIG, ENOSPC, EROFS). The message is component-dependent,
providing additional
        information describing the reason for the error. */

```

```

exception IOException {
    /* The error code that corresponds to the error message. */
    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.
The syntax for a filename is based upon the UNIX operating system.
That is, a sequence of directory
names separated by forward slashes (/) followed by the base
filename. The fileName attribute will
contain the filename given to the FileSystem::open 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.
The read operation reads, from the referenced file, the number of
octets specified by the
input length parameter and advance the value of the filePointer
attribute by the number of
octets actually read. The read operation reads less than the
number of octets specified in
the input-length parameter, when an end of file is encountered.
The read operation returns via the out Message parameter an CF
OctetSequence that equals the
number of octets actually read from the File. If the filePointer
attribute value reflects
the end of the File, the read operation returns a 0-length CF
OctetSequence.
The read operation raises the IOException when a read error occurs.
@roseuid 364B3D91DA40 */
void read (
    out OctetSequence data,
    in unsigned long length
)
    raises (IOException);

/* The write operation writes data to the file referenced. If the
write is successful,
the write operation shall increment the filePointer attribute to
reflect the number of octets written.
If the write is unsuccessful, the filePointer attribute value is
maintained or is restored to its

```

value prior to the write operation call.
 This operation does not return any value.
 The write operation raises the IOException when a write error occurs.

```
@roseuid 364B3DA2AFD0 */
void write (
    in OctetSequence data
)
    raises (IOException);
```

/* The sizeOf operation returns the current size of the file.
 The CF FileException is raised when a file-related error occurs
 (e.g. the file does not exist anymore).

```
@roseuid 36AE182BBF90 */
unsigned long sizeOf ()
    raises (FileException);
```

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

The close operation makes the file unavailable to the component. A client should release its CORBA File reference after closing the File. The close operation raises CF FileException exception when it cannot successfully close the file.

```
@roseuid 388E0477F138 */
void close ()
    raises (FileException);
```

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

The setFilePointer operation sets the filePointer attribute value to the input filePointer.

This operation does not return anyvalue.

The setFilePointer operation raises the CF FileException when the file pointer for the

referenced file cannot be set to the value of the input filePointer parameter.

The setFilePointer operation raises the InvalidFilePointer exception when the value of the

filePointer parameter exceeds the file size.

```
@roseuid 39088B800D38 */
void setFilePointer (
    in unsigned long filePointer
)
    raises (InvalidFilePointer, FileException);
```

```
};
```

/* A ResourceFactory is used to create and tear down a Resource. The ResourceFactory interface is

designed after the Factory Design Patterns. Each ResourceFactory object creates a specific type of

Resource within the radio. The ResourceFactory interface provides a one-step solution for creating a

Resource, reducing the overhead of starting up Resources. In CORBA, there are two separate object reference

counts. One for the client side and one for the server side. The Factory keeps a server-side reference count of the number of clients that have requested the resource. When a client is done with a resource, the client releases the client resource reference and calls `releaseResource` to the `ResourceFactory`.

When the server-side reference goes to zero, the server resource object is released from the ORB that causes the resource to be destroyed. */

```
interface ResourceFactory {
    /* This exception indicates the resource ID does not exist in the
ResourceFactory. */

    exception InvalidResourceId {
    };

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

    exception ShutdownFailure {
        /* This message indicates the reason for the shutdown
failure. */
        string msg;
    };

    /* The CreateResourceFailure exception indicates that the
createResource method failed to create the
Resource. The error number indicates an ErrorNumberType value
(e.g., NOTSET, EBADMSG, EINVAL,
EMSGSIZE, ENOMEM). The message is component-dependent, providing
additional information
describing the reason for the error. */

    exception CreateResourceFailure {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* The readonly identifier attribute contains the unique identifier
for a device 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.
The resourceId is the identifier for Resource. The qualifiers are
parameter values used by the
```

ResourceFactory in creation of the Resource. The ApplicationFactory can determine the values to be supplied for the qualifiers from the description in the ResourceFactory's Software Profile.

The qualifiers may be used to identify, for example, specific subtypes of Resources created by a ResourceFactory.

If no Resource exists for the given resourceId, the createResource operation creates a Resource.

If the Resource already exists, the Resource's reference is returned.

The createResource operation assigns the given resourceId to a new Resource and either set a reference count to one, when the Resource is initially created, or increment the reference count by one, when the Resource already exists. The reference count is used to indicate the number of times that a specific Resource reference has been given to requesting clients.

This ensures that the ResourceFactory does not release a Resource that has a reference count greater than 0. When multiple clients have obtained a reference to the same Resource, each client will request release of the Resource when through with the Resource.

However, the Resource must not be released until the release request comes from the last client in existence.

The createResource operation returns a reference to the created Resource or the existing Resource.

The createResource operation returns a nil CORBA component reference when the operation is unable to create the Resource

The createResource operation raises the CreateResourceFailure exception when it

```
cannot create the Resource.  
@roseuid 356B1F02C620 */  
Resource createResource (  
    in string resourceId,  
    in Properties qualifiers  
)  
    raises (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.

The releaseResource operation decrements the reference count for the specified resource, as indicated by the resourceId. The releaseResource operation makes the Resource no longer available (i.e., it is released from the CORBA environment) when the Resource's reference count is zero.

This operation does not return a value.

The releaseResource operation raises the InvalidResourceId exception if an

```
invalid resourceId is received.  
@roseuid 356B1F4E9140 */  
void releaseResource (  
    in string resourceId  
)  
    raises (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.

The shutdown operation results in the ResourceFactory being unavailable to any subsequent calls to its object reference (i.e. it is released from the CORBA environment).

This operation does not raise any exceptions.

The shutdown operation raises the ShutdownFailure exception for any error that

```
prevents the shutdown of the ResourceFactory.  
@roseuid 356C2593F700 */  
void shutdown ()  
    raises (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. This is called a federated file system.

A federated file system is created using the mount and unmount operations. Typically, the

Domain Manager or system initialization software will invoke these operations.

The FileManager inherits the IDL interface of a FileSystem. Based upon the pathname of a directory or

file and the set of mounted filesystems, the FileManager will delegate the FileSystem operations to the

appropriate FileSystem. For example, if a FileSystem is mounted at /ppc2, an open operation for a file

called /ppc2/profile.xml would be delegated to the mounted FileSystem. The mounted FileSystem will be

given the filename relative to it. In this example the FileSystem's open operation would receive

/profile.xml as the fileName argument.

Another example of this concept can be shown using the copy operation.

When a client invokes the copy operation, the FileManager will delegate operations to the

appropriate FileSystems (based upon supplied pathnames) thereby allowing copy of files between filesystems.

If a client does not need to mount and unmount FileSystems, it can treat the FileManager as a

FileSystem by CORBA widening a FileManager reference to a FileSystem reference. One can always widen

a FileManager to a FileSystem since the FileManager is derived from a FileSystem.

The FileManager's inherited FileSystem operations behavior implements the FileSystem operations

against the mounted file systems. The FileSystem operations ensure that the filename/directory arguments

given are absolute pathnames relative to a mounted FileSystem. The FileManager's FileSystem operations removes

the FileSystem mounted name from the input fileName before passing the fileName to an operation on a mounted FileSystem.

The FileManager uses the mounted FileSystem for FileSystem operations based upon the mounted FileSystem name

that exactly matches the input fileName to the lowest matching subdirectory.

The query operation returns the combined mounted file systems information to the calling client based upon

the given input fileSystemProperties' IDs. As a minimum, the query operation supports the following input

fileSystemProperties IDs:

1. SIZE - a property item ID value of "SIZE" will cause the query operation to return the combined total size

of all the mounted file system as an unsigned long long property value.

2. AVAILABLE_SPACE - a property item ID value of "AVAILABLE_SPACE" will cause the query operation to return

the combined total available space (in octets) of all the mounted file system as unsigned long long property value.

The query operation raises the UnknownFileSystemProperties exception when the input fileSystemProperties parameter contains an invalid property ID. */

```

interface FileManager : FileSystem {
    /* The Mount structure identifies a FileSystem mounted within a
FileManager. * */

    struct MountType {
        string mountPoint;
        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 file manager. */

    exception MountPointAlreadyExists {
    };

    /* The FileManager supports the notion of a federated file system.
To create a federated file system,
the mount operation associated a FileSystem with a mount point (a
directory name).

The mount operation associates the specified FileSystem with the
given mountPoint.
A mountPoint name begins with a "/". A mountPoint name is a
logical directory name for a FileSystem.

The mount operation raises the NonExistentMount exception when the
mountPoint (directory) name is not an
absolute pathname relative to the mounted file.

The mount operation raises the MountPointAlreadyExists exception
when the mountPoint already
exists in the file manager.
The InvalidFileSystem exception is raised when the input FileSystem
is a null object reference.
@roseuid 36FAA01001E0 */

```

```

        void mount (
            in string mountPoint,
            in FileSystem file_System
        )
            raises (InvalidFileName, InvalidFileSystem,
MountPointAlreadyExists);

        /* The unmount operation removes a mounted FileSystem from the
FileManager whose
        mounted name matches the input mountPoint name. The unmount
operation raises NonExistentMount
        when the mount point does not exist within the FileManager.
        @roseuid 36FAA07A010E */
        void unmount (
            in string mountPoint
        )
            raises (NonExistentMount);

        /* The getMounts operation returns the FileManager's mounted
FileSystems.-
        @roseuid 3895C9C3A0F0 */
        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.
    An application establishes the operations for transferring data and
control.
    The application also establishes the meaning of the data and control
values. Examples of how
    applications may use ports in different ways include: push or pull,
synchronous or asynchronous,
    mono- or bi-directional, or whether to use flow control (e.g., pause,
start, stop).

    The nature of Port fan-in, fan-out, or one-to-one is component dependent.

    Note 1: The CORBA specification defines only a minimum size for each
basic IDL type. The actual size of the
    data type is dependent on the language (defined in the language mappings)
as well as the Central Processing Unit
    (CPU) architecture used. By using these CORBA basic data types,
portability is maintained between components
    implemented in differing CPU architectures and languages.

    Note 2: How components' ports are connected is described in the software
assembly descriptor file
    of the Domain Profile. */

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

```

```

    · errorCode 1 means the Port component is invalid (unable to narrow
object reference)
    or illegal object reference,
    · errorCode 2 means the Port name is not found (not used by this
Port). */

```

```

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.

```

```

A port may support several connections. The input connectionID is
a unique identifier to be used by
disconnectPort when breaking this specific connection.

```

```

The connectPort operation raises the InvalidPort exception when the
input connection parameter is an
invalid connection for this Port.

```

```

The OccupiedPort exception is raised when the Port is fully
occupied and unable to accept connections.

```

```

@roseuid 38C1759DA718 */
void connectPort (
    in Object connection,
    in string connectionId
)
    raises (InvalidPort, OccupiedPort);

```

```

/* The disconnectPort operation breaks the connection to the
component identified by the input parameters.
The InvalidPort exception is raised when the name passed to the
operation is invalid.

