Software Communications Architecture

Neli Hayes
Principal Software Architect
SCA Core Framework Team
JTRS Cluster 1 Program
The Boeing Company, Anaheim, CA
persnaz.n.hayes@boeing.com
(714) 762-9266

OMG Workshop on Distributed Object Computing for Real-Time and Embedded Systems
July 2003
Objectives

To provide an overview of reasons for creation of the SCA.
To describe what the SCA is, what it is not (i.e. where it fits into a communications systems procurement), and its goals.
To introduce all SCA specifications, supplements, and available formal training.
To provide a comprehensive overview of the SCA core architecture rule set.
To provide some general design guidelines regarding development of SCA-compliant components.
Outline

Why was the SCA Created? 8
- Tactical Communication Systems Limitations 9
- The JTRS Program 10
- JTRS Vision & Mission 11
- JTRS Requirements 12
- What is a Software Defined Radio? 13
- JTRS Target – A Software Defined Radio 14
- Inception of the SCA 15
# Outline (cont.)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the SCA?</td>
<td></td>
</tr>
<tr>
<td>- The SCA Is...</td>
<td>16</td>
</tr>
<tr>
<td>- The SCA Is Not...</td>
<td>17</td>
</tr>
<tr>
<td>- SCA Objectives</td>
<td>19</td>
</tr>
<tr>
<td>- How the SCA Achieves its Objectives</td>
<td>22</td>
</tr>
<tr>
<td><strong>SCA Specifications</strong></td>
<td>23</td>
</tr>
<tr>
<td>- SCA Specification</td>
<td>24</td>
</tr>
<tr>
<td>- SCA Support &amp; Rationale Document</td>
<td>26</td>
</tr>
<tr>
<td>- SCA API Supplement</td>
<td>28</td>
</tr>
<tr>
<td>- SCA Security Supplement</td>
<td>29</td>
</tr>
<tr>
<td>- SCA Developer’s Guide</td>
<td>30</td>
</tr>
<tr>
<td>- Subject of This Briefing</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>
Outline (cont.)

SW Architecture Overview
- JTR Set Components Integrated through the SCA
- Any Domain’s Component’s Integrated through the SCA
- SW Architecture Birds Eye View
- SW Architecture Detailed View
  - SW Architecture Major Divisions (Infrastructure & Application Layers)
  - Infrastructure Layer
    - Bus Layer
    - Network & Serial Interface Services
    - Operating System Layer
    - CORBA Middleware
    - Core Framework (CF)
  - Application Layer
    - CORBA-Capable Components
    - Adapters Allowing Use of Non-CORBA-Capable Components
- SW Architecture Benefits
Outline (cont.)

SW Architecture Infrastructure Layer

- Operating Environment (OE) 51
  - Operating System & CORBA Middleware 53
  - Core Framework (CF) 65
    - Base Application Interfaces 71
    - Framework Control Interfaces 96
      - Base Device Interfaces 102
      - Node Component Deployment & Removal Interface 110
      - Domain Component Deployment & Removal Interface 121
      - Application Installation/Un-installation Interface 121
      - Application Creation Interface 129
      - Application Configuration/Control/Termination Interface 136
    - File Service Interfaces 149
      - Log Service 154
  - Domain Profile – Component Packaging & Deployment 159
Outline (cont.)

Application Development Using the SCA (CF) 191
  - Component Development 193
  - Interconnection of Components Via Ports 195
  - General Design/Implementation Considerations 204

How CF Interacts with Application Components 208
  - Application Installation 210
  - Application Instantiation 212
    - Creation of a Single Application Component 213
    - Connecting Application Components Together 215
  - Invoking Operations on a Single Application Component 217
  - Application Tear-Down 218

SCA IDL Modules 219

References 225
Why was the SCA Created?
Tactical Communication Systems Limitations

Current tactical communications systems evolved to meet service-specific and mission-specific requirements.

Specialized functionality resulted in major limitations in communicating from one system to another, due to current radio-specific design.

Real-time information exchange and effective communication among joint forces, all the way to the war fighter in the field, are critical to assure combatant effectiveness and safety.
The Joint Tactical Radio System (JTRS) Program

A series of related joint acquisition activities, conducted by the JTRS Joint Program Office (JPO), designated Services’ cluster Program Management Offices, and other DoD agencies.

The JPO was established to pursue the development of future communications systems, capturing the benefits of technology advances in the recent years, to greatly enhance interoperability of communications systems and reduce development and deployment costs.
JTRS Vision & Mission

Provide optimal communications support for joint operations, by enabling coordination and integration of military communications through a cohesive joint tactical radio infrastructure, that provides the means of digital information exchanges, among joint war fighting elements, while enabling connectivity to civil and national authorities.

Acquire a family of affordable, high-capacity, tactical radio systems to provide interoperable and upgradeable line-of-sight, beyond line-of-sight, and wireless mobile network Control, Communications, Computers, and Intelligence (C4I) capabilities to the war fighters in the field.
To realize the JTRS Program Mission, the JTRS must be

- **Modular**, enabling additional capabilities and features to be added to JTR sets,
- **Scaleable**, enabling additional capacity (bandwidth and channels) to be added to JTR sets, and
- **Backwards-compatible**, allowing JTRS to communicate with the legacy radios it will eventually replace, allowing dynamic intra-network and inter-network routing for data transport that is transparent to the radio operator.

Hence, the JTRS target became a family of *Software Defined Radios*. 
What is a Software Defined Radio (SDR)?

Radio functionality is provided through software rather than hardware.

- Software applications provide waveform generation and processing, encryption, signal processing, and other major communications functions.
- Programmable and able to accommodate various physical layer formats and protocols.
- Multiple software modules allow implementation of different standards in the same radio system.
- Flexibility of incorporation of new functionality, without need to upgrade or replace hardware components.
- Decreased maintenance costs, due to radio receivers being re-configurable over-the-air.
JTRS Target – A Software-Defined Radio

Its characteristics can be altered allowing it to operate with any legacy waveform.

- A single unit can accommodate multi-service and multi-national capabilities.

It can be upgraded or reprogrammed using standardized APIs.

- New or improved capabilities can be incorporated easily.

Hence, the Software Communications Architecture was conceived.
Inception of the SCA

To realize the JTRS Goals and Mission, the family of JTRS software-defined radios will be designed around a Software Communications Architecture (SCA) that can support the needed functionality and expandability of the JTRS.
What is the SCA?
The SCA Is...

Is an open architecture framework that tells communications systems designers how elements of hardware and software are to operate in harmony within an SCA-compliant system.

Enables communication platforms (e.g. software defined radios) to load applications (e.g. waveforms), run these applications, and be networked into an integrated system.

Is used by communication platform (e.g. radios, etc.) hardware and software design engineers just as a building architect or planner uses a local building code to design and build homes.
The SCA Is... (cont.)

Is based on
- CORBA
- CORBA services
- CORBA Component Model
- POSIX

Is expressed in
- CORBA IDL
- UML
- XML

Not only is the SCA meant to be an open and commercially-adopted standard, but, it is based on and uses many open commercial standards.
The SCA Is Not...

**Is not** a system specification.

- It is a set of rules that constrain the design of systems, to achieve SCA objectives.
  - SCA objectives, originally conceived to meet JTRS objectives, are applicable to any communications system with the goals of component portability, interchangeability, and interoperability.
  - Comprised of interface and behavioral specifications, general rules, waveform APIs, and security requirements.
The SCA Is Not... (cont.)

Does not tell hardware and software designers how to design their equipment and programs.

- SCA requirements limited to those necessary to meet SCA-compliant system criteria, w/o restricting innovation or domain-specific requirements.
- Through adherence to standards detailed in the SCA definition document, both hardware and software designers know what equipment and programs to design.
- SCA-compliant networked systems (e.g. JTRS compliant radios, other communications systems, etc.), when designed in compliance with the SCA, will meet SCA-compliancy standards for interoperability, just as properly designed plumbing or electrical systems meet local codes for construction and safety.
The SCA Is Not... (cont.)

Is not an implementation architecture.

• It is an implementation-independent framework for development of SCA-compliant systems (e.g. JTRS SDRs).

• There are many valid definitions of the SCA architecture.
  • Both at a high system level, and at a low software and hardware level.

• A particular SCA implementation architecture is a result of joining between the SCA, the particular domain and project technical specifications, and other procurement requirements.
SCA Objectives Aligned with JTRS Goals

JTRS: Greatly increased operational flexibility and interoperability of globally deployed communications systems,

SCA: Provide for portability of applications software between different SCA implementations,

JTRS: Reduced supportability costs,

SCA: Leverage commercial standards to reduce development costs,

JTRS: Upgradeability in terms of easy technology insertion and capability upgrades, and

SCA: Reduce development time of new waveforms or other SCA-compliant applications through the ability to reuse design modules, and

JTRS: Reduced system acquisition and operation cost.

SCA: Build on evolving commercial frameworks and architectures.
The SCA Achieves its Objectives by...

Defining an open, distributed, component-based, object-oriented architecture
Separating applications from the operating environment
Segmenting application functionality
Defining common interfaces for Managing & Deploying Software Components
Defining common services & APIs to support device and application portability
SCA Specifications
To maximize SCA’s commercial application and resulting benefits…

Available at http://jtrs.army.mil/
SCA Specification

Software Architecture Definition and Rule Sets

• The SCA “operating environment” (OE).

• Services and interfaces that applications use from the OE.

• Use of OO concepts to define partitions of OE and partitions of applications built on top of the OE.

• Design constraints imposed on applications built on top of and running on the OE, in order to maximize application portability from one SCA-compliant platform to another.

SCA Specification

V2.2
Hardware Architecture Definition and Rule Sets.

- Uses OO concepts to define typical hardware partitions on most realizable systems.
- Supports platform functions being implemented in hardware or software.
- Requires complete and comprehensive documentation of all interfaces and attributes when system is built.
- Promotes Hardware Modularity
  - To support technology insertion as future programmable elements increase in capability.
  - To support additional vendors providing modules in a system.
  - Easy identification of hardware modules needed for a specific waveform or other application by software developers.
- Similar in structure to software architecture definition, but not as detailed.
  - Domain constraints play a big role in driving hardware implementation.
SCA SRD

SCA Specification companion.

Background, tutorial, and support information for the development of SCA-compliant communication systems, including JTRS software configurable radios.

