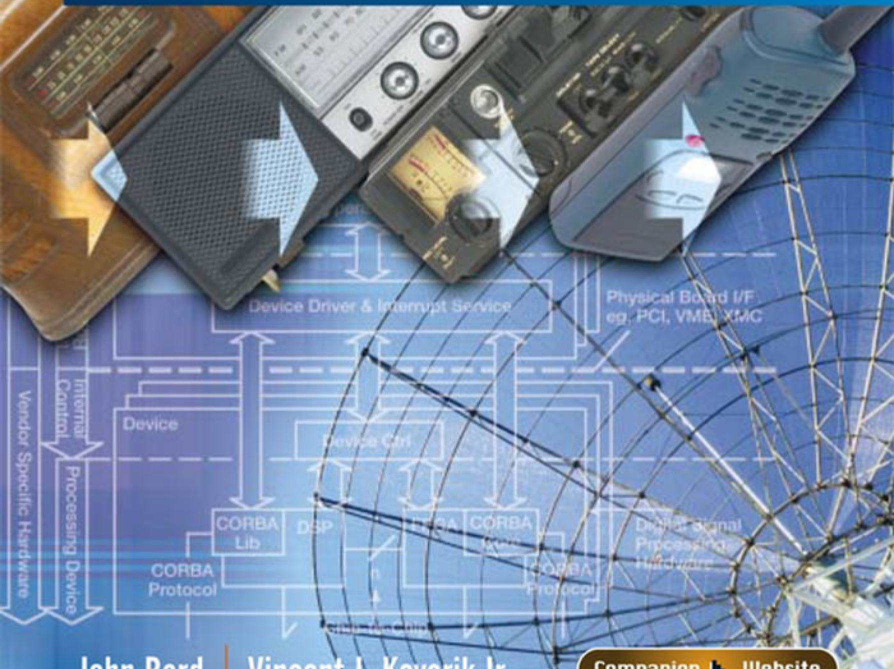
WILEY

software defined radio

The Software Communications Architecture



John Bard | Vincent J. Kovarik Jr.

Companion Website

Software Defined Radio
The Software Communications Architecture

John Bard, Space Coast Communication Systems Inc., USA
Vincent J. Kovarik Jr., Harris Corporation, USA

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Software Defined Radio

Software Defined Radio
The Software Communications Architecture

John Bard, Space Coast Communication Systems Inc., USA
Vincent J. Kovarik Jr., Harris Corporation, USA

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

John Wiley & Sons, Ltd

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont, Blvd, Mississauga, ONT, L5R 4J3

Anniversary Logo Design: Richard J. Pacifico

Library of Congress Cataloging in Publication Data

Bard, John.
Software defined radio : the software communications architecture / John Bard, Vincent J. Kovarik Jr.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-470-86518-7 (cloth : alk. paper)
ISBN-10: 0-470-86518-0 (cloth : alk. paper)
1. Software radio. I. Kovarik, Vincent J. II. Title. III. Title: Software communications architecture.
TK5103.4875.B37 2007
621.384—dc22

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-470-86518-7 (HB)

Typeset in 10/12pt Times by Integra Software Services Pvt. Ltd, Pondicherry, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Disclaimer

Acknowledgments

Indeed, this has been a wonderful journey for Vince and I, but it hasn't been nearly as pleasant for our loved ones. To this end, I dearly want to thank Pam, Amela, and Amanda for their long suffering and understanding. They stopped believing me when I said 'I'm almost done', but now, for sure, I'm done.

I am grateful to Dr. Sam Aslam-Mir for providing source material used in Chapter 16. His knowledge and pre-eminence in the CORBA community has added integrity and depth to an otherwise hackers approach to CORBA.

John Bard

First, I would like to thank John for the invitation to collaborate on this book. I would also like to thank Eric Held for providing valuable comments and Lance Starr for his support in developing the Domain Management ToolKit (dmTK$^{®}$), an implementation of the SCA. Finally, a sincere thank you to my wife Madeline for her support and understanding throughout this process.

Vincent J. Kovarik, Jr.

Foreword

When first approached by Wiley a few years back to write a book on software radio, I immediately knew that the topic could not be condensed to fit within a single volume – thus was conceived the Wiley book series on Software Defined Radio.

The early volumes addressed the emergence, concepts and international activities, and the technological foundations – radio, baseband and software. These were followed by a volume describing European research on software radio architectures and systems, from the mobile telecom industry perspective, and then one addressing the specific application of software radio baseband technologies to the emerging 3G marketplace.

This latest volume, by John Bard and Vince Kovarik, represents an essential and in many ways a long overdue element of the series, addressing as it does the Software Communications Architecture, or SCA, which lies at the heart of the world's largest software radio project, the US JTRS programme and which had its own origins a decade or more ago.

The SCA is a term that many have heard, but few truly understand. John Bard and Vince Kovarik most definitely do not fit that description – indeed, they have spent many years now not simply developing their own understanding and contributing to the development of the SCA, but supporting others in applying the SCA to real world implementations. Thus, this volume is very much a book by practitioners, for practitioners. The book is effectively structured into three major sections – the first addressing the operating environment, the second focusing on the domain profile and finally a substantial set of chapters on building an SCA-compliant system.

Whilst originating within the context of the JTRS programme and the defence industry, the SCA has potential applicability beyond – this is an area which has yet to be commercially explored. Existing SCA practitioners have to date been so busy and focused on the defence requirements and contracts, that few experts have as yet had time and opportunity to begin to apply it in the commercial domain. Perhaps availability of this volume may contribute to the opening of such doors.

I would like to conclude this foreword by congratulating John and Vince on this epic work. When I first proposed to John that he consider writing a volume on the SCA he wisely said he would think about it and then, very quickly, proposed Vince as his co-author. At that time he knew, far better than I, the true scale of the task, one that has not been made any easier by the way in which the SCA has continued to dynamically evolve. That John and Vince have chosen to make the time to write this book – from personal experience I know that one doesn't *find* time for such things – is a service to the industry for which many will be grateful; their book will serve as a foundation for many engineers in the coming years. Thank you John & Vince – well done.

Dr. Walter Tuttlebee
Chief Executive, Mobile VCE

Preface

Over the past decade radio system design has seen an inexorable march towards more of the waveform signal processing being performed digitally. As Moore's law continues to push the capabilities of the General Purpose Processor (GPP), the processing power of the Digital Signal Processor (DSP), and the Field Programmable Gate Array (FPGA), this trend will continue to accelerate along with the power of these devices. The natural consequence of this trend is that more of the radio signal processing is being performed by software.

Although the use of software to perform more of the core radio functions has increased dramatically, each radio manufacturer developed solutions that differed in their architecture and implementation. Therefore, radio systems became more flexible as more capabilities were provided via software, and each implementation was unique. So, although the characteristics of the radio system could be changed through software, there was still little commonality in the control structure and management architecture across radio systems.

This lack of common management and control architecture was a significant problem in the military and public safety sectors. In these sectors, special-purpose radios were the norm rather than the exception. These radios were typically limited to a small set of capabilities or waveforms (or sometimes just one). Also, a significant number of legacy radios were hardware-based and, consequently, could not be re-configured without physical modifications or re-design. Those radios that were software-based were closed to software implemented by sources other than the original manufacturer. Compounding the problem was the fact that the radio system could not be managed, configured, or controlled using a consistent set of interfaces and protocols. So, when multiple radio sets from different manufacturers came together in response to some coordinated exercise, military operation, or disaster, the radios did not interoperate and could not be easily reconfigured to do so.

To address this problem, the United States government initiated a series of programs leading towards the specification of a common software infrastructure for software defined radios. The initiative started in the mid-1990s and evolved into the Software Communications Architecture (SCA). Although there have been prior radio infrastructures and architecture, the SCA is the first such specification that represents the combined contributions of many of the key radio system manufacturers for the United States government.

Audience

This book focuses on the SCA architecture; the use, issues, and benefits associated with developing a radio system in compliance with the specification. The book is intended to provide practical information on building an SCA-compliant system along with historical and conceptual background information to help fill in the gaps between the intent of the specification and the practice.

This book does not provide instruction on how to construct an implementation of the SCA specification, i.e. a Core Framework. Rather, it is intended to provide guidance on how to use a Core Framework implementation to build a software radio system, provide an SCA-compliant hardware component for a radio system, or deliver a waveform for use within an SCA radio system.

Consequently, this book is intended for anyone who has the need to design, develop, support, or understand hardware or software in an SCA-compliant software radio system. It provides information, background data, and interpretations of requirements presenting design tradeoffs and decisions that must be addressed in fielding an SCA-compliant system.

Scope

As with any technical book that addresses a dynamic and changing technology, certain decisions and tradeoffs were made regarding the scope. At the time this project was initiated, SCA version 2.2 was the current version of the standard and was chosen as the baseline for this book. SCA 2.2 continues to be the SCA version that is the contractual baseline for a number of on-going contracts.

However, the SCA has continued to evolve. In April 2004, SCA 2.2.1 was released. SCA 2.2.1 provided several clarifications and corrections to issues identified in SCA 2.2. Then, in August 2004, SCA 3.0 was released. SCA 3.0 was the first SCA release to attempt to address issues related to waveform component construction and portability in the realm of DSPs and FPGAs. These processors, although an integral part of a software radio, introduce coding, transport, and interface requirements not addresses in prior SCA specifications. Upon its release, however, it was apparent that SCA 3.0 had a number of shortcomings. Primarily, although it began to address the issues noted above, SCA 3.0 did not address portability issues related to DSP and FPGA processors substantively enough to provide a workable set of requirements and guidance for these processors. In the area of Core Framework functionality, there was little change to the baseline requirements.

As the final version of this book was being prepared for the publisher, version of SCA 2.2.2 was released. SCA 2.2.2 further refined the specification, corrected errors and omissions, and provided clarifications to the specification. Where possible, notes and information regarding key aspects of SCA 2.2.1 and 2.2.2 have been incorporated into this book.

Conventions Used in This Book

The following typography and terminology are used throughout the book to help the reader.

Source Code

Source code is presented using Courier New font, as shown below.

```
int add1( int number) {
    return number++;
}
```

Terminology

In this book, the term **SCA-compliant** will be used to denote that a part or the whole system adheres to the requirements specified in the SCA specification. Hence, it is compliant with the specification. The term **SCA-certified** or just **certified** is used only when referring to a system or component that has passed the suite of tests performed by JTRS Test Application (JTAP) tool.

Unified Modeline Language (UML)

Diagrams of software architecture and design are drawn using the UML graphical modeling language developed by the Object Management Group (OMG). Information on the UML standard can be found at http://www.uml.org/.

Organization

The book is divided into three parts. Part I focuses on the Operating Environment (OE). The OE forms the common software infrastructure for implementing a software defined radio that is SCA-compliant. General background on software defined radio systems, the origins and history of the SCA evolution, and organization of the SCA specification and components are provided. The SCA is discussed from the perspective of a hardware supplier, waveform developer, radio system integrator, or other individual or corporation that must utilize and adhere to the SCA requirements. Topics include the OE components, the requirements for each component, a discussion of the essential concepts or rationale behind the component along with discussion of any pros and cons, and a section on the certification process.

Part II discusses the Domain Profile. The Domain Profile describes the hardware and software that form an SCA-compliant radio system and the applications that are hosted on the system. A set of eXtensible Markup Language (XML) files are used as a platform-neutral language to describe the hardware and waveform application components, their composition, and the underlying SCA-compliant platform.

Part III provides code examples that apply and use the capabilities of the SCA as described in the first two parts. The first offering is an in-depth view of the SCA Application Environment Profile (AEP) including code examples for the use of POSIX threads. This is followed by an examination of the OMG's CORBA specification. Finally, a set of straightforward examples are presented with explanation and commentary to help understand and visualize the content presented in the first two parts of the book.

Additional Material and Author Contact

Additional reference material may be found at the website for the book hosted by the publisher. For more details, please visit the URL http://wiley.com/go/bard.

Although every effort has been made to ensure the accuracy of the content of this book, there may be errors. If you happen to find any errors or simply wish to provide comments and suggestions, you can contact the authors at vkovarik@acm.org and jbard@spacecoastcomm.com.

PART I

The Operating Environment (OE)

Vincent J. Kovarik Jr.

In Part I, the Software Communications Architecture (SCA) specification is presented. The aim of this part is to provide an overview of the specification with specific discussions related to the interpretation and functional behavior of the SCA requirements. The intent is that this part can be utilized as a self-contained reference document for the SCA specification, with annotations and background information. As noted in the Preface, the objective of this book is to provide information on the understanding and use of an SCA Core Framework.

1 Introduction

The fundamental objective of the SCA is to provide a common software infrastructure for managing radio systems. Although software comprises a significant part of most recent radios – thus enabling new capabilities and functions to be added to the radio at some future times – the software is loaded and controlled through proprietary mechanisms and each radio manufacturer typically employs a unique infrastructure or architecture. A software defined radio, as interpreted here, refers to a class of radios, the capabilities of which are not simply provided by software but utilize an infrastructure that supports interchangeable components as well as functionality.

This chapter provide background information regarding the SCA. The SCA specification desribes a collection of components, the configuration of the components, and the assembly of the components into a functional waveform application on a radio system. Taken together, these form an infrastructure for defining and constructing a software defined radio system.

1.1 Software Radios

Figure 1.1 illustrates the abstraction space of bandwidth versus waveform abstraction. At the lowest level is a set of hardware that provides the actual processing of the waveform and support software. The processing is provided by one of four options, General Purpose Processor (GPP), Digital Signal Processor (DSP), Field Programmable Gate Array (FPGA), and Application Specific Integrated Circuit (ASIC). The ASIC is typically not considered part of the solution set within a software radio because, once programmed, it cannot be modified after deployment – one of the fundamental tenets of a software radio.[1]

The aim of Figure 1.1 is to illustrate the two orthogonal perspectives of software radio design. The waveform design starts as a set of requirements, simulation, mathematical

[1] *This does not necessarily imply that an ASIC cannot be applied within a software defined radio. It is the authors' opinion that, given certain circumstances and architectural approaches, it is feasible to integrate ASICs within the design of a software defined radio. Such a design would need to support the ability to interchange flexibly functional components of the waveform processing implemented in ASICs. This would require some mechanism such as the ability to call algorithms implemented on an ASIC as though it were a function call.*

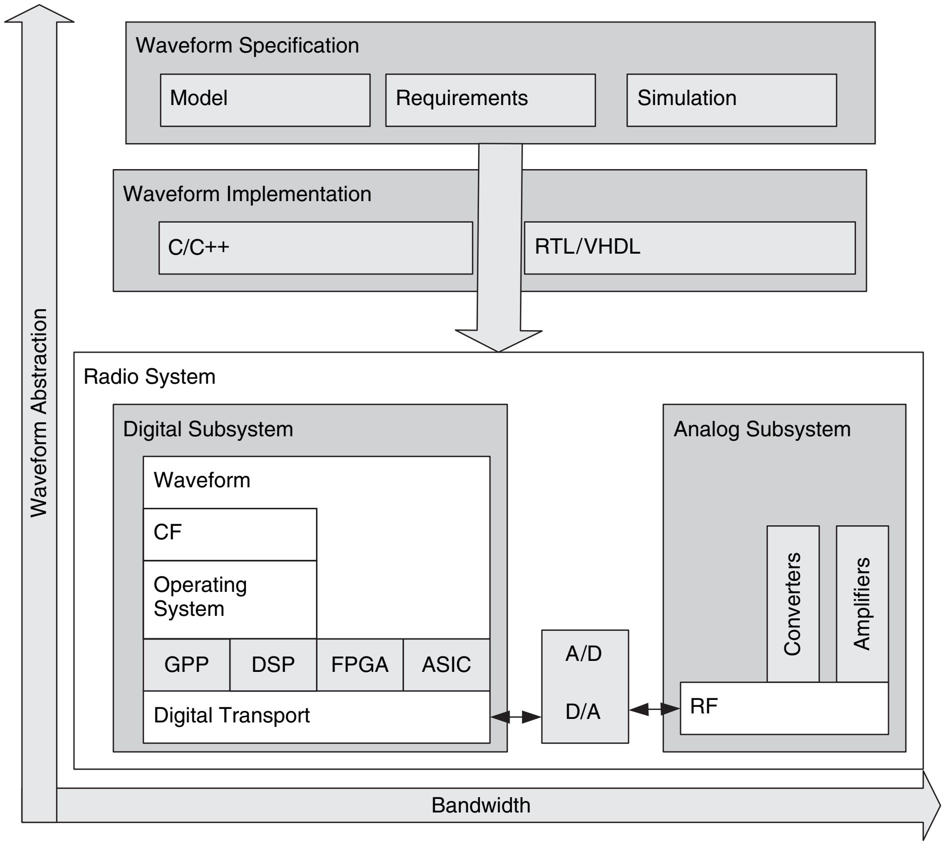


Figure 1.1. Waveform abstraction relative to bandwidth

model, or some other conceptual representation. As the waveform progresses from design to implementation, the capabilities of the waveform, in terms of throughput and capacity, typically drive the implementation to a high-level language for deployment on a GPP or DSP. Higher throughput demands drive the deployment towards and FPGA or an ASIC.

The GPP processor typically provides the management and control services for the system. Overlaid on top of the processor is an operating system and, integrated with the operating system, is a collection of software that provides the run-time infrastructure for the radio set. The infrastructure, in SCA terms, is called the Core Framework. On top of the Core Framework sits the waveform and other applications.

1.1.1 Software Radio Aspects

A software radio system can be viewed through one of four perspectives or aspects. Each aspect forms a functional grouping of objects and services provided by the radio system. Illustrated in Figure 1.2, these aspects are:

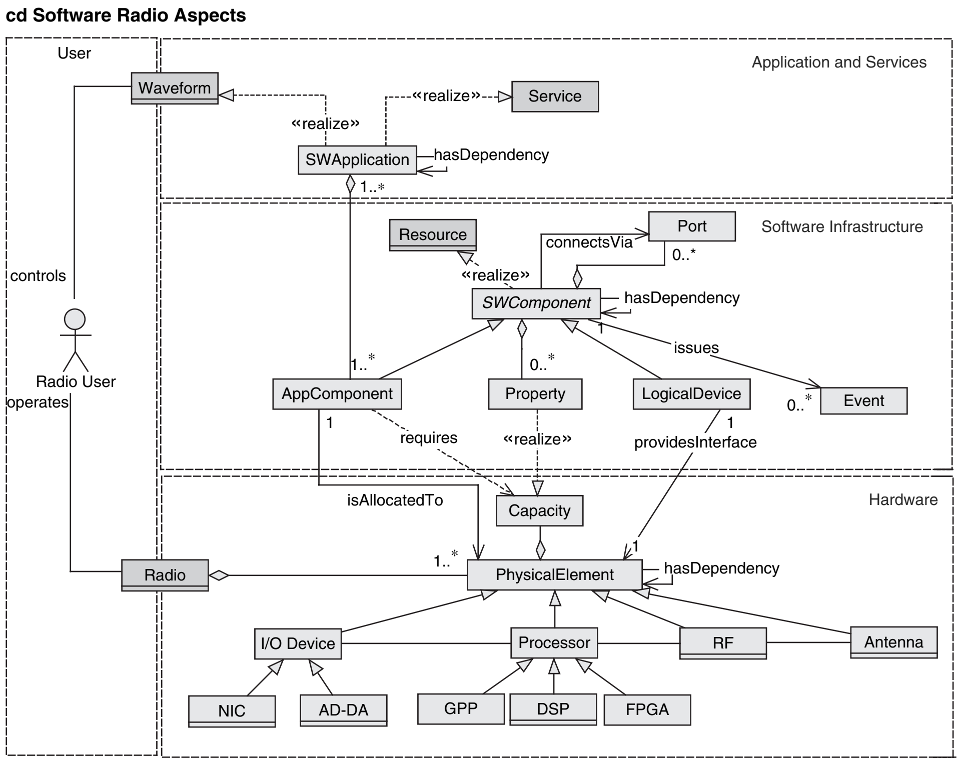


Figure 1.2. Software defined radio aspects

- **Hardware** – This aspect describes the physical set of devices and components that comprise the radio set.
- **Software** – This aspect defines the set of services and interfaces through which all waveform applications must interface to the underlying hardware.
- **Application** – This aspect defines the application and service layer. All waveforms and common services execute in this aspect.
- **User** – This aspect is the view through which the user interacts with the radio set. There are two basic modes of interaction within this aspect. The user is either performing radio control operations, e.g. setting system parameters, or performing application control and data transfer, e.g. setting the gain parameter for a specific waveform instance.

The SCA can be viewed as one realization of the Software Infrastructure aspect with some parts within the Applications and Services aspect. It defines a logical infrastructure for management and abstraction of physical hardware components, a standard set of abstractions for software components that form the digital processing portion of a waveform implementation, general services available for use by the system, and a set of common interfaces for managing, deploying, and configuring waveform applications within the system.

1.2 The Software Communications Architecture

Any new concept or technology has a learning curve associated with it and the SCA is no exception. The SCA defines a software infrastructure for the management, control, and configuration of a software defined radio. It does not mandate any specific architecture, design, or implementation for the radio system hardware or waveform application. Before launching into the detailed discussion of the SCA, it is advisable to spend a short bit of time providing some background data and explanation on what the SCA is, and is not, the history of its evolution, and the reasons why you would (or would not) want to apply the SCA to your system.

The SCA is based on several related technologies: Object-Oriented (OO) techniques in software engineering, the Common Object Request Broker Architecture (CORBA), and the CORBA Components Model (CCM). Object-oriented languages have been around for a number of years from Simula in the late 1960s, Smalltalk and Flavors in the early 1980s, to current object-oriented languages such as C++, Python, Ruby, and Java, to name a few.

As systems evolved towards distributed architectures and a client-server model, CORBA evolved as an industry standard for describing the interfaces provided or used by two components using a pseudo-code called an Interface Definition Language (IDL). IDL provided the means for specifying the available interfaces and, through the IDL 'compiler', generated source code that is compiled into each of the applications. The code generated includes the support routines necessary to support remote procedure calls between processes on the same computer and between computers, i.e. in a distributed environment. Thus, the developer was freed from the drudgery of writing low-level, inter-process communications code and, more importantly, CORBA code built by one individual could interoperate with code built by another individual, the only requirement being that both the author of the client application and the server application use the same IDL. This was an important step forward in the ability to develop modular software while encapsulating the internal logic and requiring only that each of the developers agree on a set of IDL.

Although the CORBA technology provided several important advances, it became apparent that the mechanism by which systems were deployed was still dependent on manual configuration. The CCM evolved to address the need for specifying the requirements for deploying a set of application software by describing what resources were required to deploy the system successfully on a set of hardware. The method for describing the components of a system and the related deployment requirements is through a set of eXtensible Markup Language (XML) files. XML is a text-based language that utilizes tags to define items, their attributes, and values. This CCM XML was the genesis of the SCA Domain Profile XML.

With this brief summary of background information and foundation technology as a backdrop, the next sections provide a summary of what the SCA is, is not, why you would (or would not) want to use it, and a brief history of its evolution.

1.2.1 The Evolution of the SCA

The United States military was (and is) facing an increasingly critical need to support communications for multiple missions, rapid deployment, diverse mission scenarios and objectives, increased interoperability, and to reduce the cost of operations. One of the primary obstacles to meeting these challenges was that the bulk of the radio systems were predominantly hardware-based, limited to those waveforms that were designed into the system, and incapable of being upgraded or adding new waveforms without significant cost due to hardware re-design.

Concurrently, over the past two decades, the capabilities of processors have increased dramatically, special purpose processors such as DSPs and FPGAs have become commonly available, and the speed and resolution of Analog to Digital and Digital to Analog circuits have steadily increased. The result is that more of the waveform signal processing that once was exclusively the preserve of the analog domain was migrating into the digital domain implemented in software. Early experiments in software-based radios such as SpeakEasy showed that there were significant benefits to be gained by moving towards a software-based architecture. Many of the radio manufacturers had already started down the path of implementing core signal processing components in software. Early multi-channel radio systems developed in the 1990s, such as the Joint Combat Information Terminal (JCIT) and the Digital Modular Radio (DMR), provided a software infrastructure for the management of radio resources.

With the need to enhance reconfigurability, support multiple missions, and reduce long-term operations and maintenance costs as a background, the Joint Tactical Radio System (JTRS) Joint Program Office (JPO) was formed to develop a new family of software-based, reconfigurable, radio systems. One of the first activities was to define a common software infrastructure that would be applied to this new family of radio systems. Thus, the SCA was born.

The timeline in Figure 1.3 illustrates several key milestones in the evolution of the SCA specification. There were several preliminary versions but 1.0 was the first version of the specification used to develop an initial implementation of the SCA. After an incremental release with version 1.1, significant portions of the specification were re-worked resulting in version 2.0. Version 2.0 had a number of issues and required some additional details and specifications to address all the aspects required of a software infrastructure. Nonetheless, there were several 2.0 implementations that provided valuable feedback to the specification development process. Again, an incremental version was released in mid-2001, version 2.1,

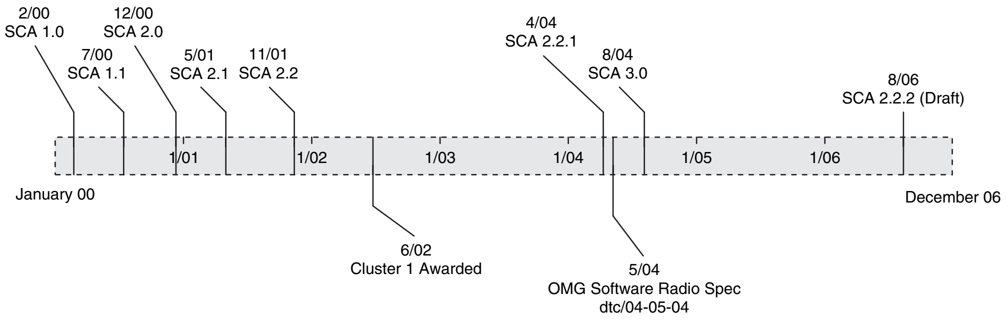


Figure 1.3. SCA specification timeline

followed by 2.2 in November of the same year. Version 2.2 was generally considered to be complete enough to implement and apply to a fielded software radio system.[2]

In June 2002, the first major program to apply the SCA was awarded to Boeing by the Communications and Electronics Command (CECOM). The Cluster 1, later renamed the Ground Mobile Radio (GMR) program, was the inaugural project using version 2.2 of the SCA. Other JTRS Cluster programs were awarded, then in April 2004, almost three years after the 2.2 specification, version 2.2.1 was released. This version cleaned up many of the errors in 2.2 and incorporated several clarifications and enhancements. One of the significant changes was that with version 2.2.1, the Log Service was removed from the SCA specification and the OMG Lightweight Log specification was referenced instead. In mid-2004, the OMG released its Software Radio Specification. The OMG specification was initiated by a number of the individuals who had contributed to the SCA development. The original objective was to evolve the SCA into an industry standard rather than a military-only specification. However, as the specification evolved in the OMG, it took on a life of its own with the resultant OMG specification being significantly different from the SCA specification.

At the same time, issues with waveform portability were being raised through the on-going JTRS Cluster programs. The basic problem was that the code developed for a GPP was reasonably portable between platforms. However, the code developed for a DSP and FPGA generally remained specific to the particular processor and architecture of the radio.[3]

This portability issue came to a head in late 2004, resulting in several special workshops called by the JPO to address the DSP and FPGA portability issue. The result of these workshops was the SCA 3.0 specification. This version of the SCA changed little of the core requirements describing the SCA. It did, however, define additional constraints on DSP software related to what system calls could be used by DSP code, defined a proposed set of waveform components, proposed a high-level data transport design (called HAL-C), and had an Antenna API section. The general reaction in the community was that the specification required additional work and, although the concepts and approaches were potentially useful, more detail and analysis was required in order to achieve a set of descriptions that could be implemented efficiently.

From late 2005 to early 2006, the JPO was re-organized to address resolution of problems with the Cluster programs more effectively and move forward. The program office was moved to San Diego, CA, from Washington, D.C. and is now administered by the Navy SPAWAR office. In mid-2006, version 2.2.2 was released. As the number implies, this is an incremental version from the 2.2.1 version of the SCA. Furthermore, the 3.0 version is shown on the JTRS website as 'not supported.' Thus, at the time of this publication, version 2.2.2 is the latest release supported by the program office.

[2] *Although in principle this was true, there remained a number of important clarifications, corrections, and additional requirements that were necessary to transform the 2.2 release into a reasonably solid specification. For example, the IDL provided in appendix C of the SCA specifications would not compile as defined due to name conflicts with POSIX named values.*

[3] *Waveform portability became a significant issue within the SCA community. Although the portability of waveforms across multiple platforms is certainly a desirable objective, the interpretation of portability evolved into reuse without modification. This was never a fundamental requirement of the SCA. Waveform portability can be viewed as a cost function with the objective to minimize the cost as much as possible. Due to differences in physical hardware, processor, and transport architectures between different board manufacturers, simply moving a waveform from one system to another without modification is unrealistic.*

1.2.2 What is the SCA?

The main purpose of the SCA specification is to define the Operating Environment (OE) software, also commonly referred to as the Core Framework, which implements the core management, deployment, configuration, and control of the radio system and the applications that run on the radio platform. In order to provide a common reference for describing what the SCA is and isn't, it is useful to refer back to the introduction provided with the SCA specification. The quote below is an excerpt from the Introduction.

> The Software Communication Architecture (SCA) specification is published by the Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This program office was established to pursue the development of future communication systems, capturing the benefits of the technology advances of recent years, which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs. The goals set for the JTRS program are:
>
> - greatly increased operational flexibility and interoperability of globally deployed systems;
> - reduced supportability costs;
> - upgradeability in terms of easy technology insertion and capability upgrades; and
> - reduced system acquisition and operation cost.
>
> In order to achieve these goals, the SCA has been structured to
>
> - provide for portability of applications software between different SCA implementations;
> - leverage commercial standards to reduce development cost;
> - reduce development time of new waveforms through the ability to reuse design modules; and
> - build on evolving commercial frameworks and architectures.

SCA V2.2, November 17, 2001, p. vii

As the above quote states, the key objectives are to increase flexibility and interoperability, reduce support costs, provide upgradeability through technology insertion, and reduce system acquisition costs. None of the stated objectives are related to any technical design, development, or waveform aspect of the radio system Thus, the first fundamental objective of the SCA is to improve the business case for evolving and enhancing communications systems and their procurement.

In order to achieve this objective, the SCA structure is intended to 'provide' for portability of applications software, leverage commercial standards, support the reuse of waveform design modules, and build on evolving commercial frameworks. As with the objectives noted in the previous paragraph, the SCA structure focuses on design and development processes to reuse design modules and leverage commercial software and standards.

1.2.3 Common SCA Perceptions

The previous section provided a brief description of the SCA. It is a truism that any technology is often received and perceived differently by each individual: Some of the perceptions are based in fact and some are based on an incomplete understanding of the technology. The following paragraphs discuss some of the commonly cited misconceptions about the SCA.

1.2.3.1 The SCA defines a technical architecture for a software radio

There is nothing in the SCA specification that provides technical data or guidance on the design and implementation of a software radio. The SCA, based on the CORBA Components Model, defines an architecture for the deployment of applications. In the case of a software radio, those applications tend to be waveforms.

1.2.3.2 The SCA enables reusable components

The SCA enhances reusability from two perspectives. First, the SCA specification defines a common set of interfaces for basic deployment configuration, and control of applications. So, from the perspective of user interfaces and external control of the system, the same interface calls that are used to load, start, and stop a SINCGARS waveform are identical to the FM3TR waveform. Second, the Application Programmer Interface (API) appendix to the specification is intended to promote reusability of waveform software components through common waveform interfaces. This continues to be an area of on-going discussion because all radio system developers have different perspectives as to what the interfaces should be for a specific waveform.

1.2.3.3 A waveform can be moved from one SCA platform to another without modification

Many individuals have interpreted the portability objective of the SCA as reusability without modification. The SCA specification defines common, high-level interfaces for deploying, configuring, controlling, and monitoring the hardware and software applications within an SCA-based radio system. This simplifies the effort required to port applications because the interfaces do not change. However, deploying waveforms across multiple radio systems without modification was never a stated requirement.

1.2.3.4 The SCA results in a waveform performance impact on my system

The simple fact is that, once the SCA deploys the waveform on the radio system, the SCA Core Framework goes into a quiescent state and does not utilize significant processor cycles. Also, for waveforms implemented largely in FPGA or DSP processors, there is typically no impact due to the SCA on functioning waveforms in those processors. There are some impacts in terms of the memory footprint required to support an SCA framework. However, the SDR Forum, NASA, and other groups are looking into reduced footprint architectures. Where the framework is running on the same GPP being used for waveform processing, some performance impact may be encountered. In this case, standard systems analysis to evaluate the load margins is necessary.

1.2.3.5 I must use CORBA for the data transport

CORBA should be the starting point but is not mandatory if performance reasons prohibit it. Also, individuals often confuse the latency impacts of the underlying transport mechanism, which typically defaults to TCP/IP, as being synonymous with CORBA. In reality, CORBA is a protocol layer, much like Hypertext Transfer Protocol (HTTP), that rides on top of the data transport mechanism. Most modern ORBs support plugable transports allowing customization and optimization of the actual data transport.

1.2.3.6 The SCA is only applicable for small radios

The SCA is not specifically targeted for any one type or class of radio system. Small form-factor, resource-limited radio systems have a more significant set of issues to overcome when building an SCA-compliant handheld or manpack radio, due to their Size Weight, and Power (SWaP) constraints. This is usually due to the fact that the GPP on the small radio is already used extensively for waveform code, and processing impacts due to adding the framework can be significant.

1.2.3.7 The SCA and/or CORBA is not suitable for large, complex systems

The origins of the SCA are based on the JTRS program which focused on tactical radio systems. These systems ranged from small handhelds to rack-mount systems in vehicles, ships, and aircraft. Although these systems do not have the complexity of large terminal systems, it is possible to apply the SCA to larger systems. More thought must go into the architecture of the system, however. It may be the case that the SCA manages the core set of radio equipment and waveform deployment under the direction of a higher-level system or network management operation. The key aspect is that the SCA is targeted towards the management of the hardware and software that implement and support the end-to-end waveform application.

As for the applicability of CORBA to large scale systems, it can be said that it is in wide use throughout industry. Large, distributed Java-based applications are, in fact, using CORBA and Java remote procedure calls are using the CORBA protocol. Also, the Iridium satellite Command and Control segment integrated a COTS-based system, OS/COMET, within a comprehensive CORBA framework. The resultant system ran on over 50 computers and was comprised of several hundred processes.

1.2.3.8 The SCA is not suited to systems above 2 GHz

This reason is typically rooted in the fact that the SCA originated in the JTRS program, which was focused on the tactical radio spectrum under 2 GHz. The simple fact of the matter is that there is nothing in the SCA specification, either explicitly or implicitly, that limits the usefulness or applicability of the SCA to systems above 2 GHz.

1.2.4 Why Use the SCA?

Given that the SCA is not a technical reference architecture for a software radio, why should you use it? Part of the answer lies in the fact that, as discussed earlier, the objective of the SCA is to improve the business case for reduced cost for enhancements, upgrades, and logistics. These are benefits that are primarily realized by the customer or recipient of the SCA system. Thus, more often than not, the reason cited for using the SCA is that it is a requirement levied on the project. In order to achieve longevity and acceptance, there must be business reasons for the developers of software radios to use the SCA. Some of the business reasons to consider using the SCA architecture are:

1.2.4.1 The SCA provides a common infrastructure for distributed application deployment

Although the SCA was founded and focused on the radio domain, the basic infrastructure for deployment of the application components is applicable across virtually any domain. This includes radio-related areas, such as signal processing systems, as well as unrelated domains, e.g. device and software management across processors within a vehicle.

1.2.4.2 The SCA allows quicker integration of external applications

Because of the small set of interfaces defined within the SCA, external applications can easily be integrated using the IDL specified for loading, starting, stopping, and controlling applications within an SCA system. One such example is the integration an SCA system within a network of systems. Overall management of the radio nodes is often performed by a network management or network control operation. These management systems often utilize Simple Network Management Protocol (SNMP) or Java, which includes native support for CORBA. So, when the network control software decides that it needs to load a particular waveform or communications software on an SCA radio, it simply issues a Java Remote procedure Call (RPC) to the SCA radio to load the desired waveform. If SNMP is used, then an SNMP proxy can be implemented that provides the SNMP interface to the network system and issues Java calls to the SCA radio.

1.2.4.3 The SCA enables easier insertion of new technology

Because the SCA specifies the interfaces for deployment, configuration, control, and monitoring of the hardware and software with an SCA system, new technology can be inserted with less cost and impact. As an example, part of the description of the application in the SCA identifies a software component that provides a certain function. Within that description, there may be several different implementations, e.g. one on a DSP, one for a Intel GPP running VxWorks, and so on. As technology capabilities progress, an implementation that could once only be realized on a DSP may now be realizable on a GPP.[4]

Because the SCA allows multiple implementations, the new implementation can be deployed on an existing system that has the GPP resource but not the DSP without changing other components. This concept also applies to new hardware.

[4] *This does not imply that no effort is required to port a component from one system to another. It does mean that, if the implementation suitable for a particular system is now available, it can be deployed and configured on the system without rebuilding the entire application.*

1.2.4.4 The SCA provides an infrastructure for extensibility and integration

The SCA is a foundation for the design and development of a comprehensive radio system. It defines the essential interfaces and behaviors provided in the infrastructure. Applications may be developed and built that provide higher level capabilities and features. Technologies such as cognitive radio may be integrated with an SCA system providing higher-level control and management of the radio resources. For example, a set of SCA radio systems that have a cognitive map of their surroundings and temporal logic can negotiate configurations between them to enhance reliability of the communications. This may take the form of changing parameters of the existing waveforms within the radio or loading new waveforms based on the collaboration with other radio systems.

1.2.4.5 The SCA reduces the development of Non-Recurring Engineering (NRE) for system development

Standardizing on a common software infrastructure reduces the effort required for future systems because a common set of control routines and implementations are developed. This can have a positive impact on the development cost and schedule. The key is to develop a standard infrastructure that is applied across a variety of systems that supports the SCA capabilities. This last item is central to how an organization approaches the development and use of the SCA.

Now that we've covered several of the positive and negative viewpoints of the SCA, let's continue with a brief overview of the technical aspects of the SCA. The remainder of this chapter will introduce the SCA Operating Environment and specification structure as a foundation for the subsequent chapters.

1.3 The Operating Environment

The JTRS program Software Communications Architecture specifies the requirements for a common software radio Operating Environment (OE). The OE consists of the Core Framework (CF), the CORBA ORB, and the operating system. The operating environment specifies the interfaces, rules, constraints, and procedures that must be adhered to in order to implement an SCA-compliant radio system. The Core Framework provides

- a collection of common services used by the waveform and other applications;
- software enabling the installation, configuration, management, and control of waveforms;
- a federated file system enabling common file system operations and access across multiple processing platforms;
- device interfaces that provide a common abstraction of the underlying physical hardware.

As illustrated in Figure 1.4, the SCA Core Framework is shown along the vertical axis and can be thought of as the management plane of the SCA system. The waveform is illustrated as a set of components that are assembled across the horizontal axis forming the application plane.

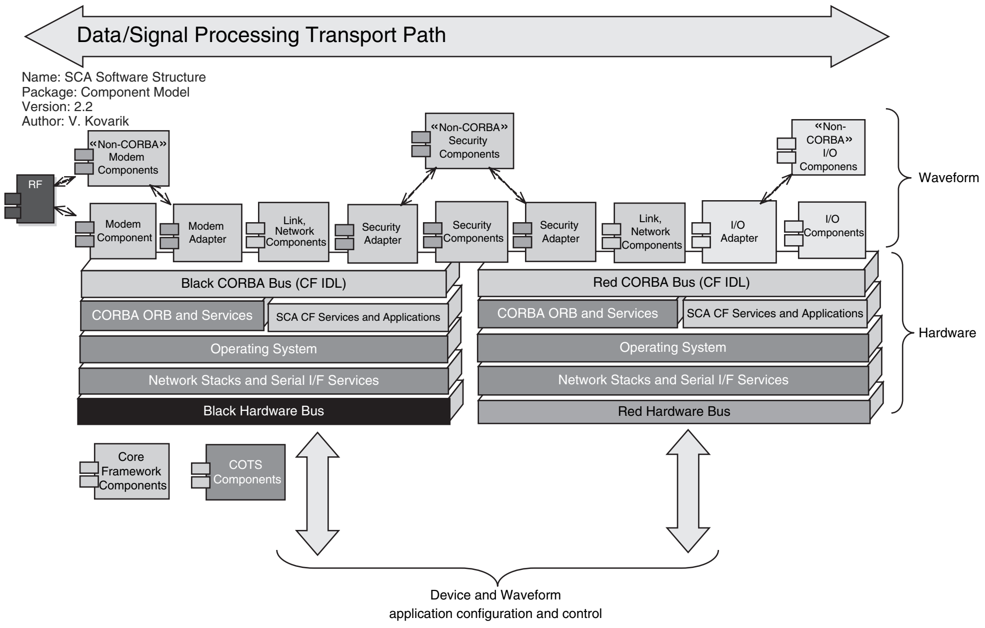


Figure 1.4. The SCA waveform component organization

1.3.1 Conceptual Organization

Conceptually, a SCA radio has three segments: i) The Waveform Deployment; ii) the Core Framework; and iii) the Domain Profile. These three segments are each divided into physical and logical views. In the Waveform Deployment segment, the radio hardware is the physical view of the radio system. However, the waveforms are realized through software that is loaded on the physical radio elements. There are two layers in the logical view of the radio system. The first consists of the set of components that form a waveform application or other service on the system. The second is the application that provides the top-level interface and control for the set of components.

The Core Framework segment includes all software required to manage the radio system and deploy applications. It has a physical view and a logical view as well. The physical view of the Core Framework provides high-level management of the physical devices in the radio system. The logical view provides the same for the waveform applications and other services.

The Domain Profile segment consists of the set of XML files that describe the hardware resources within the radio system, the waveform application structure, and dependencies between waveform components, connections between components, and dependencies on hardware resources. This is illustrated in Figure 1.5.

1.3.2 OE Interface Constraints

Figure 1.6 illustrates the relationships between the primary components of an SCA-compliant system. At the base level are the services provided as POSIX interfaces by the operating system that form the AEP.

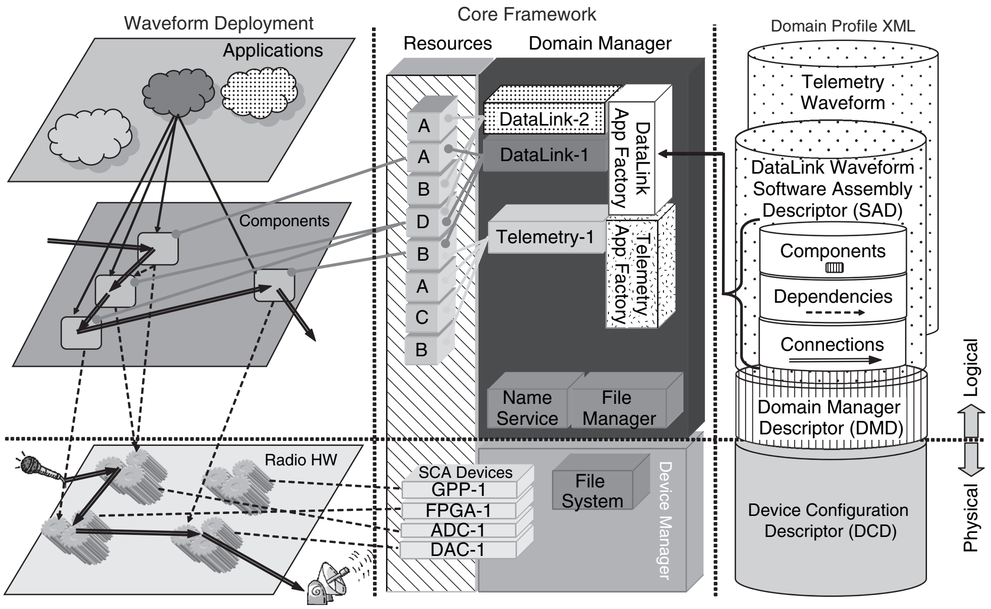


Figure 1.5. Abstraction layers in an SCA system

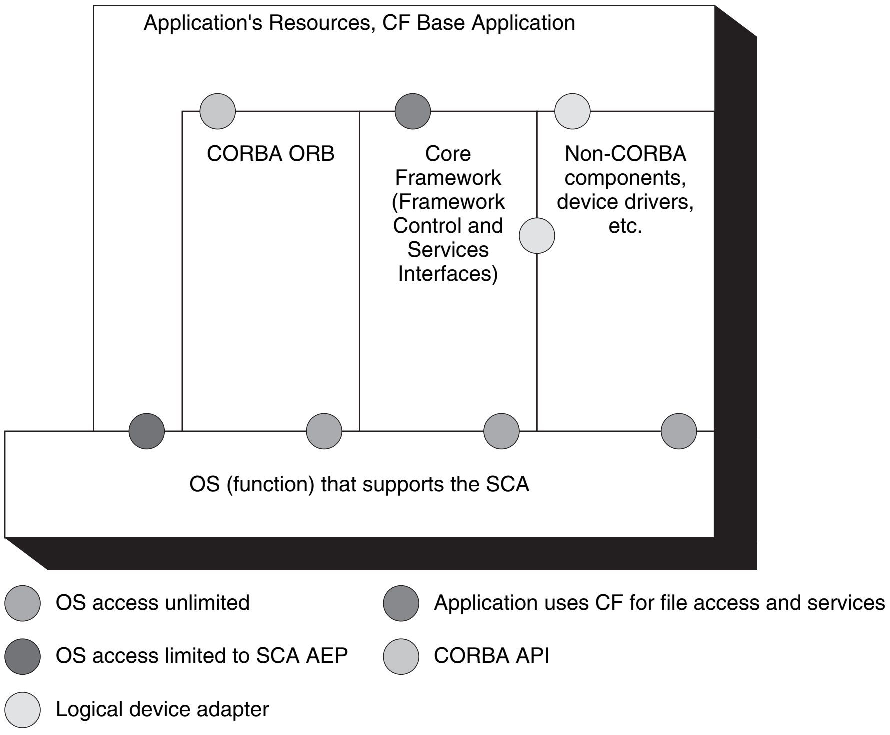

Application's Resources, CF Base Application
CORBA ORB
Core Framework (Framework Control and Services Interfaces)
Non-CORBA components, device drivers, etc.
OS (function) that supports the SCA
OS access unlimited
Application uses CF for file access and services
OS access limited to SCA AEP
CORBA API
Logical device adapter

Figure 1.6. The SCA interface constraints

The objective of the Application Environment Profile is to provide a constrained set of well-defined operating systems calls that minimize the impact to application code. This objective is only valid for general purpose processor components because DSP and FPGA processors do not have an operating system. However, some DSPs do support an operating system and CORBA.[5]

The CORBA ORB provides a common middleware for the system and has unlimited access to the underlying operating system. The core framework is an implementation of the SCA specification providing the essential infrastructure components and services for the radio system. The non-CORBA components and device drivers are comprised of low-level drivers such as those provided by a device manufacturer for a particular physical device for one or more operating systems. The application is composed of the set of application components and resources that form an operational waveform. Finally, the operating system provides the underlying set of platform specific services.

The CORBA ORB, Core Framework, and Device Drivers have unlimited access to the underlying operating system calls and services. However, as illustrated in Figure 1.6, the application is limited to a set of POSIX calls to the operating system. The rationale behind this limitation is that the application will be more easily ported to other platforms if it is constrained to a specific set of interfaces and the software radio platform is mandated to support the set of POSIX calls specified in order to be SCA-compliant.

Although this rationale has some merit, the porting of a waveform has a wide ranging set of complex issues and the POSIX constraints, in and of themselves, help but do not achieve, portability of waveforms between platforms. This becomes particularly evident when a waveform application is developed that uses a DSP or a FPGA as the processor for some potion of the waveform functional chain.

1.4 The SCA Specification Structure

The SCA specification consists of three major components:

- Software Communication Architecture Specification (JTRS-5000SCA)
- Application Program Interface Supplement (JTRS-5000API)
- Security Supplement (JTRS-5000SEC)

The SCA specification is the primary specification for building an SCA-compliant radio systems. The SCA specification defines the operational environment requirements and basic functional requirements. The contents of the SCA specification are the primary focus of this book.

The Application Program Interface (API) Supplement provides guidelines and requirements for building modular and portable application components. Certain aspects of the API Supplement will be referenced or discussed within this book. However, a thorough treatment of the API Supplement and the construction of a portable SCA application would requirement significantly more detail than can be reasonably included in this book.

[5] *The POSIX interface is required to be used by the application when accessing operating system services. All other components, e.g. the CORBA ORB, the Core Framework, non-CORBA components, and device drivers, may have unlimited access to the operating system.*

The Security Supplement defines specific security requirements and APIs related to the design and construction of a Type 1 secure radio system. This book will reference some portions of the Security Supplement as it relates to the rest of the content. However, as with the API supplement, the level of detail required to address fully the security issues associated with the design and development of an SCA-compliant system are beyond the scope of this book.

Section 3 of the SCA specification identifies the requirements for the core software infrastructure architecture definition. The organization of Section 3 is shown in Figure 1.7. The subsection on the Operating Environment addresses the Core Framework and services. Section 3.2 focuses on the application interfaces and functional requirements. Section 3.3 provides the requirements for the SCA device interface and Section 3.4 identifies general requirements.

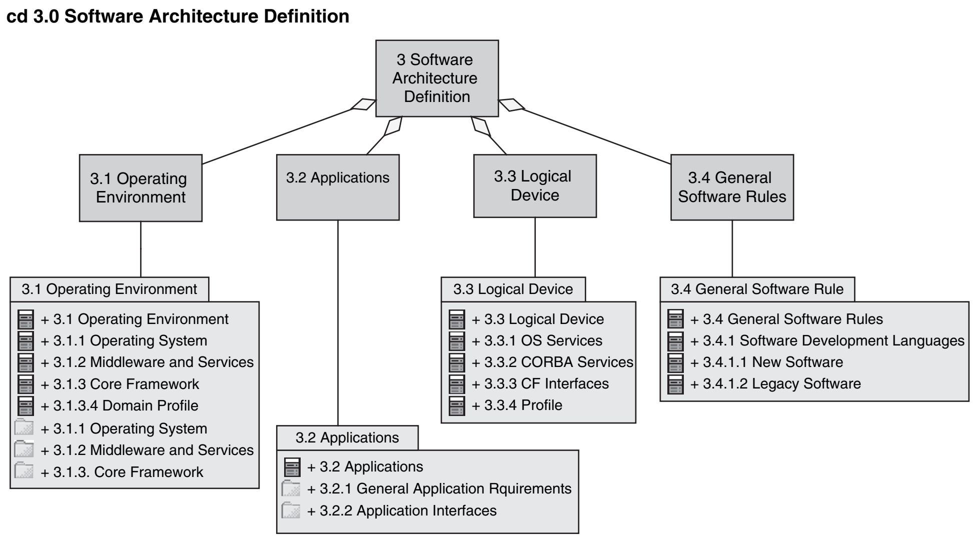

cd 3.0 Software Architecture Definition
3 Software Architecture Definition
3.1 Operating Environment
3.2 Applications
3.3 Logical Device
3.4 General Software Rules
3.1 Operating Environment
+ 3.1 Operating Environment
+ 3.1.1 Operating System
+ 3.1.2 Middleware and Services
+ 3.1.3 Core Framework
+ 3.1.3.4 Domain Profile
+ 3.1.1 Operating System
+ 3.1.2 Middleware and Services
+ 3.1.3. Core Framework
3.2 Applications
+ 3.2 Applications
+ 3.2.1 General Application Rquirements
+ 3.2.2 Application Interfaces
3.3 Logical Device
+ 3.3 Logical Device
+ 3.3.1 OS Services
+ 3.3.2 CORBA Services
+ 3.3.3 CF Interfaces
+ 3.3.4 Profile
3.4 General Software Rule
+ 3.4 General Software Rules
+ 3.4.1 Software Development Languages
+ 3.4.1.1 New Software
+ 3.4.1.2 Legacy Software

Figure 1.7. The SCA specification organization

The Operating Environment, Section 3.1, of the SCA specification, defines the bulk of the requirements that must be met for a system to be SCA-compliant. It contains the common components of the Core Framework including the Domain Manager, the Log Service, CORBA requirements, and other components.

In Section 3.2, the requirements that must be met by the SCA application are defined. The application is comprised of the software that implements an end-to-end waveform. Thus, this section is met primarily by the waveform application developer. However, certain common aspects of the application are provided as part of the Core Framework components.

In order to manage, configure, and control the devices that make up a radio system, a Logical Device interface must be implemented. Thus, the device manufacturer and/or radio system integrator must provide implementations of the SCA Logical Device in order to integrate the hardware into the overall SCA system. These requirements are defined in Section 3.3. Sections 3.1 through 3.3 will be presented in more detail as each of the functional requirements areas are discussed in later chapters. The general software requirements, Section 3.4, and other, non-functional requirements, are discussed in this section.

Figure 1.8 shows the relationships between the SCA specification and an implementation. The SCA specification provides the interface and high-level behavioral specification to be implemented by the Core Framework. Figure 1.8 also shows the standards developed by the Object Management Group (OMG) that are referenced by the SCA specification.

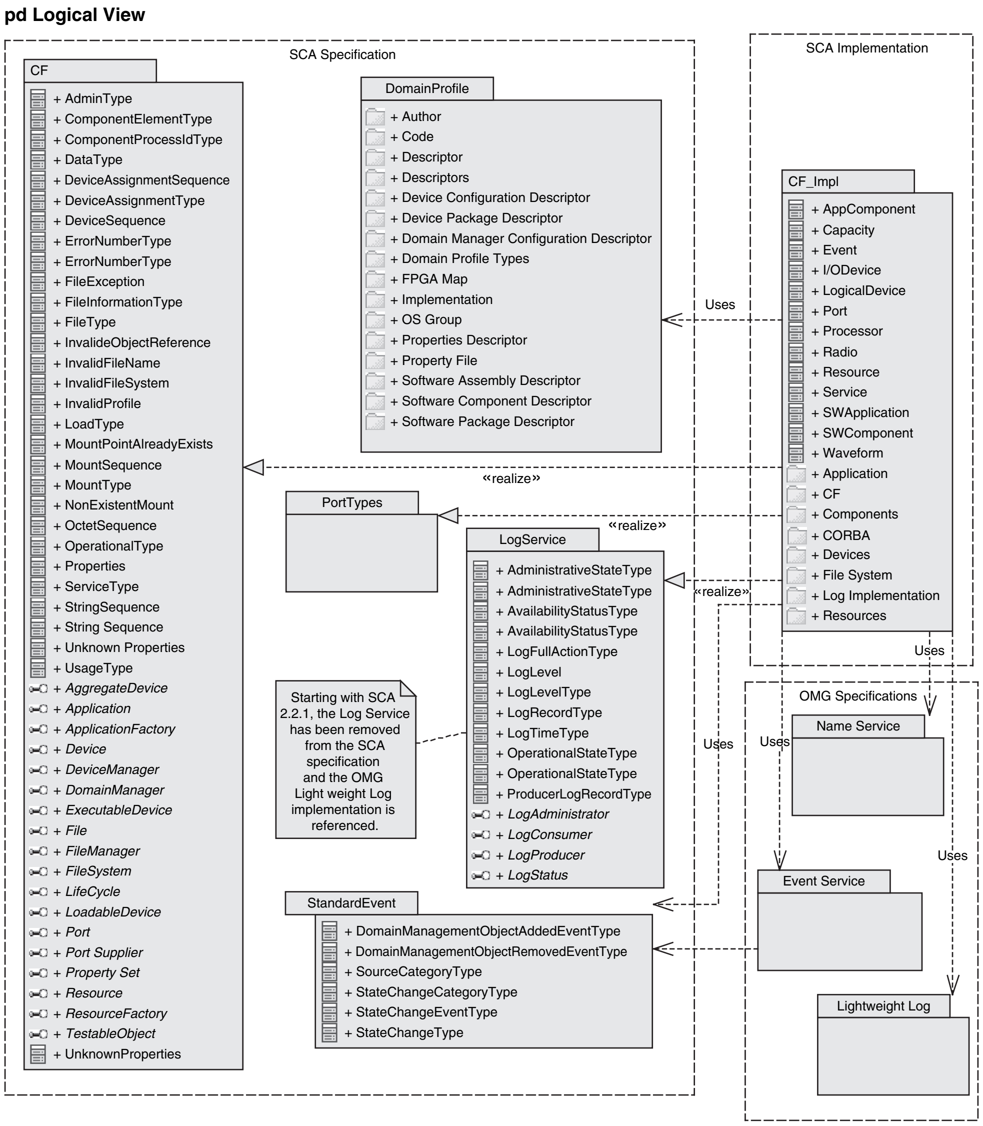


Figure 1.8. SCA specification versus implementation

The bulk of the Core Framework interfaces and behavioral specifications are contained within the Core Framework package. The IDL for the set of interfaces is defined as a single interface module. The Domain Profile specifies the XML files that are used to describe the underlying hardware and the software components and the interconnections that are required to deploy the waveform. There are some differing interpretations of what constitutes the Domain Profile. Some interpretations refer to the set of XML files as the Domain Profile, some view the internal parsed XML data structures as the Domain Profile, and others view the Domain Profile as a collection of internal data structures that form the internal domain knowledge of the SCA radio system.

This book takes the stance that the XML files represent a human readable version of the Domain Profile and that the internal data structures built as part of the processing of the XML profile form the internal Domain Profile used by the Core Framework in the process of instantiating a waveform. The hierarchical tree structure generated by the parser is viewed as an intermediate data structure that takes the text form of the XML files and parses them into a canonical form that allows easy extraction of the salient information required to deploy a waveform. It is true that the XML parse tree can be traversed and used directly as the internal representation of the Domain Profile. However, this form of the data is inefficient for the types of constraint enforcement required of the Application Factory during application instantiation.

Underlying all the above abstractions are a set of common services that provide critical capabilities for all the SCA components:

- **Name Service** – The Name Service enables a component within the radio system to locate a required service or application component and connect to that component.
- **Event Service** – The Event Service provides an asynchronous mechanism for components to publish events on an Event Channel and other components to register to receive those events.
- **Log Service** – The Log Service provides a basic logging capability within the SCA radio system that allows components to create a record describing some activity or state and have that record time-stamped and saved within a log for subsequent retrieval.
- **File Service** – Perhaps one of the most crucial services provided is the File Service which provides a common abstraction of a file system for all SCA radio components independent of the actual underlying operating system and file system implementation.

Part I of this book is organized along the high-level abstractions identified above. Chapter 2 addresses the common framework services to provide a frame of reference for components and references used in the description of other logical interfaces. After describing the common services, each of the subsequent chapters addresses each of the logical abstractions defined above starting with the Resource and ending with the Domain Manager.

1.5 Summary

The SCA specification provides an implementation-neutral framework for implementing and deploying applications. The general requirements of the SCA define a set of hardware and software constraints that the system must adhere to or provide. In the next chapter, a conceptual view of the operation of an SCA-compliant system is presented using the Use Case model approach of the Unified Modeling Language (UML).

2

Operational Scenarios

In order to provide a context of the operational behavior of a system, it is helpful to define a set of use cases that describe the overall behavior of the system from an external or user's point of view. This section provides a set of use cases for the software defined radio. The use cases discussed are intended to provide a broad overview of the essential operational characteristics of an SCA-compliant software radio. Any specific radio system would require a more extensive set of operational scenarios and uses cases to describe the detailed dependencies and interactions of that particular system.

Figure 2.1 shows the simplified set of operations and users that interact with the radio system. Two broad user types or roles are shown in the figure. The Radio User represents a typical operator of the radio set. As such, she would interact with the radio at a basic equipment level, e.g. power on the radio, basic hardware initialization, and shutdown. At a more abstract level, the Radio User interacts with the radio to instantiate and control a waveform application. At this level, the uniqueness of the software defined radio begins to emerge. The Radio User can configure the radio to load and run a waveform dynamically; that is, the radio may be (re)configured to execute a set of components that supports different waveform applications on demand.

In addition to the Radio User, there is another role, the Radio Engineer, who provides the radio system configuration, integration, testing, and waveform installation.

The following paragraphs provide a high-level overview of essential activities involved in each of the use cases identified in Figure 2.1. The aim of these use cases is to provide a common foundation of behavior and interrelationships for the key functionality of an SCA-compliant radio system. The objective is to identify essential steps and conditions in each of the high-level use cases. These will provide a foundation that will be referred to in subsequent chapters as we explore the specific behaviors of the components of an SCA Core Framework. It should be noted that the sequence diagrams presented in the following sections are intended to provide a high-level view of the interaction between the use cases and not to provide a rigorous UML model of the use case interaction.

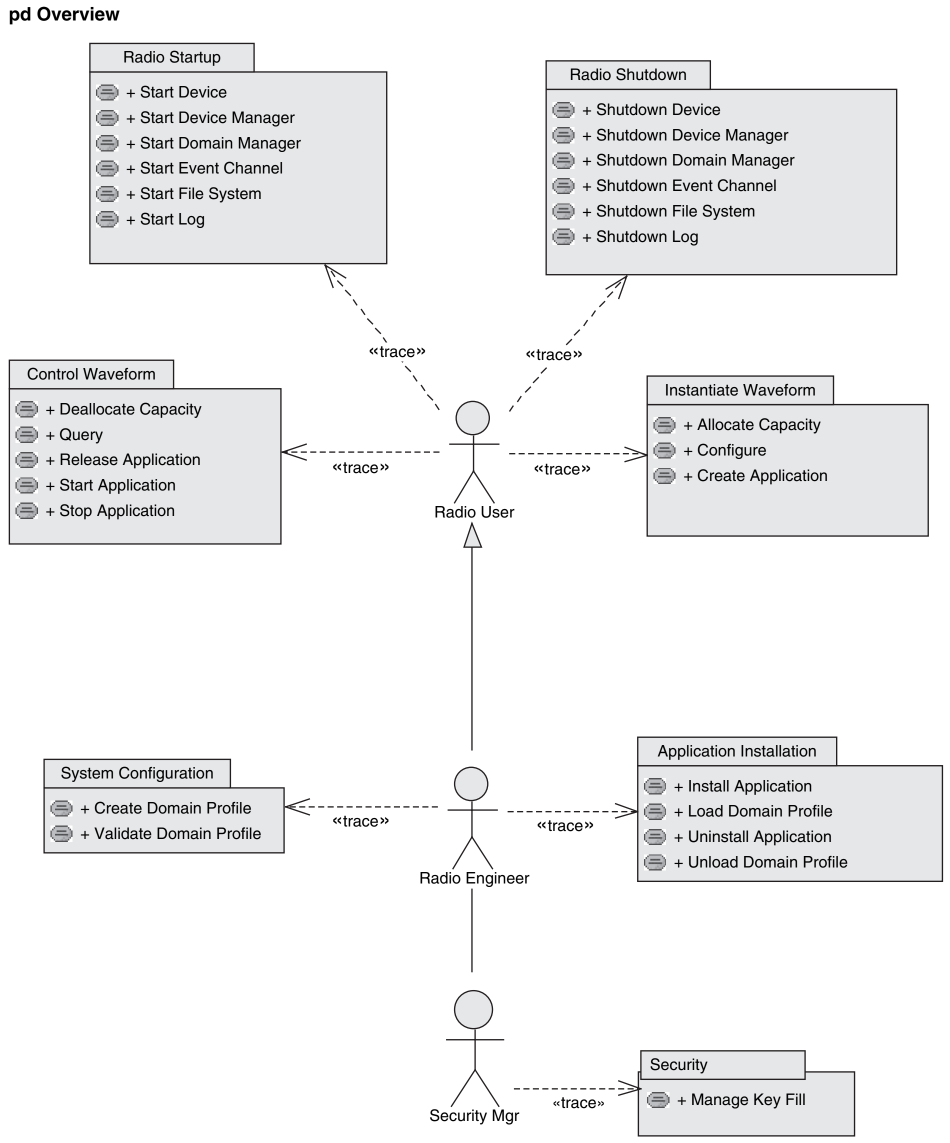


Figure 2.1. Top-Level radio user roles and activities

2.1 Startup

The Startup use case addresses the process of bringing up the radio system from a power off state to power on, with each of the hardware components powered on and initialized and the essential components of the SCA Core Framework loaded and initialized. Figure 2.2 illustrates the core relationship between the use cases that form the startup sequence.

The Start Device Manager use case describes the startup of what is commonly referred to as a node. Here 'node' will be used to identify a physical hardware component, such as a VME or Compact PCI card that contains one or more physical devices that provide some part of the processing necessary to implement an SCA-compliant software defined radio system. Typically, the initial or boot node is a Single Board Computer (SBC) or some other configuration of a general purpose processor and operating system.

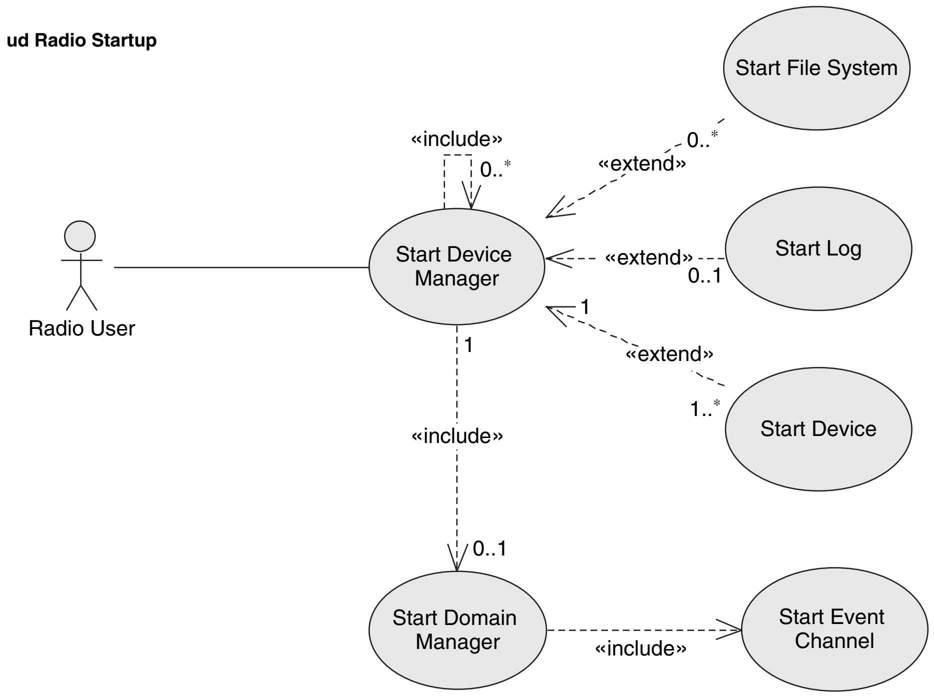


Figure 2.2. Radio system Startup use cases

When starting up an SCA system, the premise is that there is an initial boot device and it is the Device Manager for this initial boot device that orchestrates the startup and initialization of the SCA Core Framework and its related components. As the Device Manager starts, it launches a number of services and applications (Figure 2.3), beginning by initializing a file system. This file system then provides a storage repository for the Device Manager and its associated Devices. The Device Manger then starts up the Log Service, which provides the ability to record system messages, warnings, and failures across the entire set of SCA hardware and software. The Device Manager then starts the Domain Manager which will serve as the primary repository and control point for the system. The Device Manager then starts each of the Devices associated with the Device Manager.

This approach is new to the current version of the SCA. Prior versions had the Domain Manager starting first which then provided an initialized and running Domain Manager for the Device Manager and Devices to access. In the current approach, it is not guaranteed that a component is completely initialized and ready to accept incoming CORBA calls when initiated by a client.

For example, since the Device Manager on the boot device can start the Domain Manager, when the Device Manager attempts to register with the Domain Manager, the Domain Manager may not have completed loading and initializing. Consequently, a mechanism for automatic retry must be incorporated into the design of certain components. This ensures that the Core Framework components have a reasonable chance of completing the start up process without failing by hitting a deadlock condition.

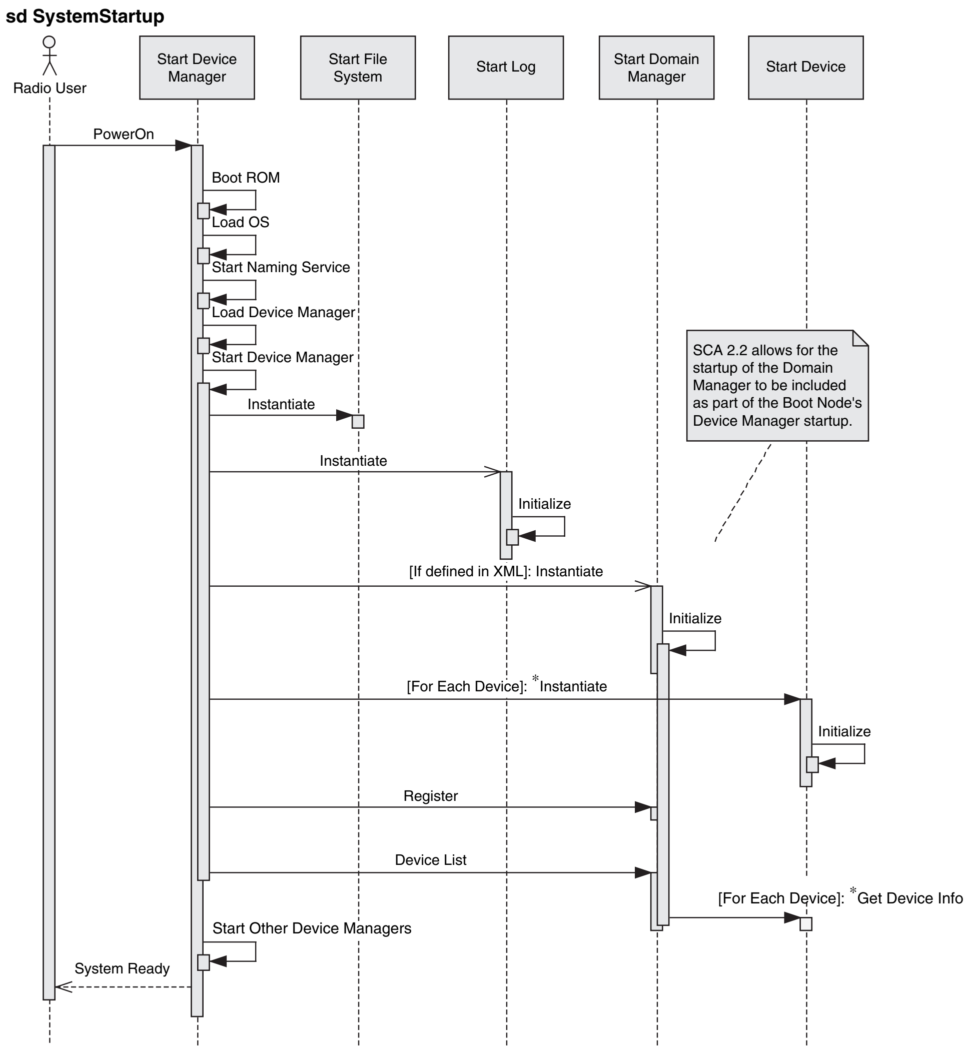


Figure 2.3. Radio system Startup sequence

Upon power up, the boot node will execute the Boot ROM for that particular hardware configuration. The Boot ROM provides the power-on logic and initialization for the hardware. Once the Boot ROM has been loaded, an operating system is loaded. This provides the core set of services and functions to load and manage the core SCA software components for the system.

Once the operating system is loaded, the system is ready to start loading components of the Core Framework. The first entity that needs to be loaded is the CORBA Name Service. The Name Service provides a 'well known' location for applications to look up available services and applications. Several components of the Core Framework will register with the Name Service; hence, it needs to be available and running prior to the initiation of Core Framework components.

At this point, the Device Manager software implementation is loaded. As the Device Manager software component initializes, one of the first tasks it performs is to read the Device Configuration Descriptor (DCD) associated with the Device Manager. This is the XML file that contains the configuration and startup data required. Part II of this book addresses the syntax of the DCD and XML files and Part III provides some examples.

As the DCD is read and processed by the Device Manager software, it specifies the additional software components and Devices that must be started as part of the Device Manager's overall initialization. Typically, one of the first tasks the Device Manager performs is the instantiation of a File System. The SCA File System provides a CORBA-based abstraction of the specific file system implemented by the underlying hardware and operating system. Usually, the Device Manager instantiates one file system. However, it may instantiate more than one or none, depending on the specific type of hardware and operating system configuration and capabilities of the Device Manager's underlying hardware.

The next component instantiated is usually the Event Channel. The Event Channel provides an asynchronous notification mechanism enabling components to post status changes and other state events to an event channel. Other components interested in receiving the event notification need only subscribe to the event channel of interest. The functionality of the Event Channel referenced in the SCA is defined in OMG Document formal/01-03-01: Event Service, v1.1.

Once the event channel is created, the Device Manager starts the Log Service. Prior to SCA v2.2, the Log Service was defined as part of the SCA Specification. Beginning with SCA v2.2.1, the SCA specification references the OMG Lightweight Log specification. Both are similar in functionality and IDL interface. The Log Service provides a short-term, in-memory log that may be accessed by any SCA component. The persistent mechanism used by the log is a memory buffer.

Subsequent to starting the log, the Device Manager starts the Domain Manager, if specified in the DCD XML file. The Domain Manager provides the top-level repository and control over the collection of hardware and software that comprises the 'domain' of the radio system.

At this point the Device Manager then starts and initializes each of the Devices specified in the DCD XML file. Each Device is started and initialized. Once the Devices are started by the Device Manager, the Device Manager then notifies the Domain Manager by registering each Device with the Domain Manager. For each Device registered, the Domain Manager then queries the Device to obtain its properties and configuration information.

Finally, any additional Device Managers that need to be started as part of the system startup would be started by the boot node Device Manager. Each of the subsequent Device Managers started would follow the same startup sequence just described for the boot node Device Manager. However, subsequent Device Managers would typically not start a Domain Manager, or Event Channel, as only one of those per system would be started. A File

System and Log Service may or may not be instantiated depending on the resources and requirements of the radio system.

2.2 Shutdown

Radio shutdown is, interestingly enough, not explicitly addressed in the SCA specification. The logical assumption one could make (illustrated in Figure 2.4) is that the Domain Manager functions as the master control component over the entire radio set. Thus, it is the component that is instructed to shutdown the radio set. This is performed by iterating through the set of the Device Managers registered with the Domain Manager and instructing each Device Manager to shutdown (Figure 2.5). Each Device Manager, in turn, instructs each of the Devices registered with the Device Manager to shutdown. Subsequently, the Device Manage terminates the Log Service, if provided, and the File System associated with the Device Manager is shutdown. Once the Device Manager has completed each of these activities, it shuts down. Note that since the Device Manager, and its related components, are terminated, there is no response back from the device being terminated because it ceases to exist.

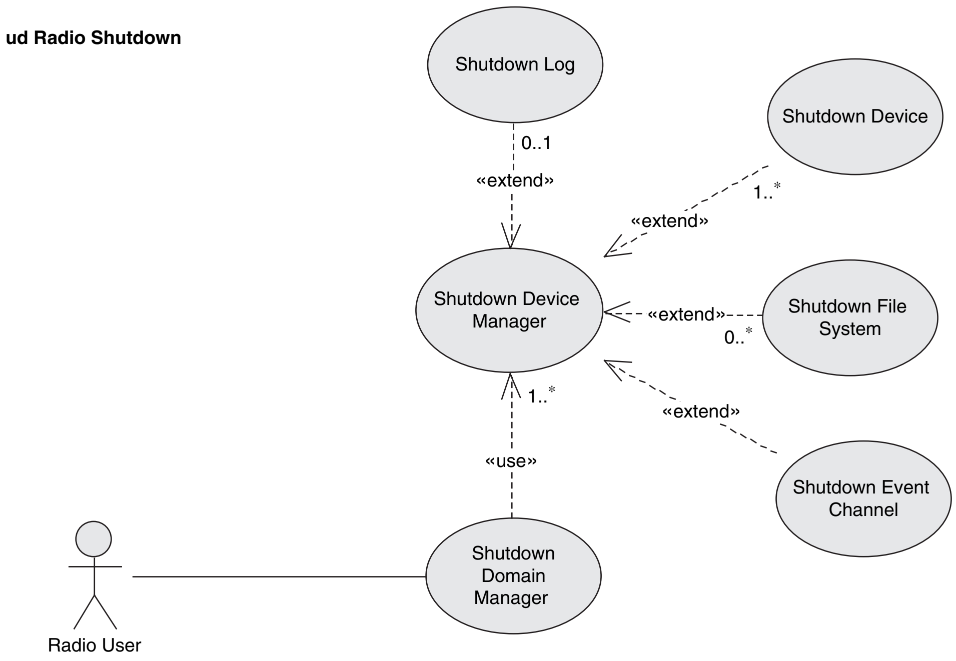


Figure 2.4. Radio system Shutdown use case

After instructing each Device Manager to shutdown, the Domain manager terminates. Again, because of the potential for errors or other notifications being issued by components as they are shut down, the Domain Manager should wait for a reasonable amount of time before shutting down to be sure that the collection of components have successfully terminated.

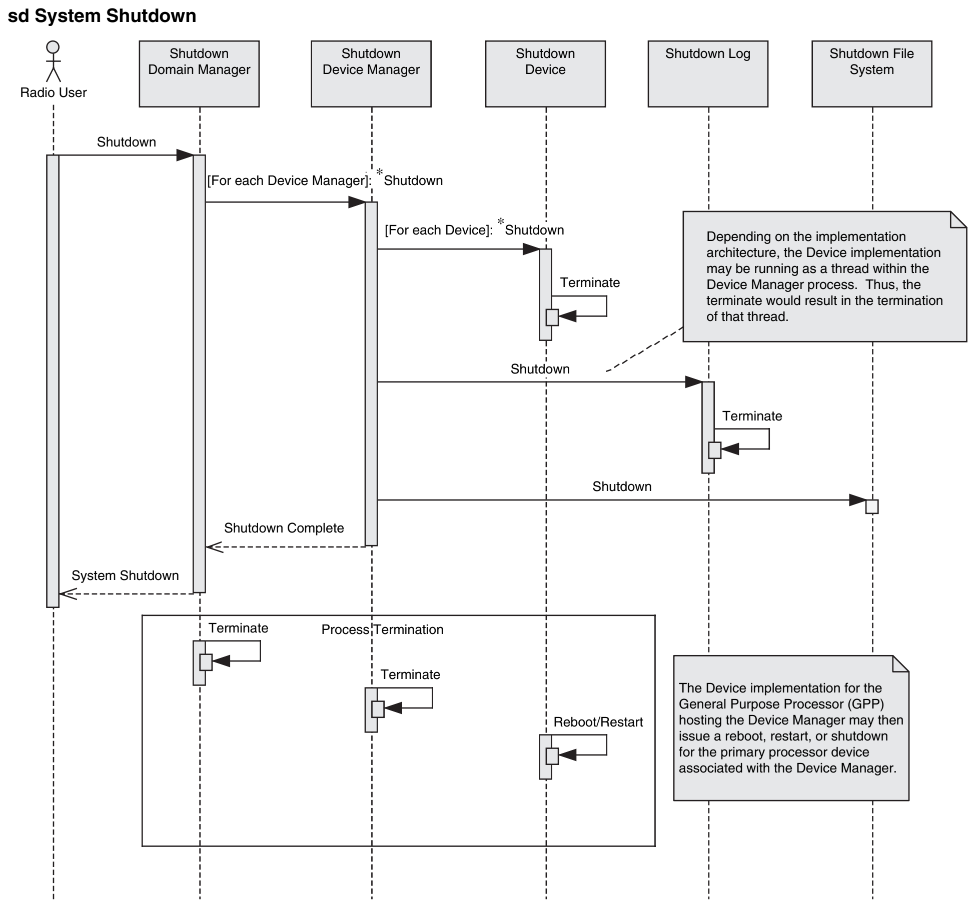

sd System Shutdown
Radio User
Shutdown Domain Manager
Shutdown Device Manager
Shutdown Device
Shutdown Log
Shutdown File System
Shutdown
[For each Device Manager]: *Shutdown
[For each Device]: *Shutdown
Terminate
Depending on the implementation architecture, the Device implementation may be running as a thread within the Device Manager process. Thus, the terminate would result in the termination of that thread.
Shutdown
Terminate
Shutdown
Shutdown Complete
System Shutdown
Terminate
Process Termination
Terminate
Reboot/Restart
The Device implementation for the General Purpose Processor (GPP) hosting the Device Manager may then issue a reboot, restart, or shutdown for the primary processor device associated with the Device Manager.

Figure 2.5. Radio system Shutdown sequence diagram

Since one of the Device Managers functions as the boot node and typically is the node running the Domain Manager, it may include interface calls to the operating system to address hardware shutdown or system reboot. Whether it includes this behavior or not, the Device Manager for the processor running the Domain Manager should wait until the Domain Manager has terminated prior to initiating its shutdown sequence. Once the Domain Manager has terminated, the Device Manager will perform its shutdown sequence. Thus, when the Domain Manager receives the call from the Domain Manager to shutdown, it defers its actual shutdown process until the Domain Manager, and any other SCA processes, have completed their shutdown processes. Then it shuts down and/or initiates a call to the operating system requesting a reboot or restart.

Device Managers that do not have a general purpose process associated with a Device that is part of the collection of Devices associated with the Device Manager, simply issue the shutdown call to the set of Devices associated with the Device Manager. Each Device then performs its shutdown actions.

2.3 Application (Un)Installation

Installation of an application in the context of an SCA-compliant radio system refers to the reading of a set of XML files that describe the hardware and software components necessary to deploy a waveform on a system. The key concept here is that the waveform software is not loaded onto the hardware at this point; only the list of components required, the hardware devices on which the software components are to be loaded, and the connections that tie the components together into a functioning application.

The installation of an application results in the loading of the Software Assembly Descriptor (SAD) file. The SAD file is the top-level XML file that specifies the components, connections, and constraints for a given waveform application.

The term Domain Profile is sometimes used interchangeably to refer to both the set of XML files and the resultant internal set of data structures that represent the content of the XML files in a form useable by the Core Framework components. Although the intent of the original SCA specification intends the XML files to comprise the domain information, it is computationally expensive to re-parse the textual representation of the domain each time a particular waveform is instantiated.

Most Core Framework implementations parse the information into an internal, machine usable representation. This internal representation may be simply nothing more than the XML tree or Document Object Model (DOM) created by the XML parser. Using the XML parse tree, however, does not capture and represent the semantics between the components of the profile. This results in the need to perform tree traversal when instantiating the application.

Consequently, several Core Framework implementations create an internal set of data structures that provide a more efficient representation for retrieving information and updating when creating or destroying an application instance.

Uninstalling an application is the process of removing the Domain Profile information for the specified waveform application. (Note that the SCA specification states that the XML files containing the Domain Profile information are to be removed from the file system as well. This was finally changed in SCA 2.2.1 to allow the XML files to remain on the system.)

This requirement creates a problem for Core Framework implementations that utilize internal data structures to represent a parsed and installed waveform application, because uninstalling the application simply means removal of the internal data structures representing the waveform profile.

Additionally, removal of the files presents operational problems if the end-user is provided the ability to uninstall an application. Rather than merely removing the internal data structures, which prevents the radio from instantiating the waveform until the XML files are once again loaded (i.e. the waveform is again installed), the waveform cannot be instantiated at all because the XML files describing the waveform profile no longer exist.

Some implementations provide the ability to specify optionally whether or not the XML files associated with the waveform should be physically deleted from the system or merely that the internal data structures should be removed. This affords the ability to uninstall an application, freeing up internal resources and memory for other uses while still allowing the user to re-install the waveform as some point in the future, should it be required.

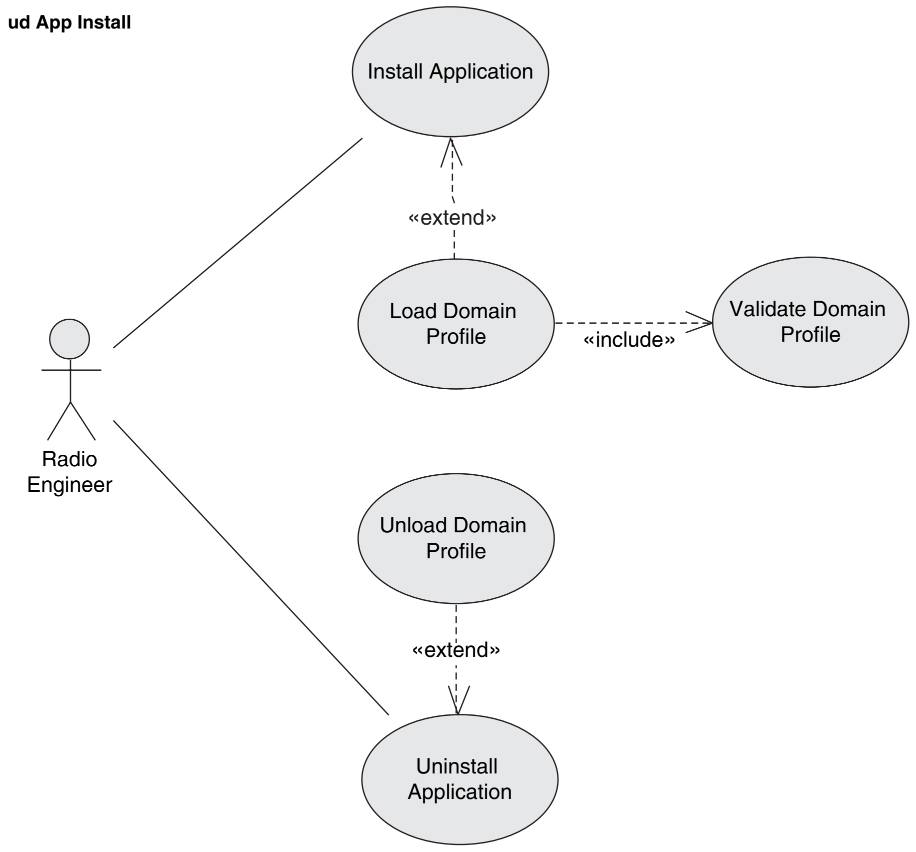


Figure 2.6. (Un)Install application use case

Installation of a waveform application is performed by reading and parsing the set of XML files that define the waveform. The Install application use case (Figure 2.6) creates the basic internal data structures that represent the set of components and hardware necessary to load the waveform application. The Load Domain Profile provides the parsed XML information necessary to create the internal model of the waveform profile. The Validate Domain Profile performs the validation parsing of the XML files against the Document Type Definitions (DTDs) specified in the SCA specification. The sequence diagram is shown in Figure 2.7.

Uninstalling an application entails the removal of any internal data structures representing the waveform profile information and, optionally, removing the XML files that form the waveforms profile from the file system. Note that the removal of the XML files is specified as a mandatory functional capability. However, this makes little sense because, once the XML files are removed, to install the application again the set of XML files must be re-loaded to the radio system.

Also, as noted in Figure 2.6, uninstallation of the waveform application has no effect on an instantiation of the waveform. Although, at first glance, the concept of having

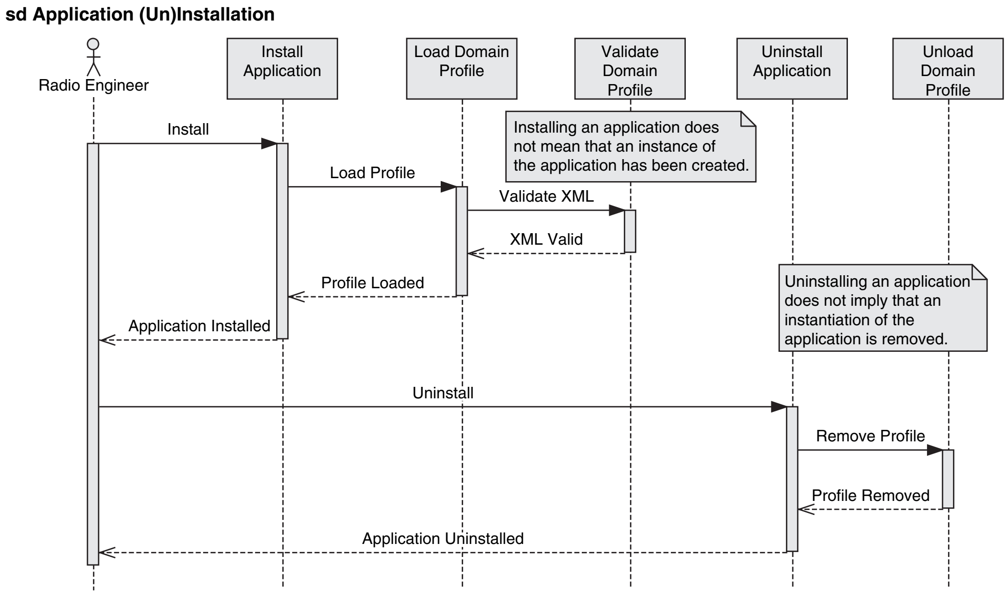


Figure 2.7. (Un)Install application sequence diagram

an instantiation of a waveform that has been uninstalled seems contradictory, it makes perfect sense. Once the waveform has been instantiated, it no longer requires the profile information for that waveform. That is because the profile information is used during the instantiation process. After the waveform is instantiated, all the components necessary have been loaded and device capacities have been allocated. Thus, once instantiated, the waveform may be started, stopped, and configured, and any other operation valid for an instantiated waveform may be performed without requiring access or reference to the profile information.

This allows a waveform to be instantiated, then uninstalled, freeing up memory and other support processing resources to allow, for example, another waveform to be installed. However, if the waveform is uninstalled, even though any instantiated waveforms will continue to function without the profile information, it does mean that no new instantiations of the waveform can be performed until the waveform is once again installed providing the necessary profile information.

Similarly, if the radio is shutdown, the Domain Manager is required to re-install any waveforms that were installed at the time of shutdown. If the waveform is uninstalled, even if there is an instance of the waveform at shutdown, the Domain Manager will not re-install the waveform.

2.4 Instantiate Application

Once an application has been installed, the necessary information required to instantiate the application, i.e. load the software components onto the requisite hardware and connect the software components together, is completed. The process of applying the Domain Profile information to the instantiation of a waveform is performed as part of the Create Application use case (Figure 2.8).

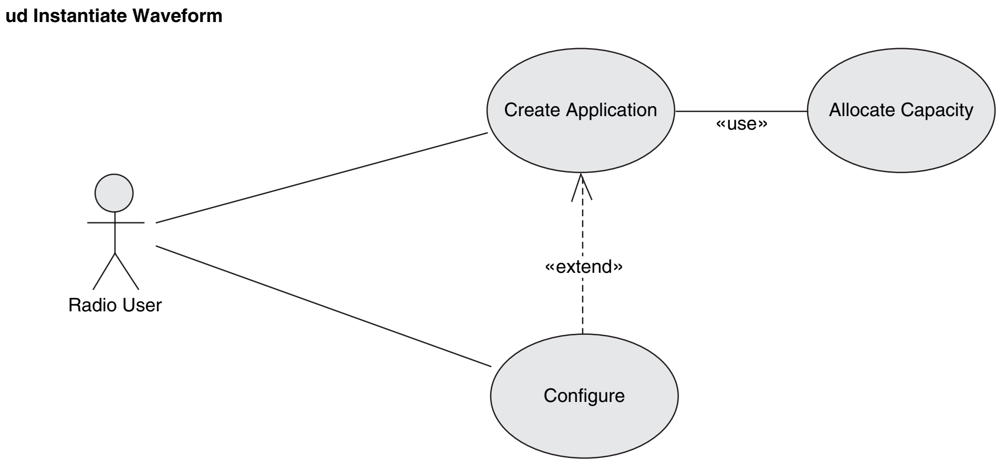


Figure 2.8. Waveform instantiation use cases

The Create Application process has two major aspects. The primary function is the process of loading the software components onto the hardware. However, a prerequisite function is the allocation of capacity on the devices where the software components are to be loaded (see Figure 2.9).

The Allocate Capacity use case checks to see that the Device on which a component is to be loaded has the available capacity required to load the component and, in the case of general purpose processors, execute the component.

For example, the component may require a specific amount of memory or a certain minimum clock cycle. If it is a FPGA load, it may require a certain type of FPGA or a minimum number of gates. Issues related to device capacity allocation are discussed in more detail in Chapter 6, where application instantiation is covered.

As part of the application instantiation, properties associated with the components and devices are often set to certain known or initial values. These values are defined in the Domain Profile XML files and reflected in the internal Domain Profile representation, if used. However, subsequent to the instantiation of the application, property values may be set via the configure operation. This allows the user to instantiate an application and then, prior to starting and using the application, customize properties based on specific requirements or needs at a given point in time. For example, although a waveform may be instantiated to use a default hopping rate, a different hopping rate may be required based on the current situation in which the waveform will be utilized. This is a situation when configuration subsequent to the instantiation of the waveform using the default property values may be used.

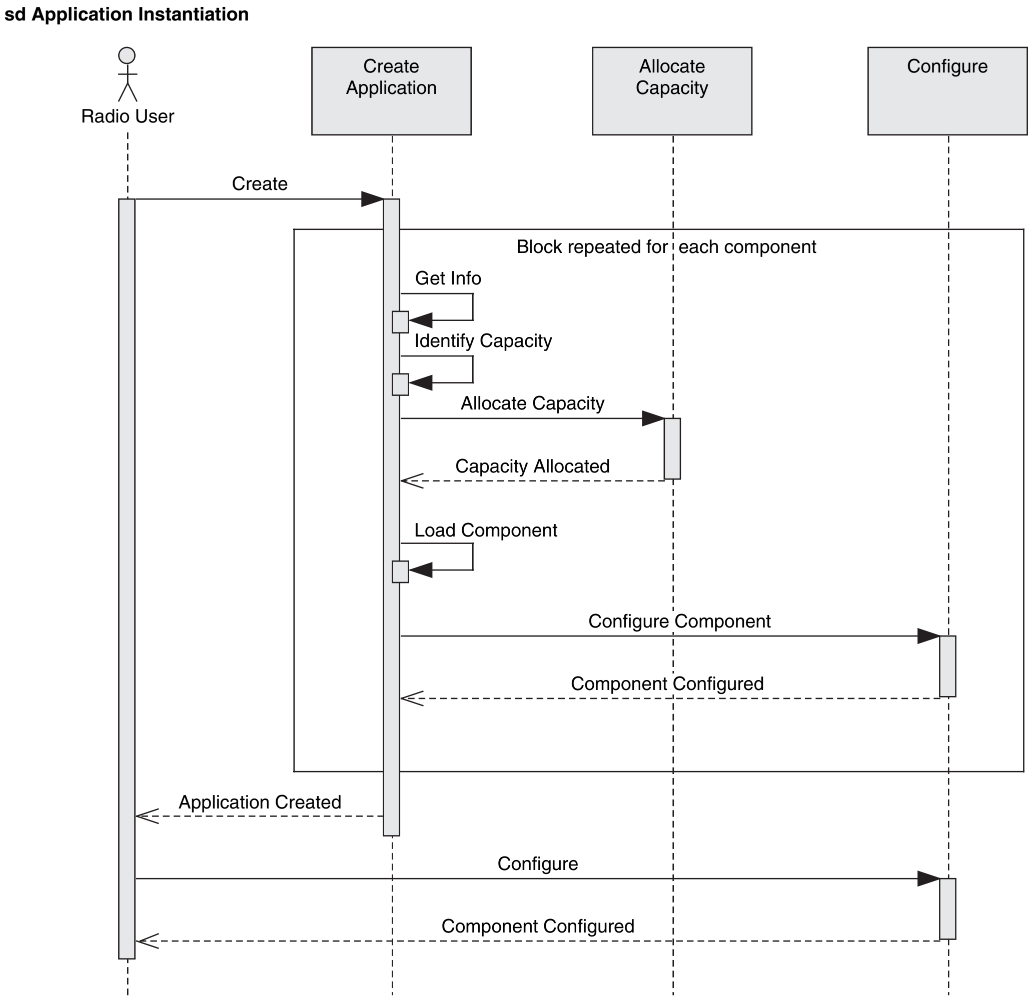


Figure 2.9. Waveform instantiation sequence diagram

2.5 Control Application

Once an application has been instantiated it is ready to be used and controlled by the user (see Figures 2.10 and 2.11). The user may Start or Stop the application, Query and Configure properties of the application or any of its components, or Release the application, which will remove the component software from the memory of the system.[1]

Starting the application is the process by which an application that has been already instantiated (i.e. loaded into memory and all the components have been connected), is

[1] *The removal of the waveform performed by the Release operation discussed in this section is the destruction of the instantiated waveform and* not *the uninstallation of the waveform profile as discussed in the previous section.*

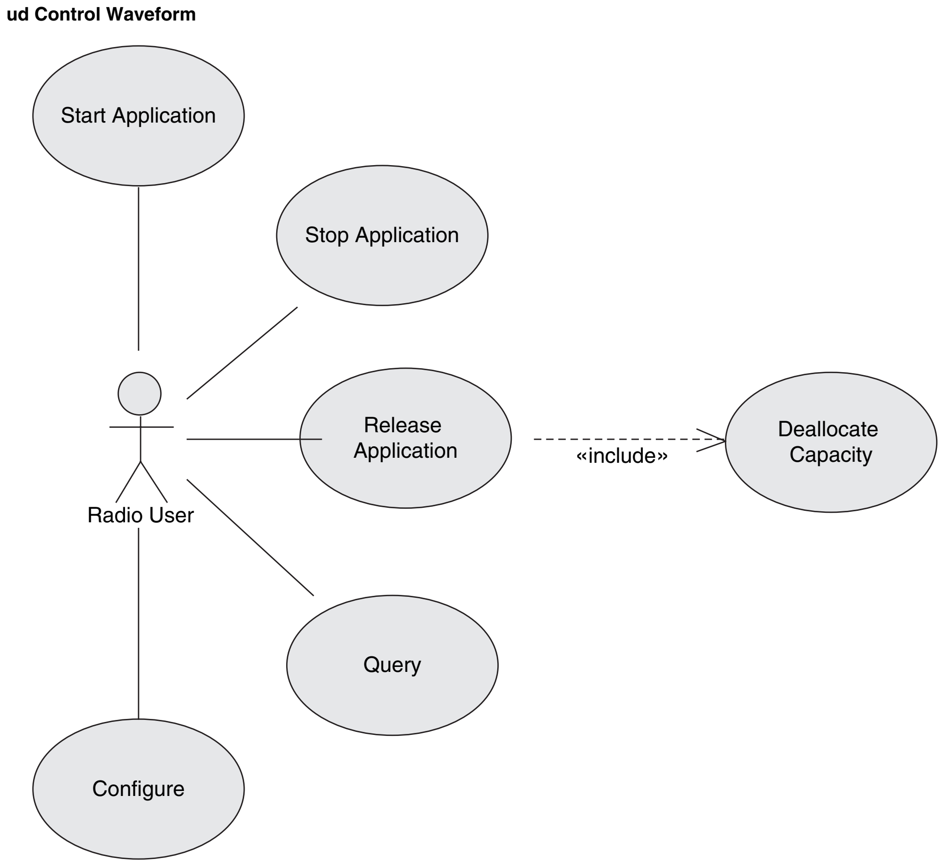


Figure 2.10. Radio system application control use cases

instructed to start performing waveform processing. Start essentially turns on the flow of the data through the system for that particular waveform.[2]

For waveform components loaded onto an executable device, such as a General Purpose Processor, this may seem a bit odd since, once loaded by the operating system, the executable device is running, i.e. receiving time slices by the operating system. The distinction is, however, that the executable image is in a quiescent state and is not processing the waveform data stream.