```

```

@roseuid 38C175A5DC10 */
void disconnectPort (
    in string connectionId
)
    raises (InvalidPort);

```

```

};

```

```

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

```

```

interface LifeCycle {
    /* This exception indicates an error occurred during component
initialization. The messages provides

```

```

        additional information describing the reason why the error
occurred. */

        exception InitializeError {
            StringSequence errorMessages;
        };

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

        exception ReleaseError {
            StringSequence errorMessages;
        };

        /* The purpose of the initialize operation is to provide a
mechanism to set an object to an known
        initial state. (For example, data structures may be set to initial
values, memory may be allocated,
        hardware devices may be configured to some state, etc.).

        Initialization behavior is implementation dependent.

        This operation raises the InitializeError when an initialization
error occurs.
        @roseuid 37DD15FA01C2 */
        void initialize ()
            raises (InitializeError);

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

        The releaseObject operation releases all internal memory allocated
by the component during the
        life of the component. The releaseObject operation tears down the
component (i.e. released from
        the CORBA environment). The releaseObject operation releases
components from the Operating Environment.

        This operation raises a ReleaseError when a release error occurs.
        @roseuid 37DD15FA01C3 */
        void releaseObject ()
            raises (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 Test (BIT) to be implemented and this provides a means to isolate faults (both software and hardware) within the system.

The runTest operation uses the testid parameter to determine which of its predefined test implementations should be performed. The testValues parameter Properties (id/value pair(s)) are used to provide additional information to the implementation-specific test to be run. The runTest operation returns the result(s) of the test in the testValues parameter.

Tests to be implemented by a component are component-dependent and are specified in the component's Properties Descriptor. Valid testid(s) and both input and output testValues (properties) for the runTest operation, at a minimum, are test properties defined in the properties test element of the component's Properties Descriptor (refer to Appendix D Domain Profile). The testid parameter corresponds to the XML attribute testid of the property element test in a propertyfile.

Before an UnknownProperties exception is raised by the runTest operation all inputValues properties are validated (i.e., test properties defined in the propertyfile(s) referenced in the component's SPD).

The runTest operation does not execute any testing when the input testid or any of the the input testValues are not known by the component or are out of range.

This operation does not return a value.

The runTest operation raises the UnknownTest exception when there is no underlying test implementation that is associated with the input testid given.

The runTest operation raises CF UnknownProperties exception when the input parameter testValues contains any DataTypes that are not known by the component's test implementation or any values that are out of range for the requested test. The exception parameter invalidProperties contains the invalid inputValues properties id(s) that are not known by the component or the value(s) are out of range.

```
@roseuid 38A583C40208 */  
void runTest (  
    in unsigned long testid,  
    inout Properties testValues  
)
```

```
        raises (UnknownTest, 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;
        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 {
        Properties invalidProperties;
    };

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

    The configure operation shall assign values to the properties as
indicated in the configProperties argument.
    An component's SPD profile indicates the valid configuration
values. Valid properties for the
    configure operation are at a minimum the configure readwrite and
writeonly properties referenced
    in the component's SPD.

    The configure operation raises an InvalidConfiguration exception
when a configuration error
    occurs that prevents any property configuration on the component.

    This operation raises PartialConfiguration exception when some
configuration properties were
    successful and some configuration properties were not successful.
@roseuid 38A583FFC998 */
    void configure (
        in Properties configProperties
    )
        raises (InvalidConfiguration, PartialConfiguration);

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

If the configProperties are zero size then, the query operation returns all component properties.

If the configProperties are not zero size, then the query operation returns only those id/value pairs

specified in the configProperties. An component's SPD profile indicates the valid query types. Valid properties for the query operation are at a minimum the configure readwrite and readonly properties, and allocation properties that have an action value of "external" as referenced in the component's SPD.

This operation raises the CF UnknownProperties exception when one or more properties

being requested are not known by the component.

```
@roseuid 38A583FFC99A */
```

```
void query (
    inout Properties configProperties
)
    raises (UnknownProperties);
```

```
};
```

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

The DomainManager interface can be logically grouped into three categories:

Human Computer Interface (HCI), Registration, and Core Framework (CF) administration.

1. The HCI operations are used to configure the domain, get the domain's capabilities

(Devices, Services, and Applications), and initiate maintenance functions.

Host operations are performed by a client user interface capable of interfacing to the

Domain Manager.

2. The registration operations are used to register / unregister DeviceManagers,

DeviceManager's Devices, DeviceManager's Services, and Applications at startup or

during run-time for dynamic device, service, and application extraction and insertion.

3. The administration operations are used to access the interfaces of registered

DeviceManagers, FileManagers, and Loggers of the domain.

During component construction the DomainManager registers itself with the CORBA Naming Service. During Naming Service registration the DomainManager creates a "naming context" using "/DomainName" as its name.ID component and "" (Null string) as its name.kind component, then create a "name binding" to the "/DomainName" naming context using "/DomainManager" as its name.ID component, "" (Null string) as its name.kind

component, and the DomainManager's object reference. (See also 3.1.3.2.2.5.1.3)

Since a log service is not a required component of a JTRS installation, a DomainManager implementation may, or may not have access to a Log. However, if log service(s) are available, a DomainManager implementation may use one or more of them. The Logs utilized by the DomainManager implementation shall be defined in the DMD. See Appendix D for further description of the DMD file.

Once a service specified in the DMD is successfully registered with the DomainManager (via registerDeviceManager or registerService operations), the DomainManager begins to use the service (e.g., Log).

The DomainManager creates its own FileManager component that consists of all registered DeviceManager's FileSystems.

The DomainManager restores ApplicationFactories after startup for applications

that were previously installed by the DomainManager installApplication operation.

The DomainManager adds the restored ApplicationFactories to the DomainManager's applicationFactories attribute.

The DomainManager creates the Incoming Domain Management and Outgoing Domain Management event channels */

```
interface DomainManager : PropertySet {
    /* The ApplicationInstallationError exception type is raised when
an Application installation
    has not completed correctly. The error number indicates an
ErrorNumberType value (e.g., EINVAL,
    ENAMETOOLONG , ENOENT, ENOMEM, ENOSPC, ENOTDIR, ENXIO). The message
is component-dependent,
    providing additional information describing the reason for the
error. */

    exception ApplicationInstallationError {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* This type defines an unbounded sequence of Applications. */
    typedef sequence <Application> ApplicationSequence;

    /* This type defines an unbounded sequence of application
factories. */
    typedef sequence <ApplicationFactory> ApplicationFactorySequence;

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

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

```
    /* The DeviceManagerNotRegistered 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 {
    };

    /* The ApplicationUninstallationError exception type is raised when
an Application uninstallation
    has not completed correctly. The error number indicates an
ErrorNumberType value. The message is
    component-dependent, providing additional information describing
the reason for the error. */

    exception ApplicationUninstallationError {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* The RegisterError exception indicates that an internal error has
occurred to prevent DomainManager
    registration operations from successful completion. The error
number indicates an ErrorNumberType value.
    The message is component-dependent, providing additional
information describing the reason for the error. */

    exception RegisterError {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* The UnregisterError exception indicates that an internal error
has occurred to prevent
    DomainManager unregister operations from successful completion. The
error number indicates an
    ErrorNumberType value. The message is component-dependent,
providing additional information
    describing the reason for the error. */

    exception UnregisterError {
        ErrorNumberType errorNumber;
        string msg;
    };

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

    exception AlreadyConnected {
    };

    /* The InvalidEventChannelName 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 domainManagerProfile attribute contains the DomainManager's
profile.
The readonly domainManagerProfile attribute contains either a
profile element with a file reference
to the DomainManager's (DMD) profile or the XML for the
DomainManager's (DMD) profile. Files referenced within the profile
will have to be obtained from
the DomainManager's FileManager. */

readonly attribute string domainManagerProfile;
/* The deviceManagers attribute is read-only containing a sequence
of registered DeviceManagers in
the domain. The DomainManager contains a list of registered
DeviceManagers that have registered with
the DomainManager. The DomainManager writes an
ADMINISTRATIVE_EVENT log to a DomainManager's Log,
when the deviceManagers attribute is obtained by a client. */

readonly attribute DeviceManagerSequence deviceManagers;
/* The applications attribute is read-only containing a sequence of
instantiated Applications in
the domain. The DomainManager contains a list of Applications that
have been instantiated. The
DomainManager writes an ADMINISTRATIVE_EVENT log record to a
DomainManager's Log, when the
applications attribute is obtained by a client. */

readonly attribute ApplicationSequence applications;
/* The readonly applicationFactories attribute contains a list with
one ApplicationFactory per
application (SAD file and associated files) successfully installed
(i.e. no exception raised).
The DomainManager writes an ADMINISTRATIVE_EVENT log record to a
DomainManager's Log, when the
applicationFactories attribute is obtained by a client. */

readonly attribute ApplicationFactorySequence applicationFactories;
/* The fileMgr attribute is read only containing the mounted
FileSystems in the domain. The
DomainManager writes an ADMINISTRATIVE_EVENT log record to a
DomainManager's Log, when the
fileMgr attribute is obtained by a client. */

readonly attribute 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 verifies that the input parameters,
registeringDevice and registeredDeviceMgr,
are not nil CORBA component references.
```

```
    The registerDevice operation adds the registeringDevice and the
registeringDevice's attributes
    (e.g., identifier, softwareProfile's allocation properties, etc.)
to the
    DomainManager, if it does not already exist.
```

```
    The registerDevice operation associates the input registeringDevice
with the input registeredDeviceMgr
    in the DomainManager when the input registeredDeviceMgr is a
    valid registered DeviceManager in the DomainManager.
```

```
    When the registering Device's parent DeviceManager's DCD describes
service connections for the registering
    Device, the registerDevice operation shall establish the
connections.
```

```
    The registerDevice operation, upon successful device registration,
writes an ADMINISTRATIVE_EVENT log
    record to a DomainManager's Log, to indicate that the device has
successfully registered with the DomainManager.
```

```
    Upon unsuccessful device registration, the registerDevice operation
writes a FAILURE_ALARM log record to a
    DomainManager's Log, when the InvalidProfile exception is raised to
indicate that the
    registeringDevice has an invalid profile.
```

```
    Upon unsuccessful device registration, the registerDevice operation
logs a Failure_Alarm event with
    DomainManager's Logger for the DeviceManagerNotRegistered exception
to indicate that the device that
    cannot be registered to the Device due to the DeviceManager is not
registered with the DomainManager.
```

```
    Upon unsuccessful device registration, the registerDevice operation
writes a FAILURE_ALARM log record to a
    DomainManager's Log, indicating that the device could not register
because the DeviceManager is not
    registered with the DomainManager.
```

```
    Upon unsuccessful device registration, the registerDevice operation
writes a FAILURE_ALARM log record to a
```

DomainManager's Log, because of an invalid reference input parameter.

Upon unsuccessful device registration, the registerDevice operation shall write a FAILURE_ALARM log record to a DomainManager's Log, because of an internal registration error.

The registerDevice operation, upon successful Device registration, sends an event to the Outgoing

Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.

The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.
2. The sourceId is the identifier attribute of the registered Device.
3. The sourceName is the label attribute of the registered Device.
4. The sourceIOR is the registered Device object reference.
5. The sourceCategory is DEVICE.

The registerDevice operation raises the CF InvalidProfile exception when:

1. The Device's SPD file and the SPD's referenced files do not exist or cannot be processed due to the file not being compliant with XML syntax, or
2. The Device's SPD does not reference allocation properties.

The registerDevice operation raises a DeviceManagerNotRegistered exception when

the input registeredDeviceMgr (not nil reference) is not registered with the DomainManager.

The registerDevice operation raises the CF InvalidObjectReference exception when input parameters

registeringDevice or registeredDeviceMgr contains an invalid reference.

The registerDevice operation raises the RegisterError exception when an internal error exists which

causes an unsuccessful registration.