SCA and SRD combination provides the framework, the rationale for that framework, and examples to illustrate the implementation of the architecture for differing domains/platforms and selected waveforms (applications).

Similar outline to SCA Specification.
SCA API Supplement

A building block structure for defining APIs between application software components.

- Defines structures associated with communication system services at various interfaces such as physical, MAC, link layer, networking, security, and I/O.

Provides significant flexibility for developers to define application-specific APIs.

Improves portability of applications and interchangeability of devices on which these applications run within SCA-compliant implementations.

Makes reuse of application components easier.
SCA Security Supplement

Requirements and services for implementing security in the SCA architecture, in order to

- Protect military secure communications, and
- Facilitate NSA certification of JTRS or other SCA-compliant products.

In form of standard APIs for implementing security in an SCA-compliant system.

Follow approaches specified in SCA API Supplement.
Overview of SCA, its services, and SCA APIs (Physical, MAC, and Link Layer APIs)

Waveform development

Creation of SCA compliant application and device components by the SCA OE

How to design SCA compliant application and device components

How to express deployment of SCA-compliant application and device components to the SCA domain

User interface connections to SCA-compliant systems
Subject of this Briefing

SCA Specification V2.2

SCA Software Architecture Overview
• The SCA Operating Environment (OE)
• OE services and interfaces used by applications
• Design Constraints imposed on applications built and running on top of OE, in order to maximize their portability from one SCA-compliant platform to another
• Packaging and deployment of SCA-compliant components

SCA Component Development Overview
• Designing SCA-compliant components
• SCA Operating Environment Sequences for installation, creation, interaction with, and tear-down of SCA-compliant components
Software Architecture Overview
JTR Set Components
Integrated through the SCA

JTR Set
JTRS: Networked JTR Sets

Radio software applications,
including Wide-band Networking Waveform

Waveforms  WNW  Minimal Network Services

SCA

Programmable Radio Hardware

Provides a layer of abstraction between radio applications software and hardware,
thus promoting portability of radio applications among SCA-compliant radio platforms,
and radio applications interoperability.
Any Domain’s Components Integrated through the SCA

Provides a layer of abstraction between a specific domain’s applications software and hardware, thus promoting portability of those applications among SCA-compliant platforms for that domain, and applications interoperability.
Software Architecture
Birds Eye View

A Communication Domain’s Set (e.g. JTR Set)
Partitioned per SCA

Example Application Layer software partitions typical of how waveforms might be implemented using the SCA

SCA Application Layer
Black Net & Link Apps
Security Apps
Red Net & Link Apps
Host Apps

SCA Infrastructure Layer

Domain Hardware

e.g. a JTR Set
Programmable Radio Hardware
Software Architecture

Major Divisions

Application Layer

SW partitions typical of how WF's might be implemented using SCA

Infrastructure Layer

Core Framework (CF)
Commercial Off-the-Shelf (COTS)

Non-CORBA Components

Modem Components
Modem Adapter
Link, Network Components
Security Adapter
Security Components
Link, Network Components
I/O Adapter
I/O Components

MAC API
LLC/Network API
Security API
LLC/Network API
I/O API

Core Framework IDL
("Logical Software Bus" via CORBA)

CORBA ORB & Services (Middleware)
CF Services & Applications
Board Support Package (Bus Layer)
Network Stacks & Serial Interface Services
Operating System

CORBA ORB & Services (Middleware)
CF Services & Applications
Board Support Package (Bus Layer)
Network Stacks & Serial Interface Services
Operating System
The Software Architecture supports reliable transport mechanisms that might include error checking and correction at the bus support level. Possible commercial bus architectures include VME, PCI, CompactPCI, Firewire, Ethernet, etc. If desired, different bus architectures could be used on the Red and Black Subsystems.
The Software Architecture relies on commercial components to support multiple unique serial and network interfaces (e.g. RS-232, RS-422, RS-423, RS-485, Ethernet, 802.x, etc.). To support these interfaces, various low-level network protocols may be used (e.g. PPP, SLIP, LAPx, and others). Elements of waveform networking functionality may also exist at the Operating System layer (e.g. a commercial IP stack that performs routing between waveforms.)
The Software Architecture includes real-time embedded OS functions to provide needed support for Infrastructure and Application layers. For example:

- Booting the processor
- Multi-threading support
- I/O support
- Inter-process and inter-device support
- Other general-purpose capabilities required for real-time embedded applications

As such, may be present on all CORBA-capable processors in a system.
May be tailored to support specific target/board environments. For example:

- A Board Support Package (BSP), can tailor the RTOS for specific processor board elements, including the device drivers that support the specific devices/chip sets resident on the board.
- Device drivers to support the necessary inter-processor communication required by the ORB.
The architecture requires a standard OS interface for OS services in order to facilitate portability of applications. As POSIX is an accepted industry standard, and POSIX and its real-time extensions are compatible with the requirements to support the OMG CORBA specification, the SCA defines a minimal POSIX profile, known as the SCA Application Environment Profile (AEP; SCA Appendix A), based on the Real-time Controller System Profile (PSE52) as defined in POSIX 1003.13, to meet SCA requirements. More on OS usage restrictions enforced on the Infrastructure and Application Layers later.
Infrastructure
CORBA Middleware

Distributed processing is a fundamental aspect of the system architecture and is based on the OMG CORBA specification. It is used in the Infrastructure Layer as the message passing technique for the distributed processing environment.

Why CORBA?
Infrastructure CORBA Middleware (cont.)

**OMG’s CORBA specification:** a specification for a “software bus” or “software back plane”.
- Middleware for establishing relationships between clients and servers in a distributed system.
- Provides interoperability between applications on different machines, different OSs, different networking protocols, and different programming languages.
- Supports location transparent invocation of server objects by client objects.
- Components are “plugged into” the software bus, and are visible to other components.
- More later on the ORB and CORBA services used in the Infrastructure Layer.
The CF is the essential ("core") set of open application-layer interfaces and services to provide an abstraction of the underlying software and hardware layers for software application designers. More on CF later.
Application Layer
CORBA Capable Components

Applications perform user level functions.

Required to use CF interfaces & services.

Direct OS access limited to SCA POSIX Profile.

Black Hardware Bus

Red Hardware Bus

Operating System
Network Stacks & Serial Interface Services
Board Support Package (Bus Layer)

Corresponding hardware bus layers:
Red Hardware Bus
Black Hardware Bus

Application Layer
CORBA Capable Components

Applications
Core Framework (CF)
Commercial Off-the-Shelf (COTS)

Core Framework IDL
"Logical Software Bus" via CORBA

I/O API
Security API
LLC/Network API
MAC API
Physical API

Applications perform user level functions.
Adapters Allowing Use of Non-CORBA Capable Components in the Application Layer

- **Based on Adapter Design Pattern**
- Translate between “Legacy” & CORBA-capable components

- **Implement CF Interface**
- Translation transparent to CORBA-capable components

**APPLICATION LAYER**

**APPLICATION LAYER**

**INFRASTRUCTURE LAYER**

- **Applications**
- **Core Framework (CF)**
- **Commercial Off-the-Shelf (COTS)**

**Non-CORBA Components**

- **Non-CORBA Modem Components**
- **Non-CORBA Security Components**

**Modem Components**

**Link, Network Components**

**Security Components**

**I/O Components**

**I/O Adapter**

**Network Stacks & Serial Interface Services**

**Board Support Package (Bus Layer)**

**CORBA ORB & Services (Middleware)**

**CF Services & Applications**

**Operating System**

**Network Stacks & Serial Interface Services**

**Board Support Package (Bus Layer)**

**Black Hardware Bus**

**Red Hardware Bus**

**“Logical Software Bus” via CORBA**

**MAC API**

**LLC/Network API**

**Security API**

**LLC/Network API**

**I/O API**

**Core Framework IDL**

**Core Framework (CF)**

**Commercial Off-the-Shelf (COTS)**

**Applications**

**Non-CORBA Components**

**Non-CORBA Modem Components**

**Non-CORBA Security Components**

**Modem Components**

**Link, Network Components**

**Security Components**

**I/O Components**

**I/O Adapter**

**Network Stacks & Serial Interface Services**

**Board Support Package (Bus Layer)**

**CORBA ORB & Services (Middleware)**

**CF Services & Applications**

**Operating System**

**Network Stacks & Serial Interface Services**

**Board Support Package (Bus Layer)**

**Black Hardware Bus**

**Red Hardware Bus**
Software Architecture

Benefits

Maximizes use of commercial protocols and products.

Isolates core and non-core applications from underlying hardware through multiple layers of open, commercial software infrastructure.

Provides for a distributed processing environment through CORBA.

Hence, provides software application portability, reusability, and scalability.
Infrastructure Layer
Operating Environment
Operating Environment

- **ORB Naming Service Event Service**
  - Non-CORBA Modem Components
  - Modem Components
  - Modem Adapter
  - Link, Network Components
  - Security Adapter
  - Security Components
  - Security Adapter
  - Link, Network Components
  - I/O Components
  - I/O Adapter
  - I/O Components
  - MAC API
  - LLC/Network API
  - Security API
  - LLC/Network API
  - I/O API

- **CF Log Service**
  - Non-CORBA Security Components
  - Security Components
  - Security Components
  - Security Components
  - I/O Adapter
  - I/O Components

- **Core Framework IDL**
  - (“Logical Software Bus” via CORBA)
  - CORBA ORB & Services (Middleware)
  - CORBA ORB & Services (Middleware)
  - CF Services & Applications
  - CF Services & Applications
  - Operating System
  - Operating System
  - Network Stacks & Serial Interface Services
  - Network Stacks & Serial Interface Services
  - Board Support Package (Bus Layer)
  - Board Support Package (Bus Layer)

- **Applications**
- **Core Framework (CF)**
- **Commercial Off-the-Shelf (COTS)**

- **Physical API**
- **RF**

- **Operating Environment**

- **Black Hardware Bus**
- **Red Hardware Bus**
Operating Environment’s Job

Impose design constraints on waveform and other applications.

- To provide increased portability of applications from one SCA-compliant platform to another.

These design constraints include:

- Specified interfaces between the CF and application software.
- Restrictions on waveform usage of OS and CORBA middleware APIs.
Operating Environment
Operating System
OS Usage Restrictions

CORBA API

All Application Components

CORBA ORB

Core Framework: Framework Control & Framework Services Interfaces

Logical Device is an Adapter for HW-specific devices
More on Logical Device later

CORE COMPONENTS OR DEVICE DRIVERS

non-CORBA components

OS access unlimited

OS (function) that supports SCA AEP

(unlimited proprietary APIs for system development).