Use can also be made of the Stop command. This puts the waveform application back into a paused or quiescent state in which all the software components are still loaded and connected but the data stream for the waveform is not being processed.

Once the waveform is installed, Query and Configure operations may be performed on the waveform application and its components. These operations may be performed while

[2] *Start only applies to the specific instance for which the Start command is issued. Any other instances of the same waveform will remain in the state they were in prior to the Start call for another waveform instance.*

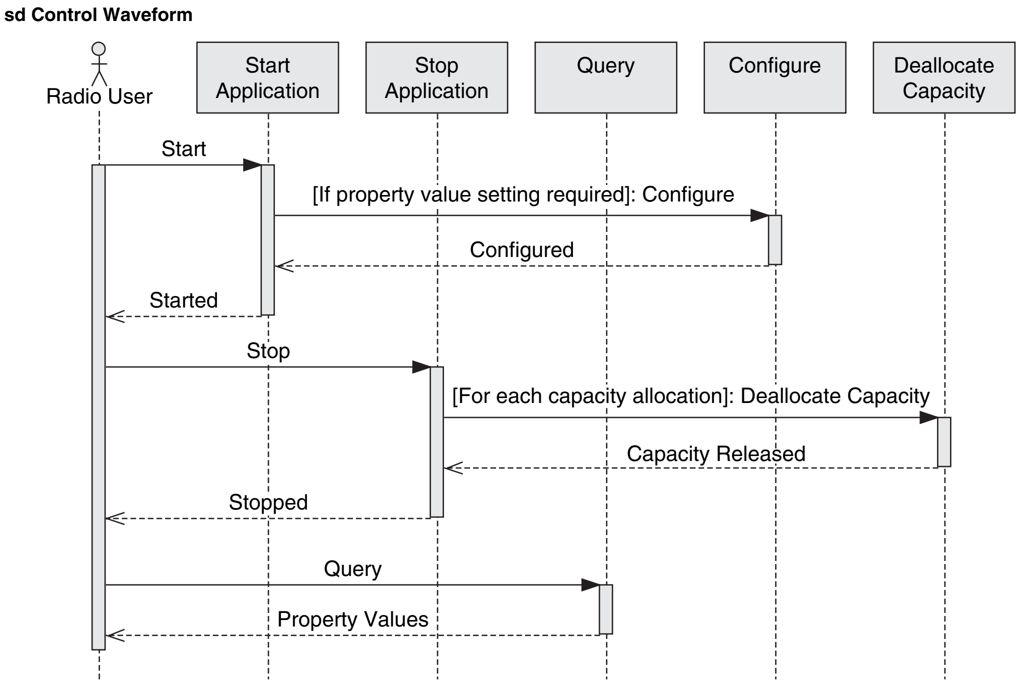


Figure 2.11. Radio system application control sequence diagram

the waveform application is started or stopped. The waveform implementor decides whether or not specific Query and Configure operations can be performed when the waveform is running. For example, certain property values may only be accessible while the waveform application is in a stopped state, whereas other properties may be accessible at any time.

Finally, an installed waveform may be released. This operation is performed on a waveform that is stopped and instructs the waveform application to remove itself from memory. Part of this process is the deallocation of capacity on those devices that had been hosting the components. This activity is crucial to the total operation of the system because it frees up device capacity making it available for use on subsequent waveform applications. Removing an instance of a waveform application does not affect any other instances of the same waveform currently in memory.

2.6 System Configuration

The final set of use cases (Figure 2.12) address the development of the XML files that contain the Domain Profile information necessary to install, instantiate, and operate a waveform. The Domain Profile XML files can roughly be categorized as being descriptions of the physical radio system (e.g. processors, antennas, amplifiers, etc.) or descriptions of the software components that comprise a waveform and their dependencies and relationships. Generally speaking, there is a single set of XML files that describe the physical radio system and one or more sets of XML files describing a waveform application.

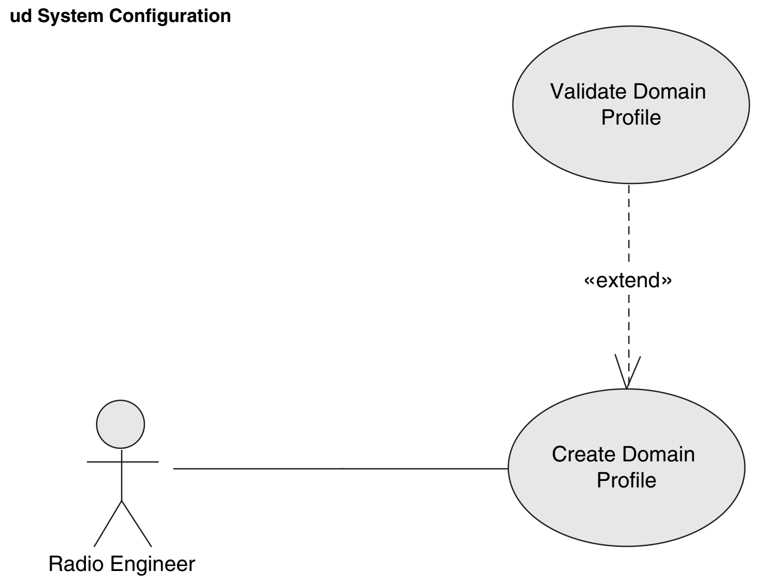


Figure 2.12. System configuration use cases

Certain components of a waveform may be referenced or used by multiple waveform implementations. Thus, an XML file may be used in more than one waveform application. The set of XML files and their organization are discussed in more detail in Part II. This section addresses the high-level tasks that must be performed in order to build the XML files containing the Domain Profile information.

This chapter has provided a use case view of the SCA architecture and organization. The following section provides an overview of the common services and functions provided or used by the SCA Core Framework.

3

General Requirements and Services

This chapter begins the SCA discussion by presenting several common services used by or implemented as part of the SCA Core Framework. The services provide common capabilities and functions utilized by several SCA components.

As with any complex system, there are several common services that provide support capabilities to the components that comprise the primary system. The SCA has several basic services that are identified. These services are:

1. Naming
2. Event
3. Log
4. FileSystem

These services are described in the following sections. Although not identified in the specification as a service, File System is also discussed in this chapter. The File System is a capability implemented by the Device Manager and the federated file system is implemented by the Domain Manager, having a common file system interface across multiple operating systems and processing platforms is a capability utilized by any component accessing data within the context of the SCA. Consequently, basic FileSystem operations are discussed in this chapter. The File System is discussed in the Device Manager chapter and the File Manager is discussed in the Domain Manager chapter (Chapters 5 and 6 respectively).

3.1 Non-Functional Requirements

There are several types of non-functional requirements specified in the SCA that identify limitations, preferences, and parameters that must be adhered to but do not define or contribute to the functional behavior of the SCA. These requirements are referenced in several areas of the SCA specification. They have been collected in this section as a convenience.[1]

3.1.1 General Requirements

The general requirements (Table 3.1) define a small set of features common to the entire SCA environment. For the most part they address operating system, object request broker, and other services or third party components of the system.

Table 3.1. General software requirements

Section	ID	Resp	Requirement
3.1.1	SR:1	OS	The OS shall provide the functions and options designated as mandatory by the AEP defined in Appendix B.
3.1.1	SR:2	OS/CF	The OS and related file systems shall support at a minimum a file name length of 40 characters and at a minimum a combined pathname/filename length of 1024 characters.
3.1.2.1	SR:3	%ORB	The OE shall use middleware that, at a minimum, provides the services and capabilities of minimumCORBA as specified by the OMG Document orbos/98-05-13, May 19, 1998.
3.1.3.2	SR:120	CF	Framework Control Interfaces shall be implemented using the CF IDL presented in Appendix C.
3.1.3.3	SR:506	CF	Framework Services Interfaces shall be implemented using the CF IDL presented in Appendix C.

The POSIX requirements levied by Appendix B of the SCA specification, as noted in requirement SR:1, are intended to provide a minimal set of mandatory operating system interfaces to enhance portability between platforms. This constrained set of interfaces facilitates portability for those waveforms that utilize operating system services as part of their implementation. Consequently, those waveforms that are predominantly, if not entirely, implemented by GPP code are the primary beneficiaries of the POSIX requirements. It should be noted that the POSIX interface requirements are intended to provide a common set of interfaces used by the application thereby improving the portability of waveform applications across operating systems and SCA platforms. It is entirely possible to achieve compliance with this requirement through the use of a POSIX library layer that implements the interfaces on top of an operating system. The applicability and usefulness of this requirement tends to lessen for application components written for a DSP and more so for FPGA implementations.[2]

The file name size requirement (SR:2) leaves some room for interpretation and has resulted in much discussion within the SCA community. The requirement states that the operating

[1] *In addition, there is a small set of requirements that identify security capabilities for an SCA system. These are discussed in Chapter 7.*

[2] *In SCA 3.x, a portion of the POSIX interfaces in defined for DSP implementations.*

system will support a minimum of 40 characters *and* a minimum of 1024 characters for the composite file name consisting of the directory path and file name. The requirement should be modified to specify a maximum as well since several operating systems enforce some maximum, usually 255 characters, file name length.

The last two requirements (SR:120, SR:506) specify that the core framework components are to implement the control and service interfaces specified by the Interface Definition Language (IDL). The IDL specification is provided in Appendix C of the SCA specification.

3.1.2 General Software Rules

There are two general software requirements within the SCA specification (Table 3.2). These requirements provide general guidance for the implementation of an SCA-compliant system.

Table 3.2. General software rules

Section	ID	Resp	Requirement
	SR:628	CF, SI	Software developed for an SCA-compliant product shall be developed in a standard higher order language, except as provided below, for ease in processor portability.
	SR:630	SI	Legacy software shall be interfaced to the core framework in accordance with this specification.

The use of a high-level language (SR:628) is intended to minimize the effort required to port a waveform or other functional component of an SCA-compliant system from one platform to another. The specification mentions that if there are performance requirements that can't be met using a high-order language, then assembly language can be utilized. This statement underscores the general orientation of the SCA specification towards a GPP. Although a variety of waveforms can be implemented in a GPP – as the throughput of the waveform in terms of bits per second that must be processed – the implementation approach is driven towards specialized processors such as a DSP or a (FPGA). These processors have significantly different architectures and implementation schemes impacting portability.

Any legacy software must be integrated into the SCA-compliant system (SR:630). This is typically accomplished through the use of an adapter or wrapper that provides an SCA-compliant set of interfaces to the SCA components and interfaces to the legacy software using the interfaces provided by that software.

3.1.3 Hardware Architecture Requirements

The hardware requirements (Table 3.3) do not provide a mandatory or even recommended hardware configuration. This is both a help and a hindrance to the proliferation of the SCA architecture and is symptomatic of a larger problem that is endemic to the software radio industry as a whole. In order to truly achieve interoperability between radio system components, a common set of hardware interface specifications must be established. To date, the SCA specification has focused on the software aspect of the SDR and little attention has been given to the standardization of hardware infrastructure components.

Table 3.3. Hardware requirements

Section	ID	Resp	Requirement
4.5.1	SR:632	SI	Each supplied hardware device shall be provided with its associated Domain Profile files as defined in section 3.1.3.4, Domain Profile.
4.5.2.1	SR:633	SI	Hardware critical interfaces shall be defined in Interface Control Documents that are available to other parties without restriction.
4.5.2.2	SR:634	SI	Hardware critical interfaces shall be in accordance with commercial or government standards, unless there are program performance requirements that require a non-standard interface.
4.5.2.2	SR:635	SI	If so required, the non-standard interface shall be clearly and openly documented to the extent that interfacing or replacement hardware can be developed by other parties without restriction.

XML files are used to specify the Domain Profile information (SR:632) for the hardware comprising an SCA-compliant system. The use of the term Domain Profile in the requirement is somewhat at odds with uses of the term in other sections of the specification. The term Domain Profile loosely refers to the set of XML files that provide the necessary descriptive information regarding the hardware and software components of an SCA-compliant system. Taken together, the set of hardware related XML files for a system forms the Domain Profile for the physical system. In addition, there are XML files that describe the essential software components of the Core Framework. Finally, one or more sets of XML files are used to describe the functional components of a waveform application and the resources, in terms of hardware and software, required to load and run the waveform application. The organization and construction of the Domain Profile is addressed in Part III of this book.

Since one of the primary objectives of the SCA is to promote interoperability, reconfigurability, and portability of hardware and software, each hardware device requires an Interface Control Document (SR:633). This specifies the hardware critical interfaces and provides the information necessary to enable a third party to use the hardware in an SCA-compliant system.

Any interfaces that are critical to the operation of the hardware are to be documented (SR:634) using industry or government standards; and any non-standard interfaces are to be documented (SR:635) such that the hardware may be integrated into third party systems.

3.1.4 Interface Organization

Figure 3.1 illustrates the collection of interfaces for the SCA Core Framework. The UML diagram in the figure shows the set of defined interfaces and their organization. As the figure shows, there is a group of common interfaces, LifeCycle, TestableObject, PortSupplier, and

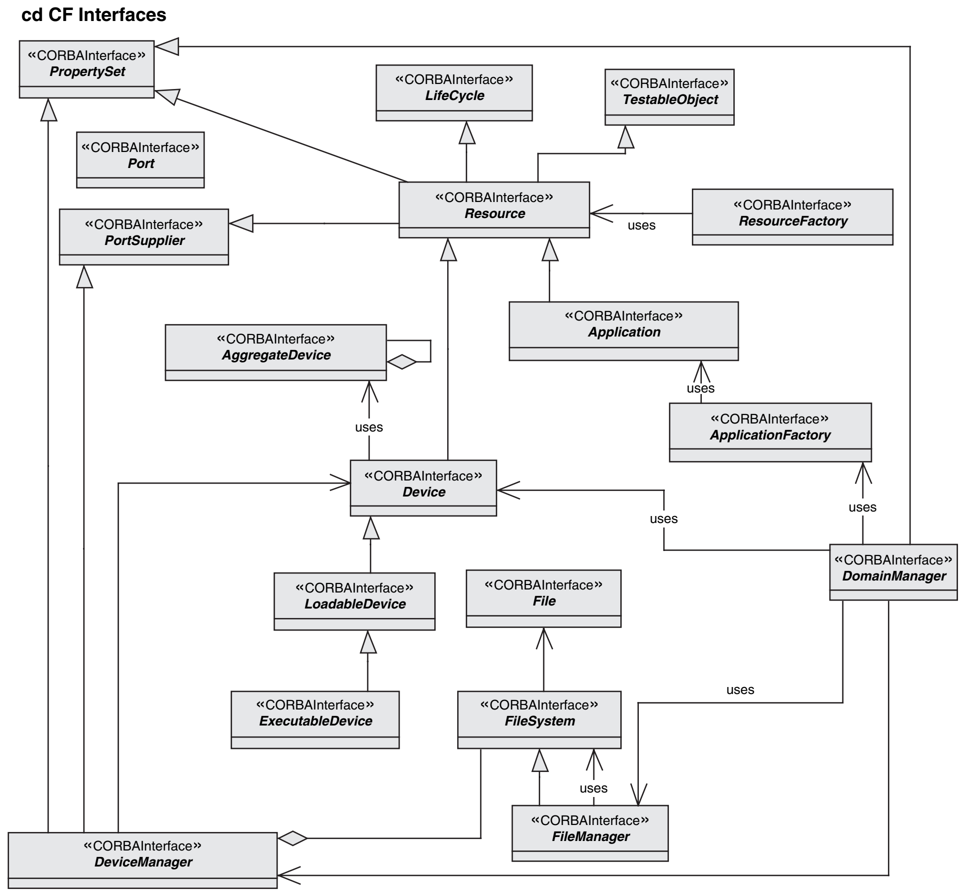

cd CF Interfaces
«CORBAInterface» PropertySet
«CORBAInterface» LifeCycle
«CORBAInterface» TestableObject
«CORBAInterface» Port
«CORBAInterface» Resource
«CORBAInterface» ResourceFactory
uses
«CORBAInterface» PortSupplier
«CORBAInterface» Application
«CORBAInterface» AggregateDevice
uses
«CORBAInterface» ApplicationFactory
uses
«CORBAInterface» Device
uses
uses
«CORBAInterface» DomainManager
«CORBAInterface» LoadableDevice
«CORBAInterface» File
uses
«CORBAInterface» ExecutableDevice
«CORBAInterface» FileSystem
uses
«CORBAInterface» FileManager
«CORBAInterface» DeviceManager

Figure 3.1. SCA IDL interface organization

PropertySet that converge into a single Resource Interface. Each of these base interfaces provides a distinct set of capabilities for the SCA environment. The Resource can be thought of as one of the fundamental building blocks of an SCA system.

The remaining chapters in Part I explore the above interfaces and the associated requirements to provide a view into the strategies and approach to development of a SCA implementation or Core Framework. The interfaces can be grouped into six high level abstractions:

- **Resource** – This abstraction incorporates key foundation interfaces and melds them into a collection at the Resource that provides a base set of functions for most of the Core Framework components.
- **Device** – This abstraction extends the Resource specification and provides the basic abstraction for all device interfaces and common behavior within an SCA system.
- **Device Manager** – The Device Manager provides several key services for a set of devices and provides the logical encapsulation of a device that is capable of performing actions such as booting a node in the system.

- **Application** – The Application interfaces specify the common control and data items for an application within the SCA-compliant radio system.
- **Domain Manager** – The Domain Manager defines the overall configuration and control behavior for the SCA radio system. Included within the domain management functions is the Application Factory which is responsible for the instantiation of the waveform application.
- **Port** – The Port abstraction provides a high-level formalism for connecting the different component resources within an application to perform the waveform processing.

The rest of the interfaces are subsumed within the above groups. For example, the File and FileSystem interfaces are implemented within a Device Manager providing the interface to the native file system provided by the Device Manager (Figure 3.2).

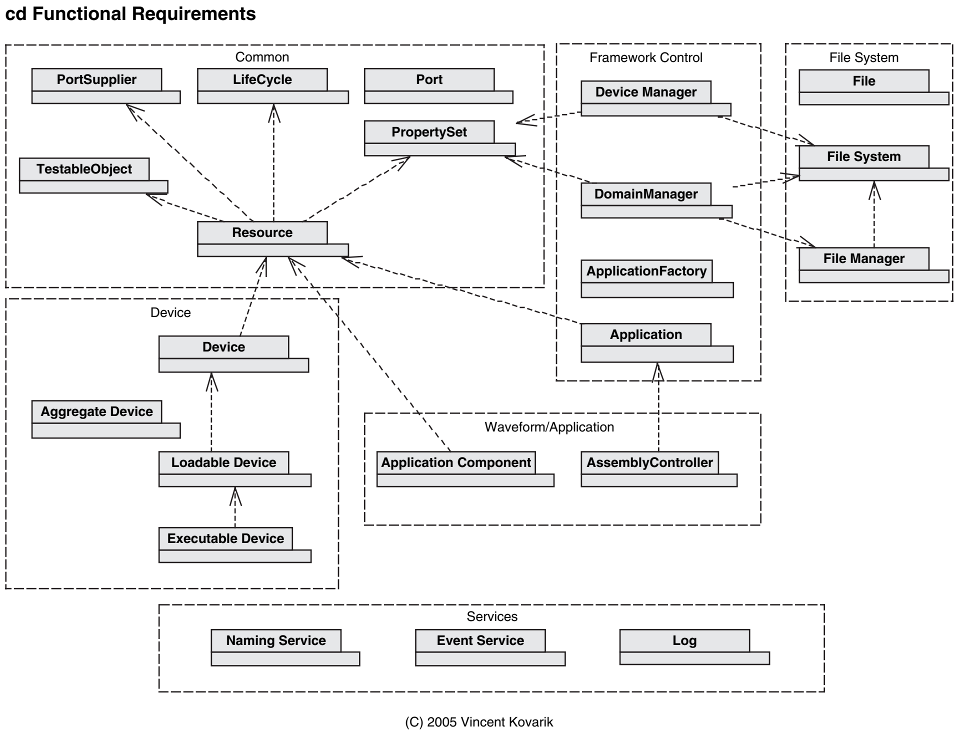


Figure 3.2. Functional organization of requirements

3.2 Name Service

The Name Service provides a central registry of services and applications within the SCA radio (Table 3.4). Since, in a distributed system, there is no guarantee that an application or resource will be located at the same load point between invocations, a Name Service is used as the method for applications to locate services and programs required by the application and as the point where an application registers the service(s) that it provides for use by other applications and components.

The premise of the naming service is very simple. It is a hierarchical name structure, much like that of a hierarchy of directories within the file system of an operating system. An application that wishes to make its services or capabilities available to other applications registers with the naming service using a well-known name. A simple example might be a program named FFT that accepts a sequence of binary data representing a digital sampling of an RF waveform over a given period of time and calculates the frequency components of the waveform. It would register with the naming service using a string such as '\FFT' and provide the information required by a client program to contact and make a call on the program. This information is typically referred to as a CORBA endpoint.

Thus, any program that wishes to use the FFT application need only request the naming service to provide the endpoint for the name '/FFT' and, if the name exists in the naming service, the endpoint is provided to the requestor.

Table 3.4. Naming Service requirements

Section	ID	Resp	Requirement
3.1.2.2.1	SR:4	CF, OV	A CORBA Naming Service shall be provided in the OE.
3.1.2.2.1	SR:5	CF, OV	A CORBA Naming Service supplied by an OE shall support the CosNaming CORBA module and its NamingContext interface's operations: bind, bind_new_context, unbind, destroy, and resolve.
3.1.2.2.1	SR:6	CF, OV	These operations shall meet the requirements of OMG Document formal/00-11/01: Interoperable Naming Service Specification.
3.1.2.2.1	SR:7	CF, OV	The 'kind' element of each NameComponent shall be "" (null string).

Providing a naming service (SR:4) is one area of functionality that goes beyond the Minimum CORBA requirement (SR:3). Basic naming service functionality requires a core set of operations (SR:5) as defined in the OMG specification (SR:6). The 'name' for a component is a hierarchical tree of strings, much like a directory structure or hierarchy of folders in a file system. The hierarchy starts with a Context or initial string defining the top level portion of the name. Each level of the hierarchical name, as with a file system path, is separated by a slash '/'. Each level of the hierarchical name is a NameComponent.

In a naming service, a NameComponent may have a portion of the NameComponent called a 'kind', which is an additional string appended to the base portion of the NameComponent with a period '.'. The SCA specifies (SR:7) the 'kind' portion of the NameComponent is to be null "". Therefore so, for an SCA implementation, a starting context might be 'CF' and there might be three items in the Naming Service as shown below and in Figure 3.3.

```
/CF/EventService

/CF/DomainManager

/CF/LogService
```

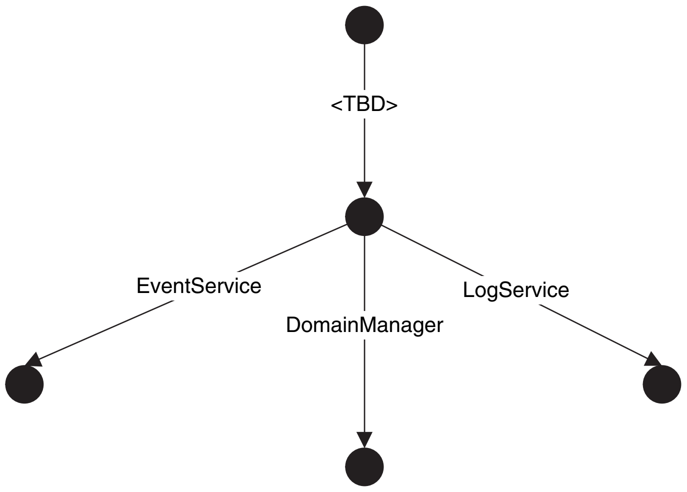


Figure 3.3. Naming Service context graph

Each of the entries in the Name Service has a reference to the component or program providing the service. An application requiring an AGC2 capability, for example, would query the name service for `/CF/LogService`. The Name Service would look up the name string provided by the requesting program and return the reference information, a CORBA endpoint, to the requestor. The requesting process would then use the endpoint to establish a communications link with the service provider.

3.3 Event Service

The Event Service provides a mechanism for processes within a distributed application to post data concerning events from the originator of the event to one or more event subscribers. It is, essentially, a simple, asynchronous message system between processes. The requirements related to the event channel are illustrated in Table 3.5. The event channel specification is a referenced OMG specification that is incorporated into the SCA specification. Thus, any third party supplier of an event channel implementation that complies with the OMG specification will work within an SCA Core Framework system.

The basic requirement establishes that a CORBA Event Service shall be provided (SR:61) as part of the operating environment. The requirement references the OMG standard specification for an Event Service. The required functionality of the Event Service is limited to the interfaces for the event producer and the event consumer (SR:62).

Figure 15 illustrates the relationship of requirements for the Event Service. The top two requirements establish the need for an Event Service to be provided (SR:61) and that the Event Service supports the 'Push' interfaces (SR:62). There are two styles of event channel implementation, push or pull. The pull implementation requires the consumer to request or 'pull' the event from the producer. The push implementation, conversely, implements the model in which the producer of an event pushes the data to the consumer when the event occurs.

Table 3.5. Event Service requirements

Section	ID	Resp	Requirement
3.1.2.4.1	SR:61	CF, OV	A CORBA Event Service (e.g. OMG's Event Service) shall be provided in the OE.
3.1.2.4.1	SR:62	CF, OV	The CORBA Event Service shall support Push interfaces (PushConsumer and PushSupplier) of the CosEventComm CORBA module as described in OMG Document formal/01-03-01: Event Service, v1.1.
3.1.2.4.1	SR:63	CF, DS, SI	A component (e.g. Resource, DomainManager, etc.) that consumes events shall implement the CosEventComm PushConsumer interface.
3.1.2.4.1	SR:64	CF, DS, SI	A component (e.g. Resource, Device, DomainManager, etc.) that produces events shall implement the CosEventComm PushSupplier interface and use the CosEventComm PushConsumer interface for generating the events.
3.1.2.4.1	SR:65	CF, DS, SI	A producer component shall handle all cases, without raising any exceptions outside of the producer component, due to the connections to a CosEventComm PushConsumer interface being nil or an invalid reference.
3.1.2.4.1	SR:66	CF	The Incoming Domain Management Channel name shall be 'IDM_Channel'.
3.1.2.4.1	SR:67	CF	The Outgoing Domain Management Channel name shall be 'ODM_Channel'.

The push model tends to be more applicable to systems where it is important for a consumer to be notified of an event as soon as it occurs. This is more appropriate for the types of operations and notification requirements between processes in an SCA system. Consequently, it is the mandatory implementation for the SCA OE and is explicitly specified in requirements SR:63 and SR:64. Associated with the push model is the requirement (SR:65) that no exceptions shall be raised outside of the producer in the event of a PushConsumer reference being nil or invalid. This requirement essentially states that, in the event that the consumer of a particular event is terminated, crashes, or otherwise ceases to exist, the reference to the consumer becomes invalid or nil and the producer shall not raise the exception. This requirement specifically references the connection to the PushConsumer being invalid or nil. However, in the larger sense, the event channel is a self-contained service that is not actively managed by a higher level entity of the system. Consequently, it could be argued that the event producer should handle all error conditions associated with the push channel.

Although the event service may be used by multiple applications components, only two event channels are identified as the mandatory requirements for the Core Framework: the Incoming Domain Management (IDM) channel (SR:66) and the Outgoing Domain Management (ODM) event channel (SR:67).

The Event Service is a key mechanism for notifying the Domain Manager of changes in state of devices, software modules, and other components of the SCA environment. The notification of state changes in SCA components is a primary application of the Event Service. The standard Event Types are described in the next section.

3.3.1 Event Types

There are three basic Event Types defined for the SCA. These Event Types can be categorized as state change events, removal of an object from the SCA event, or insertion of an object to the SCA OE event. These Event Types are illustrated in Figure 3.4.

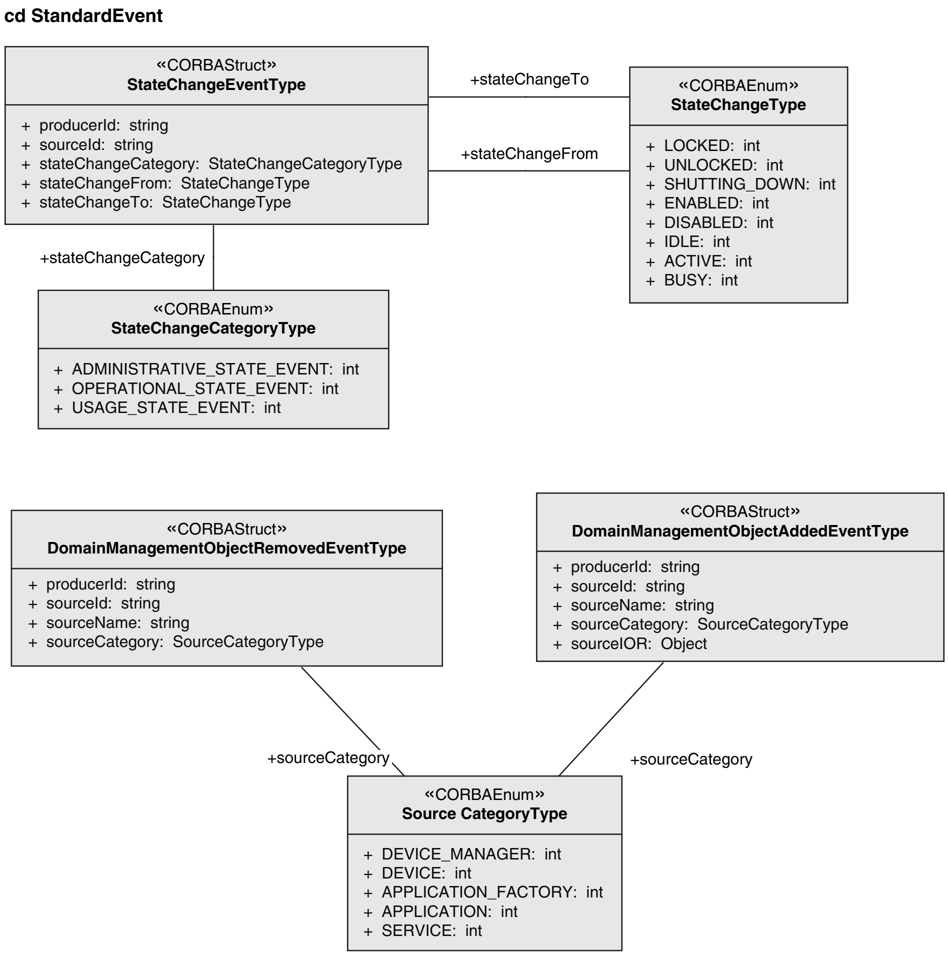


Figure 3.4. Event channel types

The StateChangeEventType messages represent the bulk of the messages sent via the Event Service. These messages notify components, typically the Domain Manager, of changes to component state within the system. The producerID and sourceID provide information about the source of the message, the producer, and the component that has changed state, the source. The message contains three additional fields describing the category of the state change, the new state value, and the original state value.

The state change categories are not explicitly defined but generally can be thought of as state changes that are the result of system control, initialization, and shutdown, for example. These would generally be classified as Administrative events. Changes in state due to functional behavior and logic, e.g. initiated processing thus changing the state to Active, would be Usage state changes. Finally, changes due to capacity changes, e.g. the capacity is allocated or is increased as part of the processing, are Operational state changes.

The remaining two event types, the DomainManagementObjectRemovedEventType and the DomainManagementObjectAddedEventType provide notification when components are added to the system or removed from the system. Because components, hardware or software, may be added or removed dynamically, events are produced to announce the arrival or departure of the component.

As with the state change event, a producerID and a sourceID are part of the message. The added and removed events also have a sourceName parameter that provides a string name for the object entering or leaving the system. A sourceCategory is also provided identifying the type of component that is the source of the event. Currently this is constrained to the values shown in Figure 16, Device Manager, Device, Application Factory, Application, and Service. It has been proposed that the sourceCategory be expanded to include additional component types: the Resource, Loadable Device and Executable Device are potential additions. The Loadable Device and Executable Device are extensions of the Device; so, it could be argued that they could utilize the current Device category. However, it would be more precise to specify which type of device was entering or leaving the system. Adding Resource to the collection of types would be valuable because there are often multiple libraries and other code components that provide some capability in support of a waveform implementation. These resources may be part of an application for a shared library supporting multiple applications as a component on which the applications are dependent. Consequently, being able to support the notification of a Resource entering or leaving the system would be valuable.

3.4 Log Service

SCA version 2.2 specified a Log Service as part of the specification. With the release of SCA 2.2.1, the Log Service has been deleted from the SCA and the SCA specification now references the OMG Lightweight Log specification. The basic functions and IDL of the OMG Lightweight Log Service and the SCA Log Service are the same. The differences between the two versions lie in the organization of the IDL. The Lightweight Log Service organizes the IDL into logical groups as shown in Figure 3.5. In the SCA 2.2 Log Service, all the IDL operations are grouped into a single interface.

The Log Service requirements are organized into four subsets following the organization of the interface IDL. The first subset is Log Status. Log Status requirements address the common log capabilities across all users of the Log Service. The capabilities address the ability to retrieve status information regarding the state of the Log Service including the number of records currently in the log, the availability of the log, and the operational state of the log, to name a few. These requirements are subsumed within each of the following roles as

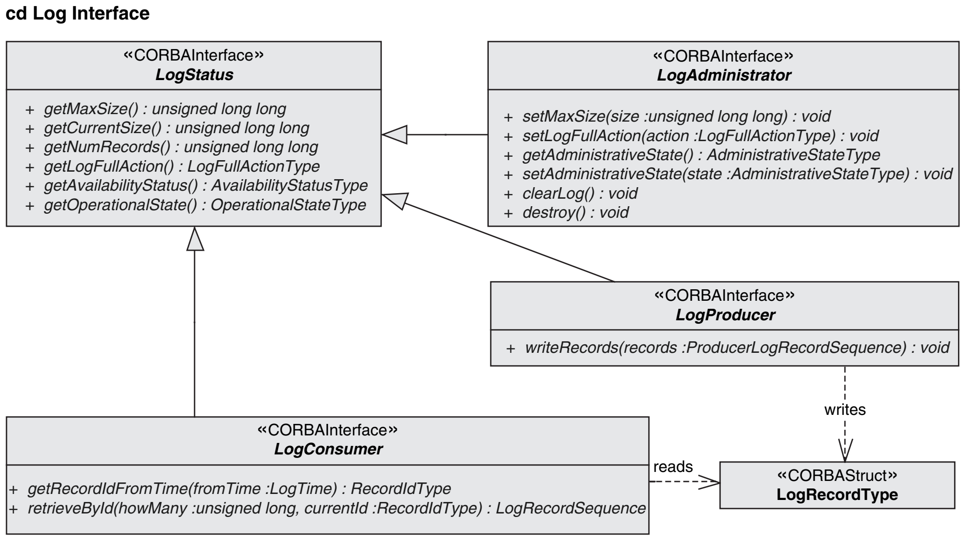

cd Log Interface
«CORBAInterface»
LogStatus
+ getMaxSize() : unsigned long long
+ getCurrentSize() : unsigned long long
+ getNumRecords() : unsigned long long
+ getLogFullAction() : LogFullActionType
+ getAvailabilityStatus() : AvailabilityStatusType
+ getOperationalState() : OperationalStateType
«CORBAInterface»
LogAdministrator
+ setMaxSize(size :unsigned long long) : void
+ setLogFullAction(action :LogFullActionType) : void
+ getAdministrativeState() : AdministrativeStateType
+ setAdministrativeState(state :AdministrativeStateType) : void
+ clearLog() : void
+ destroy() : void
«CORBAInterface»
LogProducer
+ writeRecords(records :ProducerLogRecordSequence) : void
writes
«CORBAInterface»
LogConsumer
+ getRecordIdFromTime(fromTime :LogTime) : RecordIdType
+ retrieveById(howMany :unsigned long, currentId :RecordIdType) : LogRecordSequence
reads
«CORBAStruct»
LogRecordType

Figure 3.5. CORBA Lightweight Log Service interfaces

capabilities of the users of the log. The **`LogAdministrator`** encapsulates the requirements associated with the configuration and management of the Log Service. Therefore, it allows the log full action to be set as well as clearing and removing the log from service. The **`LogProducer`** requirements define the capabilities implemented by a component that writes entries to the log. Finally, the **`LogConsumer`** allows two basic methods of retrieving entries from the log, by record Id or by time.

Figure 3.6 illustrates the Lightweight log implementation aspects. The log implementation implements the CORBA calls for all aspects of the log. On the client side, only those IDL interfaces required would be included. Typically a client would include the LogStatus interfaces and then either the LogProducer or LogConsumer interfaces or both, depending on the type of client access required. The applications that will manage the log would include the LogStatus and LogAdministrator interfaces.

3.4.1 Data Types

The Log Service has several data types defined for use within the Log Service as well as the format of the log records that are created by the **`LogProducer`**, stored by the Log Service, and retrieved by the **`LogConsumer`**.

3.4.2 Exceptions

Several exceptions are defined for the log interfaces. The IDL definition of the exceptions is shown in the following two sections.

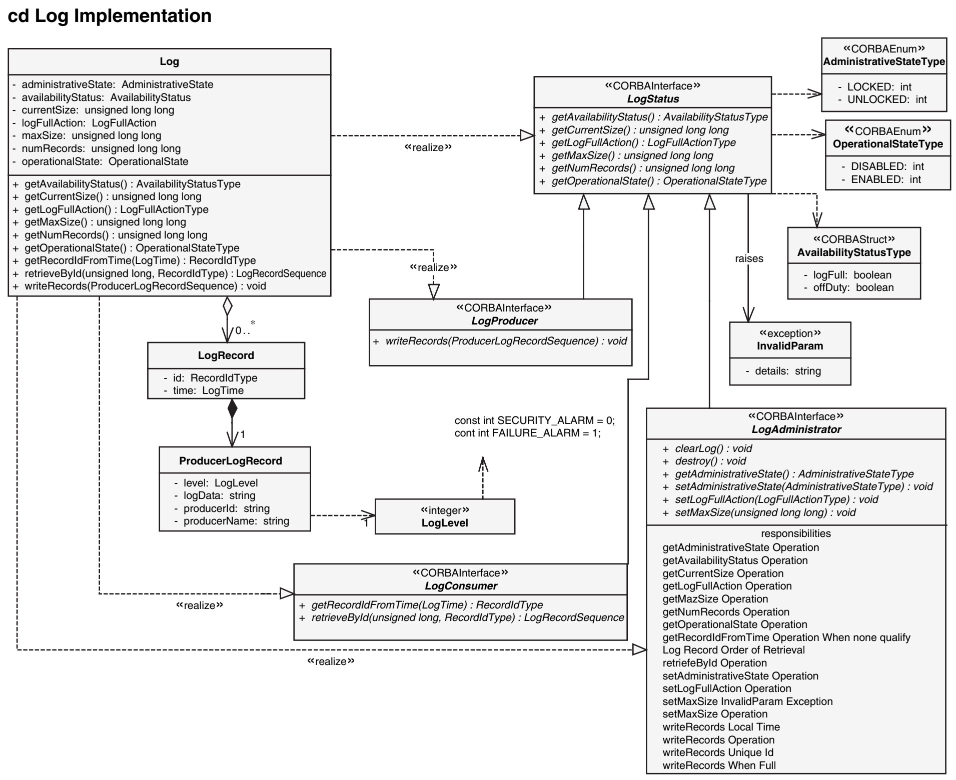

cd Log Implementation
Log
- administrativeState: AdministrativeState
- availabilityStatus: AvailabilityStatus
- currentSize: unsigned long long
- logFullAction: LogFullAction
- maxSize: unsigned long long
- numRecords: unsigned long long
- operationalState: OperationalState
+ getAvailabilityStatus() : AvailabilityStatusType
+ getCurrentSize() : unsigned long long
+ getLogFullAction() : LogFullActionType
+ getMaxSize() : unsigned long long
+ getNumRecords() : unsigned long long
+ getOperationalState() : OperationalStateType
+ getRecordIdFromTime(LogTime) : RecordIdType
+ retrieveById(unsigned long, RecordIdType) : LogRecordSequence
+ writeRecords(ProducerLogRecordSequence) : void
«realize»
«CORBAInterface» LogStatus
+ getAvailabilityStatus() : AvailabilityStatusType
+ getCurrentSize() : unsigned long long
+ getLogFullAction() : LogFullActionType
+ getMaxSize() : unsigned long long
+ getNumRecords() : unsigned long long
+ getOperationalState() : OperationalStateType
«CORBAEnum» AdministrativeStateType
- LOCKED: int
- UNLOCKED: int
«CORBAEnum» OperationalStateType
- DISABLED: int
- ENABLED: int
«CORBAStruct» AvailabilityStatusType
- logFull: boolean
- offDuty: boolean
raises
«exception» InvalidParam
- details: string
«realize»
«CORBAInterface» LogProducer
+ writeRecords(ProducerLogRecordSequence) : void
0..*
LogRecord
- id: RecordIdType
- time: LogTime
1
ProducerLogRecord
- level: LogLevel
- logData: string
- producerId: string
- producerName: string
1
«integer» LogLevel
const int SECURITY_ALARM = 0;
cont int FAILURE_ALARM = 1;
«CORBAInterface» LogAdministrator
+ clearLog() : void
+ destroy() : void
+ getAdministrativeState() : AdministrativeStateType
+ setAdministrativeState(AdministrativeStateType) : void
+ setLogFullAction(LogFullActionType) : void
+ setMaxSize(unsigned long long) : void
responsibilities
getAdministrativeState Operation
getAvailabilityStatus Operation
getCurrentSize Operation
getLogFullAction Operation
getMazSize Operation
getNumRecords Operation
getOperationalState Operation
getRecordIdFromTime Operation When none qualify
Log Record Order of Retrieval
retriefeById Operation
setAdministrativeState Operation
setLogFullAction Operation
setMaxSize InvalidParam Exception
setMaxSize Operation
writeRecords Local Time
writeRecords Operation
writeRecords Unique Id
writeRecords When Full
«realize»
«CORBAInterface» LogConsumer
+ getRecordIdFromTime(LogTime) : RecordIdType
+ retrieveById(unsigned long, RecordIdType) : LogRecordSequence
«realize»

Figure 3.6. Lightweight Log implementation

InvalidLogFullAction

The InvalidLogFullAction exception is raised when an application call attempts to set the logFullAction of the log to a value that is not defined.

```
exception InvalidLogFullAction {
        string Details;
        };
```

InvalidParam

The InvalidParam exception is raised when a parameter supplied to a call is incorrect.

```
exception InvalidParam {
        string details;
        };
```

The organization of the LogService data types is shown in Figure 3.7.

Each of the data types and structures in Figure 3.7 are described individually in the following sections.

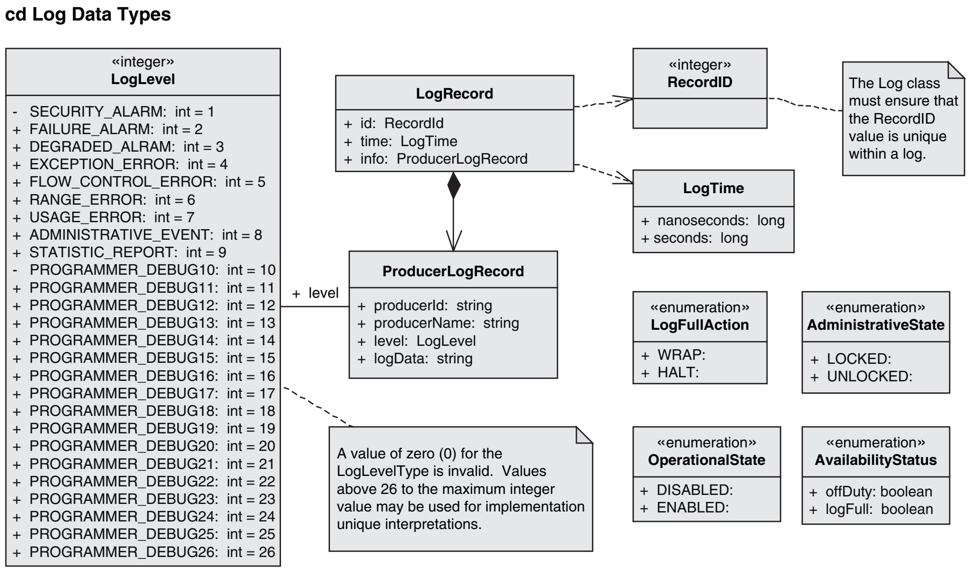

cd Log Data Types
«integer»
LogLevel
- SECURITY_ALARM: int = 1
+ FAILURE_ALARM: int = 2
+ DEGRADED_ALRAM: int = 3
+ EXCEPTION_ERROR: int = 4
+ FLOW_CONTROL_ERROR: int = 5
+ RANGE_ERROR: int = 6
+ USAGE_ERROR: int = 7
+ ADMINISTRATIVE_EVENT: int = 8
+ STATISTIC_REPORT: int = 9
- PROGRAMMER_DEBUG10: int = 10
+ PROGRAMMER_DEBUG11: int = 11
+ PROGRAMMER_DEBUG12: int = 12
+ PROGRAMMER_DEBUG13: int = 13
+ PROGRAMMER_DEBUG14: int = 14
+ PROGRAMMER_DEBUG15: int = 15
+ PROGRAMMER_DEBUG16: int = 16
+ PROGRAMMER_DEBUG17: int = 17
+ PROGRAMMER_DEBUG18: int = 18
+ PROGRAMMER_DEBUG19: int = 19
+ PROGRAMMER_DEBUG20: int = 20
+ PROGRAMMER_DEBUG21: int = 21
+ PROGRAMMER_DEBUG22: int = 22
+ PROGRAMMER_DEBUG23: int = 23
+ PROGRAMMER_DEBUG24: int = 24
+ PROGRAMMER_DEBUG25: int = 25
+ PROGRAMMER_DEBUG26: int = 26
LogRecord
+ id: RecordId
+ time: LogTime
+ info: ProducerLogRecord
«integer»
RecordID
The Log class must ensure that the RecordID value is unique within a log.
LogTime
+ nanoseconds: long
+ seconds: long
+ level
ProducerLogRecord
+ producerId: string
+ producerName: string
+ level: LogLevel
+ logData: string
«enumeration»
LogFullAction
+ WRAP:
+ HALT:
«enumeration»
AdministrativeState
+ LOCKED:
+ UNLOCKED:
A value of zero (0) for the LogLevelType is invalid. Values above 26 to the maximum integer value may be used for implementation unique interpretations.
«enumeration»
OperationalState
+ DISABLED:
+ ENABLED:
«enumeration»
AvailabilityStatus
+ offDuty: boolean
+ logFull: boolean

Figure 3.7. Log Service data types

3.4.3 Types

OperationalStateType

Several items define the state of the Log Service. The **`OperationalState`** identifies if the log is available. If the value is **`ENABLED`**, the log is available to support log producers and log consumers. Also, if the log is full and the LogFullAction has been set to `HALT`, then the logFull value is set to true.

```
enum OperationalStateType {
     DISABLED,
     ENABLED
};
```

The enumeration OperationalStateType defines the log states of operation. When the log is `ENABLED` it is fully functional and is available for use by log producer and log consumer clients. A log that is `DISABLED` has encountered a runtime problem and is not available for use by log producers or log consumers. The internal error conditions that cause the log to set the operational state to `ENABLED` or `DISABLED` are implementation specific.*

AdministrativeStateType

The AdministrativeState specifies whether log records may be written to the log. If the value is **`LOCKED`**, the log will not accept new records from log producers. However, log records may still be read by log consumers or deleted by the LogAdministrator.

```
enum AdministrativeStateType {
     LOCKED,
     UNLOCKED
};
```

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

AvailabilityStatusType

The AvailabilityStatus indicates whether or not the log is available for use. The **`offDuty`** value, if true, indicates that the log is not available for use. A log is unavailable is the **`OperationalState`** is **`DISABLED`** or the **`AdministrativeState`** is `LOCKED`.

```
struct AvailabilityStatusType {
      boolean offDuty;
      boolean logFull;
};
```

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

LogFullActionType

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

```
enum LogFullActionType {
     WRAP,
     HALT
```

The LogFullAction indicates the behavior to be followed when the number of records written to the log reaches the maximum available for the log. If the value is **`WRAP`**, then the log behaves as a circular queue and the next record written by a producer is inserted at the beginning of the log. If the value is **`HALT`**, then the log stops accepting records and sets the appropriate **`AvailabilityStatus`** property.

LogLevelType

The LogLevelType is an enumeration type that is utilized to identify log levels. Each of the LogLevel is represented as a bit value within the enumeration type. This allows multiple LogLevel attributes to be specified within a single data value through the logical OR operation (Table 3.6).

```
enum LogLevelType {
     FAILURE_ALARM,
     DEGRADED_ALRAM,
     EXCEPTION_ERROR,
     FLOW_CONTROL_ERROR,
     RANGE_ERROR,
     USAGE_ERROR,
        ADMINISTRATIVE_EVENT,
        STATISTIC_REPORT,
        PROGRAMMER_DEBUG1,
        PROGRAMMER_DEBUG2,
        PROGRAMMER_DEBUG3,
        PROGRAMMER_DEBUG4,
        PROGRAMMER_DEBUG5,
        PROGRAMMER_DEBUG6,
        PROGRAMMER_DEBUG7,
        PROGRAMMER_DEBUG8,
        PROGRAMMER_DEBUG9,
        PROGRAMMER_DEBUG10,
        PROGRAMMER_DEBUG11,
        PROGRAMMER_DEBUG12,
        PROGRAMMER_DEBUG13,
        PROGRAMMER_DEBUG14,
        PROGRAMMER_DEBUG15,
        PROGRAMMER_DEBUG16
};
```

Table 3.6. LogLevelType values

Attribute	Description
`FAILURE_ALARM`	Identifies a major failure within the LogProducer.
`DEGRADED_ALARM`	Identifies the LogProducer as being in a degraded state. The originator of the message may still function but in limited capacity.
`EXCEPTION_ERROR`	Identifies that an exception was raised by the LogProducer subsequent to writing the LogRecord.
`FLOW_CONTROL_ERROR`	An error has occurred in managing the flow of data between the LogProducer and another component. The LogProducer may be the sender or the receiver of the data stream that encountered the error.
`RANGE_ERROR`	A value provided to the LogProducer through an API call, such as a configure Property call.
`USAGE_ERROR`	This attribute identifies an error in the usage of a parameter or call to set the state of the LogProducer. An example is incorrectly setting the AdministrativeState
`ADMINISTRATIVE_EVENT`	The LogProducer has changed its AdministrativeState value.
`STATISTIC_REPORT`	The LogProducer is providing some operational statistic.
`PROGRAMMER_DEBUG1`	This attribute is open for definition and use by the system and application developer.

PROGRAMMER_DEBUG2	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG3	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG4	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG5	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG6	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG7	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG8	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG9	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG10	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG11	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG12	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG13	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG14	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG15	This attribute is open for definition and use by the system and application developer.
PROGRAMMER_DEBUG16	This attribute is open for definition and use by the system and application developer.

LogTimeType

This type provides the time format used by the log to time-stamp LogRecords. The log implementation is required to produce time-stamps that are compatible with the POSIX-defined `time_t type`.

```
struct LogTimeType {
      long seconds;
      long nanoseconds;
};
```

LogRecordType

The LogRecordType defines the format of the LogRecords as stored in the log. The ‘info’ field is the ProducerLogRecord that is written by a client to the log. The basic structure in the log is the LogRecord definition. There are three attributes that comprise an instance of a LogRecord: i) the RecordId, ii) the LogTime, and iii) the ProducerLogRecord. The first two items are relatively straightforward. The RecordId is an integer data item that provides a unique value for the record within the log. Note that it is the responsibility of the log implementor to ensure that the RecordId is unique within the log. However, it is permissible to have duplicate RecordId values across instances of a log. The LogTime contains a time-stamp consisting of two parts, nanoseconds and seconds, and map to the POSIX timespec structure.

```
 struct LogRecordType {
        RecordIdType id;
        LogTimeType time;
        ProducerLogRecordType info;
};
 typedef sequence <LogRecordType> LogRecordSequence;
```

3.4.4 LogStatus Operations

The LogStatus set of interfaces provides access to the status information of the log. This interface is inherited by all other log interfaces. The LogStatus requirements address the accessibility of configuration and status information associated with the log (see Figure 3.7). These capabilities are, in turn, flowed down to the various roles, administrator, producer, and consumer.

Table 3.7. LogStatus requirements

Section	ID	Resp	Requirement
3.1.2.3.3.5.1.4	SR:29	CF, OV	The getMaxSize operation shall return the integer number of bytes that the log is capable of storing.
3.1.2.3.3.5.3.4	SR:33	CF, OV	The getCurrentSize operation shall return the current size of the log storage in bytes. (i.e. if the log contains no records, getCurrentSize will return a value of 0 (zero).)
3.1.2.3.3.5.4.4	SR:34	CF, OV	The getNumRecords operation shall return the current number of LogRecords contained in the log.
3.1.2.3.3.5.5.4	SR:35	CF, OV	The getLogFullAction operation shall return the log's log full action setting.
3.1.2.3.3.5.7.4	SR:37	CF, OV	The getAvailabilityStatus operation shall return the current availability status of the log.
3.1.2.3.3.5.8.4	SR:38	CF, OV	The getAdministrativeState operation shall return the current administrative state of the log.
3.1.2.3.3.5.10.4	SR:40	CF, OV	The getOperationalState operation shall return the current operational state of the log.

getMaxSize

The getMaxSize operation allows an SCA application to retrieve the upper limit of the storage capabilities of the log.

```
 unsigned long long getMaxSize ();
```

The size of a log is specified in bytes. So, when requesting the maximum size of a log (SR:29), the value returned will be the number of bytes that the log is capable of storing.

getCurrentSize

The current size of the log (SR:33) provides the number of bytes allocated to the entries currently in the log.

```
unsigned long long getCurrentSize ();
```

getNumRecords

The number of records (SR:34) can also be requested. Note that, because the actual message data is an unbounded string, there is no direct mapping from the number of records in a log to the number of bytes currently allocated to the records. Nor is there a method of calculating the number of records that can be stored in the log based on the number of available bytes of storage.

```
unsigned long long getNumRecords ();
```

getLogFullAction

The LogFullAction (SR:35) specifies the action to be performed by the log when it is full. This interface allows the action to be retrieved by an SCA application.

```
LogFullActionType getLogFullAction ();
```

getAvailabilityStatus

This interface provides the ability to retrieve the AvailabilityStatus of the log (SR:37).

```
AvailabilityStatusType getAvailabilityStatus ();
```

getAdministrativeState

This interface provides the ability to retrieve the AdministrativeState of the log (SR:38).

```
AdministrativeStateType getAdministrativeState ();
```

getOperationalState

The OperationalState may be retrieved through this interface (SR:40).

```
OperationalStateType getOperationalState ();
```

3.4.5 LogAdministrator Operations

The LogAdministrator interface provides access to set the operational parameters of the log. The LogAdministrator requirements are described in Table 3.8.

The LogAdministrator is the entity that configures and manages the log. This is usually an SCA DeviceManager although there is no explicit reference prohibiting another entity within the framework, such as the DomainManager, from managing a log.

Table 3.8. Log Service configuration and management requirements

Section	ID	Resp	Requirement
3.1.2.3.3.5.2.3	SR:30	CF, OV	The setMaxSize operation shall set the maximum size of the log measured in number of bytes.
3.1.2.3.3.5.2.5	SR:31	CF, OV	The setMaxSize operation shall raise the InvalidParam exception if the size parameter passed is less than the current size of the log.
3.1.2.3.3.5.2.5	SR:32	CF, OV	The setMaxSize operation shall raise the InvalidParam exception if the input size parameter is greater than the storage space available to the log.
3.1.2.3.3.5.6.3	SR:36	CF, OV	The setLogFullAction operation shall set the action taken by a log, when its maximum size has been reached, to the value specified in the action parameter.
3.1.2.3.3.5.9.3	SR:39	CF, OV	The setAdministrativeState operation shall set the administrative state of the log.
3.1.2.3.3.5.14.3	SR:55	CF, OV	The clearLog operation shall delete all records from the log.
3.1.2.3.3.5.14.3	SR:56	CF, OV	The clearLog operation shall set the current size of the log storage to zero.
3.1.2.3.3.5.14.3	SR:57	CF, OV	The clearLog operation shall set the current number of records in the log to zero.
3.1.2.3.3.5.14.3	SR:58	CF, OV	The clearLog operation shall set the logFull availability status element to false.
3.1.2.3.3.5.15.3	SR:59	CF, OV	The destroy operation shall release all internal memory and/or storage allocated by the log.
3.1.2.3.3.5.15.3	SR:60	CF, OV	The destroy operation shall tear down the component (i.e. it is released from the CORBA environment).

setMaxSize

The LogAdministrator may set the maximum size of the log in bytes (SR:30). Although it seems that the size of the log may be dynamically modified using the setMaxSize operation, the operation is monotonic, i.e. once instantiated the setMaxSize operation may only increase the size of the log. If the request is made to reduce the size of the log through the setMaxSize operation, an exception is raised (SR:31). Also, if the amount of storage requested by the setMaxSize operation exceeds the available storage, then an exception is raised (SR:32) as well.

```
void setMaxSize (
     in unsigned long long size
)
raises (InvalidParam);
```

It should be noted that the wording of the requirements assumes that the storage requested by the setMaxSize operation is allocated in its entirety. Practically speaking, it is feasible to view the MaxSize of the log as just that: the maximum size that the log may use. This would allow storage space to be allocated incrementally with the error being returned only when the requested allocation exceeds available memory. This strategy is analogous to disk storage algorithms with size limits imposed by the operating system. A maximum amount of storage for a particular user or process may be specified but it is not allocated at once. Instead, 'chunks' of memory are allocated when more storage is required and each allocation request checks to see if it is still within the bounds specified by the MaxSize value.

setLogFullAction

When the log reaches the maximum amount of storage allocated, it follows the behavior specified by the LogFull value (SR:36). The two possible behaviors are **`WRAP`** and **`HALT`**.

```
void setLogFullAction (
     in LogFullActionType action
);
```

As implied by the logFull types, if the LogFull action is set to `HALT`, the log will stop storing entries to the log. This is an important aspect to be aware of in your design because the log does not return an error or raise an exception to the producer that is writing a record to the log. So, unless the status of the log is checked, it is possible for a log producer to write records to the log that are ignored by the log because it is full. If the LogFull action is set to `WRAP`, then the log behaves like a circular queue and starts overwriting the log records at the start of the log.

setAdministrativeState

This interface enables the ability to set the AdministrativeState of the log (SR:39).

```
void setAdministrativeState (
     in AdministrativeStateType state
);
```

clearLog

The clearLog operation removes all of the entries within the log (SR:55). If all of the records are deleted successfully, the logSize and the numberOfRecords are set to zero (SR:56 and SR:57). It also sets the LogFull availability status property to false (SR:58)

```
void clearLog ();
```

destroy

The destroy operation initiates the termination of a LogService. When a destroy operation is received, it releases all the internal memory used by the log (SR:59), tears down the log program (SR:60), and terminates.

```
void destroy ();
```

3.4.6 LogProducer Operations

The LogProducer is any software component of an SCA system that needs to write to the log. For most systems, this includes all implementations of the Core Framework components such as Resources, Devices, DeviceManagers, and the Domain Manager.

ProducerLogRecordType

The ProducerLogRecord class defines the content of the data portion of the LogRecord. It has four components: i) producerId; ii) producerName; iii) level; and iv) logData. The producerId is a string value that identifies the component that created the log entry. The producerName is a string value providing a human understandable name for the producer of the log entry. The level is an integer that identifies the type of log entry according the definitions of the LogLevel described earlier. The logData is a string that contains the actual body of the log record.

```
typedef unsigned long long RecordIdType;
struct ProducerLogRecordType {
      string producerId;
      string producerName;
      LogLevelType level;
      string logData;
};
typedef sequence <ProducerLogRecordType>
    ProducerLogRecordSequence;
```

Logproducers format log records as defined in the structure ProducerLogRecordType (see Tabel 3.9).

- producerID: This field uniquely identifies the source of a log record. The value is the component's identifier and is unique for each SCA Resource and Core Framework component with the Domain.
- producerName: This field identifies the producer of a log record in textual format. This field is assigned by the log producer, and thus is not unique within the Domain (e.g. multiple instances of an application will assign the same name to the ProducerName field).
- level: The level field can be used to classify the log record according to the LogLevelType.
- logData: This field contains the informational message being logged.

Table 3.10 shows the requirements associated with a LogProducer. There are several general requirements associated with the LogProducer and several sub-requirements associated with the writeRecords operation.

Table 3.9. ProducerLogRecordType fields

Attribute	Type	Notes
`producerId`	public: *string*	This attribute uniquely identifies the source of a log record. The value is unique within the Domain. The DomainManager and ApplicationFactory are responsible for assigning this value.*
`producerName`	public: *string*	This attribute identifies the producer of a log record in textual format. This field is assigned by the log producer, thus is not unique within the Domain (e.g. multiple instances of an application will assign the same name to the ProducerName field).*
`level`	public: LogLevelType	This attribute identifies the type of message being logged as defined by the type LogLevelType.*
`logData`	public: *string*	This attribute contains the informational message being logged.

Table 3.10. Logproducer requirements

Section	ID	Resp	Requirement
3.1.2.3.1	SR:9	CF, OV	Log producers shall implement a configure property with an Id of '`PRODUCER_LOG_LEVEL`'. The `PRODUCER_LOG_LEVEL` configure property provides the ability to 'filter' the log message output of a log producer.
3.1.2.3.1	SR:10	CF, OV	The type of this property shall be a LogLevelSequence.
3.1.2.3.1	SR:11	CF, OV	Only the messages that contain an enabled log level shall be sent by a log producer to a log. Log levels that are not in the LogLevelSequence are disabled.
3.1.2.3.1	SR:12	CF, OV	Log producers shall use their component identifier in the producerId field of the ProducerLogRecord.
3.1.2.3.1	SR:13	CF, OV	Log producers shall operate normally in the case where the connections to a log are nil or an invalid reference.
3.1.2.3.1	SR:14	CF, OV	Log producers shall output only those log records that correspond to enabled LogLevelType values.
3.1.2.3.3.5.11.3	SR:41	CF, OV	The writeRecords operation shall add each log record supplied in the records parameter to the log.
3.1.2.3.3.5.11.3	SR:42	CF, OV	When there is insufficient storage to add one of the supplied log records to the log, and the LogFullAction is set to `HALT`, the writeRecords method shall set the availability status logFull state to true.

Table 3.10. Continued

Section	ID	Resp	Requirement
3.1.2.3.3.5.11.3	SR:43	CF, OV	The writeRecords operation shall write the current local time to the time field of each record written to the log.
3.1.2.3.3.5.11.3	SR:44	CF, OV	The writeRecords operation shall assign a unique record Id to the Id field of the LogRecord.
3.1.2.3.3.5.11.3	SR:45	CF, OV	Log records accepted for storage by the writeRecords shall be available for retrieval in the order received.

The LogProducer must have a `LOG_PRODUCER_LEVEL` property (SR:9) that is used to store the log level as a LogLevelSequence (SR:10). The log level is used to filter which messages are actually written to the log. Only those messages which contain a log level that is present in the `LOG_PRODUCER_LEVEL` property will be sent to the log (SR:11). All other log messages are not written.

When constructing the ProducerLogRecord, the producerId field is set to the componentId of the LogProducer (SR:12). The LogProducer does not output any log record where the logLevel is not one of the entries in the `LOG_PRODUCER_LEVEL` (SR:14). Also, if the LogProducer does not have a connection to a log, the LogProducer must continue to operate (SR:13). In other words, if a waveform component also produces log records as part of its operation but does not have – or loses – a connection to a log, it must continue to perform the primary functions.

writeRecords

The writeRecords operation writes log records to the log. As seen in the IDL below, the call is defined as a CORBA oneway call. This means that the LogProducer process making the call does not wait for a function call return. Even though there is no return value, normally the caller would wait until there is a positive acknowledgement that the function has completed on the server side of the call. A oneway call does not wait for this return. However, the oneway still waits for the TCP acknowledgement (or some other verification method if a different underlying protocol is used) to verify that the call was received successfully by the server before continuing execution. Thus, there is still the opportunity for performance impacts due to latency. So, it is usually advisable to issue the writeRecords call in a LogProducer in a thread. Then, if there is a problem on the receiving side of the call, the normal processing performed by the LogProducer may continue uninterrupted.

```
oneway void writeRecords (
      in ProducerLogRecordSequence records
);
```

Each record in the records field is added to the log sequentially (SR:41). During the call, if there is insufficient storage to add the next record and the LogFullAction is set to `HALT`, then the logFull value is set to true (SR:42) and the call is terminated. Any records that may have been added prior to the insufficient storage condition will remain in the log. The record being processed at the time the logFull condition was reached, along with any subsequent records in the sequence, are discarded.

As the log records are processed by the writeRecords operation, the current local time is stored in the time field of each record (SR:43) and a unique record Id is generated and stored in the record Id field (SR:44). The order in which records are written to the log is the order in which they may be retrieved (SR:45).

3.4.7 LogConsumer Operations

The LogConsumer interface to the log allows client programs to retrieve records from the log. As noted in the previous section on the writeRecords call, log records are available for retrieval in the order they were written. However, there are several options available to specify at what point in the log the retrieval should start and how many records should be retrieved.

As the requirements listed in Table 3.11 show, the basic method for retrieving log records is via the retrieveById operation. The record Id to be used in the retrieval call must first be obtained through the getRecordFromTime call.

Table 3.11. LogConsumer requirements

Section	ID	Resp	Requirement
3.1.2.3.3.5.12.3	SR:47	CF, OV	The getRecordIdFromTime operation returns the record Id of the first record in the log with a time-stamp that is greater than, or equal to, the time specified in the fromTime parameter. If the log does not contain a record that meets the criteria provided, then the RecordType returned shall correspond to the next record that will be returned in the future.
3.1.2.3.3.5.12.4	SR:48	CF, OV	If the log does not contain a record that meets the criteria provided, then the RecordIdType returned shall correspond to the next record that will be recorded in the future.
3.1.2.3.3.5.13.3	SR:49	CF, OV	The retrieveById operation shall set the inout parameter currentId to the LogRecord Id of the record following the last record in the LogRecordSequence returned.
3.1.2.3.3.5.13.3	SR:50	CF, OV	If the record sequence returned exhausts the log records, then the currentId parameter shall be set to the LogRecordId of where the log will resume writing logs on the next write.
3.1.2.3.3.5.13.4	SR:51	CF, OV	The retrieveById operation shall return a LogRecordSequence that begins with the record specified by the currentId parameter.

Table 3.11. Continued

Section	ID	Resp	Requirement
3.1.2.3.3.5.13.4	SR:52	CF, OV	The number of records in the LogRecordSequence returned by the retrieveById operation shall be equal to the number of records specified by the howMany parameter, or the number of records available if the number of records specified by the howMany parameter is greater than the number of records available.
3.1.2.3.3.5.13.4	SR:53	CF, OV	If the record specified by currentId does not exist, the retrieveById operation shall return an empty list of LogRecords and leave the currentId unchanged.
3.1.2.3.3.5.13.4	SR:54	CF, OV	If the Log is empty, or has been exhausted, the retrieveById operation shall return an empty list of LogRecords and leave the currentId unchanged.

getRecordIDFromTime

This operation returns the Id of the first record in the log that has a time-stamp that is equal to or greater than the time provided in the call (SR:47). If the time-stamps of all the entries in the log are less than the time provided, then the log returns the record Id that will be assigned to the next log record (SR:48).

```
RecordIdType getRecordIdFromTime (
      in LogTimeType fromTime
);
```

It should be noted that it is possible to issue the getRecordIDFromTime call with a time that is in the future. If such a call is issued and the log returns the next Id to be assigned to a log record, then the next record written to the log may be earlier than the time requested. For example, if the log contains records written through Id 100 and the time-stamp for that record is (1 000 000 000) and the call is made with a time of (1 100 000 000) or 1000 seconds in the future, the log will return the next Id to be assigned to a log record which, in this case is 101. It is entirely possible for another LogProducer to write multiple records prior to the time of (1 100 000 000). Thus, when the process that issued the original `getRecordIDFromTime` actually retrieves the records using the `retrieveById` call below, it will actually receive records that are earlier than the time requested. Although this scenario does not cause any operational problems, the developer should be aware that such a situation may occur.

retrieveById

The retrieveById operation retrieves a sequence of records from the log starting with the record identified by the value in the currentId parameter (SR:51). The number of records to be retrieved is specified by the howMany parameter and a sequence of log records is returned.

```
LogRecordSequence retrieveById (
      inout RecordIdType currentId,
      in unsigned long howMany
);
```

As shown in the IDL, the currentID is an inout parameter. If the log has additional records subsequent to the last record returned in the log record sequence, then the currentId is set to the Id of the next log record in the log (SR:49). If, however, the number of records to be retrieved exceeds the number of records in the log (i.e. the howMany parameter exceeds the number of records subsequent to the first record to be retrieved), the currentId parameter is then set to the Id of the next record that will be written to the log (SR:50). The number of records in the log record sequence is equal to the value of the howMany parameter if the number of records specified were available for retrieval. If the number of records available is less than what was requested via the howMany parameter, then the number of records in the log record sequence returned reflects the number that were available (SR:52). This allows the caller to check if the retrieval process encountered the end of the log prior to retrieving the number of records requested. If the record specified by the currentId parameter does not exist, then an empty log record sequence is returned and the currentId parameter is not changed (SR:53). Similarly, if the log is empty when the retrieveById call is received, an empty sequence of log records is returned and the currentID parameter is not changed (SR:54). This can happen if one process issues a getRecordIDFromTime call and receives a log record Id as a return value. Then a second process issues the clearLog call which will remove all the records from the log. Then the original process issues the retrieveById call on a log that is now empty.

3.5 FileSystem

One of the key capabilities of the SCA is to provide a federated file system across a heterogeneous processor and operating system environment. The FileSystem provides the fundamental interfaces to a file system and is the common abstraction across all file systems. It should be noted that the SCA FileSystem is not a file system that is implemented as part of an operating system. Instead, it is a common set of file and file system interfaces defined in the CORBA IDL that provide the expected interfaces and operations of a file system. When running on a general purpose processor on top of a standard operating system, the SCA FileSystem is, in fact, an application process running on the system that acts as a façade to the actual file system.

The question might be asked: ‘Why bother to define such file system?’ The benefit is gained in two key areas. First, for those hardware configurations that might not have a native file system, such as flash memory residing on a VME board, an SCA file system can be developed that provides the standard file system interfaces enabling external applications to access the flash as if it were a disk system. Second, in the context of a heterogeneous collection of hardware, there may be multiple operating systems with different file system characteristics. Having a single, consistent set of interfaces allows SCA applications to access the storage devices across all hardware and operating systems using a single set of calls. Finally, through the File Manager of the Domain Manager the aggregate set of FileSystems appear as a single, federated file system. So, not only can data storage be accessed across multiple hardware and operating systems using a single set of interfaces, the entire set of file systems can be accessed as a single entity.

The rest of this chapter discusses the three basic divisions that provide this service, the FileSystem, File, and FileManager. Since access to a file is obtained through the file system, the FileSystem willbe discussed first, followed by the File, and the FileManager will be discussed in the Domain Management Chapter (Chapter 6).

The SCA FileSystem provides the essential interface to access and manipulate files as an abstract entity. The intent of the SCA FileSystem, as with many of the other components of the Core Framework, is to provide an implementation-neutral method of interacting with the lower level services, software, and hardware in the system. In this case, a common set of interfaces for interacting with the file system is provided on the underlying hardware and operating software.

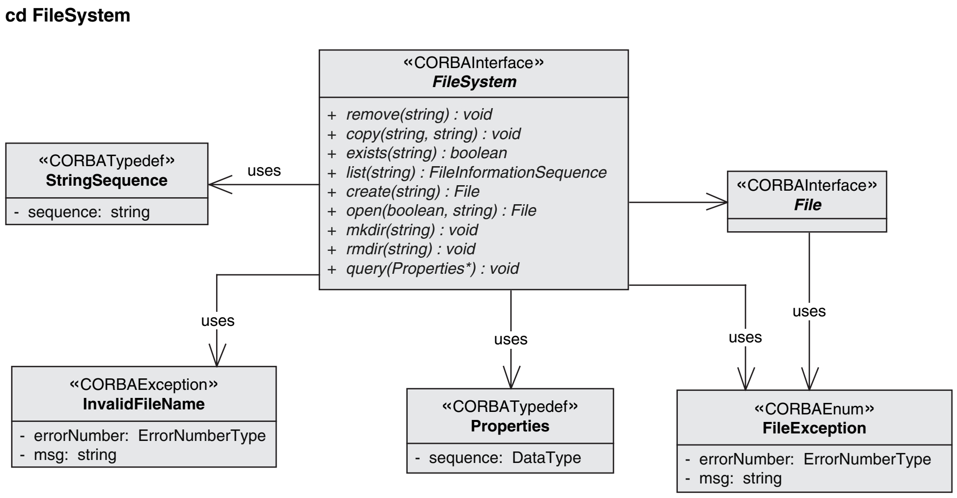

cd FileSystem
«CORBAInterface» FileSystem
+ remove(string) : void
+ copy(string, string) : void
+ exists(string) : boolean
+ list(string) : FileInformationSequence
+ create(string) : File
+ open(boolean, string) : File
+ mkdir(string) : void
+ rmdir(string) : void
+ query(Properties*) : void
«CORBATypedef» StringSequence
- sequence: string
«CORBAInterface» File
«CORBAException» InvalidFileName
- errorNumber: ErrorNumberType
- msg: string
«CORBATypedef» Properties
- sequence: DataType
«CORBAEnum» FileException
- errorNumber: ErrorNumberType
- msg: string
uses

Figure 3.8. FileSystem interface

Figure 3.8 illustrates the FileSystem interfaces, related interfaces, data types, and exceptions used or referenced by the FileSystem. As can be seen from the figure, essential file manipulation, creation, deletion, and other operations are provided.

3.5.1 Exceptions

UnknownFileSystemProperties

The UnknownFileSystemProperties exception is raised in cases where one or more property names used in a configure or query call are not recognized by the FileSystem.

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

3.5.2 Types and Constants

There are several data types and constants defined for the file system. Several string constants are related to FileSystem information and others provide information about each file managed by the FileSystem. The constants are shown in the following subsections.

SIZE

This string constant is used to refer to the current size of the FileSystem. It maintains the total size in octets of the entire file system. This is a summation of each of the size of each file plus any additional overhead incurred by the FileSystem. In most cases this value reflects the summation of file sizes of the underlying operating system file system.

```
const string SIZE = "SIZE";
```

AVAILABLE_SPACE

This string constant is used to refer to the total available space of the FileSystem. In most cases this corresponds to the available space remaining on a disk drive or other storage volume.

```
const string AVAILABLE_SIZE = "AVAILABLE_SPACE";
```

CREATED_TIME

This string refers to the file property that maintains the time that the file was created. The create time reference is stored as the number of seconds since midnight, January 1, 1970.

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

MODIFIED_TIME

This string refers to the file property that maintains the time that the file was last modified. The modified time reference is stored as the number of seconds since midnight, January 1, 1970.

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

LAST_ACCESS_TIME

This string refers to the file property that maintains the time that the file was accessed. The access time reference is stored as the number of seconds since midnight, January 1, 1970.

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

What appears to be missing from the general file information is information as to whether the file is currently in use, i.e. it is locked. This can lead to seemingly errant conditions in the operation of the FileSystem. For example, if an SCA process opens a file system, the operating system associates the file Id of the file with the process that opened it. Another process may ‘delete’ the file, removing it from the directory, but because the file is in use by another process, it would not actually remove the file until the file Id was released by the process that owns it. This ensures that the process using the file would not abort because the file was deleted from under it.

Since the SCA FileSystem does not maintain information about the mapping of a file to a process, it has no method of inhibiting the deletion of the file and no method for integrating the information from the lower level file system. Consequently, it is possible to start an SCA process that accesses a file, delete the file, and have the process continue to access the file normally.

Also, it is unclear as to the utility of maintaining the last access time. Unless there is some method of binding the file to a process, knowing when the file was last accessed is of limited value. This area of the SCA is continuing to evolve.

3.5.3 Types

FileType

The FileType enumeration is used to identify whether the file within a FileSystem is a directory, a basic file, or another FileSystem. The use of the `FILE_SYSTEM` enumeration value enables support for nested FileSystems.

```
enum FileType {
    PLAIN,
    DIRECTORY,
    FILE_SYSTEM
};
```

FileInformationType

Each file entry within the FileSystem has a set of descriptive information. This information provides common information normally available in a file's directory entry. The information is specified in the FileInformationType below.

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

The name of the file is stored in the 'name' field. The 'kind' field identifies the type of file as defined by the FileType above. The current size of the file is maintained in the 'size' field. This field is stored as the number of octets that comprise the file. Finally, an instance of the SCA 'Properties' is part of the FileInformationType `struct` that provides the capability to store additional descriptive information about the file.

FileInformationSequence

The collection of FileInformationType instances for each file is managed in the FileInformationSequence.

```
typedef sequence <FileInformationType>
    FileInformationSequence;
```

The specific requirements are listed in Table 3.12.

The FileInformationType defines the `struct` that maintains the minimum information required for a file (SR:528). The time values associated with the creation (SR:529),

Table 3.12. FileType requirements

Section	ID	Resp	Requirement
3.1.3.3.2.3.3	SR:528	CF	At a minimum, the FileSystem shall support name, kind, and size information for a file.
3.1.3.3.2.3.6	SR:529	CF	The value for created time shall be `unsigned long long` and measured in seconds from midnight, January 1, 1970.
3.1.3.3.2.3.7	SR:530	CF	The value for modified time property shall be `unsigned long long` and measured in seconds from midnight, January 1, 1970.
3.1.3.3.2.3.8	SR:531	CF	The value for last access time property shall be unsigned long long and measured in seconds midnight, January 1, 1970.

modification (SR:530), and last access (SR:531) times are maintained for each file as discussed earlier in this section.

3.5.4 Operations

create

The create operation creates a new file in the FileSystem with the name specified in the call. The create call allocates the initial space to be used for the file on the underlying operating system file system, creates an entry in the native file system, creates an instance of an SCA File, populates the instance with data about the file, and returns the instance.

Exceptions raised by the create call are the InvalidFileName if the string name provided is invalid and the FileException if the file already exists or another file error occurs.

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

The create method instantiates a new instance of an SCA File within the FileSystem (SR:547) with the name specified in the filename argument (see Table 3.13). The SCA File created is mapped to an underlying file on the hardware and operating software of the system. If successful, an instance of a File is returned. The create operation returns a File reference upon creation of the file requested (SR:548). If a file with the same name already exists, then a FileException is raised (SR:550). If the name provided in the create call is invalid, then an InvalidFileName exception is raised (SR:551). If any other error is encountered, then a null File reference is returned (SR:549).

copy

The copy operation copies the contents of a source file to a target file within a FileSystem. The IDL for the call is shown below. The source and destination file names are provided as

Table 3.13. FileSystem create file requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.5.3	SR:547	CF	The create operation shall create a new File based upon the provided fileName attribute.
3.1.3.3.2.5.5.4	SR:548	CF	The create operation shall return a File component reference to the opened file.
3.1.3.3.2.5.5.4	SR:549	CF	The create operation shall return a null file component reference if an error occurs.
3.1.3.3.2.5.5.5	SR:550	CF	The create operation shall raise the CF FileException if the file already exists or another file error occurred.
3.1.3.3.2.5.5.5	SR:551	CF	The create operation shall raise the InvalidFileName exception when a fileName is not a valid file name or not an absolute pathname.

string inputs to the call and both must be absolute pathnames to the function call (despite the fact that it is not explicitly called out in the requirements). The copy call copies the contents of the source file to the destination file.

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

Exceptions raised by the copy call are the InvalidFileName or FileException. The InvalidFileName is raised if the file name provided does not meet the requirements for a valid SCA file name. The FileException is used for all other types of file processing errors, e.g. insufficient space to copy the file.

There are no explicit requirements regarding the absence or presence of the target file when the copy call is made. The nominal case is when the destination file does not exist prior to the copy. In this case, the copy call implicitly includes the create call to create a target file to receive the contents of the source file and then initiates the copy operation. In the case where a file already exists with the destination file name, it is equally valid to copy over the existing file as it is to raise an exception.

In most cases the interpretation is simply to copy over the file if it exists as this is a normal interpretation on UNIX and other operating systems. However, when using a framework implementation, it is important to be sure that the copy behavior implemented by the framework is documented and understood.

The copy operation instructs the FileSystem to copy the contents of the file identified by the sourceFileName parameter to the file specified by the destinationFileName parameter (SR:535) (see Table 3.14). If an error related to the files specified in the function call is encountered during the execution of the copy operation, then the operation raises a FileException exception (SR:536). The FileException errorNumber provides an indication of the type of error encountered. There is some ambiguity in the requirements with respect to the default behavior when the destinationFileName already exists, i.e. should an exception

Table 3.14. File copy requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.2.3	SR:535	CF	The copy operation shall copy the source file with the specified sourceFileName to the destination file with the specified destinationFileName.
3.1.3.3.2.5.2.5	SR:536	CF	The copy operation shall raise the CF FileException when a file-related error occurs.
3.1.3.3.2.5.2.5	SR:537	CF	The copy operation shall raise the InvalidFileName exception when the filename is not a valid file name or not an absolute pathname.

be raised or not? Generally speaking the default approach, as implemented by most operating systems, is to copy over the existing file if it already exists.

If either the sourceFileName or the destinationFileName arguments are not valid file names or absolute pathnames, then the InvalidFileName exception is raised (SR:537). Requirement SR:2 defines the format of a legal SCA file name. The SCA FileSystem copy operation must map through to the underlying file system of the hardware and operating software that is implementing the file system.

open

The open operation opens a file for access by the process issuing the call. The file name is provided to the call as a full path name. Additionally, a second argument specifies whether the open call is for read access only. If the `read_Only` parameter is true, then the file is opened for read only access. If it is false, then the file is opened for read/write access. An SCA File instance is returned to the caller upon successful completion of the call.

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

The standard exceptions are raised for the open call. The InvalidFileName is raised if the file name provided is not a full path name or otherwise does not meet the requirements for a standard SCA file name. The FileException is raised when any other file related error is encountered.

The specific requirement details for the open operation are provided in table 12.

The open call takes the file name provided and checks that it is a valid SCA file name and a full path name to the file desired. If not, it raises the InvalidFileName exception (SR:559). If the file name is valid, the operation then attempts to open the file in the native file system (SR:552). The file is opened with write access, i.e. the caller issuing the open operation can add data to the file, if the `Read_ONLY` parameter is false (SR:555).

Table 3.15. FileSystem open requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.6.3	SR:552	CF	The open operation shall open a file based upon the input fileName.
3.1.3.3.2.5.6.3	SR:555	CF	The open operation shall open the file for write access when the `read_Only` parameter is false.
3.1.3.3.2.5.6.4	SR:556	CF	The open operation shall return a File component parameter on successful completion.
3.1.3.3.2.5.6.4	SR:557	CF	The open operation shall return a null file component reference if the open operation is unsuccessful.
3.1.3.3.2.5.6.5	SR:558	CF	The open operation shall raise the CF FileException if the file does not exist or another file error occurred.
3.1.3.3.2.5.6.5	SR:559	CF	The open operation shall raise the InvalidFileName exception when the filename is not a valid file name or not an absolute pathname.

If any error occurs during the open operation, other than an InvalidFileName exception (SR:559), then a FileException exception is raised (SR:558). If the file does not exist or another error is encountered, then a null File object reference is returned (SR:557).[3]

An instance of an SCA File object, i.e. CORBA reference, is returned on successful completion (SR:556).

query

The query operation provides the ability for a client program to request and receive information about the SCA FileSystem. The FileSystem must implement two standard properties: `SIZE` and `AVAILABLE_SPACE.` These property names are defined as string constants and were discussed in Section 3.5.2.

```
void query (
    inout Properties fileSystemProperties
    )
raises (UnknownFileSystemProperties);
```

The UnknownFileSystemProperties is raised if a property is requested through the query operation that is undefined.

When called, the query operation provides the FileSystem property information, as specified in the fileSystemProperties parameter (SR:567) (see Table 3.16). A minimum of two properties must be supported providing the overall size of the FileSystem and the available space on the storage device on which the FileSystem resides (SR:568). Finally,

[3] *Requirement SR:557 states that a null File object reference is to be returned when an error is encountered. However, SR:559 and SR:558 indicate that an exception is to be thrown. This is an instance of conflicting requirements because return values are not provided when an exception is raised.*

Table 3.16. FileSystem query requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.9.3	SR:567	CF	The query operation shall return file system information to the calling client based upon the given fileSystemProperties' Id.
3.1.3.3.2.5.9.3	SR:568	CF	As a minimum, the FileSystem query operation shall support the following fileSystemProperties: 1. SIZE – an Id value of '`SIZE`' causes a query to return an `unsigned long long` containing the file system size (in octets). 2. AVAILABLE SPACE – an Id value of '`AVAILABLE_SPACE`' causes the
3.1.3.3.2.5.9.5	SR:569	CF	The query operation shall raise the UnknownFileSystemProperties exception when the given file system property is not recognized.

if the properties requested are not defined in the set of properties on the FileSystem, an UnknownFileSystemProperties exception is raised (SR:569).

remove

The remove operation supports the removal of a file from the FileSystem. The name of the file to be removed is provided in the filename parameter. The file name must provide the full path name to the file within the FileSystem.

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

The InvalidFileName exception is raised if the name provided in the filename parameter does not meet the requirements of a valid SCA file name. If any file related error occurs during the execution of the remove operation, then the FileException is raised.

Table 3.17. FileSystem remove requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.1.3	SR:532	CF	The remove operation shall remove the file with the given filename.
3.1.3.3.2.5.1.3	SR:533	CF	The remove operation shall raise the InvalidFileName exception when the filename is not a valid filename or not an absolute pathname.
3.1.3.3.2.5.1.5	SR:534	CF	The remove operation shall raise the CF FileException when a file-related error occurs.

The remove operation first ensures that the file name provided meets the requirements for an SCA file (see Table 3.17). If it does not, then the InvalidFileName exception is raised (SR:533). The primary action of the remove operation is to remove the file specified by the filename parameter from the FileSystem (SR:532). If any other file related error is encountered, the operation aborts, raising the FileException (SR:534).

It should be noted that the remove operation may execute successfully on a file that has been opened by another process. As discussed earlier, the file will appear to be deleted because it will be removed from the list of files not only in the SCA FileSystem but also in the native file system. However, the process that opened and is still using the file will continue to have access to the file until it has completed execution. At that point, the operating system will delete the file.

exists

The exists operation provides the capability to test if a specific file name exists within the FileSystem. The fileName parameter must contain the full path name for the file.

```
boolean exists (
     in string fileName
     )
raises (InvalidFileName);
```

The InvalidFileName exception is raised if the file name provided does not meet the requirements for a valid SCA file name.

Table 3.18. File exists requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.3.3	SR:538	CF	The exists operation shall check to see if a file exists based on the fileName parameter.
3.1.3.3.2.5.3.4	SR:539	CF	The exists operation shall return True if the file exists, or False if it does not.
3.1.3.3.2.5.3.5	SR:540	CF	The exists operation shall raise the InvalidFileName exception when fileName is not a valid file name or not an absolute pathname.

The exists operation will check the files within the FileSystem to see if the file specified by the filename argument exists (SR:538) (see Table 3.18). If the file does exist in the FileSystem then the function returns True and if it is not in the FileSystem it returns False (SR:539) . If the file name provided is invalid, then a InvalidFileName exception is raised (SR:540).

list

The list operation provides a listing of the files within a FileSystem. The operation returns a sequence of FileInformationType that provides the file name and the additional information maintained by the native file system and the SCA FileSystem. This is analogous to the directory list command, e.g. ls in UNIX or dir in DOS. The operation supports the use of search patterns and wildcards in the file name provided.

Search patterns are supported through the use of the normal wildcard characters, question mark (?) and asterisk (*). The question mark matches any single character in a sequence of characters that make up a file name. The asterisk matches one or more characters in a sequence of characters that make up a file name.

For example, the pattern '/fs/DeviceMgr1/f??' would match any three letter file name in /fs/DeviceMgr1/ that begins with an 'f'. So, /fs/DeviceMgr1/foo would match and /fs/DeviceMgr1/foo.exe would not.

```
FileInformationSequence list (
     in string pattern
     )
 raises (FileException, InvalidFileName);
```

Exceptions raised are InvalidFileName and FileException. The InvalidFileName exception in this case is raised whenever the search pattern cannot be interpreted or resolved to a legal SCA file name. For example, if the file pattern contains unexpected characters, i.e. characters that are illegal in a POSIX file name, or if the file name pattern does not start with an initial slash '/'. The FileException is raised for any other type of file processing error that occurs.

Table 3.19. FileSystem list requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.4.3	SR:541	CF	The list operation shall return a list of file information based upon the search pattern given.
3.1.3.3.2.5.4.3	SR:542	CF	The list operation shall support the following wildcard characters for base file names (i.e., the part after the right-most slash): 1. * used to match any sequence of characters (including null). 2. ? used to match any single character.
3.1.3.3.2.5.4.4	SR:543	CF	The list operation shall return a FileInformationSequence for files that match the wildcard specification as specified in the input pattern parameter.
3.1.3.3.2.5.4.5	SR:545	CF	The list operation shall raise the InvalidFileName exception when the input pattern does not start with a slash '/' or cannot be interpreted due to unexpected characters.
3.1.3.3.2.5.4.5	SR:546	CF	The list operation shall raise the FileException when a file-related error occurs.

The general requirement for the list call is to return a sequence of FileInformationType (SR:543) for each of the files in the FileSystem whose name matches the search pattern specified in the call (SR:541) (see Table 3.19). The search pattern, as noted earlier, may contain two wild card characters, the question mark (?), which matches a single character, and the asterisk ‘*’, which matches one or more characters (SR:542). When the file search string provided does not begin with a slash ‘/’ or cannot be interpreted, then the InvalidFileName exception is raised (SR:545). Any other file related error encountered during the process raises the FileException (SR:546).

mkdir

The mkdir operation supports the creation of a directory within an SCA FileSystem. If the directory name provided is a hierarchical name, e.g. /tmp/myfile/test, the then mkdir operation will create all directories in the hierarchical name, as required.

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

Two exceptions may be raised by the mkdir operation. If the directory name provided does not conform to a valid SCA file name, then the InvalidFileName exception is raised. If any other file related error is encountered the FileException is raised.
The requirements for the mkdir operation are listed in Table 3.20.

Table 3.20. FileSystem mkdir requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.7.3	SR:560	CF	The mkdir operation shall create a FileSystem directory based on the directoryName given.
3.1.3.3.2.5.7.3	SR:561	CF	The mkdir operation shall create all parent directories required to create the directoryName path given.
3.1.3.3.2.5.7.5	SR:562	CF	The mkdir operation shall raise the CF FileException if a file-related error occurred during the operation.
3.1.3.3.2.5.7.5	SR:563	CF	The mkdir operation shall raise the InvalidFileName exception when the directoryName is not a valid directory name.

The mkdir first checks that the directory name provided is a valid SCA file name. If it is not a valid file name then the InvalidFIleName exception is raised (SR:563). The operation then checks to see if the parent directories of the base directory exist. If one or more do not exist, mkdir creates each one starting with the highest level working to the directory just above the lowest level directory specified (SR:561). The operation then creates the base directory specified (SR:560). If any other file related error occurs, the FileException is raised (SR:562) and the operation terminates.

rmdir

The rmdir operation removes a directory from the FileSystem (see Table 3.21). The directory to be removed is identified by the directoryName parameter. Unlike the mkdir operation, which will create any parent directories in the path, rmdir only removes the lowest level or leaf directory specified in the directory. For example, if the call is made with the directory name '/tmp/files/test', only the directory 'test' will be removed.

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

If the directory name provided does not conform to the requirements for SCA file names, an InvalidFileName exception is raised. If any other file related error occurs, e.g. the directory specified does not exist, the FileException is raised.

Table 3.21. FileSystem rmdir requirements

Section	ID	Resp	Requirement
3.1.3.3.2.5.8.3	SR:564	CF	The rmdir operation shall remove a FileSystem directory, based on the directoryName given, only if the directory is empty (no files exist in directory).
3.1.3.3.2.5.8.5	SR:563	CF	The rmdir operation shall raise the CF FileException when the directory does not exist, if the directory is not empty, or another file-related error occurred.
3.1.3.3.2.5.8.5	SR:566	CF	The rmdir operation shall raise the InvalidFileName exception when the directoryName is not a valid directory name.

The rmdir operation first checks that the directoryName contains a valid SCA file name. If not, the InvalidFileName exception is raised (SR:566) and the operation terminates. If the directoryName is a valid SCA file name, then the directory is removed from the FileSystem. However, the directory is removed only if no file exists in the directory (SR:564). If any file-related error is encountered, then the FileException is raised (SR:563) and the operation terminates.

3.6 File

Basic file operations such as read and write must be supported within the SCA file system. The File interfaces provide this functionality. In addition to these essential operations, three additional operations are defined: `close, setFilePointer`, and `sizeOf`.