```
@roseuid 364B4CF92ED0 */
```

```
void registerDevice (
    in Device registeringDevice,
    in DeviceManager registeredDeviceMgr
)
```

```
    raises (InvalidObjectReference, InvalidProfile,
    DeviceManagerNotRegistered, RegisterError);
```

```
/* The registerDeviceManager operation verifies that the input
parameter, deviceMgr,
is a not a nil CORBA component reference.
```

The registerDeviceManager operation adds the input deviceMgr to the DomainManager's deviceManagers attribute, if it does not already exist. The registerDeviceManager operation adds the input deviceMgr's registeredDevices and each registeredDevice's attributes (e.g., identifier, softwareProfile's allocation properties, etc.) to the DomainManager. The registerDeviceManager operation associates the input deviceMgr's with the input deviceMgr's registeredDevices in the DomainManager in order to support the unregisterDeviceManager operation.

The registerDeviceManager operation adds the input deviceMgr's registeredServices and each registeredService's names to the DomainManager. The registerDeviceManager operation associates the input deviceMgr's with the input deviceMgr's registeredServices in the DomainManager in order to support the unregisterDeviceManager operation.

The registerDeviceManager operation performs the connections as specified in the connections element of the deviceMgr's Device Configuration Descriptor (DCD) file. If the DeviceManager's Device Configuration Descriptor (DCD) describes a connection for a service that has not been registered with the DomainManager, the registerDeviceManager operation establishes any pending connection when the service registers with the DomainManager by the registerDeviceManager operation.

For connections established for a CORBA Event Service's event channel, the registerDeviceManager operation connects a CosEventComm PushConsumer or PushSupplier object to the event channel as specified in the DCD's domainfinder element. If the event channel does not exist, the registerDeviceManager operation creates the event channel.

The registerDeviceManager operation obtains all the Software profiles from the registering DeviceManager's FileSystems.

The registerDeviceManager operation mounts the DeviceManager's FileSystem to the DomainManager's FileManager. The mounted FileSystem name will have the format, "/DomainName/HostName", where DomainName is the name of the domain and HostName is the input deviceMgr's label attribute.

The registerDeviceManager operation, upon unsuccessful DeviceManager registration, writes a FAILURE_ALARM log record to a DomainManager's Log.

The registerDeviceManager operation, upon successful DeviceManager registration,

sends an event to the Outgoing Domain Management event channel with event data

consisting of a DomainManagementObjectAddedEventType. The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.
2. The sourceId is the identifier attribute of the registered DeviceManager.
3. The sourceName is the label attribute of the registered DeviceManager.
4. The sourceIOR is the registered DeviceManager object reference.
5. The sourceCategory is DEVICE_MANAGER

The registerDeviceManager operation raises the CF InvalidObjectReference exception when the input parameter deviceMgr contains an invalid reference to a DeviceManager interface.

The registerDeviceManager operation raises the RegisterError exception when an internal error exists which causes an unsuccessful registration.

```
@roseuid 364B4D632938 */
void registerDeviceManager (
    in DeviceManager deviceMgr
)
    raises (InvalidObjectReference, InvalidProfile,
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.

The unregisterDeviceManager operation unregisters a DeviceManager component from the DomainManager.

The unregisterDeviceManager operation releases (client-side CORBA release) all device(s) and service(s) associated with the DeviceManager that is being unregistered.

The unregisterDeviceManager operation disconnects consumers and producers (e.g., Devices, Log, DeviceManager, etc.) from a CORBA Event Service event channel based upon the software profile.

The unregisterDeviceManager operation may destroy the CORBA Event Service event channel when no more consumers and producers are connected to it.

The unregisterDeviceManager operation shall unmount all DeviceManager's FileSystems from its File Manager.

The unregisterDeviceManager operation, upon the successful unregistration of a DeviceManager, writes an ADMINISTRATIVE_EVENT log record to a DomainManager's Log.

The unregisterDeviceManager operation, upon unsuccessful unregistration of a DeviceManager, writes a FAILURE_ALARM log record to a DomainManager's Log.

The unregisterDeviceManager operation, upon successful unregistration, sends an event to the Outgoing Domain Management event channel with event data

consisting of a DomainManagementObjectRemovedEventType. The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.
2. The sourceId is the identifier attribute of the unregistered DeviceManager.
3. The sourceName is the label attribute of the unregistered DeviceManager.
4. The sourceCategory is DEVICE_MANAGER.

The unregisterDeviceManager operation raises the CF InvalidObjectReference when the input parameter DeviceManager contains an invalid reference to a DeviceManager interface.

The unregisterDeviceManager operation raises the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

```
@roseuid 364B4EAB15E0 */
```

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

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

The unregisterDevice operation releases (client-side CORBA release) the unregisteringDevice from the DomainManager.

The unregisterDevice operation disconnects the Device's consumers and producers from a CORBA Event Service event channel based upon the software profile.

The unregisterDevice operation may destroy the CORBA Event Service event channel when no more consumers and producers are connected to it.

The unregisterDevice operation, upon the successful unregistration of a Device, writes an ADMINISTRATIVE_EVENT log record to a DomainManager's Log.

The unregisterDevice operation, upon unsuccessful unregistration of a Device,

writes a FAILURE_ALARM log record to a DomainManager's Log.

The unregisterDevice operation, upon successful Device unregistration, sends an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType. The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.
2. The sourceId is the identifier attribute of the unregistered Device.
3. The sourceName is the label attribute of the unregistered Device.
4. The sourceCategory is DEVICE.

The unregisterDevice operation raises the CF InvalidObjectReference exception

when the input parameter contains an invalid reference to a Device interface.

The unregisterDevice operation raises the UnregisterError exception when an internal error

exists which causes an unsuccessful unregistration.

```
@roseuid 364B4EC8DEC0 */
void unregisterDevice (
    in Device unregisteringDevice
)
    raises (InvalidObjectReference, UnregisterError);
```

/* This operation is used to register new application software in the DomainManager.

An installer application typically invokes this operation when it has completed the installation of a new Application into the domain.

The profileFileName is the absolute path of the profile filename.

The installApplication operation verifies the application's SAD file exists in the DomainManager's FileManager and all the files the application is dependent on are also resident.

The installApplication operation writes an ADMINISTRATIVE_EVENT log Record to a

DomainManager's Log, upon successful Application installation.

The installApplication operation, upon successful application installation, sends an event to the Outgoing Domain Management event channel with event data consisting of a

DomainManagementObjectAddedEventType. The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.

2. The sourceId is the identifier attribute of the installed ApplicationFactory.
3. The sourceName is the name attribute of the installed ApplicationFactory.
4. The sourceIOR is the installed ApplicationFactory object reference.
5. The sourceCategory is APPLICATION_FACTORY.

The installApplication operation raises the ApplicationInstallationError exception when the installation of the Application file(s) was not successfully completed.

The installApplication operation raises the CF InvalidFileName exception when the input SAD file or any referenced file name does not exist in the file system as defined in the absolute path of the input profileFileName. The installApplication operation logs a FAILURE_ALARM log record to a DomainManger's Log when the InvalidFileName exception occurs and the logged message shall be "installApplication:: invalid file is xxx", where "xxx" is the input or referenced file name is bad.

The installApplication operation raises the CF InvalidProfile exception when the input SAD file or any referenced file is not compliant with XML DTDs defined in Appendix D or referenced property definitions are missing. The installApplication operation logs a FAILURE_ALARM log record of a DomainManager's Log when the CF InvalidProfile exception occurs and the logged message shall be "installApplication:: invalid Profile is yyy," where "yyy" is and the input or referenced file name that is bad along with the element or position within the profile that is bad.

```
@roseuid 3896F0D83588 */
void installApplication (
    in string profileFileName
)
    raises (InvalidProfile, InvalidFileName,
ApplicationInstallationError);
```

/* This operation is used to uninstall an application from the DomainManager.

The CF Installer typically invokes this operation when removing an application from the radio domain.

The ApplicationID parameter is the softwareassembly element id attribute of the ApplicationFactory's Software Assembly Descriptor file.

The `uninstallApplication` operation removes all files associated with the Application.

The `uninstallApplication` operation makes the `ApplicationFactory` unavailable from the `DomainManager` (i.e. its services no longer provided for the Application).

The `uninstallApplication` operation, upon successful uninstall of an Application, writes an `ADMINISTRATIVE_EVENT` log record to a `DomainManager`'s Log.

The `uninstallApplication` operation, upon unsuccessful uninstall of an Application, writes a `FAILURE_ALARM` log record to a `DomainManager`'s Log.

The `uninstallApplication` operation, upon successful uninstall of an application, sends an event to the Outgoing Domain Management event channel with event data consisting of a `DomainManagementObjectRemovedEventType`. The event data will be populated as follows:

1. The `producerId` is the identifier attribute of the `DomainManager`.
2. The `sourceId` is the identifier attribute of the uninstalled `ApplicationFactory`.
3. The `sourceName` is the name attribute of the uninstalled `ApplicationFactory`.
4. The `sourceCategory` is `APPLICATION_FACTORY`.

The `uninstallApplication` operation raises the `InvalidIdentifier` exception when the `ApplicationId` is invalid.

The `uninstallApplication` operation raises the `ApplicationUninstallationError` exception when an internal error causes unsuccessful uninstall of the application.

```
@roseuid 3896F13747C8 */
void uninstallApplication (
    in string applicationId
)
    raises (InvalidIdentifier, ApplicationUninstallationError);
```

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

The `registerService` operation verifies the input `registeringService` and `registeredDeviceMgr` are valid object references.

The `registerService` operation verifies the input `registeredDeviceMgr` has been previously registered with the `DomainManager`.

The registerService operation adds the registeringService's object reference and the registeringService's name to the DomainManager, if the name for the type of service being registered does not exist within the DomainManager. However, if the name of the registering service is a duplicate of a registered service of the same type, then the new service is not registered with the DomainManager.

The registerService operation associates the input registeringService parameter with the input registeredDeviceMgr parameter in the DomainManager's, when the registeredDeviceMgr parameter indicates a DeviceManager registered with the DomainManager.

The registerService operation, upon successful service registration, establishes any pending connection requests for the registeringService. The registerService operation, upon successful service registration, writes an ADMINISTRATIVE_EVENT log record to a DomainManager's Log.

The registerService operation, upon unsuccessful service registration, writes a FAILURE_ALARM log record to a DomainManager's Log.

The registerService operation, upon successful service registration, sends an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType. The event data will be populated as follows:

1. The producerId is the identifier attribute of the DomainManager.
2. The sourceId is the identifier attribute from the componentinstantiation element associated with the registered service.
3. The sourceName is the input name parameter for the registering service.
4. The sourceIOR is the registered service object reference.
5. The sourceCategory is SERVICE.

This operation does not return a value.

The registerService operation raises a DeviceManagerNotRegistered exception when the input registeredDeviceMgr parameter is not a nil reference and is not registered with the DomainManager.

The registerService operation raises the CF InvalidObjectReference exception when input parameters registeringService or registeredDeviceMgr contains an invalid reference.