Any vendor-provided OS function calls

Commercial Off-the-Shelf (COTS)

OE Operating System

OS access limited to SCA AEP

OS access unlimited

OS access unlimited

Applications use CF for all File access

CORE COMPONENTS OR DEVICE DRIVERS

non-CORBA components provide
access to hardware devices / functionality
not available on a CORBA-capable processor

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted

Commercial Off-the-Shelf (COTS)

OE Operating System

OS usage restricted
Operating Environment
Object Request Broker
Object Request Broker

- Infrastructure Layer OE components (e.g. CF)
- Application Layer components (e.g. waveform and non-waveform)

As of SCA V2.2, all Infrastructure and Application Layer components are restricted to minimum CORBA for portability.

Adapter wrapping a Legacy non-CORBA capable Application Layer component with a CORBA wrapper.
Operating Environment
CORBA Naming Service
Naming Service

1: Servers bind their objects to names so that clients can find them

2: Clients find object references distributed throughout the system by their assigned name

3: Request Services

• A CORBA server that associates human readable names with CORBA object references.
• Based on OMG Naming Service Specification OMG Document formal/00-11/01.
• Must support bind(), bind_new_context(), unbind(), destroy(), resolve().

• Centralized repository of object references in the CORBA environment.
• Server Bindings organized in Naming Contexts.

Commercial Off-the-Shelf (COTS)

OE CORBA Middleware Naming Service
Naming Service Usage in the OE

Typical OE Naming Service Servers
- CF component that installs applications in the CF domain
- Deployed domain components
- Instantiated application components

Typical OE Naming Service Clients
- CF: connect deployed components to services in CF domain
- Application Layer components: installation of application software through CF
- CF: create, setup, connect, and tear down instantiated application components.
- More on this later.

1: Servers bind their objects to names so that clients can find them.
2: Clients find object ORs distributed throughout the system by their assigned name.
3: Request Services.
Operating Environment
CORBA Event Service
Event Service

Based on OMG’s Event Service specification.
- Push interfaces (*PushConsumer* and *PushSupplier*) of the *CosEventComm* CORBA module as described in OMG Document formal/01-03-01: Event Service, v1.1.
- Compatible IDL for the *CosEventComm* CORBA module in the OMG Document formal/01-03-02: Event Service IDL, v1.1.

Decoupled, asynchronous communication model between clients and servers.
- Event producers and consumers

Suppliers “publish” information without knowing who the consumers are.
Consumers “subscribe” to information without regard to the information source.
Suppliers are shielded from exceptions resulting from any of the consumer objects being unreachable or poorly behaved.
Event Service (cont.)

Central Role in Event Service Specification
- Supplier(s)/consumer(s) intermediary.
- Acts as consumer to supplier(s) and supplier to consumer(s).
- Supplier and consumer registrations.
- Timely and reliable event delivery to all registered consumers.
- Error handling associated with unresponsive consumers.

Supplier 1

Supplier 2

Supplier …

Supplier N

Event Channel

Consumer 1

Consumer 2

Consumer …

Consumer N
Event Service Usage in the OE

**Used by CF for**
- Connection of CF domain components to required event channels, and
- Connection/disconnection of application components to required event channels in the CF domain, during instantiation/tear down of an application in the CF domain.
- Logging SCA-specified events to SCA-specified event channels.

![Diagram](image)
Event Service Usage in the OE (cont.)

OE provides two standard event channels:

- **Incoming Domain Management Event Channel**
  - Allows the CF domain to become aware of changes in the domain.
    - e.g. device state changes.

- **Outgoing Domain Management Event Channel**
  - Allows CF domain clients to become aware of changes in the domain.
    - e.g. Human Computer Interface (HCI) application receiving events pertaining to device/service/application software/instantiated application additions/removals from domain.

Application developers allowed to setup other “non-standard” event channels.
Operating Environment
Core Framework
Core Framework

The essential ("core") set of open application-layer interfaces and services that

- Provide an abstraction of the underlying software and hardware layers for software application designers.
- Provide for deployment, management, interconnection, and intercommunication of software application components in a distributed embedded communication system.
- Through imposed design constraints provide increased portability of these application components from one SCA-compliant platform to another.
  - Specified interfaces between CF and application software.
  - Specified behavioral requirements for application software.
CF Interfaces
Top-Level View
CF Interface Groupings

Base Application Interfaces

Framework Control Interfaces

File Service Interfaces
CF Interfaces and Services

**Base Application Interfaces** (Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory, and Resource) that provide a common set of interfaces for developing software application components and exchanging information between them.

**Framework Control Interfaces** that provide the means for control and management of hardware assets, applications, and domain (system).

- **Device Interfaces** (Device, LoadableDevice, ExecutableDevice, AggregateDevice)
- **Device Management Interfaces** (DeviceManager)
- **Domain Management Interfaces** (Application, ApplicationFactory, DomainManager)

**File Service Interfaces** (File, FileSystem, FileManager) that provide the means for distributed file access services.
CF Component Packaging and Deployment

The **Domain Profile**, is a series of eXtensible Markup Language (XML) files, based on...

the CORBA Component Specification, and SCA-defined XML Document Type Definitions (DTDs)...

...expressing the packaging and deployment of software application components and related hardware assets into an SCA-compliant system.
CF Base Application Interfaces
CF Base Application Interfaces

Defined by CF requirements, implemented by application developers.

Unimplemented framework of operations for creation, initialization, configuration, control, test, and tear down of application components in the system.
Port Interface

Provides a specialized connectivity to a component. Used to setup and tear down connections between CORBA components (mainly application components) in the CF domain. How?

<<Interface>>

Port

connectPort(connection : in Object, connectionId : in string) : void
disconnectPort(connectionId : in string) : void
A Component can have a Set of Ports

A component may have 0 or more ports as needed.
The implementation for each port is specific to the component.
A Component Defines a Component-Specific Port by...

Creating a specific port type by defining an interface that inherits from Port.

- Port Types
  - Command and Control, Data, Status.
  - Basic push and pull Ports defined in the SCA.

Establishing the operations for transferring data and control.

Establishing the meaning of data and control values.

Specifying whether the port is a “uses” or a “provides” port.
“Uses” and “Provides”

Ports

Uses Port
- A port that uses some set of services provided by a provider component.
- All uses ports implement the Port interface.

Provides Port
- A port that provides some set of services at an interface.
How do Component Ports get Connected?

Connection Definitions
- Definition of how the component ports get connected is described in the following Domain Profile files:
  - (An application’s) Software Assembly Descriptor
  - (A CF domain node’s) Device Configuration Descriptor
  - Covered later when we go over the Domain Profile.

By Whom?
- Ports are connected by CF’s Framework Control Interfaces in charge of deploying and managing components in the CF domain (i.e. DomainManager), and instantiating applications in the CF domain (i.e. ApplicationFactory).

How?
- Covered later when we cover CF’s Framework Control Interfaces (DomainManager and ApplicationFactory).
Port Interface Advantages

Supports the connection concepts in the CORBA Components Specification where any provides port type can be connected to another uses port, as long the uses port understands the port, which it is connecting.

The uses port can be behave either as a push producer or a pull consumer. Likewise, the provides port type can be behave either as a push consumer or pull producer.

The fan-in, fan-out, or on-one-on implementation of a port is allowed, and is component dependent.
PortSupplier Interface

Provides the means for a component that has a Port to make the Port available to those who need it. Mainly, those in charge of connecting the Port to other(s).

```java
<<Interface>>
PortSupplier

getPort(name : in string) : Object
```
Use of PortSupplier Interface in CF

The main PortSupplier interface users in the CF are the Framework Control interfaces DomainManager (during deploying components to the CF domain) and ApplicationFactory (during instantiation of an application in the CF domain).

They use a component’s getPort operation, as directed by the Domain Profile, to retrieve uses and provides ports in order to connect the component to services or other components in the domain.
LifeCycle Interface

Provides a generic means for initializing and releasing a resource, application, or device in the CF domain.

<<Interface>>

LifeCycle

initialize() : void
releaseObject() : void
LifeCycle Users

The component initialization *LifeCycle* interface users in the CF are the Framework Control interfaces *DeviceManager* (during deploying components to the CF domain) and *ApplicationFactory* (during instantiation an application in the CF domain), after a component has been deployed and its object reference has been obtained.

The component tear-down *LifeCycle* interface users in the CF are the Human Control Interface (HCI) or other domain client to release an application or device, and the CF Application interface to release an application’s components as a part of application termination.
TestableObject Interface

Provides a generic means for black box testing a resource, application, or device in the CF domain.

<<Interface>>
TestableObject

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

Location of Component Test IDs, Input, and Output Values
Component’s Properties Descriptor referenced from the component’s Software Package Descriptor in the CF Domain Profile.
TestableObject Interface

Uses and Advantages

May be used by an HCI to test an Application or Device component within the domain.

Using this interface, it would be possible to implement a generic HCI that could work for all Applications and Devices since this interface is based upon a CF Domain Profile XML element.

Currently there is no requirement to perform initial testing of a deployed domain component DeviceManager or an application’s components by the ApplicationFactory.
PropertySet Interface

Provides a generic means of configuring and retrieving a component’s properties.

<<Interface>>

PropertySet

configure(configProperties : in Properties) : void
query(configProperties : inout Properties) : void
Properties or id (name)/value pairs, is a concept from the CORBA Components, Telecommunications Information Networking Architecture (TINA), and CORBA property service specifications.
Use of PropertySet Interface in CF

If the component has any configure properties defined in the CF Domain Profile, this interface is used to configure the component by `DeviceManager` (during deploying components to the CF domain) and `ApplicationFactory` (during instantiation of an application in the CF domain), after the component has been deployed and its object reference has been obtained.

Can also be used by a HCI to configure an `Application`, `Device`, `DeviceManager`, and `DomainManager` within the domain.
PropertySet Interface

Advantages

Promotes the possibility of having a generic HCI based on the corresponding CF Domain Profile XML element to configure and query components.

Provides the capability of adding configure and query properties to a component’s implementation without having to change an IDL interface definition.