As shown in Figure 3.9, there are five basic operations that can be called on an SCA File and (read, write, close, setFilePointer, and sizeOf there are two attributes (filePointer

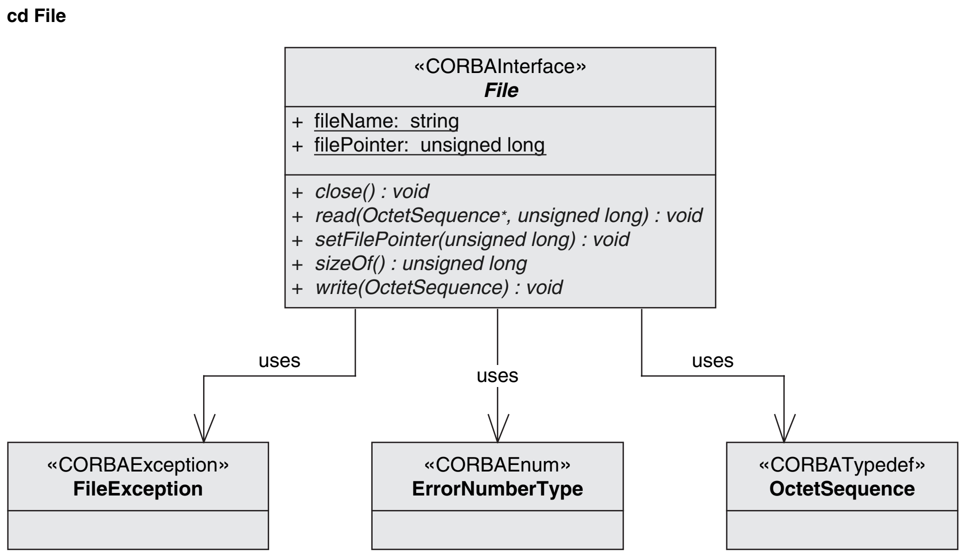

cd File
«CORBAInterface»
File
+ fileName: string
+ filePointer: unsigned long
+ close() : void
+ read(OctetSequence*, unsigned long) : void
+ setFilePointer(unsigned long) : void
+ sizeOf() : unsigned long
+ write(OctetSequence) : void
uses
uses
uses
«CORBAException»
FileException
«CORBAEnum»
ErrorNumberType
«CORBATypedef»
OctetSequence

Figure 3.9. File interfaces

and fileName). The following sections discuss the requirements and implementation of the SCA File.

3.6.1 Exceptions

IOException

Several exceptions and types are defined as part of the File interface. The IOException is used to identify errors that occur during the basic read or write operations.

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

The errorNumber identifies the type of error (SR:507) (see Table 3.22) and the msg field provides the ability to insert a user readable string.

Table 3.22. File Error type requirements

Section	ID	Resp	Requirement
3.1.3.3.1.3.1	SR:507	CF	The errorNumber shall indicate an ErrorNumberType value (e.g. EFBIG, ENOSPC, EROFS). The message is component-dependent, providing additional information describing the reason for the error.

InvalidFilePointer

The InvalidFilePointer is raised in cases where the file pointer provided in a call is outside the range of the file's size.

```
exception InvalidFilePointer {
};
```

3.6.2 Attributes

The SCA File requirements are organized into several groups. The first group discusses the File attributes; fileName and filePointer (see Table 3.23).

fileName

```
readonly attribute string fileName;
```

filePointer

```
readonly attribute unsigned long filePointer;
```

When a file is opened for access in an operating system, a file handle or file descriptor (FD) is returned. The file descriptor is then used by the application that opened the file for issuing calls and performing operations on the file. Typically, the file descriptor is an unsigned integer that is uniquely assigned by the operating system such that the same file descriptor number will never be assigned to different files at the same time. In an SCA file system, the file descriptor is the instance of an SCA File that is returned by the open operation on the SCA FileSystem.

Table 3.23. File Attribute requirements

Section	ID	Resp	Requirement
3.1.3.3.1.4.1	SR:509	CF	The `readonly` fileName attribute shall contain the file name given to the FileSystem open/create operation.
3.1.3.3.1.4.2	SR:510	CF	The `readonly` filePointer attribute shall contain the file position where the next read or write will occur.

The filename attribute of the SCA file contains the name provided during the open/create operation (SR:509). It is a read-only that is set when the file is instantiated. The filePointer attribute `readonly` contains the offset used to retrieve the next octet for a read operation or place the next octet for a write operation (SR:510).

3.6.3 Operations

read

The read operation provides the basic capability to read data from the file. The operation is provided by the number of bytes to be read through the length parameter. The data obtained by the read operation is provided to the caller as an out parameter named data. Data returned is provided as an OctetSequence.

```
void read (
     out OctetSequence data,
     in unsigned long length
     )
raises (IOException);
```

If any error occurs during the execution of the read operation, the IOException is raised.

Table 3.24. File Read requirements

Section	ID	Resp	Requirement
3.1.3.3.1.5.1.3	SR:511	CF	The read operation shall read, from the referenced file, the number of octets specified by the input length parameter and advance the value of the filePointer attribute by the number of octets actually read.
3.1.3.3.1.5.1.3	SR:512	CF	The read operation shall read less than the number of octets specified in the input-length parameter, when an end of file is encountered.
3.1.3.3.1.5.1.4	SR:513	CF	The read operation shall return via the out Message parameter a CF OctetSequence that equals the number of octets actually read from the File.
3.1.3.3.1.5.1.4	SR:514	CF	If the filePointer attribute value reflects the end of the File, the read operation shall return a 0-length CF OctetSequence.
3.1.3.3.1.5.1.5	SR:515	CF	The read operation shall raise the IOException when a read error occurs.

The read operation initiates the read at the current offset into the file as specified by the value of the filePointer (see Table 3.24). The read operation reads the number of octets specified by the length parameter on the read operation (SR:511). If the number of octets to be read is greater than the number of octets remaining in the file, i.e. between the filePointer and the end of the file (EOF), then the number of octets passed back to the caller will be the octets between the filePointer and the EOF (SR:512). The number of octets returned is equal to the number of octets read (SR:513). If the read operation is invoked when the filePointer is set to the end of the file, then a zero length octet sequence is returned (SR:514). If an error occurs during the read operation, then the IOException is raised (SR:515).

write

The write operation provides the capability to insert data into a File. The data to be written to the file is provided as an OctetSequence. The data is written to the file starting at the location specified by the filePointer.

```
void write (
     in OctetSequence data
     )
raises (IOException);
```

Upon completion of the write operation, the filePointer is set to the position of the last octet written to the file. If any error occurs during the write operation, then the IOException is raised.

Table 3.25. File Write requirements

Section	ID	Resp	Requirement
3.1.3.3.1.5.2.3	SR:516	CF	The write operation shall write data to the file referenced.
3.1.3.3.1.5.2.3	SR:517	CF	If the write is successful, the write operation shall increment the filePointer attribute to reflect the number of octets written.
3.1.3.3.1.5.2.3	SR:518	CF	If the write is unsuccessful, the filePointer attribute value shall maintain or be restored to its value prior to the write operation call.
3.1.3.3.1.5.2.5	SR:519	CF	The write operation shall raise the IOException when a write error occurs.

The write operation writes the octets provided in the data parameter to the file starting at the position specified by the filePointer (SR:516) (see Table 3.25). Upon successfully writing the data to the file, the filePointer is updated to reflect the new offset into the file (SR:517). If an error occurs during the write, then an IOException is raised (SR:519) and, as part of the exception handler routine, the filePointer is restored to the offset value at the initiation of the write operation (SR:518).

close

The close operation terminates the connection between the calling process that was utilizing the file.

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

If the file cannot be closed for any reason, the FileException is raised.

The close operation terminates the connection between the calling program that was using the file and the file (see Table 3.26). Any resources associated with the connection, e.g.

buffers, are released (SR:522) and the file is made unavailable to the caller (SR:523). If the file cannot be closed for any reason, then the FileException is raised (SR:524).

Table 3.26. File Close requirements

Section	ID	Resp	Requirement
3.1.3.3.1.5.4.3	SCA443	CF	The error number shall indicate an ErrorNumberType value (e.g. EFBIG, ENOSPC, EROFS). The message is component-dependent, providing additional information describing the reason for the error.
3.1.3.3.1.5.4.3	SR:522	CF	The close operation shall release any OE file resources associated with the component.
3.1.3.3.1.5.4.3	SR:523	CF	The close operation shall make the file unavailable to the component.
3.1.3.3.1.5.4.5	SR:524	CF	The close operation shall raise the CF FileException when it cannot successfully close the file.

setFilePointer

As with standard file systems, the file within an SCA FileSystem allows the index into the file system, i.e. offset to the next byte to be read, to be set. The setFilePointer operation allows the calling program to set the index into the file specified. This offset is then used as the starting offset for the next read or write operation.

```
void setFilePointer (
     in unsigned long filePointer
     )
raises (InvalidFilePointer, FileException);
```

Table 3.27. File Pointer requirements

Section	ID	Resp	Requirement
3.1.3.3.1.5.5.3	SR:525	CF	The setFilePointer operation shall set the filePointer attribute value to the input filePointer.
3.1.3.3.1.5.5.5	SR:526	CF	The setFilePointer operation shall raise the CP FileException when the file pointer for the referenced file cannot be set to the value of the input filePointer parameter.
3.1.3.3.1.5.5.5	SR:527	CF	The setFilePointer operation shall raise the InvalidFilePointer exception when the value of the filePointer parameter exceeds the file size.

The setFilePointer provides the ability to set the position reference of the underlying file to a byte offset from the beginning of the file (SR:525) (see Table 3.27). If the offset specified in the call is outside the range of the file, e.g. greater than the number of bytes in the file or less than zero, then the call raises the InvalidFilePointer exception (SR:527). If the file pointer cannot be set for any other reason, e.g. less than zero, then the FileException is raised (SR:526).

sizeOf

The sizeOf operation provides the means for obtaining the file size. The file size is return

```
unsigned long sizeOf ()
     raises (FileException);
```

The sizeOf operation provides a method for obtaining the size of the file in octets, i.e. bytes.

Table 3.28. File Size requirements

Section	ID	Resp	Requirement
3.1.3.3.1.5.3.4	SCA443	CF	The error number shall indicate an ErrorNumberType value (e.g. EFBIG, ENOSPC, EROFS). The message is component-dependent, providing additional information describing the reason for the error.
3.1.3.3.1.5.3.4	SR:520	CF	The sizeOf operation shall return the number of octets stored in the file.
3.1.3.3.1.5.3.5	SR:521	CF	The sizeOf operation shall raise the CF FileException when a file-related error occurs (e.g. file does not exist anymore).

When called, the sizeOf operation returns the number of octets within the file, i.e. the size of the file (SR:520) (see Table 3.28). If any error is encountered accessing the file, then the FileException is raised (SR:521). The error number within the FileException provides some indication of the type of error encountered (SCA443).

4

Foundation Interfaces and Data Types

Most of the SCA implementation rests on several interfaces. These foundation interfaces define the common operational interfaces across all devices and applications within an SCA system.

Figure 4.1 illustrates the foundation IDL defined as part of the SCA. The Resource is a pivotal interface as it is inherited by all devices and applications. Thus, it forms the foundation for much of the operational components of an SCA system. As can be seen in Figure 4.1, four common interfaces are inherited by the Resource interface: i) TestableObject; ii) PortSupplier; iii) LifeCycle; and iv) PropertySet.

These interfaces are discussed in the following sections, along with the ResourceFactory, which provides the ability to instantiate a set of Resources as a logical unit.

4.1 TestableObject

The TestableObject provides the basic ability to initiate one or more internal tests for a component (Figure 4.2). The component under test may be a hardware component, such as a signal processing board, or it may be a software component running on a GPP.

4.1.1 Exceptions

UnknownTest

The UnknownTest exception is raised when the test requested by the runTest operation is unknown.

```
 exception UnknownTest {
};
```

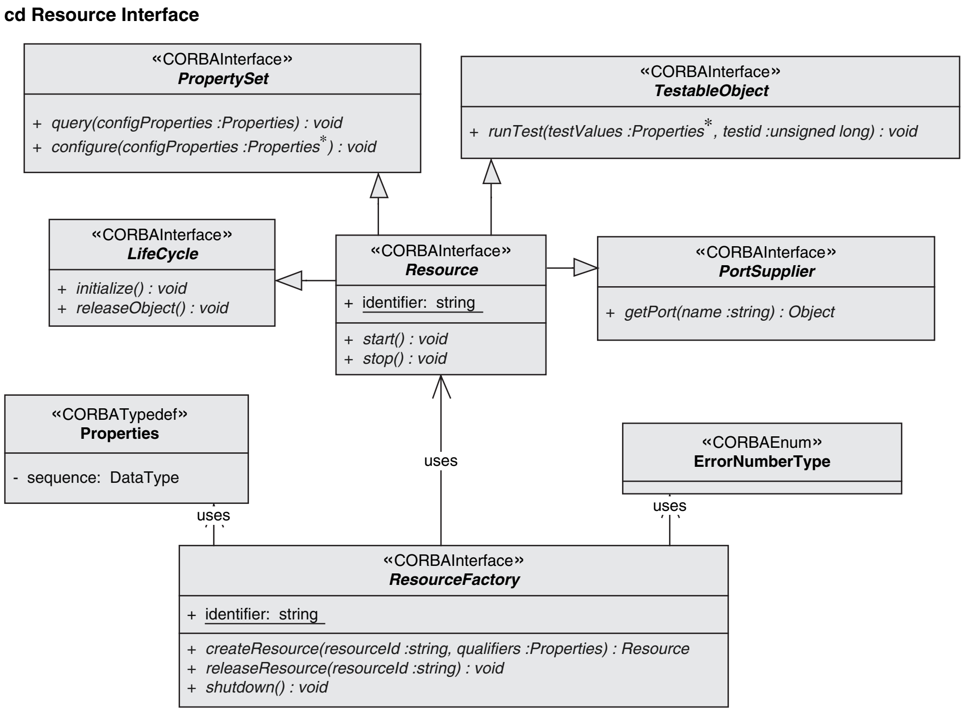

cd Resource Interface
«CORBAInterface» PropertySet
+ query(configProperties :Properties) : void
+ configure(configProperties :Properties*) : void
«CORBAInterface» TestableObject
+ runTest(testValues :Properties*, testid :unsigned long) : void
«CORBAInterface» LifeCycle
+ initialize() : void
+ releaseObject() : void
«CORBAInterface» Resource
+ identifier: string
+ start() : void
+ stop() : void
«CORBAInterface» PortSupplier
+ getPort(name :string) : Object
«CORBATypedef» Properties
- sequence: DataType
uses
«CORBAEnum» ErrorNumberType
uses
uses
«CORBAInterface» ResourceFactory
+ identifier: string
+ createResource(resourceId :string, qualifiers :Properties) : Resource
+ releaseResource(resourceId :string) : void
+ shutdown() : void

Figure 4.1. Common interfaces

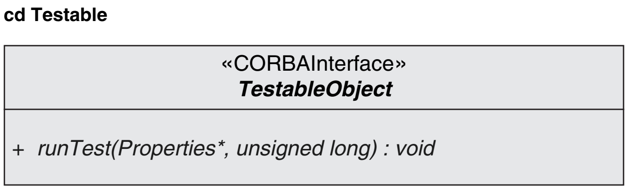

cd Testable
«CORBAInterface» TestableObject
+ runTest(Properties*, unsigned long) : void

Figure 4.2. TestableObject interface

4.1.2 Operations

runTest

The runTest operation is the method by which an SCA program may initiate a test for a given component. The caller provides an unsigned long which identifies the test to be run and a set of Properties that identify the test values to be applied to the test. Mapping the testid provided to the test to be performed is unique to each component and must be documented as part of the components implementation. For example, a testid value of 3 for a signal processing board may initiate a checksum calculation on the contents of flash memory on the board while a testid of 3 to a software component may check to see if all threads are in a nominal state.

```
void runTest (
     in unsigned long testid,
     inout Properties testValues
     )
raises (UnknownTest, UnknownProperties);
```

If the testid provided does not map to any internally defined tests, then the UnknownTest exception is raised. If the testid is valid but one or more properties provided in the testValues parameter do not match those expected by the specified test, then the UnknownProperties exception is raised.

The runTest operation is called with a testid parameter value that maps to a pre-defined test within the component receiving the call (SR:79) (see Table 4.1). If the testid provided does not map to any test defined by the component, then the UnknownTest exception is raised (SR:85). The caller also passes in a set of testvalues as a property set to the component to be tested (SR:80). The tests specified are run and the results are returned (SR:81). The component then verifies the validity of the test results against the test properties defined in the component's Software Package Descriptor (SPD) XML file (SR:83). The XML files are discussed in Part II of this book on The Domain Profile. If any of the properties provided are not defined, the UnknownProperties exception is raised (SR:86). The values provided with the properties are also verified. If any of the values are invalid or outside a pre-defined range, then the InvalidProperties exception is raised (SR:87). If either the testId or

Table 4.1. TestableObject requirements

Section	ID	Resp	Requirement
3.1.3.1.3.5.1.3	SR:79	DS	The runTest operation shall use the testId parameter to determine which of its predefined test implementations should be performed.
3.1.3.1.3.5.1.3	SR:80	DS	The testValues parameter CF Properties (id/value pair(s)) shall be used to provide additional information to the implementation-specific test to be run.
3.1.3.1.3.5.1.3	SR:81	DS	The runTest operation shall return the result(s) of the test in the testValues parameter.
3.1.3.1.3.5.1.3	SR:82	DS	Valid testId(s) and both input and output testValues (properties) for the runTest operation shall at a minimum be test properties defined in the properties test element of the component's Properties Descriptor (refer to Appendix D Domain Profile).
3.1.3.1.3.5.1.3	SR:83	DS	All inputValues properties shall be validated (i.e. test properties defined in the propertyfile(s) referenced in the component's SPD).
3.1.3.1.3.5.1.3	SR:84	DS	The runTest operation shall not execute any testing when the input testId or any of the input testValues are not known by the component or are out of range.

Table 4.1. Continued

Section	ID	Resp	Requirement
3.1.3.1.3.5.1.5	SR:85	DS	The runTest operation shall raise the UnknownTest exception when there is no underlying test implementation that is associated with the input testId given.
3.1.3.1.3.5.1.5	SR:86	DS	The runTest operation shall raise the UnknownProperties exception when the input parameter testValues contains any DataTypes that are not known by the component's test implementation or any values that are out of range for the requested test.
3.1.3.1.3.5.1.5	SR:87	DS	The exception parameter invalidProperties shall contain the invalid inputValues properties Id(s) that are not known by the component or the value(s) are out of range.

testValues parameters contain an error as described above, then the execution of testing shall be inhibited (SR:84).

Valid testId values are, at a minimum, the set of test properties defined in the properties test element of the component's property descriptor (SR:82). If the testId and testValues parameters are valid, the test is run and the results of the test are returned through the inout testValues parameter.

4.2 PortSupplier

Any Resource or component that inherits the Resource interface, i.e. all devices and applications, may support one or more ports. As noted earlier, the Port is an endpoint that can be used to connect two components together. The PortSupplier interface provides the mechanism for obtaining a reference to a Port.

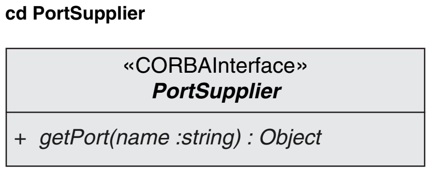


Figure 4.3. PortSupplier interface

As illustrated in Figure 4.3, a single operation is defined for the PortSupplier interface.

4.2.1 Exceptions

UnknownPort

If the port requested by the getPort operation is not defined, then the UnknownPort exception is raised.

```
exception UnknownPort {
};
```

4.2.2 Operations

getPort

The getPort operation is provided the string name of a Port. The string name is defined as part of the component's interface description in the Software Component Descriptor (SCD) file. The call matches the string name provided against those ports instantiated as part of the component. If a match is found, then an Object reference to the requested port is returned.

```
Object getPort (
    in string name
    )
 raises (UnknownPort);
```

As noted above, if the string name of the port requested is not found, the UnknownPort exception is raised.

The string name provided as input to the getPort call is matched against those ports instantiated by the component. If a match is found, then the Object reference to the port is returned (SR:89) (see Table 4.2). If no match is found, then the UnknownPort exception is raised (SR:90).[1]

Table 4.2. PortSupplier requirements

Section	ID	Resp	Requirement
3.1.3.1.4.5.1.4	SR:89	DS	The getPort operation shall return the CORBA Object reference that is associated with the input port name.
3.1.3.1.4.5.1.5	SR:90	DS	The getPort operation shall raise an UnknownPort exception if the port name is invalid.

[1] *The Object reference is a handle to an IDL interface. The object calling the getPort method would then bind to the interface returned in order to establish a communications link. This process is discussed in more detail in Section 4.7.*

4.3 LifeCycle

The LifeCycle interface addresses common operations associated with the instantiation and startup of a component and the tear down and termination of a component.

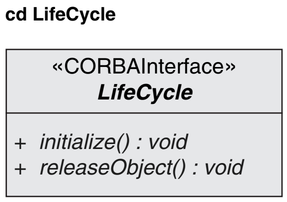


Figure 4.4. LifeCycle interface

Figure 4.4 shows the LifeCycle interface. Two operations are defined, initialize and releaseObject.

4.3.1 Exceptions

InitializeError

The InitializeError exception indicates that one or more errors were encountered during the initialization process (see Table 4.3). The exception includes one or more messages providing some descriptive data on the errors encountered.

```
exception InitializeError {
      StringSequence errorMessages;
};
```

ReleaseError

The ReleaseError exception indicates that one or more errors were encountered during the tear down process of the releaseObject operation. As with the InitializeError exception, the ReleaseError exception provides a sequence of one or more strings providing some descriptive information regarding the errors encountered.

```
exception ReleaseError {
      StringSequence errorMessages;
};
```

4.3.2 Operations

initialize

The initialize operation is unique to each component. As such, there are no requirements specific to the initialization process or states. The intent is to provide a common interface call that will initiate the initialization of the component to some defined and stable state.

Table 4.3. LifeCycle initialize and releaseObject requirements

Section	ID	Resp	Requirement
3.1.3.1.2.5.1.5	SR:74	CF, WF, DS	The initialize operation shall raise an InitializeError exception when an initialization error occurs.
3.1.3.1.2.5.2.3	SR:75	CF, WF, DS	The releaseObject operation shall release all internal memory allocated by the component during the life of the component.
3.1.3.1.2.5.2.3	SR:76	CF, WF, DS	The releaseObject operation shall tear down the component (i.e. released from the CORBA environment).
3.1.3.1.2.5.2.3	SR:77	CF, WF, DS	The releaseObject operation shall release components from the OE.
3.1.3.1.2.5.2.5	SR:78	CF, WF, DS	The releaseObject operation shall raise a ReleaseError exception when a release error occurs.

No arguments are defined and all initialization values and states must be defined and implemented internally to the initialize call.

```
void initialize ()
     raises (InitializeError);
```

If an error is encountered during the initialization process, the operation raises the InitializeError exception with one or more string messages describing the errors encountered (SR:74).

releaseObject

The releaseObject operation performs the tear down process associated with a component (SR:76). The first action is to remove itself from the CORBA ORB (SR:77). The component releases all memory or other resources allocated to the component (SR:75). The component removes itself from the Operating Environment.

```
void releaseObject ()
    raises (ReleaseError);
```

If any error is encountered during the operation, a ReleaseError is raised (SR:78) with one or more string messages providing some description of the errors encountered.

4.4 PropertySet

4.4.1 Exceptions

InvalidConfiguration

This exception is raised when the configuration of a component, i.e. setting the values of the properties specified, fails. The exception indicates that none of the property values were modified. A string value provides descriptive information regarding the failure and a set of Properties containing the error(s) is returned in the exception.

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

PartialConfiguration

A PartialConfiguration exception is raised when an error is encountered attempting to configure a component, i.e. set the property values specified but one or more properties were configured successfully. This exception is usually encountered when an attempt is made to set the value on a readonly property. The set of properties that resulted in the exception is returned.

```
 exception PartialConfiguration {
      Properties invalidProperties;
};
```

4.4.2 Operations

configure

The configure operation accepts a set of property-value pairs and attempts to set each of the properties in the sequence to the value specified.

```
void configure (
     in Properties configProperties
     )
 raises (InvalidConfiguration, PartialConfiguration);
```

The configure operation accepts a list of property names and values. Each value is assigned to the property name specified in the configure operation (SR:91) (see Table 4.4). A property must be specified as a readwrite or writeonly property for the configure operation to assign the value (SR:92). It is possible that one or more properties in the set may have been assigned values and one or more values were not set, e.g. the operation attempted to set a readonly property. In this situation, a PartialConfiguration exception is raised (SR:93). If any error is encountered that prevents a property configuration operation from successfully completing, then an InvalidConfiguration exception is raised (SR:94). For example, a property configuration operation may need to perform an operation on a physical device. If the interface to the device fails for some reason, then the InvalidConfiguration exception is raised.

query

The query operation enables an application to retrieve the values associated with one or more property names.

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

Table 4.4. PropertySet configure requirements

Section	ID	Resp	Requirement
3.1.3.1.5.5.1.3	SR:91	CF	The configure operation shall assign values to the properties as indicated in the configProperties argument.
3.1.3.1.5.5.1.3	SR:92	CF	Valid properties for the configure operation shall at a minimum be the configure readwrite and writeonly properties referenced in the component's SPD.
3.1.3.1.5.5.1.5	SR:93	CF	The configure operation shall raise a PartialConfiguration exception when some configuration properties were successfully set and some configuration properties were not successfully set.
3.1.3.1.5.5.1.5	SR:94	CF	The configure operation shall raise an InvalidConfiguration exception when a configuration error occurs that prevents any property configuration on the component.

A sequence of Properties identifying the property items to be queried is provided as an input argument. The sequence provided may be empty.

The query operation accepts a sequence of Properties and returns the values for each of the property names specified in the sequence (SR:96) (see Table 4.5). If the property sequence provided is empty, i.e. zero length, then all of the properties and the value associated for each property currently defined for the object receiving the call is returned (SR:95).

Table 4.5. PropertySet query requirements

Section	ID	Resp	Requirement
3.1.3.1.5.5.2.3	SR:95	CF	If the configProperties are zero size, then the query operation shall return all component properties.
3.1.3.1.5.5.2.3	SR:96	CF	If the configProperties are not zero size, then the query operation shall return only those Id/value pairs specified in the configProperties.
3.1.3.1.5.5.2.3	SR:97	CF	Valid properties for the query operation shall at a minimum be the configure, readwrite, and readonly properties, and allocation properties that have an action value of 'external' as referenced in the component's SPD.
3.1.3.1.5.5.2.5	SR:98	CF	The query operation shall raise the CF UnknownProperties exception when one or more properties being requested are not known by the component.

In order to return a value successfully for a property, the property must be defined as a configure, readwrite, or readonly property or an allocation property that has an action value of ‘external’ (SR:97). A configure property means that the property is initially configured when the component containing the property is instantiated. If the property is not designated as a configure property, it may be queried if the property mode is defined as readwrite or readonly. An allocation property refers to a property that identifies some capacity that is allocated through the allocateCapacity operation. This typically occurs during waveform instantiation when the system is attempting to map waveform components to the hardware devices and other resources. If the allocation property is defined to be external, then the property may be queried. If any of the properties specified in the input sequence is not defined then the operation raises an UnknownProperties exception (SR:98).

4.5 Resource

The Resource forms a fundamental component of the SCA framework. Much of the interfaces implemented by components that form the software radio inherit from the Resource interface. Resource itself inherits from several interface specifications. As illustrated in Figure 4.1 at the beginning of this chapter, the Resource incorporates the interfaces from the TestableObject, PortSupplier, LifeCycle, and PropertySet. These interfaces were discussed in the preceding sections. Thus, any implementation of the Resource interface, including those interfaces that inherit the Resource interface, must also implement the interfaces inherited by the Resource. This section discusses the individual interfaces that compose the Resource interface.

The key aspect introduced by the Resource interface is the start and stop operations. These operations provide the top-level control mechanism for the control of the Resource implementation. This ranges from physical devices to software components and applications, since the Device and Application interfaces inherit from the Resource interface.

4.5.1 Exceptions

StartError

If an error is encountered within a device or other component that implements the Resource interface during execution of the start operation, a StartError exception is raised. The exception contains an enumeration type indicating the type of error and a string providing additional information in human readable form.

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

StopError

If an error is encountered within a device or other component that implements the Resource interface during execution of the stop operation, a StopError exception is raised. The exception contains an enumeration type indicating the type of error and a string providing additional information in human readable form.

```
exception StopError {
     ErrorNumberType errorNumber;
     string msg;
};
```

4.5.2 Attributes

A single attribute is defined for the Resource: identifier.

identifier

The identifier attribute maintains the unique identifier for the resource instance.

```
readonly attribute string identifier;
```

Table 4.6. Identifier requirements

Section	ID	Resp	Requirement
3.1.3.1.6.4.1	SR:101	CF,WF, DS	The readonly identifier attribute shall contain the unique identifier for a resource instance.

The identifier is a readonly attribute that is initialized when the Resource is instantiated (SR:101) (See Table 4.6). The unique identifier value is a GUID defined or generated for the component. If the GUID is assigned, the value is specified using the Id attribute in the domain profile XML.

4.5.3 Operations

As previously noted, there are two operations defined by the Resource interface: start and stop.

start

Once a resource is instantiated and initialized, the start operation is used to place it into an operational mode. Any error encountered during the start operation triggers a StartError exception (See Table 4.7).

```
void start ()
     raises (StartError);
```

As can be observed from the requirements in Table 4.7, there are no functional requirements related to the start operation specified. This is because the start operation and associated functional logic is unique to each Resource. Therefore, the only requirements specified refer to the case when some problem occurs during the start operation. When a problem

Table 4.7. start requirements

Section	ID	Resp	Requirement
3.1.3.1.6.3.2	SR:99	CF, WF, DS	The error number shall indicate an ErrorNumberType value (e.g. EDOM, EPERM, ERANGE).
3.1.3.1.6.5.2.5	SR:105	CF, WF, DS	The start operation shall raise the StartError exception if an error occurs while starting the resource.

is encountered during the start operation, the startError exception is raised (SR:105). The startError exception contains a number that enumerates the type of error encountered (SR:99) and a text string providing additional descriptive information about the error encountered.

stop

The stop operation is used to halt the processing performed by the Resource. This means that the operational functions of the Resource are stopped (see Table 4.8). It does not terminate the Resource or remove it from the system. In the context of an executable program, the stop operation may be viewed as a halt command to the application. For example, if the application accepts a data value, performs some transformation on the data and forwards the data to the next processing component, the stop operation would cause the application to stop accepting, transforming, and forwarding data values. The application would still be in memory and running, in terms of the operating system. However, the CPU time consumed by the application would be minimal to none. In the context of a component implemented as a VHDL component on an FPGA, the stop operation may be realized by a register value or logic line that inhibits the state machine implemented by the VHDL. Similar to the application on the GPP, the functional logic is not removed from the FPGA.

```
void stop ()
    raises (StopError);
```

Table 4.8. stop requirements

Section	ID	Resp	Requirement
3.1.3.1.6.3.3	SR:100	CF, WF, DS	The error number shall indicate an ErrorNumberType value (e.g. ECANCELED, EFAULT, EINPROGRESS).
3.1.3.1.6.5.1.3	SR:102	CF, WF, DS	The stop operation shall disable all current operations and put the Resource in a non-operating condition.
3.1.3.1.6.5.1.5	SR:103	CF, WF, DS	The stop operation shall raise the StopError exception if an error occurs while stopping the resource.

As with the start operation, the functional logic of the stop operation is unique for each implementation of a Resource. The only requirement associated with the functional operation of the Resource is that the stop operation places the Resource in a non-operational state (SR:102). If an error is encountered during the stop operation, a stopError exception is raised (SR:103). The stopError exception contains an errorNumber indicating the type of error encountered (SR:100) and a string containing the additional information regarding the cause of the exception.

4.6 ResourceFactory

Although a Resource may be instantiated directly, an instance of a Resource may be utilized by multiple applications or components that require maintaining a reference count of client applications using the Resource. In addition to maintaining a set of server-side references for a Resource, a set of resources may be frequently instantiated as a logical set. The ResourceFactory construct provides the facility to perform these types of operations and capabilities (Figure 4.5). In addition, utilizing a ResourceFactory can help reduce the overhead of instantiating multiple resources. As can be inferred from the name, the ResourceFactory follows the Factory design pattern.

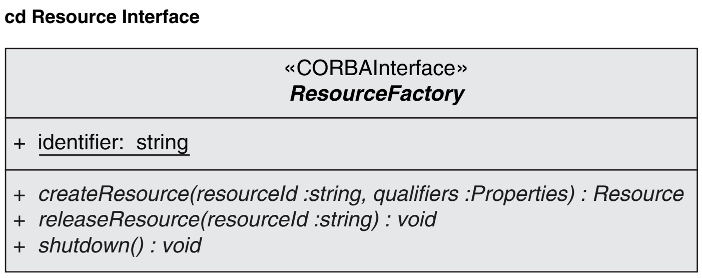

cd Resource Interface
«CORBAInterface»
ResourceFactory
+ identifier: string
+ createResource(resourceId :string, qualifiers :Properties) : Resource
+ releaseResource(resourceId :string) : void
+ shutdown() : void

Figure 4.5. ResourceFactory interface

4.6.1 Exceptions

InvalidResource

When the ResourceFactory is requested to release a Resource using the releaseResource method (described on page xxx), a resourceId is provided as an input argument to specify the Resource to be released. If the resourceId provided does not match any Resources known by the ResourceFactory, then the InvalidResource exception is raised.

```
exception InvalidResourceId {
};
```

ShutdownFailure

The shutdown method, described on page xxx, is used to halt and terminate all the Resources that have been instantiated through the ResourceFactory. If an error is encountered during the shutdown process preventing the ResourceFactory from releasing all the Resources, the ShutdownFailure is raised. The exception contains a string describing the error condition.

```
exception ShutdownFailure {
      string msg;
};
```

CreateResourceFailure

If there is a problem performing a createResource, then a CreateResourceFailure exception is raised. The exception contains an errorNumber to indicate the type of error encountered and a string providing some additional information regarding the exception.

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

4.6.2 Attributes

identifier

As previously discussed for the Resource, the ResourceFactory also has an identifier attribute that contains a unique identifier.

```
readonly attribute string identifier;
```

4.6.3 Operations

createResource

As noted previously, the ResourceFactory provides the capability to instantiate one or more Resources on demand. Another scenario is to have the ResourceFactory create a pool of resources by instantiating multiple Resources at startup and then simply returning the reference to a Resource in response to the createResource call. This approach trades a longer startup time in return for faster execution because when the createResource is called at run time, the Resource has already been created. The resourceId specifies the resource to be created.

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

When the createResource operation is called, the ResourceFactory either creates a new Resource instance (SR:109) or returns a reference to an available instance of the type specified by the resourceId argument (see Table 4.9). A reference to the Resource is returned

Table 4.9. ResourceFactory createResource requirements

Section	ID	Resp	Requirement
3.1.3.1.7.3.3	SR:108	WF, DS	The error number shall indicate an ErrorNumberType value (e.g. NOTSET, EBADMSG, EINVAL, EMSGSIZE, ENOMEM).
3.1.3.1.7.5.1.3	SR:109	WF, DS	If no Resource exists for the given resourceId, the createResource operation shall create a Resource.
3.1.3.1.7.5.1.3	SR:110	WF, DS	The createResource operation shall assign the given resourceId to a new Resource and either set a reference count to one, when the Resource is initially created, or increment the reference count by one, when the Resource already exists.
3.1.3.1.7.5.1.4	SR:111	WF, DS	The createResource operation shall return a reference to the created Resource or the existing Resource.
3.1.3.1.7.5.1.4	SR:112	WF, DS	The createResource operation shall return a nil CORBA component reference when the operation is unable to create or find the Resource.
3.1.3.1.7.5.1.5	SR:115	WF, DS	The createResource operation shall raise the CreateResourceFailure exception when it cannot create the Resource.

to the calling application (SR:111). As part of the createResource execution, a reference count is maintained internally by the ResourceFactory. As each Resource is created or provided from a pool of Resources, a reference count is set to one, if initially created during the call, or incremented if a pre-existing Resource is returned (SR:110).

If an error is encountered that prevents a Resource from being created or returned (if a pool of Resources is used), then a CreateResourceFailure exception is raised (SR:115). Within the exception structure, an error number is provided to indicate the type of error encountered (SR:108). The functional requirements also state that, in the event of an error, a null CORBA reference is returned as the Resource reference (SR:112). This last requirement and SR:115 is one of several instances within the SCA specification that specify a return value *and* an exception is to be raised when an error condition is encountered. In a programming language that supports true exception handling, *either* an exception is raised *or* a value is returned. Thus, requirements SR:112 and SR:115 are mutually exclusive because both cannot be satisfied in an implementation.

releaseResource

When a component has no further need for a Resource obtained through the createResource call described above, the component (referred to as a client), uses the Resource and notifies the ResourceFactory that it has completed using the Resource using the releaseResource call. The client component must still release the CORBA reference that it has to the Resource.

This operation does not return a value. The releaseResource operation raises the InvalidResourceId exception if an invalid resourceId is received.

```
void releaseResource (
     in string resourceId
     )
raises (InvalidResourceId);
```

Table 4.10. ResourceFactory releaseResource requirements

Section	ID	Resp	Requirement
3.1.3.1.7.5.2.3	SR:116	WF, DS	The releaseResource operation shall decrement the reference count for the specified resource, as indicated by the resourceId.
3.1.3.1.7.5.2.3	SR:117	WF, DS	The releaseResource operation shall make the Resource no longer available (i.e. it is released from the CORBA environment) when the Resource's reference count is zero.
3.1.3.1.7.5.2.5	SR:118	WF, DS	The releaseResource operation shall raise the InvalidResourceId exception if an invalid resourceId is received.

When the client has no further need of the Resource, it notifies the ResourceFactory using the releaseResource call (see Table 4.10). The ResourceFactory decrements the internal reference count of the Resource identified by the resourceId provided in the call (SR:116). If the resourceId provided does not match any Resource instances within the ResourceFactory, then an InvalidResource exception is raised (SR:118). When the last client releases the Resource, the reference count will go to zero when decremented by the ResourceFactory. When the reference count for a Resource goes to zero, then the ResourceFactory will release the Resource from the CORBA environment making it no longer available (SR:117).

shutdown

When no longer needed, the ResourceFactory must be terminated using the shutdown operation. This operation provides a mechanism for orderly termination of the ResourceFactory. The shutdown operation raises the ShutdownFailure exception for any error that prevents the shutdown of the ResourceFactory.

```
void shutdown ()
     raises (ShutdownFailure);
```

Table 4.11. ResourceFactory shutdown requirements

Section	ID	Resp	Requirement
3.1.3.1.7.5.3.3	SR:119	WF, DS	The shutdown operation shall result in the ResourceFactory being unavailable to any subsequent calls to its object reference (i.e. it is released from the CORBA environment).
	SCA575		The shutdown operation shall raise the ShutdownFailure exception, if processing errors prevent the release of the ResourceFactory from the server side CORBA environment.

The shutdown operation will be unique to each ResourceFactory. Consequently, there are no functional requirements placed on the shutdown operation. The shutdown operation, upon completion, removes the ResourceFactory from the CORBA environment making it unavailable to further calls (SR:119) (see Table 4.11).

If some error is encountered during the shutdown process, the ShutdownFailure exception is raised (SCA575). In version 2.2, the IDL included the exception specification but no requirement explicitly stated that the exception be raised. So, although it was implicitly clear that an exception should be raised and virtually all implementations of the SCA specification did so, technically there was no requirement to do so. This minor omission was corrected by adding the requirement to SCA 2.2.1.

4.7 Port

The Port is the essential mechanism for connecting components to establish communications. The Port provides a common abstraction that is inherited by component-specific interfaces that inherit the Port interface. So, in essence, the Port interface in the SCA is a means of obtaining a reference to the actual interface implemented by a component. The interface implemented by the component provides the actual interface for performing the data transfer and control operations. Specific formats, data level protocols, and any other data structure or interpretation is implemented as part of the operational interface. Thus, the Port interface is used merely for establishing a connection between two operational interfaces (Figure 4.6).

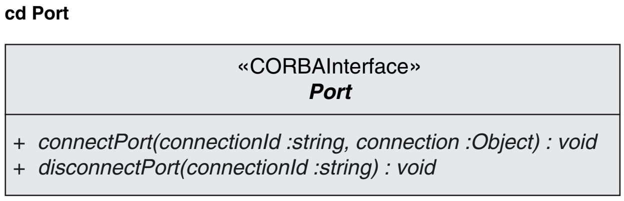

cd Port
«CORBAInterface»
Port
+ connectPort(connectionId :string, connection :Object) : void
+ disconnectPort(connectionId :string) : void

Figure 4.6. Port interface

As Figure 4.6 shows, only two functions are defined for the Port interfaces: connectPort and disconnectPort. As noted previously, the style of communications within a CORBA-based system follows a client-server style. The client application obtains a reference to a server interface that implements one or more operations defined by the IDL. Figure 4.7 provides a conceptional example of how the Port interface is used. The upper half of the figure shows the SCA portions of the connection process and the lower half shows the implementation.

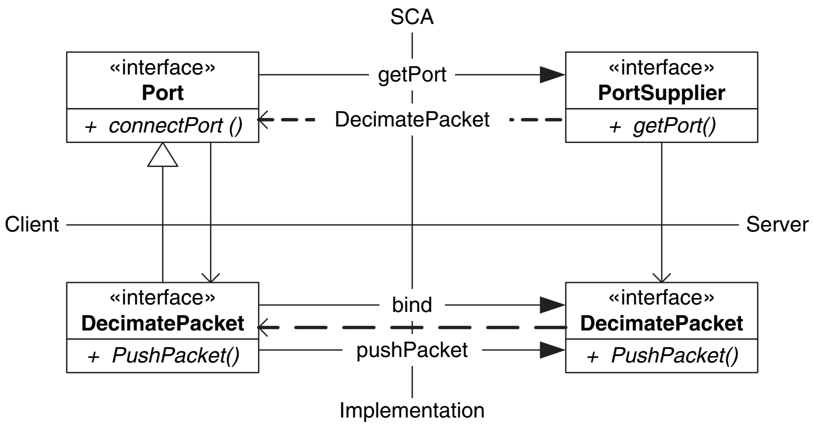


Figure 4.7. Establishing an SCA Port connection

In the example in Figure 4.7, there are two components within the application. The Client side component provides a sequence of packets that comprise the waveform in digital form. The packet data stream is to be processed through a decimation filter provided by a decimation filter component shown on the Server side. The Client obtains a reference to the Port object through the PortSupplier interface (see Section 4.2). The DecimatePacket interface object reference is returned as a Port object. Once the Port reference is obtained, the connectPort on the Client is called with the Port reference passed as an argument. The connectPort implementation then binds to the server endpoint that implements the PushPacket interface. Once the connection is established, shown in the figure by the dashed arrow from the Server DecimatePacket to the Client DecimatePacket, the Client can then issue the up-call to the Server and start sending the signal processing packet stream through the decimation filter.

The example shown in Figure 4.7 is a simple push operation and is likely to be implemented as a oneway call, i.e. there is no return data expected or provided by the server implementation. Hence, the client side need not wait for a return value. However, other interfaces are certainly valid and required for different needs. For example, the interface may implement not only a data path but also one or more functions to provide flow control.

The format of the data exchanged between any two interfaces is totally at the discretion of the designer and is only limited to the set of IDL data types. Thus the data exchanged may be discrete values, a stream of octets, or any other combination built on IDL data types.

It should be emphasized that the Port abstraction in the SCA establishes a connection between a client side component, typically a data source, and the server side that implements the interface, a data sink. Although the example in Figure 4.7 illustrates the process of establishing communications between the source and sink components using a CORBA interface, the same abstraction may be applied to other methods of data transport. So, for example, if a vendor of a single processing board provides a DSP and FPGA on the board and an flexible fabric interconnect bus between the two processors, the same connectPort interface could be used to establish a specific data path between components on the two processors. The difference would be that, at the lower-level transport, the actual data path between the two components would be realized by part of the vendor's Board Support Package (BSP) software which establishes a data path through the flexible fabric from one component to the other.

The connections to be established are unique to a particular application and are specified as part of the waveform's XML files. Specifically, the connections are specified within the Software Assembly Descriptor (SAD) file. The format and content of this and other waveform XML files are discussed in Part II of this book.

4.7.1 Exceptions

InvalidPort

The InvalidPort exception indicates one of two possible errors, and which error is the cause of the exception is identified by the value of the errorCode field in the exception. The interpretation of the errorCode value is as follows:

1. The Port component is invalid, i.e. it cannot be narrowed to an Object reference.
2. The Port name specified is not found.

A string message field is defined to provide some descriptive information regarding the cause of the exception.

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

OccupiedPort

The OccupiedPort exception is raised when an attempt is made to connect to a destination port that already has a connection.

```
exception OccupiedPort {
};
```

Note that the decision as to whether a Port is occupied or not is dependent on the implementation. It does not imply that if a component is already connected to a Port another component may not connect to the Port. A Port implementation may support multiple connections by providing a connection for the data provider and a different connection from a different component for control information. Both components would be connected to the same port but would be different connections. Multiple connections may also be supported through CORBA conventions such as enabling a thread-per-invocation model which supports multiple client connections to the same method by initiating a new thread to handle the up-call from the client.

4.7.2 Operations

connectPort

The connectPort operation establishes a connection to the port identified by the Object reference in the connection parameter. Several connections may be supported by a port. A unique Id is provided for this connection using the connectionId parameter. This name is used to identify each individual connection for tear down. The operation may raise either the InvalidPort or OccupiedPort exceptions described above.

The connectPort operation establishes only half of the association.

```
void connectPort (
     in Object connection,
     in string connectionId
)
raises (InvalidPort, OccupiedPort);
```

The connectPort operation establishes a connection to the port referenced by the Object reference parameter assigning it the string identifier provided in the connectionId parameter (SR:69) (see Table 4.12). If the port is already in use and cannot accept additional connections, then the OccupiedPort exception is raised (SR:71). If the object reference provided cannot be resolved to a valid port connection, e.g. the object reference cannot be narrowed to a valid port reference, then the InvalidPortException is raised (SR:70).

disconnectPort

Port connections are terminated using the disconnectPort operation. The operation uses the connectionId string to identify the connection uniquely at the time it was created with the

Table 4.12. connectPort requirements

Section	ID	Resp	Requirement
3.1.3.1.1.5.1.3	SR:69	WF, DS	The connectPort operation shall make a connection to the component identified by the input parameters.
3.1.3.1.1.5.1.5	SR:70	WF, DS	The connectPort operation shall raise the InvalidPort exception when the input connection parameter is an invalid connection for this Port.
3.1.3.1.1.5.1.5	SR:71	WF, DS	The connectPort operation shall raise the OccupiedPort exception when unable to accept the connections because the Port is already fully occupied.

connectPort operation. If the connectionId does not refer to a known connection, then the InvalidPort exception is raised.

```
void disconnectPort (
     in string connectionId
     )
raises (InvalidPort);
```

Table 4.13. disconnectPort requirements

Section	ID	Resp	Requirement
3.1.3.1.1.5.2.3	SR:72	WF, DS	The disconnectPort operation shall break the connection to the component identified by the input parameter.
3.1.3.1.1.5.2.5	SR:73	WF, DS	The disconnectPort operation shall raise the InvalidPort exception when the name passed to disconnectPort is not connected with the Port component.

The disconnectPort takes the connectionId provided and terminates the connection specified by the string (SR:72) (see Table 4.13). If the connectionId provided is not found within the internal table of connections maintained by the application, then the InvalidPort exception is raised (SR:73).

5

Devices and the Device Manager

5.1 Introduction

The ability to represent and manage the underlying physical hardware that implements the radio system is core to the concept of the SCA. The approach embodied within the SCA is to define a minimal set of interfaces that provide essential management and control capabilities for all devices within the radio system.

In the context of an SCA radio system, an SCA Device is a logical interface to the underlying physical hardware. This hardware includes any physical component that processes any part of the signal chain from the antenna through to the I/O connection.

Figure 5.1 illustrates some key concepts regarding the layered components of an SCA Device interface. The figure refers to an FPGA type of device but the same general abstraction layers can be applied to other types of hardware.

At the lowest layer, the physical device (an FPGA in this example), resides on a board that is sold by a vendor or manufacturer of the board. As shown in Figure 5.1, there is typically some type of control on the board that manages the physical interface between the board and the computer system in which it is integrated. The physical interface with the host computer may be a PCI type interface or some other type of standard for physical and electrical compatibility. Within the host computer, a device driver is loaded that provides the interface between the operating system and the hardware. The device driver provides the service routines that facilitate the exchange of control, status, and data between the operating system and the internal state information and processing on the board.

Moving up higher, the device service routines plug into the operating system and expose operating system level calls to the user applications using the device. The SCA Device implementation resides at the application level. Thus, there are a number of physical and logical abstraction layers that provide the basic interaction and control mechanism before implementation of an SCA Device is implemented. Therefore, the SCA Device interface provides a common abstraction of control and status that allows other SCA components and applications to utilize the device without having to integrate with the specific device interface API provided by the vendor. That does not mean that the SCA Device interface covers all the API calls to the device as provided by the manufacturer. The SCA Device

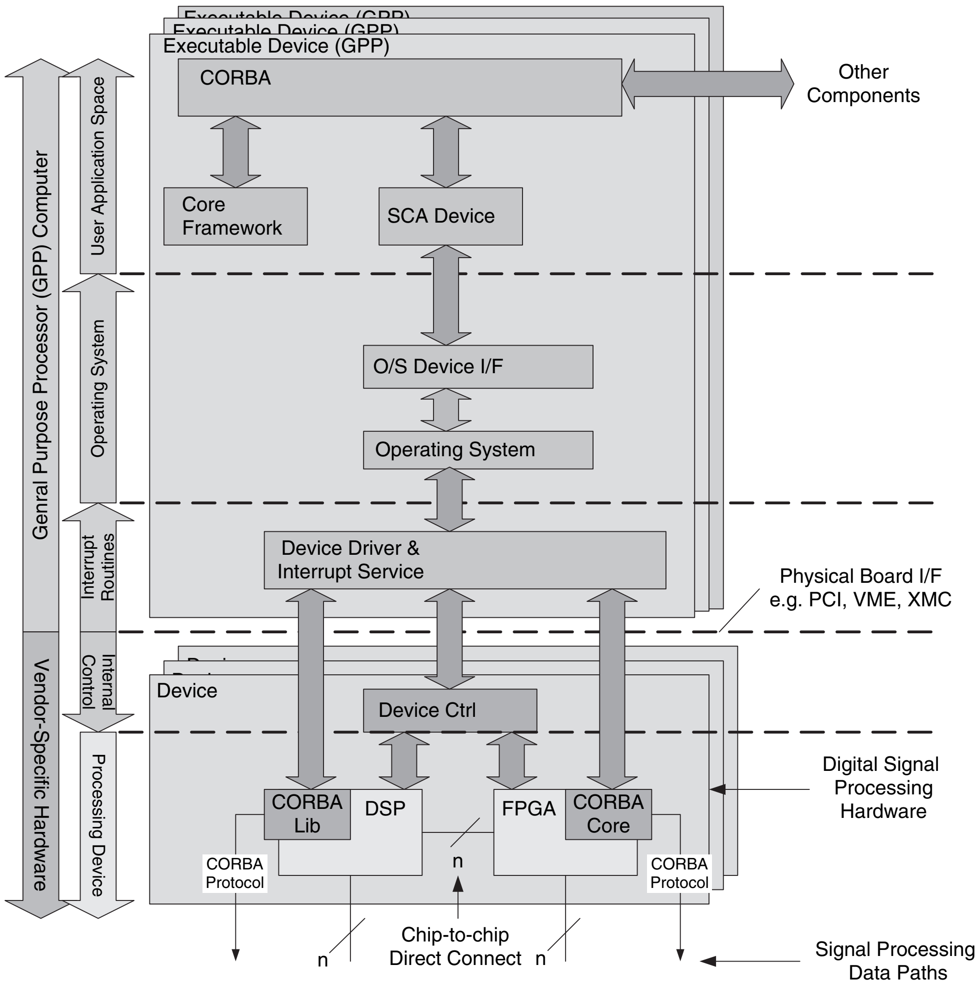


Figure 5.1. Layering in an SCA Device interface

interface is intended to provide a small set of essential, common interfaces for all devices. If there are extensions provided by the device manufacturer that are to be made available to SCA application components, then the Device interface would be extended by deriving another interface class from the Device interface.

5.1.1 SCA Device Abstraction

The SCA defines an abstraction hierarchy of three types of devices (as described below). The standard Device interface inherits from the Resource interface and, consequently, provides all of the interfaces already described for the Resource. The Device abstraction is used to allocate capacity on some physical resource within the system. For example, this may be a processor, and amplifier, a switch, or antenna. When a waveform application is instantiated by the ApplicationFactory, the collection of devices required to support the waveform are inspected and the capacity required by the waveform application is allocated. For some devices, the allocation may be a binary function. For example a high power amplifier may only be capable of supporting a single waveform at a time. Other device capacity allocations may be partial allocation of the devices total capacity. For example, the memory used by a waveform application may only be a portion of the total memory available.

A Device in the SCA is an abstraction of the underlying physical hardware within the software radio. The SCA has defined three basic device types to encompass the range of the hardware found within the radio system:

- **Device** – This type provides the basic representation and control interface and serves as the basic interface for the other device types.
- **LoadableDevice** – This type extends the basic Device definition by adding the interface to load data or signal processing code.
- **ExecutableDevice** – This type extends the LoadableDevice by adding the ability to start/stop processes.

Figure 5.2 illustrates the organization of interfaces for the SCA Device hierarchy and the AggregateDevice. As shown, the Device interface inherits from the Resource. Thus, all of the Resource requirements, as well as those inherited by the Resource, apply to the Device as well.

As a logical abstraction, the Device interface is defined as the interface to a set of zero or more hardware devices. This opens the door for a large, monolithic Device implementation, i.e. a single device that encompasses a potentially large set of physical devices. Although this is perfectly legal in the SCA, it is generally a poor design practice in that it reduces the degree of granularity of control within the system by hiding multiple devices and the control of those devices behind a high-level abstract interface. For example, a signal processing card may have multiple FPGAs, one or more DSPs, and other physical resources on the single card. Although it is within the bounds of the SCA to have a single Device interface for the set of processors on the board, it is easy to see that it is more advantageous to have Device interfaces for each of the processor on the board. Individual Device interfaces provide greater visibility and control for the physical processing resources.

An AggregateDevice is also defined in the SCA, and is typically used to provide a convenient mechanism for maintaining a logical collection of devices. As defined in the SCA, the AggregateDevice is merely a mechanism for maintaining a collection of SCA Devices. Because it does not inherit the Device interface, it cannot be used to implement a CompositeDevice. The distinction is that a CompositeDevice represents a physical aggregation of Devices such as several FPGAs on a single card. Each FPGA would be a Device and the board that contains the FPGAs would be the CompositeDevice. The Device interface contains a compositeDevice attribute. Using this attribute it is possible to build a CompositeDevice. This enables a Device that supports the normal Device behaviors, e.g. start, stop, etc., and the ability to manage the set of processors as a logical entity. However, each FPGA can be configured individually as a LoadableDevice (Section 5.3). This concept is discussed in more detail in the sections on Device and AggregateDevice (Sections 5.2 and 5.5).

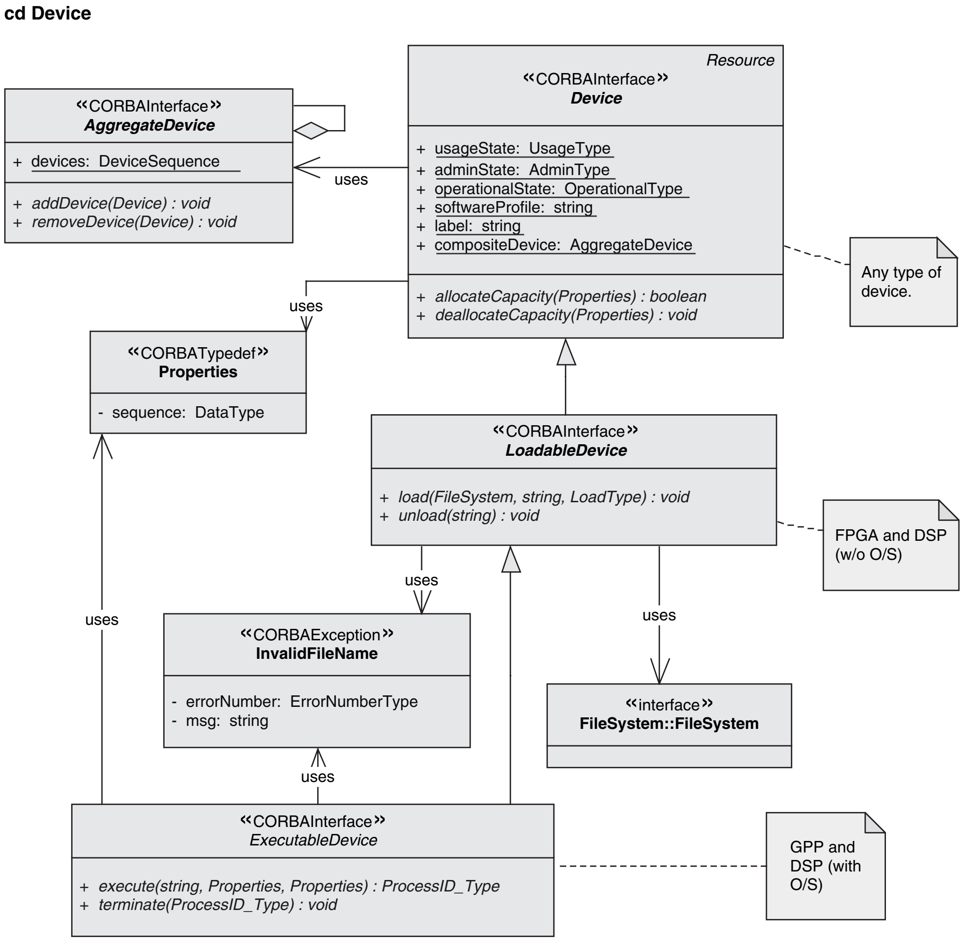

cd Device
«CORBAInterface» AggregateDevice
+ devices: DeviceSequence
+ addDevice(Device) : void
+ removeDevice(Device) : void
uses
Resource
«CORBAInterface» Device
+ usageState: UsageType
+ adminState: AdminType
+ operationalState: OperationalType
+ softwareProfile: string
+ label: string
+ compositeDevice: AggregateDevice
+ allocateCapacity(Properties) : boolean
+ deallocateCapacity(Properties) : void
Any type of device.
uses
«CORBATypedef» Properties
- sequence: DataType
«CORBAInterface» LoadableDevice
+ load(FileSystem, string, LoadType) : void
+ unload(string) : void
FPGA and DSP (w/o O/S)
uses
uses
uses
«CORBAException» InvalidFileName
- errorNumber: ErrorNumberType
- msg: string
«interface» FileSystem::FileSystem
uses
«CORBAInterface» ExecutableDevice
+ execute(string, Properties, Properties) : ProcessID_Type
+ terminate(ProcessID_Type) : void
GPP and DSP (with O/S)

Figure 5.2. Device hierarchy and interfaces

5.2 Device

The SCA Device is the top-level abstraction of a physical device in the SCA system. The Device inherits from the Resource as illustrated in Figure 5.3. The definition provides several attributes that define state information about the device. In order to manage the physical resources in any system, one must be able to manage the allocation of the resources to support operations and processes. Thus, the key capabilities defined at the top level are the allocateCapacity and deallocateCapacity. As the names imply, these interfaces provide the essential behavior of allocating and deallocating some level or capacity of one or more resources entities associated with the Device.

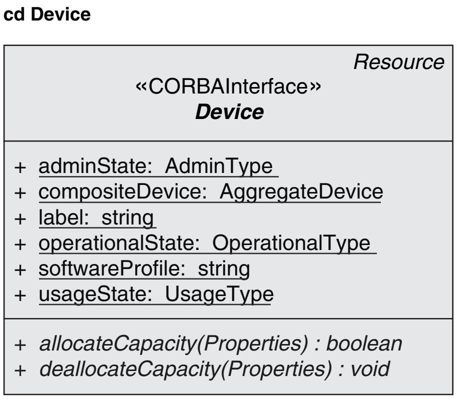

cd Device
Resource
«CORBAInterface»
Device
+ adminState: AdminType
+ compositeDevice: AggregateDevice
+ label: string
+ operationalState: OperationalType
+ softwareProfile: string
+ usageState: UsageType
+ allocateCapacity(Properties) : boolean
+ deallocateCapacity(Properties) : void

Figure 5.3. Device interface

5.2.1 Exceptions

There are two essential exceptions that are defined for the Device operations. InvalidState and InvalidCapacity.

InvalidState

This exception is raised when the Device is requested to be set into an invalid state.

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

InvalidCapacity

When a capacity allocation is requested, it must be checked against the resource type and amount available. If the resource requested is undefined or the amount requested exceeds the amount available, then this exception is raised. Other conditions may also raise an invalidCapacity exception and are dependent on the type of resource on the device.

```
exception InvalidCapacity {
   string msg;
   Properties capacities;
};
```

5.2.2 Types and Constants

AdminType

There are three possible administrative states defined by the AdminType. In the `LOCKED` state the device does not accept any requests for actions from external entities but does accept calls in the `UNLOCKED` state. The `SHUTTING_DOWN` state, as the name implies, specifies that the device is in the process of terminating and, consequently, may be in an unstable state as it shuts down. Therefore, as with the LOCKED state, the device does not accept any external calls while in the SHUTTING_DOWN state. The operational state also affects whether or not the device responds to external calls.

```
enum AdminType {
       LOCKED,
       SHUTTING_DOWN,
       UNLOCKED
};
```

OperationalType

The device may have two possible operational states. The ENABLED state means that the device has initialized, is in a stable state, and is ready to accept external requests, i.e. it is in service. If the device has been removed from service or is temporarily unavailable, the operational state will be DISABLED.

```
enum OperationalType {
      ENABLED,
      DISABLED
};
```

UsageType

The usage type reflects the activity of the device. It reflects the capacity allocations of one or more resources on the device. Three simple forms are used. A usage state of IDLE means that the device currently has none of its resources allocated. A usage state value of BUSY means that the device has allocated all of the resource capacities. When the value is ACTIVE, some of the resources of the device have been allocated but additional capacities are available to be allocated to other tasks.

```
enum UsageType {
      IDLE,
      ACTIVE,
      BUSY
};
```

5.2.3 Attributes

There are six attributes defined by the Device. These are shown in the following IDLs. All of the attributes except for the adminState are readonly. Each of the attributes are discussed in the following subsections.

label

The label attribute provides a user-readable name for the Device (SR:403) (see Table 5.1). This is typically used as the name of the Device in user interfaces to provide a more easily recognizable reference for the Device.

Table 5.1. Device label attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.5	SR:403	DS	The readonly label attribute shall contain the Device's label.

```
readonly attribute string label;
```

softwareProfile

The softwareProfile attribute stores the file name of the Software Package Descriptor (SPD) that provides the software implementation information for the Device.

```
readonly attribute string softwareProfile;
```

The softwareProfile attribute may contain a file reference, i.e. a file name, of the SPD file or the actual XML that specifies the software implementation information for the Device (SR:401) (see Table 5.2). There is no indication as to whether the text data on the attribute is a file name or the actual XML. This is an implementation detail of the SCA Device. However, in most implementations, a reference to the SPD file is stored in the attribute.

Table 5.2. Device softwareprofile requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.4	SR:401	DS	The readonly softwareProfile attribute shall contain either a profile DTD element with a file reference to the SPD profile file or the XML for the SPD profile.

compositeDevice

The compositeDevice attribute maintains a reference to the instance of the AggregateDevice, if any, of which the current Device is a member.

```
readonly attribute AggregateDevice compositeDevice;
```

The compositeDevice attribute has a single requirement specifying the legal values that may be stored on the attribute (see Table 5.3).

The compositeDevice attribute stores the AggregateDevice reference containing this Device and zero or more other Devices. If the Device is not part of an AggregateDevice, then the attribute contains a nil reference (SR:404). This provides the capability to represent a compositeDevice within a Device by instantiating an AggregateDevice and using it to store and manage the set of Devices as a logical unit. This is illustrated in Figure 5.4.

Table 5.3. compositeDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.6	SR:404	DS	The readonly compositeDevice attribute shall contain the object reference of the aggregateDevice, with which this Device is associated, or a nil CORBA object reference if no association exists.

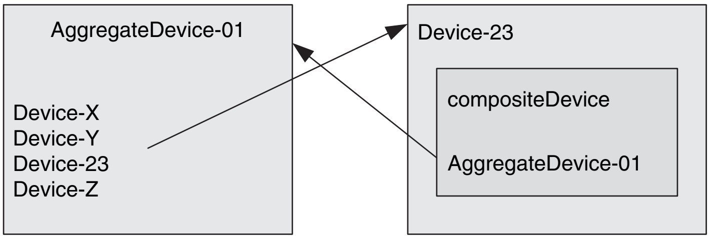


Figure 5.4. compositeDevice attribute

As shown in the figure, Device-23 is a member of AggregateDevice-01. Thus the AggregateDevice instance has a reference to each of the devices specified as part of the AggregateDevice. Device-23, as well as each of the devices of the AggregateDevice, maintains a reference to the AggregateDevice instance in which it is a member.

adminState

The basic requirement states simply that the adminState attribute contains the adminState value (SR:386) (see Table 5.4).

```
attribute AdminType adminState;
```

Whenever the adminState value changes, the Device must send an event to the DomainManager describing the state change (SR:390). There are five fields that comprise the event change message:

- **ProducerId** – This field contains the Id of the Device. This is the identifier attribute of the Device producing the message (SR:391).
- **SourceId** – Since the producer of the event is also the source of the event, the SourceId is the Id of the Device (SR:392).
- **stateChangeCategory** – Since this event is generated because of a change to the adminState, the stateChangecategory field is set to `ADMINISTRATIVE_STATE_EVENT` (SR:393).
- **stateChangeFrom** – This field is set to the initial value of the adminState attribute (SR:394).
- **stateChangeTo** – This field is set to the new value of the adminState attribute (SR:394).

Table 5.4. Device adminState requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.2	SR:386	DS	The adminState attribute shall contain the device's admin state value.
3.1.3.2.4.4.2	SR:387	DS	The adminState attribute shall only allow the setting of `LOCKED` and `UNLOCKED` values, where setting `'LOCKED'` is only effective when the adminState attribute value is `UNLOCKED`, and setting `'UNLOCKED'` is only effective when the adminState attribute value is `LOCKED` or `SHUTTING_DOWN`.
3.1.3.2.4.4.2	SR:388	DS	The adminState attribute, upon being commanded to be `LOCKED`, shall transition from the `UNLOCKED` to the `SHUTTING_DOWN` state and set the adminState to LOCKED for its entire aggregation of Devices (if it has any).
3.1.3.2.4.4.2	SR:389	DS	The adminState shall then transition to the `LOCKED` state when the Device's usageState is `IDLE` and its entire aggregation of Devices are `LOCKED`.
3.1.3.2.4.4.2	SR:390	DS	Whenever the adminState attribute changes, the Device shall send an event to the Incoming Domain Management event channel with event data consisting of a StateChangeEventType. The event data will be populated as follows:
3.1.3.2.4.4.2	SR:391	DS	The producerId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.2	SR:392	DS	The sourceId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.2	SR:393	DS	The stateChangeCategory field shall be `ADMINISTRATIVE_STATE_EVENT`.
3.1.3.2.4.4.2	SR:394	DS	The stateChangeFrom and stateChangeTo fields shall reflect the adminState attribute value before and after the state change, respectively.

The remainder of the requirements describe the conditions under which the adminState attribute is changed and which state transitions are legal.

As shown in Figure 5.5, upon Device start, the adminState value is set to `UNLOCKED`. This initial case is implicitly defined in the state diagram. However, it is not explicitly specified as a requirement. As defined in the specification, the setting of the adminState to `UNLOCKED` is only valid when the adminState is `LOCKED` or `SHUTTING_DOWN` (SR:387). When the Device is `UNLOCKED` and commanded to be `LOCKED`, it transitions to the `SHUTTING_DOWN` state and the Device commands each of the Devices contained with the AggregateDevice, if

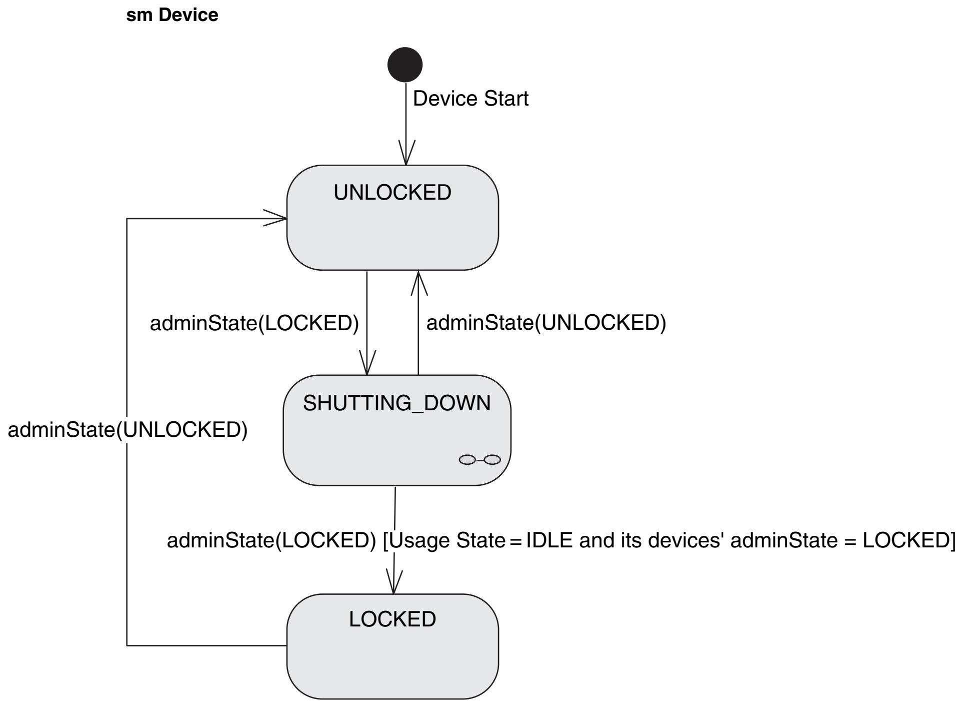


Figure 5.5. State diagram for the adminState attribute

any, stored on the compositeDevice attribute to be `LOCKED` (SR:388). This state is an interim state that allows the Device to perform all necessary control and configuration operations, enabling it to reach a stable state prior to proceeding to the `LOCKED` state.

Once i) the Device has completed all internal clean-up routines for itself; ii) each of the Devices contained in the AggregateDevice structure on the compositeDevice attribute has completed its clean-up routines and transitioned to the `LOCKED` state; and iii) the usage state for each of the aggregate devices and the Device for which the `LOCKED` state has been commanded has been set to `IDLE`, then the Device transitions to the `LOCKED` state.

While the Device is in the process of the `SHUTTING_DOWN` state and is waiting for its aggregate devices to transition to the `LOCKED` state, the Device may be commanded to transition back to the `UNLOCKED` state, as previously noted in requirement SR:387. Thus, as illustrated in Figure 5.6, there is essentially a lower level state machine that handles a request to transition back to the `UNLOCKED` state. If the aggregate devices have completed their shutdown and the Device's usageState is set to `IDLE`, then the Device transitions to the `LOCKED` state (SR:389). If the set of aggregate devices has not completed its shutdown process, or the Device is not yet in the `IDLE` usageState, then it transitions back to the `UNLOCKED` state.

The above scenario is not explicitly defined within the SCA specification and there are variations on how the state machine may be represented. Also, how the Device implements the transition back to the `UNLOCKED` state must be addressed, particularly if some subset of

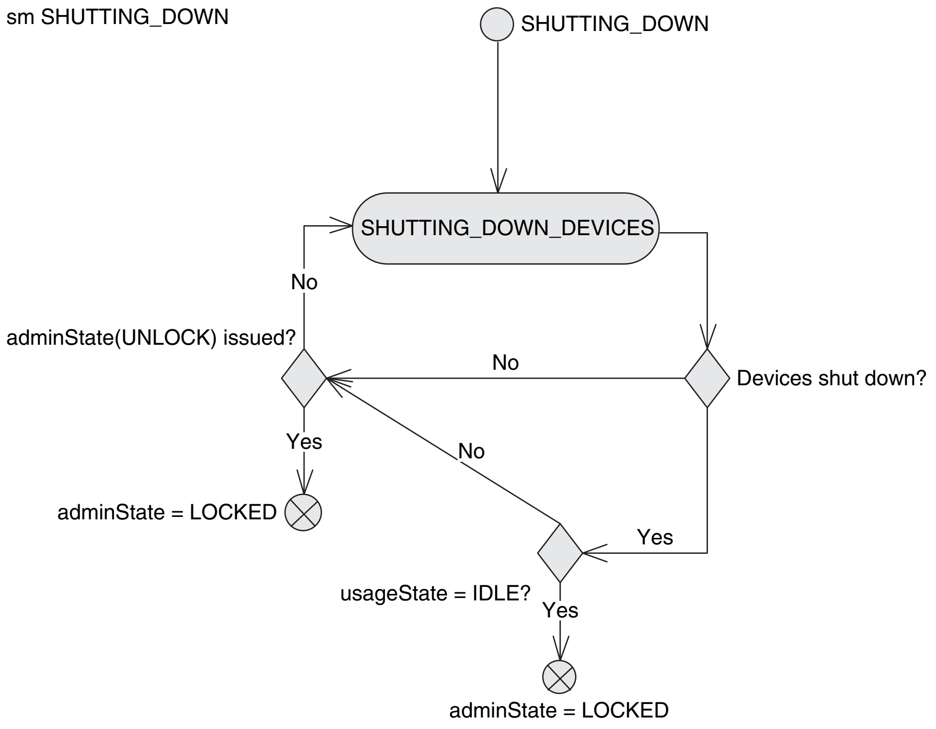


Figure 5.6. Sub-state diagram of the `SHUTTING_DOWN` state

the aggregate devices on the Device have already performed their transition to the `LOCKED` state. Each of the devices contained in the aggregateDevice structure on the compositeDevice attribute must be commanded to transition back to the `UNLOCKED` state. Furthermore, in order to be in a consistent state, each of the aggregate devices must be set back to the `UNLOCKED` state prior to the top-level Device transitioning back to the `UNLOCKED` state.

usageState

The usageState attribute contains the current state of the Device (see Table 5.5).

```
readonly attribute UsageType usageState;
```

There are three legal values for this attribute (SR:345):

- **`IDLE`** – This value indicates that the Device has been initialized and is available for use.
- **`ACTIVE`** – This state indicated that the Device has some portion of its capabilities in use but still has additional capacity that could be utilized.
- **`BUSY`** – When all of the capacity available on the Device is in use, the usageState is set to `BUSY` indicating that the Device is not available for any additional requests for use.

Table 5.5. usageState requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.1	SR:345	DS	The readonly usageState attribute shall contain the Device's usage state (`IDLE`, `ACTIVE`, or `BUSY`). UsageState indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for allocation at that instant.
3.1.3.2.4.4.1	SR:381	DS	Whenever the usageState attribute changes, the Device shall send an event to the Incoming Domain Management event channel with event data consisting of a StateChangeEventType. The event data will be populated as follows:
3.1.3.2.4.4.1	SR:382	DS	The producerId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.1	SR:383	DS	The sourceId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.1	SR:384	DS	The stateChangeCategory field shall be `USAGE_STATE_EVENT`.
3.1.3.2.4.4.1	SR:385	DS	The readonly usageState attribute shall contain the Device's usage state (`IDLE`, `ACTIVE`, or `BUSY`). UsageState indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for allocation at that instant.

Whenever the usageState value changes, the Device must send an event to the DomainManager describing the state change (SR:381). There are five fields that comprise the event change message:

- **producerId** – This field contains the Id of the Device. This is the identifier attribute of the Device producing the message (SR:382).
- **sourceId** – Since the producer of the event is also the source of the event, the SourceId is the Id of the Device (SR:383).
- **stateChangeCategory** – Since this event is generated because of a change to the adminState, the stateChangeCategory field is set to `USAGE_STATE_EVENT` (SR:384).
- **stateChangeFrom** – This field is set to the initial value of the adminState attribute (SR:385).
- **stateChangeTo** – This field is set to the new value of the adminState attribute (SR:385).

operationalState

The operationalState attribute identifies the current state of the Device (see Table 5.6).

Table 5.6. Device operationalState requirements

Section	ID	Resp	Requirement
3.1.3.2.4.4.3	SR:395	DS	The readonly operationalState attribute shall contain the device's operational state (`ENABLED` or `DISABLED`).
3.1.3.2.4.4.3	SR:396	DS	Whenever the operationalState attribute changes, the Device shall send an event to the Incoming Domain Management event channel with event data consisting of a StateChangeEventType. The event data will be populated as follows:
3.1.3.2.4.4.3	SR:397	DS	The producerId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.3	SR:398	DS	The sourceId field shall be the identifier attribute of the Device.
3.1.3.2.4.4.3	SR:399	DS	The stateChangeCategory field shall be `OPERATIONAL_STATE_EVENT`.
3.1.3.2.4.4.3	SR:400	DS	The stateChangeFrom and stateChangeTo fields shall reflect the operationalState attribute value before and after the state change, respectively.

```
readonly attribute OperationalType operationalState;
```

There are two values for the operationalState attribute (SR:395):

- **ENABLED** – The Device is operational and available to process calls from clients.
- **DISABLED** – The Device is not currently accepting calls from clients and may not be in a stable state.

Whenever the operationalState value changes, the Device must send an event to the DomainManager describing the state change (SR:396). There are five fields that comprise the event change message:

- **ProducerId** – This field contains the Id of the Device. This is the identifier attribute of the Device producing the message (SR:397).
- **SourceId** – Since the producer of the event is also the source of the event, the SourceId is the Id of the Device (SR:398).
- **stateChangeCategory** – Since this event is generated because of a change to the adminState, the stateChangeCategory field is set to `OPERATIONAL_STATE_EVENT` (SR:399).
- **stateChangeFrom** – This field is set to the initial value of the adminState attribute (SR:400).
- **stateChangeTo** – This field is set to the new value of the adminState attribute (SR:400).

5.2.4 Operations

allocateCapacity

The allocateCapacity is the basic call used to request a Device to allocate the capacities specified in the capacities argument. In the general interpretation of the allocateCapacity requirements, the call is passed to the Device instance which checks if it has the capacities requested and, if they are available, the Device allocates the capacities specified and returns true indicating success.

In practicality, this call needs to be passed to the actual Device instance when the Device may have allocations that occur outside the scope of the SCA. For example, in the case of a GPP, an SCA ApplicationFactory may request a certain amount of memory to be allocated. In addition to the SCA request for some memory resource, other applications executing within the processor and operating system may require memory. For example, a process management application running on the GPP outside the SCA may use some block of memory resulting in a decrease in the available memory independently of the SCA environment.

However, for a significant number of devices, the capacity allocation may be shortcircuited by maintaining a table of device resources and their allocation internally. For example, a simple antenna has a simple capacity model and typically does not implement sophisticated behavior as part of its operational behavior. It is either in use or not. Thus, such a simple, binary allocation could be maintained as part of a device table within the SCA Core Framework rather than passing the call through to a separate process or thread implementing the device.

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

The allocateCapacity call provides a sequence of Properties that specify a value representing the amount of the capacity to allocate for the Property specified (see Table 5.7). If the capacity of the Property specified is available, the Device adminState is `UNLOCKED`, the operationalState is `ENABLED`, and the usageState is not `BUSY`, the total capacity of the Device is reduces by the requested amount (SR:405).

For example, a Device Property may be memory and the allocateCapacity may request some amount of memory to be allocated for use by the object requesting the allocation. Note that, in the case of memory on a processor, this typically corresponds to the amount of memory to be decremented from the total capacity device.

After a capacity is allocated for a device, if the Device determines that it has no further capacity that can be requested, the usageState is set to `BUSY` (SR:406). However, if capacity is still available for allocation, then the usageState attribute is set to `ACTIVE` (SR:407). These requirements do not explicitly address how to handle Devices that have multiple Properties with capacities that may be allocated. For example, it may be possible to request an allocation of capacities on a Device that exhausts the capacity available for a single Property while other types of capacity may be available. In this case, it is up to the developer to decide whether the Device's usageState should be set to `BUSY` when any one of a set of capacities is depleted or to remain `ACTIVE` if it is possible to allocate other capacities independently of the depleted capacity.

Table 5.7. Device allocateCapacity requirements

Section	ID	Resp	Requirement
3.1.3.2.4.5.1.3	SR:405	DS	The allocateCapacity operation shall reduce the current capacities of the Device based upon the input capacities parameter, when the Device's adminState is `UNLOCKED`, Device's operationalState is `ENABLED`, and Device's usageState is not `BUSY`.
3.1.3.2.4.5.1.3	SR:406	DS	The allocateCapacity operation shall set the Device's usageState attribute to `BUSY`, when the Device determines that it is not possible to allocate any further capacity.
3.1.3.2.4.5.1.3	SR:407	DS	The allocateCapacity operation shall set the usageState attribute to `ACTIVE`, when capacity is being used and any capacity is still available for allocation.
3.1.3.2.4.5.1.3	SR:408	DS	The allocateCapacity operation shall return 'True', if the capacities have been allocated, or 'False', if not allocated.
3.1.3.2.4.5.1.3	SR:409	DS	The allocateCapacity operation shall raise the InvalidCapacity exception, when the capacities are invalid or the capacity values are the wrong type or Id.
3.1.3.2.4.5.1.3	SR:410	DS	The allocateCapacity operation shall raise the InvalidState exception, when the Device's adminState is not `UNLOCKED` or operationalState is `DISABLED`.

The allocateCapacity call returns a value of 'True' if the capacities requested were successfully allocated (SR:408). The requirement also states that a value of 'False' will be returned in the event that the capacities requested were not successfully allocated. However, when using the exception mechanism of a programming language, the return value specified by a function is not provided if the function encounters an error and raises the exception. If any of the capacities or the values specified in the argument are invalid, the allocateCapacity call will raise the InvalidCapacity exception (SR:409).

In order to allocate capacity, the operationalState of the Device may not be set to DISABLED and the adminState may not be `UNLOCKED`. If the allocateCapacity is called when either of these states does not meet the conditions, an InvalidState exception is raised (SR:410).

deallocateCapacity

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

```
void deallocateCapacity (
```

Table 5.8. deallocateCapacity requirements

Section	ID	Resp	Requirement
3.1.3.2.4.5.2.3	SR:411	DS	The deallocateCapacity operation shall adjust the current capacities of the Device based upon the input capacities parameter.
3.1.3.2.4.5.2.3	SR:412	DS	The deallocateCapacity operation shall set the usageState attribute to `ACTIVE` when, after adjusting capacities, any of the Device's capacities are still being used.
3.1.3.2.4.5.2.3	SR:413	DS	The deallocateCapacity operation shall set the usageState attribute to `IDLE` when, after adjusting capacities, none of the Device's capacities are still being used.
3.1.3.2.4.5.2.3	SR:414	DS	The deallocateCapacity operation shall set the adminState attribute to `LOCKED` as specified in 3.1.3.2.4.4.2.
3.1.3.2.4.5.2.5	SR:415	DS	The deallocateCapacity operation shall raise the InvalidCapacity exception, when the capacity Id is invalid or the capacity value is the wrong type.
3.1.3.2.4.5.2.5	SR:416	DS	The deallocateCapacity operation shall raise the InvalidState exception, when the Device's adminState is `LOCKED` or operationalState is `DISABLED`.

```
    in Properties capacities
   )
 raises (InvalidCapacity, InvalidState);
```

The deallocateCapacity is used to make a Device's resources available for use when the current application no longer requires the resources. The capacities specified in the call will be returned to the available pool of the Device's capacities (SR:411) (see Table 5.8). If any of the Device's capacities are still in use after adjusting the capacities, the usageState will be set to `ACTIVE` to denote that the Device is still in use by one or more applications (SR:412). If the capacities returned to the Device result in no capacities in use then the usageState is set to `IDLE` (SR:413).

As with the allocation of capacities, if the capacity Id is invalid or the value is the wrong type, the deallocateCapacity raises the InvalidCapacity exception (SR:415). If the Device adminState is set to `LOCKED` or the operationalState is `DISABLED`, the deallocateCapacity raises the InvalidState exception (SR:416).

releaseObject

The releaseObject is defined as part of the LifeCycle IDL. However, there are specific requirements that must be satisfied when a Device is torn down and released. The requirements are discussed below and shown in Table 5.9.

Table 5.9. releaseObject requirements

Section	ID	Resp	Requirement
3.1.3.2.4.5.3.3	SR:418	DS	The releaseObject operation shall call the releaseObject operation on all of the Device's aggregated Devices (i.e. those Devices that are contained within the AggregateDevice's devices attribute).
3.1.3.2.4.5.3.3	SR:419	DS	The releaseObject operation shall transition the Device's adminState to `SHUTTING_DOWN` state, when the Device's adminState is `UNLOCKED`.
3.1.3.2.4.5.3.3	SR:420	DS	The releaseObject operation shall cause the Device to be unavailable (i.e. released from the CORBA environment, and its logical Device's process terminated on the OS when applicable), when the Device's adminState transitions to `LOCKED`, meaning its aggregated Devices have been removed and the Device's usageState is `IDLE`.
3.1.3.2.4.5.3.3	SR:421	DS	The releaseObject operation shall cause the removal of its Device from the Device's compositeDevice.
3.1.3.2.4.5.3.3	SR:422	DS	The releaseObject operation shall unregister its Device from its DeviceManager.
3.1.3.2.4.5.3.5	SR:423	DS	The releaseObject operation shall raise the ReleaseError exception when releaseObject is not successful in releasing a logical Device due to internal processing errors that occurred within the Device being released.

When a Device receives the releaseObject call, it sets the value of adminState to `SHUTTING_DOWN` (SR:419). If the Device as a list of Devices in an AggregateDevice instance on the compositeDevice attribute, it issues the releaseObject call on all devices contained within the instance of AggregateDevice on the compositeDevice attribute (SR:418). As each Device in the aggregate device list completes the releaseObject call, it is removed from the list of aggregate devices (SR:421). When the list of devices in the compositeDevice attribute is empty, the AggregateDevice instance is destroyed and the Device transitions to the `LOCKED` state for the adminState attribute. The Device then issues the unregisterDevice call to the DeviceManager that instantiated the Device (SR:422) removing it from the list of registeredDevices on the DeviceManager. At this point the Device instance is released from the CORBA environment making it unavailable and the Device process is terminated (SR:420).

5.3 LoadableDevice

The LoadableDevice extends the definition of the Device by supporting the loading of 'software' to a device and unloading 'software' from the device. The term 'software' is highlighted because, in the context of a LoadableDevice, the term is applied to any image that may be loaded or unloaded. So, although the typical interpretation of a program that might run on a GPP such as found in a personal computer is true, the LoadableDevice also encompasses loading programs on a DSP, and bit images into a FPGA. Thus, for the remainder of this section on LoadableDevice, the term image will be used to denote the item that is loaded or unloaded.

5.3.1 Types

LoadType

As shown in the IDL declaration in Figure 5.7, the LoadableDevice inherits from and extends the Device. There are four types of images that may be loaded:[1]

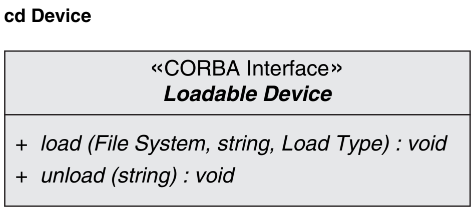


Figure 5.7. LoadableDevice interface

```
enum LoadType {
     KERNEL_MODULE,
     DRIVER,
     SHARED_LIBRARY,
     EXECUTABLE
};
```

- **KERNEL_MODULE** – This type encompasses images that are loaded as part of or extensions to, the operating system. For example, this might include an object file that provides a specific extension to the operating system for a operating system call or service.
- **DRIVER** – This type typically encompasses the low-level drivers that bridge the gap between a hardware device, such as a network card or Analog to Digital Converter (ADC), and the operating system or, if no operating system is used (i.e. on a DSP), bridges the gap to the program that might use the device directly.

[1] *The above types are specified in the Software Package Descriptor (SPD) XML file that describes the implementation details of the implementation. It can be said, in looking at the above set of load types, that the types defined are strongly oriented towards a GPP running an operating system. Load types that provide bit images for an FPGA or a program that runs within a DSP without an operating system do not have a corresponding load type. Consequently, these loads must be 'coerced' into one of the above types. Extending the load types to include these types of images could be accomplished by adding types such as* `NATIVE` *for a DSP or GPP program running natively on the processor without an operating system. The type* `BIT_IMAGE` *might be used for an FPGA load. For now, however, these loads must utilize one of the above in order to be compliant with the specification.*

- **SHARED_LIBRARY** – A shared library is a collection of code that typically performs a set of logically related functions that has been compiled and organized in a single file as a set of binary routines that may be integrated, i.e. linked, to a program. A library is shared when only a single copy of the code is loaded into the processor's memory and multiple programs can call the routines in the library.
- **EXECUTABLE** – The executable type refers to a program that is loaded and run under the control of an operating system, e.g. Linux, Windows, VxWorks, Integrity, and has a process Id or number that uniquely identifies the program within the operating system.

5.3.2 Exceptions

InvalidLoadKind

The InvalidLoadKind exception indicates that the LoadType provided is not one of the valid, defined types (see Section 5.3.1). No additional information is provided as part of the exception.

```
exception InvalidLoadKind {
};
```

LoadFail

The LoadFail exception indicates that the requested load operation did not successfully complete. An errorNumber is provided to identify the type of error encountered. A string is also part of the exception structure to provide some additional information.

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

5.3.3 Operations

load

The load operation provides the functionality to load an image onto a device. Typically, a LoadableDevice is a processor such as an FPGA or DSP (as noted earlier, a DSP is included as a type of LoadableDevice when it does not provide an operating system that enables separate processes that may be identified and controlled through a unique process Id). However, a LoadableDevice may also include Programmable Logic Arrays, digital matrix or crossbar switches that provide data pathways based on a binary image that is loaded into those devices.

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

The load call is provided with the name of the file to be loaded, the SCA FileSystem in which the file resides, and a LoadType parameter.

When a LoadableDevice receives the load call, it locates the file containing the image to be loaded as specified in the fileName argument of the call and, based on the loadKind argument in the call, loads the file on the LoadableDevice (SR:426) (see Table 5.10). The load types supported by the load operation on the LoadableDevice must include those types stated in the LoadType allocation properties specified in the Domain profile XML files (SR:427).[2]

Table 5.10. LoadableDevice load requirements

Section	ID	Resp	Requirement
3.1.3.2.5.5.1.3	SR:426	DS	The load operation shall load a file on the specified device based upon the input loadKind and fileName parameters using the input FileSystem parameter to retrieve the file.
3.1.3.2.5.5.1.3	SR:427	DS	The load operation shall support the load types as stated in the Device's software profile LoadType allocation properties.
3.1.3.2.5.5.1.3	SR:428	DS	The load operation shall keep track of the number of times a file has been successfully loaded.
3.1.3.2.5.5.1.5	SR:429	DS	The load operation shall raise the InvalidState exception when the Device's adminState is not `UNLOCKED` or operationalState is `DISABLED`.
3.1.3.2.5.5.1.5	SR:430	DS	The load operation shall raise the InvalidLoadKind exception when the input loadKind parameter is not supported.
3.1.3.2.5.5.1.5	SR:431	DS	The load operation shall raise the InvalidFileName exception when the file designated by the input filename parameter cannot be found.
3.1.3.2.5.5.1.5	SR:432	DS	The load operation shall raise the LoadFail exception when an attempt to load the device is unsuccessful.

As noted in the IDL definition, the fileName argument specifies the string name of the file to be loaded. The SCA FileSystem containing the file is also supplied. This provides

[2] *The load types defined as valid do not effectively capture the range of images, binaries, programs, etc. that may be loaded. See Section 5.3.1 for more details.*

the information necessary to locate and open the file based on the SCA FileSystem that has the mount point for the underlying platform file system. However, for a load operation to be successful for a GPP and operating system, i.e. to load a file to the GPP from the underlying operating file system, the SCA FileSystem and file string must be resolved to a file descriptor (FD) that the operating system can successfully load.

The LoadableDevice also maintains a reference count for files loaded (SR:428). This is typically used to manage the load process such that repeated load requests for the same file simply increment the reference count, thereby maintaining only a single copy for the load. This is the same strategy used internally by an operating system for a shared library. In the case of a LoadableDevice, such as an FPGA however, it is unclear as to the use of the load count at the SCA level other than for informational purposes.

The load operation is only valid when the LoadableDevice is `ENABLED` and `UNLOCKED`. If it is in any other state when the load call is received, the load fails and an InvalidState exception is raised (SR:429).

If the loadKind parameter does not match one of the defined types, then the load fails raising an InvalidLoadKind exception (SR:430). Thus, as noted previously, for FPGA and DSP loads, the loadKind must use one of the existing types, even when there is not a good match to the type of load actually being performed.

Also, if the file referenced by the fileName and FileSystem parameters cannot be found, an InvalidFileName exception is raised (SR:431).

If, for any reason other than those noted above, the load call fails, the Loadfail exception is raised (SR:432) indicating that some error condition prevented the call from successfully completing.

As described above, when a load request fails, the LoadFail exception is raised (SR:424) and includes an error number indicating the type of error encountered (Table 5.11). Error types that may be returned for the LoadFail exception include:

Table 5.11. LoadableDevice LoadFail requirements

Section	ID	Resp	Requirement
3.1.3.2.5.3.3	SR:424	DS	The error number shall indicate an ErrorNumberType value (e.g. `EACCES`, `EAGAIN`, `EBADF`, `EINVAL`, `EMFILE`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOSPC`, `ENOTDIR`).

- **`EACCES`** – Access to the file specified is not available or not granted. This may be due to underlying operating system restrictions based on the privileges associated with the process making the request.
- **`EAGAIN`** – The resource requested is temporarily unavailable.
- **`EBADF`** – The contents of the file are corrupted or do not match the type of load specified.
- **`EINVAL`** – The file specified is invalid.
- **`EMFILE`** – Too many files are currently open.

- **ENAMETOOLONG** – The file name specified exceeds the SCA limit or exceeds the limit of the underlying file system.
- **ENOENT** – No such file or directory.
- **ENOMEM** – Insufficient memory is available to complete the load. For files that map to a specific area of memory, this error code may also be used.
- **ENOSPC** – No space is left on the device specified.
- **ENOTDIR** – The file path provided in the load call is not a valid directory.

The above error types are not intended to be exhaustive. Thus, it is possible to use some other error number indicating the type of load error if none of the list applies.

unload

The onload operation supports the capability to unload software previously loaded using the load operation. The unload decrements a reference counter and, when the reference count goes to zero, removes the software from the device. There are some unique conditions and exceptions to implementing this behavior, however. For example, some loadable devices, such as FPGAs, may not have a low-level device operation that 'unloads' a bit image from the FPGA. The FPGA may be re-loaded with a new image or, if clearing the device is required (i.e. for security reasons), the unload operation may write a zero load to the FPGA.

Two exceptions may be raised by the operation. InvalidState is raised if the device is not in the proper state to perform the unload function. If the file specified by the filename parameter is invalid, then the InvalidFileName is raised.

```
void unload (
     in string fileName
     )
raises (InvalidState, InvalidFileName);
```

Table 5.12. LoadableDevice unload requirements

Section	ID	Resp	Requirement
3.1.3.2.5.5.2.3	SR:433	DS	The unload operation shall decrement the load count for the input filename parameter by one.
3.1.3.2.5.5.2.3	SR:434	DS	The unload operation shall unload the application software on the device based on the input fileName parameter, when the file's load count equals zero.
3.1.3.2.5.5.2.3	SR:435	DS	The unload operation shall raise the InvalidState exception when the Device's adminState is `LOCKED` or its operationalState is `DISABLED`.
3.1.3.2.5.5.2.3	SR:436	DS	The unload operation shall raise the InvalidFileName exception when the file designated by the input filename parameter cannot be found.

When the unload operation is received by the LoadableDevice, the load count for the file specified is decremented (SR:433) and, when the load count eaches zero, the file is unloaded from the LoadableDevice (SR:434) (see Table 5.12).

If the LoadableDevice is `LOCKED` or `DISABLED` then the InvalidState exception is raised (SR:435).

If the file name provided in the unload call cannot be found, then the InvalidFileName exception is raised (SR:436). This requirement raises some questions in that the wording specifies that the exception is raised if the file name reference by the parameter cannot be found. However, the file referenced by the fileName parameter has already been loaded and, therefore, already been found within the file system. Furthermore, the critical factor is not locating the file within a file system but identifying the image of the file previously loaded.

So, for the unload operation, the InvalidFileName exception applies to the situation when an unload operation is requested and the file name specified in the filename parameter cannot be found in the list of files previously loaded on to the LoadableDevice. So, although the exception is the same as the load operation, the root cause of the exception is different.