The registerService operation raises the RegisterError exception when an internal error exists which causes an unsuccessful registration.

```
@roseuid 3B33926D032F */
void registerService (
    in Object registeringService,
    in DeviceManager registeredDeviceMgr,
```

```
        in string name
        )
        raises (InvalidObjectReference, InvalidProfile,
DeviceManagerNotRegistered, RegisterError);

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

    The unregisterService operation removes the unregisteringService
entry specified by the input parameter
    from the DomainManager.

    The unregisterService operation releases (client-side CORBA
release) the
    unregisteringService from the DomainManager.

    The unregisterService operation, upon the successful unregistration
of a Service,
    writes an ADMINISTRATIVE_EVENT log record to a DomainManager's Log.

    The unregisterService operation, upon unsuccessful unregistration
of a Service, writes a
    FAILURE_ALARM log record to a DomainManager's Log.
    The unregisterService operation, upon successful service
unregistration, sends an event
    to the Outgoing Domain Management event channel with event data
consisting of a
    DomainManagementObjectRemovedEventType. The event data will be
populated as follows:
    1. The producerId is the identifier attribute of the
DomainManager.
    2. The sourceId is the ID attribute from the
componentinstantiation element associated with the unregistered service.
    3. The sourceName is the input name parameter for the
unregistering service.
    4. The sourceCategory is SERVICE.

    This operation does not return a value.

    The unregisterService operation raises the CF
InvalidObjectReference exception when the
    input parameter contains an invalid reference to a Service
interface.

    The unregisterService operation raises the UnregisterError
exception when an internal error
    exists which causes an unsuccessful unregistration.
    @roseuid 3B3392750114 */
    void unregisterService (
        in Object unregisteringService,
        in string name
    )
    raises (InvalidObjectReference, UnregisterError);

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

The registerWithEventChannel operation connects the input registeringConsumer to event channel as specified by the input eventChannelName.

This operation does not return a value.

The registerWithEventChannel operation raises the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a CosEventComm PushConsumer interface.

The registerWithEventChannel operation raises the InvalidEventChannelName exception when the input parameter contains an invalid event channel name (e.g., "ODM_Channel").

The registerWithEventChannel operation raises AlreadyConnected exception when the input parameter contains a connection to the event channel for the input registeringConsumerId parameter.

```
@roseuid 3BB49D7901AE */
void registerWithEventChannel (
    in Object registeringObject,
    in string registeringId,
    in string eventChannelName
)
    raises (InvalidObjectReference, InvalidEventChannelName,
AlreadyConnected);

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

The unregisterFromEventChannel operation shall disconnect a
registered consumer from
the event channel as identified by the input parameters.

This operation does not return a value.

The unregisterFromEventChannel operation raises the
InvalidEventChannelName exception
when the input parameter contains an invalid reference to an event
channel (e.g., "ODM_Channel").
The unregisterWithEventChannel operation raises NotConnected
exception when the input
parameter registeringConsumerId parameter is not connected to
specified input event channel.
@roseuid 3BB49DF903CF */
void unregisterFromEventChannel (
    in string unregisteringId,
    in string eventChannelName
)
    raises (InvalidEventChannelName, NotConnected);
};
```

```
/* The ApplicationFactory interface class provides an interface to
request the
creation of a specific type (e.g., SINGGARS, LOS, Havequick, etc.) of
Application in the domain. The ApplicationFactory interface class is
designed
using the Factory Design Pattern. 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 (1) or more
invalid Application component-to-device assignment(s). */

    exception CreateApplicationRequestError {
        DeviceAssignmentSequence invalidAssignments;
    };

    /* The CreateApplicationError exception is raised when a create
request is valid but the Application
is unsuccessfully instantiated due to internal processing errors.
The error number indicates an
ErrorNumberType value (e.g., E2BIG, ENAMETOOLONG, ENFILE, ENODEV,
ENOENT, ENOEXEC, ENOMEM, ENOTDIR,
ENXIO, EPERM). The message is component-dependent, providing
additional information describing the
reason for the error. */

    exception CreateApplicationError {
        ErrorNumberType errorNumber;
        string msg;
    };

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

    exception InvalidInitConfiguration {
        Properties invalidProperties;
    };

    /* The name attribute contains the name of the type of Application
that can be instantiated
by the ApplicationFactory (e.g., SINGGARS, LOS, Havequick, DAMA25,
etc.). */

    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
this factory uses when
```

creating an application. The string value contains either a profile element with a file reference to the SAD profile file or the actual xml for the SAD profile. Files referenced within the profile will have to be obtained from a FileManager. The ApplicationFactory will have to be queried for profile information for Component files that are referenced by an ID instead of file name. */

```
readonly attribute string softwareProfile;
```

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

The create operation provides a client interface to request the creation of an Application on client requested device(s) or the creation of an Application in which the ApplicationFactory determines the necessary device(s) required for instantiation of the Application.

An Application can be comprised of one or more components (e.g., Resources, Devices, etc.).

The SAD contains Software Package Descriptors (SPDs) for each Application component. The

SPD specifies the Device implementation criteria for loading dependencies (processor kind, etc.) and processing capacities (e.g., memory, process) for an application component. The create

operation uses the SAD SPD implementation element to locate candidate devices capable of loading and executing Application components.

If deviceAssignments (not zero length) are provided, the ApplicationFactory verifies each device assignment, for the specified component, against the component's SPD implementation element.

The create operation allocates (Device allocateCapacity) component capacity requirements against

candidate devices to determine which candidate devices satisfy all SPD implementation criteria

requirements and SAD partitioning requirements (e.g., components HostCollocation, etc.). The

create operation only uses Devices that have been granted successful capacity allocations for

loading and executing Application components, or used for data processing. The actual Devices

chosen will reflect changes in capacity based upon component capacity requirements allocated to them, which may also cause state changes for the Devices.

The create operation loads the Application components (including all of the

Application-dependent components) to the chosen device(s).

The create operation executes the application components (including all of the application-dependent

components) as specified in the application's Software Assembly Descriptor (SAD) file.

The create operation uses each component's SPD implementation code's stack size and priority elements,

when specified, for the execute options parameters.

The create operation passes the mandatory execute parameters of a Naming Context IOR,

Name Binding, and the identifier for the component in the form of CF Properties to the

entry points of Resource components to be executed via a Device's execute operation.

The execute parameter for the Naming Context IOR is inserted into a CF Properties type.

The CF Properties ID element is set to "NAMING_CONTEXT_IOR" and the CF Properties value

element set to the stringified IOR of a naming context to which the component will bind.

The create operation creates any naming contexts that do not exist to which the component

will bind to the Naming Context IOR. The structure of the naming context path is "/ DomainName

/ [optional naming context sequences]". In the naming context path, each "slash" (/) represents

a separate naming context.

The execute parameter of Name Binding is inserted into a CF Properties type. The CF Properties

ID element is set to "NAME_BINDING" and CF Properties value element set to a string in the format

of "ComponentName_UniqueIdentifier". The ComponentName value is the SAD componentinstantiation

findcomponent namingservice element's name attribute. The UniqueIdentifier is determined by the

implementation. The Name Binding parameter is used by the component to bind its object reference

to the Naming Context IOR parameter.

The create operation uses "ComponentName_UniqueIdentifier" to retrieve the component's object

reference from the Naming Context IOR (See also section 3.2.1.3.). Due to the dynamics of

bind and resolve to CORBA Naming Service, the create operation should provide sufficient

attempts to retrieve component object references from CORBA Naming Service prior to generating an exception.

For the component identifier execute parameter, the create operation is inserted in a CF Properties type.

The CF Properties ID element is set to "COMPONENT_IDENTIFIER" and the CF Properties value element to

the string format of

Component_Instantiation_Identifier:Application_Name. The

Component_Instantiation_Identifier

is created using the componentinstantiation element id attribute for the component in the application's

SAD file. The Application_Name field is identical to the create operation's input name parameter.

The `Application_Name` field provides a specific instance qualifier for executed Resource components.

The create operation passes the `componentinstantiation` element "execparam" properties that have values as parameters to execute operation. The create operation passes "execparam" parameters values as string values.

The create operation, in order, initializes Resources, then establishes connections for Resources, and finally configures the Resources.

The create operation will only configure the application's `assemblycontroller` component.

The create operation initializes an Application component provided the component implements the `LifeCycle` interface.

The create operation configures an application's `assemblycontroller` component provided

the `assemblycontroller` has `configure` `readwrite` or `writeonly` properties with values.

The create operation uses the union of the input `initConfiguration` properties of the create

`writeable` "configure" properties

that have values. The input `initConfiguration` parameter has precedence over the `assemblycontroller`'s

`writeable` "configure" property values. The create operation, when creating a component from a

`ResourceFactory`, passes the `componentinstantiation` `componentresourcefactoryref` element

"factoryparam" properties that have values as qualifiers parameters to the referenced

`ResourceFactory` component's `createResource` operation.

The create operation interconnects Application components' (Resources' or Devices')

ports in accordance with the SAD. The create operation obtains Ports in accordance

with the SAD via `Resource` `getPort` operation. The create operation uses the SAD `connectinterface`

element `id` attribute as the unique identifier for a specific connection when provided.

The create operation creates a connection ID when no SAD `connectinterface` element attribute

`id` is specified for a connection. The create operation obtains a Resource in accordance

with the SAD via the CORBA Naming Service or a `ResourceFactory`.. The `ResourceFactory` can be

obtained by using the CORBA Naming Service.

The create operation passes, with invocation of each `ResourceFactory` `createResource` operation,

the `ResourceFactory` configuration properties associated with that Resource as dictated by the SAD.

The dependencies to Log, FileManager, FileSystem, CORBA Event Service, and CORBA Naming Service will be specified as connections in the SAD using the domainfinder element. The create operation will establish these connections. For connections established for a CORBA Event Service's event channel, the create operation connects a CosEventComm PushConsumer or PushSupplier object to the event channel as specified in the SAD's domainfinder element. If the event channel does not exist, the create operation creates the event channel. If the Application is successfully created, the create operation shall return an Application component reference for the created Application. A sequence of created Application references can be obtained using the DomainManager's readonly applications attribute.

The create operation, upon successful Application creation, writes an ADMINISTRATIVE_EVENT log record.

The create operation, upon unsuccessful Application creation, writes a FAILURE_ALARM log record.

The dependencies to Log, FileManager, and FileSystem will appear as connections in the SAD using the domainfinder element. The create operation will establish these connections.

The create operation, upon successful Application creation, sends an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.

The event data will be populated as follows:

1. The producerId is the identifier attribute of the ApplicationFactory.
2. The sourceId is the identifier attribute of the created Application.
3. The sourceName is the name attribute of the created Application.
4. The sourceIOR is the Application component reference for the created Application.
5. The sourceCategory is APPLICATION.

The create operation raises the CreateApplicationRequestError exception when the parameter CF DeviceAssignmentSequence contains one (1) or more invalid Application component to device assignment(s).

The create operation raises the CreateApplicationError exception when the create request is valid but the Application can not be successfully instantiated due to internal processing error(s).

The create operation raises the InvalidInitConfiguration exception when the input initConfiguration parameter is invalid. The InvalidInitConfiguration invalidProperties1

```

        identifies the property that is invalid.
        @roseuid 38B7D97BCF98 */
        Application create (
            in string name,
            in Properties initConfiguration,
            in DeviceAssignmentSequence deviceAssignments
        )
        raises (CreateApplicationError,
CreateApplicationRequestError, InvalidInitConfiguration);

};

/* This 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. The exact number
is specified in the component's Software Profile SCD
(section Error! Reference source not found.). These Ports can be
either push or pull types. Multiple
input and/or output ports provide flexibility for PortSuppliers
that must
manage varying priority levels and categories of incoming and
outgoing messages, provide multi-threaded
message handling, or other special message processing.