- In this case, the component’s appropriate Domain Profile XML file would have to be updated.
Resource Interface

Resource Interface

PortSupplier

getPort()

LifeCycle

initialize()
releaseObject()

PropertySet

configure()
query()

TestableObject

runTest()

Unique identifier, so that each component logging messages can be identified as the producer of the log message.

Specialization of a minimal set of interfaces needed to initialize, configure, control, and tear-down a component (e.g. an application’s resources, a Device, or an Application).
Resource Interface
Advantages

*Resource* supports a set of behavior that enables *Application* delegation and teardown and consistent component deployment between *ApplicationFactory* (during application instantiations) and *DeviceManager* (during domain component deployment).

*Resource* is used by an HCI (or other domain management clients) to configure an *Application* or *Device* within the domain.

Makes it possible to have a generic HCI that could work for all *Applications* and *Devices*. 
# ResourceFactory Interface

*Modeled after the Factory design pattern*

*Provides a generic framework of operations for creating and destroying Resources and obtaining a resource without knowing its identity*

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
</table>
| ResourceFactory | - identifier : string  
|               | - createResource(resourceld : in string, qualifiers : in Properties) : Resource  
|               | - releaseResource(resourceld : in string) : void  
|               | - shutdown() : void                                                         |

*Resource Factory Can Create Resources of Different Types*

- The resource’s Software Package Descriptor in the Domain Profile specifies properties dictating the resource type.
- These properties are passed in as “qualifiers” to createResource(), dictating the type of resource to be created.
ResourceFactory Resource
Creation and Destruction

Creation
- If the Resource has not yet been created, it is created in the same process space as the ResourceFactory.
- If the Resource has already been created, its reference count is incremented and a copy of it is returned.

Destruction
- If the specified Resource’s reference count is greater than one, it is decremented.
- Otherwise, the specified Resource is torn down.
ResourceFactory Interface

Advantages

The reference counting feature of the ResourceFactory is beneficial for managing Resources that are used by multiple applications.

Eliminates the constant redeployment of the shared components across application boundaries.
Implementation of a ResourceFactory is Optional

An application is not required to use ResourceFactories to obtain, create, or tear down resources. The application’s Software Profile specifies which application components require ResourceFactories.

- That is,
  - for which application’s components does the CF ApplicationFactory create and use a ResourceFactory (instead of directly executing the application’s component’s entry point) during instantiation of the application, and correspondingly,
  - for which application components does the CF Application interface use a ResourceFactory to destroy the application component, (instead of directly terminating the component’s running task or process).
Base Application Interfaces Model A
Component’s Maturity Cycle

- **Initializeable**
  - construct
- **Startable**
  - start
  - stop
  - initialize
  - configure (may also be required)
  - releaseObject
- **Started**
  - destroy

inherits from

uses

Port Supplier LifeCycle TestableObject PropertySet

Resource

ResourceFactory
CF Framework Control
Interfaces
Definition of “Domain” in the CF Problem Space

Abstraction of a computing node that
- allows the control and monitoring of node resources (i.e. CPU, processes, memory, file systems),
- Implements the OE,
- Executes a DeviceManager, and
- Executes installed OE and CF services (CORBA Naming Service, Event Service, FileSystem, Log Service, etc.).

Provides instances of aggregated objects (get X, add X, remove X)

- IDM Event Channel - Keeps Domain aware of Domain Changes
- ODM Event Channel - Keeps Domain clients aware of Domain Changes

Node

Domain

SW Application Instance

Event Channel

FileManager

SW Application Instance

0..n

0..n

+2..n

1

0..n
How does the CF Domain Get “Created”? 

Each node and its resources are created (i.e. OS booted, file systems created, installed OE and CF services started (CORBA Naming Service, Event Service, FileSystem, Log Service), DeviceManager representing the Node created, DeviceManager components (devices, services, additional file systems, etc.) created and deployed on the Node.

The entity in charge of managing the domain is created.

Node

+2..n

Event Channel

Domain

1

FileManager
Deploying and removing a node’s components to/from the domain (i.e. registering and un-registering a DeviceManager along with its devices and services to/from the domain)

Retrieving the domain’s nodes (DeviceManagers), software applications, and software application instances.

Creating/controlling/terminating sw application instances.

Installing/uninstalling application software to/from the domain.
CF Domain Services are Provided by the following Framework Control Interface Groupings

Base Device Interfaces

Device Management Interfaces

Domain Management Interfaces
CF Domain Services are Mapped to the following Framework Control Interfaces

- Application Instantiation Interface
- Application Configuration/Control/Termination Interface
- Base Device Interfaces
- Node Component Deployment/Removal Interface
- Domain Component Deployment/Removal & Application Installation/Uninstallation Interface
CF Framework Control
Base Device Interfaces
CF Base Device Interfaces

Defined by CF requirements, implemented by application developers.

Framework for a device’s software profile attribute, state management and status attributes, and capacity allocation/deallocation operations

Framework of operations for loading/unloading software modules onto a device processor’s memory

Framework of operations for executing/terminating processes/threads on a device

Framework of attributes and unimplemented operations representing the functional abstraction of a logical device (zero or more hardware devices).
Device Interface

Basic definition of all Logical Devices in a system. Logical Device - an abstraction of 0 or more hardware devices. Since it’s a Resource, it can be created, controlled, and terminated like a Resource.

Software Profile attribute – Each device has a Software Package Descriptor (SPD) in the Domain Profile that defines the logical Device capabilities (data/command uses and provides ports, configure/query properties, capacity properties, status properties, etc.).

Implementation depends on the device’s capacity model.

State Management and Status Attributes – Devices have states as they are viewed as managed resources within the system. Based on X.731 OSI State Management Functions.

adminState : AdminType
compositeDevice : AggregateDevice
label : string
operationalState : OperationalType
softwareProfile : string
usageState : UsageType
allocateCapacity(capacities : in Properties) : boolean
deallocateCapacity(capacities : in Properties) : void

LOCKED, UNLOCKED, SHUTTING_DOWN
ENABLED, DISABLED
IDLE, ACTIVE, BUSY
Device Interface
Advantages

The use of the Domain Profile Software Profile Descriptor allows a generic HMI to be written to interface with any vendor’s Device and to be used by a CF DomainManager and ApplicationFactory for deployment of components onto devices.

Since the Device interface merely provides an abstraction of hardware, it may be the case that many Devices can exist per physical hardware element.

- A device such as a programmable modem could present itself as both a Code Division Multi-Access (CDMA) and a Time Division Multiple Access (TDMA) modem Device simultaneously.

This abstraction also allows physical devices to be emulated.

- For example, a Device such as a printer could be emulated on a general-purpose processor. Instead of providing a printed page of the information sent to it, the printer Device could present the page graphically on a display.
LoadableDevice Interface

- Extends the Device interface by adding software loading and unloading behavior for a particular device.
- Implementation is OS-dependent.

Examples: modem, FPGA

The load module file name and load type are located in the device component’s Software Package Descriptor in the Domain Profile.

Possible Load Types
- KERNEL_MODULE
- DRIVER – Device Driver
- SHARED_LIBRARY – Dynamic Linking
- EXECUTABLE – Main POSIX Process

```
<<Interface>>
Device

<<Interface>>
LoadableDevice

load(fs : in FileSystem, fileName : in string, loadKind : in LoadType) : void
unload(fileName : in string) : void
```
**ExecutableDevice Interface**

- Extends the LoadableDevice interface by adding execute and terminate behavior for a device.
- Used for devices with an OS (e.g. VxWorks, LynxWorks, Linux, Integrity, etc.) that supports creation of threads/processes.
- Implementation is OS-dependent.

**Examples:** General Purpose Processors (Intel, PowerPC)

```
<<Interface>>
ExecutableDevice

execute(name : in string, options : in Properties, parameters : in Properties) : ProcessID_Type
terminate(processId : in ProcessID_Type) : void
```
ExecutableDevice Interface (cont.)

Name of a function (thread) or file (process), depending on the OS.

- Optional stack size and priority
- Specified in the device’s SPD in the Domain Profile

• Id/value pairs converted to (argc, argv) format.
• Some are SCA-defined required parameters (e.g. DEVICE_MANAGER_IOR, NAMING_SERVICE_IOR, etc.).
• Some are optional user-specified parameters specific to the device component’s implementation. (Specified in the device’s SPD in the Domain Profile.)

execute(name : in string, options : in Properties, parameters : in Properties) : ProcessID_Type
terminate(processId : in ProcessID_Type) : void
AggregateDevice Interface

This Interface can be provided by a Device through a "provides port" for any Device that is capable of an aggregate relationship.

An aggregated logical Device adds itself to a composite Device. An aggregate Device uses this interface to add or remove itself from the composite Device when it is being created or torn-down.

List of Devices that have been added to this Device through the addDevice( ) operation and have not yet been removed through the removeDevice( ) operation.

Examples:
- INFOSEC Device with “n” crypto channel Devices
- I/O Device with “n” ports

AggregateDevice

devices : DeviceSequence

addDevice(associatedDevice : in Device) : void
removeDevice(associatedDevice : in Device) : void

<<Interface>> Device

0..n

<<Interface>> AggregateDevice

0..1
CF Framework Control Interface for Node Component Deployment and Removal
Node Component Deployment and Removal Interface

Responsible for creation of logical Devices, and launching service applications on these logical Devices during a CF Node startup, and deploying and removing these logical Devices and services to/from the CF Domain, during CF startup, shutdown, or dynamically at runtime.
What the DeviceManager Represents in the CF Domain

A given system is composed of a set of nodes (CORBA-capable boards) with persistent components deployed and running on them.

Each node is associated with a DeviceManager. And each node is viewed as an independent object within the system.
DeviceManager’s Job

The DeviceManager manages a set of persistent components for a given node within a domain.

- Logical Devices and services (e.g., Log, Event Service, Naming Service, domain-specific services and devices).

It manages these persistent components by

- Starting them up as specified in the Domain Profile file, known as the Device Configuration Descriptor (DCD) file, AKA DeviceManager Profile
- Provides operations for deploying them to the CF domain as directed by the DCD during DeviceManager startup,
- And removing them from the CF domain upon DeviceManager shutdown.
DeviceManager’s Job (cont.)

During startup of Logical Devices and service components, the DeviceManager supplies a set of well-defined executable parameters to the component to promote portability of logical Devices and services and consistent behavior from one SCA-compliant system to another SCA-compliant system that hosts these components.