5.4 ExecutableDevice

The ExecutableDevice extends the LoadableDevice by defining a device that supports the execution of a program within an operating system (see Figure 5.8). It should also be noted, however, that an ExecutableDevice also supports loading of libraries, drivers, and kernel components. Thus, although the intent of the ExecutableDevice is to extend the LoadableDevice by supporting processes running under an operating system, loads that are not executable are still supported and frequently required.

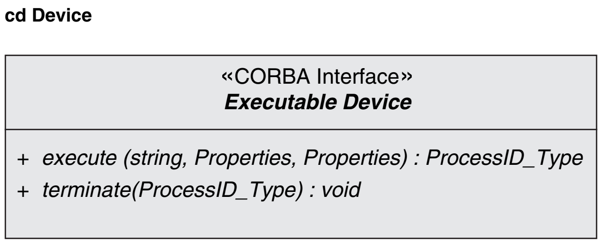

cd Device
«CORBA Interface»
Executable Device
+ execute (string, Properties, Properties) : ProcessID_Type
+ terminate(ProcessID_Type) : void

Figure 5.8. ExecutableDevice interface

5.4.1 Types and Constants

Two string constants are specified for the `STACK_SIZE` and `PRIORITY` arguments passed to the execute call.

ProcessID_Type

When an executable program is loaded and started on an ExecutableDevice, such as a GPP running an operating system, a unique process or task Id is created by the operating system. This Id is used by the operating system to manage the program within the computer. The process Id is also maintained by the core framework in order to be able to control the overall set of processes. The `ProcessID_Type` defines the data type used to store the Id of the process.

```
typedef unsigned long ProcessID_Type;
```

STACK_SIZE

Under certain operating systems, the size of the stack to be allocated or allowed for a given program can be specified. The string constant `STACK_SIZE` defines a common string that is used as a command line argument to the call that starts the program.

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

PRIORITY_ID

As with the `STACK_SIZE`, certain operating systems allow the execution priority of a program to be specified at the time the program is started. The `PRIORITY_ID` is a string constant that is used as a common command line argument name to pass in a numeric value that specifies the priority to be assigned to the program.

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

5.4.2 Exceptions

The exceptions requirements are shown in Table 5.13.

Table 5.13. ExecutableDevice exceptions requirements

Section	ID	Resp	Requirement
3.1.3.2.6.3.1	SR:438	DS	The error number shall indicate an ErrorNumberType value (e.g. `ESRCH`, `EPERM`, `EINVAL`).
3.1.3.2.6.3.6	SR:442	DS	The value for a stack size shall be an unsigned long.
3.1.3.2.6.3.7	SR:443	DS	The value for a priority shall be an unsigned long.
3.1.3.2.6.3.8	SR:444	DS	The error number shall indicate an ErrorNumberType value (e.g. `EACCES`, `EBADF`, `EINVAL`, `EIO`, `EMFILE`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`).

ExecuteFail

Due to the dynamic nature of general purpose computers and operating systems, there are a number of reasons why the operating system may fail to start an application program. When this occurs the SCA component launching the program must be notified. The ExecuteFail exception provides that notification. The errorNumber value contains an enumerated value indicating the type of error (SR:444).

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

InvalidProcess

The InvalidProcess exception is raised when a process Id provided is invalid, e.g. it is unknown, for the ExecutableDevice that implements the call. The errorNumber parameter in the exception provides an indication of the type of error encountered (SR:438). The msg parameter provides additional, textual information about the exception.

```
 exception InvalidProcess {
   ErrorNumberType errorNumber;
    string msg;
};
```

InvalidFunction

The InvalidFunction exception is raised when a function specified within a call cannot be found, i.e. it has not been loaded on the ExecutableDevice servicing the call. This exception typically occurs within executable devices that implement the ability to load an executable image and then start a task or thread at a specified function call.

For applications or components implemented as a standard executable on a GPP in an operating system such as Linux, a 'well known' initial function is defined, i.e. main in a C/C++ program, as part of the implementation or implicitly inserted as part of the development tools or environment, such as the Windows event loop.

```
exception InvalidFunction {
};
```

InvalidParameters

The InvalidParameters exception is raised when the execution parameters provided are invalid. Each execution parameter is provided as a pair of strings consisting of the name of the parameter and the value, similar to the argv** parameter on the top-level function, `main`, in a C program.

The exception provides the set of invalid parameters as a set of Properties where each entry in the invalidParms property set consists of the Property, i.e. parameter, name and value.

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

InvalidOptions

The InvalidOptions exception is raised when one or more of the options provided to the execute call are invalid. The invalidOpts parameter on the exception identifies the invalid options using a Property set where each Property, i.e. option, in the set consists of the name and value of the invalid option.

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

5.4.3 Operations

execute

The execute call provides the means for initiating the execution of a program or task on an ExecutableDevice that includes basic operating system functionality such as the ability to create a process or task within the operating system, uniquely identify the program or task, and terminate the task based on the process Id.

```
ProcessID_Type execute (
    in string name,
    in Properties options,
    in Properties parameters
    )
 raises (InvalidState, InvalidFunction, InvalidParameters,
   InvalidOptions, InvalidFileName, ExecuteFail);
```

This is not the same as the start operation defined on the Resource. Although the ExecutableDevice inherits the Resource interface and therefore implements the start and stop operations, the start/stop operations for the ExecutableDevice refer to the device itself and not the application components running on the device. This is illustrated in Figure 5.9.

The start and stop calls on the ExecutableDevice provide the control interface for starting and stopping the device. The execute operation initiates the execution of the Application process on the ExecutableDevice. That does not mean that the processing performed by the Application has started. The start and stop calls on the Application (or any type of Resource) start and stop the processing performed by the Application.

Thus, the execute operation is used to instruct the underlying operating system to begin running the process associated with the software, as specified in the file name argument. As with other similar functions for starting processes, the initial or starting function is specified followed by a set of execution parameters that are used to configure the software execution. The standard input options defined are `STACK_SIZE_ID` and `PRIORITY_ID`. When provided, the execute operation uses the values specified to set

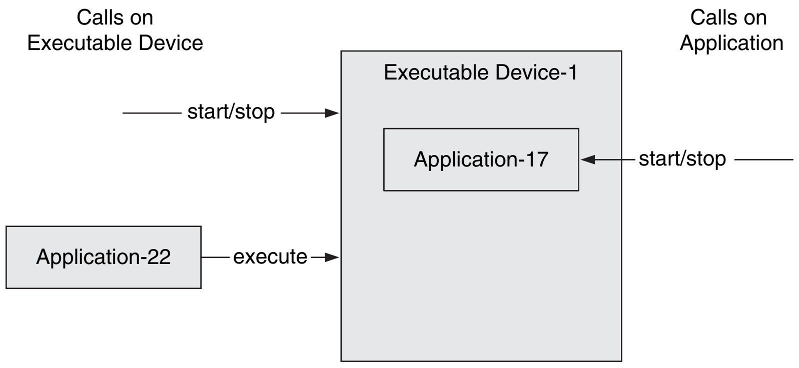


Figure 5.9. Calls on ExecutableDevice and Application

the priority of the process or thread created. The `STACK_SIZE_ID` provided must be an unsigned long (SR:442) and the `PRIORITY_ID` provided must also be an unsigned long (SR:443).

Several possible exceptions may be raised depending on the type of error encountered. The exceptions InvalidFunction, InvalidOptions, InvalidParameters, and InvalidFileName all indicate a problem encountered with the input parameters. The InvalidState exception indicates that the ExecutableDevice was not in a valid state for performing the execute operation. The ExecuteFail exception indicates that a problem was encountered while attempting to start the process or thread, which resulted in a failure to do so.

The execute operation checks that the ExecutableDevice is in a valid state to perform the operation (see Table 5.14). If it is not then the InvalidState exception is raised (SR:450) and the operation terminates. The execute operation provides the means to execute the file name specified by the name argument (SR:445). The function attempts to open the file name specified in order to initiate execution on the ExecutableDevice. If the file name provided cannot be found, then the InvalidFileName exception is raised (SR:452). The execute function converts the parameters provided into POSIX form using the initial argument as the function name at which execution should begin (SR:446). The execute operation converts the parameters into a set of input arguments to the process following the standard convention of name and value pairs (SR:447). When the parameters provided are not strings, then the InvalidParameters exception is raised (SR:453) If the `STACK_SIZE_ID` and `PRIORITY_ID` options are provided, then the execute operation uses the values provided to set the priority and stack size of the process (SR:448). If the options provided are invalid then the InvalidOptions exception is raised (SR:454). The execute operation then initiates the operating system call to begin the execution of the file specified, passing the parameters to the execute operation as input arguments to the function specified, and returning the unique process or thread Id provided by the operating system at the completion of its operation (SR:449). If an error is encountered by the operating system that prevents it from properly

Table 5.14. ExecutableDevice execute requirements

Section	ID	Resp	Requirement
3.1.3.2.6.5.1.3	SR:445	DS	The execute operation shall execute the function or file identified by the input name parameter using the input parameters and options parameters.
3.1.3.2.6.5.1.3	SR:446	DS	The execute operation shall convert the input parameters (iD/value string pairs) parameter to the standard argv of the POSIX exec family of functions, where argv(0) is the function name.
3.1.3.2.6.5.1.3	SR:447	DS	The execute operation shall map the input parameters to argv starting at index 1 as follows: argv (1) maps to input parameters (0) iD and argv (2) maps to input parameters (0) value and so forth.
3.1.3.2.6.5.1.3	SR:448	DS	The execute operation shall use these options, when specified, to set the operating system's process/thread stack size and priority, for the executable image of the given input name parameter.
3.1.3.2.6.5.1.4	SR:449	DS	The execute operation shall return a unique processID for the process that it created or a processID of minus 1 (−1) when a process is not created.
3.1.3.2.6.5.1.5	SR:450	DS	The execute operation shall raise the InvalidState exception when the Device's adminState is not `UNLOCKED` or operationalState is `DISABLED`.
3.1.3.2.6.5.1.5	SR:451	DS	The execute operation shall raise the InvalidFunction exception when the function indicated by the input name parameter does not exist for the Device.
3.1.3.2.6.5.1.5	SR:452	DS	The execute operation shall raise the InvalidFileName exception when the file name indicated by the input name parameter does not exist for the Device.
3.1.3.2.6.5.1.5	SR:453	DS	The execute operation shall raise the InvalidParameters exception hen the input parameters parameter item ID or value are not string types.
3.1.3.2.6.5.1.5	SR:454	DS	The execute operation shall raise the InvalidOptions exception when the input options parameter does not comply with sections 3.1.3.2.6.3.5 `STACK_SIZE_ID` and 3.1.3.2.6.3.6 `PRIORITY_ID`.
3.1.3.2.6.5.1.5	SR:455	DS	The execute operation shall raise the ExecuteFail exception when the operating system 'execute' function for the device is not successful.

initiating the execution of the software, then the execute operation raises the ExecuteFail exception (SR:455).[3]

terminate

As the name implies, the terminate operation provides the capability to terminate a process or thread previously initiated with the execute operation. A single argument, the processId of the process to be terminated, is provided. There is no return value specified.

If the ExecutableDevice adminState is `LOCKED` or the operationalState `DISABLED`, then the InvalidState exception is raised. If the processId is not found then the InvalidProcess is raised.

```
void terminate (
   in ProcessID_Type processId
   )
raises (InvalidProcess, InvalidState);
```

The terminate operation terminates the execution of the process or thread specified by the processId argument (SR:456) (see Table 5.15). The terminate operation checks that ExecutableDevice adminState is not `LOCKED` and the operationalState is not `DISABLED`. If either is true then the InvalidState exception is raised (SR:457). If the processId specified does not exist, then the InvalidProcess exception is raised (SR:458).

Table 5.15. ExcecutableDevice terminate requirements

Section	ID	Resp	Requirement
3.1.3.2.6.5.2.3	SR:456	DS	The terminate operation shall terminate the execution of the process/thread designated by the processId input parameter on the Device.
3.1.3.2.6.5.2.5	SR:457	DS	The terminate operation shall raise the InvalidState exception when the Device's adminState is `LOCKED`or operationalState is `DISABLED`.
3.1.3.2.6.5.2.5	SR:458	DS	The terminate operation shall raise the InvalidProcess exception when the processId does not exist for the Device.

[3] *The execute operation raises an interesting problem related to the file. In order for the operating system call to load and execute the file specified, the operating system must be able to open the file using the native file I/O routines within the operating system and obtain a native File Descriptor (FD) in order to access the file. Since the SCA FileSystem is an abstraction layered on top of a native file system in the operating system, there are two issues that must be addressed. First, as noted, the native FD must be provided to the operating system call. If the SCA FileSystem containing the file is on the same ExecutableDevice as where the process is to be run, then the native FD can be provided as part of the internal state when opening the SCA File. However, if the file is located on another node, then the file will need to be copied so there is a copy of the file that is located within the native file system on the ExecutableDevice where the process is to be executed. The implementation approaches to handling this problem vary but the implementation of the execute operation must address this issue.*

5.5 AggregateDevice

As briefly described earlier in this chapter, the AggregateDevice provides a mechanism for managing a set of Devices as a logical unit. The AggregateDevice is a simple entity that provides an interface for adding and removing a Device (see Figure 5.10). The interface is somewhat limiting, however, as it does not provide a method for iterating over the list of Devices or looking up an individual Device within the AggregateDevice.

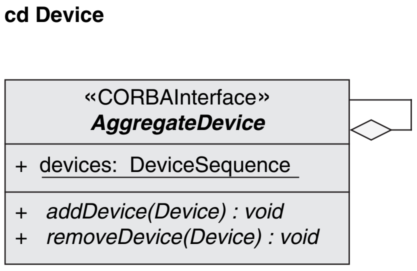

cd Device
«CORBAInterface»
AggregateDevice
+ devices: DeviceSequence
+ addDevice(Device) : void
+ removeDevice(Device) : void

Figure 5.10. AggregateDevice interface

5.5.1 Types and Attributes

devices

The Aggregate Device attribute requirements are shown in Table 5.16.

The devices attribute contains a list of Devices that have been added to the AggregateDevice (SR:459).

```
readonly attribute DeviceSequence devices;
```

Table 5.16. AggregateDevice attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.7.4.1	SR:459	DS	The readonly devices attribute shall contain a list of Devices that have been added to this Device or a sequence length of zero if the Device has no aggregation relationships with other Devices.

5.5.2 Operations

addDevice

The addDevice operation inserts a Device reference into the sequence of devices stored on the devices attribute.

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

Table 5.17. AggregateDevice addDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.7.5.1.3	SR:460	DS	The addDevice operation shall add the input associatedDevice parameter to the AggregateDevice's devices attribute when the associatedDevice does not exist in the devices attribute.
3.1.3.2.7.5.1.3	SR:461	DS	The addDevice operation shall write a Failure_Alarm log record, upon unsuccessful adding of an associatedDevice to the AggregateDevice's devices attribute.
3.1.3.2.7.5.1.5	SR:462	DS	The addDevice operation shall raise the CF InvalidObjectReference when the input associatedDevice is a nil CORBA object reference.

The addDevice call checks that the Device identified in the associatedDevice parameter is not already in the devices attribute (see Table 5.17). If it is not already a member of the sequence, it is inserted into the list of Devices in the devices attribute (SR:460). If the addDevice operation fails for any reason, it writes a `FAILURE_ALARM` record to the log (SR:461). If the addDevice call receives a nil CORBA reference for the associatedDevice parameter, it raises an InvalidObjectReference exception (SR:462).

removeDevice

The removeDevice removes the specified reference from the sequence of devices maintained on the devices attribute.

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

The removeDevice operation will remove the Device reference specified in the associatedDevice parameter from the devices attribute (SR:463) (see Table 5.18). If the Device reference provided in the operation is not a member of the list of Devices stored on the devices attribute, the call terminates without error. If the operation fails for any reason, a `FAILURE_ALARM` record is written to the log (SR:464). If the associatedDevice parameter provided contains a nil object reference, an InvalidObjectReference exception is raised (SR:465).

5.6 DeviceManager

The Device Manager is an abstract entity intended to provide a top-level set of interfaces and services for a logical set of devices, services and a file system. Examples of a logical set of hardware resources include a single-board computer (SBC), a card within a rack that

Table 5.18. AggregateDevice removeDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.7.5.2.3	SR:463	DS	The removeDevice operation shall remove the input associatedDevice parameter from the AggregateDevice's devices attribute.
3.1.3.2.7.5.2.3	SR:464	DS	The removeDevice operation shall write a Failure_Alarm log record, upon unsuccessful removal of the associatedDevice from the AggregateDevice's devices attribute.
3.1.3.2.7.5.2.5	SR:465	DS	The removeDevice operation shall raise the CF InvalidObjectReference when the input associatedDevice is a nil CORBA object reference or does not exist in the AggregateDevice's devices attribute.

provides processing resources in the form of a FPGA, a DSP, or other type of radio system processing component.

Although the previous example described a set of resources that were physically co-located, there is no requirement within the SCA that mandates such an organization. In fact, it is entirely possible to have a single Device Manager for all Devices within the system, although it may be argued that such a coarse level of abstraction reduces the level of insight and control of the radio system components.

Services include the implementation of an SCA FileSystem, discussed in Chapter 3, and the Log Service. The set of SCA Devices associated with the Device Manager register with the Device Manager as well. However, the Device Manager does not provide any management or control functions for the set of Devices registered with it.

As Figure 5.11 shows, the Device Manager implements the PropertySet and PortSupplier interfaces in addition to those specified for the DeviceManager.

5.6.1 Types

ServiceType

The ServiceType struct provides a mapping between a service on the system and an associated service name. The serviceName field may be any string and currently does not have any fixed naming convention.

```
struct ServiceType {
   Object serviceObject;
   string serviceName;
};
```

ServiceSequence

The ServiceSequence type defines a data structure used to maintain a set of services.

```
typedef sequence <ServiceType> ServiceSequence;
```

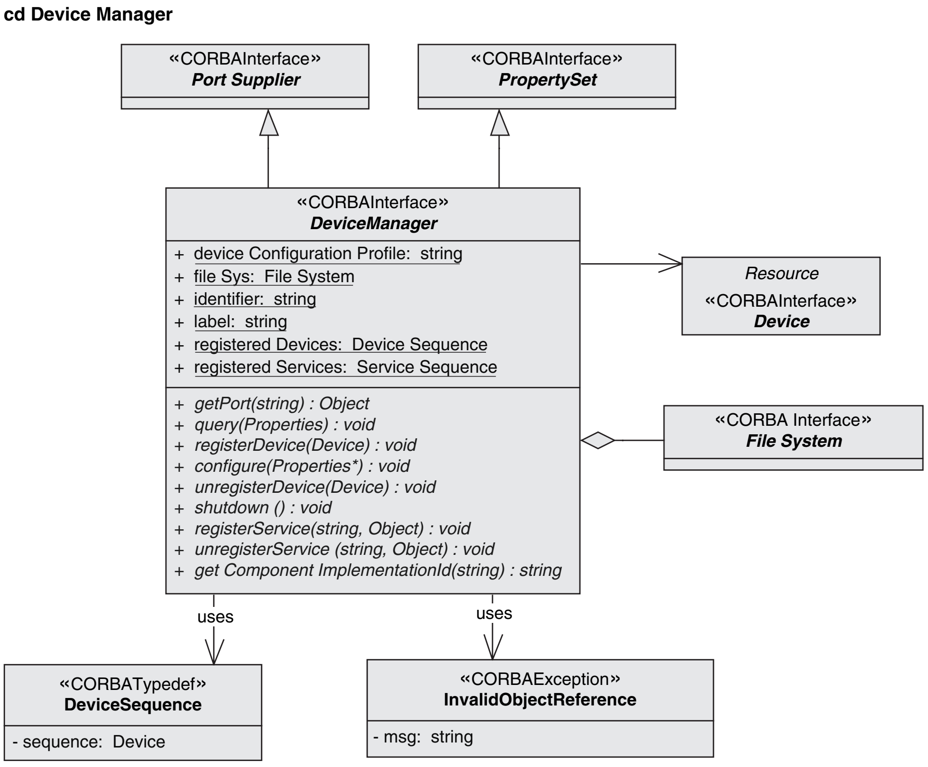

cd Device Manager
«CORBAInterface» Port Supplier
«CORBAInterface» PropertySet
«CORBAInterface» DeviceManager
+ device Configuration Profile: string
+ file Sys: File System
+ identifier: string
+ label: string
+ registered Devices: Device Sequence
+ registered Services: Service Sequence
+ getPort(string) : Object
+ query(Properties) : void
+ registerDevice(Device) : void
+ configure(Properties*) : void
+ unregisterDevice(Device) : void
+ shutdown () : void
+ registerService(string, Object) : void
+ unregisterService (string, Object) : void
+ get Component ImplementationId(string) : string
Resource
«CORBAInterface» Device
«CORBA Interface» File System
uses
uses
«CORBATypedef» DeviceSequence
- sequence: Device
«CORBAException» InvalidObjectReference
- msg: string

Figure 5.11. Device Manager interface

5.6.2 Attributes

Primarily the attributes provide descriptive information of the Device Manager and contain instances of Services, Devices, and File Systems. The IDLs for the DeviceManager attributes are shown below.

deviceConfigurationProfile

The deviceConfigurationProfile attribute contains a reference to the DeviceManager's profile. The early versions of SCA, e.g. 2.2, specified that the deviceConfigurationProfile attribute should contain either a file reference to the DeviceManager's Device Configuration Descriptor (DCD) profile or the XML for the DeviceManager's DCD profile. Files referenced within the profile are obtained from a FileSystem. Realistically speaking, as with other profile attributes, the only value that makes sense operationally is the file reference.

```
readonly attribute string deviceConfigurationProfile;
```

As shown above, this attribute is readonly. It is set as part of the startup and initialization logic of the DeviceManager.

fileSys

If the DeviceManager has created an instant of an SCA FileSystem, then the object reference will be stored in this attribute. If no FileSystem has been instantiated then the fileSys attribute contains a nil CORBA object reference.

```
readonly attribute FileSystem fileSys;
```

As shown above, this attribute is readonly. It is set as part of the startup and initialization logic performed by the DeviceManager.

identifier

The identifier attribute contains a unique identifier for the DeviceManager instance. The value of the identifier is specified by the Id attribute for the deviceconfiguration element in the DeviceManager's DCD XML file.

```
readonly attribute string identifier;
```

This attribute is readonly. The value is set during the initialization of the DeviceManager.

label

The label attribute contains a descriptive name for the DeviceManager. This is typically some name that conveys a particular interpretation or meaning to the reader/user.

```
readonly attribute string label;
```

This attribute is readonly and set as part of the initialization of the DeviceManager.

registeredDevices

As Devices register with the DeviceManager, the registeredDevices attribute is used to store and manage the list of registered Devices. If no Devices register with the DeviceManager, then the Device sequence is empty with a length of zero.

```
readonly attribute Devicesequence registereddevices;
```

This attribute is managed as part of the registerDevice operation and is a readonly attribute to external operations.

registeredServices

The DeviceManager attribute requirements are shown in Table 5.19.

As the Device Manager is instantiated and initialized, it processes the DCD XML file to obtain information about the configuration and initialization of the DeviceManager being instantiated. Upon instantiation, a globally unique identifier is stored in the identifier attribute (SR:466) (see Table 5.19). The identifier stored on the attribute is specified within the DCD file (SR:467). In addition to the identifier, a human-readable label, also defined in the DCS file is stored in the label attribute (SR:468). The deviceConfigurationProfile attribute (SR:470) contains the string name of the DCD file for the Device Manager, although the

Table 5.19. Device Manager attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.8.4.1	SR:466	DS	The readonly identifier attribute shall contain the instance-unique identifier for a DeviceManager.
3.1.3.2.8.4.1	SR:467	DS	The identifier shall be identical to the deviceconfiguration element Id attribute of the DeviceManager's Device Configuration Descriptor (DCD) file.
3.1.3.2.8.4.2	SR:468	DS	The readonly label attribute shall contain the DeviceManager's label. The label attribute is the meaningful name given to a DeviceManager.
3.1.3.2.8.4.3	SR:469	DS	The readonly fileSys attribute shall contain the FileSystem associated with this DeviceManager or a nil CORBA object reference if no FileSystem is associated with this DeviceManager.
3.1.3.2.8.4.4	SR:470	DS	The readonly deviceConfigurationProfile attribute shall contain either a profile element with a file reference to the DeviceManager's DCD profile or the XML for the DeviceManager's DCD profile.
3.1.3.2.8.4.5	SR:471	DS	The readonly registeredDevices attribute shall contain a list of Devices that have registered with this DeviceManager or a sequence length of zero if no Devices have registered with the DeviceManager.
3.1.3.2.8.4.6	SR:472	DS	The readonly registeredServices attribute shall contain a list of Services that have registered with this DeviceManager or a sequence length of zero if no Services have registered with the DeviceManager.

requirement states that the attribute may contain the DCD XML instead of the file name. However, in practical application, the XML file name is stored rather than the XML itself.

The remaining attributes provide storage for instances of services, devices, and the file system. It is not mandatory that the Device Manager include any services or a file system. If a File System is provided by the Device Manager, the fileSys attribute contains the instance of the FileSystem instantiated by the Device Manager (SR:469). If no FileSystem is instantiated for the DeviceManager then a nil CORBA reference is stored in the data member.

Any services provided by the Device Manager are instantiated and stored on the registeredServices attribute (SR:472). This attribute is a list of zero or more services provided through the interface as a CORBA sequence. For example, if the Device Manager instantiates a Log Service, it is added to this list. As the attribute name implies, the service is registered.

Each service entry is a **`ServiceType`** struct consisting of the CORBA object reference to the service and the name of the service.

Similarly, Devices are instantiated, registered with the Device Manager and inserted into the list of registeredDevices (SR:471). Duplicate references are not permitted in the registeredDevices attribute.

5.6.3 Operations

Table 5.20 shows the requirements for Device Manager behavior.

registerDevice

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

Initially, the Device Manager starts as an executable process and reads the XML file to identify the services that should be instantiated and the devices to be created (SR:474). If the DeviceManager is to provide a FileSystem, it instantiates the file system object (SR:475) and mounts the underlying file system provided by the operating system or other host software to the SCA FileSystem Since the FileSystem maps to a single underlying file system on the host platform of the DeviceManager, if there are multiple file system mount points to be mapped into the SCA environment, the DeviceManager may instantiate multiple FileSystems within a FileManager and the fileSys attribute will then contain the FileManager (SR:476).

The logical Devices specified within the DCD file are instantiated. Each Device within the DCD identifies the executable file that implements the Device, through a reference to the Software Package Descriptor (SPD), to be loaded and started. As the Device is started, i.e. the DeviceManager starts the executable, startup parameters, also defined within the DCD for each Device, are passed to the executable file in the form of Name/Value pairs (SR:478). The specific parameters include:

- `DEVICE_MGR_IOR` – This parameter provides the Inter-ORB Reference converted to string format.
- `PROFILE_NAME` – This parameter is the string that specifies the full file system path name.
- `DEVICE_ID` – This is the globally unique identifier that provides a unique Id for the Device. It is stored in the identifier attribute of the Device.
- `DEVICE_LABEL` – This is a human readable string providing a recognizable name for the Device. It is stored on the label attribute of the Device.
- `COMPOSITE_DEVICE_IOR` – If the Device being instantiated is part of an SCA AggregateDevice, this parameter provides the stringified IOR for the AggregateDevice that contains the Device.

Any additional execution parameters (execparam) that are specified with values in the properties definition for the Device (SR:479) are passed as string pairs (SR:480).

When a particular Device's SPD includes values for the stacksize or priority elements, those values are also passed as string pairs as part of the execution parameters (SR:481).

Table 5.20. Device Manager behavior requirements

Section	ID	Resp	Requirement
3.1.3.2.8.5	SR:473	DS	The DeviceManager upon start up shall register itself with a DomainManager. This requirement allows the system to be developed where at a minimum only the DomainManager's component reference needs to be known.
3.1.3.2.8.5	SR:474	DS	A DeviceManager shall use the DeviceManager's deviceConfigurationProfile attribute for determining: 1. services to be deployed for this DeviceManager (for example, log(s)); 2. devices to be created for this DeviceManager (when the DCD deployondevice element is not specified, then the DCD componentinstantiation element is deployed on the same hardware device as the DeviceManager); 3. devices to be deployed on (executing on) another Device; 4. devices to be aggregated to another Device; 5. mount point names for FileSystems; 6. the DCD's Id attribute for the DeviceManager's identifier attribute value; and 7. the DCD's name attribute for the DeviceManager's label attribute value.
3.1.3.2.8.5	SR:475	DS	The DeviceManager shall create FileSystem components implementing the FileSystem interface for each OS file system.
3.1.3.2.8.5	SR:476	DS	If multiple FileSystems are to be created, the DeviceManager shall mount created FileSystems to a FileManager component (widened to a FileSystem through the fileSys attribute).
3.1.3.2.8.5	SR:478	DS	The DeviceManager shall supply execute operation parameters (IDs and format values) for a Device consisting of: • DeviceManager IOR – The Id is `'DEVICE_MGR_IOR'` and the value is a string that is the DeviceManager stringified IOR.

Table 5.20. (Continued)

Section	ID	Resp	Requirement
			• Profile Name – The Id is `'PROFILE_NAME'` and the value is a CORBA string that is the full mounted file system file path name. • Device Identifier – The Id is `'DEVICE_ID'` and the value is a string that corresponds to the DCD componentinstantiation Id attribute. • Device Label – The Id is `'DEVICE_LABEL'` and the value is a string that corresponds to the DCD componentinstantiation usage element. This parameter is only used when the DCD componentinstantiation usage element is specified. • Composite Device IOR – The Id is `'Composite_DEVICE_IOR'` and the value is a string that is an AggregateDevice stringified IOR. This parameter is only used when the DCD componentinstantiation element is a composite part of another componentinstantiation element.
3.1.3.2.8.5	SR:479	DS	The DeviceManager shall pass the componentinstantiation element 'execparam' properties that have values as parameters.
3.1.3.2.8.5	SR:480	DS	The DeviceManager shall pass 'execparam' parameters' Ids and values as string values.
3.1.3.2.8.5	SR:481	DS	The DeviceManager shall use the componentinstantiation element's SPD implementation code's stacksize and priority elements, when specified, for the execute options parameters.
3.1.3.2.8.5	SR:482	DS	The DeviceManager shall initialize and configure logical Devices that are started by the DeviceManager after they have registered with the DeviceManager.
3.1.3.2.8.5	SR:483	DS	The DeviceManager shall configure a DCD's componentinstantiation element provided the componentinstantiation element has 'configure' readwrite or writeonly properties with values. If a Service is deployed by the DeviceManager, the DeviceManager shall supply execute operation parameters (IDs and format values) consisting of:

3.1.3.2.8.5	SR:484	DS	• DeviceManager IOR – The Id is `'DEVICE_MGR_IOR'` and the value is a string that is the DeviceManager's stringified IOR. • Service Name – The Id is `'SERVICE_NAME'` and the value is a string that corresponds to the DCD componentinstantiation usagename element

Once a Device has been started, it then comes back to the DeviceManager that started the Device using the DeviceManager IOR parameter and registers with the DeviceManager (see reference requirement SR:485 in Table 5.21 on page xxx). The DeviceManager then performs any initialization required and performs any additional configuration of properties for the Device (SR:482).

Any configure properties identified in the DCD for a componentinstantiation element that are as readwrite or writeonly are set by the DeviceManager (SR:483).

As with the set of Devices identified in the DCD, the set of Services specified in the DCD are also instantiated. As each Service is instantiated by the DeviceManager, the DeviceManager passes startup information in the form of execution parameters (SR:484). The two parameters provided are:

Table 5.21. Registration requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.1.3	SR:485	DS	The registerDevice operation shall add the input registeringDevice to the DeviceManager's registeredDevices attribute when the input registeringDevice does not already exist in the registeredDevices attribute.
3.1.3.2.8.6.1.3	SR:486	DS	The registerDevice operation shall register the registeringDevice with the DomainManager when the DeviceManager has already registered to the DomainManager and the registeringDevice has been successfully added to the DeviceManager's registeredDevices attribute.
3.1.3.2.8.6.1.3	SR:487	DS	The registerDevice operation shall write a `FAILURE_ALARM` log record to a DomainManager's log, upon unsuccessful registration of a Device to the DeviceManager's registeredDevices.
3.1.3.2.8.6.1.5	SR:488	DS	The registerDevice operation shall raise the CF InvalidObjectReference when the input registeringDevice is a nil CORBA object reference.

- `DEVICE_MGR_IOR` – This is the stringified IOR that provides a reference back to the DeviceManager that started the Service.
- `SERVICE_NAME` – The name of the Service defined in the DCD is provided as the second argument.

As with the instantiation of the Devices, once the Service has been successfully started and initialized, it uses the `DEVICE_MGR_IOR` to register with the DeviceManager.

The DeviceManager then registers with the Domain Manager (SR:473). Whether the DeviceManager registers with the DomainManager at the completion of its initialization process or some other point in time is not an explicitly specified requirement in the SCA specification. However, the UML sequence diagram illustrates the registration being performed at the completion of the DeviceManager startup and initialization process. The rationale for registering upon completion of the initialization process are that the system and user processes is assured that the DeviceManager entries maintained by the DomainManager reference instances of DomainManagers that are initialized and ready to accept requests.

The balance of this section breaks the remaining requirements into specific behavioral categories. Figure 54 illustrates the requirements associated with the registerDevice method.

As described above, when a Device is started by the DeviceManager, it is provided by the stringified IOR of the DeviceManager. The Device then uses the IOR provided to notify the DeviceManager that it has started and initialized successfully. Table 5.21 lists the requirements associated with this method.

When the DeviceManager receives a registerDevice call from a Device, the Device reference is provided. The DeviceManager adds the Device reference to the registeredDevices attribute (SR:485). As noted previously, duplicate Device references are not permitted in the registeredDevices attribute. So, if the DeviceManager receives a registerDevice call from a Device that it has already placed in the registeredDevices attribute, it does not add it again.

Also, as described earlier, as part of the DeviceManager startup every Device registered with the DeviceManager is also registered with the DomainManager. So, in the event that a Device registration occurs after the DeviceManager has registered with the Domain Manager, the DeviceManager must let the DomainManager know that the set of Devices has changed. So, the DeviceManager notifies the DomainManager by issuing the registerDevice call to the DomainManager on behalf of the Device (SR:486).

If the Device reference received in the registerDevice call is nil, the DeviceManager raises the InvalidObjectReference (SR:488). This exception is raised in multiple calls when a nil object reference is provided. This is somewhat limiting in that the exception name implies that an incorrect object reference has been provided. While nil is certainly an invalid object reference, it does not capture the case when an object reference for something other than a Device is provided. This can be enforced by narrowing the reference to the Device to ensure that the object reference provided is, in fact, a proper reference.

If the registerDevice call is unsuccessful, the DeviceManager publishes a `FAILURE_ALARM` log record to the DomainManager's log (SR:487).

registerService

This operation provides registers a Service with a DeviceManager (see Table 5.22). Part of the functionality of the operation is to update the list of registeredServices as a new service registers with the DeviceManager. In the event that a service registration occurs after the DeviceManager has already registered with the DomainManager, the operation will also register the service with the DomainManager.

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

Similar to the registration of a Device, each Service instantiated is registered with the DeviceManager.

When the DeviceManager receives a registerService call from a Service, the Service reference is provided. The DeviceManager adds the Service reference to the

Table 5.22. registerService requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.3.3	SR:493	DS	The registerService operation shall add the input registeringService to the DeviceManager's registeredServices attribute when the input registeringService does not already exist in the registeredServices attribute.
3.1.3.2.8.6.3.3	SR:494	DS	The registerService operation shall register the registeringService with the DomainManager when the DeviceManager has already registered to the DomainManager and the registeringService has been successfully added to the DeviceManager's registeredServices.
3.1.3.2.8.6.3.3	SR:495	DS	The registerService operation shall write a `FAILURE_ALARM` log record, upon unsuccessful registration of a Service to the DeviceManager's registeredServices.
3.1.3.2.8.6.3.3	SR:496	DS	The registerService operation shall raise the CF InvalidObjectReference exception when the input registeringService is a nil CORBA object reference.
3.1.3.2.8.6.3.5	SR:497	DS	The registerService operation shall add the input registeringService to the DeviceManager's registeredServices attribute when the input registeringService does not already exist in the registeredServices attribute.

registeredServices attribute (SR:497). As noted previously, duplicate Service references are not permitted in the registeredServices attribute. So, if the DeviceManager receives a registerService call from a Service that it has already placed in the registeredServices attribute, it does not add it again.

Also, as described earlier, as part of the DeviceManager startup every Service registered with the DeviceManager is also registered with the DomainManager. So, in the event that a Service registration occurs after the DeviceManager has registered with the Domain Manager, the DeviceManager must let the DomainManager know that the set of Services has changed. So, the DeviceManager notifies the DomainManager by issuing the registerService call to the DomainManager on behalf of the Service (SR:494).

If the Service reference received in the registerService call is nil, the DeviceManager raises the InvalidObjectReference (SR:496). This exception is raised in multiple calls when a nil object reference is provided. This is somewhat limiting in that the exception name implies that an incorrect object reference has been provided. While nil is certainly an invalid object reference, it does not capture the case when an object reference for something other than a Service is provided. This can be enforced by narrowing the reference to the Service to ensure that the object reference provided is, in fact, a proper reference.

If the registerService call is unsuccessful, the DeviceManager publishes a `FAILURE_ALARM` log record to the DomainManager's Log (SR:495).

unregisterDevice

As the inverse of the registerDevice operation, this operation unregisters a Device from a DeviceManager (see Table 5.23). Part of the responsibility of the operation is to remove unregistered Devices from the registeredDevices attribute.

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

When a Device is shutdown or removed from service for some reason, it issues an unregisterDevice call to the DeviceManager that instantiated it. Upon receiving an unregisterDevice call, the DeviceManager removes the Device from the list of registeredDevices maintained by the DeviceManager (SR:489). Then, on behalf of the Device, the DeviceManager unregisters the Device with the DomainManager by issuing the unregisterDevice call on the DomainManager (SR:490).

If for any reason an error is encountered during the unregisterDevice call on the DeviceManager, the DeviceManager writes a `FAILURE_ALARM` log record to the Log Service (SR:491). In the event that the Device reference provided in the call is nil, the InvalidObjectReference exception is raised (SR:492).

unregisterService

The unregisterService removes the service specified from the registeredServices attribute (see Table 5.24). If the DeviceManager is not in the process of `SHUTTING_DOWN`, the operation will also perform an unregisterService call on the DomainManager.

Table 5.23. unregisterDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.2.3	SR:489	DS	The unregisterDevice operation shall remove the input registeredDevice from the DeviceManager's registeredDevices attribute.
3.1.3.2.8.6.2.3	SR:490	DS	The unregisterDevice operation shall unregister the input registeredDevice from the DomainManager when the input registeredDevice is registered with the DeviceManager and the DeviceManager is not shutting down.
3.1.3.2.8.6.2.3	SR:491	DS	The unregisterDevice operation shall write a `FAILURE_ALARM` log record, when it cannot successfully remove a registeredDevice from the DeviceManager's registeredDevices.
3.1.3.2.8.6.2.5	SR:492	DS	The unregisterDevice operation shall raise the CF InvalidObjectReference when the input registeredDevice is a nil CORBA object reference or does not exist in the DeviceManager's registeredDevices attribute.

Table 5.24. unregisterService requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.4.3	SR:497	DS	The unregisterService operation shall remove the input registeredService from the DeviceManager's registeredServices attribute.
3.1.3.2.8.6.4.3	SR:498	DS	The unregisterService operation shall unregister the input registeredService from the DomainManager when the input registeredService is registered with the DeviceManager and the DeviceManager is not in the shutting down state.
3.1.3.2.8.6.4.3	SR:499	DS	The unregisterService operation shall write a `FAILURE_ALARM` log record, when it cannot successfully remove a registeredService from the DeviceManager's registeredServices.
3.1.3.2.8.6.4.5	SR:500	DS	The unregisterService operation shall raise the CF InvalidObjectReference when the input registeredService is a nil CORBA object reference or does not exist in the DeviceManager's registeredServices attribute.

```
void unregisterService (
     in Object registeredService,
     in string name
      )
raises (InvalidObjectReference);
```

Similar to the unregisterDevice call, the unregisterService call performs the same function for services provided by the DeviceManager.

When a Service is shutdown or removed from service, it issues an unregisterService call to the DeviceManager that instantiated it. Upon receiving an unregisterService call, the DeviceManager removes the Service from the list of registeredServices maintained by the DeviceManager (SR:497). If the Service had been registered with the DomainManager (it is not mandatory that all services register with the DomainManager), the DeviceManager unregisters the Service with the DomainManager by issuing the unregisterService call on the DomainManager (SR:498).

If for any reason an error is encountered during the unregisterService call on the DeviceManager, the DeviceManager writes a `FAILURE_ALARM` log record to the Log Service (SR:499). In the event that the object reference provided for Service in the call is nil, the InvalidObjectReference exception is raised (SR:500).

getComponentImplementationId

When a software component is instantiated, the specific implementation is chosen at run-time by the ApplicationFactory that instantiates the waveform Application based on the available resources. The Id of the specific implementation chosen for instantiation is stored on the component instance. The getComponentImplementationId provides the means to retrieve the Id of the implementation component that was selected and loaded (see Table 5.25).

```
string getComponentImplementationId (
   in string componentInstantiationId
);
```

The SCA supports multiple implementations of a given component, e.g. one implementation might be for a Pentium processor running Linux and another might be for the same processor running Windows. Each of the implementations has a unique implementation Id as defined in the Software Package Descriptor (SPD) XML file. At load time, based on the resources available, the framework will select a particular implementation that matches the available set of resources. Because it is not necessarily known which implementation will be selected, there needs to be a way to identify which implementation was instantiated.

The getComponentId operation will search the set of instantiated Devices for one with a implementation Id that matches the input componentId on the call. If a match is found, the Id string is returned (SR:504). If not, an empty string is returned (SR:505).

This function call would be more useful if it returned the Device's object reference based on the component Id and returned nil if no match was found. This would support the capability to search a set of instantiated Devices and components on a DeviceManager for a specific implementation and return a reference that could be used to interact with the Device directly.

Table 5.25. getComponentImplementationId requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.6.4	SR:504	DS	The getComponentImplementationId operation shall return the SPD implementation element's Id attribute that matches the SPD implementation element used to create the component identified by the input componentInstantiationId parameter.
3.1.3.2.8.6.6.4	SR:505	DS	The getComponentImplementationId operation shall return an empty string when the input componentInstantiationId parameter does not match the Id attribute of any SPD implementation element used to create the component.

shutdown

The shutdown operation initiates the termination process for a DeviceManager (see Table 5.26). This includes unregistering the DeviceManager with the DomainManager, releasing all known, i.e. registered, Devices, terminating any services provided by the DeviceManager, and unmounting any FileSystem provided by the DeviceManager.

```
void shutdown ();
```

A DeviceManager may be instructed to shutdown via the shutdown call. This call enables an orderly termination of the DeviceManager and all the Devices associated with it.

When the DeviceManager is instructed to shutdown via the shutdown call, it first unregisters itself with the DomainManager (SR:501). Then, having removed itself from the set of registered DeviceManagers on the DomainManager, it iterates over the set of registered

Table 5.26. shutdown DeviceManager requirements

Section	ID	Resp	Requirement
3.1.3.2.8.6.5.3	SR:501	DS	The shutdown operation shall unregister the DeviceManager from the DomainManager.
3.1.3.2.8.6.5.3	SR:502	DS	The shutdown operation shall perform releaseObject on all of the DeviceManager's registered Devices (DeviceManager's registeredDevices attribute).
3.1.3.2.8.6.5.3	SR:503	DS	The shutdown operation shall cause the DeviceManager to be unavailable (i.e. released from the CORBA environment and its process terminated on the OS), when all of the DeviceManager's registered Devices are unregistered from the DeviceManager.

devices and issues the releaseObject call (SR:502) (see releaseObject in Section 4.3.2. When all of the Devices registered with the DeviceManager have been released, i.e. terminated and removed from the list of registered devices, the DeviceManager terminates (SR:503).

6

Domain Management

6.1 DomainManager

The SCA specification organizes the DomainManager interfaces into three categories:

1. **Human Computer Interface (HCI):** These are identified as interfaces to configure the radio, provide access to services, devices, and applications, and initiate maintenance functions.
2. **Registration:** These are identified as operations to register or unregister DeviceManagers, Devices, Applications, and services with the DomainManager.
3. **Core Framework Administration:** These are defined as those operations that provide access to the interfaces of registered DeviceManagers, FileManagers, and Loggers in the system.

Although the categorization of some of the interfaces is as HCI, it should be noted that the interfaces provided may be called by another application, such as a network management function, and are not exclusively the purview of an HCI. In the case of Core Framework Administration operations, access to interfaces supported by DeviceManagers, for example, is gained through the deviceManagers attribute which contains the set of registered DeviceManagers. Therefore, this chapter follows the same pattern as previous chapters, providing information on data types, exceptions, attributes and operations. The DomainManager is discussed first followed by the FileManager, the ApplicationFactory, and then the Application.

We now describe the types defined or used within the DomainManager (see Figure 6.1).

6.1.1 Types

ApplicationSequence

The ApplicationSequence defines an unbounded sequence of Application instances. An Application is added to the applications attribute when it has been successfully instantiated

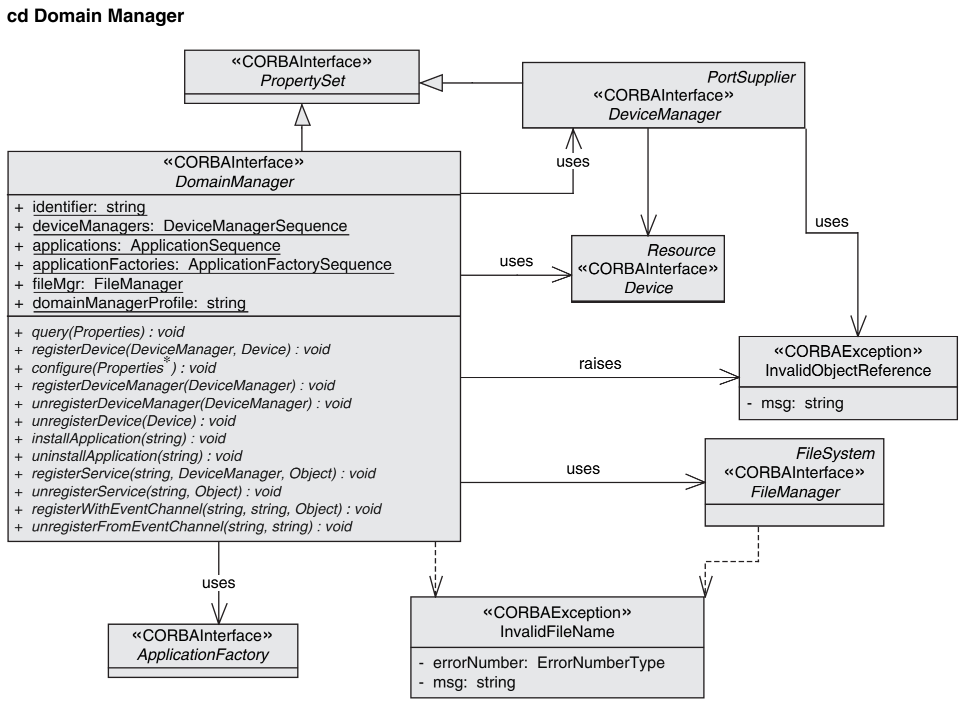

cd Domain Manager
«CORBAInterface» PropertySet
PortSupplier «CORBAInterface» DeviceManager
«CORBAInterface» DomainManager
+ identifier: string
+ deviceManagers: DeviceManagerSequence
+ applications: ApplicationSequence
+ applicationFactories: ApplicationFactorySequence
+ fileMgr: FileManager
+ domainManagerProfile: string
+ query(Properties) : void
+ registerDevice(DeviceManager, Device) : void
+ configure(Properties*) : void
+ registerDeviceManager(DeviceManager) : void
+ unregisterDeviceManager(DeviceManager) : void
+ unregisterDevice(Device) : void
+ installApplication(string) : void
+ uninstallApplication(string) : void
+ registerService(string, DeviceManager, Object) : void
+ unregisterService(string, Object) : void
+ registerWithEventChannel(string, string, Object) : void
+ unregisterFromEventChannel(string, string) : void
uses
Resource «CORBAInterface» Device
uses
raises
«CORBAException» InvalidObjectReference
- msg: string
uses
FileSystem «CORBAInterface» FileManager
uses
«CORBAInterface» ApplicationFactory
«CORBAException» InvalidFileName
- errorNumber: ErrorNumberType
- msg: string

Figure 6.1. Domain Manager Interfaces

by the ApplicationFactory. If no Applications have been instantiated, the applications value will be nil.

```
typedef sequence <Application> ApplicationSequence;
```

ApplicationFactorySequence

The ApplicationFactorySequence defines an unbounded sequence of ApplicationFactory instances. An ApplicationFactory is instantiated and added to the sequence of applicationFactories attribute as part of the installApplication operation on the DomainManager. If no applications have been installed then the applicationFactories value will be nil.

```
typedef sequence <ApplicationFactory>
      ApplicationFactorySequence;
```

DeviceManagerSequence

The DeviceManagerSequence defines an unbounded sequence of DeviceManager instances. A DeviceManager is added to the list of deviceManagers when it is registered with the

DomainManager. If no DeviceManager instances have registered with the DomainManager then the value of deviceManagers will be nil.

```
typedef sequence <DeviceManager>
      DeviceManagerSequence;
```

6.1.2 Exceptions

ApplicationInstallError

If the installApplication operation on the DomainManager does not successfully complete, the ApplicationInstallError is raised. A numeric value is included as part of the exception to identify the type of error encountered. Also, a string parameter is available to provide additional information regarding the error encountered.

```
exception ApplicationInstallationError {
      ErrorNumberType errorNumber;
      string msg;
      };
typedef sequence <DeviceManager> DeviceManagerSequence;
```

InvalidIdentifier

When an applicationID is provided, if the Id does not match any known applications, then an InvalidIdentifier exception is raised.

```
exception InvalidIdentifier {
};
```

DeviceManagerNotRegistered

When registering a Device, the DeviceManager with which the Device is associated must be registered. If the DeviceManager is not in the registeredDevices sequence, then the DeviceManagerNotRegistered exception is raised.

```
exception DeviceManagerNotRegistered {
};
```

ApplicationUninstallationError

When the DomainManager is instructed to uninstall an application, and the uninstall-Application operation does not complete successfully, then the ApplicationUninstallError exception is raised.

```
exception ApplicationUninstallationError {
      ErrorNumberType errorNumber;
      string msg;
};
```

RegisterError

When registering with the DomainManager, if any error is encountered during the registration process, then the RegisterError exception is raised. This exception may be raised when attempting to register a DeviceManager, Device, or Service. An error number is provided to indicate the type of error encountered and a string is included on the exception to provide more specific detail regarding the error.

```
exception RegisterError {
      ErrorNumberType errorNumber;
      string msg;
};
```

UnregisterError

When unregistering with the DomainManager, if any error is encountered during the unregistration process, then the UnregisterError exception is raised. This exception may be raised when attempting to unregister a DeviceManager, Device, or Service. An error number is provided to indicate the type of error encountered and a string is included on the exception to provide more specific detail regarding the error.

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

AlreadyConnected

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

```
exception AlreadyConnected {
};
```

InvalidEventChannel

The InvalidEventChannelName exception indicates that a DomainManager was not able to locate the event channel.

```
exception InvalidEventChannelName {
};
```

NotConnected

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

```
exception NotConnected {
};
```

6.1.3 Attributes

The DomainManager Attribute requirements are shown in Table 6.1.

Table 6.1. DomainManager Attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.3.4.1	SR:205	CF	The DomainManager shall write an `ADMINISTRATIVE_EVENT` log to a DomainManager's log, when the deviceManagers attribute is obtained by a client.
3.1.3.2.3.4.2	SR:206	CF	The readonly applications attribute shall contain the list of Applications that have been instantiated.
3.1.3.2.3.4.3	SR:208	CF	The readonly applicationFactories attribute shall contain a list with one ApplicationFactory per application (Software Assembly Descriptor (SAD) file and associated files) successfully installed (i.e. no exception raised).
3.1.3.2.3.4.3	SR:209	CF	The DomainManager shall write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log, when the applicationFactories attribute is obtained by a client.
3.1.3.2.3.4.4	SR:210	CF	The readonly fileMgr attribute shall contain the DomainManager's FileManager.
3.1.3.2.3.4.4	SR:211	CF	The DomainManager shall write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log, when the fileMgr attribute is obtained by a client.
3.1.3.2.3.4.5	SR:212	CF	The readonly domainManagerProfile attribute shall contain either a profile element with a file reference to the DomainManager Configuration Descriptor (DMD) profile or the XML for the DomainManager's (DMD) profile.
3.1.3.2.3.4.6	SR:213	CF	The readonly identifier attribute shall contain a unique identifier for a DomainManager instance.
3.1.3.2.3.4.6	SR:214	CF	The identifier shall be identical to the domainmanagerconfiguration element Id attribute of the DomainManager's Descriptor (DMD) file.

domainManagerProfile

The domainManagerProfile is a readonly attribute that contains a file reference to the Domain Manager Descriptor (DMD) XML file (SR:212). Its value is set during the startup and initialization of the DomainManager.

```
readonly attribute string domainManagerProfile;
```

deviceManagers

The deviceManagers attribute is readonly and contains a sequence of DeviceManagers that have registered with the DomainManager. If no DeviceManagers have registered with the DomainManager, then the value of this attribute is nil.

```
readonly attribute DeviceManagerSequence deviceManagers;
```

Whenever another application or program requests the contents of the deviceManagers attribute, the DomainManager writes an `ADMINISTRATIVE_EVENT` log record (SR:205) to the log identifying the client requesting the information and when the request was made.

applications

The applications attribute is readonly and contains a sequence of Applications (SR:206) that have been instantiated by the appropriate ApplicationFactory within the DomainManager. If no applications have been instantiated, then the value of this attribute is nil.

```
readonly attribute ApplicationSequence applications;
```

Whenever another application or program requests the contents of the applications attribute, the DomainManager writes an `ADMINISTRATIVE_EVENT` log record to the log identifying the client requesting the information and when the request was made.

applicationFactories

The applicationFactories attribute is readonly and contains a sequence of ApplicationFactory instances that have been instantiated by the DomainManager (SR:208) as part of the installApplication operation. If no application has been installed, then the value of this attribute is nil.

```
readonly attribute ApplicationFactorySequence
     applicationFactories;
```

Whenever another application or program requests the contents of the application-Factories attribute, the DomainManager writes an `ADMINISTRATIVE_EVENT` log record (SR:209) to the log identifying the client requesting the information and when the request was made.

fileMgr

The fileMgr attribute is readonly and contains a list of the mounted FileSystems (SR:210). A FileSystem may be hosted by a DeviceManager. Thus, this attribute contains references to FileSystems mounted by one or more DeviceManagers in the system.

```
readonly attribute FileManager fileMgr;
```

Whenever another application or program requests the contents of the fileMgr attribute, the DomainManager writes an `ADMINISTRATIVE_EVENT` log record (SR:211) to the log identifying the client requesting the information and when the request was made.

identifier

The identifier attribute is readonly and contains a unique identifier for the DomainManager (SR:213). The value of the attribute is populated by the DomainManager as part of its startup and initialization. The value of the identifier attribute is specified by the attribute of the domainmanagerconfiguration element in the Domain Manger Descriptor (DMD) file (SR:214).

```
readonly attribute string identifier;
```

6.1.4 DomainManager Instantiation

There are a number of actions that must be performed as part of the startup and initialization of the DomainManager. These requirements are, effectively, levied upon the constructor or initialization routine called upon instantiation of the DomainManager (see Table 6.2).

Table 6.2. DomainManager Startup requirements

Section	ID	Resp	Requirement
3.1.3.2.3.5	SR:215	CF	During component construction the DomainManager shall register itself with the CORBA Naming Service.
3.1.3.2.3.5	SR:216	CF	During Naming Service registration the DomainManager shall create a ‘naming context’ using ‘/DomainName’ as its name.ID component and “” (Null string) as its name.kind component, then create a ‘name binding’ to the ‘/DomainName’ naming context using ‘/DomainManager’ as its name.ID component.
3.1.3.2.3.5	SR:217	XML	The logs utilized by the DomainManager implementation shall be defined in the DMD.
3.1.3.2.3.5	SR:218	CF	Once a service specified in the DMD is successfully registered with the DomainManager (via registerDeviceManager or registerService operations), the DomainManager shall begin to use the service (e.g. log).
3.1.3.2.3.5	SR:219	CF	The DomainManager shall create its own FileManager component that consists of all registered DeviceManager’s FileSystems.
3.1.3.2.3.5	SR:220	CF	The DomainManager shall restore ApplicationFactories after startup for applications that were previously installed by the DomainManager installApplication operation.
3.1.3.2.3.5	SR:221	CF	The DomainManager shall add the restored ApplicationFactories to the DomainManager’s applicationFactories attribute.
3.1.3.2.3.5	SR:222	CF	The DomainManager shall create the Incoming Domain Management and Outgoing Domain Management event channels.

Any logs that are to be used by the DomainManager are identified within the DMD file (SR:217). In addition, any services registered with the DomainManager, either via the registerDeviceManager or registerService operations, may be used by the DomainManager (SR:218).

The DomainManager also creates an instance of a FileManager and stores it on the fileMgr attribute. Initially the fileMgr is empty. As DeviceManagers register with the DomainManager, all FileSystems hosted by the DeviceManager are added to the fileMgr attribute (SR:219).

As the DomainManager continues its initialization, any ApplicationFactories that were present in the DomainManager at the time it was last shutdown are restored (SR:220) via the installApplication operation. As each Application is installed, the ApplicationFactory instance is added to the application Factories attribute (SR:221).

The Domain Manager then creates the Incoming Domain Management (IDM) and Outgoing Domain Management (ODM) event channels (SR:222).

As the DomainManager completes its initialization, it registers with the Naming Service (SR:215). The registration creates a naming context of /<*SomeDomainName*> and creates a name binding of */DomainManager* to this naming context (SR:216). This makes the DomainManager accessible to other applications. Note that the requirements do not specify when the DomainManager registers with the Naming Service. However, once registered, it should be completely initialized and ready to accept calls from other components.

6.1.5 Operations

registerDeviceManager

The registerDeviceManager, as the name implies, is used to register a DeviceManager with the DomainManager. This operation has a several responsibilities as part of its execution.

```
void registerDeviceManager (
     in DeviceManager deviceMgr
     )
raises (InvalidObjectReference, InvalidProfile,
      RegisterError);
```

The only parameter is the DeviceManager instance to be registered with the Domain-Manager. As shown, the possible exceptions are InvalidObjectReference, InvalidProfile, and RegisterError. The requirements are listed in Table 6.3.

The operation first checks that the deviceMgr parameter is a valid CORBA reference for a DeviceManager instance. If it is an invalid or nil reference then the InvalidObjectReference exception is raised (SR:240) and the operation terminates. If the reference is valid, then the DeviceManager is added to the deviceMgr attribute (SR:223), if the DeviceManager is not already a member of the attribute. The operation then adds each of the DeviceManager's registeredDevices and the attributes for each of the registeredDevices to the DomainManager (SR:224). There is no attribute specified in the IDL or requirement specified regarding the processing and storage of the Device information provided by the registeredDevices attribute on the registering DeviceManager. How the information is stored and managed within the DomainManager is at the discretion of the Core Framework developer. Each Device must

Table 6.3. registerDeviceManager requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.1.3	SR:223	CF	The registerDeviceManager operation shall add the input deviceMgr to the DomainManager's deviceManagers attribute, if it does not already exist.
3.1.3.2.3.6.1.3	SR:224	CF	The registerDeviceManager operation shall add the input deviceMgr's registeredDevices and each registeredDevice's attributes (e.g. identifier, softwareProfile's allocation properties, etc.) to the DomainManager.
3.1.3.2.3.6.1.3	SR:225	CF	The registerDeviceManager operation shall add the input deviceMgr's registeredServices and each registeredService's names to the DomainManager.
3.1.3.2.3.6.1.3	SR:226	CF	The registerDeviceManager operation shall perform the connections specified in the connections element of the deviceMgr's Device Configuration Descriptor (DCD) file.
3.1.3.2.3.6.1.3	SR:227	CF	If the DeviceManager's DCD describes a connection for a service that has not been registered with the DomainManager, the registerDeviceManager operation shall establish any pending connection when the service registers with the DomainManager by the registerDeviceManager operation.
3.1.3.2.3.6.1.3	SR:228	CF	For connections established for a CORBA Event Service's event channel, the registerDeviceManager operation shall connect a CosEventComm PushConsumer or PushSupplier object to the event channel as specified in the DCD's domainfinder element.
3.1.3.2.3.6.1.3	SR:229	CF	If the event channel does not exist, the registerDeviceManager operation shall create the event channel.
3.1.3.2.3.6.1.3	SR:230	CF	The registerDeviceManager operation shall obtain all the Software profiles from the registering DeviceManager's FileSystems.
3.1.3.2.3.6.1.3	SR:231	CF	The registerDeviceManager operation shall mount the DeviceManager's FileSystem to the DomainManager's FileManager.
3.1.3.2.3.6.1.3	SR:232	CF	The mounted FileSystem name shall have the format '/DomainName/HostName', where DomainName is the name of the domain and HostName is the input deviceMgr's label attribute.
3.1.3.2.3.6.1.3	SR:233	CF	The registerDeviceManager operation shall, upon unsuccessful DeviceManager registration, write a `FAILURE_ALARM` log record to a DomainManager's log.

Table 6.3. Continued

Section	ID	Resp	Requirement
3.1.3.2.3.6.1.3	SR:233	CF	The registerDeviceManager operation shall, upon unsuccessful DeviceManager registration, write a `FAILURE_ALARM` log record to a DomainManager's log.
3.1.3.2.3.6.1.3	SR:234	CF	The registerDeviceManager operation shall, upon successful DeviceManager registration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.
3.1.3.2.3.6.1.3	SR:235	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.1.3	SR:236	CF	The sourceId shall be the identifier attribute of the registered DeviceManager.
3.1.3.2.3.6.1.3	SR:237	CF	The sourceName shall be the label attribute of the registered DeviceManager.
3.1.3.2.3.6.1.3	SR:238	CF	The sourceIOR shall be the registered DeviceManager object reference.
3.1.3.2.3.6.1.3	SR:239	CF	The sourceCategory shall be `DEVICE_MANAGER`.
3.1.3.2.3.6.1.5	SR:240	CF	The registerDeviceManager operation shall raise the Core Framework InvalidObjectReference exception when the input parameter deviceMgr contains an invalid reference to a DeviceManager interface.
3.1.3.2.3.6.1.5	SR:241	CF	The registerDeviceManager operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

be associated with the input DeviceManager to ensure the proper cleanup of Devices as part of the unregisterDevice Manager.

The operation then adds the Services registered with the input DeviceManager to the DomainManager (SR:225). As with the Devices, the DomainManager associates each service with the input DeviceManager so that the services will be cleaned up as part of the unregisterDeviceManager operation.

Any connections specified in the deviceMgr's Device Descriptor Definition (DCD) file are established (SR:226). If a connection specified in the DCD specifies a service that has not yet registered, then the connection request is put in a pending state and, when the service registers with the DomainManager, the connection is established (SR:227). If a connection specifies an event channel, then a PushConsumer or PushSupplier object, as specified in the DCD's domainfinder element, is connected to the event channel (SR:228). If the event channel does not exist, then the event channel is created (SR:229).

The software profiles are obtained through the FileSystem hosted by the registering DeviceManager (SR:230). The FileSystem identified is then added to the set of FileSystems maintained by the fileMgr attribute (SR:231). Each FileSystem mounted in the FileManager uses the name format `/<DomainName>/<HostName>` where `<DomainName>` is the name of the domain and `<HostName>` is the name associated with the DeviceManager as specified by the input DeviceManager's label attribute (SR:232).

If the registration process does not successfully complete, a `FAILURE_ALARM` log record is written to a DomainManager log (SR:233). If the registration process is successful, then an event is sent on the ODM event channel (SR:234) containing the ProducerId set to the DomainManager identifier (SR:235), the SourceId set to the input DeviceManager identifier (SR:236), the SourceName set to the label attribute of the input DeviceManager (SR:237), the sourceIOR set to the input DeviceManager's object reference (SR:238), and the sourceCategory set to `DEVICE_MANAGER` (SR:239).

If at any point in the above processing an error is encountered, then the RegisterError exception is raised (SR:241).

registerDevice

The registerDevice operation adds the Device to the set of Devices maintained by the DomainManager across the entire domain. The Device to be registered and the registered DeviceManager with which the Device is associated is provided as input.

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

Exceptions that may be raised are the InvalidObjectReference if the Device reference provided is not valid, InvalidProfile if the Device's domain profile is not valid, DeviceManagerNotRegistered if the DeviceManager provided is not already registered with the DomainManager, or RegisterError if an error is encountered during the registration process. The requirements are listed in Table 6.4.

The registerDevice operation first checks the Device and DeviceManager references provided by the input parameters registeringDevice and registeredDeviceMgr, respectively. If either reference is nil or does not reference a valid Device or DeviceManager object, then the InvalidObjectReference exception is raised (SR:257) and the operation terminates. The operation then checks that the DeviceManager reference provided has already been registered with the DomainManager, i.e. it is already in the deviceManagers attribute. If not, then a `FAILURE_ALARM` log record is written to a DomainManager log indicating the DeviceManager is not registered with the DomainManager (SR:246) and the DeviceManagerNotRegistered exception is raised (SR:256) and the execution terminates. In both exception cases above, a `FAILURE_ALARM` log record is written to a DomainManager log (SR:247; SR:248) prior to raising the exception and terminating execution.

The operation then associates the Device with the input DeviceManager so that proper cleanup of the Device is supported when the DeviceManager is unregistered. The Device is

Table 6.4. registerDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.2.3	SR:242	CF	The registerDevice operation shall add the registeringDevice and the registeringDevice's attributes (e.g. identifier, softwareProfile's allocation properties, etc.) to the DomainManager, if it does not already exist.
3.1.3.2.3.6.2.3	SR:243	CF	When the registering Device's parent DeviceManager's DCD describes service connections for the registering Device, the registerDevice operation shall establish the connections.
3.1.3.2.3.6.2.3	SR:244	CF	The registerDevice operation shall, upon successful device registration, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log, to indicate that the device has successfully registered with the DomainManager.
3.1.3.2.3.6.2.3	SR:245	CF	Upon unsuccessful device registration, the registerDevice operation shall write a `FAILURE_ALARM` log record to a DomainManager's log, when the InvalidProfile exception is raised to indicate that the registeringDevice has an invalid profile.
3.1.3.2.3.6.2.3	SR:246	CF	Upon unsuccessful device registration, the registerDevice operation shall write a `FAILURE_ALARM` log record to a DomainManager's log, indicating that the device could not register because the DeviceManager is not registered with the DomainManager.
3.1.3.2.3.6.2.3	SR:247	CF	Upon unsuccessful device registration, the registerDevice operation shall write a `FAILURE_ALARM` log record to a DomainManager's log, because of an invalid reference input parameter.
3.1.3.2.3.6.2.3	SR:248	CF	Upon unsuccessful device registration, the registerDevice operation shall write a `FAILURE_ALARM` log record to a DomainManager's Log, because of an internal registration error.
3.1.3.2.3.6.2.3	SR:249	CF	The registerDevice operation shall, upon successful Device registration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.

3.1.3.2.3.6.2.3	SR:250	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.2.3	SR:251	CF	The sourceId shall be the identifier attribute of the registered Device.
3.1.3.2.3.6.2.3	SR:252	CF	The sourceName shall be the label attribute of the registered Device.
3.1.3.2.3.6.2.3	SR:253	CF	The sourceIOR shall be the registered Device object reference.
3.1.3.2.3.6.2.3	SR:254	CF	The sourceCategory shall be `DEVICE`.
3.1.3.2.3.6.2.5	SR:255	CF	The registerDevice operation shall raise the CF InvalidProfile exception when: the Device's SPD file and the SPD's referenced files do not exist or cannot be processed due to the file not being compliant with XML syntax, or the Device's SPD does not reference allocation properties.
3.1.3.2.3.6.2.5	SR:256	CF	The registerDevice operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr (not nil reference) is not registered with the DomainManager.
3.1.3.2.3.6.2.5	SR:257	CF	The registerDevice operation shall raise the CF InvalidObjectReference exception when input parameters registeringDevice or registeredDeviceMgr contains an invalid reference.
3.1.3.2.3.6.2.5	SR:258	CF	The registerDevice operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

added to the set of Devices maintained by the DomainManager (SR:242), if the Device does not already exist in the Device set. Each of the Device's attributes are then retrieved from the Device and stored in the DomainManager.

The Device's Software Package Descriptor (SDP) XML file is then read and the Device's properties are retrieved. This step is optional if the XML file has already been processed and has not changed since the last time the Device was registered. If the SPD file does not exist or contains errors, then a `FAILURE_ALARM` log record is written to a DomainManager log (SR:245) and the InvalidProfile exception is raised (SR:255) and execution terminates.

If any service connections are specified by the input DeviceManager for the registering Device, then the operation will establish those connections (SR:243).

Upon successful completion of the registration process, the DomainManager writes an event to the ODM event channel containing a DomainManagementObjectAddedEventType (SR:249). The contents of the record include the DomainManager as the producerId (SR:250), the identifier attribute of the registered Device as the sourceId (SR:251), the label attribute of the registered Device as the sourceName (SR:252), the registered Device's object reference as the sourceIOR (SR:253), and `DEVICE` as the sourceCategory (SR:234).

If any other error is encountered during the registration process then the RegisterError exception is raised (SR:258) and the execution terminates.

In all cases when an exception is raised and the operation terminates, it is expected that the exception handler will restore the DomainManager to the state it was in prior to initiation of the operation.

installApplication

The installApplication is used to initiate the processing of the XML files that define, the components, dependencies, and connections of a waveform application, starting with the Software Assembly Descriptor (SAD) file. The result of the installApplication operation is the creation of an instance of an ApplicationFactory for the waveform application defined in the XML files.

The only input parameter is a string value which is the name of the SAD file for the application.

```
void installApplication (
     in string profileFileName
     )
raises (InvalidProfile, InvalidFileName,
     ApplicationInstallationError);
```

The operation may raise an InvalidProfile exception if the profile file name provided does not exist or contains error, an InvalidFileName if the file name provided does not adhere to the naming conventions imposed in the general requirements, or an ApplicationInstallationError if any error is encountered during the installApplication operation that prevents it from completing successfully. The requirements are listed in Table 6.5.

Table 6.5. installApplication requirements

Section	ID	Resp	Requirement
3.1.3.2.3.3.1	SR:200	CF	The error number shall indicate an ErrorNumberType value (e.g. EINVAL, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, ENXIO).
3.1.3.2.3.6.3.3	SR:259	CF	The installApplication operation shall verify that the application's SAD file exists in the DomainManager's FileManager and all the files on which the application is dependent are also resident.
3.1.3.2.3.6.3.3	SR:260	CF	The installApplication operation shall write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log, upon successful Application installation.
3.1.3.2.3.6.3.3	SR:261	CF	The installApplication operation shall, upon unsuccessful application installation, write a `FAILURE_ALARM` log record to a DomainManager's log.

3.1.3.2.3.6.3.3	SR:262	CF	The installApplication operation shall, upon successful application installation, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.
3.1.3.2.3.6.3.3	SR:263	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.3.3	SR:264	CF	The sourceId shall be the identifier attribute of the installed ApplicationFactory.
3.1.3.2.3.6.3.3	SR:265	CF	The sourceName shall be the name attribute of the installed ApplicationFactory.
3.1.3.2.3.6.3.3	SR:266	CF	The sourceIOR shall be the installed ApplicationFactory object reference.
3.1.3.2.3.6.3.3	SR:267	CF	The sourceCategory shall be `APPLICATION_FACTORY`.
3.1.3.2.3.6.3.5	SR:268	CF	The installApplication operation shall raise the ApplicationInstallationError exception when the installation of the Application file(s) was not successfully completed.
3.1.3.2.3.6.3.5	SR:269	CF	The installApplication operation shall raise the InvalidFileName exception when the input SAD file or any referenced file name does not exist in the file system as defined in the absolute path of the input profileFile Name.
3.1.3.2.3.6.3.5	SR:270	CF	When the InvalidFileName exception occurs, the installApplication operation shall log a FAILURE_ALARM log record to a DomainManager's log with a message consisting of 'installApplication::invalid file is xxx', where 'xxx' is the input or referenced file n.
3.1.3.2.3.6.3.5	SR:271	CF	The installApplication operation shall raise the CF InvalidProfile exception when the input SAD file or any referenced file is not compliant with XML DTDs defined in Appendix D of the specifications or referenced property definitions are missing.
3.1.3.2.3.6.3.5	SR:272	CF	When the CF InvalidProfile exception occurs, the installApplication operation shall log a `FAILURE_ALARM` log record to a DomainManager's log with a message consisting of 'installApplication::invalid Profile is yyy,' where 'yyy' is the input or referenced f.

The installApplication operation starts by validating the string input parameter that specifies the file name for the top-level SAD file defining the application and the existence of the file (SR:259). The file name provided must include the absolute path name, in terms of the SCA FileSystem mount point, for the file. If the file name is invalid, i.e. it does not adhere to the naming conventions specified in the SCA specification or does not exist in the FileSystem specified by the absolute path of the file name parameter, then a log record is written to a DomainManager log identifying that an InvalidFileName exception was encountered and identifying the file name (SR:269), an InvalidFileName exception is raised (SR:269), and execution is terminated.

The operation then starts processing the XML file specified by the file name parameter. If a syntax error is encountered, based on the Document Type Definition (DTD) files, then a log record is written to a DomainManager log specifying that an InvalidProfile exception was encountered with the file name in the message (SR:272), the InvalidProfile exception is raised (SR:271), and execution is terminated.

If any other error is encountered that prevents the installApplication from successfully completing, a `FAILURE_ALARM` log record is written to a DomainManager log (SR:260), the ApplicationInstallationError is raised (SR:268), and execution terminates. The ApplicationInstallationError exception contains an errorNumber that indicates the type of error encountered (SR:200). In all cases when an exception is raised and execution terminates, the installApplication operation must return the DomainManager to the stable state prior to initiation of the installApplication operation.

If installation of the application is successful, an event is sent on the ODM with the event data consisting of a DomainManagementObjectAddedEventType (SR:262). The log record data includes the producerId set to the identifier of the DomainManager (SR:263), the sourceId set to the identifier attribute of the ApplicationFactory (SR:264), the sourceName set to the name attribute of the installed Application Factory (SR:265), the sourceIOR set to the new ApplicationFactory instance object reference (SR:266), and the sourceCategory set to `APPLICATION_FACTORY` (SR:267).

unregisterDeviceManager

The unregisterDeviceManager unregisters the DeviceManager from the DomainManager's profile. This operation has a number of side effects including removal of the Device associated with the DeviceManager from the DomainManager, disconnection from any services used by the DeviceManager or any of the Devices associated with the DeviceManager, removal of the services hosted by the DeviceManager, and removal of any FileSystem hosted by the DeviceManager.

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

The requirements are listed in Table 6.6.

The unregisterDeviceManager unregisters a DeviceManager from the DomainManager (SR:273). The first action performed is to validate that the DeviceManager object reference provided refers to a valid DeviceManager. If not, the InvalidObjectReference exception is raised (SR:284). As noted above, there are several side effects and processing performed as part of the operation. Although there is no requirement, the operation should ensure that the input DeviceManager is one of the registeredDeviceManagers in the DomainManager. If it is not, then the operation should terminate with no error conditions raised.

All consumers and producers are disconnected from the event service to which they are connected (SR:275). Then the operation releases all Devices associated with the DeviceManager being unregistered (SR:274). The DomainManager unmounts, i.e. removes

Table 6.6. unregisterDeviceManager requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.4.3	SR:273	CF	The unregisterDeviceManager operation shall unregister a DeviceManager component from the DomainManager.
3.1.3.2.3.6.4.3	SR:274	CF	The unregisterDeviceManager operation shall release all device(s) and service(s) associated with the DeviceManager that is being unregistered.
3.1.3.2.3.6.4.3	SR:275	CF	The unregisterDeviceManager operation shall disconnect consumers and producers (e.g. Devices, Log, DeviceManager, etc.) from a CORBA Event Service event channel based upon the software profile.
3.1.3.2.3.6.4.3	SR:276	CF	The unregisterDeviceManager operation shall unmount all DeviceManager's FileSystems from its File Manager.
3.1.3.2.3.6.4.3	SR:277	CF	The unregisterDeviceManager operation shall, upon the successful unregistration of a DeviceManager, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log.
3.1.3.2.3.6.4.3	SR:278	CF	The unregisterDeviceManager operation shall, upon unsuccessful unregistration of a DeviceManager, write a `FAILURE_ALARM` log record to a DomainManager's log.
3.1.3.2.3.6.4.3	SR:279	CF	The unregisterDeviceManager operation shall, upon successful unregistration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType.
3.1.3.2.3.6.4.3	SR:280	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.4.3	SR:281	CF	The sourceId shall be the identifier attribute of the unregistered DeviceManager.
3.1.3.2.3.6.4.3	SR:282	CF	The sourceName shall be the label attribute of the unregistered DeviceManager.
3.1.3.2.3.6.4.3	SR:283	CF	The sourceCategory shall be `DEVICE_MANAGER`.
3.1.3.2.3.6.4.5	SR:284	CF	The unregisterDeviceManager operation shall raise the CF InvalidObjectReference when the input parameter DeviceManager contains an invalid reference to a DeviceManager interface.
3.1.3.2.3.6.4.5	SR:285	CF	The unregisterDeviceManager operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

from the fileMgr attribute, any FileSystem hosted by the DeviceManager (SR:276) that has been mounted as part of the DomainManager's federated file system.

After the DeviceManager has been unregistered and removed from the deviceManagers attribute, a ADMINISTRATIVE_EVENT log record is written to a DomainManager log (SR:277). In addition, a DomainManagementObjectRemovedEventType event is sent on the ODM channel indicating that the DeviceManager has been removed (SR:279) with the producerId set to the identifier attribute of the DomainManager (SR:280), the sourceId set to the identifier attribute of the unregistered DeviceManager (SR:281), the sourceName set to the label attribute of the unregistered DeviceManager (SR:282), and the sourceCategory set to DEVICE_MANAGER (SR:283).

If the DeviceManager does not successfully, it writes a FAILURE_ALARM (SR:278). If any other error is encountered during the unregisterDeviceManager operation, an UnregisterException is raised (SR:285). It is expected that the exception handler for this and any other error encountered will return the DomainManager to the stable state prior to initiation of the operation.

unregisterDevice

The unregisterDevice operation is similar in concept to the unregisterDeviceManager operation discussed in the previous section. The difference is that the unregisterDevice is usually called within the context of an unregisterDevice operation. It is available as an accessible operation because it allows the DeviceManager, for example, to remove a Device from the DomainManager should the Device fail or need to be taken out of service.

```
void unregisterDevice (
    in Device unregisteringDevice
    )
raises (InvalidObjectReference, UnregisterError);
```

The only input parameter is the object reference for the device to be unregistered. The possible exceptions are the InvalidObjectReference exception if the object reference provided is invalid or the UnregisterError exception if some error is encountered during the unregistration process. The requirements are listed in Table 6.7.

The unregisterDevice operation is used to remove a Device from the DomainManager (SR:286). The operation first checks the object reference provided for the Device. If it cannot be resolved to a Device then an InvalidObjectReference exception is raised (SR:296).

The Device's event channel consumers and producers are disconnected (SR:288), the Device is then released from the DomainManager (SR:287), and an ADMINISTRATIVE_EVENT log record is written to a DomainManager log (SR:289). An event is sent to the ODM (SR:291) indicating that the device has been unregistered with the producerId set to the identifier attribute of the DomainManager (SR:292) the sourceID set to the identifier attribute of the unregistered Device (SR:293), the sourceName set to the label attribute of the unregistered Device (SR:294), and the sourceCategory set to DEVICE (SR:295).

If any other error is encountered during the unregistration process, then an UnregisterError exception is raised (SR:297). If this or any other error is encountered, the exception handler should return the DomainManager to a stable state.

Table 6.7. unregisterDevice requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.5.3	SR:286	CF	The unregisterDevice operation shall remove a device entry from the DomainManager.
3.1.3.2.3.6.5.3	SR:287	CF	The unregisterDevice operation shall release (client-side CORBA release) the unregisteringDevice from the Domain Manager.
3.1.3.2.3.6.5.3	SR:288	CF	The unregisterDevice operation shall disconnect the Device's consumers and producers from a CORBA Event Service event channel based upon the software profile.
3.1.3.2.3.6.5.3	SR:289	CF	The unregisterDevice operation shall, upon the successful unregistration of a Device, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log.
3.1.3.2.3.6.5.3	SR:291	CF	The unregisterDevice operation shall, upon successful Device unregistration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType.
3.1.3.2.3.6.5.3	SR:292	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.5.3	SR:293	CF	The sourceId shall be the identifier attribute of the unregistered Device.
3.1.3.2.3.6.5.3	SR:294	CF	The sourceName shall be the label attribute of the unregistered Device.
3.1.3.2.3.6.5.3	SR:295	CF	The sourceCategory shall be `DEVICE`.
3.1.3.2.3.6.5.5	SR:296	CF	The unregisterDevice operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a Device interface.
3.1.3.2.3.6.5.5	SR:297	CF	The unregisterDevice operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

uninstallApplication

The uninstallApplication removes an installed application from the DomainManager. This means that the ApplicationFactory associated with the application is destroyed. It does not imply, however, that any instantiated application is destroyed or halted.

```
void uninstallApplication (
     in string applicationId
     )
raises (InvalidIdentifier, ApplicationUninstallationError);
```

The input parameter provided is the string name of the ApplicationFactory identifier. This is the name provided by the identifier attribute of the ApplicationFactory. Exceptions that may be raised are the InvalidIdentifier exception if the string name provided does not match any instantiated Applications and the ApplicationUninstallationError exception if any other error is encountered as part of the uninstall process. The requirements are listed in Table 6.8.

Table 6.8. uninstallApplication requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.6.3	SCA182	CF	The error number shall indicate an ErrorNumberType value.
3.1.3.2.3.6.6.3	SR:298	CF	The uninstallApplication operation shall remove all files associated with the Application.
3.1.3.2.3.6.6.3	SR:299	CF	The uninstallApplication operation shall make the ApplicationFactory unavailable from the DomainManager (i.e. its services no longer provided for the Application).
3.1.3.2.3.6.6.3	SR:300	CF	The uninstallApplication operation shall, upon successful uninstall of an Application, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log.
3.1.3.2.3.6.6.3	SR:301	CF	The uninstallApplication operation shall, upon unsuccessful uninstall of an Application, log a `FAILURE_ALARM` log record to a DomainManager's log.
3.1.3.2.3.6.6.3	SR:303	CF	The uninstallApplication operation shall, upon successful uninstall of an application, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType.
3.1.3.2.3.6.6.3	SR:304	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.6.3	SR:305	CF	The sourceId shall be the identifier attribute of the uninstalled ApplicationFactory.
3.1.3.2.3.6.6.3	SR:306	CF	The sourceName shall be the name attribute of the uninstalled ApplicationFactory.
3.1.3.2.3.6.6.3	SR:307	CF	The sourceCategory shall be `APPLICATION_FACTORY`.
3.1.3.2.3.6.6.5	SR:308	CF	The uninstallApplication operation shall raise the InvalidIdentifier exception when the ApplicationId is invalid.
3.1.3.2.3.6.6.5	SR:309	CF	The uninstallApplication operation shall raise the ApplicationUninstallationError exception when an internal error causes unsuccessful uninstall of the application.

The first action performed is to search the list of applicationFactories for the ApplicationFactory with the name provided. If the name specified is not found in the sequence of applicationFactories, then an InvalidIdentifier exception is raised (SR:308) and the operation terminates.

The ApplicationFactory specified is made unavailable (SR:299). This means calls to instantiate an application are no longer accepted. The XML files that comprise the Domain Profile for the ApplicationFactory are removed (SCA2798).[1]

The ApplicationFactory is then removed from the DomainManager's applicationFactories attribute and destroyed. An `ADMINISTRATIVE_EVENT` log record is written to a DomainManager log (SR:300), and an event is sent to the ODM (SR:303) indicating that the application has been uninstalled with the producerId set to the identifier attribute of the DomainManager (SR:304), the sourceID set to the identifier attribute of the uninstalled ApplicationFactory (SR:305), the sourceName set to the name attribute of the uninstalled ApplicationFactory (SR:306), and the sourceCategory set to `APPLICATION_FACTORY` (SR:308).

If any other error is encountered during the unregistration process, then a `FAILURE_ALARM` log record is written to a DomainManager log (SR:301) and an ApplicationUninstallationError exception is raised (SR:309). If this or any other error is encountered, the exception handler should return the DomainManager to a stable state.

registerService

The registerService operation registers a service with the DomainManager. The service is hosted by a DeviceManager. The operation provides an object reference to the service to be registered, an object reference to the DeviceManager hosting the service, and a string name that identifies the service.

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

Exceptions that may be raised include InvalidObjectReference if the object reference provided for either the service or the DeviceManager are invalid, InvalidProfile if the XML files associated with the DeviceManager cannot be found or contain syntax errors, DeviceManagerNotRegistered if the DeviceManager specified is not already registered with the DomainManager, and RegisterError if any other error is encountered as part of the registration process. The requirements are listed in Table 6.9.

[1] *This requirement has resulted in continued debate as to whether or not the XML files should be removed. In the event that the application is to be re-installed, the XML files no longer exist and must be re-loaded onto the system in order to re-install the application. Most Core Framework implementations provide a means to keep the XML files as an option. In version 2.2.2, this requirement has been amended to state that the XML files should not be removed.*

Table 6.9. registerService requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.7.3	SR:310	CF	The registerService operation shall verify the input registeringService and registeredDeviceMgr are valid object references.
3.1.3.2.3.6.7.3	SR:311	CF	The registerService operation shall verify that the input registeredDeviceMgr has been previously registered with the DomainManager.
3.1.3.2.3.6.7.3	SR:312	CF	The registerService operation shall add the registeringService's object reference and the registeringService's, name to the DomainManager, if the name for the type of service being registered does not exist within the DomainManager.
3.1.3.2.3.6.7.3	SR:313	CF	However, if the name of the registering service is a duplicate of a registered service of the same type, then the new service shall not be registered with the DomainManager.
3.1.3.2.3.6.7.3	SR:314	CF	The registerService operation shall associate the input registeringService parameter with the input registeredDeviceMgr parameter in the DomainManager's, when the registeredDeviceMgr parameter indicates a DeviceManager registered with the DomainManager.
3.1.3.2.3.6.7.3	SR:315	CF	The registerService operation shall, upon successful service registration, establish any pending connection requests for the registeringService.
3.1.3.2.3.6.7.3	SR:316	CF	The registerService operation shall, upon successful service registration, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log.
3.1.3.2.3.6.7.3	SR:317	CF	The registerService operation shall, upon unsuccessful service registration, write a `FAILURE_ALARM` log record to a DomainManager's log.
3.1.3.2.3.6.7.3	SR:318	CF	The registerService operation shall, upon successful service registration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType. The event data will be populated as follows:
3.1.3.2.3.6.7.3	SR:319	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.7.3	SR:320	CF	The sourceId shall be the identifier attribute from the componentinstantiation element associated with the registered service.

3.1.3.2.3.6.7.3	SR:321	CF	The sourceName shall be the input name parameter for the registering service.
3.1.3.2.3.6.7.3	SR:322	CF	The sourceIOR shall be the registered service object reference.
3.1.3.2.3.6.7.3	SR:323	CF	The sourceCategory shall be `SERVICE`.
3.1.3.2.3.6.7.5	SR:324	CF	The registerService operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr parameter is not a nil reference and is not registered with the DomainManager.
3.1.3.2.3.6.7.5	SR:325	CF	The registerService operation shall raise the CF InvalidObjectReference exception when input parameters registeringService or registeredDeviceMgr contains an invalid reference.
3.1.3.2.3.6.7.5	SR:326	CF	The registerService operation shall raise the RegisterError exception when an internal error exists which causes an unsuccessful registration.

This operation registers a service hosted by a DeviceManager with the DomainManager. The first action performed is to verify the registeringService and registeredDeviceMMgr object references (SR:310). If the parameters do not refer to a valid service or DeviceManager, respectively, then an InvalidObjectReference exception is raised (SR:325) and the operation is terminated.

If the object references are valid, then the registeredDeviceMgr value is checked against the deviceManagers attribute sequence to validate that the DeviceManager has previously registered with the DomainManager (SR:311). If the DeviceManager referenced has not already registered with the DomainManager, then a DeviceManagerNotRegistered exception is raised (SR:324) and the operation is terminated.

Once both the registeringService and registeredDeviceMgr object references have been validated, the service name specified is checked against the set of services in the DomainManager. If the service name specified is not found then the object reference and name of the service is added to the DomainManager (SR:312). If the name provided is found within the DomainManager, then the service is not added (SR:313) and the operation terminates without error.

The service added to the DomainManager is associated with the DeviceManager (SR:314) so that the proper cleanup can be performed for the unregisterDeviceManager operation.

Any pending requests for connections to the newly registered service are performed (SR:315).

An `ADMINISTRATIVE_EVENT` log record is then written to a DomainManager log indicating the successful registration of the service (SR:316) and an event is sent to the ODM event channel indicating that a service has registered with the DomainManager (SR:318). The producerId is set to the identifier attribute of the DomainManager (SR:319), the sourceId set to the identifier attribute of the componentinstantiation element in the XML file associated with the registered service (SR:320), the sourceName set to the input string name parameter provided on the registerService call (SR:321), the sourceIOR set to the registered service object reference (SR:322), and the sourceCategory set to `SERVICE` (SR:323).

If the registration is unsuccessful, for any reason, a `FAILURE_ALARM` record is written to a DomainManager log (SR:317).

If any other error is encountered during the registerService operation that prevents successful completion of the operation, then the RegisterError exception is raised (SR:326) and execution terminates.

If this or any other error is encountered, it is expected that the exception handler will return the DomainManager to a stable state prior to the initiation of the operation.

unregisterService

The unregisterService makes the service unavailable for new requests, disconnects the service from any client components, and removes a service from the DomainManager. The parameters provided consist of the object reference to the service and the name of the service provided as a string value.

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

Two possible exceptions may be raised by the unregisterService operation. It may raise an InvalidObjectReference exception if the object reference provided is nil or not a valid service reference or an UnregisterError exception if any other error is encountered during the unregisterService execution. The requirements are listed in Table 6.10.

Table 6.10. unregisterService requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.8.3	SR:327	CF	The unregisterService operation shall remove the unregisteringService entry specified by the input name parameter from the DomainManager.
3.1.3.2.3.6.8.3	SR:328	CF	The unregisterService operation shall release (client-side CORBA release) the unregisteringService from the DomainManager.
3.1.3.2.3.6.8.3	SR:329	CF	The unregisterService operation shall, upon the successful unregistration of a Service, write an `ADMINISTRATIVE_EVENT` log record to a DomainManager's log.
3.1.3.2.3.6.8.3	SR:330	CF	The unregisterService operation shall, upon unsuccessful unregistration of a Service, write a `FAILURE_ALARM` log record to a DomainManager's log.
3.1.3.2.3.6.8.3	SR:331	CF	The unregisterService operation shall, upon successful service unregistration, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType. The event data will be populated as follows:

3.1.3.2.3.6.8.3	SR:332	CF	The producerId shall be the identifier attribute of the DomainManager.
3.1.3.2.3.6.8.3	SR:333	CF	The sourceId shall be the Id attribute from the componentinstantiation element associated with the unregistered service.
3.1.3.2.3.6.8.3	SR:334	CF	The sourceName shall be the input name parameter for the unregistering service.
3.1.3.2.3.6.8.3	SR:335	CF	The sourceCategory shall be `SERVICE`.
3.1.3.2.3.6.8.5	SR:336	CF	The unregisterService operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a Service interface.
3.1.3.2.3.6.8.5	SR:337	CF	The unregisterService operation shall raise the UnregisterError exception when an internal error exists which causes an unsuccessful unregistration.

The operation first validates the object reference provided as part of the call. If it is nil or not a valid reference to a service, then the InvalidObjectReference is raised (SR:336) and execution terminates. If the object reference refers to a valid, registered service then the service name is located within the set of registered services within the DomainManager. If the service identified is not found, then execution terminates without error.[2]

If the service is found, it is removed from the DomainManager (SR:327). The service is then released from the CORBA environment by the DomainManager (SR:328).

An `ADMINISTRATIVE_EVENT` log record is then written to a DomainManager log indicating the successful unregistration of the service (SR:329), and an event is sent to the ODM event channel indicating that a service has unregistered with the DomainManager (SR:331). The producerId is set to the identifier attribute of the DomainManager (SR:332), the sourceId set to the identifier attribute of the componentinstantiation element in the XML file associated with the registered service (SR:333), the sourceName set to the input string name parameter provided on the unregisterService call (SR:334), and the sourceCategory set to `SERVICE` (SR:335).

If any other error is encountered during the registerService operation that prevents successful completion of the operation, then a `FAILURE_ALARM` record is written to a DomainManager log (SR:330) and the UnregisterError exception is raised (SR:337) and execution terminates.

If this or any other error is encountered, it is expected that the exception handler will return the DomainManager to a stable state prior to the initiation of the operation.

registerWithEventChannel

The registerWithEventChannel operation connects a consumer to an event channel. The input parameters are an object reference for the object to be connected to the event channel,

[2] *There is no requirement specifying the above behavior. The assumption is that, if an unregistering service is not found, then it has already been unregistered or never registered. In either case, no further action is required.*

a string value specifying the registering Id, and a string value specifying the name of the event channel.

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

Exceptions that may be raised include InvalidObjectReference if the object reference provided is nil or does not resolve to a valid instance, InvalidEventChannelName if the string specifying the event channel is not defined by the event service, or AlreadyConnected if the object requesting connection is already connected to the channel specified. The requirements are listed in Table 6.11.

Table 6.11. registerWithEventChannel requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.9.3	SR:338	CF	The registerWithEventChannel operation shall connect the input registeringObject to an event channel as specified by the input eventChannelName.
3.1.3.2.3.6.9.5	SR:339	CF	The registerWithEventChannel operation shall raise the CF InvalidObjectReference exception when the input registeringObject parameter contains an invalid reference to a CosEventComm PushConsumer interface.
3.1.3.2.3.6.9.5	SR:340	CF	The registerWithEventChannel operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter contains an invalid event channel name (e.g. `'ODM_Channel'`).
3.1.3.2.3.6.9.5	SR:341	CF	The registerWithEventChannel operation shall raise the AlreadyConnected exception when the input parameter contains a connection to the event channel for the input registeringId parameter.

This operation connects a consumer to an event channel (SR:338). The object reference is validated. If the object reference in nil or does not resolve to a valid object, then the InvalidObjectReference exception is raised (SR:339).

If the event channel name specified does not specify a valid event channel, then the InvalidEventChannelName exception is raised (SR:340).

If the object requesting the connection is already connected to the named event channel, then the AlreadyConnected exception is raised (SR:341).

unregisterWithEventChannel

The unregisterWithEventChannel disconnects a consumer from an event channel. Two parameters are provided: a string identifying the Id to unregister and a string identifying the channel name to disconnect.

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

Two exceptions may be raised by this operation: the InvalidEventChannelName exception if the channel name specified by the input parameter is not recognized and the NotConnected exception if the consumer requesting the disconnect is not currently connected to the event channel. The requirements are listed in Table 6.12.

Table 6.12. unregisterWithEventChannel requirements

Section	ID	Resp	Requirement
3.1.3.2.3.6.10.3	SR:342	CF	The unregisterFromEventChannel operation shall disconnect a registered component from the event channel as identified by the input parameters.
3.1.3.2.3.6.10.5	SR:343	CF	The unregisterFromEventChannel operation shall raise the InvalidEventChannelName exception when the input eventChannelName parameter contains an invalid reference to an event channel (e.g. ‘`ODM_Channel`’).
3.1.3.2.3.6.10.5	SR:344	CF	The unregisterFromEventChannel operation shall raise the NotConnected exception when the input parameter unregisteringId parameter is not connected to specified input event channel.

The operation checks that the event channel name provided is a valid event channel. If not, the InvalidEventChannelName is raised (SR:343).

If the event channel name is valid but the consumer is not currently connected to the event channel, then the NotConnected exception is raised (SR:344).

If the event channel name is valid and the consumer is currently connected, then the consumer is disconnected from the event channel (SR:342).