The getPort operations returns the object reference to the named
port as stated in the
component's SCD. The getPort operation returns the CORBA object
reference that is associated
with the input port name.

The getPort operation raises an UnknownPort exception if the port
name is invalid.
@roseuid 3B336BB80213 */
    Object getPort (
        in string name
    )
    raises (UnknownPort);

};

/* The Resource interface provides a common API for the control and
configuration
of a software component.
The Resource interface inherits from the LifeCycle, PropertySet,
TestableObject, and
PortSupplier interfaces.

```

```

The inherited Lifecycle, PropertySet, TestableObject, and PortSupplier
interface
    operations are documented in their respective sections of this document.
The Resource interface may also be inherited by other application
interfaces
    as described in the Software Profile's Software Component Descriptor
(SCD) file. */

interface Resource : Lifecycle, TestableObject, PropertySet, PortSupplier
{
    /* The StartError exception indicates that an error occurred during
an attempt to start the Resource.
    The error number indicates an ErrorNumberType value (e.g., EDOM,
EPERM, ERANGE). The message is
    component-dependent, providing additional information describing
the reason for the error. */

    exception StartError {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* The StopError exception indicates that an error occurred during
an attempt to stop the Resource.
    The error number indicates an ErrorNumberType value (e.g.,
ECANCELED, EFAULT, EINPROGRESS).
    The message is component-dependent, providing additional
information describing the reason for the error. */

    exception StopError {
        ErrorNumberType errorNumber;
        string msg;
    };

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

    readonly attribute string identifier;

    /* The start operation puts the Resource in an operating condition.

    The start operation raises the StartError exception if an error
occurs while starting the resource.
    @roseuid 38BEE2457548 */
    void start ()
        raises (StartError);

    /* The stop operation disables all current operations and put the
Resource in a non-operating condition.
    Subsequent configure, query, and start operations are not inhibited
by the stop operation.

    The stop operation raises the StopError exception if an error
occurs while stopping the resource.
    @roseuid 38BEE2457549 */
    void stop ()

```

```
        raises (StopError);

};

/* A Device is a type of Resource within the domain and has the
requirements as stated in the Resource interface.
This interface defines additional capabilities and attributes for
any logical Device in the domain. A logical Device is a functional
abstraction for a set (e.g., zero or more) of
hardware devices and provides the following attributes and
operations:
1. Software Profile Attribute - This SPD XML profile defines the logical
Device capabilities (data/command uses
and provides ports, configure and query properties, capacity
properties, status properties, etc.), which could be a subset of the
hardware device's capabilities.
2. State Management Attributes - The usage, operational, and
administrative states constitutes the overall state
for a logical Device. Status properties may contain more
detailed information about aspects of the states.
3. Capacity Operations - In order to use a device, certain capacities
(e.g., memory, performance, etc.) must be
obtained from the Device. The capacity properties will vary
among devices and are described in the Software Profile. A device may
have multiple allocatable capacities, each
having its own unique capacity model.

The following behavior is in addition to the Lifecycle releaseObject
operation behavior.
The releaseObject operation calls the releaseObject operation on all of
the Device's aggregated Devices (i.e.,
those Devices that are contained within the AggregateDevice's devices
attribute).

The releaseObject operation transitions the Device's adminState to
SHUTTING_DOWN state, when the Device's adminState is UNLOCKED.

The releaseObject operation causes the Device to be unavailable
(i.e., released from the CORBA environment, and its logical Device's
process
terminated on the OS when applicable), when the Device's adminState
transitions to LOCKED, meaning its aggregated Devices have been removed
and the Device's usageState is IDLE.

The releaseObject operation causes the removal of its Device from the
Device's compositeDevice.

The releaseObject operation unregisters its Device from its
DeviceManager.
The releaseObject operation does not return a value.

The releaseObject operation shall raise the ReleaseError exception when
the releaseObject operation is not successful in releasing a logical
Device
due to internal processing errors that occurred within the Device being
released. */
```

```

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. An example of such behavior is
allocateCapacity.
exception InvalidState {string msg;}; */

    exception InvalidState {
        string msg;
    };

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

    exception InvalidCapacity {
        /* The message indicates the reason for the invalid capacity.
*/
        string msg;
        /* The invalid capacities sent to the allocateCapacity
operation. */
        Properties capacities;
    };

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

    enum OperationalType {
        ENABLED,
        DISABLED
    };

    /* This is a CORBA IDL enumeration type that defines the Device's
usage states. The
usage state indicates which of the following states a Device is in:
IDLE - not in use
ACTIVE - in use, with capacity remaining for allocation or
BUSY - in use, with no capacity remaining for allocation */

    enum UsageType {
        IDLE,
        ACTIVE,

```

```

        BUSY
    };

    /* The readonly usageState attribute contains the Device's usage
state (IDLE, ACTIVE, or BUSY).
    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.

    Whenever the usageState attribute changes, the Device sends an
event to the Incoming Domain
    Management event channel with event data consisting of a
StateChangeEvent. The event data
    will be populated as follows:
    1. The producerId field is the identifier attribute of the
Device.
    2. The sourceId field is the identifier attribute of the Device.
    3. The stateChangeCategory field is USAGE_STATE_EVENT.
    4. The stateChangeFrom and stateChangeTo fields reflects the
usageState attribute value before
    and after the state change, respectively. */

    readonly attribute 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.
The adminState attribute only
    allows the setting of LOCKED and UNLOCKED values, where setting
"LOCKED" is only effective when
    the adminState attribute value is UNLOCKED, and setting "UNLOCKED"
is only effective when the
    adminState attribute value is LOCKED or SHUTTING_DOWN. Illegal
state transitions commands are ignored.
    The adminState attribute, upon being commanded to be LOCKED,
transitions from the UNLOCKED to the
    SHUTTING_DOWN state and set the adminState to LOCKED for its entire
aggregation of Devices
    (if it has any). The adminState then transitions to the LOCKED
state when the Device's
    usageState is IDLE and its entire aggregation of Devices are
LOCKED.

    Whenever the adminState attribute changes, the Device shall sends
an event to the Incoming
    Domain Management event channel with event data consisting of a
StateChangeEvent. The
    event data will be populated as follows:
    1. The producerId field is the identifier attribute of the
Device.
    2. The sourceId field is the identifier attribute of the Device.
    3. The stateChangeCategory field is ADMINISTRATIVE_STATE_EVENT.
    4. The stateChangeFrom and stateChangeTo fields reflects the
adminState attribute value before
    and after the state change, respectively. */

    attribute AdminType adminState;

```

```

    /* The readonly operationalState attribute contains the device's
operational state
    (ENABLED or DISABLED). The operational state indicates whether or
not the device is
    functioning.

```

```

    Whenever the operationalState attribute changes, the Device sends
an event to the
    Incoming Domain Management event channel with event data consisting
of a StateChangeEvent type.

```

```

    The event data will be populated as follows:

```

```

    1. The producerId field is the identifier attribute of the
Device.
    2. The sourceId field is the identifier attribute of the Device.
    3. The stateChangeCategory field is OPERATIONAL_STATE_EVENT.
    4. The stateChangeFrom and stateChangeTo fields reflects the
operationalState attribute value
    before and after the state change, respectively. */

```

```

    readonly attribute OperationalType operationalState;
    /* The softwareProfile attribute is the XML description for this
logical Device. The readonly
    softwareProfile attribute contains either a profile DTD element
with a file reference to the
    SPD profile file or the XML for the SPD profile. Files referenced
within the softwareProfile
    are obtained via the FileManager. */

```

```

    readonly attribute string softwareProfile;
    /* The readonly label attribute contains the Device's label. The
label attribute is the
    meaningful name given to a Device. The attribute could convey
location information
    within the system (e.g., audiol, serial1, etc.). */

```

```

    readonly attribute string label;
    /* The readonly 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 AggregateDevice compositeDevice;

```

```

    /* This operation provides the mechanism to request and allocate
capacity from the Device.

```

```

    The allocateCapacity operation reduces the current capacities of
the Device based upon the input capacities
    parameter, when the Device's adminState is UNLOCKED, Device's
operationalState is ENABLED, and
    Device's usageState is not BUSY.

```

```

    The allocateCapacity operation sets the Device's usageState
attribute to BUSY, when the
    Device determines that it is not possible to allocate any further
capacity. The

```

allocateCapacity operation sets the usageState attribute to ACTIVE, when capacity is being used and any capacity is still available for allocation.

The allocateCapacity operation returns "True", if the capacities has been allocated, or "False", if not allocated

The allocateCapacity operation raises the InvalidCapacity exception, when the capacities are invalid or the capacity values are the wrong type or ID.

The allocateCapacity operation raises the InvalidState exception, when the Device's adminState is not UNLOCKED or operationalState is DISABLED.

```
@roseuid 38B7EFD077B0 */
boolean allocateCapacity (
    in Properties capacities
)
    raises (InvalidCapacity, InvalidState);
```

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

The deallocateCapacity operation adjusts the capacities of the Device based upon the input capacities parameter.

The deallocateCapacity operation sets the usageState attribute to ACTIVE when, after adjusting capacities, any of the Device's capacities are still being used.

The deallocateCapacity operation sets the usageState attribute to IDLE when, after adjusting capacities, none of the Device's capacities are still being used.

The deallocateCapacity operation sets the adminState attribute to LOCKED as specified in adminState attribute.

This operation does not return any value.

The deallocateCapacity operation raises the InvalidCapacity exception, when the capacity ID is invalid or the capacity value is the wrong type. The InvalidCapacity exception will state the reason for the exception.

The deallocateCapacity operation raises the InvalidState exception, when the Device's adminState is LOCKED or operationalState is DISABLED.

```
@roseuid 38B7EFFD48 */
void deallocateCapacity (
    in Properties capacities
)
    raises (InvalidCapacity, InvalidState);
```

```
};

/* The Application delegates the implementation of the inherited Resource
operations (runTest, start, stop, configure, and query) to the
Application's
Resource component (Assembly Controller) identified by the Application's
SAD
assemblycontroller element.. The Application propagates exceptions
raised
by the Application's Assembly Controller's operations. The initialize
operation is
not propagated to the Application's components or its Assembly
Controller.
The initialize operation causes no action within an Application.

The releaseObject operation terminates execution of the Application,
returns all allocated computing resources, and de-allocates the
Resources'
capabilities in use by the devices associated with Application. Before
terminating,
the Application removes the message connectivity with its associated
Applications (e.g. Ports, Resources, and Logs) in the domain.

For each Application component not created by a ResourceFactory,
the releaseObject operation releases the component by utilizing the
Resources's releaseObject operation. If the component was created by a
ResourceFactory, the releaseObject operation releases the component
by the ResourceFactory releaseResource operation. The releaseObject
operation
shut downs a ResourceFactory when no more Resources are managed by the
ResourceFactory.

For each allocated device capable of operation execution, the
releaseObject
operation terminates all processes / tasks of the Application's
components
utilizing the Device's terminate operation.