- DEVICE_MGR_IOR
- PROFILE_NAME
- DEVICE_ID
- DEVICE_LABEL
- CompositeDEVICE_IOR
- SERVICE_NAME
- Component-specific startup parameters defined in the component’s Software Profile (SPD) processed by DeviceManager
DeviceManager Interface

### Interface Definitions

**PortSupplier**

- `deviceConfigurationProfile : string`
- `fileSys : FileSystem`
- `identifier : string`
- `label : string`
- `registeredDevices : DeviceSequence`
- `registeredServices : ServiceSequence`

**PropertySet**

**DeviceManager**

- `configure(configProperties : in Properties) : void`
- `GetComponentImplementationId(componentInstantiationId : in string) : string`
- `getPort(name : in string) : Object`
- `query(configProperties : inout Properties) : void`
- `registerDevice(registeringDevice : in Device) : void`
- `registerService(registeringService : in Object, name : in string) : void`
- `shutdown() : void`
- `unregisterDevice(registeredDevice : in Device) : void`
- `unregisterService(registeredService : in Object, name : in string) : void`
Elements of a DeviceManager Node

As nodes are removed or added to the system (DomainManager), the set of elements belonging to a node are easily identified by the attributes of the corresponding DeviceManager interface.
DeviceManager’s FileSystem

A node usually has some file system associated with it, represented by the DeviceManager filesystem attribute. The underlying implementation for a file system may be implemented as a FileManager when the node consists of many file systems. This file manager is given out as a file system since the FileManager interface is derived from the FileSystem interface. Covered later.
Registration of Logical Devices and Services with DeviceManager

Deployed logical Devices and services created by the DeviceManager, register/unregister themselves to/from their DeviceManager via the register/unregister Device/Service operations. If the DeviceManager has already been deployed to the Domain, the DeviceManager will in turn deploy/remove these components to/from the Domain, using the DomainManager interface.

These operations can also be used to dynamically add/remove devices/services from a DeviceManager after startup, during CF’s lifetime.

registerDevice(registeringDevice : in Device) : void
registerService(registeringService : in Object, name : in string) : void
unregisterDevice(registeredDevice : in Device) : void
unregisterService(registeredService : in Object, name : in string) : void
The DeviceManager interface inherits the PropertySet interface in order to manage implementation properties that are described in its Software Package Descriptor (SPD) file and to change its producer log level properties.
Services for DeviceManager’s Use

The PortSupplier interface inherited by the DeviceManager interface is used to connect services (e.g., Log) to the DeviceManager.
CF Framework Control Interface for Domain Component Deployment and Removal & Application Installation and Un-installation
DomainManager’s Job

To support the instantiation and tear-down of applications in the CF domain, the DomainManager provides 2 kinds of services:

First, it provides the means for deployment, interconnection, disconnection, and removal of persistent components to/from the CF domain during DomainManager startup, shutdown, or dynamically at run-time (DeviceManagers, devices, services, event channels and event producer(s) and consumer(s))…
DomainManager's Job (cont.)

...Next, it services Domain Application installation and un-installation requests by creating/destroying a corresponding ApplicationFactory for each installed Application.
DomainManager Interface

The DomainManager interface allows an HCI to
- Configure the domain
- Get the domain’s capabilities (installed and instantiated applications and deployed logical Devices)
- Initiate maintenance functions

<table>
<thead>
<tr>
<th>PropertySet</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>DomainManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationFactories : ApplicationFactorySequence</td>
</tr>
<tr>
<td>applications : ApplicationSequence</td>
</tr>
<tr>
<td>deviceManagers : DeviceManagerSequence</td>
</tr>
<tr>
<td>domainManagerProfile : string</td>
</tr>
<tr>
<td>fileMgr : CF::FileManager</td>
</tr>
<tr>
<td>identifier : string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>installApplication(profileFileName : in string) : void</td>
</tr>
<tr>
<td>registerDevice(registeringDevice : in Device, registeredDeviceMgr : in DeviceManager) : void</td>
</tr>
<tr>
<td>registerDeviceManager(deviceMgr : in DeviceManager) : void</td>
</tr>
<tr>
<td>registerService(registeringService : in Object, registeredDeviceMgr : in DeviceManager, name : in string) : void</td>
</tr>
<tr>
<td>registerWithEventChannel(registeringObject : in Object, registeringId : in string, eventChannelName : in string) : void</td>
</tr>
<tr>
<td>uninstallApplication(applicationId : in string) : void</td>
</tr>
<tr>
<td>unregisterDevice(unregisteringDevice : in Device) : void</td>
</tr>
<tr>
<td>unregisterDeviceManager(deviceMgr : in DeviceManager) : void</td>
</tr>
<tr>
<td>unregisterFromEventChannel(unregisteringId : in string, eventChannelName : in string) : void</td>
</tr>
<tr>
<td>unregisterService(unregisteringService : in Object, name : in string) : void</td>
</tr>
</tbody>
</table>
DomainManager Provides Operations for...

**Registration/unregistration of DeviceManager’s, devices and services to/from the CF domain.**

```plaintext
<<Interface>>
DomainManager

applicationFactories : ApplicationFactorySequence
applications : ApplicationSequence
deviceManagers : DeviceManagerSequence
domainManagerProfile : string
fileMgr : CF::FileManager
identifier : string

installApplication(profileFileName : in string) : void
registerDevice(registeringDevice : in Device, registeredDeviceMgr : in DeviceManager) : void
registerDeviceManager(deviceMgr : in DeviceManager) : void
registerService(registeringService : in Object, registeredDeviceMgr : in DeviceManager, name : in string) : void
unregisterDevice(unregisteringDevice : in Device) : void
uninstallApplication(applicationId : in string) : void
unregisterDeviceManager(deviceMgr : in DeviceManager) : void
unregisterFromEventChannel(unregisteringId : in string, eventChannelName : in string) : void
unregisterService(unregisteringService : in Object, name : in string) : void
```
DomainManager Provides Operations for...

Installation and uninstallation of software applications to/from the CF domain.

```
<<Interface>>
PropertySet

<<Interface>>
DomainManager

applicationFactories : ApplicationFactorySequence
applications : ApplicationSequence
deviceManagers : DeviceManagerSequence
domainManagerProfile : string
fileMgr : CF::FileManager
identifier : string

installApplication(profileFileName : in string) : void
registerDevice(registeringDevice : in Device, registeredDeviceMgr : in DeviceManager) : void
registerDeviceManager(deviceMgr : in DeviceManager) : void
registerService(registeringService : in Object, registeredDeviceMgr : in DeviceManager, name : in string) : void
registerWithEventChannel(registeringObject : in Object, registeringId : in string, eventChannelName : in string) : void
uninstallApplication(applicationId : in string) : void
unregisterDevice(unregisteringDevice : in Device) : void
unregisterDeviceManager(deviceMgr : in DeviceManager) : void
unregisterFromEventChannel(unregisteringId : in string, eventChannelName : in string) : void
unregisterService(unregisteringService : in Object, name : in string) : void
```
DomainManager Provides Operations for...

**<Interface>**

**PropertySet**

- **DomainManager**

  | applicationFactories : ApplicationFactorySequence |
  | applications : ApplicationSequence               |
  | deviceManagers : DeviceManagerSequence           |
  | domainManagerProfile : string                    |
  | fileMgr : CF::FileManager                        |
  | identifier : string                              |

  | installApplication(profileFileName : in string) : void |
  | registerDevice(registeringDevice : in Device, registeredDeviceMgr : in DeviceManager) : void |
  | registerDeviceManager(deviceMgr : in DeviceManager) : void |
  | registerService(registeringService : in Object, registeredDeviceMgr : in DeviceManager, name : in string) : void |
  | registerWithEventChannel(registeringObject : in Object, registeringId : in string, eventChannelName : in string) : void |
  | uninstallApplication(applicationId : in string) : void |
  | unregisterDevice(unregisteringDevice : in Device) : void |
  | unregisterDeviceManager(deviceMgr : in DeviceManager) : void |
  | unregisterFromEventChannel(unregisteringId : in string, eventChannelName : in string) : void |
  | unregisterService(unregisteringService : in Object, name : in string) : void |
DomainManager Provides Access to the Domain’s...

<<Interface>>
PropertySet

<<Interface>>
DomainManager

- Installed Applications
- Instantiated Applications
- Registered DeviceManagers (and their devices and services)
- FileSystem

applicationFactories : ApplicationFactorySequence
applications : ApplicationSequence
deviceManagers : DeviceManagerSequence
domainManagerProfile : string
fileMgr : CF::FileManager
identifier : string

installApplication(profileFileName : in string) : void
registerDevice(registeringDevice : in Device, registeredDeviceMgr : in DeviceManager) : void
registerDeviceManager(deviceMgr : in DeviceManager) : void
registerService(registeringService : in Object, registeredDeviceMgr : in DeviceManager, name : in string) : void
registerWithEventChannel(registeringObject : in Object, registeringId : in string, eventChannelName : in string) : void
uninstallApplication(applicationId : in string) : void
unregisterDevice(unregisteringDevice : in Device) : void
unregisterDeviceManager(deviceMgr : in DeviceManager) : void
unregisterFromEventChannel(unregisteringId : in string, eventChannelName : in string) : void
unregisterService(unregisteringService : in Object, name : in string) : void
CF Framework Control
Application Creation Interface
Installation of an Application...

...results in creation of a corresponding ApplicationFactory for that type of Application.
The created ApplicationFactory provides the interface to request the instantiation of a specific type of Application in the domain, based on the Application’s software profile provided to DomainManager during application installation.
ApplicationFactory Interface

Based on the Domain Profile, creates instances of software applications (waveforms) of a specific type by:

- Allocating software (Resources) to hardware (Devices)
  - Allocating capacities against Devices
- Connecting Resources that make up an Application
- Performing initial configuration on these Resources

```
<<Interface>>
ApplicationFactory

identifier : string
name : string
softwareProfile : string

create(name : in string,
       initConfiguration : in Properties,
       deviceAssignments : in DeviceAssignmentSequence) : Application
```
ApplicationFactory Interface (cont.)