6.2 FileManager

The FileManager, implemented by the DomainManager, provides mechanism for managing multiple file systems across multiple hardware as a single entity (See Figure 6.2). Also, because it inherits the FileSystem interface, it is also a FileSystem. Thus, the File Manager can be thought of as the top-level or initial FileSystem for the SCA Core Framework. The FileManager requirements one listed in Table 6.13.

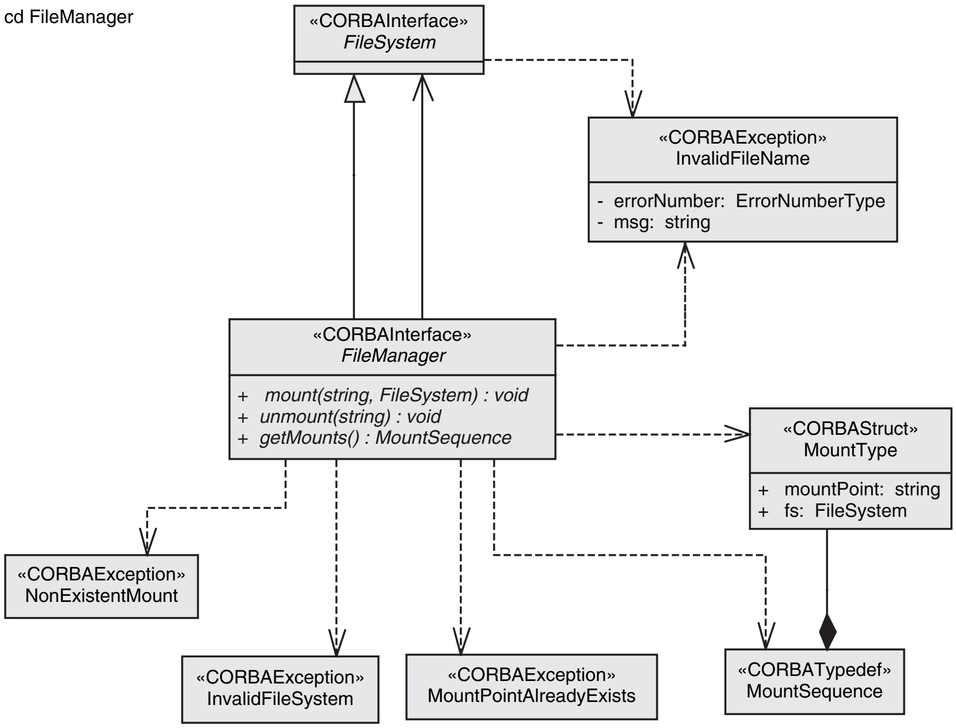

cd FileManager
«CORBAInterface» FileSystem
«CORBAException» InvalidFileName
- errorNumber: ErrorNumberType
- msg: string
«CORBAInterface» FileManager
+ mount(string, FileSystem) : void
+ unmount(string) : void
+ getMounts() : MountSequence
«CORBAStruct» MountType
+ mountPoint: string
+ fs: FileSystem
«CORBAException» NonExistentMount
«CORBAException» InvalidFileSystem
«CORBAException» MountPointAlreadyExists
«CORBATypedef» MountSequence

Figure 6.2. FileManager interfaces

The primary function of the FileManager of the DomainManager is to provide a distributed file system that functions as a single file system across the FileSystem instances hosted by multiple DeviceManagers (SR:581). In addition, the FileManager inherits the FileSystem interface thereby providing access to the multiple FileSystems as a single, federated FileSystem (SR:582).[3]

The FileManager delegates the actual FileSystem operations to the FileSystem hosted by the respective DeviceManager for that FileSystem. For example, if a DeviceManager has hosted a FileSystem that is mounted within the FileManager as /DevMgr1, then any

[3] *Since the FileManager inherits the FileSystem interface, all FileSystem operations may be called on the FileManager. Only the query operation is explicitly referenced in the requirements as it provides additional behavior beyond the standard FileSystem query.*

Table 6.13. FileManager requirements

Section	ID	Resp	Requirement
3.1.3.3.3.5.4	SR:581	CF	The FileManager interface shall support a federated, or distributed, file system that may span multiple FileSystem components.
3.1.3.3.3.5.4	SR:582	CF	The FileManager's inherited FileSystem operations behavior shall implement the requirements of the FileSystem operations against the mounted file systems.
3.1.3.3.3.5.4	SR:583	CF	The FileManager's FileSystem operations shall remove the FileSystem mounted name from the input fileName before passing the fileName to an operation on a mounted FileSystem.
3.1.3.3.3.5.4	SR:584	CF	The FileManager shall use the mounted FileSystem for FileSystem operations based upon the mounted FileSystem name that exactly matches the input fileName to the lowest matching subdirectory.
3.1.3.3.3.5.5.3	SR:585	CF	The query operation shall return the combined mounted file systems information to the calling client based upon the given input fileSystemProperties' Ids.
3.1.3.3.3.5.5.3	SR:586	CF	As a minimum, the query operation shall support the following input fileSystemProperties Ids: 1. `SIZE` – a property item Id value of '`SIZE`' will cause the query operation to return the combined total size of all the mounted file systems as an unsigned long long property value. 2. `AVAILABLE_SPACE` – a property item Id value of `'AVAILABLE_SPACE'` will cause the query operation to return the combined total available space (in octets) of all mounted file systems as an unsigned long long property value.
3.1.3.3.3.5.5.3	SR:587	CF	The query operation shall raise the UnknownFileSystemProperties exception when the input fileSystemProperties parameter contains an invalid property Id.

FileSystem operation received by the FileManager would be delegated to the FileSystem instance for the DeviceManager that hosted the FileSystem for /DevMgr1. When delegating the FileSystem operation to the DeviceManager's FileSystem, the mount name is removed (SR:583). Thus, a FileSystem operation on the directory /DevMgr1/SCAstuff would have the /DevMgr1 removed and only the /SCAstuff would be forwarded to the FileSystem on that DeviceManager.

Any operation is matched to the lowest matching subdirectory prior to delegation to the FileSystem on the DeviceManager (SR:584).

Any query operation on the FileSystem returns the combined mounted file systems information (SR:585). Two properties are supported in this operation: `SIZE` and `AVAILABLE_SPACE`. The `SIZE` property name refers to the current size (in octets) used by all FileSystems in the FileManager. The `AVAILABLE_SPACE` refers to the space available for storage across all FileSystems in the FileManager (SR:586). If an unknown property name is provided to the query operation, then the UnknownFileSystemProperties exception is raised (SR:587).

6.2.1 Types

MountType

The MountType defines a data structure that maps the string name of the mount point to the FileSystem instance hosted by the DeviceManager (See Table 6.14).

Table 6.14. FileManager MountType requirements

Section	ID	Resp	Requirement
3.1.3.3.3.3.1	SR:570	CF	The MountType structure shall identify the FileSystems mounted within the FileManager.

The MountType defines the data structure that maintains the set of FileSystems mounted within a FileManager (SR:570).

```
struct MountType {
     string mountPoint;
     FileSystem fs;
};
```

MountSequence

The MountSequence is a sequence of MountTypes that is used by the FileManager to maintain the set of FileSystems mounted within the federated file system.

```
typedef sequence <MountType> MountSequence;
```

6.2.2 Exceptions

NonExistentMount

This exception is raised when the mount point specified in a function call does not exist.

```
exception NonExistentMount {
};
```

InvalidFileSystem

This exception is raised when a FileSystem reference does not refer to a valid FileSystem.

```
exception InvalidFileSystem {
};
```

MountPointAlreadyExists

This exception is raised when a mount operation is called for a FileSystem that has already been mounted to the named mount point.

```
exception MountPointAlreadyExists {
};
```

6.2.3 Operations

mount

The mount operation mounts a FileSystem by creating an association within the FileManager between the string specifying the mount point and the FileSystem instance provided (See Table 6.15).

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

Exceptions that may be raised include InvalidFileName if the mount point specified does not adhere to the file naming conventions, InvalidFileSystem if the object reference provided

Table 6.15. FileManager mount requirements

Section	ID	Resp	Requirement
3.1.3.3.3.5.1.3	SR:573	CF	The mount operation shall associate the specified FileSystem with the given mountPoint.
3.1.3.3.3.5.1.3	SR:574	CF	A mountPoint name shall begin with a '/'. A mountPoint name is a logical directory name for a FileSystem.
3.1.3.3.3.5.1.5	SR:575	CF	The mount operation shall raise the InvalidFileName exception when the input file name is invalid.
3.1.3.3.3.5.1.5	SR:576	CF	The mount operation shall raise the MountPointAlreadyExists exception when the mountPoint already exists in the file manager.
3.1.3.3.3.5.1.5	SR:577	CF	The mount operation shall raise the InvalidFileSystem exception when the input FileSystem is a null object reference.

does not resolve to a valid FileSystem, and MountPointAlreadyExists if the mount point specified has already been used to mount a FileSystem.

The primary function of the mount operation is to associate a FileSystem with a string name representing the mount point (SR:573). If the FileSystem reference is a null object reference, then the InvalidFileSystem exception is raised and the operation terminates (SR:577).

The mount point specified must begin with a slash,'/' (SR:574). If the input name of the mount point is invalid, then the InvalidFileName exception is raised (SR:575) and the operation terminates. If the mount point name specified has already been used then the MountPointAlreadyExists exception is raised (SR:576) and the operation terminates.

unmount

The unmount operation removes (unmounts) a mounted FileSystem from the FileManager (see Table 6.16). The input parameter consists of the mount point name to be unmounted.

```
void unmount (
    in string mountPoint
    )
raises (NonExistentMount);
```

If the mount point specified does not exist then a NonExistentMount exception is raised.

Table 6.16. FileManager unmount requirements

Section	ID	Resp	Requirement
3.1.3.3.3.5.2.3	SR:578	CF	The unmount operation shall remove a mounted FileSystem from the FileManager whose mounted name matches the input mountPoint name.
3.1.3.3.3.5.2.5	SR:579	CF	The unmount operation shall raise the NonExistentMount exception when the mountPoint does not exist.

The unmount operation checks that the mount point provided is a valid mount point within the FileManager. If not, a NonExistentMount exception is raised (SR:579) and the operation terminates. If the mount point provided is valid, then the FileSystem associated with the mount point name is removed from the FileManager (SR:578) and is no longer available as part of the federated file system. It is, however, still available to the DeviceManager hosting the FileSystem.

getMounts

The getMounts call returns the current set of mount points in the FileManager (see Table 6.17). No input parameters are provided.

```
MountSequence getMounts ();
```

The operation does not raise any exceptions.

Table 6.17. FileManager getMounts requirements

Section	ID	Resp	Requirement
3.1.3.3.3.5.3.3	SR:580	CF	The getMounts operation shall return a sequence of mount structures that describe the mounted FileSystems.

The only function performed by this operation is to return a sequence of FileSystem mounts (SR:580). If no FileSystems have been mounted then a null sequence is returned.

6.3 The ApplicationFactory

Even though the ApplicationFactory has only a single operation, it is, in some respect, the most complex component of an SCA Core Framework (see Figure 6.3). The ApplicationFactory is responsible for taking the information in the waveform SAD, identifying the resources required, dependencies, and component connections, matching those against the available hardware components and physical resources available, loading the waveform components, establishing the connections, and resulting in an instantiated waveform application.

Furthermore, if some error is encountered during the waveform instantiation process, the ApplicationFactory must release any resource allocations, unload any software components loaded up to that point, and return the radio system to the stable state prior to the request to create an application instance.

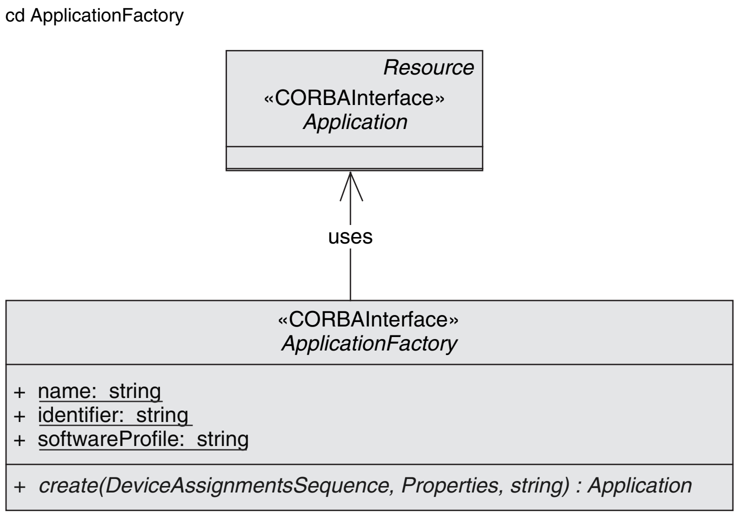


Figure 6.3. ApplicationFactory interface

6.3.1 Exceptions

CreateApplicationRequestError

This exception is raised when one or more invalid component assignments to devices are encountered.

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

CreateApplicationError

This exception is raised when an internal error is encountered that prevents the create operation from completing successfully (see Table 6.18).

Table 6.18. ApplicationFactory CreateApplicationError requirements

Section	ID	Resp	Requirement
3.1.3.2.2.3.2	SR:152	CF	The error number shall indicate an ErrorNumberType value (e.g. E2BIG, ENAMETOOLONG, ENFILE, ENODEV, ENOENT, ENOEXEC, ENOMEM, ENOTDIR, ENXIO, EPERM).

The errorNumber value (SR:152) in the exception provides an indication of the type of error encountered. The msg parameter provides additional detail regarding the error encountered.

```
exception CreateApplicationError {
    ErrorNumberType errorNumber;
    string msg;
};
```

InvalidInitConfiguration

If an invalid initConfiguration parameter is encountered, then the InvalidInitConfiguration is raised. An invalid initConfiguration refers to an error on one or more of the properties specified as part of the initialization process for the Application component.

```
exception InvalidInitConfiguration {
     Properties invalidProperties;
};
```

6.3.2 Attributes

The ApplicationFactory Attribute requirements are listed in Table 6.19.

Table 6.19. ApplicationFactory Attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.2.4.1	SR:153	CF	The readonly name attribute shall contain the type of Application that can be instantiated by the ApplicationFactory.
3.1.3.2.2.4.2	SR:154	CF	The readonly softwareProfile attribute shall contain either a profile element with a file reference to the SAD profile or the XML for the SAD profile.
3.1.3.2.2.4.3	SR:155	CF	The readonly identifier attribute shall contain the unique identifier for an ApplicationFactory instance.
3.1.3.2.2.4.3	SR:156	CF	The identifier shall be identical to the softwareassembly element Id attribute of the ApplicationFactory's Software Assembly Descriptor file.

name

The readonly name attribute is a user-readable name for the Application that may be instantiated by the ApplicationFactory, e.g. SINCGARS, Havequick, etc. (SR:153).

```
readonly attribute string name;
```

identifier

The readonly identifier attribute provides a unique identifier for the Application Factory (SR:155). The value is specified by the Id attribute of the softwareassembly element in the ApplicationFactory's Software Assembly Descriptor (SAD) file (SR:156).[4]

```
readonly attribute string identifier;
```

softwareProfile

This readonly attribute contains the reference to the SAD file that specifies the domain profile information associated with the waveform application (SR:154).

```
readonly attribute string softwareProfile;
```

6.3.3 Operations

create

The create operation instructs the ApplicationFactory instance to take the information regarding the waveform components, dependencies, and connections defined by the

[4] *This is the XML file specified on the installApplication operation.*

associated SAD file that was processed when the application was installed (see installApplication) on page 164 instantiating the associated ApplicationFactory and create an instance of the waveform (See Table 6.20).

Table 6.20. ApplicationFactory Create operations requirements

Section	ID	Resp	Requirement
3.1.3.2.2.5.1.3	SR:157	CF	The create operation shall use the SAD SPD implementation element to locate candidate devices capable of loading and executing Application components.
3.1.3.2.2.5.1.3	SR:158	CF	The create operation shall allocate (Device allocateCapacity) component capacity requirements against candidate devices to determine which candidate devices satisfy all SPD implementation criteria requirements and SAD partitioning requirements (e.g. components host co-location, etc.)
3.1.3.2.2.5.1.3	SR:159	CF	The create operation shall only use Devices that have been granted successful capacity allocations for loading and executing Application components, or used for data processing.
3.1.3.2.2.5.1.3	SR:160	CF	The create operation shall load the Application components (including all of the Application-dependent components) to the chosen device(s).
3.1.3.2.2.5.1.3	SR:161	CF	The create operation shall execute the application components (including all of the application-dependent components) as specified in the application's SAD file.
3.1.3.2.2.5.1.3	SR:162	CF	The create operation shall use each component's SPD implementation code's stack size and priority elements, when specified, for the execute options parameters.
3.1.3.2.2.5.1.3	SR:163	CF	The create operation shall pass the mandatory execute parameters of a Naming Context IOR, Name Binding, and the identifier for the component in the form of CF Properties to the entry points of Resource components to be executed via a Device's execute operation.
3.1.3.2.2.5.1.3	SR:164	CF	The execute parameter for the Naming Context IOR shall be inserted into a CF Properties type.
3.1.3.2.2.5.1.3	SR:165	CF	The CF Properties Id element shall be set to `'NAMING_CONTEXT_IOR'` and the CF Properties value element set to the stringified IOR of a naming context to which the component will bind.

3.1.3.2.2.5.1.3	SR:166	CF	The create operation shall create any naming contexts that do not exist, to which the component will bind the Naming Context IOR.
3.1.3.2.2.5.1.3	SR:167	CF	The structure of the naming context path shall be '/ DomainName / [optional naming context sequences]'.
3.1.3.2.2.5.1.3	SR:168	CF	The execute parameter of Name Binding shall be inserted into a CF Properties type.
3.1.3.2.2.5.1.3	SR:169	CF	The CF Properties Id element shall be set to '`NAME_BINDING`' and CF Properties value element set to a string in the format of '`ComponentName_UniqueIdentifier`'.
3.1.3.2.2.5.1.3	SR:170	CF	For the component identifier execute parameter, the create operation shall be inserted in a CF Properties type.
3.1.3.2.2.5.1.3	SR:171	CF	The CF Properties Id element shall be set to '`COMPONENT_IDENTIFIER`' and the CF Properties value element to the string format of Component_Instantiation_Identifier: Application_Name.
3.1.3.2.2.5.1.3	SR:172	CF	The Application_Name field shall be identical to the create operation's input name parameter.
3.1.3.2.2.5.1.3	SR:173	CF	The create operation shall pass the componentinstantiation element 'execparam' properties that have values as parameters to execute operation.
3.1.3.2.2.5.1.3	SR:174	CF	The create operation shall, in order, initialize Resources, then establish connections for Resources, and finally configure the Resources.
3.1.3.2.2.5.1.3	SR:175	CF	The create operation shall initialize an Application component provided that the component implements the LifeCycle interface.
3.1.3.2.2.5.1.3	SR:176	CF	The create operation shall configure an application's assemblycontroller component provided the assemblycontroller has configured readwrite or writeonly properties with values.
3.1.3.2.2.5.1.3	SR:177	CF	The create operation shall use the union of the input initConfiguration properties of the create operation and the assemblycontroller's componentinstantiation writeable 'configure' properties that have values.
3.1.3.2.2.5.1.3	SR:178	CF	The input initConfiguration parameter shall have precedence over the assemblycontroller's writeable 'configure' property values.
3.1.3.2.2.5.1.3	SR:179	CF	The create operation, when creating a component from a ResourceFactory, shall pass the componentinstantiation componentresoursefactoryref element 'factoryparam' properties that have values as qualifiers parameters to the referenced ResourceFactory component.

Table 6.20. Continued

Section	ID	Resp	Requirement
3.1.3.2.2.5.1.3	SR:180	CF	The create operation shall pass, with invocation of each ResourceFactory createResource operation, the ResourceFactory configuration properties associated with that Resource as dictated by the SAD.
3.1.3.2.2.5.1.3	SR:181	CF	For connections established for a log, the create operation shall create a unique producer log Id for each log producer.
3.1.3.2.2.5.1.3	SR:182	CF	The create operation shall invoke the PropertySet configure operation once, and only once, per log producer (as described by the SAD usesport element) in order to set its unique `PRODUCER_LOG_ID` (see specifications section 3.1.3.3.5.5.1.2 for details).
3.1.3.2.2.5.1.3	SR:183	CF	For connections established for a CORBA Event Service's event channel, the create operation shall connect a COSEventComm PushConsumer or PushSupplier object to the event channel as specified in the SAD's domainfinder element.
3.1.3.2.2.5.1.3	SR:184	CF	If the event channel does not exist, the create operation shall create the event channel.
3.1.3.2.2.5.1.3	SR:185	CF	If the Application is successfully created, the create operation shall return an Application component reference for the created Application.
3.1.3.2.2.5.1.3	SR:186	CF	The create operation shall, upon successful Application creation, write an `ADMINISTRATIVE_EVENT` log record.
3.1.3.2.2.5.1.3	SR:187	CF	The create operation shall, upon unsuccessful Application creation, write a `FAILURE_ALARM` log record.
3.1.3.2.2.5.1.3	SR:188	CF	For connections established for a log, the create operation shall create a unique producer log Id one time for each log producer.
3.1.3.2.2.5.1.3	SR:189	CF	The create operation shall invoke the PropertySet configure operation one time per log producer (as described by the SAD usesport element) in order to set its unique `PRODUCER_LOG_ID` (see specifications section 3.1.2.3.1 for details).
3.1.3.2.2.5.1.3	SR:190	CF	The create operation shall, upon successful Application creation, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectAddedEventType.
3.1.3.2.2.5.1.3	SR:191	CF	The producerId shall be the identifier attribute of the ApplicationFactory.
3.1.3.2.2.5.1.3	SR:192	CF	The sourceId shall be the identifier attribute of the created Application.
3.1.3.2.2.5.1.3	SR:193	CF	The sourceName shall be the name attribute of the created Application.

3.1.3.2.2.5.1.3	SR:194	CF	The sourceIOR shall be the Application component reference for the created Application.
3.1.3.2.2.5.1.5	SR:195	CF	The sourceCategory shall be `APPLICATION`.
3.1.3.2.2.5.1.5	SR:196	CF	The create operation shall raise the CreateApplicationRequestError exception when the parameter CF DeviceAssignmentSequence contains one or more invalid Application component to device assignment(s).
3.1.3.2.2.5.1.5	SR:197	CF	The create operation shall raise the CreateApplicationError exception when the create request is valid but the Application cannot be successfully instantiated due to internal processing error(s).
3.1.3.2.2.5.1.5	SR:198	CF	The create operation shall raise the InvalidInitConfiguration exception when the input initConfiguration parameter is invalid.
3.1.3.2.2.5.1.5	SR:199	CF	The InvalidInitConfiguration invalidProperties shall identify the property that is invalid.

The input parameters are a string name of the Application instance, a set of Properties specifying initial configuration parameters, and a DeviceAssignmentSequence that specifies specific device assignments for the waveform.

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

Exceptions that may be raised include CreateApplicationError if an error is encountered during the execution of the create operation, CreateApplicationRequestError if the DeviceAssignmentSequence contains invalid Device assignment requests for the waveform, and InvalidInitConfiguration if the set of Properties specifying the initial configuration contains invalid values or properties.

The create operation instantiates an Application, as defined by the domain profile information specified when the ApplicationFactory was instantiated. The ApplicationFactory may be directed to use specific devices on which to instantiate the Application or it may use requirements specified in the domain profile to identify the devices required for hosting the Application.

If specific devices are specified by the deviceAssignments parameter, then the ApplicationFactor loads the components to the specified devices (SR:160). If the domain profile information is used, then the ApplicationFactory identifies candidate devices that are capable of loading and/or executing the software components as defined in the Software Package Descriptor (SPD) associated with the component of the SAD file (SR:157). If the device specified in the deviceAssignments parameter does not represent the full set of components within the SAD file, then components specified within the deviceAssignments parameter are loaded to the specified devices and the remainder of the components are allocated based on the domain profile information.

Whether specified by the deviceAssignments parameter or by using the domain profile information, the create operation will use the allocateCapacity method on the device to verify if a particular device has the capacity available to satisfy the need of the component (SR:158). As each device is allocated, the ApplicationFactory maintains a list of devices that have been allocated (SR:159).

Once the devices have been allocated, the create operation loads the components on the selected devices (SR:160), if the component has not already been loaded on the device. This involves accessing the image to be loaded, e.g. library, executable, FGPA bit image, using the file as specified by the implementation within the SPD. For those components that are executable programs, the create operation executes the components as specified in the SAD file (SR:161).[5]

The `STACK_SIZE` and `PRIORITY` elements, if specified in the SPD for the component, are applied to the execution of the component (SR:162).

The component's Resource or ResourceFactory reference, as specified in the SAD file, is then obtained from the domain profile information and the Resource defined for the component is instantiated. If a ResourceFactory is defined, then the Resources defined by the ResourceFactory are instantiated through the ResourceFactory using the Resource interface.

The create operation passes three mandatory execute parameters to the entry points of Resource components using the execute operation. These are the Naming ContextIOR, the Name Binding, and the identifier for the component as Properties (SR:163). The NamingContextIOR is inserted into a Properties data type (SR:164) with the Id element set to '`NAMING_CONTEXT_IOR`' (SR:165) and the value set to the stringified IOR to which the component will bind. If the naming context does not exist in the Naming Service, then the create operation creates the naming context in the Naming Service (SR:166). The naming context follows the pattern of '`/<DomainName>/<optional naming contexts>`' (SR:167) where the <DomainName> is the overall domain name defined when the DomainManager was instantiated. The Name Binding parameter is also inserted into a Properties datatype (SR:168) with the Id element of the property set to '`NAME_BINDING`' and the value is a string in the form of '`<ComponentName>_<UniqueIdentifier>`' (SR:169) where the unique identifier is determined by the implementation of the ApplicationFactory. Finally, the component identifier is inserted into a Properties data type (SR:170). The Id element is set to '`COMPONENT_IDENTIFIER`' and the value is a string in the format, '`Component_Instantiation_Identifier:<Application_Name>`' (SR:171). <Application_Name> is set to the value of the name parameter provided with the create operation (SR:172). Additionally, any 'execparam' properties specified by the

[5] *For an ExecutableDevice, e.g. a GPP hosting an operating system, the executable is typically 'loaded' via a 'fork and exec' process (it may vary slightly from one operating system to the next). The user is referred to the issues mentioned in the ExecutableDevice section regarding the need for an ExecutableDevice hosting an operating system to provide a file descriptor (FD) mapped to the operating system's native file system.*

componentinstantiation element in the domain profile XML files are passed to the component by the create operation (SR:173).

Resources instantiated by the create operations are initialized, connections are established, and the Resources are configured (SR:174). Application components are instantiated by the create operation if the component implements the LifeCycle interface (SR:175).[6]

The create operation also configures the AssemblyController for the Application instance (SR:176) using the union of the initConfiguration properties of the create operation and the assemblycontroller's componentinstantiation writeable configure properties that have values (SR:177). In the event that the same property is specified in both the initConfiguration properties on the create call and in the componentinstantitation, the initConfiguration properties take precedence, i.e. the initConfiguration properties will be used instead of the componentinstantiation properties (SR:178).

When a component is created using a ResourceFactory, the create operation passes the componentinstantiation componentresourcefactoryref element factoryparam properties as a parameter to the ResourceFactory component (SR:179). For each createResource call on a ResourceFactory, the configuration properties associated with the Resource as specified in the SAD is passed to the Resource (SR:180).

Connection is established to a log as part of the create operation, and a unique producer log Id is created for each log producer (SR:181). As the unique Id is created for the LogProducer, the create operation also calls the configure operation once to set the `PRODUCER_LOG_ID` (SR:182).

If Event Service connections are specified in the SAD, the create operation connects a PushConsumer or PushSupplier to the event channel specified in the SAD (SR:183). If the channel specified does not exist, the create operation creates the event channel (SR:184).

If the Application is successfully instantiated, the create operation writes an `ADMINISTRATIVE_EVENT` log record (SR:186) identifying the Application created. It also sends an event to the Outgoing Domain Management (ODM) event channel consisting of the DomainManagementObjectAddedEventType (SR:190). The producerID is the identifier attribute of the ApplicationFactory (SR:191), the sourceId is the identified attribute of the created Application (SR:192), the sourceName is the name attribute of the created Application (SR:193), the sourceIOR is the Application component reference of the created Application (SR:194), and the sourceCategory is set to `APPLICATION` (SR:195). The create operation then returns an Application component reference for the newly instantiated Application (SR:185).[7]

If there are invalid Device assignments specified in the DeviceAssignmentSequence, then the CreateApplicationRequestError exception is raised (SR:196). If the input initConfiguration parameter is invalid, the InvalidInitConfiguration exception is raised (SR:198). If one or more properties are invalid, the InvalidProperties of the InvalidInitConfiguration exception identifies the properties (SR:199) If the Application instantiation fails due to some other internal processing error, the CreateApplicationError exception is raised (SR:197).

[6] *Since the LifeCycle interface is inherited by the Resource and the application components are typically defined as Resources, the interface will be available. However, the component may or may not implement the behavior for the function call. Thus, the call is made but the initialize call may be simply a stub.*

[7] *The Application component reference is the reference used to access the Application by subsequent function calls. The reference is added to the list of Applications managed by the DomainManager.*

If the Application is not successfully instantiated, the create operation writes a `FAILURE_ALARM` log record identifying the Application (SR:187).

For log connections that are established, the create operation generates a unique producer log Id (SR:188) and invokes the PropertySet configure operation (SR:189) once per log producer.

6.4 Application

The Application interface specifies the top-level interface for the waveform (see Figure 6.4). It is interesting to note that the Application interface does not add additional behavior. All the basic interface functions are inherited from Resource. The Application interface does define some additional attributes used in the management of the Application.

6.4.1 Types

ComponentProcessIdType

The ComponentProcessIdType provides a structure used to associate a component with its process Id. This type can be used to retrieve a process Id for a specific component.[8] The componentId is the value specified by the Id attribute value of the componentinstantiation in the XML file.

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

ComponentProcessIdSequence

This type is defined to manage the set of processes that map to managed components of an application. This is an unbounded sequence of ComponentProcessIdType entries.

```
typedef sequence <ComponentProcessIdType>
```

ComponentElementType

A component is associated with an implementation using the ComponentElementType. The componentId of the structure is the componentinstantiation Id attribute value in the SAD file.

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

[8] *This processID applies to an executable component only; that is, a component that is loaded on an ExecutableDevice.*

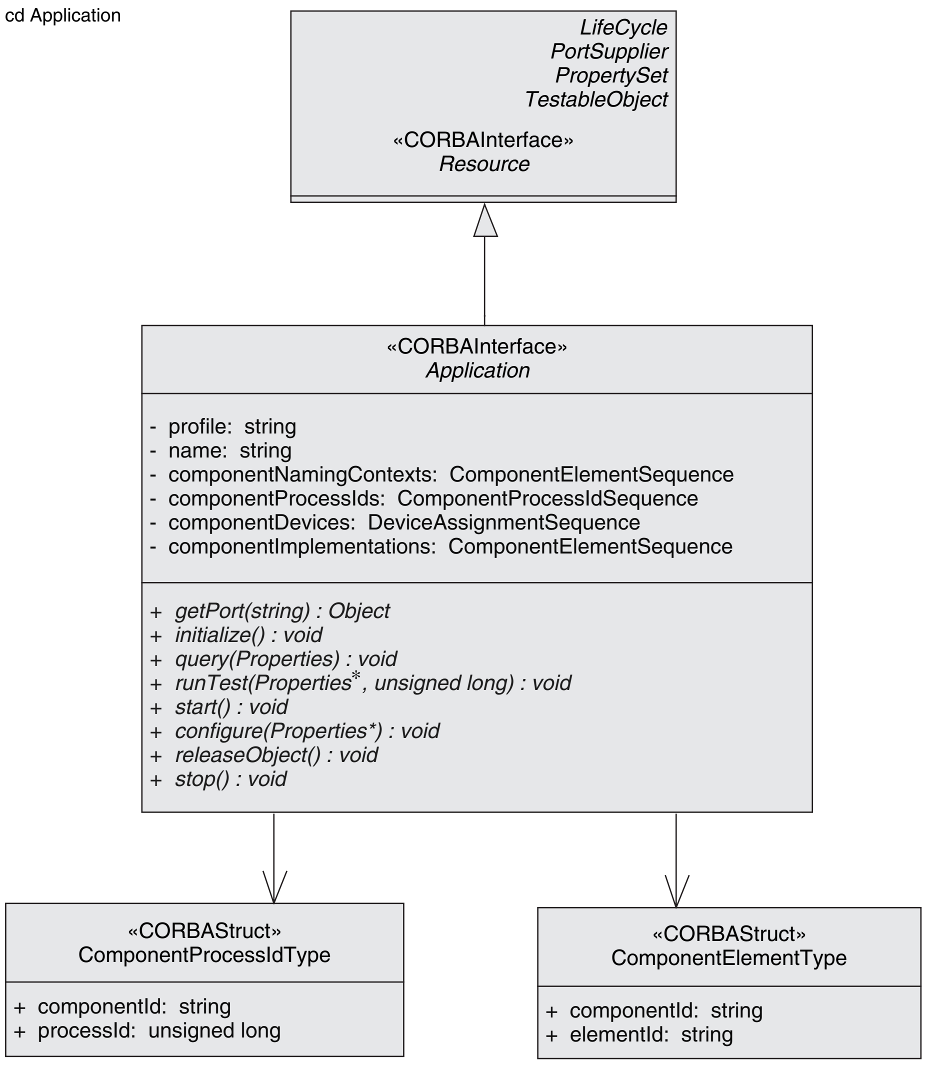

cd Application
LifeCycle
PortSupplier
PropertySet
TestableObject
«CORBAInterface»
Resource
«CORBAInterface»
Application
- profile: string
- name: string
- componentNamingContexts: ComponentElementSequence
- componentProcessIds: ComponentProcessIdSequence
- componentDevices: DeviceAssignmentSequence
- componentImplementations: ComponentElementSequence
+ getPort(string) : Object
+ initialize() : void
+ query(Properties) : void
+ runTest(Properties*, unsigned long) : void
+ start() : void
+ configure(Properties*) : void
+ releaseObject() : void
+ stop() : void
«CORBAStruct»
ComponentProcessIdType
+ componentId: string
+ processId: unsigned long
«CORBAStruct»
ComponentElementType
+ componentId: string
+ elementId: string

Figure 6.4. Application interface

ComponentElementSequence

The mapping of components to elements via the ComponentElementType is managed using the ComponentElementSequence. This is an unbounded sequence of ComponentElementType.

```
typedef sequence <ComponentElementType>
     ComponentElementSequence;
```

6.4.2 Attributes

The Application attribute requirements are listed in Table 6.21.

Table 6.21. Application attribute requirements

Section	ID	Resp	Requirement
3.1.3.2.1.4.1	SR:121	CF	The readonly profile attribute shall contain either a profile element with a file reference to the SAD profile file or the XML for the SAD profile.
3.1.3.2.1.4.2	SR:122	CF	This readonly name attribute shall contain the name of the created Application.
3.1.3.2.1.4.3	SR:123	CF	The componentNamingContexts attribute shall contain the list of components' Naming Service Context within the Application for those components using CORBA Naming Service.
3.1.3.2.1.4.4	SR:124	CF	The componentProcessIds attribute shall contain the list of components' process Ids within the Application for components that are executing on a device.
3.1.3.2.1.4.5	SR:125	CF	The componentDevices attribute shall contain a list of devices, which each component uses, is loaded on, or is executed on.
3.1.3.2.1.4.6	SR:126	CF	The componentImplementations attribute shall contain the list of components' SPD implementation Ids within the Application for those components created.

ComponentElementSequence

The componentNamingContexts attribute contains the list of components' Naming Service Context within the Application for those components using CORBA Naming Service (SR:123). These are specified using a set of ComponentElementType structures as defined by the ComponentElementSequence definition.

```
readonly attribute ComponentElementSequence
     componentNamingContexts
```

componentProcessIds

The componentProcessIds attribute is used to maintain a master list of processes that are executing on an ExecutableDevice (SR:124).

```
readonly attribute ComponentProcessIdSequence
     componentProcessIds;
```

componentDevices

The componentDevices attribute is used to maintain a list of devices to component associations for each component that uses, is loaded on, or is executed on a Device (SR:125).

The association is performed using the component's componentinstantiation element in the Application's software profile.

```
readonly attribute DeviceAssignmentSequence componentDevices;
```

componentImplementations

The componentImplementations attribute contains the list of components' SPD implementation Ids within the Application for those components created (SR:126).

```
readonly attribute ComponentElementSequence
     componentImplementations;
```

profile

This attribute contains the XML profile information for the application. The string value contains a file reference to the SAD profile file (SR:121).[9]

Files referenced within a profile will have to be obtained via a FileManager. The Application will have to be queried for profile information for Component files that are referenced by an Id instead of a file name.

```
readonly attribute string profile;
```

name

The name attribute contains the name of the created Application. The name context of the applications was provided by the ApplicationFactory create operation name (SR:122).

```
readonly attribute string name;
```

6.4.3 Operations

runTest, start, stop, configure, and query

The Application provides a standard implementation within the Core Framework. All the waveform-specific logic and processing is performed by the lower-level components and the AssemblyController (see Table 6.22).

Since the Application does not directly implement these calls, they are delegated to the AssemblyController (SR:137). Any exceptions that are raised by the AssemblyController are propagated by the Application to the caller (SR:128).

[9] *The original SCA 2.2. requirement specified that the profile attribute may contain the actual XML. Because of file names embedded within the SAD XML, there is no practical way to ascertain the relative path of the SAD file in order to obtain the path to the referenced files when the profile attribute contains only the SAD XML. Consequently, the only usable implementation uses the attribute to store the file name.*

Table 6.22. Application operation propagation requirements

Section	ID	Resp	Requirement
3.1.3.2.1.5	SR:127	CF	The Application shall delegate the implementation of the inherited Resource operations (runTest, start, stop, configure, and query) to the Application's Resource component (AssemblyController) identified by the Application's SAD assemblycontroller element.
3.1.3.2.1.5	SR:128	CF	The Application shall propagate exceptions raised by the Application's AssemblyController's operations.

initialize

Although the initialize operation is inherited by the Application from the Resource, there is no action that is performed by any lower level components of the Application (see Table 6.23).

Table 6.23. Application initialize requirements

Section	ID	Resp	Requirement
3.1.3.2.1.5	SR:129	CF	The initialize operation shall not be propagated to the Application's components or its AssemblyController.
3.1.3.2.1.5	SR:130	CF	The initialize operation shall cause no action within an Application.

The initialize operation does not perform any actions within the Application (SR:130) nor is it propagated to the Application's components or AssemblyController (SR:129).

releaseObject

The releaseObject operation is used to terminate and remove the Application from the environment (see Table 6.24). This includes tearing down all of the component connections, terminating execution of any components that may be running on an ExecutableDevice, removing the components, releasing the computing resources.

Next to the create operation on the ApplicationFactory, the releaseObject is the second-most complicated operation. Each of the components that form the Application must be terminated and the resources used by that components returned to the pool of available resources. Each component that was not created using a ResourceFactory is released by calling the releaseObject operation directly on the component (SR:131). For those components that were created using a ResourceFactory, the releaseObject operation is called on the ResourceFactory (SR:132). The ResourceFactory initiated the releaseObject operation on each of the resources created by the ResourceFactory. When all the resources associated

Table 6.24. Application releaseObject requirements

Section	ID	Resp	Requirement
3.1.3.2.1.6.1.3	SR:131	CF	For each Application component not created by a ResourceFactory, the releaseObject operation shall release the component by utilizing the Resource's releaseObject operation.
3.1.3.2.1.6.1.3	SR:132	CF	If the component was created by a ResourceFactory, the releaseObject operation shall release the component by the ResourceFactory releaseResource operation.
3.1.3.2.1.6.1.3	SR:133	CF	The releaseObject operation shall shutdown a ResourceFactory when no more Resources are managed by the ResourceFactory.
3.1.3.2.1.6.1.3	SR:134	CF	For each allocated device capable of operation execution, the releaseObject operation shall terminate all processes/tasks of the Application's components utilizing the Device's terminate operation.
3.1.3.2.1.6.1.3	SCA114	CF	For each allocated device capable of memory function, the releaseObject operation shall de-allocate the memory associated with the Application's component instances utilizing the Device's unload operation.
3.1.3.2.1.6.1.3	SR:136	CF	The releaseObject operation shall deallocate the Devices that are associated with the Application being released, based on the Application's Software Profile.
3.1.3.2.1.6.1.3	SR:137	CF	The Application shall release all client component references to the Application components.
3.1.3.2.1.6.1.3	SR:138	CF	The releaseObject operation shall disconnect Ports from other Ports that have been connected based upon the software profile.
3.1.3.2.1.6.1.3	SR:139	CF	The releaseObject operation shall disconnect consumers and producers from a CORBA Event Service's event channel based upon the software profile.
3.1.3.2.1.6.1.3	SR:140	CF	For components (e.g. Resource, ResourceFactory) that are registered with Naming Service, the releaseObject operation shall unbind those components and destroy the associated naming contexts as necessary from the Naming Service.
3.1.3.2.1.6.1.3	SR:141	CF	The releaseObject operation for an application shall disconnect Ports first, then release the Resources and ResourceFactories, call the terminate operation, and lastly call the unload operation on the devices.

Table 6.24. Continued

Section	ID	Resp	Requirement
3.1.3.2.1.6.1.3	SR:142	CF	The releaseObject operation shall, upon successful Application release, write an `ADMINISTRATIVE_EVENT` log record.
3.1.3.2.1.6.1.3	SR:143	CF	The releaseObject operation shall, upon unsuccessful Application release, write a `FAILURE_ALARM` log record.
3.1.3.2.1.6.1.3	SR:144	CF	The releaseObject operation shall, upon successful Application release, send an event to the Outgoing Domain Management event channel with event data consisting of a DomainManagementObjectRemovedEventType.
3.1.3.2.1.6.1.3	SR:145	CF	The producerId shall be the identifier attribute of the released Application.
3.1.3.2.1.6.1.3	SR:146	CF	The sourceId shall be the identifier attribute of the released Application.
3.1.3.2.1.6.1.3	SR:147	CF	The sourceName shall be the name attribute of the released Application.
3.1.3.2.1.6.1.3	SR:148	CF	The sourceCategory shall be `APPLICATION`.
3.1.3.2.1.6.1.5	SR:149	CF	The releaseObject operation shall raise a ReleaseError exception when the releaseObject operation unsuccessfully releases the Application components due to internal processing errors.

with a ResourceFactory are removed, the ResourceFactory is shutdown (SR:133). For those components that are running on an executable device, the releaseObject operation terminates the processes/tasks using the native terminate call for the particular device and operating system (SR:134). For devices that have an image loaded into their memory, the releaseObject operation releases the memory associated with the load using the unload operation (SR:114).

The releaseObject operation then deallocates the device capacities associated with the components being released (SR:136). Then the client component references are released (SR:137), the ports are disconnected (SR:138), and event consumers and producers are disconnected from the Event Service (SR:139).

If a component has registered with the Naming Service, the releaseObject unbinds the component from the entry in the naming service and removes (destroys) the entry in the Naming Service (SR:140).

For an Application, the releaseObject disconnects the ports prior to releasing the Resources and ResourceFactories, calls the terminate operation on any processes, and then unloads the components from the devices (SR:141).

The releaseObject writes an `ADMINISTRATIVE_EVENT` to the log upon successful release of an Application (SR:142) and sends an event to the Outgoing Domain Management event channel with an event type of DomainManagementObjectRemovedEventType (SR:144). The ProducerId is the identifier attribute of the released Application (SR:145), the sourceId is the identifier attribute of the released Application (SR:146), the sourceName is the name attribute of the released Application (SR:147), and the sourceCategory is APPLICATION (SR:148).

If the releaseObject is not successful, then a `FAILURE_ALARM` is written (SR:143). If the error is due to internal processing errors, then a ReleaseError exception is raised (SR:149).

getPort

In addition to those discussed in Section 4.2, there are two requirements levied on the get Port operation for an Application (Table 6.25).

Table 6.25. Application getPort requirements

Section	ID	Resp	Requirement
3.1.3.2.1.6.2.4	SR:150	CF	The getPort operation shall return object references only for input port names that match the port names that are in the Application SAD externalports element.
3.1.3.2.1.6.2.5	SR:151	CF	The getPort operation shall raise an UnknownPort exception if the port is invalid.

When returning an object reference, the references are limited to the port names that match those specified in the SAD file externalports element (SR:150). Thus, the getPort operation on the Application may not return internal ports used to connect components of the waveform. If the requested port is invalid, then an UnknownPort exception is raised (SR:151).

6.4.4 General Requirements

In addition to the specific requirements noted above, there are several general requirements associated with the Application (Table 6.26).

Table 6.26. Application Requirements

Section	ID	Resp	Requirement
3.2.1.1	SR:603	WS	Applications shall be limited to using the OS services that are designated as mandatory in the SCA AEP as specified in section 3.1.1 of the specifications.
3.2.1.1	SR:604	WS	Applications shall perform file access through the CF File interfaces.
3.2.1.1	SR:605	WS	Application file names shall not exceed 40 characters.
3.2.1.1	SR:606	WS	To ensure controlled termination, applications shall have a signal handler installed for the POSIX-defined SIGQUIT signal.
3.2.1.2	SR:607	WS	Applications shall be limited to using CORBA and CORBA services as specified in section 3.1.2.
3.2.1.3	SR:608	WS	Applications shall implement the CF interfaces as specified in section 3.1.3.1 using the corresponding IDL in Appendix C.

Table 6.26. (Continued)

Section	ID	Resp	Requirement
3.2.1.3	SR:609	WS	Each application process that uses Naming Service shall support the Naming Context IOR, Name Binding, and the identifier execute parameters as described in 3.1.3.2.2.5.1.3 in addition to their user-defined execute properties in the component's SPD.
3.2.1.3	SR:610	WS	The application shall bind its components' object reference to the Naming Context IOR using the Name Binding parameter as described in section 3.1.2.2.1.
3.2.1.3	SR:611	WS	Each executable component of an application shall set its identifier attribute using the component identifier execute parameter.
3.2.1.3	SR:612	WS	Each executable component of an application shall accept arguments of the form described in section 3.1.3.2.6.5.1.3.
3.2.1.3	SR:613	WS	Applications' components and DeviceManagers shall be provided with Domain Profile files as per section 3.1.3.4.

The application may only use those operating system calls or services that are defined in the Application Environment Profile (AEP) (SR:603). The objective of this requirement is to promote portability by limiting the O/S calls by an Application to a well-defined set. Any file access by the Application must be performed using the FileManager, FileSystem, and File interfaces as defined in the IDL (SR:604). The name of an Application may not exceed 40 characters (SR:605). The Application must have a handler to the POSIX SIGQUIT signal to ensure termination (SR:606). The Application is limited to using CORBA and the CORBA Services specified (SR:607). Those interfaces specified in the IDL (actually those that are inherited by the Application from Resource) are to be implemented using the IDL specified in Appendix C of the specification (SR:608). If an Application process uses the Naming Service, then it must support the Naming Context IOR, Name Binding, and identifier parameters specified (SR:609) in the component's SPD and must bind the component's object reference to the Naming Context IOR using the Name Binding parameter (SR:610). Each executable component must set its identifier attribute using the component identifier execute parameter (SR:611). The executable components must accept arguments (SR:612), and all components and DeviceManagers must be defined in the Domain Profile files (SR:613).

7

Operating Environment Security

The security requirements in this section refer to the security of the Core Framework and do not refer to encryption or other type 1 security functions, e.g. COMSEC, TRANSEC, INFOSEC, related to secure communications.

7.1 Core Framework Security Requirements

In a software radio, care must be taken that the critical operations, i.e. providing assured communications, are not compromised through malicious code, corrupted files, process spoofing, inadvertent corruption, or other similar problems.

The security requirements levied on the Core Framework are organized into three groups: the Application, the ApplicationFactory, and the DomainManager. Protecting the DomainManager is important because, once access is gained to the DomainManager, virtually all hardware and software within the system is accessible. The ApplicationFactory must validate that the components being loaded and insered into a waveform are valid components and have not been tampered with or modified. Finally, the Application must ensure that any port connections are authenticated prior to disconnecting.

7.1.1 Application

In addition to the Applications requirements discussed previously, there are several security requirements levied in the operation of the Application (Table 7.1).

When disconnecting ports, only those component ports authorized by an authentication service are to be disconnected (SR:639). The releaseObject requests that the Application Ports access setup are removed from the access control database (SR:640).

If authorization is not received to disconnect component ports, then the releaseObject operation logs a `Security_Alarm` event (SR:641).

As part of the SPD implementation dependency information, the propertyref elements must indicate a dependency to a Red or Black device (SR:642).

Table 7.1. Application security requirements

Section	ID	Resp	Requirement
5.1.1	SR:639	WS	The Application releaseObject operation shall only disconnect components' ports that are authorized by an authentication service.
5.1.1	SR:640	WS	The Application releaseObject operation shall request removal of the Application's Ports' access setups from the access control database.
5.1.1	SR:641	WS	The Application releaseObject operation shall log a `Security_Alarm` event when unable to disconnect components' ports because authorization was not granted by an authentication service.
5.1.1	SR:642	WS	Application components' SPD implementation dependency propertyref elements shall indicate a dependency to a red or black device (directly or indirectly).

7.1.2 ApplicationFactory

The ApplicationFactory must also perform some verification prior the instaniating a waveform Application (Table 7.2).

Table 7.2. ApplicationFactory security requirements

Section	ID	Resp	Requirement
5.1.2	SR:643	CF	The ApplicationFactory create operation shall only create components that are authorized by an authentication service.
5.1.2	SR:644	CF	The ApplicationFactory create operation shall only connect components' ports together that are authorized by an authentication service.
5.1.2	SR:645	CF	ApplicationFactory create operation shall provide an update to the access control database.
5.1.2	SR:646	CF	The ApplicationFactory create operation shall provide updates to an access control database for all components' ports connections as stated in the application's SAD file.
5.1.2	SR:647	CF	The ApplicationFactory shall log a `SECURITY_ALARM` event when unable to connect ports or create components because authorization was not granted by an authentication service.

Instantiation of a waveform requires that the ApplicationFactory verify the creation request with the authentication service (SR:643) and only connect those ports authorized (SR:644). Any necessary updates to the access control database are performed for the create operation (SR:645) and connection of the ports (SR:646). If any exception was encountered due to authorization not being granted, then the ApplicationFactory logs a SECURITY_ALARM event (SR:647).

7.1.3 DomainManager

The DomainManager also has security requirements levied on it at the system level (Table 7.3).

The DomainManager provides information specified in the Security Supplement to enable the control/bypass between red and black side components (SR:648). Upon uninstallation of an application, the DomainManager removes the control/bypass information (SR:649). The Device requires the Domain Profile XML files to indicate whether it is a black or red side device (SR:650). When a Device has sub-Devices or child Devices, the parent send the control/bypass information to the child devices (SR:651). If the device has no parent, it sends its information directly to the control/bypass mechanism (SR:652).

Table 7.3. DomainManager security requirements

Section	ID	Resp	Requirement
5.1.3	SR:648	CF	The DomainManager installApplication operation shall send the information specified in the Security Supplement to the control/bypass mechanism Resource for the black-side components being accessed by red-side components and for red-side components being accessed by black-side components.
5.1.3	SR:649	CF	The DomainManager uninstallApplication operation shall request removal of the application's information specified in the Security Supplement from the control/status bypass mechanism.
5.1.3	SR:650	CF	Devices SPD properties shall have an allocation property that indicates a red or black device.
5.1.3	SR:651	CF	Parent Devices shall send their child Devices information specified in the Security Supplement to the control/status bypass mechanism.
5.1.3	SR:652	CF	A parentless Device shall send its information specified in the Security Supplement to the control/status bypass mechanism.

8

Certification

8.1 Certification Process

The process of certifying an SCA-compliant system was originally managed by the JTRS JPO through the JTRS Technical Lab (JTeL). The JTeL had offices in San Diego, California and Charleston, North Carolina. Recent organization changes have re-allocated functional responsibilities as well as changed organizational names and alignments. This has changed the certification to a certain degree and there will likely be changes after this book is released. However, regardless of organizational and process changes that may occur, the necessity remains of ensuring that an SCA-compliant radio system adheres to the specification and meets certain compatibility requirements.

This chapter will provide some basic background information on the certification process. There are two essential certification and assessment processes for an SCA-compliant radio system and the waveforms hosted on the radio system. These are: i) Operational Environment (OE) certification and ii) Waveform Assessment. The balance of this chapter provides an overview of the certification process in these two areas.

It should be noted that prior to the deployment of any radio by the US military, all radio systems, SCA-compliant included, must undergo interoperability testing at the Joint Test and Interoperability Command (JTIC). The JTIC performs operational testing of the radio system in concert with existing radio systems. JTIC testing ensures the radio performs as expected in realistic scenarios and environments. Interoperability with existing waveforms and radios is thoroughly tested prior to acceptance and deployment.

OE certification focuses on the integration of the radio system hardware, Core Framework, CORBA ORB, and operating system. Essentially each of the testable requirements discussed in Part I of this book is verified. An automated test tool has been developed to perform the testing process for an OE. The JTRS Test Application (JTAP) was developed to perform the OE testing. The JTAP covers the defined requirements as well as a number of derived requirements. The JTAP is a comprehensive tool that provides a well-designed test harness for the SCA OE certification. It has become the 'gold standard' by which the compliance of an SCA system is evaluated.[1]

Waveform Assessment addresses the design, documentation, modularity, and other aspects of the waveform. As such, it is less of a quantitative test than the JTAP. The waveform assessment includes porting the waveform to a representative hardware set. The objective of this exercise is to gauge the modularity and portability of the waveform, not necessarily to obtain a ported version that operates in real time.

8.2 Operating Environment Certification

There are three process flows for OE certification:

- OE-1 – The Radio system supplier obtains the JTAP tool and executes the test procedures.
- OE-2 – The radio system supplier executes his or her own suite of test procedures.
- OE-3 – The JPEO executes the JTAP tool.

Practically speaking, OE-1 tends to be the path most often followed. With OE-2, the supplier must show that their test procedures verify all the requirements in the specification. For those that have already invested the time to develop the set of procedures, this may be a cost effective path. For those who have not built a comprehensive test suite, it is more cost effective to use the JTAP tool. OE-3 is applied infrequently. This is due, in part, to the level of support required to configure, run, and analyze the tests. Additional support from the radio supplier is typically required as well. So, it usually becomes more efficient to follow the OE-1 path.

The certification process was originally developed to work in concert with the SCA radio procurement programs, called Cluster 1 through Cluster 5. Thus the process flows show the participation of a representative from the Cluster Program Management Office (PMO). While these programs have undergone some transformation recently, in scope, organization, and name, the essential process remains the same.

8.2.1 OE-1

As previously mentioned, OE-1 is frequently used as the process of choice to obtain certification of the OE. For OE-1, as illustrated in Figure 8.1, the JTeL provides a copy of the JTAP tool to the JTRS Tactical Radio (JTR) manufacturer. The manufacturer then ports the JTAP tool to their system.[2]

The JTAP is then run against the system. The test may be witnessed by a representative from the JTeL and typically is for formal certification. Usually the test is run internally by the manufacturer, defects are corrected, the test is rerun, and a copy of the results may be provided informally to the JTeL for review prior to a formal certification run.

[1] *Although the JTAP is a well-designed and implemented tool, there are a number of deficiencies that require changes to the test harness. In addition, the JTAP has not been brought up to date with the current release of the SCA specification. So, additional funding and work is required to maintain currency with the SCA specification and address some discrepancies in the program implementation. Nonetheless, it provides a significant and valuable automated test harness for a complex set of requirements.*

[2] *This is not a complete port of the JTAP tool. However, there are several components that need to be compiled and built for the specific target system that is to be tested. The JTAP includes the baseline implementation of these components and the JTR manufacturer adapts and builds them for their specific hardware and software configuration.*

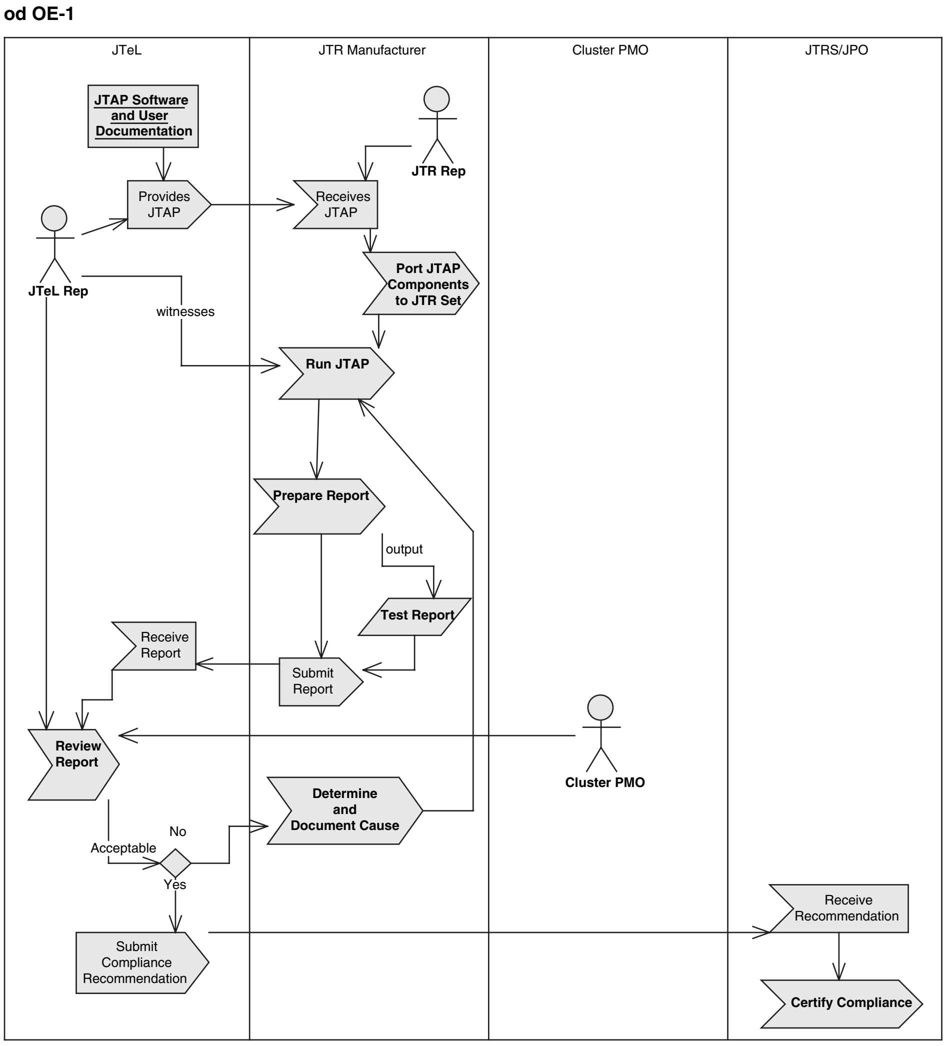


Figure 8.1. OE-1 certification process

After the formal certification run, a complete test report is prepared and submitted to the JTeL for formal review. If the JTeL decides that there are significant problems with the report results, the problems are documented and forwarded to the JTR manufacturer for correction, and the system is not recommended for certification.

If the test is completed successfully, the JTeL issues a recommendation for a compliance certification to the Joint Program Office, which then issues a certification letter to the manufacturer. The case may also arise where the JTAP run was predominantly successful and yet certain tests failed that were deemed to require correction; however, these were not critical enough to prevent certification. In this case, the JTeL may recommend and the Joint Program Office may grant certification with waivers.

8.2.2 OE-2

For OE-2, as illustrated in Figure 8.2, the manufacturer has developed its own test harness. The manufacturer prepares an OE test plan and submits it to the JTeL for review and approval.

Once the test plan has been approved by the JTeL, the remainder of the process essentially follows the same path as OE-1. Although there may have been several manufacturers that

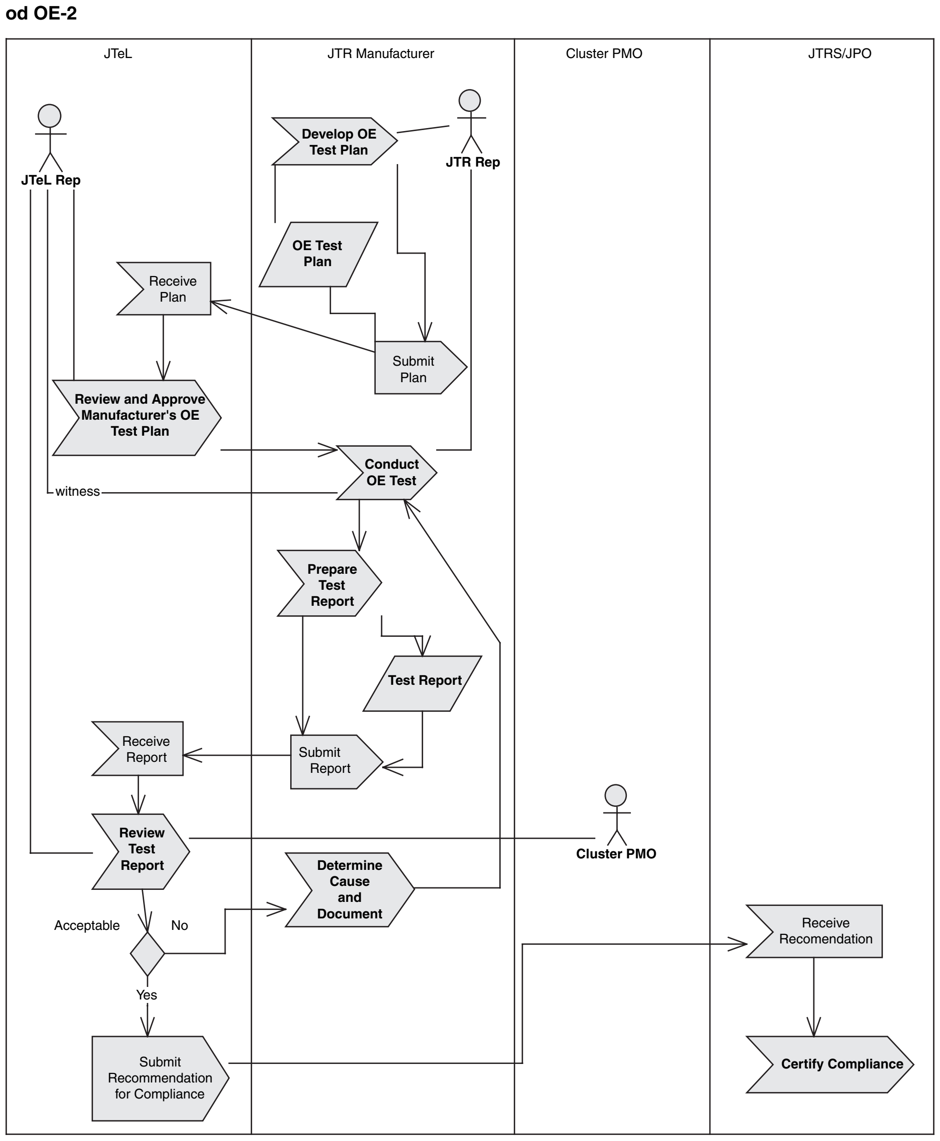


Figure 8.2. OE-2 certification process

initiated this process prior to release of the JTAP tool, few if any manufacturers continue to use the OE-2 process for certification due to the cost in time and effort on the part of the manufacturer to maintain the test code and processes, and the time required by the JTeL personnel to review the test process thoroughly and plan for approval. Once the JTAP tool was released it quickly became the standard for SCA OE testing.

8.2.3 OE-3

In OE-3, shown in Figure 8.3, the basic flow is similar to OE-1. However, the difference is that instead of the JTeL providing the JTAP tool to the manufacturer and the manufacturer

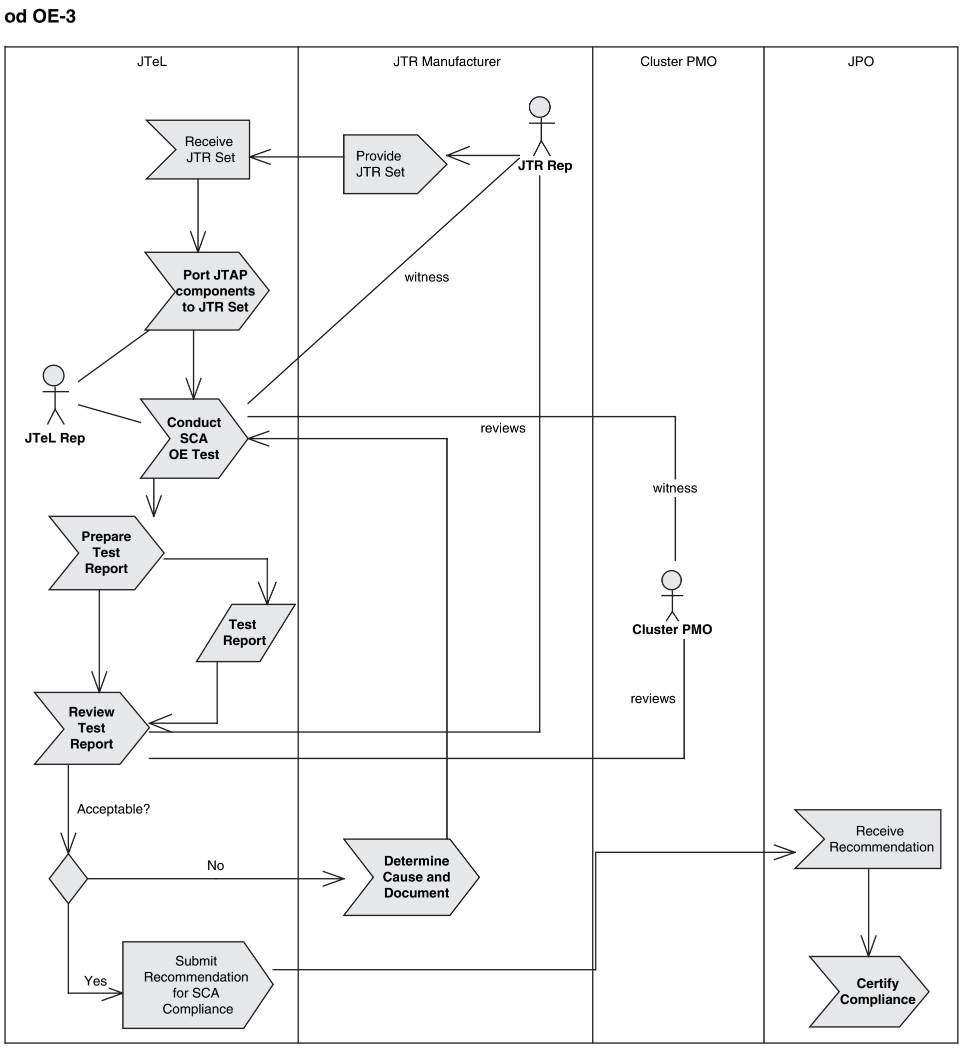


Figure 8.3. OE-3 certification process

performing the testing, the manufacturer provides the radio system to the JTeL and the JTeL performs the testing.

This process places a significant burden on the JTeL organization and personnel. In addition to the time and effort to port the JTAP components for the radio set to be tested, personnel from the manufacturer were required to provide domain knowledge and expertise to interpret test results and identify problem locations in the system or software.

As with OE-2, the OE-3 process is not a viable process because of the additional cost and diversion of personnel at the JTeL to support the testing process. Thus, OE-1 has become the defacto process for performing SCA certification.

8.3 Waveform Assessment and Certification

Although the interoperability of the radio system and the waveforms that it hosts are tested by the JTIC, the JTeL does perform an assessment of the waveforms delivered to the government to run on an SCA system.

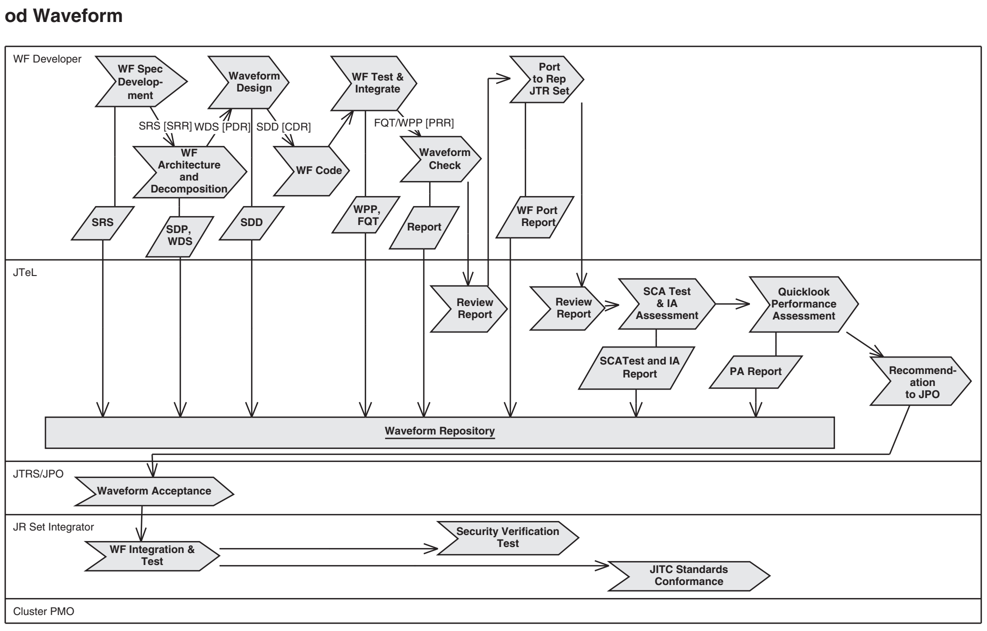


Figure 8.4. Waveform assessment process

Figure 8.4 illustrates the waveform assessment process. In general, the process involves the review and analysis of each of the work products associated with a waveform design and implementation. This includes Software Requirements Specification (SRS), Software Development Plan (SDP), Software Design Document (SDD), and other deliverables.

Between each phase of the waveform development process, key reviews and analyses are performed to assess the viability of the waveform design and implementation.[3]

In addition to performing an analysis of the deliverables, a waveform port is also performed. The objective of this effort is not necessarily to obtain a fully functional implementation of the waveform on another processing platform, although that would be preferred. The objective is to gather some empirical data on the modularity of design and portability aspect of the waveform.

Once the work products and design artifacts have been reviewed, the gathered porting data has been evaluated, and the implementation has been determined to meet the general criteria identified for portability, the JTeL issues a recommendation that the waveform be accepted into the JTRS library.

[3] *The functional aspect of the waveform is not under scrutiny in this process: rather, the focus is on traditional aspects of good software design, e.g. modularity, information hiding, encapsulation, etc. The objective is to ensure that the waveform delivered has been designed in such a way that, together with the documentation, it can be ported to another platform without undue effort. There is, of course, no guarantee that a subsequent porting activity will be cost effective or even feasible because there are certain aspects of the I/O, control, and processing architecture that can significantly increase the porting effort and cost.*

PART II

The Domain Profile

Vincent J. Kovarik Jr.

In Part I, the SCA specification, requirements, and general functional behavior were discussed. Background was also presented on the certification process for the Operating Environment and the Waveform. One of the key capabilities of an SCA radio system is its ability to deploy a waveform on a set of hardware without having any *a priori* knowledge about the waveform or radio system hardware. This is possible through the use of XML files that describe the Domain Profile for the radio system. The Domain Profile provides the essential description of the radio platform and hardware devices, and the waveform, its components, dependencies, and connections. Part II presents the format and content of the XML files that form the Domain Profile.

9

The Domain Profile

9.1 Overview

This chapter describes the use of XML as the specification language for defining components of the software radio and how these assemblies are organized. A waveform application is described within the context of the SCA using a collection of XML files. The root or origination point for describing a software waveform is the Software Assembly Descriptor (SAD) file. The SAD file provides the starting point for instantiating a waveform within an SCA-compliant radio. This is performed through inclusion of additional XML files that define waveform components, properties associated with components, interfaces, and other details of the waveform.

In addition, a set of XML files describes the radio system hardware. These files provide a specification of the Devices within the SCA radio system and the capabilities of the hardware. Mapping the requirements of the waveform, as specified in the SAD file and included files, to the available hardware resources is the fundamental process performed by the create operation on the ApplicationFactory discussed in Chapter 6. The balance of this chapter presents the organizational elements and content of the Domain Profile XML files.

9.2 SCA Domain Profile XML

The organization of the XML files is illustrated in Figure 9.1. Within any SCA-compliant radio, there is a domain. The domain can be thought of as the collection of devices, physical and logical, software components, and applications (i.e. waveforms) that all reside within the system. Thus, within a single domain, there may be multiple Device Configuration Descriptors that describe physical hardware assemblies and the software that resides on the assemblies. There may also be multiple SADs within a domain. Each SAD describes the set of logical components that comprise an application and the logical and physical resources that are required to support the application.

Together, the collection of XML files comprise the Domain Profile of the system. It should be noted, however, that a Domain Profile is not necessarily a static state; the Domain Profile can and does change over time. The Domain Profile changes when a new hardware component is installed, when a software implementation changes for a device interface, or

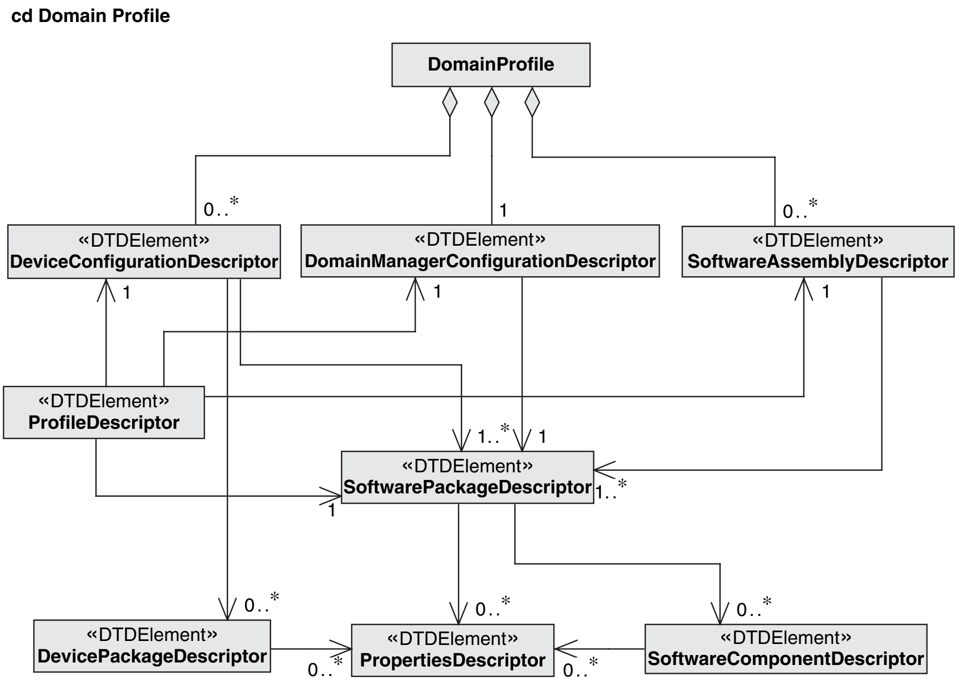

cd Domain Profile
DomainProfile
0..*
1
0..*
«DTDElement» DeviceConfigurationDescriptor
«DTDElement» DomainManagerConfigurationDescriptor
«DTDElement» SoftwareAssemblyDescriptor
1
1
1
«DTDElement» ProfileDescriptor
1..*
1
«DTDElement» SoftwarePackageDescriptor
1..*
1
0..*
0..*
0..*
«DTDElement» DevicePackageDescriptor
«DTDElement» PropertiesDescriptor
«DTDElement» SoftwareComponentDescriptor
0..*
0..*

Figure 9.1. Domain Profile XML file relationships

when a new waveform is installed. Thus, the Domain Profile is an information set that describes the overall hardware, software, and applications available to the system at a given point in time. Similarly, the Domain Profile can also change as hardware, software, and applications are removed from the system.

It should also be noted that the Domain Profile, as represented by the set of XML files, is simply that: an external representation form. In order to perform any computational processing with the information contained within the XML files, they must first be ingested into the system through an XML parser. The XML parser reads each of the constituent XML files for a Domain profile, validates each file against an XML Document Type Definition (DTD) file, and builds a hierarchical tree or Document Object Model (DOM). Although the DOM is a data structure that is traversable programmatically and can be used as the internal representation mechanism, it is not computationally efficient to perform the tree traversal. Consequently, most implementations utilize an internal set of classes or data structures that provide the domain profile representation in a much more efficient manner.

A Domain Profile contains at least a Domain Manager Descriptor (DMD) file and a Device Configuration Descriptor (DCD) file (see Table 9.1). The DMD contains the information necessary to start the Domain Manager, the top-level control entity for the SCA Core Framework. The DCD contains the information describing at least one Device, usually a GPP, on which the Domain Manager is hosted. Thus, the initial DCD identifies the startup or boot node processor and the DMD provides the information necessary to start and run the

Domain Manager. The SAD file, however, because it is not necessary to have a waveform application specified and installed when the system first boots up, is not necessary for system startup.

Table 9.1. Domain Profile requirements

Section	ID	Resp	Requirement
3.1.3.4	SR:588	SI, WS	Domain Profile files shall use the format of the Document Type Definitions (DTDs) provided in Appendix D of the specifications.
3.1.3.4	SR:589	SI, WS	DTD files are installed in the domain and shall have '.dtd' as their filename extension.
3.1.3.4	SR:590	SI, WS	All XML files shall have as the first two lines an XML declaration (?xml) and a document type declaration (!DOCTYPE).
3.1.3.4.1	SR:591	SI, WS	A Software Package Descriptor file shall have a '.spd.xml' extension.
3.1.3.4.2	SR:592	SI, WS	A Software Component Descriptor file shall have a '.scd.xml' extension.
3.1.3.4.3	SR:593	WS	A Software Assembly Descriptor file shall have a '.sad.xml' extension.
3.1.3.4.4	SR:594	SI, WS	A Properties File shall have a '.prf.xml' extension.
3.1.3.4.5	SR:595	SI, DS	A Device Package Descriptor File shall have a '.dpd.xml' extension.
3.1.3.4.6	SR:596	SI, DS	A Device Configuration Descriptor file shall have a '.dcd.xml' extension.
3.1.3.4.8	SR:597	SI, CF	A DomainManager Configuration Descriptor file shall have a '.dmd.xml' extension.

The structure and syntax of the Domain Profile XML files are defined by a DTD file (SR:588). The DTD file can be thought of as a grammar specification, similar to the Backus-Naur Form (BNF) commonly used to describe the syntaxtic structure of programming languages. The DTD files are required to have the three-letter extension '.dtd' in the file name (SR:589) and an XML declaration and document type declaration as the first two lines (SR:590).

The XML files for each of the different types of domain profile files have specific extensions for each type. These extensions and the associated file type are as shown in Table 9.2.

Rather than follow the approach used in Appendix D of the SCA specification, the structure defined by the DTD files has been represented using UML diagrams. The objective is to provide a visual representation of the grammar specified in the DTD files that is easier to follow than the textual DTD while still maintaining the syntax defined by the DTD file.[1]

[1] *In Appendix D, there are also UML diagrams. The following sections provide a logical discussion of the XML file syntax, as specified by the DTDs, using UML containment diagrams. The syntax is presented in a graphical format using UML. The key difference between the UML presented in Appendix D and as represented here is that the element relationships are modeled as containment aggregation links rather than simple associations.*

Table 9.2. XML file name requirements

File Name Extension	File Type	Requirement
'.spd.xml'	Software Package Descriptor	SR:591
'.scd.xml'	Software Component Descriptor	SR:592
'.sad.xml'	Software Assembly Descriptor	SR:593
'.prf.xml'	Profile Descriptor	SR:594
'.dpd.xml'	Device Package Descriptor	SR:595
'.dcd.xml'	Device Configuration Descriptor	SR:596
'.dmd.xml'	Domain Manager Descriptor	SR:597

The next chapter presents common descriptor files that are referenced in multiple high-level descriptors. The descriptor files discussed are the Properties Descriptor File, the Software Package Descriptor File, the Software Component Descriptor, and the Device Package Descriptor.

9.3 Domain Profile Data Types

The Domain Profile data types are illustrated in Figure 9.2.

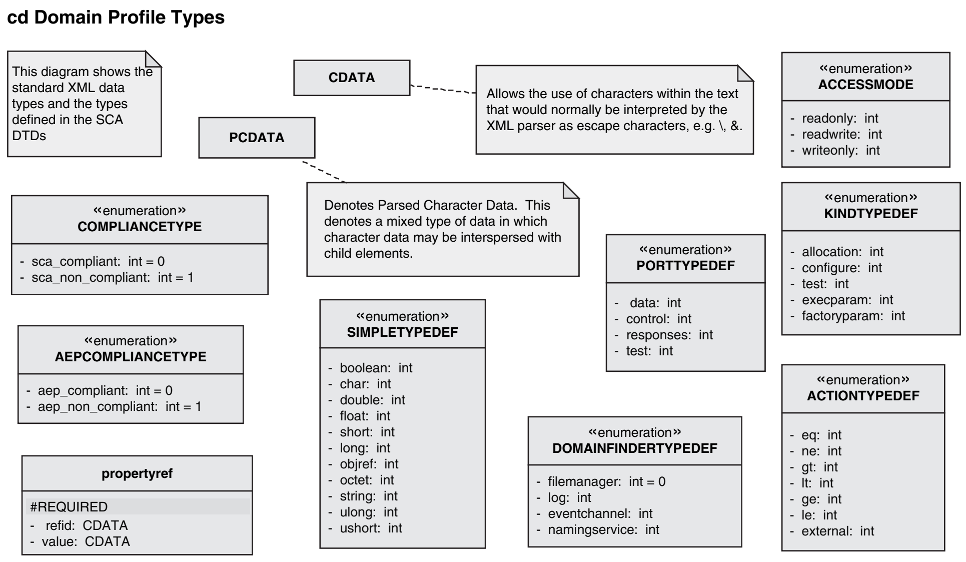


Figure 9.2. Domain Profile Data Types

10

Base Descriptor Files

10.1 Properties Descriptor

The Properties Descriptor describes the property definitions (see the Property Set IDL interface) for Software Package Descriptor (SPD), the Software Component Descriptor (SCD), and the Device Package Descriptor (DPD). The SPD, SCD, and DPD are incorporated multiple times across each of the top-level descriptors.

Properties use a name-value pair representation. One way to envision the PropertySet is as an instantiation of a hash table where the property name is the key and the value can be set or retrieved based on the key value. There is additional information associated with the property definition, e.g. access mode, data type, description, which defines access to the property.

The functions configure and query are used to set or get the value of a property. The PropertySet interface is inherited by the Resource, DomainManager, and DeviceManager interfaces.

The Property file consists of one or more property definitions and an optional description for each property (see Figure 10.1). There are two basic property types, simple and struct, and two additional types, simple sequence and struct sequence, that provide a mechanism for sequences of simple and struct property types. In addition, a test type identifies that the property is used with the runTest() operation.

10.1.1 Simple

The simple property type is, as the name implies, a simple data type. The data types, `SIMPLETYPEDEF`, are roughly equivalent to the simple data types provided in a programming language such as C. The access mode of the property is defined by the *mode* attribute and is only applicable if the kind element is configure. Three types of access are supported: readonly, readwrite, and writeonly. The *Id* attribute contains a unique Id for the property. If the property is an allocation type, then the Id must be a DCE UUID. The *name* attribute contains a string name for the attribute.

As shown in Figure 10.2, all of the subelements of the simple element are optional. The only mandatory information required to define a simple property are the Id, type, and mode

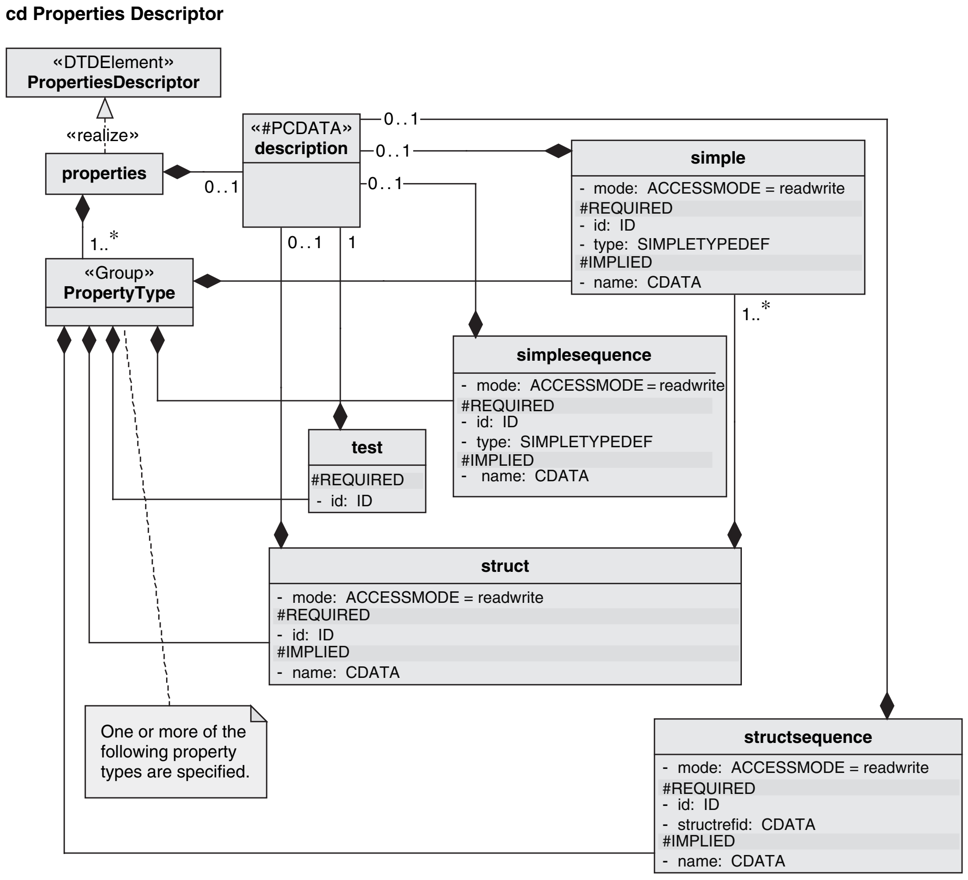


Figure 10.1. Property File Descriptor contents

attributes defined above. The name attribute, which defines a string name for the property, is optional.

The description subelement provides an optional block for text describing the property. The optional value subelement is used to specify a value for the property. This value may be subsequently changed within the running system if the mode attribute is set to readwrite or writeonly.[1]

The units' subelement provides a method for defining the interpretation of the value, i.e. the value might be 44.1 and the units might be 'KHz' signifying the standard sampling rate for an audio Compact Disc (CD). The range subelement, if specified, defines a high and low

[1] *If an initial value is not specified, then the property may not be used for input values on the runTest() interface or as an initial configuration or an execute parameter to a component.*

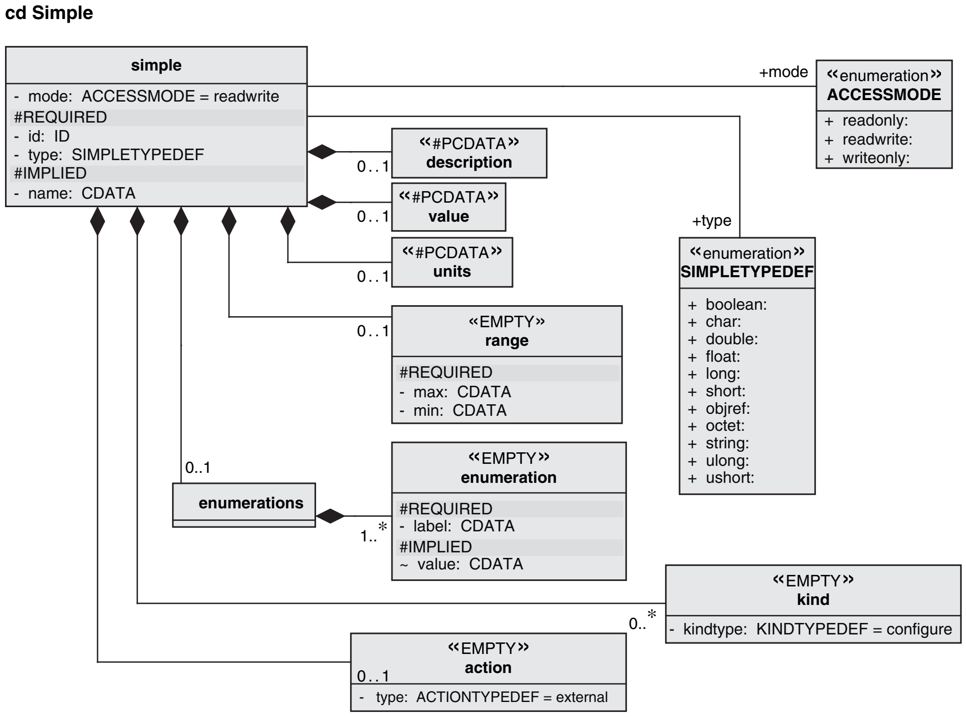


Figure 10.2. Simple element

range for the property. This element is not utilized by the ApplicationFactory or Application components.[2]

The enumeration subelement supports the ability to define a set of label/value pairs for a property. If values are not provided with the labels then the values associated with the labels are ordinal numbers starting with zero for the first enumeration element and incrementing by one for each of the subsequent enumeration elements.

The kind element is used to describe the attribute's purpose and use. The type is defined by setting the kindtype attribute on the kind subelement. There are five types:

1. **configure** – This type identifies that the property may be used in the query() and configure() operations. If the mode is defined as readonly, then the query() operation is supported. If the mode is writeonly, then the configure() operation is supported. If the mode is readwrite, then both operations are supported. During the creation of a Resource component by the ApplicationFactory, the ResourceFactory, or the DeviceManager, those properties defined as configure for a component are provided as input parameters to the Resource being created.

[2] *There is no requirement identified that the high and low ranges, if specified for a property, are enforced by the configure() call on a Property.*

2. **test** – A value of test in the kindtype property is used by the runTest() operation. Since the runTest() operation specifies a sequence of unsigned long values as input, any property with a kindtype of test must have a type attribute of ulong.
3. **allocation** – A property with a kind value of allocation is typically used within the allocateCapacity() and deallocateCapacity() operations within the create() operation on the ApplicationFactory. If the property also has an action type of external (discussed below), then the property may also be retrieved through the query() operation.[3]
4. **execparam** – The execparam kind value identifies that the property is used within the execute() operation on the ExecutableDevice. The ApplicationFactory and DeviceManager builds a sequence of the execparam properties to pass to the execute() call on the creation of a process on an ExecutableDevice.
5. **factoryparam** – The factoryparam kind value identifies that the property is to be used as part of the createResource() operation on the ResourceFactory.

The action subelement identifies how a SPD property value should be compared to a property associated with a Device when checking SPD dependencies. The action value defines the type of boolean comparison to be made, e.g. equal, greater than, etc. The convention is that the allocation property is on the left side of the comparison operator and the dependency value is on the right.

10.1.2 Simple Sequence

The simplesequence property is essentially the same as the simple property. The difference is that the simplesequence supports a sequence of values instead of a single value (see Figure 10.3).

Each of the elements in the sequence of values must be the same type and must be one of the defined simple types. The remainder of the subelement definitions remain the same as the simple property.

10.1.3 Struct

The struct subelement supports the definition of a property that is made up of a set of two or more discrete values and types (Figure 10.4). This is essentially analogous to the struct definition within the C language.

[3] *Although a value of allocation on the kind identifies that the property is used for the allocateCapacity and deallocateCapacity operations on the Device, the syntax defined states that the kind definition is optional. This is because properties are used across virtually all Domain Profile components and not all parts of the Domain Profile require an allocation kind. One might also think that there are devices that may have an API but do not require allocation to a particular application, e.g. Power System. One could make the argument that the partial allocation of power consumed based on the load imposed by the application running could be performed. However, the state of the art is not yet at that level of sophistication. The catch comes in when a Device attempts to register with the DomainManager. One of the actions performed by the DomainManager on Device registration is to obtain the allocation property of the Device that is registering. If no allocation property is provided, then an exception is thrown. This is an example of how a Device may be implemented correctly, the XML may pass the syntax validation, and yet, when the system starts, an error is encountered.*

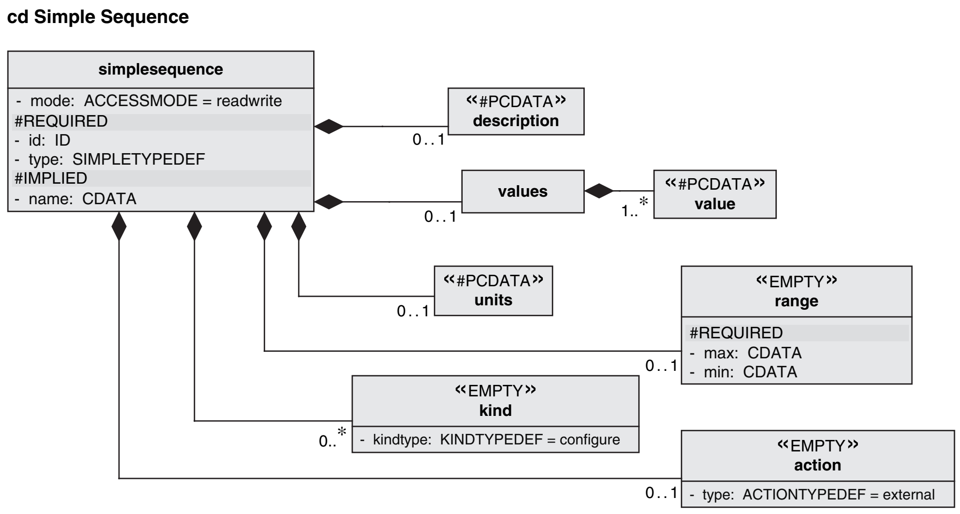


Figure 10.3. Simple Sequence element

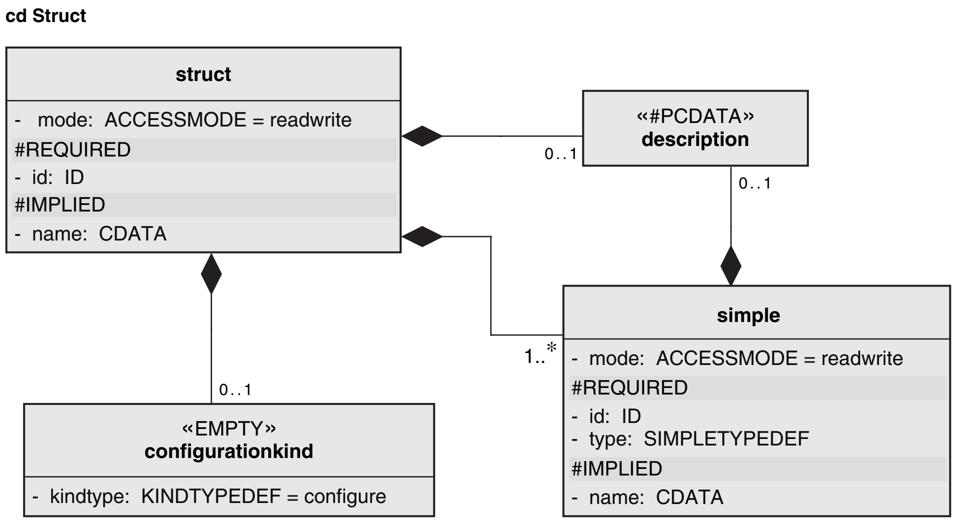


Figure 10.4. Struct element

The struct element identifies the access mode to apply, which defaults to readwrite, and has a mandatory Id and optional name attributes. The one or more entries of the simple subelement are used to define the structure and organization of the struct. Both the struct element and the simple subelement may have a description subelement.

A single configurationkind value is defined for the struct by the configurationkind element. However, only two values are allowed for the struct type instead of the five values allowed on the simple property type. The legal values are:

1. **configure** – This type identifies that the property may be used in the query() and configure() operations. If the mode is defined as readonly, then the query() operation is supported. If the mode is writeonly, then the configure() operation is supported. If the mode is readwrite, then both operations are supported. During the creation of a Resource component by the ApplicationFactory, the ResourceFactory, or the DeviceManager, those properties defined as configure for a component are provided as input parameters to the Resource being created.
2. **factoryparam** – The factoryparam kind value identifies that the property is to be used as part of the createResource() operation on the ResourceFactory.

10.1.4 Struct Sequence

The structsequence element is analogous to the simplesequence element. It defines a type that allows a sequence of struct values (Figure 10.5). As with the simplesequence, each of the values of a structsequence must be the same.