For each allocated device capable of memory function, the releaseObject
operation
de-allocates the memory associated with Application's component instances
utilizing the Device's unload operation.

The releaseObject operation deallocates the Devices that are associated
with the Application being released, based on the Application's Software
Profile.
The actual devices deallocated (Device::deallocateCapacity) will reflect
changes
in capacity based upon component capacity requirements deallocated from
them,
which may also cause state changes for the Devices.

The Application releases all client component references to the
Application
components.
```

The `releaseObject` operation disconnects Ports from other Ports that have been connected based upon the software profile.

The `releaseObject` operation disconnects consumers and producers from a CORBA Event Service's event channel based upon the software profile.

The `releaseObject` operation may destroy a CORBA Event Service's event channel

when no more consumers and producers are connected to it.

For components (e.g., Resource, ResourceFactory) that are registered with Naming Service, the `releaseObject` operation unbinds those components and destroy the associated naming contexts as necessary from the Naming Service.

The `releaseObject` operation for an application disconnects Ports first, then

release the Resources and ResourceFactories, then call the terminate operation,

and lastly call the unload operation on the devices.

The `releaseObject` operation, upon successful Application release, writes an

ADMINISTRATIVE_EVENT log record.

The `releaseObject` operation, upon unsuccessful Application release, writes a

FAILURE_ALARM log record.

The `releaseObject` operation, upon successful Application release, sends an

event to the Outgoing Domain Management event channel with event data consisting of a `DomainManagementObjectRemovedEventType`. The event data will be populated as follows:

1. The `producerId` is the identifier attribute of the released Application.
2. The `sourceId` is the identifier attribute of the released Application.
3. The `sourceName` is the name attribute of the released Application.
4. The `sourceCategory` is APPLICATION */

```
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 {
        /* The componentId is a ID of a component that corresponds to
        the application's SAD
        componentinstantiation's ID attribute value. */
        string componentId;
        /* The process ID of the executable component. */
        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 (e.g.,
naming context, implementation ID). */

struct ComponentElementType {
/* The componentId is a ID of a component that corresponds to
the application's SAD
componentinstantiation's ID attribute value. */
string componentId;
/* The element ID that is associated with component ID. */
string elementId;
};

/* The componentElementSequence defines an unbounded sequence of
ComponentElementTypes. */

typedef sequence <ComponentElementType> ComponentElementSequence;

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

readonly attribute ComponentElementSequence
componentNamingContexts;
/* The componentProcessIds attribute contains the list of
components' process IDs within the
Application for components that are executing on a device. */

readonly attribute 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 DeviceAssignmentSequence componentDevices;
/* The componentImplementations attribute contains the list of
components' SPD implementation
IDs within the Application for those components created. */

readonly attribute ComponentElementSequence
componentImplementations;
/* This attribute is the XML profile information for the
application. The string value contains
either a profile element with a file reference to the SAD profile
file or the actual xml for the
SAD profile. Files referenced within a profile will have to be
obtained via a FileManager.
The Application will have to be queried for profile information for
Component files that are
referenced by an ID instead of a file name. */

```

```

        readonly attribute string profile;
        /* This name attribute contains the name of the created
Application. The ApplicationFactory
        interface'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, which is defined in Appendix D SPD */

    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 the Load operation failed
due to device dependent reasons.
    The LoadFail exception indicates that an error occurred during an
attempt to load the device. The
    error number indicates an ErrorNumberType value (e.g. EACCES,
EAGAIN, EBADF, EINVAL, EMFILE,
    ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR ). The message is
component-dependent, providing
    additional information describing the reason for the error. */

    exception LoadFail {
        ErrorNumberType errorNumber;
        string msg;
    };

    /* This 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.

    The load operation loads a file on the specified device based upon
the input loadKind and fileName

```

parameters using the input `FileSystem` parameter to retrieve the file.

The load operation supports the load types as stated in the Device's software profile `LoadType` allocation properties.

The load operation keeps track of the number of times a file has been successfully loaded.

This operation does not return any value.

The load operation raises the `InvalidState` exception when the Device's `adminState` is not `UNLOCKED` or `operationalState` is `DISABLED`.

The load operation raises the `InvalidLoadKind` exception when the input `loadKind` parameter is not supported.

The load operation raises the CF `InvalidFileName` exception when the file designated by the input `filename` parameter cannot be found.

The load operation raises the `LoadFail` exception when an attempt to load the device is unsuccessful.

```
@roseuid 3A5DAED301AE */
void load (
    in FileSystem fs,
    in string fileName,
    in LoadType loadKind
)
    raises (InvalidState, InvalidLoadKind, InvalidFileName,
LoadFail );
```

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

The unload operation decrements the load count for the input `filename` parameter by one.

The unload operation unloads the application software on the device based on the input `fileName` parameter, when the file's load count equals zero.

This operation does not return a value.

The unload operation raises the `InvalidState` exception when the Device's `adminState` is `LOCKED` or its `operationalState` is `DISABLED`.

The unload operation raises the CF `InvalidFileName` exception when the file designated by the input `filename` parameter cannot be found.

```
@roseuid 3A5DAED301B2 */
void unload (
    in string fileName
```

```

        )
        raises (InvalidState, 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 error number indicates an
ErrorNumberType value (e.g., ESRCH, EPERM,
EINVAL). The message is component-dependent, providing additional
information describing
the reason for the error. */

    exception InvalidProcess {
        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. Process
number is unique to the
Processor operating system that created the process. */

    typedef unsigned long ProcessID_Type;

    /* The InvalidParameters exception indicates the input parameters
are invalid on the execute
operation. This exception is raised when there are invalid execute
parameters. 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 {
        /* The invalidParms is a list of invalid parameters specified
in the execute or execute operation.
Each parameter's ID and value must be a string type. */
        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 {
            /* The invalidParms is a list of invalid parameters specified
in the execute or executeProcess
            operation. Each parameter's ID and value must be a string
type. */
            Properties invalidOpts;
        };

        /* The STACK_SIZE_ID is the identifier for the ExecutableDevice's
execute options parameter.
        The value for a stack size is an unsigned long. */

        const string STACK_SIZE = "STACK_SIZE";
        /* The PRIORITY_ID is the identifier for the ExecutableDevice's
execute options parameters.
        The value for a priority is an unsigned long. */

        const string PRIORITY_ID = "PRIORITY";
        /* The ExecuteFail exception indicates that the Execute operation
failed due to device dependent reasons.
        The ExecuteFail exception indicates that an error occurred during
an attempt to invoke the execute
        function on the device. The error number indicates an
ErrorNumberType value (e.g. EACCES, EBADF,
        EINVAL, EIO, EMFILE, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR). The
message is component-dependent,
        providing additional information describing the reason for the
error. */

        exception ExecuteFail {
            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.

        The terminate operation terminates the execution of the
process/thread designated by the processId input
        parameter on the Device.

        This operation does not return a value.

        The terminate operation raises the InvalidState exception when the
Device's adminState is LOCKED or
        operationalState is DISABLED.

        The terminate operation raises the InvalidProcess exception when
the processId
        does not exist for the Device.
        @roseuid 3A5DAEC1016D */
        void terminate (
            in ProcessID_Type processId
        )

```

```
raises (InvalidProcess, InvalidState);
```

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

```
The execute operation executes the function or file identified by  
the input name parameter  
using the input parameters and options parameters. Whether the  
input name parameter is a  
function or a file name is device-implementation-specific. The  
execute operation converts  
the input parameters (id/value string pairs) parameter to the  
standard argv of the POSIX exec  
family of functions, where argv(0) is the function name. The  
execute operation maps the  
input parameters parameter to argv starting at index 1 as follows,  
argv (1) maps to input  
parameters (0) id and argv (2) maps to input parameters (0) value  
and so forth. The execute operation passes argv  
through the operating system "execute" function.
```

```
The execute operation input options parameters are STACK_SIZE_ID  
and PRIORITY_ID.
```

```
The execute operation uses these options, when specified, to set  
the operating system's  
process/thread stack size and priority, for the executable image of  
the given input name parameter.
```

```
The execute operation returns a unique process ID for the process  
that it created or a  
process ID of minus 1 (-1) when a process is not created.
```

```
The execute operation raises the InvalidState exception when the  
Device's adminState is not  
UNLOCKED or operationalState is DISABLED.
```

```
The execute operation raises the InvalidFunction exception when the  
function indicated by the  
input name parameter does not exist for the Device.
```

```
The execute operation raises the CF InvalidFileName exception when  
the file name indicated by  
the input name parameter does not exist for the Device.
```

```
The execute operation raises the InvalidParameters exception when  
the input parameters parameter  
item ID and value are not string types.
```

```
The execute operation raises the InvalidOptions exception when the  
input options parameter does  
not comply with STACK_SIZE_ID and PRIORITY_ID options.
```

```
The execute operation raises the ExecuteFail exception when the  
operating system "execute"  
function for the device is not successful.
```

```

@roseuid 3A5DAEC1016F */
ProcessID_Type execute (
    in string name,
    in Properties options,
    in Properties parameters
)
    raises (InvalidState, InvalidFunction, InvalidParameters,
InvalidOptions, InvalidFileName, ExecuteFail);

};

/* The DeviceManager interface is used to manage a set of logical Devices
and
services. The interface for a DeviceManager is based upon its
attributes, which are:

1. Device Configuration Profile - a mapping of physical device locations
to meaningful
labels (e.g., audiol, serial1, etc.), along with the Devices and services
to be
deployed.
2. File System - the FileSystem associated with this DeviceManager.
3. Device Manager Identifier - the instance-unique identifier for this
DeviceManager.
4. Device Manager Label - the meaningful name given to this
DeviceManager.
5. Registered Devices - a list of Devices that have registered with this
DeviceManager.
6. Registered Services - a list of Services that have registered with
this DeviceManager.

The DeviceManager upon start up registers itself with a DomainManager.
This requirement allows the system to be developed where at a minimum
only the
DomainManager's component reference needs to be known. A DeviceManager
uses the DeviceManager's deviceConfigurationProfile attribute for
determining:
1. Services to be deployed for this DeviceManager (for example, log(s)),
2. Devices to be created for this DeviceManager (when the DCD
deployondevice
element is not specified then the DCD componentinstantiation element is
deployed
on the same hardware device as the DeviceManager),
3. Devices to be deployed on (executing on) another Device,
4. Devices to be aggregated to another Device,
5. Mount point names for FileSystems,
6. The DCD's id attribute for the DeviceManager's identifier attribute
value, and
7. The DCD's name attribute for the DeviceManager's label attribute
value.