User-friendly name for the application type, identifier, and the software profile (Domain Profile Software Assembly Descriptor (SAD)) used for creating applications of that type.

```
<<Interface>>
ApplicationFactory

Identifier : string
name : string
softwareProfile : string

SAD ID – Used to identify the corresponding ApplicationFactory when logging messages or sending events to the Outgoing Domain Management Event Channel.
```

```
create(name : in string,
    initConfiguration : in Properties,
    deviceAssignments : in DeviceAssignmentSequence) : Application
```

The create operation allows the user to request the creation of an application, provide it with pre-selected devices, and specify its configuration properties.
How an Application Instance is Brought into Existence

- UI asks for all installed Apps (ApplicationFactories).
- AppFactory is chosen & issued a create ()
- Determines applicable Device(s) on which to load application code defined in Domain Profile.
- Resources are then configured, initialized, and started.
- CF::Application is returned – providing proxy interface to the Assembly Controller of created Resources.
ApplicationFactory
Interface Advantages

The ApplicationFactory provides an interface to request the creation of a specific type of application in the domain.

- This abstraction allows a user interface to manipulate a group of application types as objects.

There is one ApplicationFactory object for each type of application and one implementation for all ApplicationFactory objects.

- Each ApplicationFactory object has its own instance variables based upon the same servant class definition and therefore is its own unique instance.

The ApplicationFactory interface provides a standard interface for creating applications within the domain, and one common implementation within the domain for creating applications.

- Each application developer doesn’t have to develop code to parse their own software profiles and retrieve domain profile (devices, etc.) to create their application within a domain.
CF Framework Control
Application Configuration, Control and Termination Interface
CF Application Instantiation Interfaces

...provides the means to manage an instantiated application’s lifecycle, state, and behavior.

When the ApplicationFactory instantiates an Application, it returns the proxy for its implementation which...
From an end-user perspective, software applications (services, waveforms, etc.) are the primary functional element of a system.

An Application has characteristics very similar to a Resource. For example, an application has properties and communicates through ports.

Since users need to control the lifespan of applications, configure them, and control their behavior, the Application interface is derived from the Resource interface, so that it can provide the necessary operations for managing application lifecycle, state, and behavior.

In general, the implementation of an Application is comprised of a set of Resource component implementations and their interconnections.
The Application delegates its inherited Resource operations to a Resource component that has been identified as the “assembly controller for the software assembly”.

Since the Application interface is derived from Resource and an assembly controller is also a Resource, this makes the delegation very simple and elegant.
What is an Assembly Controller?

- Acts like the POC for the CF to communicate with the Application.
- All applications are required to have one component that acts like an assembly controller.
- Identified in the Application’s Software Assembly Descriptor file in the Domain Profile.

<<Interface>> Port

connectPort()
disconnectPort()

<<Interface>> Resource

identifier : string
start()
stop()

<<use>>

<<Implementation>> AssemblyController
Assembly Controller Example

Connection of Assembly Controller and other Application Components to each other, and outside entities (e.g. UI, Device(s), other application’s via ports. More on this later.
Providing Access to an Application’s Ports

The Application’s ports provide the capability of connecting the Application up to other components such as an Application.

The ports can also be used to provide additional command/control and/or status interfaces for an Application.

The ports for an Application are flexible allowing developers to determine which of their Application’s components’ ports should be made visible via the Application’s software profile definition.
## Application Interface

<table>
<thead>
<tr>
<th>&lt;&lt;Interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
</tbody>
</table>

- `identifier : string`
- `componentDevices : CF::DeviceAssignmentSequence`
- `componentImplementations : CF::Application::ComponentElementSequence`
- `componentNamingContexts : CF::Application::ComponentElementSequence`
- `componentProcessIds : CF::Application::ComponentProcessIdSequence`
- `name : string`
- `profile : string`

**Name of the Domain Profile file (Software Assembly Descriptor (SAD)), that describes the Application’s internal structure and the characteristics of the software assembly.**
**Application Interface (cont.)**

<table>
<thead>
<tr>
<th>&lt;&lt;Interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
</tbody>
</table>

- **identifier**: string
- **componentDevices**: `CF::DeviceAssignmentSequence`
- **componentImplementations**: `CF::Application::ComponentElementSequence`
- **componentNamingContexts**: `CF::Application::ComponentElementSequence`
- **componentProcessIds**: `CF::Application::ComponentProcessIdSequence`
- **name**: string
- **profile**: string

**List of object references of all Logical Device(s) on which Application’s component’s are loaded and/or are running. Used by Application for unloading and terminating component modules and processes during Application tear-down.**
Application Interface (cont.)

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier : string</td>
</tr>
<tr>
<td>componentDevices : CF::DeviceAssignmentSequence</td>
</tr>
<tr>
<td>componentImplementations : CF::Application::ComponentElementSequence</td>
</tr>
<tr>
<td>componentNamingContexts : CF::Application::ComponentElementSequence</td>
</tr>
<tr>
<td>componentProcessIds : CF::Application::ComponentProcessIdSequence</td>
</tr>
<tr>
<td>name : string</td>
</tr>
<tr>
<td>profile : string</td>
</tr>
</tbody>
</table>

List of identifiers for chosen (running) component implementations from the Application’s software profile for each created component. More on this later during the Domain Profile lecture.
Application Interface (cont.)

<table>
<thead>
<tr>
<th><strong>Application</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier : string</td>
</tr>
<tr>
<td>componentDevices : CF::DeviceAssignmentSequence</td>
</tr>
<tr>
<td>componentImplementations : CF::Application::ComponentElementSequence</td>
</tr>
<tr>
<td>componentNamingContexts : CF::Application::ComponentElementSequence</td>
</tr>
<tr>
<td>componentProcessIds : CF::Application::ComponentProcessIdSequence</td>
</tr>
<tr>
<td>name : string</td>
</tr>
<tr>
<td>profile : string</td>
</tr>
</tbody>
</table>

**List of all Naming Service Naming Contexts to which Application components are bound.** When a component is created, it binds itself to the Naming Service under a predefined naming context defined in the Application’s software profile. During Application tear-down, the component’s object ref along with this naming context are removed from the Naming Service.
### Application Interface (cont.)

<table>
<thead>
<tr>
<th><strong>&lt;&lt;Interface&gt;&gt;</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
</tbody>
</table>

- **identifier** : string
- **componentDevices** : CF::DeviceAssignmentSequence
- **componentImplementations** : CF::Application::ComponentElementSequence
- **componentNamingContexts** : CF::Application::ComponentElementSequence
- **componentProcessIds** : CF::Application::ComponentProcessIdSequence
- **name** : string
- **profile** : string

**List of identifiers for all running processes or tasks for all Application components. Used for terminating these processes or tasks during Application tear-down.**
Application Interface
Advantages

The *Application* interface provides a standard interface for all applications within the domain, and one common implementation within the domain for all applications.

Application developers don’t have to develop code to tear down their application and to behave according to the *Application’s* software profile (SAD).

Having a generic *Application* abstraction supports general purpose user interface components for controlling a system.
CF File Service Interfaces
CF File Services Interfaces

Provides the ability to read and write files within a SCA-compliant distributed file system.

Appearing as a single FileSystem, provides access to multiple distribute file systems, whose file storage may span multiple physical file systems (AKA a federated file system).

Defines CORBA operations that enable remote access to a physical file system.

Used by CF clients for all file accesses.
File Interface

- Provides access to files within the system
- Allows access across processor boundaries (distributed FileSystems)

```<<Interface>>
File

fileName : string
filePointer : unsigned long

close() : void
read(data : out OctetSequence, length : in unsigned long) : void
setFilePointer(filePointer : in unsigned long) : void
sizeOf() : unsigned long
write(data : in OctetSequence) : void
```
FileSystem Interface

Allows creation, deletion, copying of files

<<Interface>>
FileSystem

- copy(sourceFileName : in string, destinationFileName : in string) : void
- create(fileName : in string) : File
- exists(fileName : in string) : boolean
- list(pattern : in string) : FileInformationSequence
- mkdir(directoryName : in string) : void
- open(fileName : in string, read_Only : in boolean) : File
- query(fileSystemProperties : inout Properties) : void
- remove(fileName : in string) : void
- rmdir(directoryName : in string) : void
FileManager Interface

FileManager inherits from FileSystem, as it appears as a single FileSystem to clients, even though it may be managing multiple distributed FileSystem.

FileManager uses FileSystem, as it delegates all FileSystem operations to the correct FileSystem.

<<Interface>>
FileManager

getMounts() : MountSequence
mount(mountPoint : in string, file_System : in FileSystem) : void
unmount(mountPoint : in string) : void
Infrastructure Layer
Operating Environment
Log Service
Log Service

Provides one interface (Log) and types for a log producer to generate standard SCA log records and defines types necessary to control logging output of a log producer.

Log producers are required to implement configure properties that configure the producer’s log record output.
Log Interface

<<Interface>>
Log

clearLog() : void
destroy() : void
getAdministrativeState() : AdministrativeStateType
getAvailabilityStatus() : AvailabilityStatusType
getCurrentSize() : unsigned long long
getLogFullAction() : LogFullActionType
getMaxSize() : unsigned long long
getNumRecords() : unsigned long long
getOperationalState() : OperationalStateType
getRecordIdFromTime(fromTime : in LogTimeType) : RecordIdType
retrieveById(currentId : inout RecordIdType, howMany : in unsigned long) : LogRecordSequence
setAdministrativeState(state : in AdministrativeStateType) : void
setLogFullAction(action : in LogFullActionType) : void
setMaxSize(size : in unsigned long long) : void
writeRecords(records : in ProducerLogRecordSequence) : void

Log’s Capabilities
Has Operational and Administrative states
Can control the Log’s size
Can control what the Log does when it gets full (wrap or halt)
Can write a group of Log records at a time, time-stamping each log record
Can retrieve a set of Log records at a time
Can clear a Log
Log Service Usage by CF

• Application installation/uninstallation results in CF domain
• DeviceManager/device/service registration/unregistration results in CF domain

Application instantiation results in CF domain

Application tear-down results in CF domain

Unsuccessful device/service registration/unregistration results with DeviceManager
Log Service Implementation and Deployment Optional in the OE

A CF provider may deliver an SCA-compliant product without a Log Service implementation.

An SCA-compliant installation (e.g., a handheld platform with limited resources) may choose not to deploy a Log Service as part of its domain.

CF components that are required to write log records are also required to account for the absence of a log service and otherwise operate normally.
CF Domain Profile
Domain Profile

A set of files that express the packaging and deployment of SCA-complaint components.