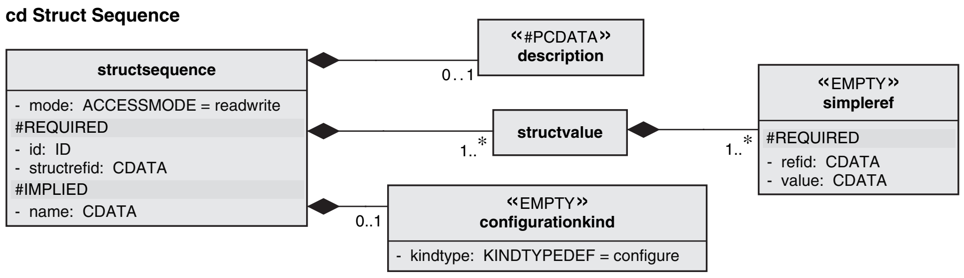


Figure 10.5. Struct Sequence element

The configurationkind and description subelements are the same as on the simplesequence. One or more structvalue entries form the sequence. Each struct value is composed of a sequence of one or more simpref entries. The simpleref has two attributes, a refid referring to the data element within the struct and the value attribute that contains the data for that element.

10.1.5 Test

The test element of the properties definition is used to define a set of properties for the runTest() operation. It has an Id attribute, which is mandatory, that provides a unique Id for the test entry. There are three subelements: description, inputvalue, and resultvalue (Figure 10.6).

On most elements the description element is optional. However, it is mandatory on the test element. This is to require some descriptive information about the test that is initiated, what the test performs, and results provided.

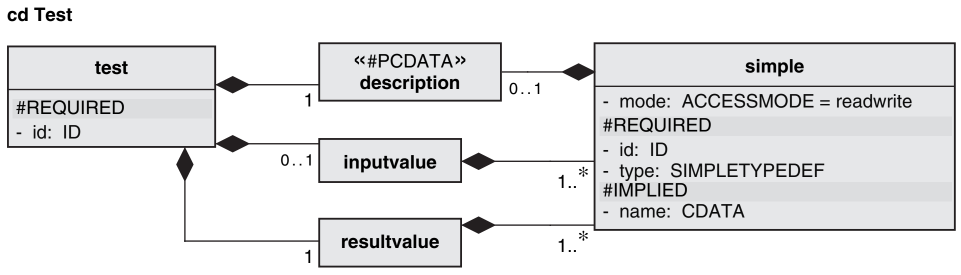


Figure 10.6. Test element

The inputvalue is optional and, if specified, is a simple property. The resultvalue properties contain the result of the test upon completion.

10.2 softpkg

The Software Package Descriptor (SPD) provides the information necessary to manage the software component (see Figure 10.7). For a waveform component, the SPD specifies

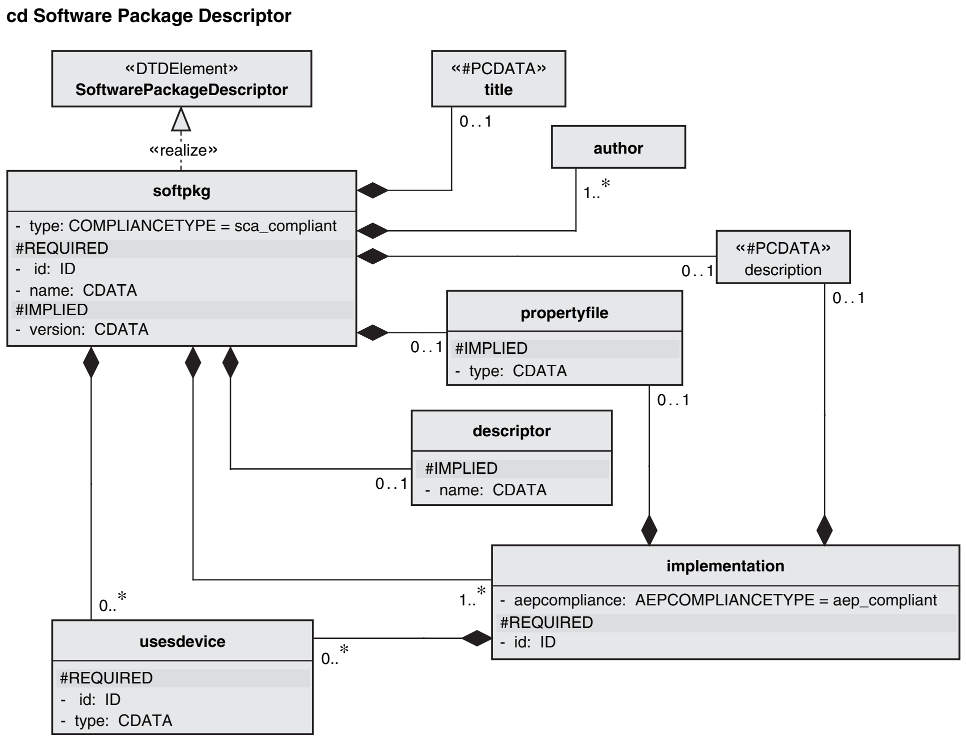


Figure 10.7. Software Package Descriptor elements

implementation choices available to the DomainManager and ApplicationFactory. The SPD may have a property file that defines general properties for the component and each implementation may have an associated property file. All files referenced by a Software Package are located in the same directory as the SPD file or a directory that is relative to the directory where the SPD file is located.

The SPD Id attribute contains a unique Id for the component in the form of a DCE UUID. The name attribute contains a readable name for the component. These two attributes are mandatory.

The SPD also has a type attribute identifying whether the component is SCA-compliant or not. The default value is `sca_compliant`. In addition, the version attribute may be provided to identify the particular version of the component.

10.2.1 title

The optional *title* element provides descriptive information about the software package described by the SPD.

10.2.2 author

One or more *author* elements are required by the *softpkg* definition. The *author* element contains information about the author's name, affiliation, and web page link

10.2.3 description

The optional *description* element provides readable information about the software package.

10.2.4 propertyfile

The *propertyfile* element identifies the local filename of the Property Descriptor file for the Software Package (Figure 10.8). At the SPD level, the *propertyfile* is used to define property elements common to all component implementations referenced within the SPD.

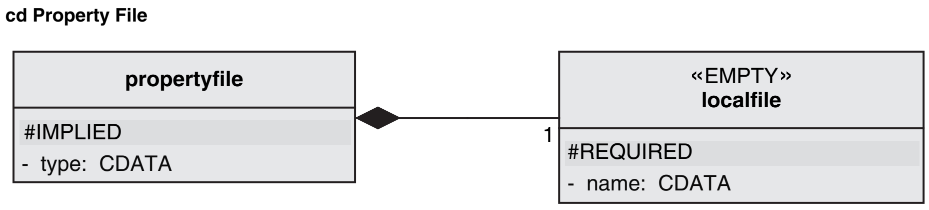


Figure 10.8. PropertyFile element

If the *propertyfile* element is present then is must have a *localfile* element defined. The name attribute of the *localfile* element provides the reference to a file in the same directory as the SPD file or a directory that is relative to the directory where the SPD file is located.

10.2.5 descriptor

The optional *descriptor* element provides a reference to the Software Component Descriptor (SCD) file that provides information on the interface for the SPD (Figure 10.9).

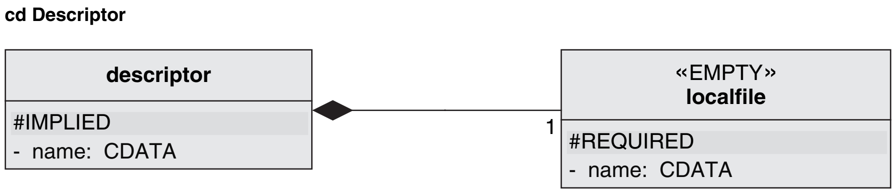


Figure 10.9. Descriptor element

If the *descriptor* element is provided, it must contain a *localfile* element. The name attribute of the *localfile* element provides the reference to a file in the same directory as the SPD or a directory relative to the SPD directory. The SCD provides information about the component type, message ports, and IDL interfaces. The SCD is optional because some components may not be SCA compliant.

10.2.6 implementation

The software package must contain one or more instance of the *implementation* element. The *implementation* element provides information regarding a specific implementation of the software package (Figure 10.10).

The aepcompliance attribute identifies whether the component adheres to the Application Environment Profile or not. The mandatory Id attribute provides a unique identifier for the implementation being described.

code

The *code* element is mandatory and has a single mandatory subelement, *localfile* (see Figure 10.11). The *localfile* element defines the name of the file that implements the component. The type attribute of the code element has four possible values:

- **Executable** – This corresponds to a program that executes as a process within an operating system.
- **Driver** – This corresponds to a load module that may be incorporated or linked into some other component.
- **Kernel** – This is similar to Driver but refers to load modules that are incorporated as part of the operating system routines.
- **SharedLibrary** – A shared library may be either an executable element or a load only element depending on whether the entrypoint value is set or not.

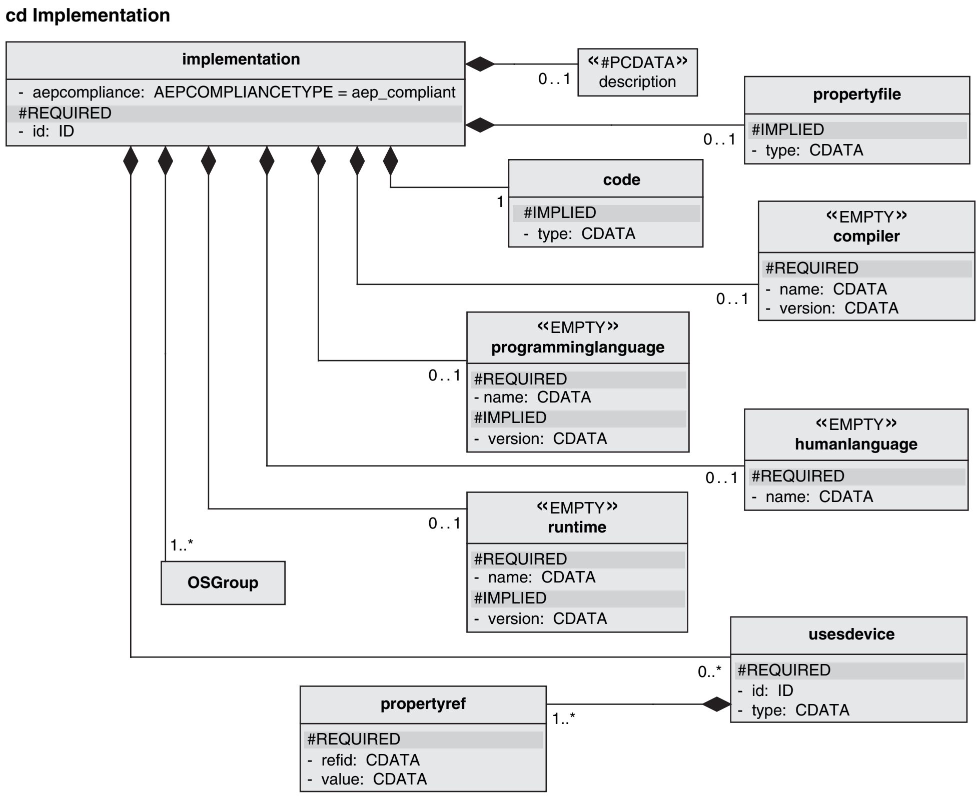


Figure 10.10. Implementation element

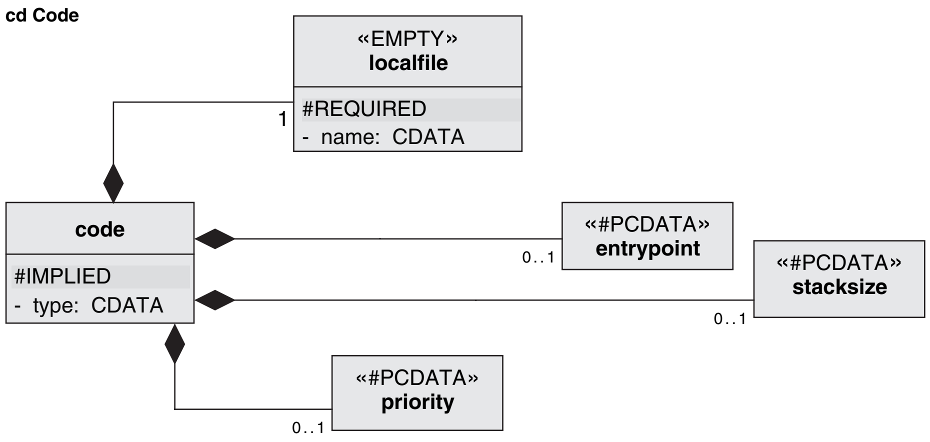


Figure 10.11. Implementation code element

The *stacksize* and *priority* are optional parameters used in the *execute*() operation. Data types for the values of these options are unsigned long. The *entrypoint* element specifies the name of the entry point of the component.

The *compiler, runtime, programminglanguage, humanlanguage, os*, and *processor* elements are optional dependency elements The first four are described below (*os* and *processor* are described in the next section):

- *compiler* – This element specifies the compiler used to build the software component. The required *name* attribute provides the name of the compiler used, and the *version* attribute identifies the compiler version.
- *runtime* – This element specifies a runtime required by a component implementation.
- *programminglanguage* – This element identifies the programming language used to build the component. The name attribute defines the language, e.g. C, C++, Java, etc.
- *humanlanguage* – This element defines the human readable language for which the component was developed. The name attribute provides the language name, e.g. English, French, etc.

OSGroup

The OSGroup identifies several mandatory elements that describe dependencies of the software implementation (Figure 10.12).

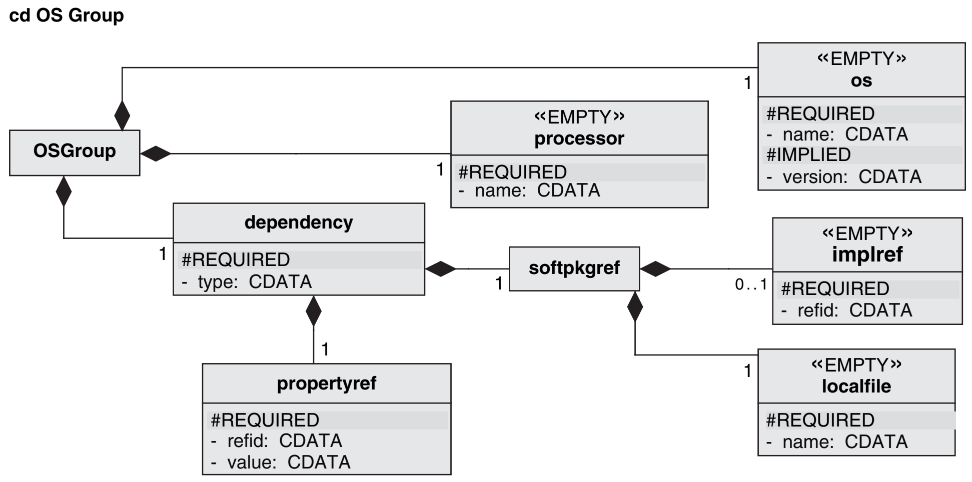


Figure 10.12. OSGroup element

The elements defined include:

- *os* – This element specifies the operating system required by the software component. The *name* attribute specifies the name of the operating system and the *version* attribute defines the operating system version. The *os* attributes are defined in a property file as an allocation property of string type and with names `os_name` and `os_version` and with an *action* element value other than ‘external’. The *os* element is automatically interpreted as a dependency and compared against allocation properties with names `os_name` and `os_version`. Legal `os_name` attribute values are listed in Attachment 2 of Appendix D of the specification.

- *processor* – This element specifies the *processor* and/or *processor family* required by the software component. The name attribute is defined in a property file as an allocation property of string type with a name of `processor_name` and an *action* element value other than ‘external’. The *processor* element is automatically interpreted as a dependency and compared against an allocation property with a name `processor_name`. Legal `processor_name` attribute values are listed in Attachment 2 of Appendix D of the specification.
- *dependency* – This element specifies dependency relationships between the components being delivered and other components and devices. The *propertyref* references a specific allocation property, using a unique identifier, and specifies the value used by a *Device* to satisfy an allocateCapacity operation. The *DomainManager* uses the dependencies to assure that components and devices necessary for proper operation of the implementation are present and available. The type attribute is descriptive information indicating the type of dependency.
- *softpkgref* – This element refers to a *softpkg* element contained in another Software Package Descriptor file and indicates a file-load dependency on that file. The file that the component is dependent on is specified by the name attribute of the *localfile* element. An optional *implref* element refers to a particular implementation-unique identifier, within the Software Package Descriptor of the other file.

The *usesdevice* element describes any ‘uses’ relationships this component has with a device in the system. The *propertyref* element references allocation properties, which indicate the *Device* to be used, and/or the capacity needed from the *Device* to be used.

10.3 Software Component Descriptor

The Software Component Descriptor (SCD) file describes the ports and interfaces of the software packages that reference the SCD.

The *softwarecomponent* element has several mandatory subelements and an optional *propertyfile* element (Figure 10.13).

The mandatory elements include:

- *corbaversion* – This element specifies the version of CORBA that the delivered component supports.
- *componentrepid* – This element uniquely identifies the interface that the component is implementing. The *componentrepid* may be referred to by the *componentfeatures* element.
- *componenttype* – This element describes properties of the component. For SCA components, the component types include resource, device, resourcefactory, domainmanager, log, filesystem, filemanager, devicemanager, namingservice, and eventservice.

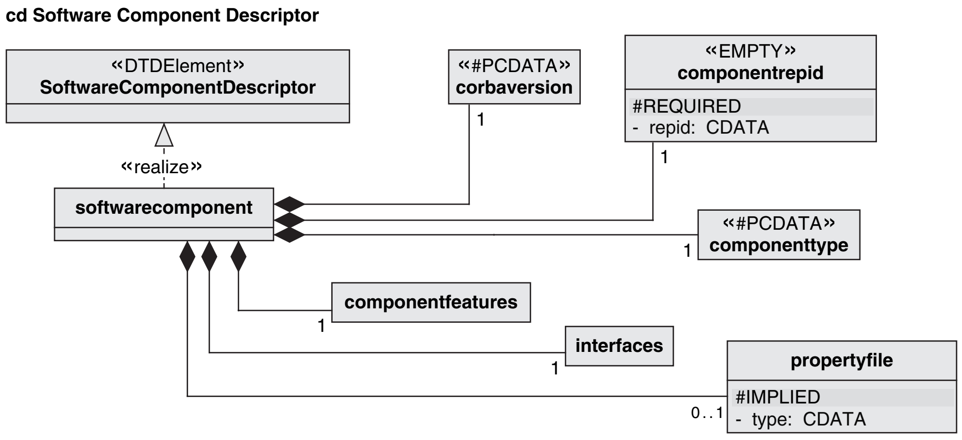


Figure 10.13. Software Component Descriptor elements

- *componentfeatures* – This element defines a component with respect to the components from which it inherits, the interfaces the component supports, and its provided and used *ports*.

  The *componentfeatures* element is further decomposed as shown in Figure 10.14.

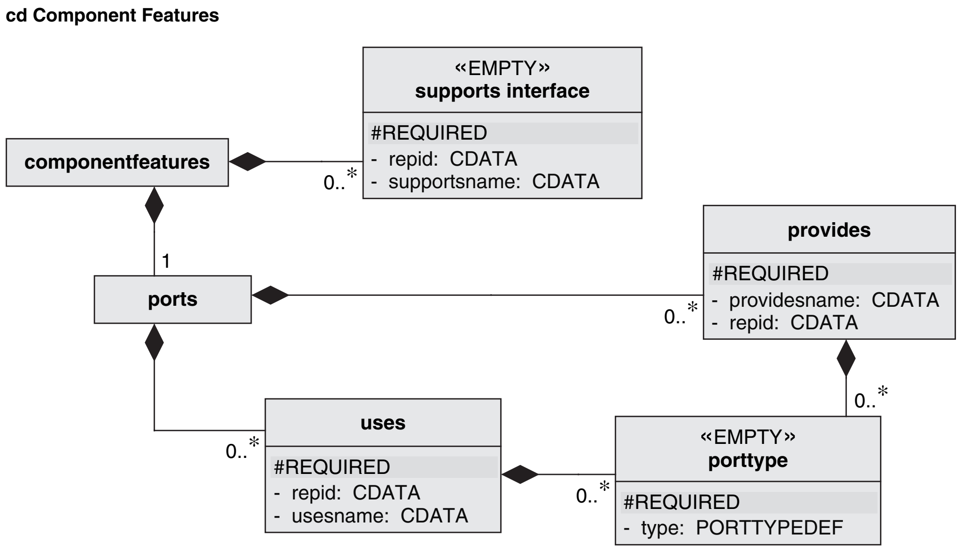


Figure 10.14. Component Features element

The optional *supportsinterface* element identifies an IDL interface that the component supports. These interfaces are distinct interfaces that were inherited by the component's specific interface. The repid is used to refer to the *interface* element.

The mandatory *ports* element defines the interfaces provided and used by a component. The *provides* elements are interfaces that are not part of a component's interface but are independent interfaces. The *uses* element is a *Port* interface type that is connected to a *provides* or *supportsinterface*. Any number of *uses* and *provides* elements can be given in any order. Each *ports* element has a name and references an interface by repid. The port names are used in the Software Assembly Descriptor to connect ports together. A *ports* element also has an optional *porttype* element that allows for identification of port classification. Values for *porttype* include 'data', 'control', 'responses', and 'test'. If a *porttype* is not given then 'control' is assumed.

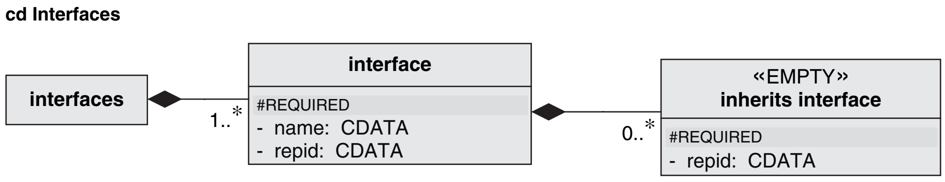


Figure 10.15. Domain Profile XML file relationships

The *interfaces* element contains one or more *interface* elements (see Figure 10.15). The name and repid attributes of the *interface* element are mandatory. Optionally, the *inheritsinterface* element may be defined to specify the interfaces that are inherited by the *interface* element.

10.4 Device Package Descriptor

The Device Package Descriptor (DPD) contains hardware device registration attributes, which are typically used by a Human Computer Interface application to display information about the device(s) resident in an SCA-compliant radio system. DPD information is intended to provide hardware configuration and revision information to a radio operator or to radio maintenance personnel. A DPD may be used to describe a single hardware element residing in a radio or it may be used to describe the complete hardware structure of a radio (see Figure 10.16).

The *devicepkg* is the top-level element of the physical radio system description. It has a unique identifier stored on the *id* attribute, a descriptive name stored on the *name* attribute, and an optional attribute, *version*, which provides version information regarding the device. The devicepkg may also provide descriptive information through the *title* element, the *author* element, and the *description* element.

The key element is the hardware device registration, *hwdeviceregistration*, element (see Figure 10.17). This element provides device-specific information and has *id*, *name*, and *version* attributes as well. The key information is provided through the sub-elements of the *hwdeviceregistration.* Basic descriptive information is provided through the *description*, *manufacturer*, *modelnumber*, and *deviceclass* elements which provide the information indicated by the element names. The *hwdeviceregistration* may have its own *propertyfile* to provide additional property definitions and values unique to the specific device.

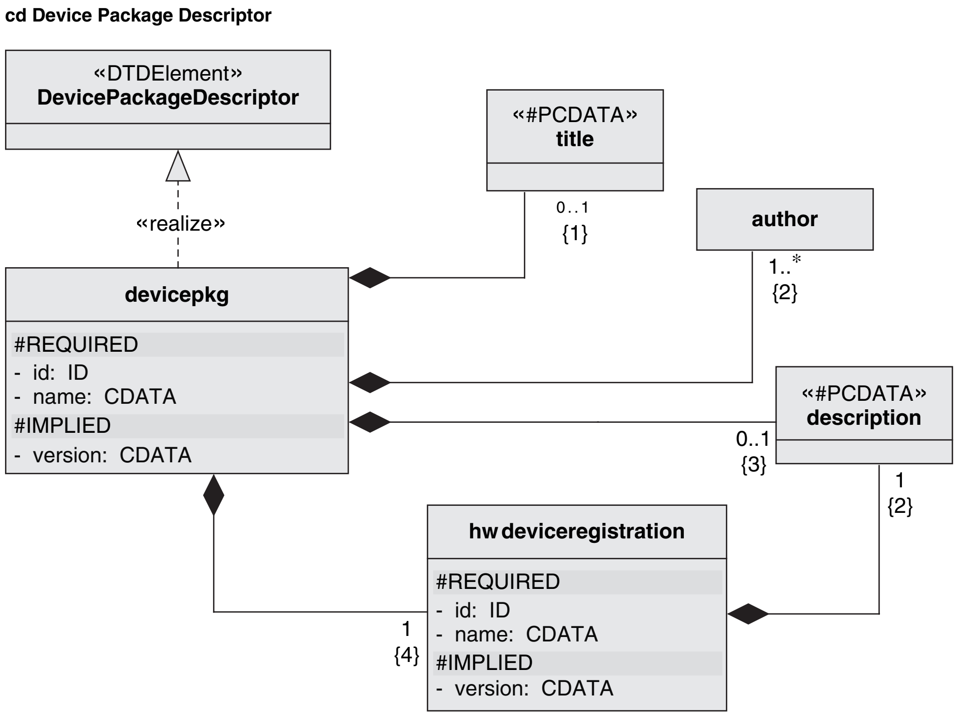


Figure 10.16. Device Package Descriptor elements

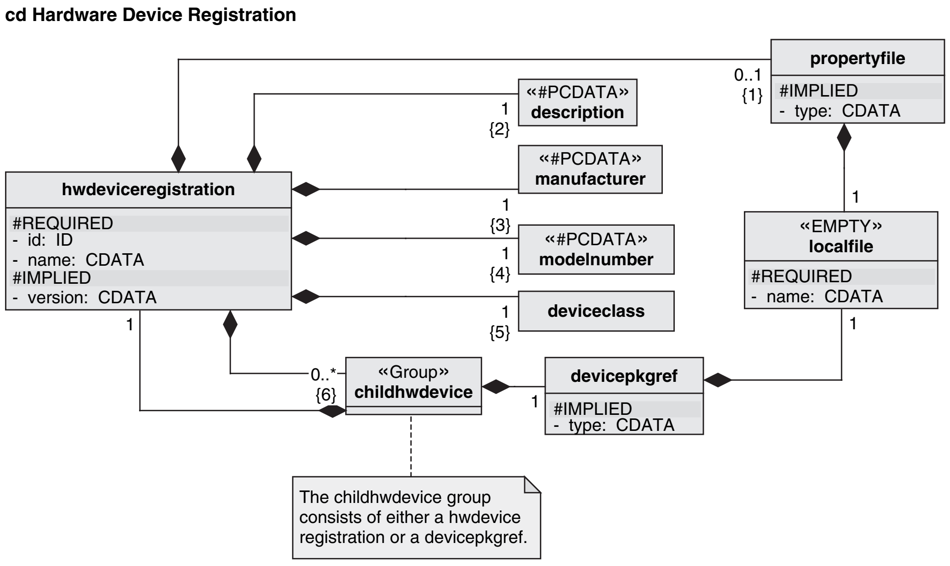


Figure 10.17. Hardware Device Registration element

If the *hwdeviceregistration* is comprised of multiple sub-devices or child devices, then there may be one or more *childhwdevice* elements defined. Each childhwdevice element consists of either a *hwdeviceregistration* element or a *devicepkgref* that provides the specific information for the child device. If the *childhwdevice* element consists of a *hwdeviceregistration*, then it may also contain *childhwdevice* elements. Thus, a hierarchical decomposition of the set of devices in a system may be represented.

11

Device Configuration Descriptor

11.1 Overview

The Device Configuration Descriptor contains the information necessary to start-up one or more devices that comprise a node. There is at least one Device Manager within an SCA system associated with the initial startup or boot node. Other devices that are not part of the boot node may also be started.[1]

Figure 11.1 shows the top level components of the DCD file. The mandatory components of the DCD are the *devicemanagersoftpkg* and *domainmanager* elements. The *devicemanagersoftpkg* element provides the information necessary to load and execute the SCA DeviceManager implementation. The *domainmanager* element provides the information necessary to load and execute the DomainManager implementation.

The remainder of the elements are optional but, in most implementations, typically contain entries providing additional information regarding the configuration and execution of the DeviceManager. Elements that are usually included are the *filesystemnames, connections*, and *componentfiles*.

11.2 deviceconfiguration

The top-level element is the *deviceconfiguration* and it contains two attributes: *Id* and *name*. The *Id* attribute is mandatory and provides a unique identifier (UUID) for the DeviceManager. The *name* attribute is optional and provides a human readable name for the DeviceManager.

[1] *Only one Device Manager is necessary. However, depending on the complexity of the system, additional DeviceManagers can help to manage the complexity of the system providing logical subsets of the Devices that comprise the system. Having more than one DeviceManager is suited for SCA systems consisting of multiple hardware components such as a bank of signal processing components within a VME or other backplane. Having a DeviceManager associated with a card or Line Replaceable Unit (LRU) is advisable.*

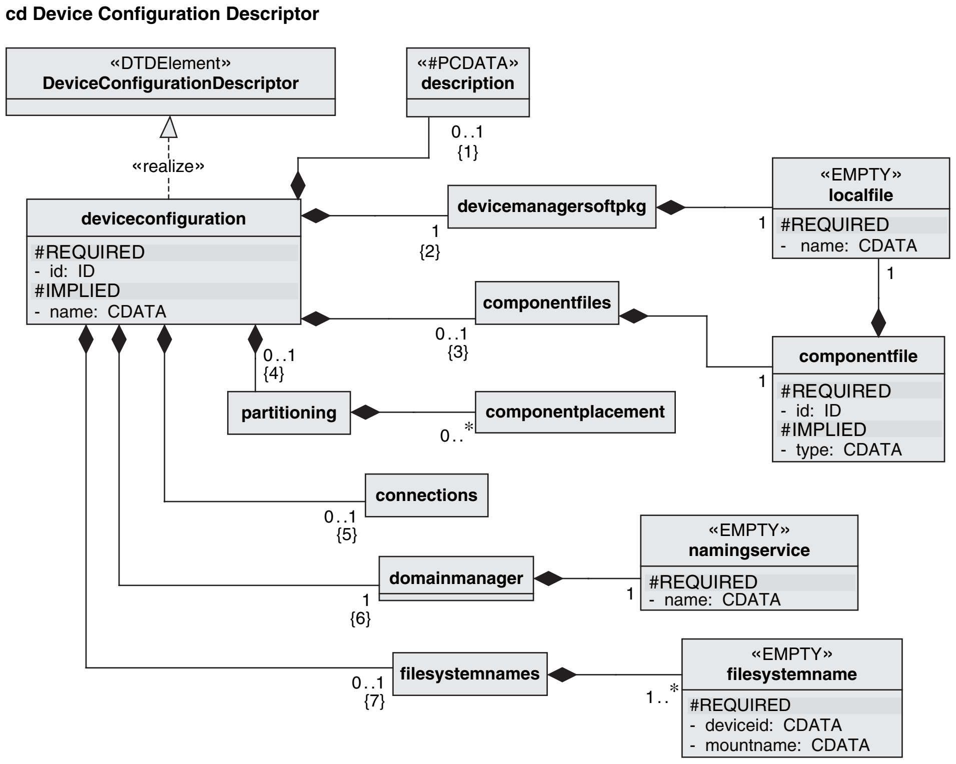


Figure 11.1. Device Configuration Descriptor components

11.2.1 description

The *description* element is optional and is used to provide a documentation, background, or other useful information useful to the developer or integrator of an SCA system using the DevicementManager implementation.

11.2.2 devicemanagersoftpkg

The *devicemanagersoftpkg* element specifies the SPD file that identifies the software to be loaded and run that implements the DeviceManager. The SPD information is specified within a *localfile* element. The name attribute of the *localfile* element contains the full pathname information to the SPD.

11.2.3 componentfiles

The *componentfiles* element is a construct to provide the ability to specify a set of one or more componentfile elements. If no componentfile elements are to be specified, then the componentfiles element may be omitted.[2]

[2] *In practice, the componentfiles element will typically be present because some Device or Service will be started as part of the DeviceManager.*

componentfile

Each *componentfile* element has a required Id attribute providing a unique identifier for the element (Figure 11.2). The type attribute on the componentfile element provides information about the component.

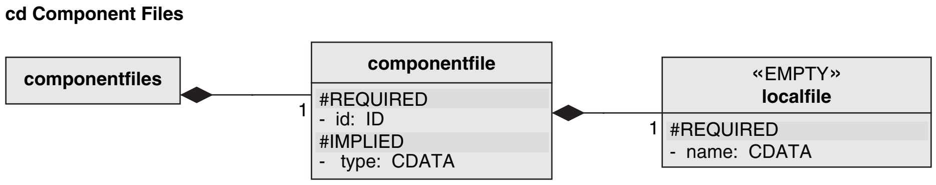


Figure 11.2. componentfiles element

Each *componentfile* element contains one *localfile* element that references a SPD file that describes the software to be started as part of the DeviceManager startup.

The SPD file may refer to a Device, DeviceManager, DomainManager, Naming Service, FileSystem, or other software component to be started as part of the DeviceManager.

11.2.4 partitioning

The *partitioning* element is simply used as a construct to describe a set of one or more *componentplacement* elements. If no *componentplacement* elements are required, then the *partitioning* element may be omitted.

componentplacement

The *componentplacement* element specifies deployment information and constraints for a component (Figure 11.3). There are two mandatory elements of the *componentplacement*: the *componentfileref* and *componentinstantiation* elements. The *deployondevice*, *compositepartofdevice*, and *devicepkgfile* elements are optional.

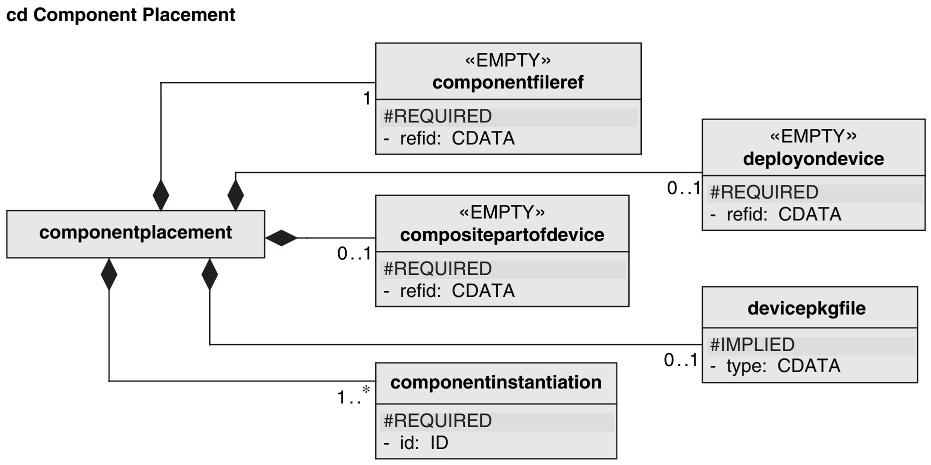


Figure 11.3. componentplacement element

componentfileref
The *componentfileref* element has a single, mandatory attribute, refid, that specifies the Id attribute of a componentfile element. This provides the reference to the component file that is to be deployed.

deployondevice
The optional *deployondevice* element indicates the Device on which the *componentinstantiation* element is deployed. The refid attribute refers to the Id of a Device.

compositepartofdevice
The optional *compositepartofdevice* element indicates that the component is part of an AggregateDevice. If present, the refid attribute references the Id of the AggregateDevice of which the component is a part. The refid attribute is mandatory.

devicepkgfile
If the component is a Device, the *devicepkgfile* element identifies the Device Package Descriptor (DPD) of the Device.

componentinstantiation
The *componentinstantiation* element identifies the component to be instantiated (Figure 11.4). The Id attribute is mandatory and provides a unique Id for the component instantiation. There must be at least one *componentinstantiation* element within the *componentplacement* element and there may be more than one.

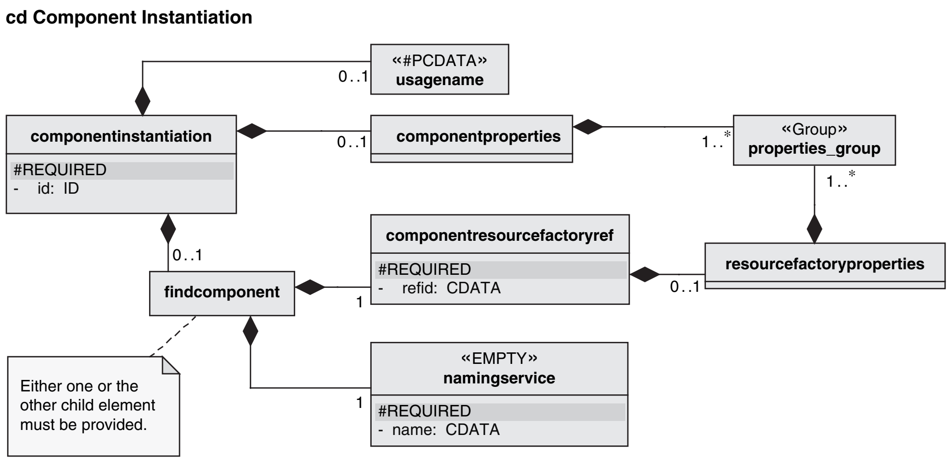


Figure 11.4. componentinstantiation element

The *componentinstantiation* provides information on the usage, properties, and method for locating a reference to the *componentinstantiation* by other components.

The *usagename* element provides an descriptive name for the component. If the component is a service type then the *usagename* element must be unique for each service type. The optional *componentproperties* element defines a list of property values used to configure the component upon instantiation.

The *findcomponent* element specifies how the component may be located by other components within the system and must contain either a *componentresourcefactoryref* or a *namingservice* element. If the *componentresourcefactoryref* element is specified then the component may be located through a ResourceFactory. If the *namingservice* element is specified, then the component may be located through the CORBA Name Service. The *componentresourcefactoryref* attribute refid identifies the Id of the ResourceFactory that created the component. The *componentresourcefactoryref* may have a set of properties defined using the *resourcefactoryproperties* element. The *namingservice* name attribute contains the name entered into the Name Service for the *componentinstantiation*.

11.2.5 connections

The *connections* element in the DCD specifies the services used by the DeviceManager and Device components within the DCD. The Core Framework DomainManager uses the information in the connections element, obtained when the DeviceManager registers with the DomainManager, to establish the connections specified.

11.2.6 domainmanager

The *domainmanager* element has a single, required subelement, *namingservice*, that specifies how to obtain the DomainManager object reference. The *namingservice* element has a single, required attribute, name, that specifies the lookup name within the Name Service.

11.2.7 filesystemnames

If the DeviceManager hosts a FileSystem, then the *filesystemnames* element specifies the mounted file system names. The FileSystem names specified are provided to the FileManager on the DomainManager as FileSystems to be mounted within the FileManager.

12

The Domain Manager Descriptor

12.1 Overview

The *DomainManager* Configuration Descriptor (DMD) XML file provides information regarding the startup, configuration, and operation of the DomainManager within the SCA radio system (Figure 12.1). The *domainmanagerconfiguration* element Id attribute is a uniqueidentifier for the DomainManager and the *name* attribute provides a descriptive name.

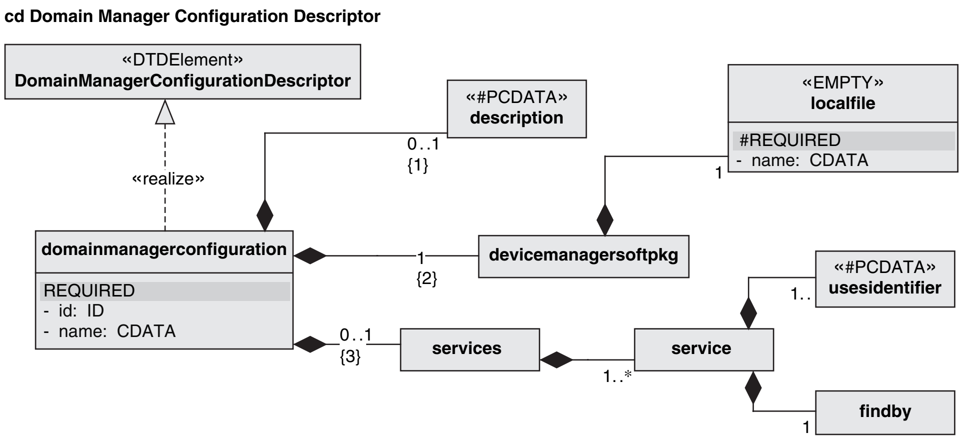


Figure 12.1. Domain Manager Descriptor elements

The optional description element provides information about the DomainManager implementation. The implementation information is provided in the *domainmanagersoftpkg* which is located via the information in the *localfile* element. The *domainmanagesoftpkg* may provide information about ports used by the DomainManager. These are used to connect to services used by the DomainManager and described in the services element. The services element contains one or more service elements.

For each service, the method by which the DomainManager will locate the service is specified through the *findby* element. There are two basic methods for locating the service specified. It may be located via the CORBA Name Service, in which case a *namingservice* element will be specified, or is may be located through the DomainManager itself, which is specified through the *domainfinder* element.

The *usesidentifier* element identifies the port that is provided by the service that the DomainManager will use. Each port referenced by a *usesidentifier* element for a service should be defined as a provides port by the service specified.

13

The Software Assembly Descriptor

13.1 Overview

The software assembly XML file is the top-level that describes an SCA application, commonly known as a waveform. The Software Assembly Descriptor (SAD) XML file describes the set of application components, their interconnection, and deployment of the components within the application. The component assembly provides four basic types of information processed by the DomainManager during installation (Figure 13.1). The first is partitioning information that indicates special requirements for the collocation of components, the second is the assembly controller for the software assembly, the third is connection information for the various components that make up the application assembly, and the fourth is the visible ports for the application assembly.

The *softwareassembly* has a unique ID stored on the *id* attribute and a readable name stored on the *name* attribute. In addition, the optional *description* element provides a multi-line explanation of the waveform assembly and objectives. The set of components that form the application are specified within the *componentfiles* element. The *partitioning* element provides information regarding any constraints or requirements regarding where components must be loaded, e.g. some components may utilize another component that is a library and therefore must be collocated on the same processor. The top-level routine that implements the control and assembly logic for the waveform is the AssemblyController and is specified in the *assemblycontroller* element. The AssemblyController is defined as one of the components in *componentfiles* element. It is identified as the AssemblyController through the *componentinstantiationref* which has a *refid* attribute containing the instantiation id of the component. In order to instantiate the waveform, the application factory must know what connections need to be established between the components defined in the *componentfiles* element. The *connections* element provides this information. Finally, the set of visible ports provided by the waveform are specified in the *externalports* element.

The set of components are specified within the *componentfiles* element as a set of one or more *componentfile* elements (Figure 13.2). Each componentfile element has a unique ID stored on the *id* attribute. The *type* attribute is set to “Software Package Descriptor.” The *localfile* element of the *componentfile* contains the reference to an SCA File containing the Software Package Descriptor for the component.

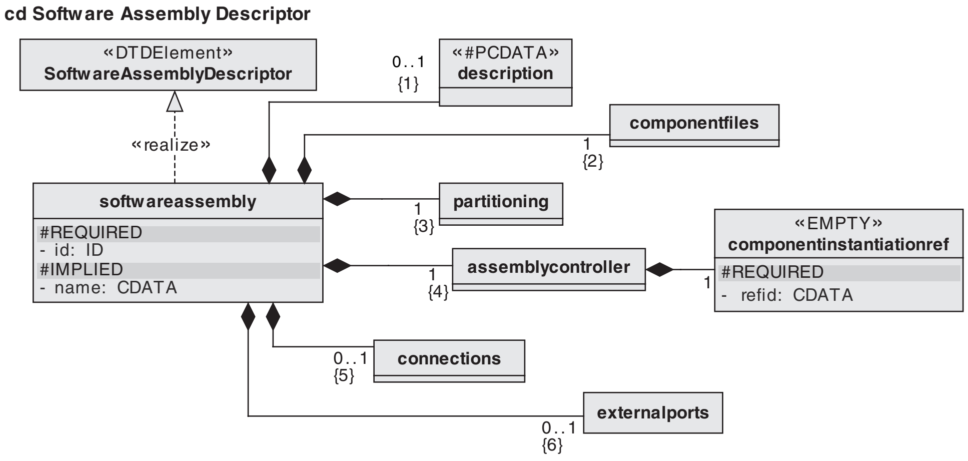


Figure 13.1. Software Assembly Descriptor

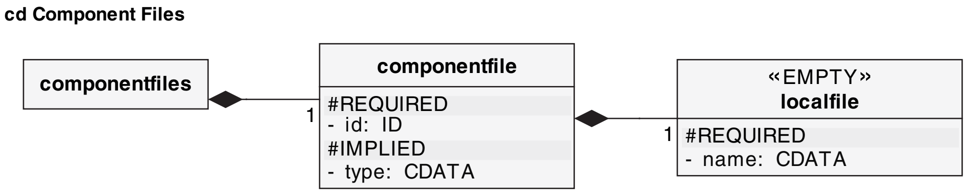


Figure 13.2. componentfiles element

The *partitioning* element (Figure 13.3) contains the set of constraints that guide where components are to be placed. There are two type of placement constraints. These are specified by the *componentplacement* and *hostcollocation* elements. The *hostcollacation* element is used to identify a set of components that must be deployed on the same host processor. It has a unique ID stored on the *id* attribute and a readable name stored on the *name* attribute. It contains a set of *componentplacement* elements that identify the components to be deployed on the same processor.

The *componentplacement* element identifies a component that may be deployed directly or by a ResourceFactory. The *componentfileref* element references a specific Software Package Descriptor file and the *id* attribute of the *componentfileref* matches the *id* attribute of the *componentfile* element. The *componentinstantiation* element of the *componentfileref* provides specific information regarding the instantiation of the component. The *id* attribute of the component may be referenced by the *usesport* and *providesport* elements within the SAD file.

The *componentinstantiation* element is composed of several sub-elements that describe the instantiation constraints of the component (Figure 13.4). The *usagename* element provides a descriptive name for the components. The *findcompnent* element describe how the CORBA reference to the component instance will be located. There are two basic methods for locating the reference and either one or the other is used but not both. The *namingservice* element

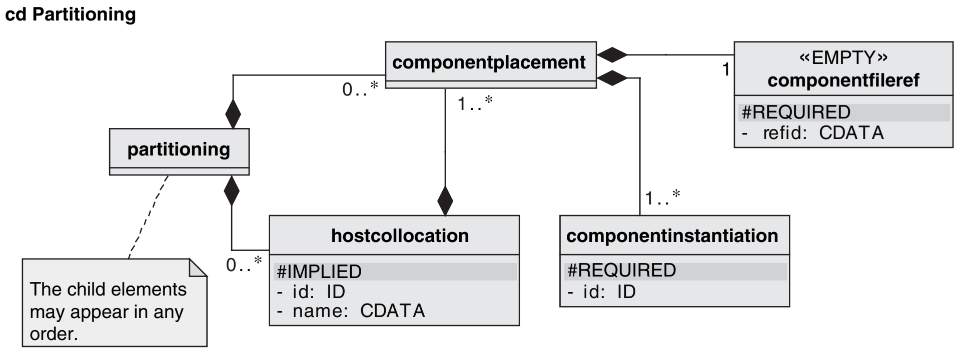


Figure 13.3. Partitioning element

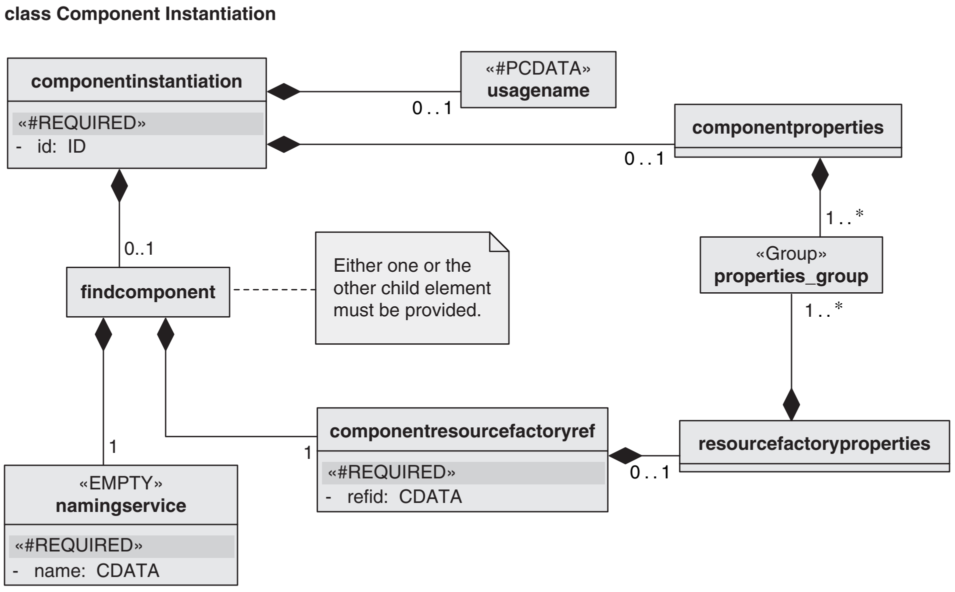


Figure 13.4. Component Instantiation element

provides a string name that is used by the ApplicationFactory to complete and find the component's CORBA reference. The *componentresourcefactoryref* specifies a reference to the id of the ResourceFactory defined in within the SAD which instantiates the component.

Both the *componentinstantiation* and *componentresourcefactoryref* elements may have properties associated with them. The *componentproperties* element defines the properties associated with the *componentinstantiation* element and the *resourcefactoryproperties* element defines the properties associated with the *componentresourcefactoryref* element.

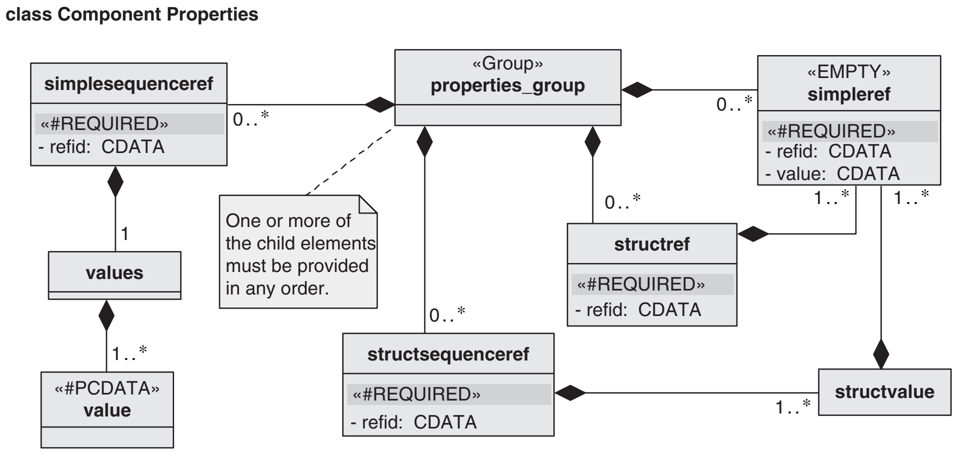


Figure 13.5. Component Properties elements

The *properties_group* contains the set of properties definitions for the component (Figure 13.5). The properties specify the factory, configuration, and execution parameters for the component.

The property type definitions correspond to the property types discussed in Chapter 10. What is important at this point is the precedence given to the search and assignment of values for these properties. For those attributes that are configure or execution, i.e. execparam, and the property may be modified, i.e. the mode is readwrite or writeonly, then the SAD is searched first for the property value to apply. Within the SAD, the precedence is the *componentinstantiation* element within the *partitioning* and *componentplacement* element hierarchy. If no value is found then the value or default value specified in the SPD is used, if available. If no property value is specified, then the property is ignored.

If the property is a factory parameter, i.e. factoryparam, then the the *resourcefactory-properties* element within the partitioning, *componentplacement*, *componentinstantiation*, *findcomponent*, *componentresroucefactoryref* element hierarchy. If no value is found then the *componentproperties* element will be searched within the *partitioning*, *componentplacement*, *componentinstantiation* element hierarchy. If no value is found within the *componentproperties* element, then the value provided in the SPD will be used. Finally, if no value is available, the property will be ignored.

The *connections* element, are noted earlier, identified the component connections that must be made to successfully deploy the application (Figure 13.6). The *connections* element contains zero or more *connectinterface* elements that specify individual connections provided or used by the component. The *usesport* element defines the port used by the component and is discussed in more detail below. The *connectinterface_grp* contains three possible elements, *providesport*, *componentsupportedinterface*, and *findby* of which one must be provided.

The *providesport* element describes a port provided by the component. The *componentsupportedinterface* describes another component which has a corresponding *supportsinterface* element defined in the Software Component Descriptor, see section 10.3. The *findby* element describes how the interface reference may be located. The interface may either be found in the CORBA Name Service using the value provided on the *name* attribute

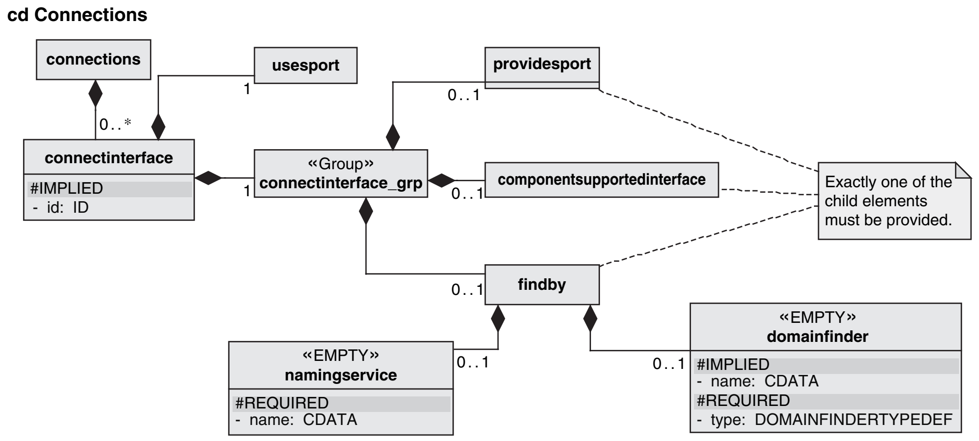


Figure 13.6. Connections element

of the *namingservice* element or may be found within the DomainManager using the values of the *name* and *type* attributes on the *domainfinder* element.

The *usesport* element (Figure 13.7) contains several sub-elements. The *usesidentifier* element specifies the specific uses port that to be used in the connection to a provides port defined and available on another component. The *usesport_grp* contains four methods for referencing the port. Only one of the element in the group may be used. The

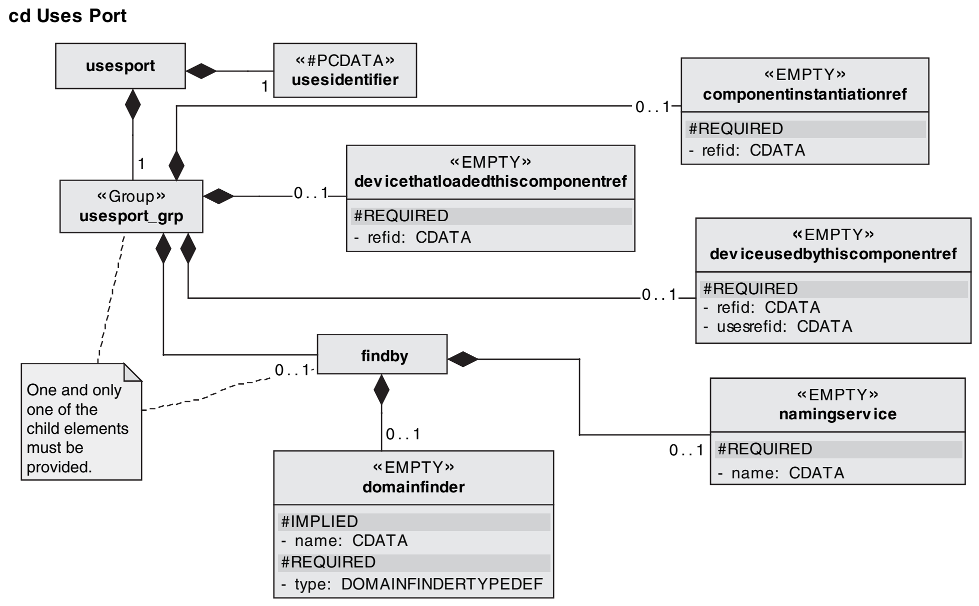


Figure 13.7. usesport element

componentinstantiationref identifies a component instantiation within the assembly using the *refid* attribute which corresponds to the instantiation id of the other component. The *findby* element, as described previously, specifies a method for locating the component either through the naming service or DomainManager. The *devicethatloadedthiscomponentref* element identifies a specific component in the assembly using the *refid* attribute. The device that loaded the component referenced is then associated with this component. The *deviceusedbythiscomponent* identifies a component within the assembly that is used to obtain the Device used by the component referenced.

The *providesport* element (Figure 13.8) is similar to the *usesport* element in content. The difference is the intent of the port. It has a *providesidentifier* element that is used to identify and locate the port. The *providesport_grp* has four elements, *componentinstantiationref*, *findby*, *devicethatloadedthiscomponent*, and *deviceusedbythiscomponent*. These elements function as described above for the *usesport* element.

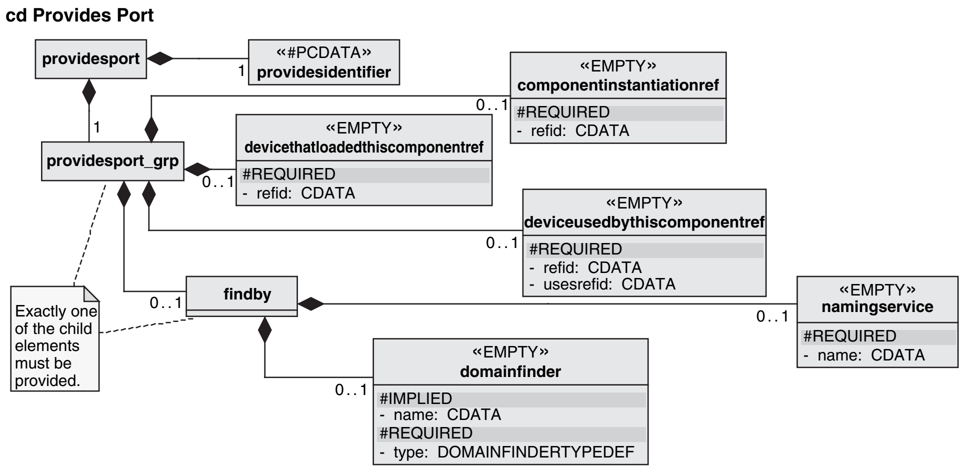


Figure 13.8. providesport element

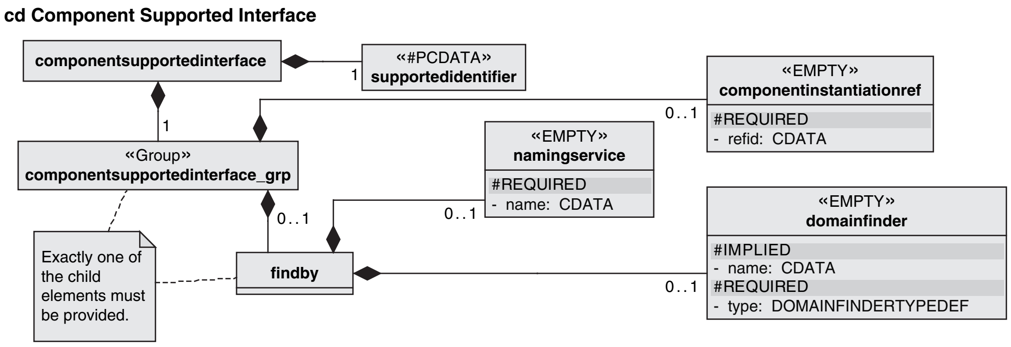

cd Component Supported Interface
componentsupportedinterface
«#PCDATA» supportedidentifier
1
1
«EMPTY» componentinstantiationref
#REQUIRED
- refid: CDATA
0..1
«Group» componentsupportedinterface_grp
«EMPTY» namingservice
#REQUIRED
- name: CDATA
0..1
0..1
«EMPTY» domainfinder
#IMPLIED
- name: CDATA
#REQUIRED
- type: DOMAINFINDERTYPEDEF
0..1
Exactly one of the child elements must be provided.
findby

Figure 13.9. componentsupportedinterface element

The *componentsupportedinterface* element (Figure 13.9) identifies a component interface that can be used by a uses port, i.e. it can be connect as a provides port. The *supportedidentifier* element identifies the port. The *componentsupportedinterface_grp* contains two sub-elements, *componentinstantiationref* and *findby*. These elements function as described above for the *usesport* element.

Finally, the *externalports* element (Figure 13.10) is used to identify port that are externally visible, i.e. may be accessed and connected to by external software. The *port* element is used to identify each externally visible port. The *refid* attribute of the *componentinstantiationref* element provides the reference for connecting to the port by an external software application. The optional *description* element contains textual description of the port, it's intent, and purpose. The *port_grp* has three sub-elements, *usesidentifer*, *providesdentifier*, and *supportedidentifier* elements. Each is a port identifier corresponding to the type of port that is being defined. Only one of these elements must be provided.

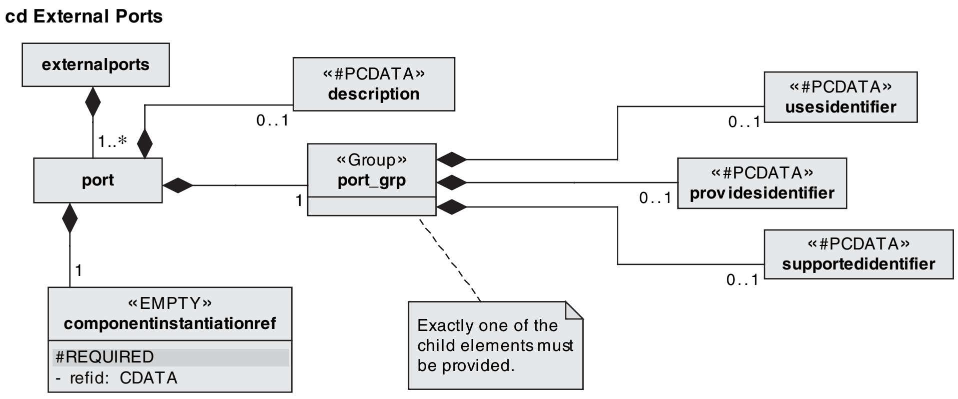

cd External Ports
externalports
«#PCDATA» description
0..1
1..*
port
«Group» port_grp
1
«#PCDATA» usesidentifier
0..1
«#PCDATA» providesidentifier
0..1
«#PCDATA» supportedidentifier
0..1
1
«EMPTY» componentinstantiationref
#REQUIRED
- refid: CDATA
Exactly one of the child elements must be provided.

Figure 13.10. externalports element

This concludes the discussion on the Domain Profile. In Part III, immediately following, examples will be provided to help visualize the application and use of these XML files.

PART III

Building an SCA-Compliant System

John Bard

Several open source implementations of an SCA Core Framework are available for use as a learning tool in conjunction with this book. The examples provided in Part III use the SCARI Core Framework, a Java open source implementation. This does not represent a blanket endorsement of SCARI for any particular application or project. There are a number of available open source and commercial implementations of the SCA. The implementation that is appropriate for your project must be chosen based on the project requirements.

14

The POSIX Operating System

In Part III we build up a series of examples that provide the reader with tangible evidence as to the capabilities offered by an SCA Operating Environment (OE). Recall that an SCA OE consists of a POSIX operating system, a minimum CORBA ORB, a Core Framework, and certain CORBA-based Services. Since SCA-compliant software radios that are capable of interfacing the ether are not widely available, we shall make use of the abstractions offered by the SCA. An SCA *Device* does not need to be a modem or an RF power amplifier. It could be an Internet appliance, a serial port, or a sound card. The sound card is particularly attractive because it has Analog-to-Digital and Digital-to-Analog converters just like a software radio. No matter what physical device is overlaid by an SCA *Device* the concept remains the same. If anything the reader should understand by now that the SCA really has nothing to do with software radios at all. It is a generic construct that can be applied to any distributed system that has applications contending for limited resources. Subsequent chapters will explore, by code example, features of each OE element available to the applications programmer.

14.1 An Operating Environment

From Chapter 1, Figure 1.6, the applications programmers must follow a few rules in accessing the layers of middleware upon which the application resides. Briefly, an application or a Core Framework's Base Application Interface is allowed to access the Operating System (OS) only through POSIX interfaces denoted as mandatory in the SCA Appendix B. An applications programmer is allowed to access the ORB only through interfaces described in the minimum CORBA specification [1]. An example of a disallowed ORB interface would be anything from the Dynamic Invocation Interface (DII) vocabulary. The applications programmer is allowed to access the IDL interfaces of the Naming Service, Event Service, and Log Service. Finally the applications programmer is allowed to access and inherit from any of the Core Framework interfaces identified in the SCA Appendix C. As a footnote, in SCA 3.2.1.1, the applications programmer 'shall perform file access through the CF *File* interfaces'. The list of mandatory POSIX interfaces (SCA Appendix B) provides all the means necessary to open, read, write, and close files but the applications programmer is disallowed the use of these interfaces. In many places in the SCA we see the escape clause 'if there are program performance requirements . . .'. This unfortunately does not apply to the use of POSIX file access versus its Core Framework counterpart.

We offer a view of SCA access constraints (Figure 14.1) that includes the definition of Radio Services. These are a concatenation of Logical *Devices* that compose and offer a portable means of accessing a radio system. Separate from that we identify device drivers. SCA allows access to devices at this most primitive level, 'if there are program performance requirements . . .'. It is recognized that such accesses are not portable and should be avoided. They are included in the SCA as means of propagating legacy. Without a doubt propagation of legacy and promotion of portability are somewhat in conflict.

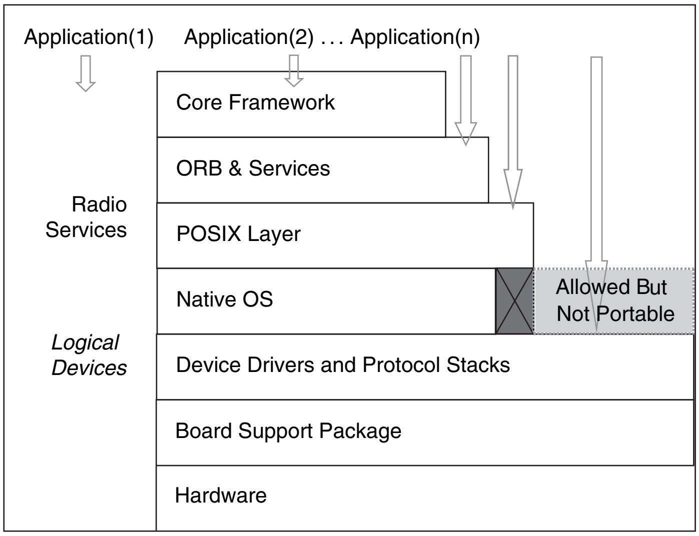


Figure 14.1. Access constraints, alternate view

Beyond the POSIX layer, application programmers who wish to be guided by the requirements of the SCA must have proficiency in CORBA. In this exploration of the SCA code, examples will be loaded with CORBA and that, as they say, is the nature of the beast. Many fine texts exist on CORBA and in applying CORBA to the SCA and software radio the developer is forced to have much more than just a passing acquaintance with CORBA. Subsequent chapters provide a view of CORBA sufficient to gain access to and utilize the power of the SCA Core Framework and Services.

The code examples are intended to illuminate aspects of the SCA that every applications programmer needs to understand. Great care has been taken to ensure that every code example compiles and executes properly. Readers are invited to code up the examples within their own environments and thus complete the hands-on experience. In providing these code examples three caveats are offered.

First, the SCA allows the developer a great deal of freedom in authoring SCA-compliant applications. Great care was taken by the authors of the SCA to use the architecture to sufficiently constrain the developer so as to foster portability and reuse of the developed product. However, the authors of the SCA wisely did not want to end up telling developers how to do their jobs; nor did they want overly to constrain the architecture so as to invalidate hundreds of thousands of lines of legacy radio software that predates the SCA. As many students of the SCA have found out, it is possible to write one or two CORBA wrappers around an existing application, drop it onto a compliant Operating Environment, and satisfy completely the requirements of the SCA. Whether the resultant implementation is portable or whether the implementation satisfies the spirit of the SCA is inconsequential. The fact is that the SCA fosters reuse and this is a legitimate example of reuse. The SCA grants significant latitude to developers by not specifying exactly how to go about their business. To look at this from the other direction, there is more that one way to use the SCA to skin your cat. This book offers just one approach, an approach selected for its educational merit and simplicity. This segues in to the second caveat.

The code segments offered are not bullet-proofed for the real world. Of note is the fact that the Core Framework defines 45 different exception types and in some cases identifies corrective actions that should be taken. A real world software radio implementation is required to survive and recover gracefully from these events should they occur. Because of the asynchronous nature of distributed computing systems, exceptions are not necessarily fatal. Perhaps a remote component was just not ready at a particular instant to perform a particular function. The client software might re-attempt the invocation of the remote interface just a few hundred milli-seconds later and everything is then OK. Therein lies some of the code complexity typical of these kinds of real-time distributed deployments. For the sake of clarity the real-time constructs necessary to support graceful exception handling are not employed in the code examples. This is not to say that exception handling, or at least exception detection, is ignored in the code examples. As a matter of fact most invocations of remote interfaces in the code examples are checked for the occurrence of an exception. What's missing is the graceful recovery software that allows the run-time to stay alive in order to react appropriately. In the code examples detection of an exception is promptly reported on the standard error output stream and then the program is aborted without grace.

The final caveat offered in consideration of the code examples is that they were written to address the requirements of this book. The SCA's incorporation of CORBA allows the applications programmer freedom from such concerns as collocation, transport methods, security, operating system, or native language types. At the source code level the CORBA application programmer need not worry about whether a component is local or remote. The component might be in the same compilation space, same thread – just a function call. The component might be in a separate process on the same processor or executing in another language on a remote processor. A well-written CORBA application will execute successfully without modification in all these scenarios. These degrees of independence exist at the source code level only. When one considers the software at the system level a programmer cannot ignore the laws of physics. For example two components within a WiMax application are required to complete a transaction within so many tens of micro-seconds. This requirement has direct implications regarding collocation, operating system, etc. Most importantly the source code implementation must meet this transactional requirement 100 % of the time.

Once it's understood that CORBA allows an implementation to exist in a run-time neverland, it follows that the successful developer must craft the implementation so as to abide by the laws of physics as it applies to the task at hand. These overarching system requirements begin to curb the degrees of freedom offered by CORBA. When all aspects of a project are considered, selection of operating systems, transport methods, ORBs, and Core Frameworks might be reduced to a small subset within the overall distributed computing universe. Developers and development houses tend to remain loyal to certain products for a bunch of good reasons. For some the choice of a native language or a particular ORB is akin to religion and not to be discussed. In respect of these mores we choose not to enter into discussions on the relative merits of one ORB versus another or the strong type checking of C++ versus the platform independence of Java. The third caveat is just this. The ORB, operating system, Core Framework, and native language selected for this book are based two requirements – low cost and wide spread availability.

The code examples presented in the following chapters are based on the elements shown in Table 14.1. Collectively these elements compose the Operating Environment.

Table 14.1. Baseline Operating Environment

Operating Environment Element	Identification
Operating System	Linux 2.6.9
Object Request Broker	Orbit2 2.13.3 (latest is 2.14)
Transport	Glib 2.10.1 (latest is 2.12)
Native Language	‘C’ gcc 3.4.4, glibc 2.3.4
Core Framework	SCARI2

The selected OE allows the motivated reader to acquire freely the elements needed to experientially follow the examples. A good question to ask at this time is: ‘is our Operating Environment SCA-compliant?’

14.2 Linux 2.6 Kernel

The Linux 2.6.9 operating system is perhaps the most substantive overhaul to the Linux kernel since its widespread adoption. A focus of the overhaul was to improve run-time performance. Linux had its genesis as a conventional non real-time, non pre-emptive version of Unix. In the last couple of years the focus, especially in the embedded community, has been on performance in the run-time. The 2.6 kernel dramatically improves run-time performance over its predecessors. The kernel was re-designed to minimize absolutely code sequences that lock out pre-emption. Most importantly the scheduler was re-designed to allow true pre-emption by privileged users. No more being ‘nice’ to all those other processes. We will briefly examine the 2.6 kernel, the pthreads and glibc libraries for compliance with the Appendix B of the SCA – the SCA Application Environment Profile (AEP) of the POSIX.1 standards. One of the difficulties in self-examination for SCA compliance is acquiring the commercial specifications that have become obsolete in the few short years since the SCA 2.2 was published in November 2001.

The SCA Appendix B cites the IEEE standards listed in Table 14.2 regarding the SCA’s mandatory use of POSIX.

The problem is that these standards are no longer available from the IEEE or ISO/IEC. Maintenance of the POSIX standards has passed to the Open Group which has made two major upgrades and a few minor updates to the 9945 standard in the last several years.

Table 14.2. SCA Appendix B POSIX citations

Standard	SCA AEP
C Standard (ISO/IEC 9899:1990	Partial
POSIX.1 (ISO/IEC 9945 -1):1997	Partial
POSIX.1b (ISO/IEC 9945 -1):1997	Partial
POSIX.1c (ISO/IEC 9945 -1):1997	Partial
POSIX.5b (IEEE 1003.5 - 1992)	Optional

Reference [2] is dated February 1997, the same year as the IEEE standards referenced within the SCA. This should be close enough to satisfy the intent of the SCA Appendix B.

A short word is necessary to describe the rather confusing relationship between the IEEE, the ISO/IEC, and the Open Group. A paper published in 2003 entitled 'The Single UNIX Specification' [3], clarifies this relationship. About a year after the publication of the Open Group's System Interfaces and Headers, Issue 5 [2], the three organizations began discussions to coordinate their standardization efforts. The culmination of their labors was the publication of nearly identical documents entitled Open Group Issue 6, IEEE 1003.1-2002, and ISO/IEC 9945:2002. The technical content of these documents is additionally coordinated with the authoritative 1999 work on ISO-C [4]. Though the works are coordinated, each organization maintains separate publications. As of early 2006 the three organizations are still working in coordination as the Austin Group [5]. It is likely that future versions of the SCA will reference these more universally supported and modernized standards.

There are about 300 functions or groups of functions listed in Appendix B of the SCA. Some are noted as 'mandatory', some as 'not required'. Recall that the basic purpose of specifying a certain set of mandatory POSIX system calls is to provide the applications programmer with a set of system functions guaranteed to be present on all SCA-compliant systems. There are two corollaries to this requirement. One, of course applies to Operating Environment suppliers. Despite the choice of native operating system – Windows, QNX, etc. – an additional POSIX layer will likely need to be developed to provide all of the mandatory POSIX interfaces. Many real-time operating systems attempt to reduce memory footprint and maximize run-time performance. It is understood that POSIX is not optimal for these factors. Many OS vendors offer POSIX as an option that can be added, if required, with the usual disclaimers about loss of run-time performance and increase in memory footprint. For an OS to be SCA-compliant, at least the mandatory functions must be present. The whole footprint issue is not nearly as critical now as it was 10 years ago. Then we had 40 MHz processors with 8 Mbytes of memory unarguably constrained by today's standard of GHz processors with Gigabytes of memory.

The second corollary applies to the applications programmer. For the application to be SCA-compliant all system calls are restricted to the set of mandatory POSIX calls. An application that uses native OS calls or even optional POSIX calls is considered not to be SCA-compliant. The reason, of course, is portability. Porting of an application from one OS to another is facilitated when system calls of both are restricted to a certain guaranteed subset.

All the mandatory POSIX function calls are to be found in the header files listed in Table 14.3.

Table 14.3. Required POSIX Header Files

1.	aio.h	11.	setjmp.h
2.	ctype.h	12.	signal.h
3.	dirent.h	13.	stdio.h
4.	fcntl.h	14.	stdlib.h
5.	locale.h	15.	string.h
6.	math.h	16.	sys/mman.h
7.	mqueue.h	17.	sys/stat.h
8.	posix_opt.h	18.	time.h
9.	pthread.h	19.	unistd.h
10.	semaphore.h	20.	utime.h

A top level check for SCA compliance is simply the presence of these header files somewhere in the include path. There might be some variation on the names of the header files but since POSIX has been around for some 20 years – an eternity for a computer technology – these names are pretty standardized. If you are developing a Core Framework then the mandatory POSIX requirement also applies to Base Application Interfaces – Port, LifeCycle, TestableObject, PortSupplier, PropertySet, Resource and ResourceFactory. *Device* software and device implementations are exempt from the requirement. Native OS calls are fair game. If you can take the performance 'hit', use of the mandatory POSIX calls is advised for two reasons: they are guaranteed to be there and it enhances the portability and re-use.

A more definitive test for POSIX compliance is offered in the code sequence of Figure 14.2.

```
/* testThread.c */
#include <pthread.h>
#include <bits/posix_opt.h>

int main()
{
  pthread_mutexattr_t myMutexAttr;
  int retVal;

  printf("_POSIX_ASYNCHRONOUS_IO %d\n",_POSIX_ASYNCHRONOUS_IO);
  printf("_POSIX_MAPPED_FILES %d\n",_POSIX_MAPPED_FILES);
  printf("_POSIX_MEMLOCK  %d\n",_POSIX_MEMLOCK);
  printf("_POSIX_MEMLOCK_RANGE %d\n",_POSIX_MEMLOCK_RANGE);
  printf("_POSIX_MEMORY_PROTECTION %d\n",_POSIX_MEMORY_PROTECTION);
  printf("_POSIX_MESSAGE_PASSING %d\n",_POSIX_MESSAGE_PASSING);
  printf("_POSIX_PRIORITIZED_IO %d\n",_POSIX_PRIORITIZED_IO);
  printf("_POSIX_PRIORITY_SCHEDULING %d\n",_POSIX_PRIORITY_SCHEDULING);
  printf("_POSIX_REALTIME_SIGNALS %d\n",_POSIX_REALTIME_SIGNALS);
  printf("_POSIX_SEMAPHORES %d\n",_POSIX_SEMAPHORES);
  printf("_POSIX_SHARED_MEMORY_OBJECTS %d\n",_POSIX_SHARED_MEMORY_
    OBJECTS);
  printf("_POSIX_SYNCHRONIZED_IO %d\n",_POSIX_SYNCHRONIZED_IO);
  printf("_POSIX_TIMERS %d\n",_POSIX_TIMERS);
  printf("_POSIX_FSYNC %d\n\n",_POSIX_FSYNC);

  printf("_POSIX_THREADS %d\n",_POSIX_THREADS);
  printf("_POSIX_THREAD_SAFE_FUNCTIONS %d\n",_POSIX_THREAD_SAFE_
    FUNCTIONS);
```

```
      printf("_POSIX_THREAD_PRIORITY_SCHEDULING
                                    %d\n",_POSIX_THREAD_PRIORITY_
        SCHEDULING);
      printf("_POSIX_THREAD_ATTR_STACKSIZE %d\n",_POSIX_THREAD_ATTR_
        STACKSIZE);
      printf("_POSIX_THREAD_ATTR_STACKADDR %d\n",_POSIX_THREAD_ATTR_
        STACKADDR);
      printf("_POSIX_THREAD_PROCESS_SHARED %d\n",_POSIX_THREAD_PROCESS_
        SHARED);
      printf("_POSIX_THREAD_PRIO_INHERIT %d\n",_POSIX_THREAD_PRIO_
        INHERIT);
      printf("_POSIX_THREAD_PRIO_PROTECT %d\n",_POSIX_THREAD_PRIO_
        PROTECT);
  return 0L;
}
```

Figure 14.2. Code sequence: test for supported POSIX features

It shouldn't be necessary to add in any special libraries at link time: everything should be in the standard path for glibc. Compile and execute the code as follows:

```
gcc testThread.c
./a.out
```

Depending on your system the output in Table 14.4 is produced. We've added a column for 'MANdatory' versus 'Not ReQuired' straight from Appendix B of the SCA.

The output produced by the code segment in Table 14.4 includes all of the POSIX.1b and POSIX.1c options listed in the SCA Appendix B. There are several features supported by Linux that are not required by the SCA. Supported but unnecessary functions include mapped files, memory protection (actually protection of mapped files), prioritized IO, priority scheduling, and shared memory objects.

Table 14.4. POSIX support for Linux 2.6.9 (RedHat Enterprise Edition)

POSIX Feature	Version	SCA Appendix B
_POSIX_ASYNCHRONOUS_IO	200112	MAN
_POSIX_MAPPED_FILES	200112	NRQ
_POSIX_MEMLOCK	200112	MAN
_POSIX_MEMLOCK_RANGE	200112	MAN
_POSIX_MEMORY_PROTECTION	200112	NRQ
_POSIX_MESSAGE_PASSING	200112	MAN
_POSIX_PRIORITIZED_IO	200112	NRQ
_POSIX_PRIORITY_SCHEDULING	200112	NRQ
_POSIX_REALTIME_SIGNALS	200112	MAN
_POSIX_SEMAPHORES	200112	MAN
_POSIX_SHARED_MEMORY_OBJECTS	200112	NRQ
_POSIX_SYNCHRONIZED_IO	200112	PRT
_POSIX_TIMERS	200112	MAN
_POSIX_FSYNC	200112	PRT

Table 14.4. Continued

POSIX Feature	Version	SCA Appendix B
_POSIX_THREADS	200112	MAN
_POSIX_THREAD_ATTR_STACKADDR	200112	MAN
_POSIX_THREAD_ATTR_STACKSIZE	200112	MAN
_POSIX_THREAD_PRIO_INHERIT	-1	MAN
_POSIX_THREAD_PRIO_PROTECT	-1	MAN
_POSIX_THREAD_PRIORITY_SCHEDULING	200112	MAN
_POSIX_THREAD_PROCESS_SHARED	-1	NRQ
_POSIX_THREAD_SAFE_FUNCTIONS	200112	PRT

Though priority scheduling is not required at the process level, i.e. `_POSIX_PRIORITY_SCHEDULING` not required, it is required for threads – `_POSIX_THREAD_PRIORITY_SCHEDULING` is mandatory. No mandatory support for priority scheduling eliminates the following eight system calls:

1. `sched_setparam()`
2. `sched_setscheduler()`
3. `sched_yield()`
4. `sched_get_priority_min`
5. `sched_getparam()`
6. `sched_getscheduler()`
7. `sched_get_priority_max()`
8. `sched_rr_get_interval()`

Since mapped files, memory protection, and shared memory are not required, according to the following eight system calls are also not required [6]:

9. `mmap()`
10. `shm_open()`
11. `shm_unlink()`
12. `mprotect()`
13. `munmap()`
14. `shm_close()`
15. `ftruncate()`
16. `msync()`

This leaves the following features also not required: `PRIORITIZED_IO` and `THREAD_PROCESS_SHARED`. There is no system call associated with `PRIORITIZED_IO`. Whether `PRIORITIZED_IO` is defined or not affects only the behavior of asynchronous IO, which is mandatory. With `PRIORITIZED_IO and PRIORITY_SCHEDULING` not required, prioritized asynchronous IO also cannot be expected to be supported on SCA-compliant systems.

It is understood that threads are independently executing sequences within the same process space. A process space is an independently executing sequence that is isolated from other processes. Say, for instance, you write a program to overwrite memory with zeros. In a process-oriented operating system like Linux the only one you can hurt is yourself. If you attempt to overwrite memory that doesn't belong to you the operating system steps in and prevents the attempt. Your process is then terminated by the operating system with a message describing some kind of memory access error. With an OS that doesn't support processes – like VxWorks – there is nothing to prevent an errant task from wiping out all memory. Now if two processes are completely isolated from each other how do they communicate? The answer, of course, is through special operating system calls – take your pick, message queues, semaphores, etc. Of course, in an SCA-compliant system `SHARED_MEMORY` is not required so that method is disallowed to the applications programmer. Now back to discussion of the not-required `THREAD_PROCESS_SHARED` functions. If you have threads in two separate processes they can't 'see' each other. In a system that is `THREAD_PROCESS_SHARED`-enabled, threads from two separate processes could share a conditional variable or mutex and thus be able to synchronize. This capability is not supported by Linux and also not required by the SCA. As a consequence the following four system calls are eliminated:

17. `pthread_condattr_getpshared()`
18. `pthread_condattr_setpshared()`
19. `pthread_mutexattr_getpshared()`
20. `pthread_mutexattr_setpshared()`

From Table 14.4 there is a capability required of SCA-compliant systems not supported by glibc 2.3.4 and that is `THREAD_PRIO` inheritance and protection. The 2.3.4 version of glibc uses the Native POSIX Thread Library (NPTL) for Linux. In terms of POSIX compliance, NPTL is a huge improvement over the previously used linuxThreads. So what capability is lost without the `THREAD_PRIO` support? In a supported system a mutex can be initialized with `THREAD_PRIO_INHERIT` or `THREAD_PRIO_PROTECT`. What's implemented on that mutex is a priority inheritance mechanism. Priority inheritance is a mechanism used to avoid priority inversions.

A priority inversion is a potentially fatal event that occurs when a resource is locked by a low priority thread and a high priority thread requires access to that resource. The high priority thread will block until the low priority thread releases the resource. In and of itself this is not much of a problem, it happens all the time. Suppose however a medium priority thread becomes run-able. It will pre-empt the low priority thread and that resource can then stay locked for a long time. With the high priority thread blocked high priority things don't get done in time and the system dies. There are a couple of means of addressing the priority inversion problem. One, design your system so that low and high priority threads never contend for the same resource. The definition of low and high priority in this context is to be sure that threads that contend for a resource cannot be pre-empted by someone with priority that lies in-between the low and the high priority levels. This design action is something the applications programmer can exercise to make the whole priority inversion problem go away.

However for the un-motivated applications programmer certain operating systems offer a solution – priority inheritance. There is a school of thought [7] that indicates that 'priority inheritance is incompatible with reliable real-time system design', and that 'priority inheritance is neither efficient nor reliable'. If priority inheritance is offered by the operating system here is how it works. Essentially, the low priority thread is set to run temporarily at the priority level of the highest priority thread blocked on that resource. Suppose that while the low priority thread has locked a resource (mutex) a high priority thread becomes run-able and attempts to gain access to the resource. The priority-inheritance enabled OS will see this contention and upgrade the priority level of the low thread so that it runs at high priority. With the temporary priority upgrade in place no mid-level thread can get in the way. When the resource is released the OS will downgrade the low thread back to its original priority level. Although conceptually very simple, the actual mechanism to implement priority inheritance is quite complex. This complexity has to do with threads that have more than one mutex that can potentially be nested.

14.2.1 Unavailable POSIX Calls

Before examining what's available, let's finish figuring out what's missing, i.e. not required. Including the 20 not required functions discussed so far there are a total of 97 POSIX system calls not required. We will explore the capabilities that are lost, starting with the system calls normally associated with process creation and control:

21. `fork( )`
22. `execl ( )`
23. `execv ( )`
24. `execle ( )`
25. `execve ( )`
26. `execlp ( )`
27. `execvp ( )`
28. `times ( )`
29. `exit ( )`
30. `_exit ( )`
31. `getenv ( )`
32. `getpid ( )`
33. `getppid ( )`
34. `uname( )`
35. `sysconf ( )`
36. `sleep ( )`
37. `wait( )`
38. `waitpid ( )`
39. `assert ( )`

The calls not required, 21–27, are at the heart of the traditional UNIX system. The `fork()` system call creates a new process called the child process that is a clone of the process that initiated the `fork()` – the parent process. There are only small differences between the parent and child. The child has a unique process Id and returns a zero from the `fork()` system call. When the parent returns from the `fork()` system call the return value is the process identification (PID) of the child. The various flavors of `exec()` are typically used by the child process. A call to exec reloads the executable image of the caller with the image pointed to by a filename passed as a parameter in the exec system call. The entry point of re-loaded image is the standard 'C' entry point main(argc,argv). With these commands a child process sheds the identity of its parent and establishes its own path of execution. Admittedly this sequence of events is a little more contrived than `taskSpawn()` – the VxWorks means of creating tasks – but there are benefits, notably inheritance and independence. But alas in an SCA-compliant system such methods of initiating new independent paths of execution cannot be counted on as being mandatorily present on all platforms. This is despite a comment made in the Core Framework IDL for the ExecutableDevice interface, execute operation that 'The execute operation converts the input parameters (id/value string pairs) parameter to the standard argv of the POSIX exec family of functions'. This mandatory requirement applies to the form of parameter passing, i.e., argv style, and not to mandatory support for exec. The same argv requirement is also extended to application components in SCA 3.2.1.3.

Simply put the SCA shows no support for the process-oriented operating system model. Most likely this is due to the dominance of the VxWorks operating system during the 1990s for any sort of real-time embedded or distributed military system. VxWorks' claim to fame was its small footprint and speed. VxWorks, however, uses a flat memory model. Everyone has access to every memory location. Without having to deal with a memory management unit (MMU) or keeping track of complete environments for each task in the system VxWorks was a hit for single board computers of the time. The blessings of VxWorks during the 1990s became a scourge during the 2000s when the government customer started asking for high reliability, high security systems capable of certifications required for safety of flight – DO

178 [8] – and secure networking – High Assurance Internet Protocol Encryption (HAIPE). The flat memory model did not serve as reliably as a process-oriented system with isolated memory spaces.

The genesis of the SCA is to be found in the task-oriented, flat memory model of VxWorks. So what is available to the applications programmer in place of these calls? In the list of SCA-mandatory system calls, the only call capable of creating independent, distinct paths of execution is `pthread_create()`. Reference [9] shows that `pthread_create()` is as easy to use as `taskSpawn()`. (We will cover threads in more detail later.) Threads are subject to the limitations of the flat-memory model. Though the created thread has its own stack and independence of execution it is in the same process space as the creator. For a system with a flat-memory model that process space is the entire memory map. For operating systems that support multiple processes, `fork()` is off-limits but there is nothing to say that multiple processes can't be launched from a script and run in the background. The applications programmer can then establish lines of communications between the processes with semaphores, message queues, or signals all of which are mandatory.

The code snippets in Figures 14.3 and 14.4 illustrate the power of inter-process communication with processes launched from the shell. The example consists of two processes: the first launched to run in the background (Figure 14.3) and the second in the foreground (Figure 14.4). A POSIX message queue is used to provide communication between the two. The foreground process will send a sequence of numbers one at a time to the background task. The background task will add five to the number and send them back to the foreground task. The foreground task will then output the before and after versions of the numbers to the standard output.

```
/* back1.c */

#include <mqueue.h>
#include <sys/stat.h> /* for mode_t flags */
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

static const struct timespec tenthSecond = {0,100000000};

#define msgSize 4 /* 4 bytes = an int */

int main()
{
mqd_t myQueue;
const char queueName[]="/myQueueName";

mode_t allowOwner = S_IRUSR | S_IWUSR| S_IXUSR;
struct mq_attr someAttributes;
int putDataHere;

someAttributes.mq_flags = 0;
```

Figure 14.3. Bi-directional blocking message queues – background

```
someAttributes.mq_maxmsg = 1; /* nice bi-directional blocking */
someAttributes.mq_msgsize = msgSize;

/* housekeeping: clean up queues from previous crashes, etc */
mq_unlink(queueName);

/* Exclusively create a read/write, blocking queue */
myQueue = mq_open(queueName,O_RDWR|O_CREAT|O_EXCL,
allowOwner,&someAttributes);
if (myQueue == -1)
{
  perror("mq_open() problem");
  abort();
}
else
  fprintf(stderr,"Queue is open for business\n");

while (1)
{
  if (mq_receive(myQueue,(char *)&putDataHere,
          msgSize,(unsigned int *)NULL) != -1)
    {
    if (putDataHere==0) break;
    putDataHere += 5;
    if (mq_send(myQueue,(const char *)&putDataHere,4,0) == 0)
    nanosleep(&tenthSecond,NULL); /* let someone else run */
    else
    {
    perror("mq_send() problem");
    abort();
    }
 }
 else
 {
   perror("mq_receive() problem");
   abort();
 }
}

if (mq_unlink(queueName) == 0)
  fprintf(stderr,"Queue unlinked shutting down\n");
else
  perror("mq_unlink() problem");

return 0;

}
```

Figure 14.3. (Continued)

The name of the message queue, line 16, is how the other task will locate the queue. The queue is setup to hold one message that is four bytes long, lines 23 and 24. Queue names are persistent as long as the OS is running. This could lead to a problem with pollution of the namespace. Line 27 ensures that any queue of the same name is wiped out. This way line 30 creates a clean queue with the specific attributes we desire. We want a bi-directional queue that is blocking. Once in the while forever loop the background process will be blocked waiting for someone to put something into the queue – line 42. Once pulled from the queue the process will add five and write it back to the queue. With no further intervention the program will loop around and read from the queue the message just written. So in order to force a context switch and allow other programs to run – namely the foreground process – a nanosleep is inserted immediately after the modified integer is written back to the queue. Nanosleep is not the best choice, `sched_yield()` is somewhat better. The SCA Appendix B does not specifically call out `sched_yield()` as either mandatory or not required. It is considered part the POSIX real-time extensions like semaphores and timers. It is also considered part of POSIX threads. Finally, it has a VxWorks equivalent so it is likely to be found on most SCA-based systems. Now consider the foreground program (Figure 14.4).

```
1. /* fore1.c */
2.
3. #include <mqueue.h>
4. #include <sys/stat.h> /* for mode_t flags */
5. #include <time.h>
6. #include <stdio.h>
7. #include <stdlib.h>
8. #include <string.h>
9. #include <errno.h>
10.
11. static const struct timespec tenthSecond = {0,100000000};
12.
13. #define msgSize 4 /* 4 bytes = an int */
14.
15. int main()
16. {
17. mqd_t myQueue;
18. const char queueName[]="/myQueueName";
19.
20. int putDataHere, i;
21. int testData[]={12,2,7,0};
22.
23. /* Attach to the queue */
24. myQueue = mq_open(queueName,O_RDWR);
25. if (myQueue == -1)
26. {
27.     printf("mq_open() problem %s\n",strerror(errno));
```

Figure 14.4. Bi-directional blocking message queues – foreground

```
28.   abort();
29. }
30.
31. for (i=0; i<3; ++i)
32.{
33.   if ( mq_send(myQueue,(const char *)(testData+i),4,0) == 0 )
34.  {
35.     nanosleep(&tenthSecond,NULL); /* let someone else run */
36.     if (mq_receive(myQueue,(char *)&putDataHere,
37.                 msgSize,(unsigned int *)NULL) != -1)
38.     printf("Sent %d Received %d\n",*(testData+i),putDataHere);
39.     else
40.     {
41.        printf("mq_receive() problem %s\n",strerror(errno));
42.        abort();
43.     }
44.  }
45.  else
46.  {
47.     printf("mq_send() problem %s\n",strerror(errno));
48.     abort();
49.  }
50. }
51.
52. /* send shutdown message to background */
53. mq_send(myQueue,(const char *)(testData+3),4,0);
54. nanosleep(&tenthSecond,NULL); /* let queue shutdown */
55.
56. return 0;
57. }
```

Figure 14.4. (Continued)

Notice that when the queue is opened it is not necessary to specify its attributes. Line 24 shows the command necessary to attach to an existing queue. Full error handling for all queue commands is shown for both foreground and background programs. For clarity's sake, subsequent examples will leave out the error handling. Again the foreground program uses nanosleep to force context switches. The following commands will compile and execute the code snippets.

```
gcc --lrt --o backProcess back1.c
gcc --lrt --o foreProcess fore1.c
./backProcess 2>errLog &
./foreProcess
```

The standard error stream on the background process is re-directed to a file named errLog. When the foreground sends a zero, this is used to tell the background to shutdown. No attempt is made to apologize for the use of nanosleep() in forcing a context switch. However this is very, very bad real-time programming practice for a variety of reasons. First, it's not real-time. As a matter of fact it takes somewhat more than 0.3 seconds to execute this program to completion. Second, although the nanosleep forces a context switch it does not guarantee that the other process runs. The two processes must ping-pong for this program to work. Nanosleep() cannot force that behavior. Finally, suppose the foreground uses a one second sleep but the background a tenth of a second. The user will be quite surprised to see that instead of adding five, the background ran ten times and added 50! A more precise real-time construct is required.