The DeviceManager creates FileSystem components implementing the
FileSystem
interface for each OS file system. If multiple FileSystems are to be
created, the

```

DeviceManager mounts created FileSystems to a FileManager component (widened to a FileSystem through the FileSys attribute). Each mounted FileSystem name must be unique within the DeviceManager.

The DeviceManager supplies execute operation parameters (IDs and format values)

for a Device consisting of:

- a. DeviceManager IOR - The ID is "DEVICE_MGR_IOR" and the value is a string that is the DeviceManager stringified IOR.
- b. Profile Name - The ID is "PROFILE_NAME" and the value is a CORBA string that is the full mounted file system file path name.
- c. Device Identifier - The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD componentinstantiation id attribute.
- d. Device Label - The ID is "DEVICE_LABEL" and the value is a string that corresponds to the DCD componentinstantiation usage element. This parameter is only used when the DCD componentinstantiation usagaelement is specified.
- e. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an AggregateDevice stringified IOR. This parameter is only used when the DCD componentinstantiation element is a composite part of another componentinstantiation element.
- f. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element. The DeviceManager shall pass the componentinstantiation element "execparam" properties that have values as parameters.

The DeviceManager passes "execparam" parameters' IDs and values as string values. The DeviceManager uses the componentinstantiation element's SPD implementation code's stacksize and priority elements, when specified, for the execute options parameters.

The DeviceManager initializes and configures logical Devices that are started by the DeviceManager after they have registered with the DeviceManager. The DeviceManager configures a DCD's componentinstantiation element provided the componentinstantiation element has "configure" readwrite or writeonly properties with values.

If a Service is deployed by the DeviceManager, the DeviceManager supplies execute operation parameters (IDs and format values) consisting of:

- a. DeviceManager IOR - The ID is "DEVICE_MGR_IOR" and the value is a string that is the DeviceManager stringified IOR.

b. Service Name - The ID is "SERVICE_NAME" and the value is a string that corresponds to the DCD componentinstantiation usagename element. */

```

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 readonly deviceConfigurationProfile attribute contains the
DeviceManager's profile.
The readonly deviceConfigurationProfile attribute contains either a
profile element with a
file reference to the DeviceManager's Device Configuration
Descriptor (DCD) profile or the XML for
the DeviceManager's DCD profile. Files referenced within the
profile are obtained from a FileSystem. */

    readonly attribute string deviceConfigurationProfile;
    /* The readonly 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 FileSystem fileSys;
    /* The readonly identifier attribute shall contain 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 readonly label attribute contains the DeviceManager's label.
The label attribute is the
meaningful name given to a DeviceManager. */

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

    readonly attribute DeviceSequence registeredDevices;
    /* The readonly registeredServices attribute shall contain a list
of Services that have registered
with this DeviceManager or a sequence length of zero if no Services
have registered with the DeviceManager. */

```

```
readonly attribute ServiceSequence registeredServices;
```

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

```
The registerDevice operation adds the input registeringDevice to  
the DeviceManager's registeredDevices  
attribute when the input registeringDevice does not already exist  
in the registeredDevices attribute. The  
registeringDevice is ignored when duplicated.
```

```
The registerDevice operation registers the registeringDevice with  
the DomainManager when the  
DeviceManager has already registered to the DomainManager and the  
registeringDevice has been  
successfully added to the DeviceManager's registeredDevices  
attribute.
```

```
The registerDevice operation writes a log record with the log level  
set to FAILURE_ALARM, upon  
unsuccessful registration of a Device to the DeviceManager's  
registeredDevices.
```

```
This operation does not return any value.
```

```
The registerDevice operation raises the CF InvalidObjectReference  
when the input registeredDevice  
is a nil CORBA object reference.  
@roseuid 3ACFA4F90122 */  
void registerDevice (  
    in Device registeringDevice  
)  
    raises (InvalidObjectReference);
```

```
/* This operation unregisters a Device from a DeviceManager.  
The unregisterDevice operation removes the input registeredDevice  
from the DeviceManager's  
registeredDevices attribute. The unregisterDevice operation  
unregisters the input registeredDevice  
from the DomainManager when the input registeredDevice is  
registered with the DeviceManager and the  
DeviceManager is not shutting down.
```

```
The unregisterDevice operation writes a log record with the log  
level set to FAILURE_ALARM,  
when it cannot successfully remove a registeredDevice from the  
DeviceManager's registeredDevices.
```

```
This operation does not return any value.
```

```
The unregisterDevice operation raises the CF InvalidObjectReference  
when the input  
registeredDevice is a nil CORBA object reference or does not exist  
in the DeviceManager's
```

```
registeredDevices attribute.  
@roseuid 3ACFA5040000 */  
void unregisterDevice (  
    in Device registeredDevice  
    )  
    raises (InvalidObjectReference);
```

/* This operation provides the mechanism to terminate a DeviceManager.

The shutdown operation unregisters the DeviceManager from the DomainManager.

The shutdown operation performs releaseObject on all of the DeviceManager's registered Devices (DeviceManager's registeredDevices attribute).

The shutdown operation causes the DeviceManager to be unavailable (i.e. released from the CORBA environment and its process terminated on the OS), when all of the DeviceManager's registered Devices are unregistered from the DeviceManager.

This operation does not return any value.

This operation does not raise any exceptions.

```
@roseuid 3ACFA50E0302 */  
void shutdown ();
```

/* This operation provides the mechanism to register a Service with a DeviceManager.

The registerService operation adds the input registeringService to the DeviceManager's registeredServices attribute when the input registeringService does not already exist in the registeredServices attribute. The registeringService is ignored when duplicated.

The registerService operation registers the registeringService with the DomainManager when the DeviceManager has already registered to the DomainManager and the registeringService has been successfully added to the DeviceManager's registeredServices attribute.

The registerService operation writes a log record with the log level set to FAILURE_ALARM, upon unsuccessful registration of a Service to the DeviceManager's registeredServices.

This operation does not return any value.

The registerService operation raises the CF InvalidObjectReference exception when the input

```
registeredService is a nil CORBA object reference.  
@roseuid 3B338F910156 */  
void registerService (  
    in Object registeringService,  
    in string name  
)  
    raises (InvalidObjectReference);
```

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

The unregisterService operation removes the input registeredService from the

DeviceManager's registeredServices attribute. The unregisterService operation unregisters the input registeredService from the DomainManager when the input registeredService is registered with the DeviceManager and the DeviceManager is not in the shutting down state.

The unregisterService operation writes a log record with the log level set to FAILURE_ALARM, when it cannot successfully remove a registeredService from the DeviceManager's registeredServices.

This operation does not return any value.

The unregisterService operation raises the CF InvalidObjectReference when the input registeredService is a nil CORBA object reference or does not exist in the

```
DeviceManager's registeredServices attribute.  
@roseuid 3B338F950007 */  
void unregisterService (  
    in Object registeredService,  
    in string name  
)  
    raises (InvalidObjectReference);
```

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

The getComponentImplementationId operation will return the SPD implementation element's ID attribute that matches the ID attribute of the SPD implementation element used to create the component specified by the input componentInstantiation ID parameter.

The getComponentImplementationId operation returns the SPD implementation element's ID attribute that matches the SPD implementation element used to create the component identified by the input componentInstantiationID parameter. The getComponentImplementationId

operation returns an empty string when the input componentInstantiationId parameter does not match the ID attribute of any SPD implementation element used to create the component.

This operation does not raise any exceptions.

```
@roseuid 3B45B6E301FB */
```

```
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. The Basic Sequence Types IDL was generated from the Rational Rose model, version 98i.

//Source file: PortTypes.idl

```
#ifndef __PORTTYPES_DEFINED
```

```
#define __PORTTYPES_DEFINED
```

```
/* CmIdentification
```

```
 %X% %Q% %Z% %W% */
```

```
module PortTypes {
```

```
    /* This type is a CORBA unbounded sequence of Wstrings. */
```

```
    typedef sequence <wstring> WstringSequence;
```

```
    /* This type is a CORBA unbounded sequence of booleans. */
```

```
    typedef sequence <boolean> BooleanSequence;
```

```
    /* This type is a CORBA unbounded sequence of characters. */
```

```
    typedef sequence <char> CharSequence;
```

```
    /* This type is a CORBA unbounded sequence of doubles. */
```

```
    typedef sequence <double> DoubleSequence;
```

```
    /* This type is a CORBA unbounded sequence of long Doubles. */
```

```
    typedef sequence <long double> LongDoubleSequence;
```

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

typedef sequence <long long> LongLongSequence;

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

typedef sequence <long> LongSequence;

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

typedef sequence <short> ShortSequence;

/ This type is a CORBA unbounded sequence of unsigned long longs. */*

typedef sequence <unsigned long long> UlongLongSequence;

/ This type is a CORBA unbounded sequence of unsigned longs. */*

typedef sequence <unsigned long> UlongSequence;

/ This type is a CORBA unbounded sequence of unsigned shorts. */*

typedef sequence <unsigned short> UshortSequence;

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

typedef sequence <wchar> WcharSequence;

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

typedef sequence <float> FloatSequence;

};

#endif

#

C.3 LogService MODULE.

The LogService module contains the *Log* servant interface and the types necessary for a log producer to generate standard SCA log records as depicted in Figure C-5.

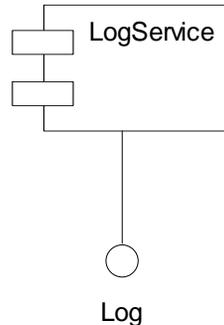


Figure C-5. LogService Module.

The following is the LogService Module IDL generated from the Rational Rose model, version 2000e.

```
//Source file: D:/projects/jtrs/sca 2.2/IDL/Log IDL Implementation
Component/LogService.idl

#ifndef __LOGSERVICE_DEFINED
#define __LOGSERVICE_DEFINED

/* CmIdentification
   %X% %Q% %Z% %W% */

/* The LogService module contains the Log servant interface and the types
   necessary for a log producer to generate standard SCA log records.
   This module also defines the types necessary to control the logging output
   of a log producer. Components that produce logs are required to implement
   configure properties that allow the component to be configured as to what
   log records it will output. */

module LogService {

    /* The LogLevelType is an enumeration type that is utilized to identify
    log levels. */

    enum LogLevelType {
        FAILURE_ALARM,
        DEGRADED_ALARM,
        EXCEPTION_ERROR,
        FLOW_CONTROL_ERROR,
        RANGE_ERROR,
        USAGE_ERROR,
        ADMINISTRATIVE_EVENT,
        STATISTIC_REPORT,
        PROGRAMMER_DEBUG1,
    }
}

```

```

PROGRAMMER_DEBUG2,
PROGRAMMER_DEBUG3,
PROGRAMMER_DEBUG4,
PROGRAMMER_DEBUG5,
PROGRAMMER_DEBUG6,
PROGRAMMER_DEBUG7,
PROGRAMMER_DEBUG8,
PROGRAMMER_DEBUG9,
PROGRAMMER_DEBUG10,
PROGRAMMER_DEBUG11,
PROGRAMMER_DEBUG12,
PROGRAMMER_DEBUG13,
PROGRAMMER_DEBUG14,
PROGRAMMER_DEBUG15,
PROGRAMMER_DEBUG16
};

/* The LogLevelSequence type is an unbounded sequence of LogLevelTypes.
The PRODUCER_LOG_LEVEL configure/query property is of the
LogLevelSequence type. */

typedef sequence <LogLevelType> LogLevelSequenceType;

/* Log producers format log records as defined in the structure
ProducerLogRecordType.

producerID : This field uniquely identifies the source of a log record.
The value is the component's identifier and is unique for each SCA
Resource
and Core Framework component with the Domain. producerName : This field
identifies the producer of a log record in textual format. This field is
assigned by
the log producer, thus is not unique within the Domain (e.g. multiple
instances of
an application will assign the same name to the ProducerName field.)

level : The level field can be used to classify the log record according
to the
LogLevelType.

logData : This field contains the informational message being logged. */

struct ProducerLogRecordType {
    /* This attribute uniquely identifies the source of a log record.
The value is unique within the Domain. The DomainManager and
ApplicationFactory are responsible for assigning this value. */
    string producerId;
    /* This attribute identifies the producer of a log record in
textual format.
This field is assigned by the log producer, thus is not unique
within the Domain
(e.g. - multiple instances of an application will assign the same
name to the
ProducerName field.) */
    string producerName;
};