- Specific characteristics of software components or hardware devices.
  - Interfaces, functional capabilities, logical location, interconnections, inter-dependencies, and other pertinent properties and parameters.
  - e.g. Description of applications, startup requirements of devices, etc.
Defined in XML and Based on XML DTDs

Defined in eXtensible Markup Language (XML) format.

Based upon SCA-defined XML Document Type Definitions (DTDs).

- A DTD defines the meaning and compliant syntax for a XML file.
Based on OMG CORBA Components Specification

Based upon the OMG's draft CORBA Components specification (orbos/99-07-01).

Customized from the OMG CORBA Component Specification.

- Addition, or customization of existing elements to better address the needs of an SCA-compliant system.
- Elimination of unneeded elements in order to reduce the associated XML parsing and aid in XML file content validation.
- Advantageous, as it builds upon work already completed by OMG and does not duplicate effort.
Domain Profile Files

Software Package Descriptor (SPD)
- Describes a component’s (CORBA and non-CORBA) implementation(s).

Property File (PRF)
- Describes properties for a component.

Software Component Descriptor (SCD)
- Describes a CORBA component’s characteristics.

Software Assembly Descriptor (SAD)
- Describes an application’s deployment characteristics.

Device Configuration Descriptor (DCD)
- Describes configuration characteristics for a DeviceManager.
Domain Profile Files (cont.)

DomainManager Configuration Descriptor (DMD)
- Describes configuration characteristics for a DomainManager.

Device Package Descriptor (DPD)
- Identifies a class of hardware device and its characteristics.

Profile Descriptor
- Describes a type of file (SAD, SPD, DCD, DMD) along with the file name.
0 or more DeviceManagers, deploying persistent components to the CF domain

A component’s (CORBA and non-CORBA) implementations

A hardware device class & its characteristics

0 or more ApplicationFactories, representing installed, to be instantiated, or already running applications in the domain

A CORBA component’s characteristics (e.g. “use” & “provides” ports.

Both DCD and SAD are assembly types that deploy component implementations.
Software Package Descriptor (SPD)

Based on CORBA Components Software Package Descriptor.

Used to describe any software component.
  - CORBA or non-CORBA.

An application’s software profile (Software Assembly Descriptor), contains one SPD for each software component of that application.
A Software package may contain multiple implementations (for different hardware) using the same properties and Software Component Descriptor (SCD).

Local file name of the SCD file used to describe interface information for the component being delivered to the system.

Any “uses” relationship this component has with a device in the system. Will contain all allocation properties for that device, indicating capacities needed from that Device by the software component.
Local file name of the load module, with a possible entry point task name, for this specific implementation of the software component.

Module’s Load Type
- “Executable”: load & execute
- “KernelModule” & “Driver”: load only
- “SharedLibrary”: load only, or load an execute depending on existence of a code entry point

SPD implementation Element

Local file name of the load module, with a possible entry point task name, for this specific implementation of the software component.

Module’s Load Type
- “Executable”: load & execute
- “KernelModule” & “Driver”: load only
- “SharedLibrary”: load only, or load an execute depending on existence of a code entry point
Properties Descriptor

Derived from the CORBA Components properties element.

Contains information about properties applicable to a software package or a device package.

It is a sequence of 1 or more properties, where...

Each property is of a particular property type and property kind.
Properties Descriptor

Property Types

- simple
- simplesequence
- struct
- structsequence

<<DTDElement>> properties

<<DTDSimpleGroup>>

<<DTDElementPCDATA>> description

<<DTDChoiceGroup>>

<<DTDSimpleGroup>> properties_grp_grp (from properties)

<<DTDSimpleGroup>> properties_grp (from properties)

0..1
1..n

Properties:
- Frequency, Power Level
- Scan Frequency List
- Commanded Level BIT
- Date Structure
- Preset Data
## Properties Descriptor

### Property Kinds

- **Capacity type properties for logical Devices**
  - Configuration or query properties for a component
  - Test properties for a component
  - User defined execute parameters for main programs

### XML Code

```xml
<DTDElementPCDATA>
  <description>
    Capacity type properties for logical Devices
  </description>
</DTDElementPCDATA>
```

### Diagram

![Diagram showing the relationship between different property kinds and their attributes.](image-url)

- **Simple Properties**
  - id : ID
  - type : (boolean | char | double | float | short | long | objref | octet | string | ulong | ushort)
  - name : CDATA
  - mode : (readonly | readwrite | writeonly) = readwrite

- **Simple Group Properties**
  - kindtype : (allocation | configure | test | execparam | factoryparam) = configure

- **ResourceFactory Create Options Parameters**
  - Value
  - Range
    - min : CDATA
    - max : CDATA

- **Enumerations**
  - type : (eq | ne | gt | lt | ge | le | external) = external
Software Component Descriptor (SCD)

Based on the CORBA Component Descriptor (CCD) file definition.
- interfaces, uses port, provides port, supporting interfaces.

Provides the means for a component to describe its interfaces in two ways.
- Supporting interfaces which are IDL interfaces inherited by the component IDL interface.
- Provides interfaces or “facets” - Distinct CORBA interfaces that a component implementation supports in addition to the component’s interface.
SCD Elements

The type of software component object, and hence, its characteristics (e.g. resource, device, resourcefactory, domainmanager, log, filesystem, filemanager, namingservice, eventservice.

The supported message ports for the component.

List of all interfaces that the component, either directly or through inheritance, provides, uses, or supports.
SCD Supported Message Ports Element

Identifies an IDL interface supported by the component.

Refers to the supported interface in the SCD “interfaces” element.

Interfaces “provided” and/or “used” by the component.
SCD Ports Element

A “Facet” – An independent interface from the component’s main interface, providing a specialized connectivity to the component.

Each “uses” and “provides” port references its corresponding interface (i.e. the corresponding SCD interface element) by its id.

A CF Port interface type that is to be connected to a “provides” or “supportinterfaces” interface.

Each “uses” and “provides” port references its corresponding interface (i.e. the corresponding SCD interface element) by its id.
Device Package Descriptor (DPD)

Not derived from the CORBA Component Specification.

Identifies a class of a hardware device along with its properties.

- Typically used by an HCI application to display information about the device(s) resident on an SCA-compliant system.

Based on the SCA Hardware Architecture definition.
DPD Elements

<<DTDElement>>
devicepkg

- id : ID
- name : CDATA
- version : CDATA

<<DTDElement>>
hwdeviceregistration

- id : ID
- name : CDATA
- version : CDATA

<<DTDElementPCDATA>>
title

<<DTDElement>>
author

<<DTDElementPCDATA>>
description

<<DTDElementPCDATA>>
devicepkg_grp
(from devicepkg)

{4}

0..1

1..n

0..1

{3}
DPD Elements (cont.)

Device identification information, including the device name, device class, manufacture, and model number.
The DPD allows for composition device definitions to be formed up completely in one file …

… or in multiple files (DPD can reference other DPDs).
Software Assembly Descriptor (SAD)

Derived from the CORBA Component Assembly Descriptor (CAD) file definition.

Describes what makes up an application, how to create the application’s pieces, configure them, and interconnect them.

Processed by $CF::ApplicationFactory$ during application instantiation.
SAD Contents

Based on the CORBA Components Assembly Descriptor concepts of component files, partitioning, and connections, it describes:

- What component instances makeup an assembly.
- Which components are located together.
- How to find components (e.g., naming service).
- How to connect components together.
SAD Contents (cont.)

The required identification of a component as the main assembly controller.
- Simplifies the behavior of the CF Application that acts a proxy for the assembly controller.
- By having a mandated assembly controller, the Application proxy simply delegates its Resource operations to the assembly controller Resource.

The optional visible ports for a software assembly.
- Allows the assembly to be treated like a component that can be connected to for its data ports, or additional status and command/control ports.
SAD Top-Level Elements

The local file names of the SPDs of the 1..n software components that make up the software assembly.

The deployment pattern of the SAD components and their components-to-hosts relationships.

The main CF Resource for controlling the Assembly. The corresponding instantiated CF::Application delegates its Resource configure, query, start, stop, and runTest operations to this Assembly Controller.

Provides the connection map between the sw components in the assembly.

Assembly’s visible Ports. Returned by CF::Application getPort().
SAD Component Deployment Pattern

Comprised of a grouping of one or more componentplacement elements, describing a group of component instances that are to be deployed together (i.e collocated) on a single host.

... different ways of instantiating the same component. What different ways? Why do this?

Defines one or more deployments of a component. Meaning…
SAD Component Deployment Pattern (cont.)

Optional list of
• Component-specific execute parameter properties used for component creation.
• ResourceFactory parameter properties used for creating the component via a ResourceFactory.
• Configuration properties used for initial configuration of the component

Optional means of finding the component after it has been created:
• Via a ResourceFactory, if it has been created by one.
• Via the Naming Service, if the component is directed to register itself with the Naming Service upon creation.

Diagram:
- `<DTDElement>` componentinstantiation
  - `<DTDElement>` usagename
  - `<DTDElement>` componentproperties
  - `<DTDElement>` findcomponent
  - `<DTDElement>` id : ID
  - `<DTDElement>` componentinstantiation grp (from componentinstantiation)

Notations:
- 0..1
- {1}
- {2}
- {3}
Device Configuration Descriptor (DCD)

Derived from the Software Assembly Descriptor file definition.

Processed by each DeviceManager and by DomainManager upon DeviceManager registration with CF domain.

Provides a generic means of deploying persistent components (logical Device(s) and services) to the CF domain during each node (DeviceManager) startup.

Allows a System Administrator flexibility to decide what nodes within the system services should be deployed on and how many services should be within a system.
DCD Elements

DeviceManager SPD.
All components (logical Devices, services, others) to be started by the DeviceManager on the node represented by DeviceManager. Similar to SAD.

Connections of DeviceManager and/or DCD Logical Devices to domain services. Similar to SAD. Processed by DomainManager upon DeviceManager registration.

How the DeviceManager can obtain the DomainManager object reference.

Deployment pattern of DCD components and their component-to-host relationships. Similar to SAD.

Mounted FileSystem names for the DeviceManager’s FileManager.
The DCD provides the flexibility of deploying a DCD component either on a pre-determined General Purpose Processor Device (in the absence of a deployondevice element), or on a DCD-specific Logical Device that is yet to be deployed by DeviceManager, as specified by the deployondevice element.