```

```

        /* This attribute identifies the type of message being logged as
defined by the type LogLevelType. */
        LogLevelType level;
        /* This attribute contains the informational message being logged.
*/
        string logData;
    };

    /* A Log is utilized by CF and CORBA capable application components to
store
    informational messages. These informational messages are referred to as
'log records' in this document. The interface provides operations for
writing
    log records to a Log, retrieving LogRecords from a Log, controlling of a
Log,
    and getting the status of a Log. */

    interface Log {
        /* The AdministrativeStateType denotes the active logging state of
an
        operational Log. When set to UNLOCKED the Log will accept records
for storage, per its operational parameters. When set to LOCKED the
Log will not accept new log records and records can be read or
deleted only. */

        enum AdministrativeStateType {
            LOCKED,
            UNLOCKED
        };

        /* The AvailabilityStatusType denotes whether or not the Log is
available
        for use. When true, offDuty indicates the Log is LOCKED
(administrative state) or DISABLED (operational state). When true,
logFull indicates the Log storage is full. */

        struct AvailabilityStatusType {
            boolean offDuty;
            boolean logFull;
        };

        /* This type specifies the action that the Log should take when its
internal buffers
        become full of data, leaving no room for new records to be written.
Wrap indicates that the Log
        will overwrite the oldest LogRecords with the newest records, as
they are written to the Log.
        Halt indicates that the Log will stop logging when full. */

        enum LogFullActionType {
            WRAP,
            HALT
        };

        /* The enumeration OperationalStateType defines the Log states of
operation. When the Log is ENABLED it is fully functional and is

```

```

that
conditions
DISABLED
available for use by log producer and log consumer clients. A Log
is DISABLED has encountered a runtime problem and is not available
for use by log producers or log consumers. The internal error
that cause the Log to set the operational state to ENABLED or
are implementation specific. */

enum OperationalStateType {
    DISABLED,
    ENABLED
};

exception InvalidLogFullAction {
    string Details;
};

/* This exception indicates that a provided parameter was invalid.
*/

exception InvalidParam {
    string details;
};

/* This type provides the time format used by the Log to time-
stampLogRecords.
The Log implementation is required to produce time-stamps
compatible with the POSIX-defined time_t type. */

struct LogTimeType {
    /* This value corresponds to POSIX time_t type */
    long seconds;
    long nanoseconds;
};

/* This type provides the record ID that is assigned to a
LogRecord. */

typedef unsigned long long RecordIdType;

/* The LogRecordType defines the format of the LogRecords as stored
in the Log.
The 'info' field is the ProducerLogRecord that is written by a
client to the Log. */

struct LogRecordType {
    RecordIdType id;
    LogTimeType time;
    ProducerLogRecordType info;
};

/* The ProducerLogRecordSequence type defines a sequence of
ProducerLogRecordTypes. */

typedef sequence <ProducerLogRecordType> ProducerLogRecordSequence;

```

```
/* The LogRecordSequence type defines an unbounded sequence of
LogRecords. */

typedef sequence <LogRecordType> LogRecordSequence;

/* This operation provides the maximum number of bytes that the Log
can store.
The getMaxSize operation returns the integer number of bytes that
the Log is capable of storing.
This operation does not raise any exceptions.
@roseuid 3B268C6203B5 */
unsigned long long getMaxSize ();

/* This operation sets the maximum number of bytes that the Log can
store.
The setMaxSize operation sets the maximum size of the log measured
in number of bytes.

This operation does not return a value.

The setMaxSize operation raises the InvalidParam exception if the
size parameter passed
in is less than the current size of the Log.

The setMaxSize operation raises the InvalidParam exception if the
size parameter passed
in is less than the current size of the Log
@roseuid 3B268CAF0207 */
void setMaxSize (
    in unsigned long long size
)
    raises (InvalidParam);

/* The getCurrentSize operation provides the current size of the
log storage in bytes.

The getCurrentSize operation returns the current size of the log
storage in bytes
(i.e. if the log contains no records, get CurrentSize will return a
value of 0.).

This operation does not return any exceptions.
@roseuid 3B268D1500C4 */
unsigned long long getCurrentSize ();

/* The getNumRecords operation provides the number of records
present in the Log.

The getNumRecords operation returns the current number of
LogRecords contained in the Log.

This operation does not raise any exceptions.
@roseuid 3B268D2B00D9 */
unsigned long long getNumRecords ();
```

/* The getLogFullAction operation provides the action taken when the Log becomes full.

The getLogFullAction operation returns the Log's full action setting.

```
This operation does not return any exceptions.  
@roseuid 3B268D4603BD */  
LogFullActionType getLogFullAction ();
```

/* The setLogFullAction operation provides the mechanism to configure the action taken by a Log when it becomes full.

The setLogFullAction operation sets the action taken by a Log when its maximum size has been reached to the value specified in the action parameter.

The valid values for the action parameter, WRAP and HALT, are described by LogFullActionType.

This operation does not return a value.

```
This operation does not return any exceptions.  
@roseuid 3B268D6503B8 */  
void setLogFullAction (  
    in LogFullActionType action  
);
```

/* The getAvailabilityStatus operation is used to read the availability status of the Log.

The getAvailabilityStatus operation returns the current availability status of the Log.

```
This operation does not raise any exceptions.  
@roseuid 3B268DD302CF */  
AvailabilityStatusType getAvailabilityStatus ();
```

/* The getAdministrativeState is used to read the administrative state of the Log.

The getAdministrativeState operation returns the current administrative state of the Log.

```
This operation does not raise any exceptions.  
@roseuid 3B268DEC0376 */  
AdministrativeStateType getAdministrativeState ();
```

/* The setAdministrativeState operation provides write access to the administrative state value.

The setAdministrativeState operation sets the administrative state of the Log.

This operation does not return a value.

This operation does not raise any exceptions.

```
@roseuid 3B268E0503AE */
void setAdministrativeState (
    in AdministrativeStateType state
);
```

/* The getOperationalState operation returns the operational state of the Log.

This operation does not raise any exceptions.

```
@roseuid 3B268F0B02D8 */
OperationalStateType getOperationalState ();
```

/* The writeRecords operation provides the method for writing log records to the Log.

The operation is defined as oneway to minimize client overhead while writing to the Log.

The writeRecords operation adds each log record supplied in the records parameter to the Log.

When there is insufficient storage to add one of the supplied log records

to the Log, and the LogFullAction is set to HALT, the writeRecords method sets the availability

status logFull state to true. (i.e. if 3 records are provided in the records parameter,

and while trying to write the second record to the log, the record will not fit, then the

Log is considered to be full therefore the second and third records will not be stored in the

Log but the first record would have been successfully stored.).

The writeRecords operation writes the current local time to the time field of each record written to the Log.

The writeRecords operation assigns a unique record ID to the id field of the LogRecordlog record.

Log records accepted for storage by the writeRecords are made available for retrieval in the order received.

This operation does not return a value.

This operation does not raise any exceptions.

```
@roseuid 3B32456C03B8 */
oneway void writeRecords (
    in ProducerLogRecordSequence records
);
```

/* The getRecordIDFromTime operation is used to get the record ID of the first record in the

log with a time-stamp that is greater than, or equal to, the time specified in the parameter.

The getRecordIdFromTime operation returns the record ID of the first record in the log with a time

stamp that is greater than, or equal to, the time specified in the fromTime parameter. If the

Log does not contain a record that meets the criteria provided, then the RecordIDType returned shall correspond to the next record that will be recorded in the future. In this way, if this "future" recordID is passed into the retrieveById operation, an empty record will be returned, unless since that time records have been recorded. Note that if the time specified in the fromTime parameter is in the future, there is no guarantee that the resulting records returned will have a time stamp after the fromTime parameter if the returned recordID is subsequently used as input to the retrieveById operation.

The getRecordIdFromTime operation returns the record ID of the first record in the log with a time stamp that is greater than, or equal to, the time specified in the fromTime parameter. If the Log does not contain a record that meets the criteria provided, then the RecordIDType returned corresponds to the next record that will be recorded in the future. In this way, if this "future" recordID is passed into the retrieveById operation, an empty record will be returned, unless since that time records have been recorded.

This operation raises no exceptions.

```
@roseuid 3B33581D02C4 */
RecordIDType getRecordIdFromTime (
    in LogTimeType fromTime
);
```

/* The retrieveById operation is used to get a specified number of records from a Log.

The retrieveById operation returns a list of LogRecords that begins with the record specified by the currentID parameter and contains less than or equal to the number of records specified in the howMany parameter.

The retrieveById operation sets the inout parameter currentId to the LogRecord Id of next record after the last record in the LogRecordSequence returned. If the record sequence returned exhausts the log records, then the currentId parameter sets the LogRecordId of where the log will resume writing logs on the next write.

The retrieveById operation returns a LogRecordSequence that begins with the record specified by the currentId parameter. The number of records in the LogRecordSequence returned by the retrieveById operation is equal to the number of records specified by the howMany parameter, or the number of records available if the number of records specified by the howMany parameter cannot be met. If the record specified by currentId does not exist, the retrieveById operation returns an empty list of LogRecords and leave the currentId unchanged.

If the Log is empty, or has been exhausted, the retrieveById operation returns an empty list of LogRecords and leave the currentId unchanged.

This operation does not raise any exceptions.

```
@roseuid 3B32456D000C */
LogRecordSequence retrieveById (
    inout RecordIdType currentId,
    in unsigned long howMany
);
```

/* The clearLog operation provides the method for removing all of the LogRecords from the Log.

The clearLog operation deletes all records from the Log. The clearLog operation sets the current

size of the Log storage to zero. The clearLog operation sets the current number of records in the

Log to zero. The clearLog operation sets the logFull availability status element to false.

The clearLog operation does not return a value.

This operation does not raise any exceptions.

```
@roseuid 3B32456D003E */
void clearLog ();
```

/* The destroy operation provides a means by which an instantiated Log may be torn down.

The destroy operation releases all internal memory and/or storage allocated by theLog.

The destroy operation tears down the component (i.e. released from the CORBA environment).

The destroy operation does not return a value.

This operation does not raise any exceptions.

```
@roseuid 3B32456D007A */
void destroy ();
```

```
};
```

```
};
```

```
#endif
```

C.4 STANDARDEVENT MODULE.

The StandardEvent module contains the types necessary for a standard event producer to generate standard SCA events as depicted in Figure C-4.



Figure C-4. StandardEvent Module

```
//Source file: StandardEvent.idl
```

```
#ifndef __STANDARDEVENT_DEFINED
```

```
#define __STANDARDEVENT_DEFINED
```

```
/* CmIdentification
```

```
  %X% %Q% %Z% %W% */
```

```
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 StateChangeEvent 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 StateChangeEvent {  
    string producerId;  
    string sourceId;  
    StateChangeCategoryType stateChangeCategory;  
    StateChangeType stateChangeFrom;  
    StateChangeType stateChangeTo;  
};
```

/* Type SourceCategoryType is an enumeration that is utilized in the DomainManagementObjectAddedEventType and DomainManagementObjectRemovedEventType. Is 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;
    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;
    SourceCategoryType sourceCategory;
    Object sourceIOR;
};

};

#endif
```