A DCD Logical Device may be aggregated by another DCD Logical Device that has or has not yet been deployed by DeviceManager. The aggregated device may be deployed only after its parent device has been deployed, as it needs its parent device object reference.
DomainManager Configuration Descriptor

- Not based on CORBA Component Specification.
- Describes the DomainManager’s SPD and services to be connected to DomainManager for DomainManagement functions (DomainManager, ApplicationFactory, Application) use.
- No other concept of domain management.
A software application is an "assembly" of 1..n software components. Core Framework objects responsible for installing, starting up and tearing down applications.

**DomainManager**
- 1
- 0..n

**ApplicationFactory**
- 1
- 0..n

**Application**
- Described by 1

**Properties**
- Described by 1

**Properties Descriptor**
- 1

**SW Component**
- Described by 1

**SW Assembly Descriptor**
- 1

**SW Package Descriptor**
- 1

**Non-CORBA Component**
- Described by 1

**CORBA Component**
- Described by 1

**Device Package Descriptor**
- 1

**Device**
- 1

**Resource**
- 1

**ResourceFactory**
- Implements CF Device, Resource &

**Consumer**
- 1

**Uses Port**
- 0..n

**Producer**
- 1

**Provides Port**
- 0..n

**Consumer**
- Used to access a Provides Port of a Producer

**Properties**
- Described by a "connectinterface" element within the SAD
General Design Considerations for Development of SCA-Compliant Components
Some Special Terms

The following terms have multiple meanings – depending on context and capitalization.

- **Application** versus **application**
  - *CF::Application* Interface versus an application running on CF

- **Port** versus **port**
  - *CF::Port* interface versus using & provides ports

- **Resource** versus **resource**
  - *CF::Resource* interface versus a specific implementation of *CF::Resource* running on CF

- **Device** versus **device**
  - *CF::Device* interface (logical) to device (physical)
1. Creation of Specific Components by Specialization or Implementation of CF Base Application and Device Interfaces
Specialization of the CF Base Application and Device Interfaces

Component-based design inherits from one the following base interfaces and as needed for that specific implementation, provides the component-specific implementation, adhering to the specified CF requirements:

- **CF::Resource**
  - Creation/tear-down
  - Initialization/configuration
  - Start/stop

- **CF::Device**
  - **CF::LoadableDevice**
  - **CF::ExecutableDevice**
  - **CF::AggregateDevice**
    - Allocating/deallocating capacities based on a device’s capacity model
    - Loading/unloading modules onto a device processor’s memory
    - Executing/terminating software processes/threads
2. Connecting to Other Components Via Ports
Component A is connected to Component B. Component A’s “uses” port is connected to Component B’s “provides” port. Component A will have to call a method in Component B.
Connecting Components

Interface Relationships

Component A <<Uses>> Component B

<<Interface>> Resource
getPort()

<<Interface>> Port
connectPort()

<<Interface>> A

<<Interface>> B

<<inherits>>

<<interface>> Resource
getPort()

<<interface>> Port
connectPort()

<<interface>> B
methodB()

<<inherits>>

<<use>>

Interface A is connected to Interface B.
Interface A will call a method in Interface B.
Connecting Components

Connection of Classes

Class A is connected to Class B. Class A will call a method in Class B.
3. When does a Component Need a Port Interface?
When does a component need a Port Interface?

When a component has a “using” relationship to another component.
4. Port Interface
Inheritance versus Aggregation
If *using* only one component, then inherit from *Port* interface.

```
<<Interface>>
Resource
getPort()

<<Interface>>
Port
connectPort()

<<realizes>>
Class A

<<realizes>>
```
Port Interface Aggregation

If *using* more than one component, then use an aggregation of *Port* interfaces (with a cardinality of 1..n).

```
<<Interface>>
Resource

getPort()

<<realizes>>

<<Impl>>
Class B

1..n

<<impl>>
Class B Use Port

connectPort()

<<realizes>>
```
5. Port “Using” Relationship Design Consideration
Intermediate Interface Shared between Components

Does the component use the same type of operation on multiple components?
- If so, define an intermediate interface that is shared between the components.
- Benefit: Reduces coupling to other components in the system.
6. When do You Create a New Base Application or Base Device Interface?
Do You Need to Create a New Interface?

Only if you are extending the base interface.
- New interfaces adds coupling to a system.

If you are not extending `CF::Resource` or `CF::Device`, you do not need to create a new interface.
- `XYZ Class` realizes the `Resource` and `PushPacket` interfaces.
- `XYZ Class` has multiple “using” ports that realize the `Port` interface and uses a `PushPacket` interface.
How Does CF Interact with Application Components?
Some Examples of...

Installing an application
Retrieving the list of installed applications
Creating an application
  - Creating a single application component
  - Connecting an application’s components together
Retrieving the list of running applications
Communicating with one of those running applications
An application installation request is sent to the DomainManager.

Add the application's S/W Profile into the Domain Profile.

Construct an ApplicationFactory for the specified application software profile.

The DomainManager sends an event to the ODM Event Channel to indicate the installation of the application.

Log the result of application installation.
Listing Installed Applications

A request for getting a list of installed applications is sent to DomainManager.

1: applicationFactories()
2: List of installed Applications (created ApplicationFactories)
**Application Creation**

An application creation request is sent to ApplicationFactory. After evaluating the application’s SW profile, the ApplicationFactory allocates any required capacities on all applicable devices.

Loads all required SW on all applicable devices.

Creates all Resource Factories and Resources on all applicable devices.

Commands all Resource Factories to create their corresponding resources.

Initializes all app’s resources.

Establishes all port connections for all app’s resources as defined by the app’s SW profile.

Configures all app’s resources as specified by the app’s SW profile.

Generates an event to indicate the application has been created.

Logs the app creation result.

For a user to start an application, they must be aware of the types of applications available in the system and then be able to command the creation of a new instance of a selected application type. User Interface asks (DomainManager) for all installed applications, and chooses the corresponding ApplicationFactory.

The ApplicationFactory retrieves object references of all Resource Factories from the Naming Service.

The ApplicationFactory retrieves object references of all Resources it directly creates from the Naming Service. Object references of ResourceFactory-created Resources are returned by the ResourceFactory createResource operation.

Finally, a CF Application is returned, providing proxy interface to Assembly Controller of created Resources.
Launching a Single Application Component

1. load (B.out)
2. execute(EntryPointTask, executeParams)
3. spawn(EntryPointTask, executeParams)
4. bind()
5. resolve()
6. initialize()/getPort()/connectPort()/configure()
7. <<create>>
Use of Configuration and Execute Parameters during Application Component creation

Configuration and Execute properties and their default values are defined in a Properties Descriptor, located in a Property File of the Application’s Software Assembly Descriptor (SAD) in the Domain Profile.

CF will modify those properties based on the following precedence order:
- ApplicationFactory *create* method (platform specific overrides)
- SAD file (application specific overrides)
- SPD file (different implementation overrides)
- SPD Property file
- SCD Property file
Connecting Application Components

Application Component A

Uses the service

4. requestService()

<<create>>

2. getPort()

3. connectPort()

Application Factory

Return Created Application

create(initConfiguration, deviceAssignments)

Domain Profile

Application Component B

Provides a service

<<create>>

1. getPort()
A request for list of running applications is sent to the DomainManager.

1: applications()

2: List of running applications
   (CF::Applications instantiated by installed ApplicationFactories)
Invoking Operations on a Single Application Component

1: configure(), query(), start(), stop(), runTest()

2: configure(), query(), start(), stop(), runTest()

3: configure(), query(), start(), stop(), runTest()

1) initialize() is implemented as a null operation at the Application object and not forwarded to an Assembly Controller.
2) releaseObject() means application tear down and is directed to each component in the application.
3) getPort() returns an application port that was identified by the application’s SAD file as being “external”.
Application Teardown

1: releaseObject()
2: disconnectPort()
3: releaseResource()
4: shutdown()
5: releaseObject()
6: terminate()
7: unload()
8: unload()
9: deallocateCapacity()
10: deallocateCapacity()
11: unbind()
12: writeRecords()
13: push
SCA IDL Modules
SCA IDL Modules

- **LogService**
  - Contains the interfaces and types for a log service.

- **PortTypes**
  - Contains sequences of CORBA basic types for optional Port operations.

- **CosEventComm**
  - Contains the interfaces and types for an event service with SCA-defined event types.

- **StandardEvent**
  - Contains Standard event types for Domain Management.

- **CF**
  - Contains all required CF interfaces and types.

- **CosNaming**
  - Contains the interfaces and types for a lightweight naming service.
CF IDL Module
The Log interface, types for a log producer to generate standard SCA log records, and types necessary to control the logging output of a log producer.
StandardEvent
IDL Module

Types necessary for an event producer to generate standard SCA events.

<<CORBAEnum>>
StateChangeCategoryType
ADMINISTRATIVE_STATE_EVENT
OPERATIONAL_STATE_EVENT
USAGE_STATE_EVENT

<<CORBAEnum>>
StateChangeType
LOCKED
UNLOCKED
SHUTTING_DOWN
ENABLED
DISABLED
IDLE
ACTIVE
BUSY

<<CORBAEnum>>
SourceCategoryType
DEVICE_MANAGER
DEVICE
APPLICATION_FACTORY
APPLICATION
SERVICE

<<CORBAStruct>>
DomainManagementObjectAddedEventType
producerId : string
sourceId : string
sourceName : string
sourceIOR : Object

<<CORBAStruct>>
DomainManagementObjectRemovedEventType
producerId : string
sourceId : string
sourceName : string
PortTypes
IDL Module

Sequences of base CORBA types for optional Port operations

PortTypes

<<CORBATypedef>> CharSequence
<<CORBATypedef>> WcharSequence
<<CORBATypedef>> WstringSequence

<<CORBATypedef>> ShortSequence
<<CORBATypedef>> UshortSequence
<<CORBATypedef>> FloatSequence

<<CORBATypedef>> LongSequence
<<CORBATypedef>> LongLongSequence
<<CORBATypedef>> UlongSequence

<<CORBATypedef>> DoubleSequence
<<CORBATypedef>> LongDoubleSequence
<<CORBATypedef>> UlongLongSequence

<<CORBATypedef>> BooleanSequence
References
References

http://jtrs.army.mil/
- Joint Tactical Radio System Program Background.
- Software Communications Architecture Overview, Goals, and Standardization.
- *Software Communications Architecture Support and Rationale Document*, V2.2.
- Software Communications Architecture Training Course.