Sticking with the SCA mandatory interfaces we see that POSIX semaphores are required to be supported in all SCA-compliant systems. The next code examples (Figures 14.5 and 14.6) use semaphores to signal each end of the bi-directional message queue. Two semaphores are required, one for the foreground to tell the background task to pickup a message and the other for the background to tell the foreground to pickup a message. First the background process, shown in Figure 14.5.

```
/* back2.c */

#include <mqueue.h>
#include <sys/stat.h> /* for mode_t flags */
#include <time.h>
#include <stdio.h>
#include <semaphore.h>

static const struct timespec tenthSecond = {0,100000000};

#define msgSize 4 /* 4 bytes = an int */

int main()
{
mqd_t myQueue;
const char queueName[]="/myQueueName";

sem_t *p_myRSem=NULL, *p_mySSem=NULL, *p_myXSem=NULL;
const char semRName[]="/myRSemName";
const char semSName[]="/mySSemName";
const char semXName[]="/myXSemName";

mode_t allowOwner = S_IRUSR | S_IWUSR | S_IXUSR;
struct mq_attr someAttributes;
int semCreated=0L;
int putDataHere;

someAttributes.mq_flags = 0;
someAttributes.mq_maxmsg = 1;
someAttributes.mq_msgsize = msgSize;

/* housekeeping: get rid of any queues from previous crashes, etc */
mq_unlink(queueName); sem_unlink(semRName);
```

Figure 14.5. Named semaphores – background

```
34. sem_unlink(semSName); sem_unlink(semXName);
35.
36. /* Exclusively create a read/write, non-blocking queue */
37. /* Blocking will be handled by the semaphores */
38. myQueue = mq_open(queueName,O_RDWR|O_CREAT|O_EXCL|O_NONBLOCK,
39. allowOwner,&someAttributes);
40. fprintf(stderr,"Queue is open for business\n");
41.
42. /* Wait for the foreground to create the semaphores */
43. while (!semCreated)
44. {
45.      p_myXSem = sem_open(semXName,0L);
46.      if (p_myXSem == SEM_FAILED)
47.         nanosleep(&tenthSecond,NULL); /* let someone else run */
48.      else
49.         semCreated = 1;
50. }
51. p_myRSem = sem_open(semRName,0L);
52. p_mySSem = sem_open(semSName,0L);
53. fprintf(stderr,"Have accessed the semaphores\n");
54. sem_post(p_myXSem); /* Let foreground know we're ready */
55.
56. while (1)
57. {
58.     sem_wait(p_myRSem);
59.     mq_receive(myQueue,(char *)&putDataHere,
60.     msgSize,(unsigned int *)NULL);
61.     if (putDataHere==0) break;
62.     putDataHere += 5;
63.     mq_send(myQueue,(const char *)&putDataHere,4,0);
64.     sem_post(p_mySSem);
65. }
66.
67. sem_post(p_myXSem); /* let foreground know we're done */
68. if ((mq_unlink(queueName) == 0)&&(sem_unlink(semRName)==0)&&
69.      (sem_unlink(semSName) == 0)&&(sem_unlink(semXName)==0))
70.     fprintf(stderr,"Unlink complete shutting down\n");
71. else
72.     perror("unlink() problem");
73.
74. return 0;
75.
76. }
```

Figure 14.5. (Continued)

Similar to our first attempt, the background task is responsible for creating the message queue. This time however the queue is non-blocking – line 38, O_NONBLOCK. The semaphores will provide the blocking function. The foreground process (Figure 14.6) will be responsible for creating the semaphores. The background process tries to open the semaphore in line 45. This open call does not specify the `O_CREAT` flag and if the semaphore does not already exist, the call will fail. The background will check every tenth of a second for the existence of the semaphore. The program has three semaphores: two to control access to the message queue and one to synchronize the foreground and background.

```
1. /* fore2.c */
2.
3. #include <mqueue.h>
4. #include <sys/stat.h> /* for mode_t flags */
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <semaphore.h>
8.
9. #define msgSize 4 /* 4 bytes = an int */
10.
11. int main()
12. {
13.    mqd_t myQueue;
14.    const char queueName[]="/myQueueName";
15.
16.    sem_t *p_myRSem=NULL, *p_mySSem=NULL, *p_myXSem=NULL;
17.    const char semRName[]="/myRSemName";
18.    const char semSName[]="/mySSemName";
19.    const char semXName[]="/myXSemName";
20.
21.    mode_t allowOwner = S_IRUSR | S_IWUSR | S_IXUSR;
22.
23.    int putDataHere, i;
24.    int testData[]={12,2,7,0};
25.
26.    /* create semaphores */
27.    p_myRSem = sem_open(semRName,O_CREAT|O_EXCL,allowOwner,0L);
28.    p_mySSem = sem_open(semSName,O_CREAT|O_EXCL,allowOwner,0L);
29.    p_myXSem = sem_open(semXName,O_CREAT|O_EXCL,allowOwner,0L);
30.    if ((p_myRSem == SEM_FAILED)||
31.        (p_mySSem == SEM_FAILED)||(p_myXSem == SEM_FAILED))
32.    {
33.       perror("sem_open() problem");
34.       abort();
35.    }
36.
37.    /* Attach to the queue */
38.    myQueue = mq_open(queueName,O_RDWR|O_NONBLOCK);
39.
40.    /* wait for background to acquire the semaphores */
41.    sem_wait(p_myXSem);
42.
```

Figure 14.6. Named semaphores – foreground

```
43.   for (i=0; i<3; ++i)
44.   {
45.     mq_send(myQueue,(const char *)(testData+i),4,0);
46.     sem_post(p_myRSem);
47.     sem_wait(p_mySSem);
48.     mq_receive(myQueue,(char *)&putDataHere,
49.                       msgSize,(unsigned int *)NULL);
50.     printf("Sent %d Received %d\n",*(testData+i),putDataHere);
51.   }
52.
53.   /* send shutdown message to backEnd */
54.   mq_send(myQueue,(const char *)(testData+3),4,0);
55.   sem_post(p_myRSem);
56.   sem_wait(p_myXSem); /* let the background shutdown */
57.
58.   return 0;
59. }
```

Figure 14.6. (Continued)

So the foreground program will execute up to and block at line 41. Once the background has acknowledged acquiring the semaphores the background will then post the 'X' semaphore to fire-up the blocked the foreground. The same 'X' semaphore is used again at the end of the program for the background process to tell the foreground task it is done using the queue and that it is OK to shutdown. Not only does this program run significantly faster than the nanosleep version, but also its architecture guarantees the ping-pong behavior required for a successful execution.

The final code examples (Figures 14.7 and 14.8) use two uni-directional blocking queues instead of a single bi-directional queue.

```
/* back3.c */

#include <mqueue.h>
#include <sys/stat.h> /* for mode_t flags */
#include <stdio.h>
#include <semaphore.h>

#define msgSize 4 /* 4 bytes = an int */

int main()
{
   mqd_t myRQueue,mySQueue;
   const char rcvrName[ ]="/myRcvrQueue";
   const char sendName[ ]="/mySendQueue";
```

Figure 14.7. Uni-directional message queues – background

```

   mode_t allowOwner = S_IRUSR | S_IWUSR |S_IXUSR;
   struct mq_attr someAttributes;
   int putDataHere;

   someAttributes.mq_flags = 0;
   someAttributes.mq_maxmsg = 1; /* nice bi-directional blocking */
   someAttributes.mq_msgsize = msgSize;

   /* housekeeping: get rid of queues from previous crashes, etc */
   mq_unlink(rcvrName);
   mq_unlink(sendName);

   /* Exclusively create a read only blocking queue */
   myRQueue = mq_open(rcvrName,O_RDONLY|O_CREAT|O_EXCL,
                              allowOwner,&someAttributes);

   /* Exclusively create a write only blocking queue */
   mySQueue = mq_open(sendName,O_WRONLY|O_CREAT|O_EXCL,
                              allowOwner,&someAttributes);
   fprintf(stderr,"Queues are open for business\n");

   while (1)
   {
   mq_receive(myRQueue,(char *)&putDataHere,
                      msgSize,(unsigned int *)NULL);     /* blocking */
   if (putDataHere==0) break;
   putDataHere += 5;
   mq_send(mySQueue,(const char *)&putDataHere,4,0); /* blocking */
   }

   if ((mq_unlink(rcvrName) == 0)&&(mq_unlink(sendName)==0))
      fprintf(stderr,"Unlink complete shutting down\n");
    else
       perror("unlink() problem");

    return 0;

}
```

Figure 14.7. (Continued)

The background process (Figure 14.7) establishes the polarity for the message queue names. The ‘R’ queue is used by the background as read-only. It will be opened by the foreground as write-only.

Similarly the ‘S’ queue is opened by the background as write-only but will be the receive-only queue from the perspective of the foreground process – line 21 in (Figure 14.8). Of each of the versions presented this version executes the quickest. Because the write-side of the queue fills up and blocks after just one message is written to it, there is an immediate context

```
/* fore3.c */

#include <mqueue.h>
#include <time.h>
#include <stdio.h>

static const struct timespec tenthSecond = {0,100000000};

#define msgSize 4 /* 4 bytes = an int */

int main()
{
  mqd_t myRQueue,mySQueue;
  const char sendName[]="/myRcvrQueue"; /* swap names relative to */
  const char rcvrName[]="/mySendQueue"; /* background process */

  int putDataHere, i;
  int testData[]={12,2,7,0};

  /* Attach read only blocking queue */
  myRQueue = mq_open(rcvrName,O_RDONLY);

  /* Attach write only blocking queue */
  mySQueue = mq_open(sendName,O_WRONLY);
  fprintf(stderr,"Queues are open for business\n");

  for (i=0; i<3; ++i)
  {
    mq_send(mySQueue,(const char *)(testData+i),4,0);
    mq_receive(myRQueue,(char *)&putDataHere,
                  msgSize,(unsigned int *)NULL);
     printf("Sent %d Received %d\n",* (testData+i),putDataHere);
  }

  /* send shutdown message to backEnd */
  mq_send(mySQueue,(const char *)(testData+3),4,0);
  nanosleep(&tenthSecond,NULL); /* let queue shutdown */

  return 0;
}
```

Figure 14.8. Uni-directional message queues – foreground

switch to the reader of that queue. Instead of semaphores performing the blocking operation, the queue itself optimally blocks and un-blocks in ping-pong fashion. So even without the `fork()` and `exec()` commands the applications programmer is quite unrestricted in process-oriented programming through the shell. The SCA *ExecutableDevice* can be used to create true processes in operating systems that support that capability.

14.2.2 More Unavailable POSIX Calls

`_POSIX _JOB _CONTROL` is not required in SCA-compliant systems. This puts the SCA somewhat at odds with the Federal Information Processing Standard 151-2 [10], which specifies job control as mandatory. So what is it and will we miss this capability in our software radio? Job control harkens back to the earliest days of POSIX. Job control allows sufficiently privileged users the ability to control who gets access to a terminal device [11]. Processes launched from shell scripts can run in the foreground or the background. Foreground tasks have access to terminal IO whereas background processes do not. The following terminal control calls are not expected to be found on an SCA-compliant system:

40. `tcdrain()`
41. `tcflush()`
42. `tcgetsid()`
43. `tcgetattr()`
44. `tcflow()`
45. `tcsetpgrp()`
46. `tcgetpgrp()`
47. `tcsendbreak()`
48. `tcsetattr()`

We can conceive of scenarios in which an applications programmer might want to change the attributes of a terminal. We consider this example in Section 19.5.1 on *Devices*. The SCA specification, Appendix B, also excludes other system administration-type calls from the SCA POSIX vocabulary. All of these fall under the context of file access control, group access, login control, and privileges, etc.

49. `getegid()`
50. `getgid()`
51. `getlogin()`
52. `getuid()`
53. `chmod()`
54. `getgrgid()`
55. `getgrnam()`
56. `getpwnam()`
57. `getpwuid()`
58. `setgid()`
59. `chown()`
60. `geteuid()`
61. `getgroups()`
62. `getpgrp()`
63. `setsid()`
64. `umask()`
65. `getgrgid_r()`
66. `getgrnam_r()`
67. `getpwnam_r()`
68. `getpwuid_r()`
69. `setuid()`
70. `getlogin_r()`

The applications programmer is unlikely to miss these system calls. If an applications programmer finds a need to access these calls, there is probably some bad functional allocation going on at the system design level. These calls are most closely associated with systems and security administration.

A few more batches of calls are not required in SCA-compliant systems. This next set provides device specific support for terminals:

71. `cfgetispeed()`
72. `cfsetispeed()`
73. `ctermid()`
74. `ttyname()`
75. `tzset()`
76. `cfgetospeed()`
77. `cfsetospeed()`
78. `isatty()`
79. `ttyname_r()`

These clearly are functions more appropriately accessed from Logical *Device* or device drivers. Remember though an application is prohibited the use of these system calls; a Device or device driver is not.

The next batch of unsupported calls concern advanced controls applied to streams and files. The applications programmer is discouraged the use of these calls because they are highly non-portable. The applications programmer is required to use the SCA-mandatory Core Framework *File* and *FileSystem* interfaces instead of their low-level POSIX equivalents. The following system calls are not to be used by the applications programmer.

80. `dup()`
81. `fcntl()`
82. `mkfifo()`
83. `getchar_unlocked()`
84. `ftrylockfile()`
85. `putc_unlocked()`
86. `fsync()`
87. `readdir_r( )`
88. `dup2()`
89. `pipe()`
90. `getc_unlocked()`
91. `flockfile()`
92. `funlockfile()`
93. `putchar_unlocked()`
94. `fdatasync()`

The only system call in this not-required list that is somewhat perplexing is the `readdir_r()` call. This is the re-entrant version of the `readdir()` call which is mandatory. Each of the constructs excluded here, for example, pipes and fifos, can easily be replaced by the mandatory message queues. There is a tacit requirement to be found in the partially required `SYNCHRONOUS_IO`. The elements of `SYNC_IO` that are excluded are `fsync()`, `fdatasync()`, and `fnctl()`.

The only system calls remaining are the various flavors of open. There are five 'open' system calls required, besides message queues and semaphores: `opendir()`, `open()`, `fopen()`, `fdopen()`, and `freopen()`. `SYNC_IO` requires that the following flags be supported in the open calls: `O_DSYNC`, `O_SYNC`, and `O_RSYNC`. These flags provide the same effect as `fsync()` and `fdatasync()` but are applied when the file is first opened.

Three remaining system calls are not to be used by the applications programmer.

95. `siglongjmp()`
96. `sigsetjmp()`
97. `alarm()`

The set jump call (96) specified as not required is a version of setjmp that also saves the signal mask of the caller. The mandatory version of `setjmp()` saves the environment only. The `alarm()` call is easily superseded by the mandatory POSIX timers. Here is one final note on what is not expected to be available to the applications programmer. The `#define_POSIX_VDISABLE` is not mandatory. In systems that do support this, the `#define` is used to identify the character that disables special control characters used by the terminal. Reference [10] provides another means of accomplishing the same feat using the mandatory `fpathconf()` system call. So it is possible to retrieve the terminal's disable character but the applications programmer is not allowed to change it. The `tcsetattr()` system call is not required by the SCA. A device level programmer does not have the same restriction.

This chapter introduced the Operating Environment including the one we will be using throughout the remainder of this book. The chapter also focused on the first layer of OE-required functionality, the mandatory POSIX system calls. Our approach was to highlight those system calls which are not required with the intent to identify certain functions that the applications programmer just cannot use.

After examining the 97 system calls not guaranteed to be present on an SCA-compliant system we see that in each case there are other mandatory calls that can be used instead. The SCA identifies 256 mandatory POSIX system calls – see Appendix A of this book. This is more than enough to satisfy the needs of the applications programmer. Recall that even the mandatory POSIX file system calls are not supposed to be used by the applications programmer but rather their Core Framework equivalents. In the thirty or so not required calls that concern system administration, file-locking, and terminal control we question whether, in a properly designed system, an applications programmer would ever require access to these commands. In Chapter 19, we examine how a logical *Device* might want to access the terminal, but device software is not subject to the same constraints as applications software. Finally there is some concern on the lack of support for `fork()/exec()` and most importantly `_POSIX_PRIORITY_SCHEDULING`. If you have an application designed to utilize priority scheduling you must either encapsulate prioritized tasks as *ExecutableDevices* or use POSIX threads. Each method will be explored in Chapters 15 and 19.

15

POSIX Threads

The SCA mandates 58 POSIX thread systems calls (out of 80 possible) [2]. This chapter introduces the POSIX thread system calls that are both mandatory and the most commonly used. POSIX threads are composed of five primary functional 'objects' – historically they are called variables. Since POSIX threads somewhat pre-date the object-oriented paradigm of C++ or CORBA, the term 'object' is used allegorically. First, the system calls are implemented in straight 'C' – structs instead of classes. Second, without the implicit C++ 'this' pointer, the POSIX thread structs must be passed to routines referenced by pointers. The five objects are:

- the thread object – `pthread_t`
- the mutex variable – `pthread_mutex_t`
- the conditional variable – `phtread_cond_t`
- the key variable – `pthread_key_t`
- the read-write lock – `pthread_rwlock_t`

With the exception of the key variable, each functional object has a corresponding attribute structure: i) `pthread_attr_t`; ii) `pthread_mutexattr_t`; iii) `pthread_condattr_t`; and iv) `pthread_rwlockattr_t`. Read-write locks were introduced after the IEEE 1995 standard and are not mandatory for SCA-compliant systems. There are a total of 11 system calls associated with read-write locks that we don't discuss further. Incidentally there is nothing that can be done with read-write locks that cannot otherwise be accomplished with conditional and mutex variables.

As discussed in the previous chapter POSIX threads offer a means of supporting independent execution paths without the overhead of full-fledged process-oriented context switching. Threads can be implemented on hardware that doesn't have a memory-management unit (MMU) or on a operating system that doesn't support the notion of virtual address spaces. Most importantly for the applications programmer who desires portability across SCA-compliant platforms, POSIX threads offer the only way to implement i) priority-based scheduling and ii) shared memory constructs such as unnamed semaphores. From Chapter 14 we are reminded that `_POSIX_PRIORITY_SCHEDULING` and `_POSIX_SHARED_MEMORY_OBJECTS` are not required on SCA-compliant systems.

Threads provide a convenient and portable way to implement both of these powerful constructs vital to the real-time software designer.

15.1 The Thread Object

A typical hierarchy for a threaded program involves a main program – int main() in 'C'– that launches one or more threads and then merely goes to sleep until the threads terminate. In some cases the main program might run in a loop providing thread status to a user or perhaps receiving user input and passing it down to the threads for processing. More complex programs could involve multiple threads running within multiple processes. For SCA-based applications this is not advisable because `_POSIX_THREAD_PROCESS_SHARED` is not mandatory on SCA-compliant systems. This lack makes it impossible for mutex and conditional variables to be shared across processes. Inter-process synchronization could certainly be done with POSIX.1b semaphores or message queues, but that takes away the advantage of the lightweight context switching offered by threads. And, of course, prioritized inter-process synchronization is also not mandatorily supported by the SCA. If prioritized synchronization is to be used in an SCA-compliant application the only fully supported means of implementation is with POSIX thread mutex and conditional variables.

Figure 15.1 provides an overview of the primary data types used to construct POSIX threads. The *pthread_t* parameter is required on the six commonly used calls shown in the figure. Separately listed are two calls to manipulate scheduling parameters – this will be discussed later. *pthread_t* is a transparent type in that the user need not worry about its internal details. Its attributes should only be manipulated through set/get operations on *pthread_attr_t*.

Figure 15.1 shows that *pthread_t* has a 'create' method. This is not the corollary to the object-oriented constructor. The reader will notice that there is no corresponding destructor. Rather the user is required to allocate storage for the thread structure and then pass a pointer to the create function which then fills in the structure. The full prototype for the create function is as follows:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine)(void*),void *arg);
```

The actual contents of `thread` are implementation-dependent. There is nothing in *thread* that is directly accessible to the user. The user has the ability to establish attributes of the thread upon its creation by passing a pointer to an attributes structure, *`phtread_attr_t`*. With a single exception once these attributes are set they cannot be read back, i.e. thread attributes are write-only. If the attribute pointer points to NULL the thread is created with default attributes. The POSIX standard does not mandate default attributes; they are implementation dependent. Two other parameters fill out the parameter list. One is the entry point to the thread routine. This is a pointer to a function, `start_troutine`, that accepts a pointer to a single argument – *arg*, and returns a single value also cast as a *void* pointer. A simple code example is warranted. The code segment in Figure 15.2 creates a thread with default attributes.

The code example in Figure 15.2 makes use of a function `printAttributes` which is found in Section 15.4 on thread attributes. The command used to compile and link executable is as follows:

```
gcc – I/usr/include/nptl – L/usr/lib/nptl attr1.c printAttributes.c –
  lpthread
```

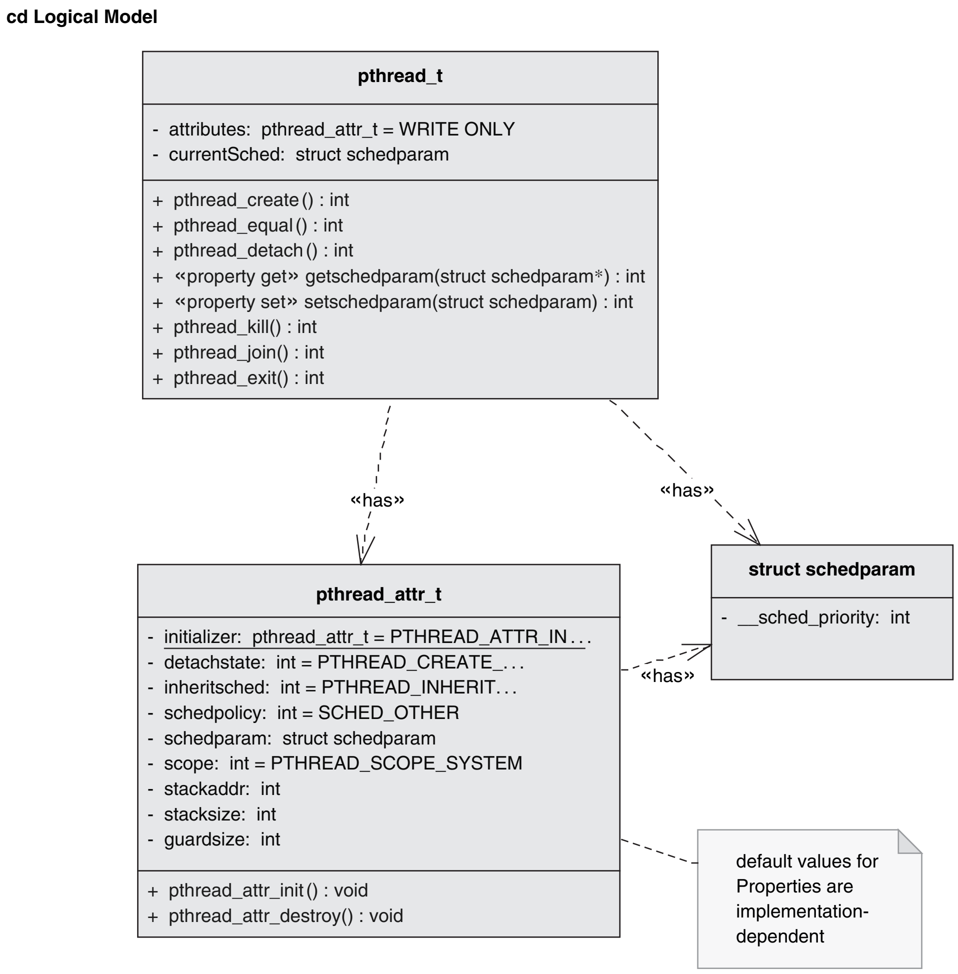

cd Logical Model
pthread_t
- attributes: pthread_attr_t = WRITE ONLY
- currentSched: struct schedparam
+ pthread_create () : int
+ pthread_equal () : int
+ pthread_detach () : int
+ «property get» getschedparam(struct schedparam*) : int
+ «property set» setschedparam(struct schedparam) : int
+ pthread_kill() : int
+ pthread_join() : int
+ pthread_exit() : int
«has»
«has»
pthread_attr_t
- initializer: pthread_attr_t = PTHREAD_ATTR_IN...
- detachstate: int = PTHREAD_CREATE_...
- inheritsched: int = PTHREAD_INHERIT...
- schedpolicy: int = SCHED_OTHER
- schedparam: struct schedparam
- scope: int = PTHREAD_SCOPE_SYSTEM
- stackaddr: int
- stacksize: int
- guardsize: int
+ pthread_attr_init () : void
+ pthread_attr_destroy() : void
struct schedparam
- __sched_priority: int
«has»
default values for Properties are implementation-dependent

Figure 15.1. pthread_t component diagram

```
/* attr1.c */

#include <stdio.h>
#include <pthread.h>

void printAttributes(const pthread_attr_t* );

int sampleThread(int* );

int main()
{
```

Figure 15.2. pthread attributes test code

```
12.  int start=12, finish=0;
13.  pthread_attr_t myPthreadAttr;
14.  pthread_t myFirstThread;
15.
16.  pthread_attr_init(&myPthreadAttr); /*default~attributes */
17.  printAttributes((const pthread_attr_t*)&myPthreadAttr);
18.
19.  pthread_create(&myFirstThread,
              (const pthread_attr_t *)&myPthreadAttr,
              (void *)sampleThread,(void *)&start);
20.
21.  pthread_attr_destroy(&myPthreadAttr); /* not needed anymore */
22.
23.  /* wait for thread to exit */
24.  pthread_join(myFirstThread,(void **)&finish);
25.  printf("Value returned from thread %d\n",finish);
26.
27.  return 0;
28. }
29.
30. int sampleThread(int *pInt)
31. {
32.    int localCopy = *pInt;
33.    localCopy += 5;
34.    return localCopy;
35. }
```

Figure 15.2. (Continued)

This command creates the default executable a.out. The reader is advised that the path to the 'pthread.h' header file and corresponding library might be different than that cited above in their own system. In the case of the 2.6 kernel with glibc 2.3.4 the old LinuxThreads and the new NPTL threads exist side-by-side. For backwards compatibility reasons the default path to pthread.h points to the old LinuxThreads version. The NPTL and LinuxThread prototypes are identical. However, what's under the covers – most importantly, the behaviors – is different. At this time, support for the old LinuxThreads is slated to be discontinued in future releases of glibc so the user should be particularly careful to pickup the NPTL prototypes and library.

The program creates a thread with default attributes. The thread will accept a single integer argument, add five, and return the result – lines 30 through 35. Line 16 in the main program initializes a thread attributes structure to its default values. Line 17 calls the routine to print out those default values – this routine is used again later and is found in Figure 15.7. Line 19 actually creates the thread. Another technique for creating a thread with default attributes is to specify a NULL for argument 2 of `pthread_create()`. This technique would not work in this example because there is no way to query a thread directly for its attributes. The attributes structure itself can be 'queried' before or after the thread is actually created. Finally, Line 24 shows a very important means of keeping the main program synchronized with the threads. The `pthread_join()` function blocks the caller – in this case the main program – until the indicated thread exits. Better yet a return code from the terminating thread can be retrieved and processed. Table 15.1 provides the output from a successful run of the sample program.

Table 15.1. Default POSIX thread attributes

```
PTHREAD_CREATE_JOINABLE
PTHREAD_ INHERIT_SCHED
PTHREAD_SCOPE_SYSTEM
SCHED_OTHER
Sched param 0
Value returned from thread 17
```

Use of the `pthread_create()` function requires that both the input parameter and the return parameter must be cast as *void* pointers. Anytime *void* pointers are used in 'C' code there is bound to be abuse of the *void* pointer type cast for the purpose of surviving parameter checks performed by the compiler. Line 8 of the code example in Figure 15.2 shows the prototype for our simple thread. The function prototype indicates that the user will pass a pointer to an integer and the function will return an integer. When the thread is created, line 19, both the argument and function return value are re-cast as *void* pointers. For the thread argument, re-casting a pointer to an *int* as a pointer to *void* is allowed and portable. The user is equally enabled to perform the same cast on a pointer to a *struct*. Thus complex data types can be passed in to the thread via this mechanism. The abuse being performed in this example is on the return value. The thread returns an integer result which is then re-cast as a *void* pointer. This is not portable and quite risky. In this case an *int* type and a *void* pointer type happen to be the same size (in bytes). So what if the user wants to pass back a more complex result? Simply put, don't think about using the thread return value for this purpose. The return value should be treated the same as an exit value from the main program. Typical usage is for a return value of zero to indicate normal execution and non-zero to indicate some kind of error.

An even more heinous abuse of the return code would be to make it point to a local variable. Of course as the system cleans up the thread's stack upon exit that pointer ends up essentially pointing to garbage. The clever programmer might allocate storage on the heap while inside the thread and pass that pointer back to the `pthread_join()`. While this indeed would work brilliantly it might lead to memory leaks as the caller of the `_join` function would be responsible for freeing the memory. Ultimately the return code should be used only for that and nothing else. If the user wants to return more complex results, then space should be allocated for those results in the user's own context and a pointer passed to the results holding area as part of the input argument. If the user is passing both complex data types IN to the thread and pulling complex data types OUT of the thread, then a single structure should be created that holds both.

15.2 Un-named Semaphores

Since the SCA does not mandate support for inter-process shared memory it is not possible to implement un-named semaphores between processes. It is however quite easy to implement in POSIX threads because all threads exist within the same process space. Un-named threads are considered part of the real-time extensions to the POSIX.1b family. They are advantaged over their named cousins because of less system overhead and thus faster switching. All POSIX semaphores are counting semaphores. That is every sem_post() operation will increase its count by one. When the count is non-zero a sem_wait() will decrement the count and return immediately. If the caller of sem_wait() desires to be blocked that blocking will only occur when the value of the semaphore is zero. If posting the semaphore and blocking on the semaphore somehow become un-synchronized bad things could happen. Figure 15.3 shows a code example that is a threaded version of the Chapter 14's fore2.c and back2.c. A server thread will add five to what ever number is passed to it and return the result to 'caller'.

```
/* thread1.c */

#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>

typedef struct {
        sem_t myRSem, mySSem;
        volatile int putDataHere;
        } sharedMem;

void client(sharedMem* );
void server(sharedMem* );

int main()
{
    pthread_t clientT,serverT;
    sharedMem msgQ;

    /* create unnamed semaphores */
    sem_init(&(msgQ.myRSem),0,0);
    sem_init(&(msgQ.mySSem),0,0);

    /* create threads, pass address of msgQ */
    pthread_create(&serverT,
          (const pthread_attr_t *)NULL,(void *)server,(void *)&msgQ);
    pthread_create(&clientT,
          (const pthread_attr_t *)NULL,(void *)client,(void *)&msgQ);

    /* wait for threads to terminate */
```

Figure 15.3. Un-named semaphores – main

```
31.    pthread_join(clientT,(void **)NULL);
32.    pthread_join(serverT,(void **)NULL);
33.
34.    sem_destroy(&(msgQ.myRSem));
35.    sem_destroy(&(msgQ.mySSem));
36.
37.    printf("Threads have exited, shutting down\n");
38.
39.    return 0;
40. }
```

Figure 15.3. (Continued)

A single structured variable is setup to share between the server and client threads – lines 7 through 10. In the structure are two unnamed semaphores: one to signal a read to the server and the other to signal a read to the client. Instead of a message queue, the common data structure includes a single integer field called `putDataHere`. Server and client will read and write this common field as a means of passing data back and forth. The reason for the volatile qualifier in line 9 will be made evident shortly. Lines 12 and 13 contain the prototypes for the client and server entry points. A pointer to the common structured variable is passed to each routine. Lines 21 and 22 initialize the send/receive semaphores with `sem_init()`. This is quite different to initialization of a named semaphore, which requires use of `sem_open()`. The life cycle of named semaphores, is handled with calls to `sem_open()`, `sem_close()`, and `sem_unlink()`, whereas the named semaphores are handled with `sem_init()` and `sem_destroy()`. The two means of life cycle management are never to be mixed without serious consequences. For use on an SCA-compliant system the second argument to `sem_init()` should always be zero. This flag indicates an inter-process semaphore which is not mandatorily supported in SCA-compliant systems. The final parameter in `sem_init()` is the initial state of the semaphore. In this case zero indicates that the semaphore is locked – i.e. the first `sem_wait()` will block until someone performs a `sem_post()`.

The threads are actually created in lines 25–28. The pointer to the attributes parameter is NULL thus invoking the creation of threads with default attributes. A pointer to the common `msqQ` variable is cast as a *void* pointer and passed as arguments to each of the threads. Finally the main program goes to sleep on a `pthread_join()` waiting for the threads to exit.

Now let's see what's going on in the code for the client and server threads. Figure 15.4 shows a code segment that is a continuation of the source file containing the main program.

We examine the server thread first since it is created first. It enters a while forever loop and then blocks on the `sem_wait` – line 63. So at some point the client thread gets created and runs. The client writes an integer to the common data area – line 47 – and then posts the semaphore upon which the server has been waiting. The client then hits his or her own `sem_wait` – line 49 – and blocks waiting for a response from the server. Very neat, very clean. But let's start obscuring things a bit. First off, consider the work of an optimizing compiler. It's always looking for ways to simplify and save clock cycles. The putDataHere variable gets written in line 47 and then gets printed out in line 50. From the optimizing

```
41. void client(sharedMem* mQ_p)
42. {
43.    int i,testData[]={12,2,7,0};
44.
45.    for (i=0; i<3; ++i)
46.    {
47.       mQ_p->putDataHere = testData[i];
48.       sem_post(&(mQ_p->myRSem));
49.      sem_wait(&(mQ_p->mySSem));
50.      printf("Sent %d Received %d\n",*(testData+i),mQ_p->
         putDataHere);
51.   }
52.
53.   /* send shutdown */
54.   mQ_p->putDataHere = testData[3];
55.   sem_post(&(mQ_p->myRSem));
56.
57. }
58.
59. void server(sharedMem* mQ_p)
60. {
61.     while (1)
62.     {
63.         sem_wait(&(mQ_p->myRSem));
64.         if ( mQ_p->putDataHere==0 ) break;
65.         mQ_p->putDataHere += 5;
66.         sem_post(&(mQ_p->mySSem));
67.     }
68. }
```

Figure 15.4. Un-named semaphores – client/server

compiler's perspective nothing happens to the variable between the time it gets written till the time it gets output. Why not save a few clock cycles and eliminate the middle man? In this case instead of `putDataHere` getting written, the variable `testData[i]` would be written. This would be disastrous for our demo program. This simple thought experiment illustrates the necessity for the *volatile* qualifier on the `putDataHere` variable. It instructs the optimizing compiler to make no assumptions about the variable: Simply that it can be changed by forces beyond its ken and should be distinctly read from memory each and every time it needs to be accessed – no caching, no register variables. Many of our multi-threaded examples would not even work without the *volatile* qualifier.

Another obscuration comes from the fact that there are no guarantees as to who runs when. We assume the server thread runs first because it's created first. This is an invalid assumption. Let's pretend the client thread runs first. It will write the value to the common memory and post the semaphore before the server even runs. Fortunately for us when it hits the `sem_wait` – line 49 – it will block waiting for a response from the server instead of looping out of control. So in this case our design somewhat protects us from disaster.

So now the deterministic execution of our design is predicated upon the assumption that sem_wait is impervious to all the corrupting forces of nature. Nothing could be further from the truth because a sem_wait can return in response to a signal ([2], p. 757). Other more insane scenarios can happen within a real system. An unnamed semaphore is in fact a chunk of shared, unprotected memory. What's to say a thread couldn't attempt to wait on a semaphore that doesn't even exist? The reader is encouraged to perform an experiment on the code example in Figure 15.3. 1) Move initialization of the R semaphore – line 21 – to just after creation of the server thread, i.e. insert before line 27. 2) After creating the server thread but before initializing the R semaphore, insert a one second nanosleep. This will delay initialization of the semaphore and creation of the client thread thus allowing the server to run wild for awhile. 3) Finally, in the main program, say line 19, initialize msgQ.putDataHere to some non-zero, positive number. Re-compile and execute the code in order to witness the incorrect and undesirable results. How can we bullet-proof our code against wayward counting semaphores? POSIX threads offers such a method, which we discuss in the next section.

15.3 Mutex Variables

The mutex variable is used to guarantee that only one thread at a time can access a particular resource. In our previous example this resource is the variable putDataHere. As we ended the previous section we came to understand that breaking out of a sem_wait is not always due to the presence of valid data in the common area. Ultimately – and every text on semaphores says the same thing – it is necessary to check some kind of pre-condition or predicate to ascertain the validity of the release from the sem_wait. So in addition to the actual data being passed we'd also like to examine some kind of condition code that verifies the validity of the data. The problem is this: On the client side one operation writes the integer parameter and a following operation sets the validity code. What if the two operations were interrupted by a context switch? Our database would be in an intermediate state and the server thread would be more confused than ever. There must be a way to atomically update both the integer field and the validity code. Enter the POSIX threads mutex variable, which provides a means to lock the database so that only one thread can have access at a time. The code segment in Figure 15.5 adds the necessary features to our shared memory structure.

```
/* thread2.c */

#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>

#define DC_NO_VALUE 0x00000000
#define DC_TOSERV_OK 0x00000001
#define DC_TOCLNT_OK 0x00000002
#define DC_QUIT 0x00000003

```

Figure 15.5. Mutex variables

```
pthread_mutex_t Sprotect = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    sem_t myRSem, mySSem;
    volatile int putDataHere;   /* <-- this field AND */
   volatile int ctrlCode;       /* <-- this field protected by a mutex */
                                /* they are read/modified atomically */
   pthread_mutex_t *protect;
} sharedMem;

void client(sharedMem* );
void server(sharedMem* );

int main()
{
     pthread_t clientT,serverT;
     sharedMem msgQ;

     msgQ.protect = &Sprotect; /* already initialized */

     /*  initialize semaphores */
     sem_init(&(msgQ.myRSem),0,0);
     sem_init(&(msgQ.mySSem),0,0);
     msgQ.ctrlCode=DC_NO_VALUE; /* initialize data invalid */

    /*  create threads, pass address of msgQ */
     pthread_create(&serverT,
          (const pthread_attr_t *)NULL,(void *)server,(void *)&msgQ);
     pthread_create(&clientT,
          (const pthread_attr_t *)NULL,(void *)client,(void *)&msgQ);

     /* wait for threads to terminate */
     pthread_join(clientT,(void **)NULL);
     pthread_join(serverT,(void **)NULL);

     sem_destroy(&(msgQ.myRSem));
     sem_destroy(&(msgQ.mySSem));
     pthread_mutex_destroy(msgQ.protect);

     printf("Threads have exited, shutting down\n");

      return 0;
}

void client(sharedMem* mQ_p)
{
     int i,testData[]={12,2,7,0};
     volatile int localCopy;

```

Figure 15.5. (Continued)

```
   for (i=0; i<4; ++i)
   {
       pthread_mutex_lock(mQ_p->protect);
          mQs_p->putDataHere = testData[i];
          mQ_p->ctrlCode = DC_TOSERV_OK;
   pthread_mutex_unlock(mQ_p->protect);
   sem_post(&(mQ_p->myRSem));
   while (1)
   {
       sem_wait(&(mQ_p->mySSem));
       pthread_mutex_lock(mQ_p->protect);
         if ( mQ_p->ctrlCode==DC_TOCLNT_OK) break;
       pthread_mutex_unlock(mQ_p->protect);
   }
   localCopy=mQ_p->putDataHere;
   pthread_mutex_unlock(mQ_p->protect); /* unlock before performing
                                          time consuming IO */
   printf("Sent %d Received %d\n",*(testData+i),localCopy);
   }

   /* send shutdown */
   pthread_mutex_lock(mQ_p->protect);
      mQ_p->ctrlCode = DC_QUIT;
   pthread_mutex_unlock(mQ_p->protect);
   sem_post(&(mQ_p->myRSem)); /* "send" the message */

}

void server(sharedMem* mQ_p)
{
   while (1)
   {
      while (1)
      {
         sem_wait(&(mQ_p->myRSem));
         pthread_mutex_lock(mQ_p->protect);
           if ( (mQ_p->ctrlCode==DC_TOSERV_OK)||
               (mQ_p->ctrlCode==DC_QUIT) ) break;
         pthread_mutex_unlock(mQ_p->protect);
      }
    if ( mQ_p->ctrlCode==DC_QUIT ) break;
    mQ_p->putDataHere += 5;
    mQ_p->ctrlCode = DC_TOCLNT_OK; /* mark as valid msg to client
*/
    pthread_mutex_unlock(mQ_p->protect);
    sem_post(&(mQ_p->mySSem));
   }
   pthread_mutex_unlock(mQ_p->protect);

}
```

Figure 15.5. (Continued)

Lines 17 and 19 add the new fields to the common memory area. One is a control code and the other the mutex itself. Similar to the data field, the control code is qualified as *volatile* in order to notify the optimizing compiler that the fields are modifiable in a manner unknown to the compiler. Line 12 initializes the mutex to its default settings (more about mutex attributes later). Line 35 initializes the control code to indicate that the data field is not valid. The threads are launched and everything proceeds as before. In the server and client threads we see that each thread locks the data structure before reading or writing the data and control fields lines 63, 71, 82, and 96. A mutex is different than a counting semaphore. Repeated posts to a counting semaphore increment the value of the semaphore. A thread will block on a counting semaphore only when its count is zero. A mutex on the other hand can only be locked or unlocked. The thread that holds the lock is the thread that has access to the data structure.

In both threads we see that upon coming out of the `sem_wait` state the structure is locked and the threads use the control code to determine the validity of the data field – lines 72 and 97. `DC_TOCLNT_OK` marks a valid message to the client and `DC_TOSERV_OK` marks a valid message to the server. Of course, the mutex is locked to ensure that while reading the control code it doesn't change from underneath the thread that is doing the reading. This implementation also solves a problem we had in the prior examples where the numerical value of zero was interpreted as a shutdown command. Our adding machine could not operate on zero. In the new implementation zero is included in the range of valid integers and the separate control code is used to pass the quit command.

Lines 68–74 and 93–100 bullet-proof the `sem_wait` in that data validity is checked after becoming unblocked from the semaphore for whatever reason. The ping-pong dynamic of the software is ensured despite outside influences. Lines 75 and 76 warrant further observation. Upon entry to line 75 the resource – in this case the shared memory – is still locked. We want to output the data to the terminal but realize that the `printf` system call might take many, many clock cycles – perhaps even more than the entire example program itself. We are wise to copy the data out of the shared memory into a `localCopy` so that the mutex can be unlocked as quickly as possible. IO to the terminal can then happen without disrupting the timely flow of our executable. In general a good real-time design will absolutely minimize the time that a mutex is locked.

Now let's intentionally disrupt the program as prescribed in the previous example. 1) Move the `sem_init` in line 33 to just after the `pthread_create` for the server – line 38/39. 2) Put a one second nanosleep just after the `pthread_create` but before the `sem_init`. This delays the initialization of the semaphore and the creation of the client thread. Before we bullet-proofed the code the server thread ran amok by repeatedly posting the 'S' semaphore. Once the client thread started it was no longer synchronized with the server so the ping-pong relationship needed for this program to work was violated and incorrect results were generated. In the current implementation the server thread will test the validity of the data before performing the add operation and before posting the 'S' semaphore. Although the client thread startup is delayed the ping-pong relationship is preserved. This implementation will resist the many bullets a real-time system can throw our way.

In line 12 we used the default initializer, `PTHREAD_MUTEX_INITIALIZER,` to initialize the mutex variable. This default initializer only works for mutex variables that are statically allocated. It cannot be used to initialize a dynamically allocated mutex. In line 30 the address of the statically allocated mutex is used to initialize the mutex pointer variable in our common data structure. The following code sequence shows how to initialize the mutex variable dynamically:

```
msgQ.protect=(pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(msgQ.protect,(const pthread_mutexattr_t *)NULL);
```

By specifying the mutex attributes pointer to NULL, the default attributes are used. The POSIX98 standard provides static initialization capability for three other mutex types. These are not mandatory in a SCA-compliant system and for portability reasons should not be used.

Figure 15.6 shows the relationship between the mutex and its attributes. This diagram shows only the mandatory methods and attributes supported in an SCA-compliant system.

The figure excludes a couple of additional attributes that are not universally supported by the SCA: mutex `type` and mutex `pshared`. The `pshared` attribute allows a mutex to be shared with threads in other processes. It requires `_POSIX_SHARED_MEMORY_OBJECTS`, which is only optional in SCA-compliant systems. The `type` attribute controls

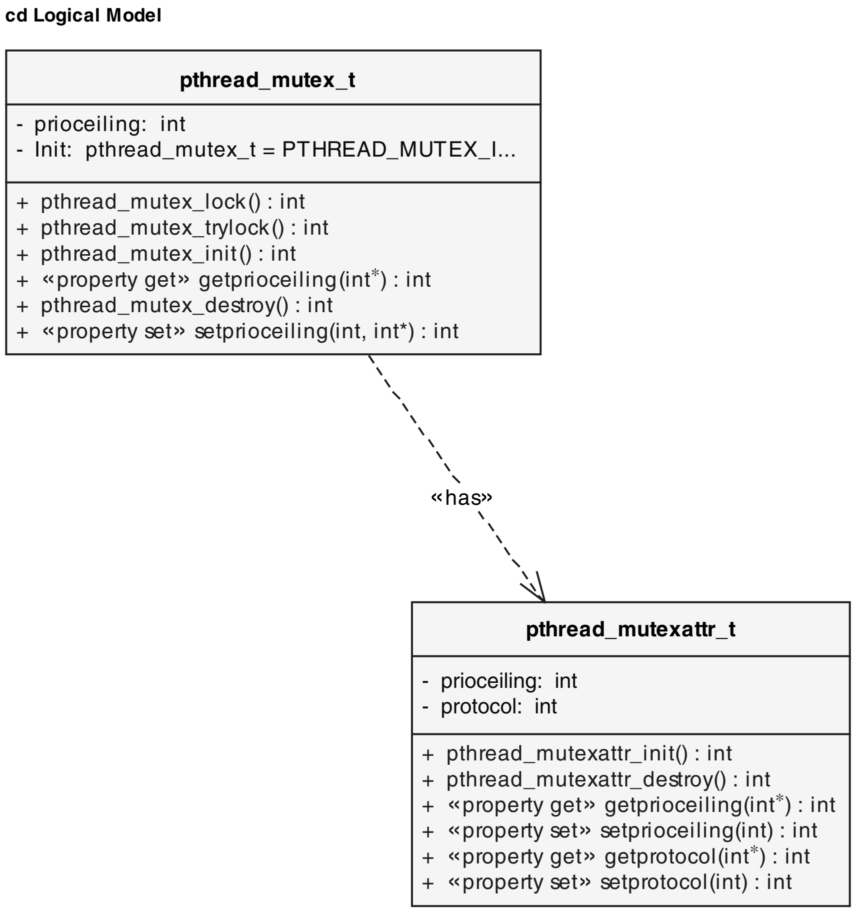

cd Logical Model
pthread_mutex_t
- prioceiling: int
- Init: pthread_mutex_t = PTHREAD_MUTEX_I...
+ pthread_mutex_lock() : int
+ pthread_mutex_trylock() : int
+ pthread_mutex_init() : int
+ «property get» getprioceiling(int*) : int
+ pthread_mutex_destroy() : int
+ «property set» setprioceiling(int, int*) : int
«has»
pthread_mutexattr_t
- prioceiling: int
- protocol: int
+ pthread_mutexattr_init() : int
+ pthread_mutexattr_destroy() : int
+ «property get» getprioceiling(int*) : int
+ «property set» setprioceiling(int) : int
+ «property get» getprotocol(int*) : int
+ «property set» setprotocol(int) : int

Figure 15.6. pthread_mutex_t component diagram error and deadlock detection and should not be used in SCA-compliant applications. Some discussion of priority ceiling and protocol is warranted. SCA systems support the three following protocols `PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT`. If a mutex is initialized with `PRIO_NONE` there is no relationship between locking and unlocking a mutex and thread priority. If the mutex is created with the protocol `PRIO_INHERIT` priority inheritance is enforced. If a low priority thread owns the mutex and a higher priority thread attempts to lock the same mutex, the OS will raise the priority of the low thread to that of the thread attempting the lock. This will ensure that the low thread will not be pre-empted by any threads having priority between that of the low and high threads. Timely execution of the low thread is assured, and its ability to complete its task and release the mutex is also assured. Once the mutex is released the OS will lower the low thread priority back to its original level.

Also supported in SCA systems is `PRIO_PROTECT`. The behavior of a mutex initialized with this protocol behaves exactly like that of a mutex with `PRIO_INHERIT` with one notable exception: An additional parameter prioceiling establishes the maximum priority to which the low priority thread can be elevated. When the mutex is contended. This protocol protects or preserves threads of priority higher than prioceiling from not being able to execute immediately.

In SCA operating environments four additional system calls are supported: `pthread_ mutexattr_ getprioceling` and `pthread_mutexattr_ setprioceling` which allow the user to set and get the priority ceiling value before the mutex is created and `pthread_mutex_getprioceiling` and `pthread_ mutex_setprioceiling` which allow the user to query and configure the prioceiling of an already instantiated mutex.

A final mutex system call is the `pthread_mutex_trylock`. This call will either lock the mutex or return immediately if the mutex is owned by somebody else. The return value from the `pthread_mutex_trylock()` call will be zero if the lock was acquired and non-zero, i.e. `EBUSY`, if the thread is locked by someone else.

15.4 Thread Attributes

Given the previous discussion of priorities, how is a thread created with a particular priority? There are a series of operations that allow attributes to be configured on a thread as it is created. All of these operations on the `pthread_attr_t` structure are referenced in Figure 15.1 and explicitly listed in Table 15.2.

Earlier in Table 15.1 we allowed the system to initialize the thread with default attribute values. Valid values for each attribute are as follows:

detachstate = `PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED`

schedpolicy = `SCHED_OTHER, SCHED_FIFO, SCHED_RR`

scope = `PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS`

inheritsched = `PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED`

The code segment in Figure 15.7 is that used by the program in Section 15.1 to print out the thread attributes structure.

Table 15.2. Set and get pthread attributes

pthread_attr_getdetachstate( )	pthread_attr_setdetachstate( )
pthread_attr_getguardsize( )	pthread_attr_setguardsize( )
pthread_attr_getinheritsched( )	pthread_attr_setinheritsched( )
pthread_attr_getschedparam( )	pthread_attr_setschedparam( )
pthread_attr_getschedpolicy( )	pthread_attr_setschedpolicy( )
pthread_attr_getscope( )	pthread_attr_setscope( )
pthread_attr_getstackaddr( )	pthread_attr_setstackaddr( )
pthread_attr_getstacksize( )	pthread_attr_setstacksize( )

```
/* printAttributes.c */

#include <stdio.h>
#include <pthread.h>

#define printAttr(x) if (detS==(x))printf(#x"\n");

void printAttributes(const pthread_attr_t *myPthreadAttr_p)
{
  struct sched_param mySchedParam;
  int detS;

  pthread_attr_getdetachstate(myPthreadAttr_p,&detS);
  printAttr(PTHREAD_CREATE_JOINABLE)
  printAttr(PTHREAD_CREATE_DETACHED)

  pthread_attr_getinheritsched(myPthreadAttr_p,&detS);
  printAttr(PTHREAD_INHERIT_SCHED)
  printAttr(PTHREAD_EXPLICIT_SCHED)

  pthread_attr_getscope(myPthreadAttr_p,&detS);
  printAttr(PTHREAD_SCOPE_SYSTEM)
  printAttr(PTHREAD_SCOPE_PROCESS)

  pthread_attr_getschedpolicy(myPthreadAttr_p,&detS);
  printAttr(SCHED_OTHER)
  printAttr(SCHED_FIFO)
  printAttr(SCHED_RR)

  pthread_attr_getschedparam(myPthreadAttr_p,&mySchedParam);
  printf("Sched param%d\n",mySchedParam.__sched_priority);

}
```

Figure 15.7. pthread attributes – print function

This routine will be used again shortly as we modify thread attributes. There are three attributes associated with establishing a customized stack for a thread. These parameters are stackaddr, stacksize, and guardsize. These calls are mandatorily supported by SCA-compliant systems but the user is cautioned that their use could create portability problems because stack management schemes are different from platform to platform.

The detachstate attribute allows the user to create a thread that is joinable or detached. A detached thread is a thread with memory storage that can be reclaimed by the system after the thread terminates. A detached thread cannot be 'join'ed. A joinable thread can be detached after it's created by calling the `pthread_detach` system call. The only requirement from the POSIX standard is that `_POSIX_THREAD_PRIORITY_SCHEDULING` be defined, which is mandatory on SCA systems.

The priority scheduling of threads is a powerful mechanism that allows the applications programmer precise contol over how components react to events within the system. There are three attributes that affect priority scheduling: `scope`, `inherit sched`, and `sched policy`. The first, scope, has meaning only if `_POSIX_PRIORITY_SCHEDULING` is supported. An SCA application cannot make such an assumption. The second, inheritsched, establishes whether or not a created thread inherits its policy and priority from the caller of `pthread_create` or whether it must be set explicitly. In the Linux system the default is to inherit scheduling policy and priority from the caller. In accordance with the POSIX standard a thread created with the INHERIT attribute will ignore the fields schedpolicy and schedparam (priority) set by the user. Thus in our Linux system the user must setinheritsched to `PTHREAD_EXPLICIT_SCHED` in order to be able to change the schedpolicy and schedparam setting.

The default scheduling policy for Linux is `SCHED_OTHER`. Prior to the release of the 2.6 kernel this was the only policy supported. The OTHER policy is implementation dependent and in the case of the old Linux threads were set up as standalone processes and ran along with all the other processes in the system. There was nothing the programmer could do to assert the importance of his or her thread/process over that of all the other processes. The Linux scheduler gives everyone a chance to run and you have no control over when. Threads responding to events had to wait their turn – somewhat of a *laissez faire* scheduling policy. Our software radio system benefits from real-time responsiveness, thus we leave `SCHED_OTHER` to spreadsheet programs and the like.

Linux 2.6 and glibc/NPTL 2.4.3 support two real-time scheduling policies – `SCHED_RR` (round robin) and `SCHED_FIFO` (first in first out). Both are pre-emptive and priority-based. Any thread having either of these policies specified must be run with super-user privileges, and any such thread will pre-empt and out-prioritize any thread created with the policy `SCHED_OTHER`. POSIX does not specify if priorities of higher numerical value have higher priority or vice-versa. It does provide two system calls `sched_get_priority_max()` and `_min` for the user to determine what the minimum and maximum priorities are for each priority policy. Table 15.3 shows these priority limits for Linux 2.6.

The pre-emptive real-time scheduler operates with very few rules. Envision lists numbered 99 (high priority) down to 1 (low priority) for `SCHED_FIFO` and lists numbered 99 down to 1 for `SCHED_RR`. (Remember that other implementations might reverse the numerical meaning of priority.) At each priority level the list is composed of a sequence of threads

Table 15.3. Default min and max priorities

Policy	Min priority	Max priority
`SCHED_OTHER`	0	0
`SCHED_FIFO`	1	99
`SCHED_RR`	1	99

with a head and a tail. The head thread at a particular priority is the thread that's been sitting in that list the longest. The tail is the newcomer to that list. This construct – call it the execution queue – represents all threads able to run. If a thread is blocked it is not run-able and does not appear on this execution queue. Here are the six rules:

1. The head thread having the highest priority – both lists considered – is granted access to the CPU.
2. If the running thread is pre-empted by a higher priority thread it will be returned to the head of the list from which it came.
3. If the running thread performs a `sched_yield` it will return to the tail of the list from which it came.
4. If a running thread becomes blocked it will not be returned to the queue.
5. When a blocked thread becomes run-able it will be returned to the tail of the appropriate list.
6. A thread whose policy or priority is modified is moved to the tail of the new list.

The real trick in real-time design is to keep the execution queue empty. When a thread becomes run-able it should go straight to the CPU. The following POSIX artifacts can generate a blocking condition: message queues, semaphores, mutex, and condition variables. When an event on one of these artifacts cause a thread to become run-able, that thread is added at the tail of the correct priority list. Despite the fact that the SCA does not mandate `_POSIX_PRIORITY_SCHEDULING` once we understand that the prioritization mechanism is built in to the fabric of the operating system we can conclude that if priority scheduling is there for threads it is also there for processes.

The following example (Figure 15.8) shows how to invoke the real-time pre-emptive scheduler on Linux 2.6.

In this simple code example a thread is created under the `SCHED_FIFO` policy having a priority of 5. Once compiled and run as a user the following output is expected – the `pthread_create` fails with the following message – line 36:

```
1. /* attr2.c */
2.
3. #include <stdio.h>
4. #include <pthread.h>
5. #include <string.h>
6. #include <stdlib.h>
7.
8. void client( );
```

Figure 15.8. Priority scheduling

```
void printAttributes(const pthread_attr_t* );

static const struct timespec tenthSecond = {0,100000000};

int main()
{
    pthread_t privledgedThread;
    pthread_attr_t myPthreadAttr;
    struct sched_param mySchedParam;
    int retVal, detS;

    pthread_attr_init(&myPthreadAttr);

    detS = PTHREAD_EXPLICIT_SCHED;
    pthread_attr_setinheritsched(&myPthreadAttr,detS);

    detS = SCHED_FIFO;
    pthread_attr_setschedpolicy(&myPthreadAttr,detS);

    mySchedParam.__sched_priority = 5;
    pthread_attr_setschedparam(&myPthreadAttr,
                    (const struct sched_param *)&mySchedParam);

    retVal = pthread_create(&privledgedThread,
       (constpthread_attr_t*)&myPthreadAttr,(void*)client,
        (void*)NULL);
    if (retVal!=0)
    {
       fprintf(stderr,"pthread_create() error: %s\n",
       strerror(retVal));
       abort();
    }

    /* At this point the other thread is running at a high priority */

    printAttributes((const pthread_attr_t *)&myPthreadAttr);
    pthread_attr_destroy(&myPthreadAttr); /* done with
    pthread_attr */

    /* directly query the thread for its priority */
    pthread_getschedparam(privledgedThread,&detS,&mySchedParam);
    printf("Query priority %d\n",mySchedParam.__sched_priority);

    pthread_join(privledgedThread,(void **)NULL);

    return retVal;
   }
```

Figure 15.8. (Continued)

```
53.
54.    void client()
55.      {int i,j;
56.
57.  for (i=0; i<100; ++i)
58.  {
59.       for (j=0; j<3000000; ++j) ; /* burn up some CPU */
60.       nanosleep(&tenthSecond,NULL);
61.  }
62. }
```

Figure 15.8. (Continued)

```
pthread_create() error: Operation not permitted
```

The executable – say a.out – must be modified to run with superuser privileges. Log in as super-user and type:

```
root# chown root:root a.out

root# chmod u+s a.out

root# exit
```

Now the executable will run to completion. The user can call up all the running processes in order of CPU usage by using the top command and see our task running at the top of the list at some xth% CPU Utilization. Note the nanosleep in line 60. With the exception of the kernel our thread is running at higher priority than everything else on our Linux box. For instance we must give X windows a chance to run otherwise no keyboard input, etc. Without the `nanosleep` (a `sched_yield` will not work) your system will most likely crash – no keyboard, no ctrl-alt-delete – you'll probably have to unplug it to get it to reboot. Wouldn't it be nice if we could get the kernel to perform a timeout operation for us just enough to keep the system alive?

The `SCHED_RR` policy works exactly like the FIFO policy with one exception. A round-robin thread is only allowed to run for a certain period of time before it is moved from the CPU to the tail of its priority list. This interval can be queried with the `sched_rr_get_interval` system call. In the Linux 2.6 kernel this number is 99.9848 milli-seconds. The operating system runs on a 1/100th of a second tic timer. Thus a round-robin thread will be pre-empted in order to keep the system alive. In the previous code example change line 25 to `SCHED_RR` and get rid of the `nanosleep` in line 60. Under FIFO scheduling this would lock up your machine but under round-robin scheduling the system stays alive and the thread executes optimally to completion.

15.5 Conditional Variables

The final POSIX thread synchronization method to be considered is the thread conditional variable. A conditional variable is used in conjunction with a mutex. The combined effect of the conditional variable and associated mutex is the same as that of a semaphore. The difference is that a semaphore counts `sem_post` operations so it is possible for the thread on one side of the semaphore to get out of sync with the thread doing the `sem_wait`. However in conjunction with the mutex a conditional variable is atomic and non-counting. A thread signaling the condition variable can do so all day without getting out of sync with the thread doing the `pthread_cond_wait`. Figure 15.9 shows the operations and attributes on the conditional variable.

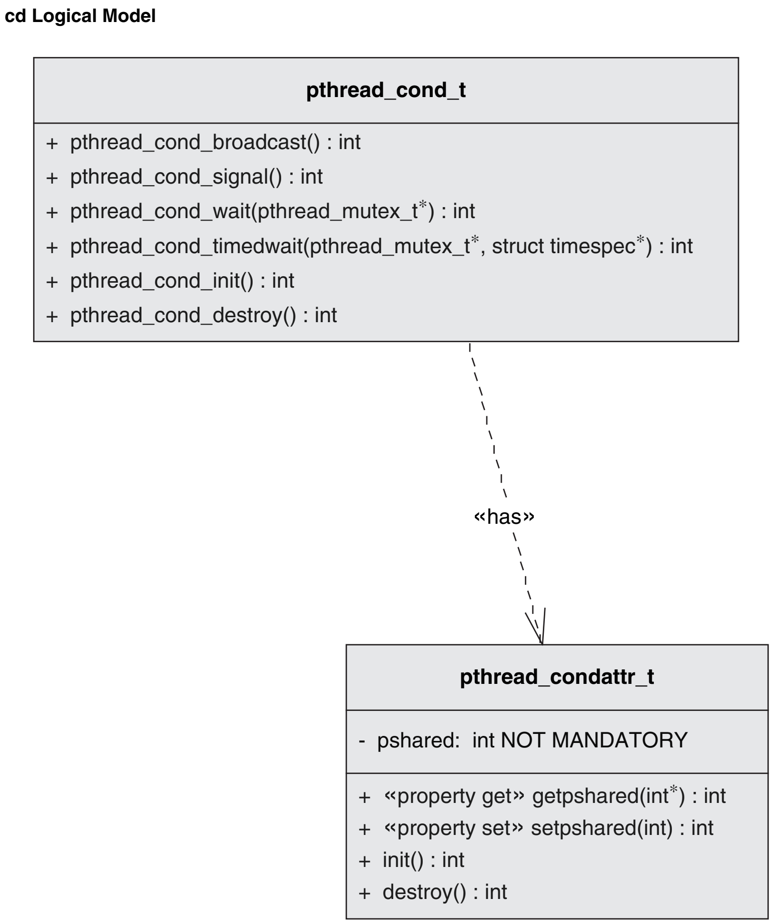

cd Logical Model
pthread_cond_t
+ pthread_cond_broadcast() : int
+ pthread_cond_signal() : int
+ pthread_cond_wait(pthread_mutex_t*) : int
+ pthread_cond_timedwait(pthread_mutex_t*, struct timespec*) : int
+ pthread_cond_init() : int
+ pthread_cond_destroy() : int
«has»
pthread_condattr_t
- pshared: int NOT MANDATORY
+ «property get» getpshared(int*) : int
+ «property set» setpshared(int) : int
+ init() : int
+ destroy() : int

Figure 15.9. pthread_cond_t component diagram

Since `POSIX_THREAD_PROCESS_SHARED` is not required in SCA-compliant systems the entire need for the attributes structure and its four system calls goes away. Default *cond* variable attributes can be specified for a statically declared cond variable with the following initializer:

```
pthread_cond_t myCondVar = PTHREAD_COND_INITIALIZER;
```

Equivalently, in the call to `pthread_cond_init` the user can pass a NULL pointer for attributes field. In the prototype for `pthread_cond_wait` the user is required to specify a corresponding mutex. The code segment in Figure 15.10 shows the use of the POSIX thread conditional variables in our client-server adding machine program.

```
/* thread3.c */

#include <stdio.h>
#include <pthread.h>

#define DC_NO_VALUE 0x00000000
#define DC_TOSERV_OK 0x00000001
#define DC_TOCLNT_OK 0x00000002
#define DC_QUIT 0x00000003

typedef struct {
  pthread_cond_t forMsg;
  volatile int putDataHere;    /* <-- this field AND */
  volatile int ctrlCode;       /* <-- this field protected by a mutex */
  pthread_mutex_t protect;
                } sharedMem;

void client(sharedMem* );
void server(sharedMem* );

int main()
{
   pthread_t clientT,serverT;
   sharedMem msgQ = {
                PTHREAD_COND_INITIALIZER,
                0, DC_NO_VALUE,
                PTHREAD_MUTEX_INITIALIZER };

   /* create threads, pass address of msgQ */
   pthread_create(&serverT,
        (const pthread_attr_t *)NULL,(void *)server,(void *)&msgQ);
   pthread_create(&clientT,
        (const pthread_attr_t *)NULL,(void *)client,(void *)&msgQ);

   /* wait for threads to terminate */
   pthread_join(clientT,(void **)NULL);
   pthread_join(serverT,(void **)NULL);

   return 0;
}

void client(sharedMem* mQ_p)
```

Figure 15.10. Condition variables

```
{
  int i,testData[]={12,2,7,0};
  volatile int localCopy;

  for (i=0; i<4; ++i)
  {
      pthread_mutex_lock(&(mQ_p->protect));
         mQ_p->putDataHere = testData[i];
         mQ_p->ctrlCode = DC_TOSERV_OK;
      pthread_mutex_unlock(&(mQ_p->protect));
      pthread_cond_signal(&(mQ_p->forMsg)); /* send */

      pthread_mutex_lock(&(mQ_p->protect));
         while (mQ_p->ctrlCode!=DC_TOCLNT_OK)
      pthread_cond_wait(&(mQ_p->forMsg),&(mQ_p->protect));
         localCopy=mQ_p->putDataHere;
      pthread_mutex_unlock(&(mQ_p->protect));

   printf("Sent %d Received %d\n",*(testData+i),localCopy);
  }

  /* send shutdown */
  pthread_mutex_lock(&(mQ_p->protect));
     mQ_p->ctrlCode = DC_QUIT;
  pthread_mutex_unlock(&(mQ_p->protect));
  pthread_cond_broadcast(&(mQ_p->forMsg));

}

void server(sharedMem* mQ_p)
{
   while (1)
  {
      pthread_mutex_lock(&(mQ_p->protect));
         while  ((mQ_p->ctrlCode!=DC_TOSERV_OK) &&
                 (mQ_p->ctrlCode!=DC_QUIT))
             pthread_cond_wait(&(mQ_p->forMsg),&(mQ_p->protect));
         if ( mQ_p->ctrlCode==DC_QUIT ) break;
         mQ_p->putDataHere += 5;
         mQ_p->ctrlCode = DC_TOCLNT_OK; /* valid msg to client */
      pthread_mutex_unlock(&(mQ_p->protect));
      pthread_cond_signal(&(mQ_p->forMsg));
   }
   pthread_mutex_unlock(&(mQ_p->protect));

}
```

Figure 15.10. (Continued)

There are a few very important items to note in the program in Figure 15.10. Our common memory is protected by a mutex – that's the same as before. However, the semaphores have been replaced by a single conditional variable – line 12. Default static initializers are used to initialize the conditional and mutex variables as well as the message code – lines 24–27. The threads are created and the main program waits for the threads to terminate.

The client code puts a lock on the mutex in order to write data and message code to the common area. The mutex lock is released and the data is 'sent' to the server via the `pthread_cond_signal` command. Now suppose that at this instant no one is waiting for data. Even if someone were blocked waiting on the condition variable, unless they are of a higher priority (and real-time scheduling is enabled) there is likely to be no context switch upon execution of line 53. So the client code continues in line 55 and again locks the data structure. The client checks the message code to see if there is a message there for it. This is the pre-condition or predicate. The only message present is the one the client just put there and that is destined for the server. So the client calls `pthread_cond_wait` which atomically executes the following actions. It unlocks the mutex, registers its reason for blocking – condition and associated mutex – and then blocks. With the mutex unlocked, another thread is free to access the common data area. In this case the server thread is free to run whether or not it was waiting for the mutex. As a thought experiment the reader should also mentally walk through the execution case where the server runs first – before the client. As a matter of course any thread should always proceed with all due haste to the point where the mutex guarding the common area is unlocked. Subsequently any thread should then proceed without impediment to the point where it blocks or acquiesces with `sched_yield`. The condition wait function does both safely and atomically – it unlocks the mutex and announces its desire to take a nap until a particular condition occurs. Never, ever, ever should any thread lock a resource and then take a nap.

On the server side, line 76, the server attempts to lock the mutex. Upon acquiring the mutex the server checks the message code to see if this message is for it. A good way to interpret the Boolean expression in lines 77 and 78 is to consider what constitutes a valid message and negate it using DeMorgan's theorem.

message for me:	`if ( (msg==TOSERV) OR (msg==QUIT) )`
message NOT for me:	`if ( (msg!=TOSERV) AND (msg!=QUIT) )`

If it is not any of the messages the server is looking for, the server uses `pthread_cond_wait()` to relinquish the mutex and block for someone to signal the presence of a message – line 79. What's similar to the semaphore method is the need to check the predicate before entering the wait and after coming out of it – line 77-79. What's different is the fact that because the condition variable and mutex are atomically coupled the directionality of the message is handled by the message code and not by having to post a send semaphore versus a receive semaphore. In this final example – two semaphores – an artifact that has been with us from the beginning is replaced with a single condition variable.

15.6 Less Interesting Thread Calls

As much fun as we're having with POSIX threads, a main stay of SCA-compliant operating environments, it's time to put a wrap on it. The focus of the examples in Chapters 14 and 15 has been the movement of data and control between independently executing code sequences. This need for timely response to asynchronously occurring events and data is the hallmark of an SCA-class software radio design. This is in almost complete contradiction with our legacy radio designs where the system was synchronous from the modem all the way to the data port. The reader might be motivated to explore POSIX threads as it relates to other architectural constructs often found in software radios. This includes multiplexing multiple data streams into a single stream (fan-in), de-multiplexing a single stream to numerous output streams (fan-out), simulcasting a single stream to multiple sinks (broadcast), and finally synchronizing commands and data when they originate from different elements within the system.

The several remaining POSIX thread calls that we now consider are:

```
pthread_setcancelstate()     pthread_testcancel()
pthread_setcanceltype()      pthread_once()
pthread_cancel( )            pthread_sigmask()
pthread_cleanup_pop( )       pthread_kill( )
pthread_cleanup_push( )      pthread_exit( )
pthread_equal( )             pthread_self()
pthread_key_create( )        pthread_key_delete( )
pthread_getspecific( )       pthread_setspecific( )
```

There are several ways to terminate a thread. In our examples we embraced the notion of a QUIT command to tell threads to exit gracefully. The main program waited – `pthread_join` – for the threads to exit and the entire process then exited. Our expectation of real world scenarios is that there can be many different kinds of shutdown and restart events and that they can originate from many different places within the system. The normal means of exiting a thread is to just run out the context, using a conventional return (with return code) or by calling `pthread_exit` – also with return code. The other method `pthread_cancel` allows one thread to wipeout another. First of all a thread must choose to be cancellable by calling `pthread_cancelstate` with a value of `PTHREAD_CANCEL_ENABLE`. A thread can also use `pthread_canceltype` to establish deferred or asynchronous cancellation. An important aspect of both types of cancellation is that they don't occur instantaneously but more or less when the system gets around to it. Finally a thread can test for pending cancel requests by calling `pthread_testcancel`. If a cancellation request is pending this call will not return.

If a thread is being cancelled, the odds are that it's not going to be leaving the context via the normal path. Normally a thread will release all its resources and signal any other threads it needs to as part of exiting gracefully. A user is allowed to specify cancellation handlers. These are functions that are called as part of exiting or cancelling. Handlers are pushed and popped with the `pthread_cleanup_push` and `pthread_cleanup_pop` commands. As a thread is leaving its context the handler stack is traversed in reverse order. Like normally exited threads cancelled threads are joinable.

The specific implementation of the thread type, *`pthread_t`*, is unknown to the user. POSIX provides two functions for retrieving and comparing thread types: `pthread_self` and `pthread_equal`.

We close with discussion of the variable *`pthread_key_t`*. Code examples are provided in reference [12]. The usual use of the key variable is to provide a thread-friendly approach to global variables. Users understand that global variables allow separate compilation units to access a variable without having to pass that variable as a parameter in the function call. Legacy usage of global variables assumes one thread of execution per process. As before, in a multi-threaded program, a global variable is accessible, via *extern*, to all compilation units but now it is additionally accessible across all threads. Now the single global variable is being acted upon by forces essentially unknown to the single-threaded legacy software. What's needed is a global variable – visible across all compilation units – wherein each thread has its own private copy: that is the function performed by keys.

The user initializes a key which has been declared at global scope using the command `pthread_key_create`. Subsequently each thread of execution binds the key to a particular thread-specific value using the call `pthread_setspecific`. Although all threads see the same key declaration, the value the key is pointing to is specific to each thread. Threads access that thread-specific value with the call `pthread_getspecific`. Essentially each thread now has its own private version of the global variable.

Many components of the SCA-compliant software radio application live on the same general purpose processor. Portable applications will confine interactions of those components to the set of POSIX system calls identified as mandatory in Appendix B of the SCA. The next chapter explores application-level compliance of components that must interact from processor to processor.

16

All ORBS are not Created Equal

The language of the SCA is the Common Object Request Broker Architecture or CORBA. Prior to its incorporation in software radio CORBA was, and still is, the language of international commerce. Airline reservation systems, as well as banking and business systems, have long enjoyed the magic of CORBA even before the moniker 'Enterprise' was descriptively attached to this kind of technology. It is this genesis that we must bear in mind: CORBA was not invented for software radio. It was a marriage of convenience that allowed SCA to achieve its portability and platform independence goals while building on a well established commercial technology. Like most marriages, software radio and CORBA is not without its share of challenges. A principal reason for these difficulties is the fact that CORBA was developed for distributed systems consisting of hundreds if not thousands of nodes. In fact, nodes might be joining or retiring from the domain at any time. An underlying principle in the development of CORBA is the ability for any client to talk to any server – anywhere. A CORBA application might be an inventory program that connects to thousands of remote sites and then generates shipping manifests. This application runs over the course of many minutes – even hours – to completion. CORBA provides for an underlying design of asynchronous, loosely-coupled timeless communications.

Now if we map this model unto the software radio domain we begin to see some disconnects. First, a software radio typically has a limited number of computational nodes. Until recently nodes such as DSPs and FPGAs are not even CORBA-capable. In a software radio computational nodes are statically defined within and across power cycles. It can be argued that for portability reasons CORBA is required to allow applications to run on a five processor system versus a two processor system. In the world of tactical radios such a use case is rare. Furthermore, waveform porting experiences are now considered with respect to a very limited number of platforms. Whereas the SCA originally was pitched as enabling any application to run anywhere on anything, a more sober view of application portability now prevails. It is hard – in fact, impossible – to write an application that runs on anything anywhere. Consider this even from the standpoint of selling the software. There will be a finite number of tests conducted on a finite number of platforms. To then conclude that the software is portable to any form factor or target hardware is sheer marketing.

Much of this has to do with the relationship between the application and its devices. A commercial enterprise application might have a relationship with an output device, say a printer, that is very de-coupled. You can batch print anything. Now compare this to a software radio application that needs to multiplex messages unto a 40 milli-second TDMA frame – there is a huge difference.

A second disconnect comes from the need for information assurance and isolation. Because of security concerns in the software radio one might specifically desire that objects not be able to talk to each other. Consider the black-side of the radio which is traditionally the part of the radio that is 'in the clear'. What's to prevent a networked black-side component – even a 'friendly' component – from telling a channel to stop transmitting? No protocol intrinsically exists within POSIX, CORBA, or the SCA to provide authentication or enforce the concept of privileged access. (We do not consider age-old implementations of message queues over the top of a traditional Unix file system. In the interest of execution speed, modern day real-time operating systems, though claiming POSIX-compliance, dispense with the file system and the notion of privileged access. The reader should keep separate the concept of privilege of execution, i.e. priority, from the concept of privilege of access, i.e. root, group, user. The reader is also encouraged to understand that operating systems that do support the notion of partitioned memory are required by SCA POSIX to support inter-process communication mechanisms that are meant specifically to allow these partitions to share data with each other.)

The final disconnect is the timelessness of CORBA. Typical software radio applications – especially legacy applications which evolved from single purpose hard-wired radios – are required to complete certain transactions in mill-seconds or fractions thereof. CORBA, even real-time CORBA, is just not set up for this.

Despite these inconveniences it is possible to develop portable applications that meet real-time requirements. Part of this is art-form, for instance, allocating functional requirements properly across sub-systems. Another part is not expecting CORBA to do what it was not designed to do. A final part is trial and error: literally, 'well that didn't work, let's try this'. Unfortunately the need to make an application work on platform X might undermine its ability to work on platform Y.

In order to write truly portable applications, they must be specified that way from the onset of the development. One can indeed specify that an application run on anything anywhere but owing to the impossibility of testing such a scenario it is a foolish requirement. At the end of the day a vendor is paid to make waveform X work on platform Y or in the case of a portable waveform, Y, Z, and α. The early validation work on the SCA demonstrated successful porting by identically specifying operating environments and target platforms. The entire executable image, ORB, Core Framework, and application was 'ported' to identical OS/hardware. In reality target radio systems will utilize different hardware. One radio might be a 4-channel radio at a Tactical Operations Center (TOC), the other a radio controlled munition, hundreds of which can be dropped from an aircraft.

The approach used by a waveform developer to accommodate differences in platform hardware is a key element missing from the JTRS program. It has proved very difficult to define a generic radio API that overlays the many different radio topologies. Part of that difficulty is that porting has been considered as an afterthought, i.e. something you do at the end of development. If portability is to become a reality, specific language defining portability must be written in to the RFP. CORBA is to be considered nothing less than a portability enabler. What must be understood by the applications programmer is that an ORB must be used with cognizance and care. It is not a magic pill that will make all your portability problems disappear. In fact misappropriation of CORBA, or even utilization of CORBA in the software radio in a manner appropriate to other domains, can exacerbate application portability.

16.1 CORBA Basics

An object publishes its services via the Interface Definition Language (IDL). IDL is a canonical language used in CORBA to define remote interfaces. It is independent of any other language. IDL defines an interface – not who, what, where, or how it is implemented. IDL's syntax is declarative for types and interfaces only. Even attributes are mapped to `_set()` and `_get()` interfaces. IDL supports user-defined types: that is, the user can define data structures of arbitrary complexity. IDL's strength is that it can be mapped to many languages including non-OO and scripting languages. IDL can not only be encoded to operate over different transports via the General Inter-ORB Protocol (GIOP), but also via specialized protocols or Environment-Specific Inter-ORB Protocols (ESIOP). IDL code generators are supplied by vendors and map IDL into various implementation languages. At the lowest level IDL is a definition between a client and a server of some 'binding contract' to which they must adhere in order to accomplish a message exchange. Once the IDL is defined and successfully compiled by a language specific code generator, the resulting code contains stubs and skeleton entities. The definition of these structures comes from the CORBA language mapping for that language. An applications programmer is encouraged to examine, but not edit, these machine-generated entities. The complexity of these files is somewhat intimidating but rest assured they contain all that is necessary for clients and servers to communicate through the CORBA machinery.

Figure 16.1 provides a top-level view of the components involved in a CORBA message exchange.

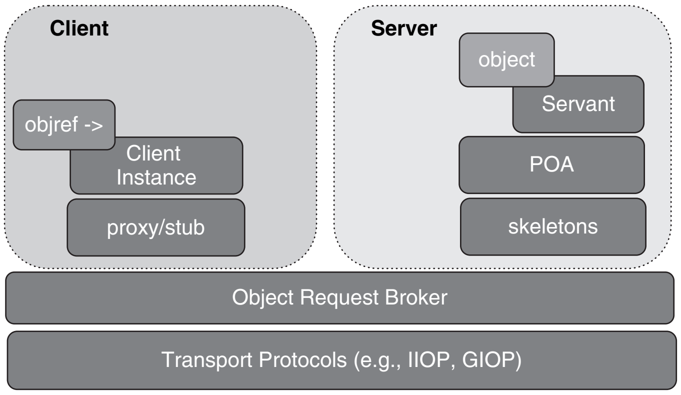


Figure 16.1. CORBA constituents

An object with an interface published in IDL runs with the help of a servant. For an example our object will be a radio. Figure 16.2 shows the IDL for our radio object.

```
1. module anExample {
2.
3.      interface radio {
4.
5.           readonly attribute float frequency;
6.
7.      };
8. };
```

Figure 16.2. Radio IDL

This example consists of a single object, `radio` in line 3, and a single attribute, `frequency` in line 5. We are informed that a user cannot modify the frequency: it is `readonly`. This implies that the object has some private, non-CORBA, means of initializing the frequency. This example is then compiled with an IDL compiler into, let's say, the C language with the simple command.

```
orbit-idl-2 radio.idl
```

Several files are produced: a header, stubs, common, and skeleton (`radio.h`, `radio-stubs.c`, `radio-common.c` and `radio-skels.c`). These stub and skeleton components are visible in Figure 16.1. A small sampling of the header file shows the 'C' function prototype that is generated by the IDL compiler.

```
CORBA_float
  anExample_radio__get_frequency(
     anExample_radio _obj, CORBA_Environment *ev);
```

In accordance with the 'C' Language Mapping [13], the IDL compiler automatically generates `__set()` and `__get()` functions for all attributes defined in the IDL. Since a CORBA object is likely to be running on a different processor (as well as different OS and native language implementation), it is meaningless to have an attribute declared like a variable – recall that a normal 'C' variable is essentially a memory address. Such a memory reference would be useless because the machine hosting our radio object is likely to have different endianness than our own machine. Since frequency was declared as read only, a `__set()` function is not generated. Examination of the function prototype reveals a few key aspects of the 'C' Language Mapping. First is the name of the function. It is the module name followed an underscore followed by the interface name and two underscores then 'get' underscore and finishing with the name of the IDL attribute. Declarations can be nested and interfaces can be inherited. Unique function names are formed by concatenating module and interface names with underscores and finishing up with the name of the operation or attribute.

Second we see that the function call requires a reference to the object, more properly, the interface itself. Again the object name is formed by the concatenation of the module name, underscore, and interface name. In the 'C' language mapping all uniquely named objects are `typedef`'d as `CORBA_Objects`. This is one of the down-sides of the 'C' language map. There is no type safety. We could easily pass an object of the wrong type on the parameter list and the compiler would not be able to catch it because everything is a `CORBA_Object`. The type `CORBA_Object` is referred to as an opaque type because there is nothing in it that the user would want to know or even care about. It contains information useful only to the particular CORBA product being used.

The remaining item of interest in the function prototype is the parameter `ev` which is a pointer to a `CORBA_Environment` variable and the return value. We see that the attribute type 'float' maps to a 'C' type of `CORBA_float`. If one is motivated to go back through the vendor's ORB header files one will eventually find that `CORBA_float` is a `typedef` of the 'C' native type `float`. So underneath the covers the call simply returns a float which happens to be the frequency of the radio object. Finally the pointer to the `CORBA_Environment` variable is a pointer to a structure that will get filled in by the implementation should an error occur. For instance if an exception is thrown in processing the `__get()` request, then the value of `ev->_major` will be set to `CORBA_SYSTEM_EXCEPTION` or `CORBA_USER_EXCEPTION`.

Without describing the detail of how an object comes into existence, we'll just assume that it is running on the server. Our radio object might be written in Java running on an x86 Windows machine. Also, without describing how the client gets the radio object's object reference, we'll just assume that the client has access to the object reference. Recall the object reference is required as a parameter to the `__get_frequency()` call. So our client software is written in 'C' running on a PPC Linux machine. The two machines are connected via Ethernet. This transport layer could be anything; the client and radio server object don't care. The client makes the call to `__get_frequency()` in accordance with the prototype found in `radio.h`. From the client's perspective this is just a local function call that returns a `float`. Now the magic begins. Through code in the stub a connection is made to the ORB which uses information in the object reference to locate the servant hosting the radio object. Certain data is sent over the Ethernet through the skeleton software to the servant. This data might be what method to run and where to send the results of the call. The servant orders the object to run the `__get_frequency()` function (or the specific Java naming convention). Remember that this `__get_frequency()` function is written in Java by the radio vendor. The vendor kindly supplied the IDL file that we compiled in to 'C'. As a client we are really only interested in the stub code. The vendor software executes by reading the frequency register inside the radio hardware and returning the result back through the servant software, back over the Ethernet to the client software. All of the details of this rather significant transaction are completely hidden from the client-side. From the client's perspective `__get_frequency()` was just a local function call. CORBA took care of everything else.

16.1.1 Starting the Servant Object

So now we go back and answer some of the details that had been accepted earlier as assumptions. Let's address how the servant object comes into existence. The vendor creates the object implementation by writing code that implements the IDL. An empty function template for this implementation is generated by the IDL compiler. In our example the servant software would simply read the frequency register and return the result. Next the server-side ORB is initialized. A request is made of the ORB to provide a pointer to the RootPOA. Every ORB has one. POA stands for Portable Object Adaptor and it provides for a uniform means of addressing servant objects. Users are allowed to create as many POAs as they want but for our example the RootPOA is sufficient. Next the RootPOA is requested to create and activate a server. After that the `impl_anExample_radio__create()` function is called on the RootPOA to install the radio object on the servant. The return value from this call is the object reference for the radio interface. We'll figure out separately how to get this info to the client. Finally call the `CORBA_ORB_run()` interface so that the local ORB starts to listen for requests that it can then forward to the servant in the desired POA.

16.1.2 Accessing the Object Reference

As discussed, the software responsible for bringing up the server-side of the radio already has access to the radio's object reference as a return value to the `__create()` operation. Now, let's convey that information to the client software. Every ORB offers a function to create a human readable, text-based, version of the object reference call a stringified Interoperable Object Reference (IOR). This stringified IOR can be transmitted to the client via an email, a disk file, or it can printed out and typed in as part of the client software's command line. Once the client software has the stringified IOR it can be converted back to the opaque `CORBA_Object` type and then passed in the parameter list to the `__get_frequency()` call. The IDL prototypes for these easy-to-use IOR conversion routines are as follows:

```
interface ORB {
     string object_to_string (in Object obj);
     Object string_to_object (in string str);
};
```

Stringified IORs are universal in that they can be published by one ORB and read and interpreted by a completely different ORB. The exception to this rule is in the use of proprietary GIOP or Environment Specific IOPs. Vendors are allowed to extend or specialize Inter-ORB protocols and this would be likely to result in IORs that are not universal. However, every ORB is required to implement Internet Inter-ORB Protocol (IIOP) whether it's enabled by default or not. IIOP-based IORs are universal.

16.2 The Object Management Group

CORBA is the flagship product for the Object Management Group (OMG) which was founded in 1989 by 11 companies including Hewlett-Packard and American Airlines. The purpose of the OMG was to create a standard governing the deployment of distributed objects using all types of development environments on all types of platforms (see Wikipedia at http://en.wikipedia.org/wiki/Object Management Group). The first CORBA standard was released in 1991. The CORBA specification currently stands at revision 3.0.3. In the last several years the OMG product line has grown to include the Unified Modelling Language (UML) and Model Driven Architecture (MDA).

The OMG's pursuit of the MDA was well underway when the SCA was written but it is safe to say that SCA predates MDA. The JTRS program office continued to fund commercialization of the SCA through the OMG with large contributions from Raytheon, MITRE, Mercury Computer, and several other Technical Advisory Group (TAG) members.

This effort resulted in the formation of the Software Based Communication Domain Task Force (SBC DTF). The primary output of this group was the publication of the 'PIM and PSM for Software Radio Components' [14]. This new form of OMG specification, the Platform Independent Model (PIM), and the Platform Specific Model (PSM), are part of the MDA. The three primary goals of the MDA are portability, interoperability, and reusability through architectural separation of concerns.

The MDA intends to accomplish these goals by de-coupling a design from the platform. Two specifications – PIM and PSM – support this notion. The PIM describes a canonical detailed target system independent of any assumptions with respect to supporting platform and environment in which this system will exist. This model describes the conceptual 'grammars' without actually specifying any implementation detail or capability. The PSM is the mapping of a PIM into a particular technology domain with a view ultimately to building an implementation from it. It is a model of the target system specified by the PIM but in the context of how to use the underlying support platform such as, say, .NET, J2EE, CORBA or some other host middleware. A PIM can be mapped/transformed into any number of multitude of PSMs.

Around the time the SCA was being written, the OMG inherited the UML. The MDA essentially supports the marriage of the OMG's two flagship products – CORBA and UML – as well as leaving plenty of room to overlay other formerly competing technologies. If you can't beat them then abstract them. Figure 16.3 provides a notional flow of a design from PIM to implementation. The new UML 2.0 specification was authored with a view to supporting implementation of automatic code generation directly from the UML diagrams.

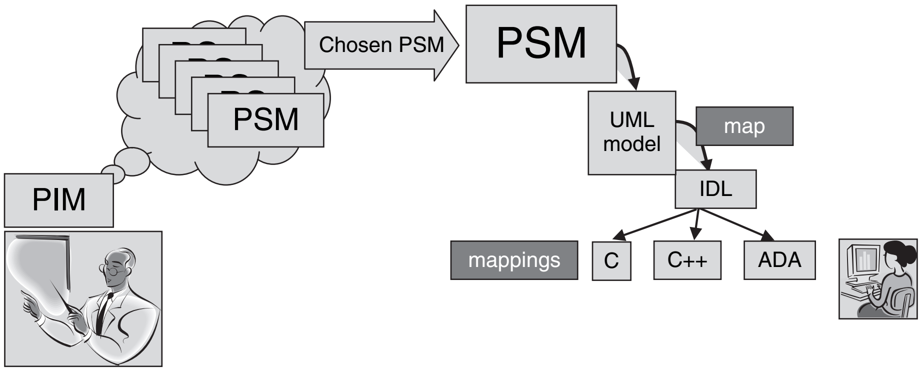


Figure 16.3. PIM to implementation design flow

Figure 16.3 shows the final 'mappings' step. In SDR usage CORBA IDL typically maps to C, C++, Ada, or Java. In fact language mappings exist for many other languages including COBOL, LISP, Python, Smalltalk, and XML. It is the function of the CORBA IDL compiler to generate language-specific files. This is how CORBA IDL achieves language independence. IDL cannot be executed: it must be compiled to native source code with can then be compiled to an executable.

16.3 'C' ORB versus C++ ORBs

Since language choice is like a religion for most programmers we will not play the fool and descend into petty comparisons. It is sufficient to say that language choice should be based on the requirements of the job and the skill set of the talent base. That being said, one of the biggest problem areas in CORBA programming is memory leaks. This is true independent of language choice. The key thing to learn in CORBA is who is responsible for cleaning up object references and variables that are no longer used. Unfortunately in CORBA these lines of responsibility, caller versus callee, are drawn on an almost case-by-case basis and are unique for every language mapping. It is beyond the scope of this book to engage in an in-depth study of memory management in CORBA. For SCA-compliant radios, mastery of CORBA's dynamic memory management policy is a must for every program. Experience has indicated that new programmers have more problems with CORBA itself than with the Core Framework.

Because of CORBA's need to allocate and de-allocate buffers dynamically, another set of problems begins to appear. These are memory fragmentation and run-time jitter. In Java the fragmentation issue is addressed through the use of garbage collectors. However this method somewhat compounds the jitter problem. Another method used is to pre-allocate buffer pools where the ORB itself takes on the role of pool manager [15]. The software radio applications programmer should take note of this technique in that 'C' or C++ code making heavy use of dynamic memory can also suffer the same ill effects. One must bear in mind that the software radio client base is used in radios that operate forever without crashing. For instance a radio that performs as a satellite uplink or an air traffic control operation is required to have 99.9999 % up-time over the course of a year. This equates to 30 seconds of down-time over that same year. The smallest memory leak will likely cause a system failure rate in excess of that requirement. With C++ dynamic memory management is often hidden beneath the covers. Depending on one's coding technique or even the compiler itself the mere invocation of a constructor can result in the heavy use of dynamic memory. We always advocate that whenever possible variables and buffers should be pre-allocated before entry into the run-time loop. This coding style is consistent with the SCA concept of instantiate, initialize, run, and clean-up.

There are a couple of features native to the C++ language that are not present in 'C'. For instance, when CORBA exceptions are mapped into 'C', a language that has no native support for exception handling, the result is somewhat of a hack. True exception handling makes provision for stack un-winding. This allows a low level exception to be 'handled' by the calling routine. There is a performance penalty that C++ incurs to offer this service but it is helpful in real-time systems to be able to implement a recovery strategy when bad things happen. In 'C' a low level routine must detect an exception before it occurs and then return to the calling routine the presence of the exception through the `CORBA_environment` variable. Without this pre-emptive action the machine exception will be handled by the operating system at the process level. Without a user-defined handler in place the process will just be terminated with some sort of nasty message.

As previously mentioned 'C' offers no type safety: that is, all objects are identical in the eyes of the compiler. There is no way for type violations to be caught at compile time. Should the ORB support it, and minimum CORBA does not, it is possible for 'C' code to access the Interface Repository to verify the identity of a particular object using the `is_a()` operation. This action however occurs in the run-time and not at compile time. Failure to detect an object type mismatch typically results in invoking an operation on an object that implements no such operation and will likely result in a `BAD_OPERATION` exception. In this chapter we make reference to minimum CORBA. This specification, maintained by the OMG, is called out as mandatory in SCA-compliant operating environments. The specification defines a subset of full CORBA 2.2 that supports implementation of ORBs in footprint constrained environments. A summary of minimum CORBA requirements can be found in Section 16.6.

Finally the C++ language offers the ability to perform late binding. This feature allows the designer to use polymorphism as a design technique. The concept of late binding is illustrated with this simple example: Say you have a parent quadrilateral CORBA object. There are child objects, square and parallelogram, that extend quadrilateral. Now, quadrilateral has a `compute_area()` operation that is overloaded by each of its children. So there exist specialized implementations of `square::compute_area()` and `parallelogram::compute_area()`. Now, C++ supports the ability to call `compute_area()` without specifically knowing whether it's a square or parallelogram. The run-time will figure that out and invoke the proper operation. Because all CORBA objects in 'C' are the same type it would be up to the user to figure out which function to call based on information extracted in the run-time from the Interface Repository. So C++ offers language features that allow certain CORBA constructs to be implemented naturally. In 'C' it is still possible to emulate these behaviors but it is up to the applications software to do so.

16.4 Initial Services

A call to `resolve_initial_references()` made with the string name of a particular service will return the object reference of that service object. As of CORBA 2.2, the basis for SCA's minimum CORBA specification, the following service names are reserved: RootPOA, POACurrent, and InterfaceRepository; and for CORBA Services, they are NameService, TradingService, SecurityCurrent, and TransactionCurrent. Of course in a minimum CORBA implementation InterfaceRepository will not exist. Rarely does a particular ORB implementation offer all of these services. IORs to the services must be provided on the command line when client or server code is started. The standardized naming convention for specifying these services on the command line is `-ORBInitRef` serviceName = IOR. For the ORBit2 orb the available initial services are RootPOA, POACurrent, and DynAnyFactory. Of course DynAnyFactory was added after the CORBA 2.2 and is not to be used by SCA-compliant application software. The string names of available services are returned by a call to `list_initial_services()`.

16.4.1 Starting a Client

The only call necessary to initialize a client with the ORB is a call to `ORB_init()`. This call returns a object reference to the ORB itself. A client can initialize into multiple ORBs but each must be uniquely identified. The object reference returned is a specialized object reference and cannot be stringified. The ORB interface itself is defined in IDL but is not subject to the same rules as user-defined IDL. It is called Pseudo Interface Definition Language or PIDL. Typical operations available to regular IDL, for example, `duplicate()` or `object_to_string()`, are not available on interfaces generated from PIDL. The code actually generated from PIDL is however subject to the same language mapping semantics as regular IDL. Other examples of PIDL pseudo-objects are RootPOA, Policy, and POACurrent.

16.5 The Interface Repository

The Interface Repository (IR) is that part of the ORB that maintains persistent storage of interface definitions. The IR allows the user to perform various type checking and inheritance graph tracing that might not be naturally supported by the language mapping. The IR not only manages these interfaces but also allows them to be accessed by distributed components – clients and servants alike. Unfortunately the IR is not guaranteed to be accessible in minimum CORBA ORBs. This is not to say that it doesn't exist within a particular ORB implementation; it's just not accessible from SCA-compliant code. To the applications programmer this means that the operations shown in Table 16.1 should not be used by SCA-compliant software.

Table 16.1. ORB/IR operations not supported in minimum CORBA

`get_interface()`
`is_a()`
`non_existent()`

16.5.1 Type Codes

Type codes are values that represent an argument or attribute of arbitrary complexity. Type codes are generated by the IDL compiler and can be found in the header file that is output. In the run-time the Interface Repository keeps track of type codes that can be put on the wire to support the Dynamic Invocation Interface (DII) or the CORBA *any* type. Because of its required use in the SCA Core Framework, the *any* type and therefore type codes must be (and are supported) by minimum CORBA ORBs.

Client re-construction of an extravagantly structured *any* type in a minimum CORBA system will, however, be limited. A client is expected to know, *a priori*, the details of complex structures, sequences, unions, or enumerations. The run-time operations used to re-construct complex user-defined structures dynamically are not available in a minimum ORB, e.g. `member_count()`, `member_type()`, etc. As far as the Core Framework is concerned, use of the *any* type is constrained and a client does, in fact, know through the Core Framework IDL just what to expect. The same is not true of the *PropertySet* interface. The SCA Appendix D, Domain Profile (XML) specification provides the ability to describe rather elaborate *Properties* constructs. To 'de-bounce' these structures, which are passed as *any*s, an application component's `configure()` or `query()` implementation would have to be privy to these structures for they will not be de-cipherable in the run-time. Before discussing object servants and Portable Object Adaptors (POA) it's good to examine all aspects of the minimum CORBA.

16.6 Minimum CORBA

Minimum CORBA is a subset of the full IIOP CORBA specification designed for systems with limited computational or memory resources. The writers of the minimum CORBA specification wanted to create a lightweight CORBA that could still interoperate 'on the wire' with other full-service ORBs. As a result minimum CORBA provides full support for the entire IDL including the *any* type and inheritance. A minimum ORB must be able to deal with TypeCodes including exception handling. De-ciphering complex data structures however becomes the compile-time responsibility of the client software. Minimum CORBA removes all aspects of the DII. In addition minimum CORBA removes the ability to access the Interface Repository. Table 16.2 summarizes all the features removed from the full CORBA specification in the creation of minimum CORBA.

Table 16.2. Features omitted in minimum CORBA

Dynamic Invocation Interface
Access to the Interface Repository
Dynamic Skeleton Interface
Dynamic Anys
POA Manager
Adapter Activator
Servant Manager
ThreadPolicy
ServantRetentionPolicy
RequestProcessingPolicy
ImplicitActivationPolicy
DCE Interoperability
COM Interworking
Interceptors
`work_pending()`
`void perform_work()`
`void shutdown()`

Table 16.2 implies that the only way of executing a servant object is `run()`. There is actually an error in the minimum CORBA specification called out by the SCA. The text of the document says to omit the `work_pending()`, `perform_work()`, and `shutdown()` commands. Then later in the same text the PIDL for minimum CORBA is given with these operations still available. A subsequent release of the minimum CORBA specification [16] fixes the problem and indeed these operations are not expected to be available in a minimum CORBA ORB. These missing commands are used when it is desired that the main program or thread hosting a servant object should be able to perform other tasks. The call to `ORB::run()` is a blocking call and will not return until a termination signal is received. Chapter 17 shows how to install a signal handler and cleanly shutdown the ORB under these circumstances. With an SCA-compliant ORB it is not possible for the thread of execution hosting a servant object to do anything else like hit a watchdog reset. Neither is it possible

for a process to `shutdown()` an ORB cleanly and then perform other tasks. Shutting down an ORB in an SCA-compliant system will take the whole process down.

This is not to say that an ORB thread can host only a single servant object. The next section on POAs covers the role of object adapters and multi-threaded ORBs that give the ability to service an unlimited number of servant objects on a single process. Before that, a short word is in order with regards to Interceptors. Interceptors are an optional extension to the ORB to allow implementation of the Replaceable Security option defined in the Security Service specification. This is important because security has recently become somewhat of a concern for SCA-compliant ORBs. Under the current SCA definition minimum ORBs interceptors are not supported.

In SCA-compliant radios the purpose of a component's SCD is to provide that same Interface Repository information through the XML that is no longer available via a minimum CORBA ORB. Minimum CORBA also eliminates the notion of a CORBA *abstract* type though this feature was added to CORBA after the 2.2 specification. Minimum CORBA supports a reduced set of language mappings keeping the all-important 'C' and C++ mappings. Although the SCA 2.2.1 references an obsolete OMG minimum CORBA specification [1], it is likely that newer versions of the SCA will reference the more current version of minimum CORBA [16].

16.7 The Portable Object Adapter (POA)

The following discussion focuses on the servant side of a CORBA transaction. Although described in a client-server context, CORBA supports the ability to connect objects in a manner so as to support objects that are both servants and clients. Furthermore, CORBA supports objects that have different persistence and threading models. This flexible framework is supported through the POA. Though the minimum CORBA POA is somewhat constrained over its fully-featured CORBA 2.2 counterpart, it is still quite flexible in supporting different servant characteristics. Using the previous section on minimum CORBA as a guide we will step through the capabilities of the minimum POA.

Initially, we wish to understand the relationship between a POA, a servant, and an object implementation. Figure 16.4 shows this relationship – the shortened version of an implementation is called 'impl'.

Figure 16.4 shows multiple POAs. An ORB provides a single initial POA called a RootPOA other POAs can be created on the RootPOA – more on that later. A POA can host numerous servants, which on the same POA share a common trait called Policy.

Figure 16.4 shows the notion of an object as a collection of operation implementations. Each operation on an active object is mapped to a servant. A servant can be mapped to support more than one operation on an object or operations on different objects. Consider the right half of the figure. Here numerous clients are mapped to a single servant. Additionally there exist servants that are mapped to no operations. When a server is mapped to an object implementation an entry called an ObjectId is made into a table called the Active Object Map. An Object Id is a value used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients.

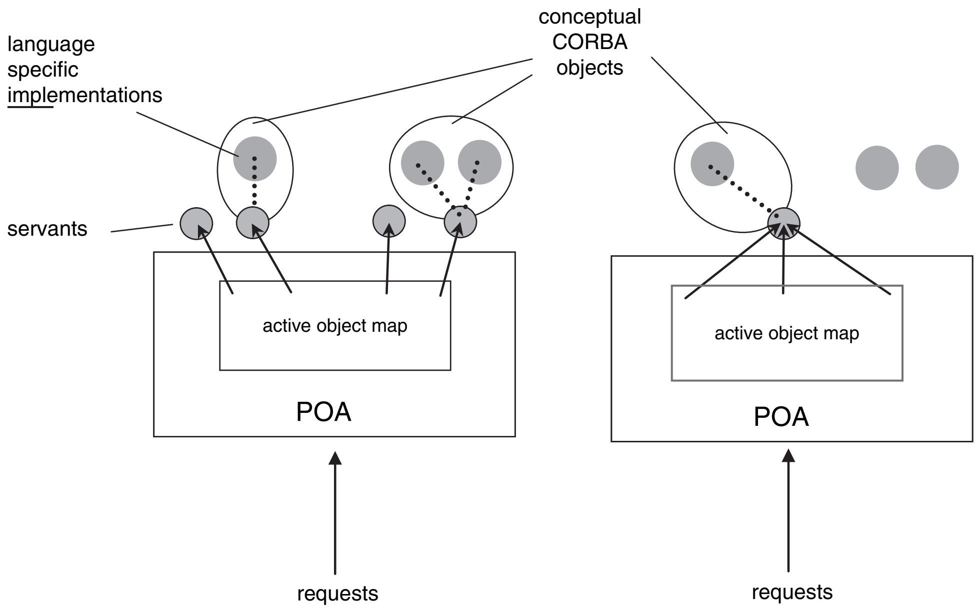


Figure 16.4. POA–servant–impl relationship

16.7.1 Policy

For the simplest applications, servants can be assigned to the universally available RootPOA. The RootPOA pseudo-object reference is available via the call `resolve_initial_references("RootPOA")`. The pseudo-reference returned cannot be stringified and has no meaning outside of the process space that invoked the call, i.e. the server process space. The only restriction on using the RootPOA is the fact that its run-time characteristics, as defined by Policy objects, are immutable. The user can create one or more new POAs on the RootPOA. Here the user is allowed to specify, at the time of creation, new Policys for the created POA. Because of limitations of minimum CORBA certain Policys remain immutable even for newly created POAs. Table 16.3 lists all POA Policy types, the default value for the RootPOA, and whether the Policy is mutable for new POAs.

Table 16.3. POA policies

Policy type	RootPOA value	Mutable under minimum ORB?
IdAssignmentPolicy	`SYSTEM_ID`	Yes
IdUniquenessPolicy	`UNIQUE_ID`	Yes
ImplicitActivationPolicy	`IMPLICIT_ACTIVATION`	No
LifespanPolicy	`TRANSIENT`	Yes
RequestProcessingPolicy	`USE_ACTIVE_OBJECT_MAP_ONLY`	No
ServantRetentionPolicy	`RETAIN`	No
ThreadPolicy	`ORB_CTRL_MODEL`	No

Now that we know the default policies and whether they're mutable or not, let's consider the resulting run-time behavior, starting with policies that describe the RootPOA that are immutable under minimum CORBA. ThreadPolicy dictates the threading mechanism that the POA will use to service a call. The default `ORB_CTRL_MODEL` indicates that servant(s) are serviced in single or multi-threaded fashion by the POA at the discretion of the ORB. ImplicitActivationPolicy is used to indicate whether servant activation is a separate user-defined action or whether the POA will automatically perform activation when a client invokes a request. The default `IMPLICIT_ACTIVATION` indicates that the POA will manage activation as required. RequestProcessingPolicy is used to tell the POA how to look up target objects for a request. Minimum ORBs support only the use of an Active Object Map (AOM). ServantRetentionPolicy is used to indicate to a POA whether or not to use an AOM. IdAssignmentPolicy is used to indicate whether keys to identify objects (ObjectIds) are to be assigned by the application developer `(USER_ID)` or by the POA `(SYSTEM_ID)`. Because of `IMPLICT_ACTIVATION`, it is also required that RETAIN and SYSTEM_ID be in effect. In other words the POA automatically generates and manages Object Ids. Table 16.3 indicates that the user can take control of the generation of Object Ids. That would actually get implemented by back-end servants being activated, managed, and known only to a single POA-generated servant.

The mutable policies include IdUniquenessPolicy and LifespanPolicy. A POA's IdUniquenessPolicy dictates whether servants activated on the POA can be associated with only one `(UNIQUE_ID)` or one or more `(MULTIPLE_ID)` Object Ids. Finally, LifespanPolicy is used to identify if the POA's servants are persistent or transient. Persistent objects live beyond the life of the POA and server in which they were activated. Implementation of persistence is specific to an ORB's implementation and is not defined in the CORBA 2.2 specification. Policies other than the defaults can only be specified in the call to `POA::create_POA()`.

16.7.2 Run-time Performance

Invariably the tall pole in the tent is the transport layer with TCP-IP sockets usually representing the worst case. ORB vendors are sensitive to the needs of the real-time programmer and offer various optimizations. The most common optimization is for objects that are co-located: that is, objects that are within the same process space. An optimizing ORB will detect this condition and skip the marshaling/de-marshaling step and reduce the operation on co-located objects to little more than a function call. Some ORBs support the concept of pre-connect or bind. Although such optimizations are outside the standard they allow an ORB to locate objects ahead of time and create an optimal transport path between client and servant. Strict SCA-compliance would forbid the exercise of these optimizing calls in that it would make for a portability problem. Of course the best way to optimize CORBA run-time performance is to select a native transport mechanism and put GIOPs directly on the wire. This approach skips the whole TCP-IP stack which is universally supported. The result is a dedicated transport that would need to be replaced as the ORB was moved from system to system. The purchase of an ORB is then tied to a particular system. Some ORBs support the notion of a plugable transport. This allows the user to create his or her own custom transport underneath the ORB. Further discussion on optimization of run-time performance leads into the next topic which is discussion of concurrency models.

16.7.3 ORB Concurrency Models

As far as CORBA 2.2 and SCA minimum CORBA is concerned this section discusses optimizations that are ‘under the hood’. For ORBs circa 2.2 these features are likely to be available through compile or run-time switches. Subsequent versions of the CORBA specification actually provide PIDL for these optimizations and offer the user some degree of control. More recently these optimizations were further refined and ended up in the Real-Time CORBA specification. The concepts involved affect run-time behaviors and are a must for the SDR developer to understand.

The simplest ORB is a single threaded ORB where client and server live as part of the same thread. The client makes a call into the ORB which makes a call into the servant. Such an ORB, though fast, would not have much of a market because everything is required to be deployed within the same process. Furthermore client and server are locked into a strictly synchronous relationship, a construct not very useful for real world applications. Besides one CORBA transaction has to work its way completely through the system before the next transaction can take place. This single-threaded ORB concept can be easily extended to the case where client and server are deployed on different platforms. Marshaling, GIOP, and transport are now involved but essentially our ORB is still single threaded. An entire transaction is required to be completed before the next transaction can take place. This is referred to as a blocking ORB. Now even if multiple client threads attach to this blocking ORB each of them will have to wait their turn for access. Again for real-time, real-world applications this is not a useful ORB model.

Let’s move forward a few years in ORB design and take advantage of multiple threads. A common approach is use of the Thread-Per-Client model. These architectural models are called concurrency models [17]. In the Thread-Per-Client approach a new thread is created every time a new client makes a connection. This thread, called a receiver thread, either spins or does a blocking call to `select()`, which listens for calls from his client. When the client makes a call the receiver is there to pick it up off the wire – this is termed an upcall. The receiver thread invokes the appropriate POA/servant and remains blocked until the POA/servant completes the transaction by sending a response to the client. Figure 16.5 shows a transaction diagram of this entire process.

With just a few clients this is a good approach because each client gets their own receiver thread. The problem is the blocking that occurs while the POA/servant is performing its processing. If the client makes back-to-back calls the second call will be blocked. Although the client side could also be threaded, this is not helpful if the client’s processing requires a response before proceeding. Another different approach, called the Thread-Pool approach, offers a solution to the blocking problem. In Thread-Pool concurrency the receiver thread drops the request into a queue. The system has pre-spawned a bunch of threads attached to the queue. When a request gets dropped into the queue a thread will awaken and dispatch the request to the POA/servant. Just after dropping the request into the queue the receiver thread goes right back to listening for additional client requests. Figure 16.6 shows a typical Thread Pool transaction.

With the design given in Figure 16.6, the server could handle three back-to-back requests without blocking the client. These two particular concurrency models work well when run-time performance is critical but they don’t scale well to thousands of clients. It is likely that the operating system has some limit on the number of threads that can be spawned.

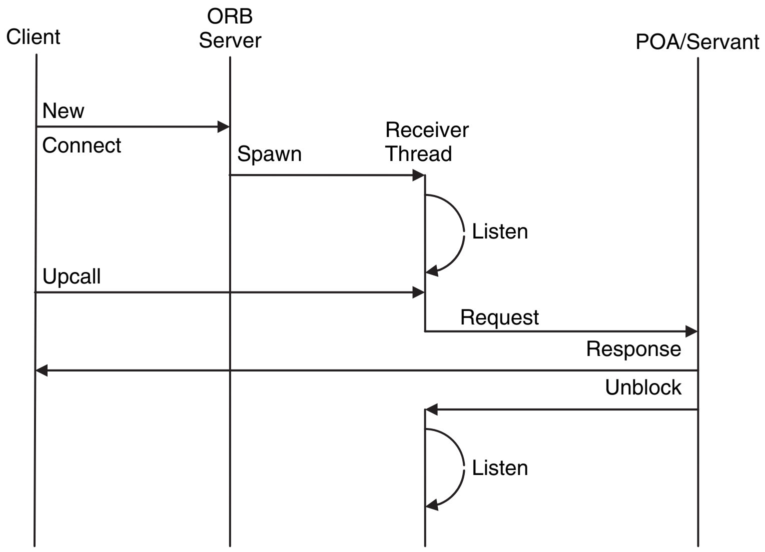


Figure 16.5. Thread-Per-Client transaction diagram

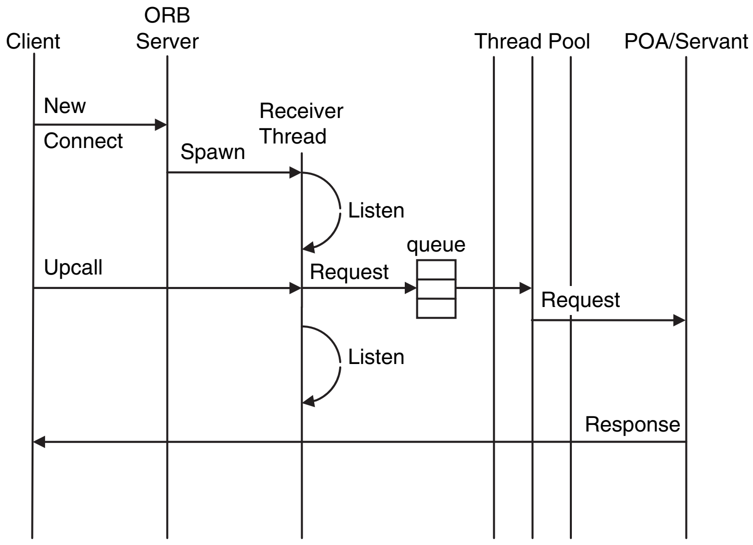


Figure 16.6. Thread Pool transaction diagram

There are other models that sacrifice run-time performance for scalability, including the reactive and leader-follower models. Instead of spawning threads for every client or even for every request these ORBs use a combination of thread pools and a limited number of receiver threads that will block clients as needed. Now consider the instance where the client does not expect a response. Why block at all?

16.7.4 One-ways, Two-ways, and Blocking

The IDL offers a means of indicating to the implementation that an object offers no return values. For an operation to be declared `oneway` all of the following must be true: 1) there can be no `out` or `inout` parameters; 2) the operation must return a `void`; 3) the operation cannot have `raises` exceptions defined. The CORBA specification offers the invocation semantics of a `oneway` call as 'best effort'. That is, there is no guarantee that the operation will be performed and the implementation offers no means of testing for the success of the operation. The beauty of the `oneway` is that it instructs the ORB to perform an operation and not wait around for a response. For most implementations this means that a `oneway` call will not block the client. The call is made and the request gets dumped into a queue and the client keeps chugging along.

Suppose the client comes back around through his processing loop and sends another request. Odds are his original request hasn't even left the processor yet. It's still sitting in an outgoing queue waiting for the ORB or client downcall thread to run. Now depending on the design of the ORB this second call will block despite the fact that it's a `oneway` call. Even though `oneways` don't support user-defined exceptions, it is still quite possible for a system exception to get thrown – consider an outgoing buffer that is full. In order for the client thread to survive, the user should check for a system exception on oneway calls. Without checking for a system exception the client process will most likely crash. The program counter will have moved and the traceback will contain bogus information.

16.8 Real-time CORBA

Real-time (RT) CORBA adds features to allow designers more control over the run-time behaviors of objects. We advocate designs that do not rely on the timeliness of ORB calls. Our view is that real-time CORBA offers the illusion of more precise control when in fact the native operating system is what's really doing the work. A portable design will be one in which CORBA objects are loosely coupled and have little or no real-time constraints. If one has objects that must be tightly coupled then use a non-CORBA inter-process mechanism to bind these objects. It is no less instructive to examine briefly what RT CORBA has to offer – if anything just to pick up on techniques that can be applied in our own designs without having the native ORB support.

There are a few goals that RT CORBA tries to address: 1) the means to do performance optimization; 2) a standardized approach to specifying and enforcing Quality of Service (QoS); 3) specification of a real-time programming paradigm to provide determinism over distributed applications; and 4) management of shared resources.

So a real-time ORB exists alongside a regular ORB. A pseudo-object reference to the real-time ORB is accessible with a call `to get_initial_references("RTORB")`. POAs created on this ORB come from a special module called `RTPortableServer`. Let's start by covering some of the more obvious extensions. RT CORBA has an attribute called Priority and a native element called PriorityMapping. A CORBA priority is then mappable to all the different kinds on priority schemes that are used in RT OSs today. This includes the sign of the monotonicity of the scheme. In some RT OSs a higher number means more privilege whereas in others a lower number implies more privilege. RT CORBA also specifies operations to transform back and forth from the native priority to CORBA priority.

Finally RT CORBA offers two different means of controlling how Priority is propagated through the distributed system. In the Client Propagated model a priority is passed on the wire as part of the request. If a contradicting policy is not in force the servant will run at the priority expressed by the client. The contradicting policy is the Server Declared policy. In this policy the server decides what priority a particular object will run at and will encode that information in the IOR.

RT CORBA provides a wrapper to the mutex functions. These operations are nearly identical to their POSIX counterparts. Of course, underneath the Mutex PIDL is, for many systems, POSIX mutexes. The Mutex interface is local and cannot be accessed remotely. Its function is, of course, to arbitrate access to a shared resource. RT CORBA mandates that these mutexes be supported by a priority inheritance mechanism. It does not mandate the type of mechanism but only that the system supports it. We saw in Chapter 14 on POSIX that priority inheritance is for the system that is not, or cannot be, subject to analysis that would preclude the priority inversion problem from ever occurring. RT CORBA mandates the priority inheritance solution despite the obvious performance hit that occurs for RT OSs that implement priority inheritance. There is really no way around this because RT CORBA is built to support generalized systems that might or might not ever be subject to static analysis. So RT CORBA mandates the problem to be solved dynamically.

RT CORBA provides several new Policies and controls on POAs. The user can specify thread pools on a POA that are set to run at a particular priority. Even better than that, a user can specify lanes of thread pools with each lane at a different priority. Finally, and perhaps, most importantly, RT CORBA makes provision for the implementation of prioritized private connections between client and server.

16.9 Overview of Available ORBs

We briefly attempt to summarize a few of the ORBs available as of early 2006. This list does not attempt to be exhaustive but rather hits a few of the more well known ORBs. The bottom line is that compared to just a few years ago when the SCA was being developed there are many more choices of ORBs available to apply to the software radio domain. The reader is encouraged to understand the differences between ORBs and requirements of the task at hand. If one's design is predicated upon having a state-of-the-art ORB then one might revisit functional allocations and find out what aspect of the design is driving this requirement. Do not impose requirements on the ORB that it is not suited to address.

16.9.1 TAO ORB

TAO is a freely available ORB produced by the Distributed Object Computing (DOC) group at the Washington University and Vanderbilt under Professor Doug Schmidt [18]. The only supported language mapping is C++. It has grown over the years to be quite extensive and detailed. TAO has been around since at least 1998. It provides version 3.x CORBA functionality as well as real-time 1.0 functionality. It has been used in a few JTRS SCA projects as well as dozens of other Defense-related programs over the years. Structurally it is built atop the ADAPTIVE Communication Engine (ACE), a C++ framework that performs common communication software tasks over the top of a variety of operating systems. ACE provides abstractions for tasks such as event dispatching, signal handling, interprocess communication, shared memory management, concurrent execution and synchronization. ACE-TAO has been built for an impressive number of systems including commercial embedded real-time systems. It has a large following, an extensive mailing list for developers, and at least half a dozen companies that offer commercial support packages. The ORB can be used in the non-real-time mode or real-time mode. This ORB can be built to support minimum CORBA or full CORBA complete with Dynamic Invocation Interface (DII), Dynamic Skeleton Interface (DSI), and the Dynamic Any Type and Value Types. TAO is supplemented by an extensive set of ORB services including Naming, Event, Notifications, and CosLifeCycle to name but a few.

16.9.2 ORBexpress

ORBexpress is produced by Objective Interface Systems (OIS) [19]. ORBexpress has language mappings to Ada, C++, and Java. This ORB is used by many vendors in the telecom, datacom, and defense industries. It is a small footprint, fast ORB, having won several highly publicized CORBA benchmark contests. ORBexpress offers plug-in transports that allow the consumer to specify an underlying media such as shared memory or PCI. This allows messages to go straight on the wire without the need for a TCP-IP stack. It is used on several SDR radio projects including Boeing's Ground Mobile Radio (GMR) formerly Cluster 1. It loosely provides 2.x CORBA as well as the 1.x Real-Time CORBA functionality. The over-arching principle of the OIS design is high-speed and small footprint, not specification compliance. If ACE/TAO is at one end of the scale in terms of the richness of CORBA specification features, ORBexpress is at the other end. It is available for many compilers and operating systems including embedded RTOSs. OIS has been very proactive in working with the US government on the implementation of Multiple Independent Levels of Security (MILS) and DO-178B (FAA) middleware.

16.9.3 ORBit2

This is the ORB used by the Gnome desktop [20]. It is CORBA 2.4 compliant and has C, C++, and Python mappings. The ORB is freely available in source form. It is built on top of glib2 – the gnome library. Glib provides the same kinds of platform independent operating system services as ACE. Because of its extensive use in the development of the Gnome desktop, it is a very stable implementation. It is actively maintained by RedHat and Ximian – now part of Novell. ORBit comes with its own Naming Service but does not support the Event or Lightweight Logging Service. Normally these services are provided by an Operating Environment vendor. This ORB is not available for embedded real-time systems nor does it support custom extensible transports. ORBit provides a POA interface and DII/DSI for the server side mapping in both language implementations.

16.9.4 MICO

MICO is an open source CORBA 2.3 implementation [21]. It offers only a C++ mapping yet is fully featured with DII, DSI, a graphical access tool to the Interface Repository, interceptors, Type Codes, and Naming and Event services. MICO was built on the principle that it is not dependent on any other middleware products or libraries. MICO sticks to the basic requirements of CORBA compliance and does not offer optional features such as CORBAservices. MICO has been successfully built and executed on a variety of UNIX-style and Windows/Cygwin platforms. It also has a company that offers commercial support.

16.9.5 OMNI

This ORB was developed by Oracle Olivetti laboratories in Cambridge, UK. It is an open source freely available ORB that has been tested as CORBA 2.1 compliant by the Open Group [22]. Since that time it has been continually upgraded and currently implements most of CORBA 2.6. Its design was unique and for a long time was the fastest C++ ORB implementation available. In addition the ORB has mappings that support the Python language. This ORB is used by a few JTRS SCA radio developers owing to its maturity and stability. It has profiles for both minimum and Full Servant managed enterprise CORBA. OmniORB supplies Naming and Event services. Several practitioners have implemented their own proprietary custom extensible transports under this ORB. The ORB supports Secure Socket Layer (SSL) as an additional transport out of the box. The ORB is built to the CORBA 2.4 specification. OmniORB is no longer in active development having passed through various labs until getting shut down by AT&T.

There are many, many other ORBs and just because they are not listed in this book does not diminish their capability or capacities to perform a particular work. Similar to operating systems, there are a couple of guiding principles when it comes to ORB selection. However for an SCA-compliant system there is a reduced number of degrees of freedom. This is because an SCA-compliant Operating Environment has an operating system, ORB, and Core Framework that come tightly bundled. Operating Environments are meant to enable waveform portability but themselves are not portable. Commercial ORBs are typically licensed to a particular operating system/host platform pair. This economic consideration couples an ORB to an operating system and target platform. Commercial ORBs are likely to be build for high speed and small footprint and hence are customized to a particular operating system and target platform. Furthermore, commercial ORB vendors typically extend the CORBA specification – which is allowed, even encouraged, by the OMG. The problem is that it is never really made clear to the user what is standard and what is extension. The applications programmer following examples in the documentation is quite likely to end up with code that would never port to another ORB. The SCA-compliant applications must restrict themselves to the interfaces of minimum CORBA and the SCA-defined services. An unfortunate side effect of this requirement is that being forced to use standard interfaces as opposed to the faster, optimized proprietary extensions might lead to run-time performance issues. Incidentally the same is true of the requirement to use only POSIX interfaces. Commercial RTOSs typically add POSIX on top of their own much faster interfaces. This is bad for two reasons: one, of course, is the performance hit, but the other is the fact that the POSIX behavior is, at best, emulated. Such faked out POSIX interfaces are unlikely to behave the same from RTOS to RTOS. It is likely that application for SCA waivers in the interest of performance will become the rule rather that the exception.

The free ORBs – available as source code – can be compiled for almost any target. This is opposed to a commercial ORB that is tied to one target. But the reader should not be fooled into thinking that a free ORB is really free. Even the process of building an ORB can be painful depending on the target. Customizing someone else's source code in order to eliminate unnecessary features or perhaps to optimize performance is also very expensive in terms of development and test labor. This does not take into account the effect of introducing instabilities into the runtime. Middleware bugs affect all the application development teams and can be particularly hard to find – this author not so fondly remembers single-stepping through someone else's ORB software. There have been many CORBA success stories but there have also been many cases of an incorrect choice in middleware undermining a program, leading to its untimely demise.

17

The Services

The SCA mandates three CORBA-based services, all of which are OMG standards. The first is the Interoperable Naming Service. The Naming Service allows users to locate objects within a distributed system by human readable names. The SCA interface *Application* contains a list of Naming Contexts used by its components.

The next mandatory service is the Event Service [23]. This service is not only available to the applications programmer but is also required for use within the Core Framework on the Domain Manager object. There are a total of three event types required by the SCA: ObjectAdded, ObjectRemoved, and StateChange. This is not to be confused with the OMG's TypedEvent specification. The SCA requires only the traditional CosEventComm Push model using CORBA's *any* type.

A new addition to the OMG family of services came about directly as a result of the efforts of the international defense community in finalizing SCA version 2.2. This service is the Lightweight Log Service. Request for Comment for the Lightweight Log Service was published in June 2002 [24]. Senior OMG staff was exposed to the SCA Log Service at the SDR Forum meeting in Edinburgh, Scotland, in September 2002. Intrigued by the usefulness of the SCA Log Service, its level of maturity, and the buy-in of the Software Defined Radio community, the SCA Log Service was published by the OMG as the Lightweight Log Service Specification [25], in November 2003.

17.1 Interoperable Naming Service

Although a little dated, Reference [26] is a must have for the CORBA programmer working in C++. It also provides a pretty thorough discussion of naming graphs. Our approach to understanding the Naming Service will be to connect with an actual Name Server running as part of our SCARI Core Framework. This is particularly illustrative because it emphasizes the language independence of CORBA. The SCARI name server is running as part of the Java run-time – see the UNIX man page for `tnamserver`. Our 'C' program will use a stringified IOR to locate the SCARI name server in order to make invocations on its interfaces.

The Naming Service is used to construct a directory-like structure called a naming graph. The naming graph allows servers to publish their object references and 'bind' them to easy-to-remember names. Similar to a UNIX or DOS file system the naming graph has a hierarchical architecture. A directory within a file system is likened to a Naming Context within the naming graph. A filename within a file system is likened to a node or leaf within a Naming Context. Each non-empty node within the naming graph has a 'name' and 'kind'. As seen in Chapter 6, the SCA requires 'kind' to always be set to the null string. An object or node is uniquely named in each context just as a file must be uniquely named within a sub-directory. With just a node 'name' the applications programmer can use the Naming Service to 'resolve' the object reference of an item of interest.

Figure 17.1 shows the structure of the Naming Service module. Methods exist for creating contexts and adding Name/Object reference pairs to the tree using the *bind* interface. For a given name an object reference can be recovered using the *resolve* or *resolve_str* interface.

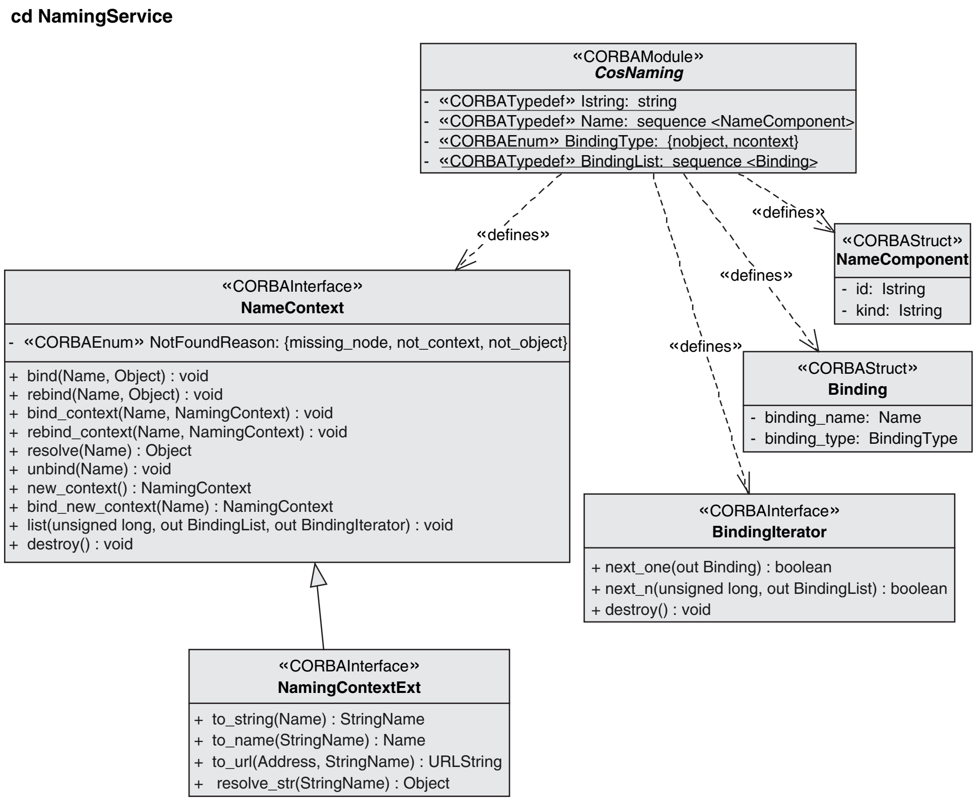

cd NamingService
«CORBAModule»
CosNaming
- «CORBATypedef» Istring: string
- «CORBATypedef» Name: sequence <NameComponent>
- «CORBAEnum» BindingType: {nobject, ncontext}
- «CORBATypedef» BindingList: sequence <Binding>
«defines»
«defines»
«defines»
«defines»
«CORBAInterface»
NameContext
- «CORBAEnum» NotFoundReason: {missing_node, not_context, not_object}
+ bind(Name, Object) : void
+ rebind(Name, Object) : void
+ bind_context(Name, NamingContext) : void
+ rebind_context(Name, NamingContext) : void
+ resolve(Name) : Object
+ unbind(Name) : void
+ new_context() : NamingContext
+ bind_new_context(Name) : NamingContext
+ list(unsigned long, out BindingList, out BindingIterator) : void
+ destroy() : void
«CORBAStruct»
NameComponent
- id: Istring
- kind: Istring
«CORBAStruct»
Binding
- binding_name: Name
- binding_type: BindingType
«CORBAInterface»
BindingIterator
+ next_one(out Binding) : boolean
+ next_n(unsigned long, out BindingList) : boolean
+ destroy() : void
«CORBAInterface»
NamingContextExt
+ to_string(Name) : StringName
+ to_name(StringName) : Name
+ to_url(Address, StringName) : URLString
+ resolve_str(StringName) : Object

Figure 17.1. Naming Service component diagram

Our code example in this chapter is a simple spider program that will walk the name graph and print out a hierarchical view. We will actually start an instance of the SCARI Core Framework and launch some *Devices*. The SCA requires certain elements to be registered with the Naming Service. Our code example will locate the Java-based name server and retrieve the contents of the naming graph. By following the directions enclosed with the SCARI Core Framework, start the Naming Service by issuing the following command:

```
./startNamingService
```

The output from the Java name service includes its stringified IOR – that is 'IOR:' followed by many lines of hexadecimal digits. Though stringified IORs are universally understood by ORBs, they are also allowed to contain data fields that are ORB-specific. Thus stringified IORs are of different lengths from ORB to ORB. We will need this long IOR string when we start our `browseTree` program. The user can cut and paste it between terminal windows or the output can be re-directed to a file. If re-directed to a file, the user must clean up the file contents so that only the IOR string is left. The file containing the isolated IOR can then be re-directed to the standard input when we start the `browseTree` program. Next start the Domain Manger, Device Manger, Log Service, and the Devices by issuing the following command:

```
./DemoPlatformNode1Bootup
```

Use the makefile in Figure 17.2 to build our sample program. This makefile will link in the libraries needed for the ORBit orb and Naming Service. Since ORBit has become part of the GNOME project, those include files – glib 2.0 – are also needed. They are dynamically loaded at run-time. Because ORBit is used by the GNOME project it has got a lot of mileage on it and is fully featured and stable. This is critical to the overall success of our software radio in that SCA depends on CORBA and a stable ORB is a prerequisite.

```
SOURCES = browseTree.c
OBJECTS = $(SOURCES:.c=.o)
CFLAGS += -x c -Wall -g
CC = gcc
INCL = -I /usr/include/orbit-2.0 \
       -I /usr/include/glib-2.0 \
       -I /usr/lib/glib-2.0/include

LIBS = -lORBit-2 -lORBitCosNaming-2

bT: $(OBJECTS)
      $(CC) -o $@ ${OBJECTS} $(LIBS)

#Compile
%.o: %.c
      $(CC) ${CFLAGS} $(INCL) -c $<
```

Figure 17.2. Makefile for use with ORBit2

Our program – `bT` – is now ready to run. Don't forget to cut and paste the naming service IOR unto the command line. Or if saved in a file simply re-direct the file to the standard input. Given a successful run, the output of Figure 17.3 should be produced.

```
context INITIAL
   context SCARI_DM
      object[1] DomainManager
   object[1] node1Logger1
   object[2] node1DeviceManager_DCE:18dd6458-494e-43cd-b823-07778bb9ca51
   object[3] node1RFDeviceImpl_DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888
   object[4] node1AudioDevice_DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB

total number of objects 5
```

Figure 17.3. SCARI Node1 via Name Service

We see that five objects from the SCARI demo are running: notably the Domain Manager, a logger, a device manager, and RF and Audio devices. The Name Service name is the concatenation of two strings taken from the Node1 Device Component Descriptor (DCD) file. The SCA specifies a naming convention for Application components but not Devices. For a *Device* within the SCARI Core Framework the name is formed by the *usagename* sub-element of *componentinstantiation* followed by an underscore followed by the *Id* attribute of *componentinstantiation*. The UUIDs in this case are attached to the component instance. There are also UUIDs that uniquely identify component implementations.

These component instantiation UUIDs are also used to identify these components in case they have ports that need to be connected. There is no application running at this time. The user can install, instantiate, and start an application on the domain with the application manager GUI:

```
./startApplicationManager
```

By starting up the Audio Effect application additional objects are launched in the Domain Manager's context. Run the browseTree application again to get the result shown in Figure 17.4.

```
context INITIAL
   context SCARI_DM
      object[1] DomainManager
       context AudioApp1
          object[1] AudioEffectController_DCE:7F19A71E-DE41-4BEF-A619-
            9EE5ECCD832C
          object[2] EchoResource_DCE:916B1F9F-25BA-43A6-896E-5B078D12B727
          object[3]ChorusResource_DCE:8FC4B3CF-5203-4874-98D6-0FF6E4F2ED5C
   object[1] node1Logger1
   object[2] node1DeviceManager_DCE:18dd6458-494e-43cd-b823-07778bb9ca51
   object[3] node1RFDeviceImpl_DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888
   object[4] node1AudioDevice_DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB

total number of objects 8
```

Figure 17.4. SCARI Audio Effect Application via Name Service

Within the `SCARI_DM` context an additional context is created – AudioApp1. This is actually the name provided by the user when the application was started. Within the AudioApp1 context there are three components: AudioEffect controller, and the Echo and

Chorus resources. As required in the SCA the string name of the *Application* components are registered with the Naming Service along with their UUIDs. These UUIDs are found in the AudioEffectApplication Software Assembly Descriptor (SAD) file. The name is formed by the *name* attribute of the *namingservice* sub-element of *componentinstantiation* followed by an underscore followed by a unique name. The SCA says only that the unique name is provided by the implementation. In the SCARI Core Framework, an application component unique name is taken from the *Id* attribute of the *componentinstantiation* element.

Now let's take a look at the software itself. We introduce two functions – Figures 17.5 and 17.6 – that are used to start the orb and then to connect to the Naming Service.

```
1. /* browseTree.c */
2.
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <string.h>
6. #include <time.h>
7.
8. #include <orbit/orbit.h>
9. #include <ORBitservices/CosNaming.h>
10.
11. void startOrb(char** local_argv,
12.         CORBA_ORB *orb, CORBA_Environment *ev)
13. {
14.    int local_argc=1;
15.
16.   CORBA_exception_init(ev);
17.   *orb=CORBA_ORB_init(&local_argc,local_argv,"orbit-local-orb", ev);
18.   if (ev->_major != CORBA_NO_EXCEPTION) {
19.     fprintf(stderr,"CORBA_ORB_init (Exception %d)\n", ev->_major);
20.     fprintf(stderr,"  Id %s\n",CORBA_exception_id(ev) );
21.         CORBA_exception_free(ev);
22.       abort();
23.    }
24.    printf("ORB initialized\n");
25. }
26.
```

Figure 17.5. ORB client-side initialization

Lines 16 and 17 are the heart of the function. Line 16 initializes an exception handler and line 17 starts the orb itself. The exception handler initialized in this case is a handler defined by the caller. Exception handlers can be defined and initialized in any context. To avoid a memory leak they should also be released when they have outlived their usefulness. Lines 18 through 23 check for and print out an error message in the event that the orb throws an exception. For brevity our subsequent code example will not test for exceptions but the reader is encouraged to follow good programming practice and always test for exceptions after every CORBA call. It's not hard to define macros that can describe the exception handler: They can be turned on during the development and integration phases and then turned off for production.

Here's the function to connect to the Naming Service.

```
27. void getNamingService(char** local_argv,
28.          CosNaming_NamingContext* myNC,
29.          CORBA_ORB *orb, CORBA_Environment *ev)
30. {
31.   *myNC = CORBA_ORB_string_to_object( *orb,local_argv[1], ev);
32.   if (CORBA_Object_is_nil(*myNC,ev)) {
33.      fprintf(stderr,"Cannot find Naming Service, Nil returned!\n");
34.      abort();
35.   }
36.   printf("Resolved NameService\n");
37. }
```

Figure 17.6. CORBA Name Service via stringified IOR

Line 31 takes the stringified object reference of the Name Service as passed from the command line and returns the object reference of the initial naming context. This is the starting point of our naming graph. We call the naming context 'INITIAL' though technically it doesn't have a name. The code segment in Figure 17.7 is the main program.

```
38. typedef struct {
39.    int objCount;      /* running count of object references */
40.    int nesting;       /* keeps track of nested Name Contexts */
41. } nestedTreeStruct;
42.
43.
44. int main( int argc, char * argv[] )
45. {
46.    CORBA_Environment ev;
47.    CORBA_ORB orb;
48.    CosNaming_NamingContext myNC;
49.    nestedTreeStruct myNT;
50.    char* dummy_argv[2];
51.
52.    dummy_argv[0] = argv[0];
53.    dummy_argv[1] = 0;
54.
55.    myNT.nesting = myNT.objCount = 0;
56.
57.    startOrb(dummy_argv,&orb,&ev);
```

Figure 17.7. Name Service browser – main

```
58.    if (argc > 1)
59.      getNamingService(argv, /* IOR from the command line */
60.           &myNC, &orb, &ev);
61.    else {
62.       fprintf(stderr,"Usage: ./bT nameServiceIOR\n");
63.       abort();
64.    }
65.
66.    /* Browse Naming Tree */
67.    printf("\ncontext INITIAL\n");
68.    myNT.nesting += 1;
69.    traverseNamingContext(&myNC,&myNT,&ev);
70.    printf("\ntotal number of objects %d\n",myNT.objCount);
71.
72.    CORBA_Object_release(myNC, &ev);
73.    return 0;
74. }
```

Figure 17.7. (Continued)

After initializing the ORB and retrieving the initial naming context, the heart of the main program is found in line 69. This call to *traverseNamingContext* takes a pointer to a context and a pointer to a data structure that is used to keep track of the nesting depth and a running count of objects encountered. This data structure is found in lines 38–41. The traverse function can be used to traverse any context and hence it can be called recursively.

As the main program is cleaning up and in order to exit one might be tempted to call the destroy method on the NamingContext object – `CosNaming_NamingContext_destroy`. Unfortunately this not only gets rid of the client-side of the naming context but also destroys the server-side. The heart of the main program – `traverseNamingContext` – is shown in Figure 17.8.

```
1. void traverseNamingContext( CosNaming_NamingContext* myNC,
2.                nestedTreeStruct* myNT, CORBA_Environment* ev)
3. {
4.    CosNaming_NamingContext newContext;
5.    CosNaming_BindingList* seqBinding;
6.    CosNaming_BindingIterator myIter=CORBA_OBJECT_NIL;
7.    CosNaming_Binding myBinding, *CNB_p;
8.    CosNaming_Name myBindingName;
9.    CORBA_string myString;
10.    int j,localCount=0;
11.    CORBA_boolean retVal;
12.
13.    CosNaming_NamingContext_list(*myNC,
```

Figure 17.8. Recursive context function with iterators

```
                0,
                &seqBinding,
                &myIter,
                ev);
if (CORBA_Object_is_nil(myIter,ev)) {
  return;
}

/* Descend into other contexts */
retVal = CosNaming_BindingIterator_next_one(myIter,&CNB_p,ev);

while (retVal==CORBA_TRUE)
{

   myBinding = *CNB_p;
   for (j=0; j< myNT->nesting; j++) printf(" ");
   if (myBinding.binding_type==CosNaming_ncontext)
  {
   printf("context ");
   myBindingName = myBinding.binding_name;

   /* print out name */
   myString = CosNaming_NamingContextExt_to_string(*myNC,
               (const CosNaming_Name *)&myBindingName,ev);
   printf("%s\n",myString);
   CORBA_free(myString);

   myNT->nesting += 1;
   newContext =
     CosNaming_NamingContext_resolve(*myNC,(const
        CosNaming_Name*)&myBindingName,ev);

   traverseNamingContext(&newContext,myNT,ev);
   CORBA_Object_release(newContext, ev);
  }
  else
  {
    /* printout leaf */
    myNT->objCount += 1;
    localCount += 1;
    printf("object[%ld] ",localCount);
    myBindingName = myBinding.binding_name;

    /* print out name */
    myString = CosNaming_NamingContextExt_to_string(*myNC,
               (const CosNaming_Name *)&myBindingName,ev);
```

Figure 17.8. (Continued)

```
59.        printf("%s\n",myString);
60.        CORBA_free(myString);
61.       }
62.
63.       retVal = CosNaming_BindingIterator_next_one(myIter,&CNB_p,ev);
64.
65.      } /* end while */
66.
67.      /* have iterated all objects in this context */
68.      /* back out one level */
69.      myNT->nesting -= 1;
70.      CosNaming_BindingIterator_destroy(myIter,ev);
71.
72. }
```

Figure 17.8. (Continued)

The first operation performed on a context is to retrieve a list of its elements. The call to `CosNaming_NamingContext_list` in line 13 instructs the context to return up to 0 elements because what we're interested in right now is the `BindingIterator` not a list of bindings. The iterator will let us walk the Name Context without knowing how many entries there are. If you did specify the number of entries to return, how would you know how many to ask for? At best it would just be a guess. In reality one does not know how many objects are in a context – there might be tens of thousands of nodes.

The call in line 13 asks the context to return zero binding elements so `seqBinding` is empty. Line 18 checks `BindingIterator` to see if is nil. This would indicate an empty context. Methods exist on the `BindingIterator` object that behave somewhat like a linked list. Instead of requesting a certain number of bindings, zero bindings are requested and a 'pointer' to the first binding is returned. It is then possible to walk through the naming graph one element at a time. Line 23 operates on the BindingIterator – `CosNaming_BindingIterator_next_one`. This call returns a pointer to a binding. Subsequent calls will return the next binding and so on until there are no bindings left to return. After there are no more bindings left within the context, the call to `CosNaming_BindingIterator_next_one` will return `CORBA_FALSE`.

Line 25 starts a code block that will iterate the context until there are no more bindings left. The pointer to the next binding is de-referenced in line 28. The binding is tested to determine whether it is another context or an object in line 30. In lines 33 and 54 the `CosNaming_Name` is extracted from the binding. Finally the string name is extracted – lines 36 and 57 – with a call to `CosNaming_NamingContextExt_to_string`. This is somewhat simpler to use than trying to access the string `_buffer` directly. This technique is universal because it can traverse a name tree of arbitrary complexity.

The calls `CosNaming_BindingIterator_next_one` and `CosNaming_NamingContextExt_to_string` allocate memory as needed in order to return bindings and strings. To avoid a memory leak the user is required to free up the allocated memory. The strings are freed in lines 39 and 60. The `BindingIterator` is destroyed in line 70.

When a binding is returned, line 30 checks to see if it is a context. If the name of the context is printed out, the indentation level on the printf is increased and a recursive call to traverseNamingContext is made. If the binding is just an object the name – a simple CORBA string – is printed out in line 59.

In order to get the nested contexts to print out with proper indentation, the variable `myNT->nesting` is used to keep track of how deeply descended is the recursion. Line 53 prints out three spaces of white space for each level of depth in the naming graph. Before entering a new context the nesting count is incremented; (line 41). After all the bindings within a context have been iterated and it's time to back out to the calling level, the nesting is decremented (line 69). Whatever method is used to traverse the naming graph, list or iterator, the purpose is invariably to gain access to an object. Once a name is isolated the user can retrieve the object's reference by a call to `CosNaming_NamingContext_resolve` using the object's `CosNaming_Name` or by a call to `CosNaming_NamingContextExt_resolve_str` using the object's `CORBA_string` name within the correct context. In later examples we will use the Name Service to gain access to Core Framework, *Device*, and *Application* components.

The Naming Service Extension supports the ability to specify a name in a couple of different formats as well as the ability to transform between the formats. If one considers the Naming Service `_resolve` operation as just another transformation, then the Naming Service can be considered a universal translator. Figure 17.9 shows the various formats and the actions used to transform between them.

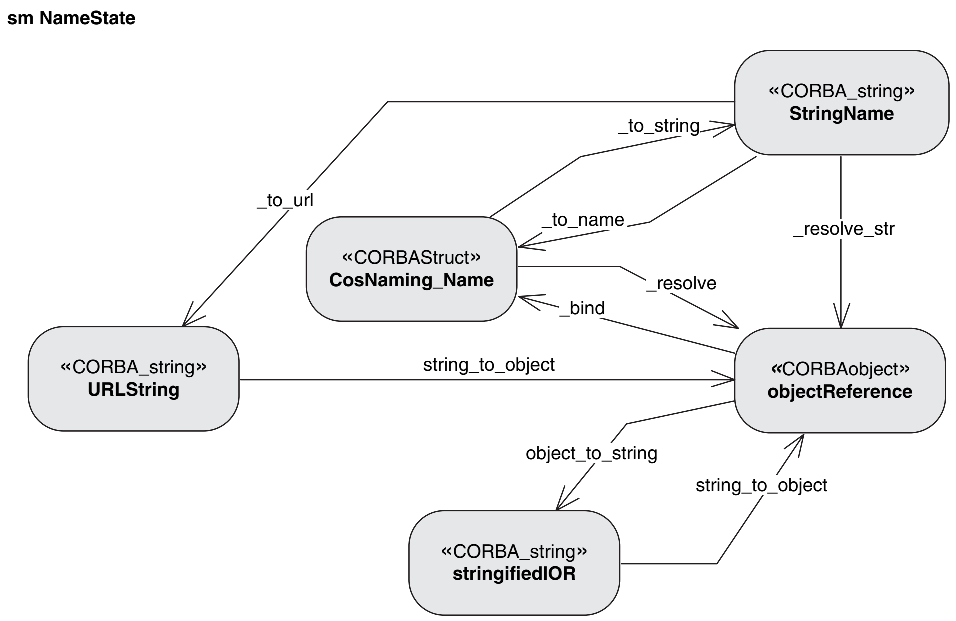


Figure 17.9. Naming Service – Name State Diagram

17.1.1 Universal Unique Identifiers

The SCA allows certain instances and interfaces identified within the Domain Profile to be tagged with a Universal Unique Identifier (UUID) [27]. Note in SCA specifications section 3.1.3.5.1, this usage is optional. Furthermore, SCA section 7.4 alludes to the creation of a Registration Body to be established for a yet to be defined purpose. To date no such body has been established. In the world at large UUIDs were invented to support the Distributed Computing Environment's Remote Procedure Calls (DCE RPC). The concept of DCE is exactly the same as CORBA and UUIDs can be likened to IORs. DCE supports client/server relationships over global networks. The principal difference between DCE and CORBA is that DCE is rooted in the procedural world of 'C' whereas CORBA extends the object-oriented paradigm to a variety of languages. Generating a UUID is easy. The web link http://www.itu.int/ITU-T/asn1/uuid.html can provide a UUID.

A 'C' programmer can also generate and manipulate using the very common UUID library as accessed through the header file 'uuid.h'. This makes available the following function calls:

1. `uuid_clear()`
2. `uuid_compare()`
3. `uuid_copy()`
4. `uuid_generate()`
5. `uuid_is_null()`
6. `uuid_parse()`
7. `uuid_time()`
8. `uuid_unparse()`

A user can also generate a UUID from the command line on most UNIX machines with `uuidgen.` The formula for generating UUIDs is quite exacting and involves a mixture of time tags and a hardware-unique tag like an Ethernet MAC address. At 128-bits the UUID is guaranteed unique over the space of all UUIDs.

Whether a Domain Profile uses UUIDs or not is inconsequential to the fact that the SCA requires certain artifacts to be uniquely identified:

1. all instances of a software package as identified in the component's Software Package Descriptor (SPD);
2. every resource as specified in the `createResource` operation;
3. all ports used in the getPort operation as specified in a component's Software Component Descriptor (SCD);
4. all testIds as specified in a *Resource*'s Property File;
5. all *Application*s as specified in an *ApplicationFactory*'s Software Assembly Descriptor (SAD);
6. *Service*s as specified in the call to *registerService*;
7. all *PushConsumers* that register with an event channel.

In most cases the Core Framework will generate an exception if artifacts being created or registered are not uniquely named.

17.1.2 Core Framework Usage of the Naming Service

First and most important, the Domain Manager is expected to make a unique Name Context under the initial context. This then supports the notion of having multiple domains using the same Name Service. After creating the unique Name Context, the Domain Manager shall register (bind) itself under this uniquely named context with a name binding equal to 'DomainManager'.

The next primary user of the Name Service is an Application Factory. When executing the create() operation, the Core Framework forms a string that will be passed to the application as an execute parameter. According to SCA, this string is of the form '`ComponentName_UniqueIdentifier`'. Where the component name comes from the application's SAD file – specifically, the *componentinstantiation findcomponent namingservice* element, *name* attribute – the unique identifier is said to be implementation dependent. In SCARI, this unique identifier is the object instantiation's UUID. The Application Factory extracts the two strings from the SAD file and concatenates them with an intervening underscore.

There is another piece of information passed as an execute parameter to the application component. This is the stringified object reference of the application's Name Context. The application factory will create this context under the Domain Name but the SCA does not mandate separate contexts for each application. The SCARI Core Framework however uses the name of the application given in the run-time as a sub-heading under the Domain Name. This approach would allow multiple instances of the same application to live in the same domain namespace. This is the name provided as a parameter to the create() operation. When the application factory forms the execute command it will provide as parameters the string '`NAMING_CONTEXT_IOR`' followed by the stringified object reference of the newly created context. It will also pass the string '`NAME_BINDING`' followed by the '`ComponentName_UniqueIdentifier`'.

After starting the component, the application factory needs to check with the Name Service in order to retrieve the components IOR. Since the component might take some time coming into existence, the application factory might have to loop a few times before the application component completes registration with the Name Service. Once the application factory has retrieved the object reference of the component it can complete the create() processing by running getPort() operations on the newly created component.

Device Managers, Devices and Services are not required to register with the Name Service.

17.1.3 Application Usage of the Naming Service

When an application component begins execution it must extract the name context IOR and name binding from its *argv* or parameter list. The application component then creates all of its *Port* objects. The application component will need to keep track of the object references for each of the Port objects it creates. The application component then creates an object reference for itself. Finally the application component binds its object reference with the Naming Service and enters an ORB_run loop waiting for clients to invoke methods.

Similar to application components, *ResourceFactory* objects are also required to register with the Name Service.

17.2 Event Service

The Event Service is somewhat of a new addition to the SCA, making its first appearance in the 2.2 version of the specification. It is based on the OMG Event Service [23]. The Event

Service functionality considered mandatory by the SCA is that of the 'push' model. The SCA does not require the presence of the Event Service 'pull' interfaces. Figure 17.10 shows the Event Service class diagram as required by the SCA. At first glance the *CosEventComm* and *CosEventChannelAdmin* modules appear daunting, but there is a symmetry to the interfaces and the Core Framework does half of the work for you via the *registerWithEventChannel* interface on the Domain Manager object.

The Event Service provides a means of implementing asynchronous communications between suppliers and consumers. As previously mentioned the SCA requires implementation only of the 'push' interfaces; so suppliers push event data to consumers. The connection between suppliers and consumers is provided by an object called an *EventChannel*. Event data from the perspective of the OMG is of CORBA type *any*. The applications programmer is free to use this service for his or her own purposes as it's guaranteed to be present on every SCA-compliant system.

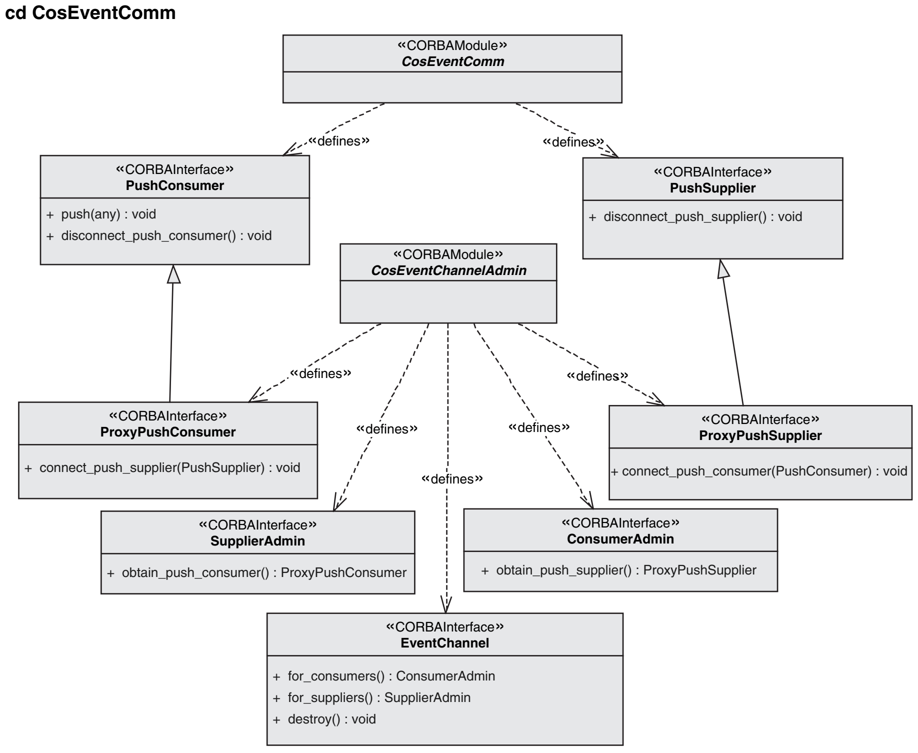

cd CosEventComm
«CORBAModule» CosEventComm
«defines»
«CORBAInterface» PushConsumer
+ push(any) : void
+ disconnect_push_consumer() : void
«CORBAInterface» PushSupplier
+ disconnect_push_supplier() : void
«CORBAModule» CosEventChannelAdmin
«CORBAInterface» ProxyPushConsumer
+ connect_push_supplier(PushSupplier) : void
«CORBAInterface» ProxyPushSupplier
+ connect_push_consumer(PushConsumer) : void
«CORBAInterface» SupplierAdmin
+ obtain_push_consumer() : ProxyPushConsumer
«CORBAInterface» ConsumerAdmin
+ obtain_push_supplier() : ProxyPushSupplier
«CORBAInterface» EventChannel
+ for_consumers() : ConsumerAdmin
+ for_suppliers() : SupplierAdmin
+ destroy() : void

Figure 17.10. Event Service class diagram

The EventChannel object supports the ability to have multiple producers sending event data to multiple consumers. This is done through the use of proxies. Figure 17.11 shows the relationship between event producers, the event channel, and event consumers. The exact mapping relationship between suppliers and consumers is implementation-dependent but typically every event supplied is sent to every consumer. The process for establishing a PushConsumer object involves a few steps but is pretty straightforward.

1. Implement a PushConsumer object with a *push*() interface – this is a server object so it is launched and permanently pends waiting for someone to invoke the *push*() interface.
2. Invoke the `for_consumers()` interface on an EventChannel object – this returns an object of type *ConsumerAdmin*.
3. Invoke the `obtain_push_supplier()` interface of the ConsumerAdmin object – this returns an object of type *ProxyPushSupplier*.
4. Invoke the `connect_push_consumer()` interface on the *ProxyPushSupplier* object – this interface requires the object reference of the *PushConsumer* object created in Step 1.

Now as events are produced they magically appear at the *push*() implementation. The applications programmer is free to do whatever is pleasing with the information contained in the *any* type passed to it.

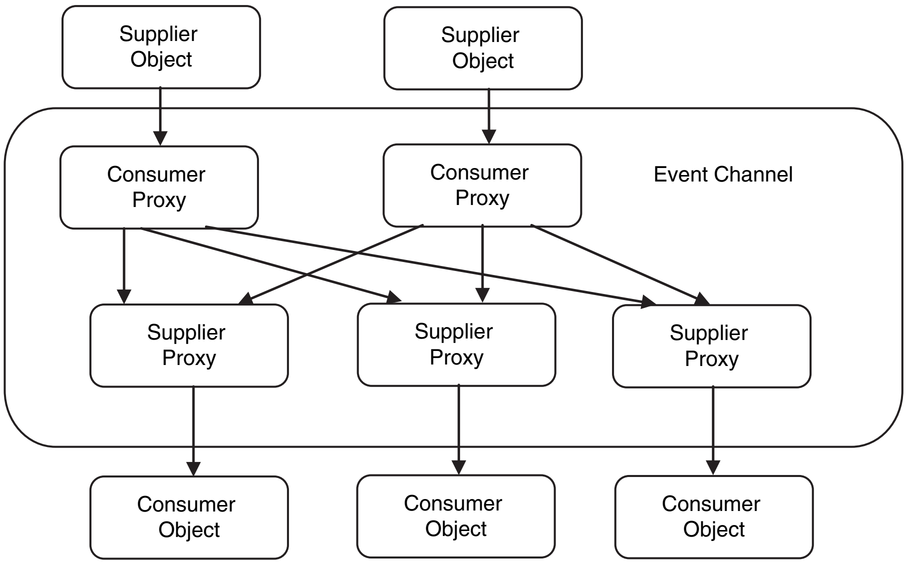


Figure 17.11. Event Channel data flow

As discussed in Section 3.3, the Core Framework is required to implement two event channels: the Incoming Domain Management (IDM) channel and the Outgoing Domain Management (ODM) channel. From the SCA perspective there are three *any* types defined: the *StateChangeEventType*, the *DomainManagementObjectAddedEventType*, and the *DomainManagementObjectRemovedEventType*. The IDM is used by *Device*s to register changes in state by sending *StateChangeEventType*s to whatever PushConsumers are connected to the IDM. The ODM event channel is used under the following circumstances:

1. by an *ApplicationFactory* upon successful creation/removal of an *Application*;
2. by the *DomainManager* whenever a Device, DeviceManager, or Service is registered or unregistered;

3. by the *DomainManager* whenever a SAD is installed/uninstalled;
4. by the *DomainManager* whenever an pushConsumer is registered/unregistered.

In order to establish a *PushConsumer*, there were four steps previously identified:

1. instantiate a PushConsumer object;
2. obtain a ConsumerAdmin object;
3. obtain a ProxyPushSupplier;
4. register the PushConsumer with the ProxyPushSupplier.

The *DomainManager* has a interface called *registerWithEventChannel* that takes care of steps 2 through 4. This takes some of the pain out of the PushConsumer establishment process. If the applications programmer wishes to connect to the IDM or ODM it is necessary only to create the *PushConsumer* object (Step 1 above) and then register it with the *DomainManager*. The *DomainManager* will then take care of everything else.

The following example shows how to connect an SCA-specific PushConsumer to the event channels. Initially, it is necessary to obtain the Event Service IDL located at http://www.omg.org/cgi-bin/doc?formal/01-03-02.

It is optional to get rid of the parts of the Event Service not required by the SCA. For our purposes it is clearer to focus only on the mandatory push-model interface. Furthermore, it is not necessary to have access to the Admin interfaces required to administer an event channel or be able to create an event channel because the *DomainManager* takes care of this. However, SCA specification section 3.1.2.2.3.1 clearly states that an OE should support the ability to create an event channel. This support for event channels could be done for the private benefit of the OE or it could be considered available to all users. The SCARI Core Framework uses a Java implementation of the Event Service from the OMG but does not make it explicitly available to the applications programmer. There are plenty of Event Service implementations around so it is not necessary to go into detail on how to create or administer event channels. The motivated reader can download Event Service implementations as part of the 'omni' [22] or 'MICO' orbs [21].

As users of the Core Framework we will let the DomainManager do all the work and restrict our attentions to the three Standard Event Types mandated in the SCA. Even though ORBit2 does not come with an event service we are required only to implement a `_push` method on a CosEventComm PushConsumer object. The IDL for our PushConsumer is given in Figure 17.12.

```
module CosEventComm      // SCA --> just the Push interfaces
{
   exception Disconnected{};

   interface PushConsumer
   {
      void push (in any data) raises(Disconnected);
   };
};
```

Figure 17.12. PushConsumer IDL

It is not even necessary to implement the disconnect method because the *DomainManager* provides an *unregisterFromEventChannel* method. We can then generate the necessary common, skeleton, and implementation files from the IDL compiler as follows:

```
orbit-idl-2 --skeleton-impl scaPushConsumer.idl
```

In the generated file scaPushConsumer_impl.c, this command will generate a function placeholder as follows:

```
static void impl_CosEventComm_PushConsumer_push ( ... )
{

}
```

Now it is up to the applications programmer to fill in what to do when the `_push` method gets invoked. The code segment in Figure 17.13 is helpful. Our `_push` implementation will be connected to both the IDM and ODM event channels. Inside the implementation we will handle each of the three event types separately. Initially, we define strings that map to the couple of enumerated types that are needed. If necessary take a look at the StandardEvents IDL contained in the Appendix C of the SCA specifications. The three SCA 'standard' event types are defined in the header given in Figure 17.13.

```
static char* sourceCategoryString[ ] = {
            "DEVICE_MANAGER",
            "DEVICE",
            "APPLICATION_FACTORY",
            "APPLICATION",
            "SERVICE"
}; /* the type of object being added-removed */

static char* stateCategoryString[ ] = {
            "ADMINISTRATIVE_STATE_EVENT",
            "OPERATIONAL_STATE_EVENT",
            "USAGE_STATE_EVENT"
}; /* for use by Devices on the IDM */

static char* stateChangeString[ ] = {
            "LOCKED",
            "UNLOCKED",
            "SHUTTING_DOWN",
            "ENABLED",
            "DISABLED",
            "IDLE",
            "ACTIVE",
            "BUSY"
}; /* for use by Devices on the IDM */
```

Figure 17.13. IDM Event Channel data structures

The first array of strings is for use with the ODM (added-removed events), the last two with IDM (state change events). Although Added events and Removed events are quite similar – the RemovedEvent data structure is a strict subset of the AddedEvent data structure – they must be parsed separately because they are, in fact, different types. The code example in Figure 17.14 shows an implementation of a `_push` method on a Consumer object. The example shows how to work with the CORBA *any* type.

In a nutshell, the CORBA *any* type contains two fields. One field contains a description of the type being passed – the `_type` field – and the other a `_value` field. The `_type` field can describe anything from simple `CORBA_long` to a `CORBA_sequence` of `CORBA_strings` to an elaborate `CORBA_struct`. The other field is a void pointer to the data itself. The example given in Figure 17.14 shows how to interpret both the `_type` and `_value` fields.

```
static void
impl_CosEventComm_PushConsumer_push(
                impl_POA_CosEventComm_PushConsumer *servant, const
                CORBA_any* data,
                CORBA_Environment *ev) {

  SCA_StandardEvent_DomainManagementObjectAddedEventType added;
  SCA_StandardEvent_DomainManagementObjectRemovedEventType removed;
  SCA_StandardEvent_StateChangeEventType state;

  printf("_push: %s\n",CORBA_TypeCode_name(data->_type,ev));

    if ( CORBA_TypeCode_kind(data->_type,ev) != CORBA_tk_struct )
    {
      printf("_push: expecting a struct\n");
      return;
    }

    if ( strcmp("DomainManagementObjectAddedEventType",
      (const char*)CORBA_TypeCode_name(data->_type,ev))==0
    {
     added = *(SCA_StandardEvent_DomainManagementObjectAddedEventType* )
       data->_value;
     printf("%s: ",CORBA_TypeCode_member_name(data->_type,0,ev));
        printf("%s\n",added.producerId);
     printf("%s: ",CORBA_TypeCode_member_name(data->_type,1,ev));
        printf("%s\n",added.sourceId);
     printf("%s: ",CORBA_TypeCode_member_name(data->_type,2,ev));
        printf("%s\n",added.sourceName);
     printf("%s: ",CORBA_TypeCode_member_name(data->_type,3,ev));
        printf("%s\n\n",sourceCategoryString[added.sourceCategory]);
    } else if
       ( strcmp("DomainManagementObjectRemovedEventType",
```

Figure 17.14. IDM/ODM pushConsumer – *any* type resolution

```
30.     (const char*)CORBA_TypeCode_name(data->_type,ev))==0 )
31.   {
32.  removed =
     *(SCA_StandardEvent_DomainManagementObjectRemovedEventType*)
      data->_value;
33.    printf("%s: ",CORBA_TypeCode_member_name(data->_type,0,ev));
34.        printf("%s\n",removed.producerId);
35.    printf("%s: ",CORBA_TypeCode_member_name(data->_type,1,ev));
36.        printf("%s\n",removed.sourceId);
37.    printf("%s: ",CORBA_TypeCode_member_name(data->_type,2,ev));
38.        printf("%s\n",removed.sourceName);
39.    printf("%s: ",CORBA_TypeCode_member_name(data->_type,3,ev));
40.        printf("%s\n\n",sourceCategoryString[removed.sourceCategory]);
41.  } else if
42.    ( strcmp("StateChangeEventType",
43.     (const char*)CORBA_TypeCode_name(data->_type,ev))==0)
44.  {
45.   state = *(SCA_StandardEvent_StateChangeEventType* )data->_value;
46.   printf("%s: ",CORBA_TypeCode_member_name(data->_type,0,ev));
47.       printf("%s\n",state.producerId);
48.   printf("%s: ",CORBA_TypeCode_member_name(data->_type,1,ev));
49.       printf("%s\n",state.sourceId);
50.   printf("%s: ",CORBA_TypeCode_member_name(data->_type,2,ev));
51.       printf("%s ",stateCategoryString[state.stateChangeCategory]);
52.       printf("From %s ",stateChangeString[state.stateChangeFrom]);
53.       printf("To %s\n\n",stateChangeString[state.stateChangeTo]);
54.  } else
55.    printf("Unknown struct %s\n",CORBA_TypeCode_name(data->_type,ev));
56.
57. }
```

Figure 17.14. (Continued)

A quick run through of the code example is in order. The basic design is to print out all 'string' or 'enumerated' fields of the SCA StandardEvents. In line 8 we print the `_type` of event being received in the call. Of course this will be Object Added, Object Removed, or State Change. If the `_type` received is not a `CORBA_struct` a message is printed and the routine returns lines 10–13. Lines 16–17, 29–30, and 41–43 each test for one of the three SCA-defined standard event types. If the type of structure passed is not one of the recognized SCA Standard Events, an error message is printed in line 55.

Each of the parsing sequences starts by casting the void pointer contained in `_value` to the correct type. Finally the pointer is de-referenced and the passed data is copied into a local variable (lines 19, 32 and 45). Each line printed contains two parts. The first part is the name of the member variable within the structure. Remember this information is passed as part of the `_type` variable and is accessed through the

CORBA_TypeCode_member_name() API. The second half of each line printed is the content appropriate member variable within the de-referenced _value field. A complete description of the *any* type, parsing of complex CORBA_structs and memory management is beyond the scope of this book, but can be found elsewhere [28, paragraph 8.7]. This reference is the full blown CORBA specification (version 2.2) upon which the SCA's minimum CORBA specification is based. The reader will come to find that much of the information sent to the event channels is also available through the Lightweight Log Service or directly accessible as attributes on the *DomainManager* object.

It is unfortunate to observe that in the minimum CORBA specification the member_name() interface, as well as several other interfaces used to decode *any* types, are generally not supported in ORBs claiming to be minimum CORBA-compliant. Since the SCA explicitly defines the members of the Standard Event data structures there is really no need to call member_name() in the run-time because the member names can be hard-coded.

With our _push implementation now complete it is time to compile it in to some additional software that will create the PushConsumer object, register it with the DomainManager, and spin like a server waiting for events to get pushed to it. We've not created a server object in any of our previous code examples. We've always acted as a client by running methods on existing CORBA objects living within the Core Framework. The code segment in Figure 17.15 shows how to create an instance of the Portable Object Adaptor (POA). Subsequently we will create our server-side object on that POA.

```
1. static
2. void startOrbPOA(int* argc, char** local_argv,
3.         CORBA_ORB *orb, PortableServer_POA *poa, CORBA_Environment *ev)
4. {
5.   PortableServer_POAManager poa_manager=CORBA_OBJECT_NIL;
6.
7.   *orb = CORBA_ORB_init(argc, local_argv, "orbit-local-orb", ev);
8.
9.   *poa = (PortableServer_POA)
10.       CORBA_ORB_resolve_initial_references(*orb,"RootPOA",ev);
11.
12.   poa_manager =
13.      PortableServer_POA__get_the_POAManager(*poa,ev);
14.
15.   PortableServer_POAManager_activate(poa_manager,ev);
16.
17.   CORBA_Object_release((CORBA_Object)poa_manager,ev);
18.   printf("ORB/POA initialized\n");
19.
20. }
```

Figure 17.15. CORBA server-side POA startup

For the sake of brevity we have left out tests for exceptions and nil references, but certainly a robust code will not make the same omission. In the same manner as in CORBA client-side code, line 7 creates an instance of an orb. As before, command line parameters are passed to the ORB initialization routine – previously this was the IOR of the Naming Service. The rest of the code example simple creates and activates a server-side POA. The ORB and POA pseudo-object references are returned to the calling program in the parameter list. We can now focus on the main program given in Figure 17.16.

```

#include <ORBitservices/CosNaming.h>
#include "stubs/EventService.h"
#include "stubs/SCA.h"

#include "scaPushConsumer_impl.c"

int main( int argc, char * argv[] )
{
    CORBA_Environment ev;
    CORBA_ORB orb=CORBA_OBJECT_NIL;
    PortableServer_POA my_poa=CORBA_OBJECT_NIL;
    CosEventComm_PushConsumer servant=CORBA_OBJECT_NIL;

    CosNaming_NamingContext myNC;
    SCA_CF_DomainManager dmObj;

    CosNaming_Name myBindingName;
    CosNaming_NameComponent
            path[2]={{"SCARI_DM",""},{"DomainManager",""}};
    int localArgc=argc;

    CORBA_exception_init(&ev);
    --localArgc;
    startOrbPOA(&localArgc,argv,&orb,&my_poa,&ev);

    getNamingService(argc, argv, &myNC, &orb, &ev);

    /* get the Domain Manager objRef */
    myBindingName._maximum=2;
    myBindingName._length=2;
    myBindingName._buffer=path;
   dmObj=(SCA_CF_DomainManager)CosNaming_NamingContext_resolve(myNC,
              (const CosNaming_Name *)&myBindingName, &ev);

   /* create the PushConsumer object */
   servant =
     impl_CosEventComm_PushConsumer__create(my_poa,&ev);
   printf("Have a valid PushConsumer\n");

   /* register with ODM */
```

Figure 17.16. Register with Event Channel – main

```
41.  SCA_CF_DomainManager_registerWithEventChannel(dmObj,
42.              (const CORBA_Object)servant,
43.              "johnConsumer", "ODM_Channel", &ev);
44.
45.  /* register with IDM */
46.  SCA_CF_DomainManager_registerWithEventChannel(dmObj,
47.              (const CORBA_Object)servant,
48.              "anotherConsumer", "IDM_Channel", &ev);
49.
50.  CORBA_Object_release(dmObj, &ev);
51.  CORBA_Object_release(myNC, &ev);
52.
53.  CORBA_ORB_run(orb,&ev); /* Server runs forever */
54.
55.  CORBA_ORB_destroy(orb,&ev);
56.
57.  return 0;
58. }
```

Figure 17.16. (Continued)

Besides the normal include files, e.g. <stdio.h>, a couple of CORBA service specific include files, as well as the SCA, are also required. Line 2 provides the prototypes for using the Naming Service to locate the Domain Manager object reference. Line 3 is needed for its definition of a the *PushConsumer* object. Line 4 is needed for the Domain Manager object and its methods `registerWithEventChannel` and `unregisterFromEventChannel`. It is assumed that the IDL for the SCA Core Framework has been complied and the output lives in a directory called `stubs` beneath the current directory. Most importantly our code for the scaPushConsumer implementation is included directly in line 6. The rest of the code example is pretty straightforward. Lines 19 and 29–33 show an abbreviated way of directly accessing the Domain Manager's node within the naming graph. Our previous encounter in exploring the naming graph led us to a recursive means of accessing the various nested levels.

Line 32 resolves the `CosNaming_Name` of the *DomainManager* and returns an object reference. We will use this later to register our *PushConsumer*. Line 36 creates the PushConsumer object (on the POA) and returns its object reference in the variable called `servant`. Finally lines 40–48 register our PushConsumer with both the ODM and IDM channels. The NamingContext and DomainManager object references are no longer needed so they are released (lines 50 and 51). At last our POA server is put into the run state. The code never actually returns from the call in line 53.

To run this executable type/paste the following unto the command line. The server will run until you kill it.

```
./PCserver –ORBIIOPIPv4=1 IOR:… paste IOR of Naming Service here…
```

The first parameter is very important. The ORBit2 orb is used extensively within the GNOME desktop environment. Typically there are four or five server objects running at any given time. The default setting for ORBit2 is to make these connections private. This security feature uses a protected GIOP interface that is not known to other ORBs. This command line flag tells ORBit2 to use the universal IIOP in constructing object references. Without this flag the object reference used to register our *PushConsumer* with the *DomainManager* would be unrecognizable by the Java ORB. Finally, the second parameter required is the IOR for the Naming Service so that our software can resolve the *DomainManager* object reference. Upon successful execution the output given in Figure 17.17 is produced.

```
Resolved NameService
ident:   DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
profile: <profile filename="/DomainManager.dmd.xml" type="DMD"/>
Have a valid PushConsumer

Event 1 → _push: StateChangeEventType
  producerId: DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d
  sourceId: DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d
  stateChangeCategory: USAGE_STATE_EVENT From IDLE To ACTIVE
Event 2 → _push: DomainManagementObjectAddedEventType
  producerId: DCE:9601C10A-249F-48B0-8C5A-BE61545BB101
  sourceId: AudioEffect0_DCE:9601C10A-249F-48B0-8C5A-BE61545BB101
  sourceName: myApp
  sourceCategory: APPLICATION
Event 3 → _push: StateChangeEventType
  producerId: DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d
  sourceId: DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d
  stateChangeCategory: USAGE_STATE_EVENT From ACTIVE To IDLE
Event 4 → _push: DomainManagementObjectRemovedEventType
  producerId: AudioEffect0_DCE:9601C10A-249F-48B0-8C5A-BE61545BB101
  sourceId: AudioEffect0_DCE:9601C10A-249F-48B0-8C5A-BE61545BB101
  sourceName: myApp
  sourceCategory: APPLICATION
Event 5 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:d58c8931-7372-473c-b9be-8f15c995b75f
  sourceName: AnalogFM
  sourceCategory: APPLICATION_FACTORY
Event 6 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB
  sourceName: node1AudioDevice
  sourceCategory: DEVICE
Event 7 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d
  sourceName: node1ExecutableDevice
  sourceCategory: DEVICE
Event 8 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888
  sourceName: node1RFDeviceImpl
  sourceCategory: DEVICE
```

Figure 17.17. Event Channel output

```
Event 9 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:f186ff96-b9aa-4d17-97e3-0999fca410b4
  sourceName: node1Logger1
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:d58c8931-7372-473c-b9be-8f15c995b75f
  sourceName: AnalogFM
  sourceCategory: APPLICATION_FACTORY
Event 10 → _push: DomainManagementObjectRemovedEventType
  producerId: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
  sourceId: DCE:18dd6458-494e-43cd-b823-07778bb9ca51
  sourceName: node1DeviceManager
  sourceCategory: DEVICE_MANAGER
```

Figure 17.17. (Continued)

Examination of the output yields a few worthwhile observations. The first bit of output comes from the pushConsumer program itself saying it has found the Domain Manager and then announcing the successful deployment of a PushConsumer server object. Subsequent output is from the event channels themselves.

Events 1 and 2 are associated with the creation of the AudioEffect Application. The first event is the Executable device changing from IDLE to ACTIVE. The application named ‘myApp’ is then added to the domain. Now the user can start and stop the AudioEffect application as desired. One would expect to find state change messages coming from the Audio Device as the application was started and stopped. This is not the case. It is likely that the Audio Device is started during startup of all the devices on the domain. Our event channel consumer was not registered at that time so those events were missed.

Events 3 and 4 are associated with shutting an application down. The ExecutableDevice is set to IDLE and then the AudioEffect application is removed. Event 5 confirms the removal of the AnalogFM application factory. And the remainder of events concern the shutdown of all devices on the domain. Event 6 confirms removal of the audio device. What’s curious is that state change events for the Audio Device are not observed as the system was shutting down – perhaps this is an omission in the SCARI Audio device implementation. For instance as part of the `releaseObject()` one expects to see adminState transition from `UNLOCKED` to `SHUTTING_DOWN`. The rest of the devices are removed, finally the Log service and then the device manager.

There is a bit of a problem in the example code and it is the ungraceful way that we are forced to shutdown down the *PushConsumer* object server. When the user sends a ctrl-C signal to the process it shuts down without un-registering from the *DomainManager*. An attempt to execute the code again will result in an ‘AlreadyConnected’ exception being generated by the *DomainManager*. Without an exception handler in place the user will just get a Segmentation Fault with no explanation. There will be an error message sent to the Log Service but we haven’t learned about that yet. The means of solving this problem is to write a signal handler that will call the *unregister* method when the user sends a signal to the process. This signal handler is shown in Figure 17.18.

```
1. static CORBA_ORB global_orb=CORBA_OBJECT_NIL;
2.
3. static void orderlyShutdown (int sig)
4. {
5.    CORBA_Environment ev;
6.    CORBA_exception_init(&ev);
7.
8.    if (global_orb != CORBA_OBJECT_NIL)
9.     {
10.      /* This will cause the main program to break */
11.             /* out of the CORBA_ORB_run call */
12.       CORBA_ORB_shutdown (global_orb, FALSE, &ev);
13.             /* put exception handler here */
14.     }
15. }
```

Figure 17.18. Server-side signal handler

This routine will be registered within the main program as a signal handler. Note in line 1 that we have moved the `CORBA_ORB` variable from `main()` context to global context. This way the signal handler can see it too in order to shut it down. Don't forget to make the necessary adjustments in the main program and `startOrbPOA` routines. This code example pushes us into a bit of a quandary for `CORBA_ORB_shutdown()` is not supported by minimum CORBA ORB's. With this restriction we would be forced to make the `unregister ()` CORBA call from within the signal handler – this is not good practice. While in the main program add the following lines to register our signal handler.

```
25.3   signal(SIGINT, orderlyShutdown);
25.7   signal(SIGTERM, orderlyShutdown);
```

The program will now require the 'signal.h' header file. The first signal registration is to intercept 'ctrl-C'; the second is to intercept the signal normally generated by the command line 'kill' command. Referring back to our main program when the user hits ctrl-C the installed signal handler will run and the main program will return from the call `CORBA_ORB_run` in line 53. This is our opportunity to clean up and unregister our *PushConsumer* from the event channels. Before attempting to unregister it is first necessary to shutdown the *PushConsumer* servant. The code segments given in Figure 17.19 will perform the necessary clean-up operations.

```
14.    PortableServer_ObjectId *objid=NULL; /* add this declaration */
       . . .
50. /* CORBA_Object_release(dmObj, &ev);still need this for unregister */
51.    CORBA_Object_release(myNC, &ev);
52.
53.   CORBA_ORB_run(global_orb,&ev);   /* Server runs forever */
54.   printf("main() back alive\n", );
55.
56.   /* destroy the PushConsumer */
57.   objid = PortableServer_POA_reference_to_id (root_poa,
58.           (const CORBA_Object)servant, &ev);
```

Figure 17.19. Unregister from Event Channel server shutdown

```
59.
60.  PortableServer_POA_deactivate_object (root_poa, objid, &ev);
61.  CORBA_free (objid);  /* don't need this anymore */
62.
63.  /* now unregister */
64.  SCA_CF_DomainManager_unregisterFromEventChannel(
65.          dmObj, "anotherConsumer", "IDM_Channel", &ev);
66.  SCA_CF_DomainManager_unregisterFromEventChannel(
67.          dmObj, "johnConsumer", "ODM_Channel", &ev);
68.  CORBA_Object_release(dmObj, &ev);  /* don't need this anymore */
69.
70.  PortableServer_POA_destroy (root_poa, TRUE, FALSE, &ev);
71.  CORBA_Object_release(root_poa, &ev);  /* don't need this anymore */
72.  CORBA_Object_release(servant, &ev);  /* nor this */
73.
74.  CORBA_ORB_destroy(global_orb,&ev);   /* bye orb */
75.
76.  return 0;
```

Figure 17.19. Unregister from Event Channel server shutdown

This code example overlays the previous `main()` example starting at line 50. As before `main()` will go in to a server loop in line 53. This time the signal handler will intercept the ctrl-C and execution will ‘re-start’ at line 54 wherein the process of shutting down and cleaning up begins. First destroy the *PushConsumer* servant. This is done by retrieving its POA identification and then de-activating it (lines 57–60). Second the *PushConsumer* is removed from the *DomainManager’s* internal list of consumers. The *DomainManager* will also take care of disconnecting the ProxyPushSupplier objects that were associated with our *PushConsumer*. Finally, the root POA is destroyed and unused objects released. The orb itself is destroyed just prior to exiting the main program. This program can be re-started at anytime without the *DomainManager* complaining about it.

We have shown how to implement and register the consumer side of the Event Channel push model. The reader is now able to listen in on the IDM and ODM event channels. The next obvious question is how to establish the supplier side. If the applications programmer wishes to write a *Device* or an *ApplicationFactory* it is necessary to access the IDM and ODM event channels as a *PushSupplier*. The SCA does not specify exactly how to do this but in the SCARI Core Framework *PushSupplier* connections to an event channel are handled as a specialized port connect.

17.2.1 Core Framework Usage of the Event Service

The usage of event channels by Core Framework objects is clearly delineated in the SCA. When a client creates an Application by invoking the `create()` operation on an ApplicationFactory interface, the successful creation is noted by an ‘ObjectAdded’ event on the ODM event channel. When a Device Manager is successfully registered with the `registerDeviceManager()` operation on the Domain Manager interface, an ‘ObjectAdded’ event is sent on the ODM event channel. When a Device is successfully registered with the `registerDevice()` operation on the Domain Manager interface, an ‘ObjectAdded’ event is sent on the ODM event channel. When an application is successfully installed by invoking `installApplication()` on the Domain Manager, an 'ObjectAdded' event is posted on the ODM event channel. As with all 'ObjectAdded' events, the newly installed ApplicationFactory stringified IOR is published on the event channel. When a service is registered with the Domain Manager's `registerService()` operation, an 'ObjectAdded' event is placed on the ODM event channel.

When a client invokes the `unregisterDeviceManager()` operation on the Domain Manager interface, an 'ObjectRemoved' event is posted on the ODM event channel. When a client invokes the `unregisterDevice()` operation on the Domain Manager interface, an 'ObjectRemoved' event is posted on the ODM event channel. When a client invokes the `uninstallApplication()` operation on the Domain Manager interface, an 'ObjectRemoved' event is placed on the ODM event channel. Finally, when a service is removed by the `unregisterService()` operation an 'ObjectRemoved' event is placed on the ODM.

17.2.2 Resource Usage of the Event Service

There are two primary usages of the event service required by a Resource. One usage applies only to Devices and the other applies only to Application. The first usage is strictly for Devices when they experience a change in state. When any of the three Device states change – usageState, adminState, or operationalState – a 'StateChange' event is required to be supplied to the IDM event channel. The second usage is when a client invokes a `releaseObject()` operation on an Application interface. The successful release of the all the application components shall be recorded with an 'ObjectRemoved' event on the ODM event channel. The `releaseObject()` operation is also inherited by Devices. As part of `releaseObject()` processing a device will change its adminState to `'SHUTTING_DOWN'` and this, of course, will be recorded on the IDM event channel. Finally a Device is required to unregister itself with a Device Manager. The Device Manager will in turn perform an `unregisterDevice()` on the Domain Manager. This event will be recorded on the ODM event channel.

17.3 Log Service

As discussed the original SCA Log Service migrated in to the Object Management Group as a separate RFP that eventually became the Lightweight Log Service. Version 2.2.1 of the SCA then removed that entire sub-section devoted to the original Log Service and replaced it with a reference to the new OMG standard. This effort was partially funded by the US DoD in that many of the authors and contributors to the SCA were under contract to 'commercialize' the SCA. This important theme of the JTRS program was spelled out very clearly in the JTRS Operational Requirements Document (ORD). The government specifically did not want another Ada on their hands. Part of the charter of the JTRS Joint Program Office was to see the commercial software radio community not only endorse the SCA but also embrace it.

There were a few aspects to the commercialization charter that weren't clear. One was the erroneous assumption that the commercial community would accept the SCA as it stood. This simply could not happen – the Object Management Group and SDR Forum both have review and voting procedures that allow an input document to mature to a standard. The paying membership of these organizations are endowed the right to contribute to the editorial maturation process of a document. This natural progression is how good ideas become great implementations, and furthermore, now not-so-good ideas 'die on the vine' for lack of commercial interest.

Also, another unclear aspect was that the government themselves had an internal Change Proposal process that allowed defense contractors to contribute to the SCA, but unfortunately excluded some members of the commercial community. Finally, and most obviously, was how to retrofit the upgrades and modifications brought forth by the commercial community into the DoD process. This last step keeps the commercialization relevant.

Commercialization is the process of staying synchronized with the community at large as opposed to it being a one-time event. The beauty of the SCA version 2.2.1 release was this retrofitting of the commercial advances back into the DoD standard, for what had so far been a one-time event. The commercial SCA – that is, the Software Radio PIM/PSM and ancillary standards – continues to diverge from its military antecedent. In conclusion one should expect the DoD standard to somewhat lag the commercial standard, as long as the commercial community stays interested.

This is not unlike the dynamic between standards organizations and the commercial marketplace. The business model of a commercial standards organization is to produce standards – where new product doesn't exist then its time to upgrade the existing product line. Commercial consumers of standards – producers of software product – will not just automatically upgrade their product lines to the next new level of compliance. There has got to be a demonstrated market/sales for the upgrade before management gives the thumbs up. So even in the commercial world there is a lag, sometimes substantial, between the current version of a specification and the version of specification that is prevalent in the marketplace.

This section will provide a code example (Figure 17.20) for the user to access the Log Service in a consumer role. As we attempt to use the Core Framework in our application development the feedback offered by the Log Service is an indispensable tool in tracking down problems as well as monitoring healthy behavior and normal operation. The UML for the Lightweight Log Service is found in Figure 3.5. It is key to understand that the *LogRecordType* is a wrapper for the *ProducerLogRecordType* which is the structure that actually contains the fields of interest. The wrapper contains an unsigned long long *Id* and a POSIX timespec *time*. The SCA really says nothing about how to treat either of the fields but logically *Id* would start at zero or one and be monotonically increasing and that *time* would be related to system time, user time, or even UTC clock time. The code segment in Figure 17.20 will playback the log record from its origin and then enter a sleepy loop to check for new log messages.

```
1. static const struct timespec fifthSecond = {0,200000000};
2.
3. int main( int argc, char * argv[] )
4. {
5.     CORBA_Environment ev;
6.     CORBA_ORB orb;
7.     CosNaming_NamingContext myNC;
8.     SCA_LogService_Log lgObj;
9.
10.      SCA_LogService_Log_RecordIdType logNum;
```

Figure 17.20. Log Service playback

```
  SCA_LogService_Log_LogRecordSequence *aPtr;
  SCA_LogService_Log_LogRecordType aLogRecord;
  SCA_LogService_ProducerLogRecordType embeddedRecord;

  CORBA_exception_init(&ev);
  startOrb(&argc,argv,&orb,&ev);

  myNC = CORBA_ORB_resolve_initial_references(
             orb,"NameService",&ev);

  /* get the logger objRef */
  lgObj = (SCA_LogService_Log)
    CosNaming_NamingContextExt_resolve_str(myNC,
        "node1Logger1", &ev);

  logNum=0;
  aPtr = SCA_LogService_Log_retrieveById(lgObj, &logNum,
           (const CORBA_unsigned_long)1, &ev);
  while (aPtr->_length==1) {
    aLogRecord = *(aPtr->_buffer);
    embeddedRecord = aLogRecord.info;
    printf("%s %s\n",
       embeddedRecord.producerName,embeddedRecord.logData);
    CORBA_free(aPtr);
    aPtr = SCA_LogService_Log_retrieveById(lgObj, &logNum,
             (const CORBA_unsigned_long)1, &ev);
  }

  /* Enter timed loop */
  while (1) {
    nanosleep(&fifthSecond,NULL);
    aPtr = SCA_LogService_Log_retrieveById(lgObj, &logNum,
             (const CORBA_unsigned_long)1, &ev);
    while (aPtr->_length==1) {
       aLogRecord = *(aPtr->_buffer);
       embeddedRecord = aLogRecord.info;
       printf("%s %s\n",
          embeddedRecord.producerName,embeddedRecord.logData);
       CORBA_free(aPtr);
       aPtr = SCA_LogService_Log_retrieveById(lgObj, &logNum,
                (const CORBA_unsigned_long)1, &ev);
    }
  }

  CORBA_Object_release(lgObj, &ev);
  CORBA_Object_release(myNC, &ev);
  return 0;
}
```

Figure 17.20. (Continued)

This code example uses `resolve_initial_references()` to retrieve the object reference to the Name Service (line 18). The user however must supply a location for the Name Service on the command line with

```
./playLog -ORBInitRef NameService=corbaloc::1.2@127.0.0.1:1050/NameService
```

or by hostname

```
./playLog -ORBInitRef NameService=corbaloc::1.2@monster:1050/NameService
```

or even by old-fashioned IOR

```
./playLog -ORBInitRef IOR:big long number goes here.
```

The code continues by obtaining the name of the Log Service object (line 22). The application programmer must know the vendor-specific name of the log service in order to use the Name Service to resolve the name of the Log Service. In this case *'node1Logger1'*, which happens to be visible from the initial context. This is somewhat non-compliant as the SCA specifications section 3.1.3.2.3.5 says that a service should be named within the /DomainManager context. The name of the Log Service can be found in the DMD domain profile file, *domainfinder* element, *name* attribute.

There is only one way to retrieve a record from the log and that is `retrieveById()`. This operation will return the specified number of records but the caller must also provide the starting *recordId*. We happen to know that the SCARI Log Service's initial *recordId* always starts at one and is monotonically increasing by one. Even without this knowledge one can likely get hands on the initial *recordId* by calling `getRecordIdFromTime()` with the timespec set to zero. So line 27 returns the first record in the log. Then lines 28 through 34 incrementally iterate through the entire log until such time that the `retrieveById()` operation returns a log sequence of length zero. This is an indication that the caller has requested a *recordId* that does not yet exist. When this occurs the code segment enters a forever loop (line 37) that wakes up five times a second (line 38) to test for the existence of the next record (line 40). Notice in line 44 that the memory allocation made by the callee is freed by the caller. If during the time the process was sleeping numerous records were written to the log, the while loop in lines 40 through 46 will iterate through all of them until exhausted.

The output produced by the Log Service is shown in Figure 17.21. The SCA specifies a minimum set of events to be logged but the platform or waveform developer can go beyond what's required.

```
1. DomainManager Connection DMDConnectionTonode1Logger1 is established.
   DomainManager has been connected to the Log.
2. node1DeviceManager Connection node1DeviceManagerToLog is established.
   node1DeviceManager has been connected to the Log.
3. DomainManager [1144002428556] installApplication::Application Factory
   Installed for /AudioEffect0/AudioEffectApplication0.sad.xml
4. DomainManager [1144002429109] installApplication::Application
     Factory
   Installed for /AnalogFM/AnalogFM.sad.xml
```

Figure 17.21. Log Service output

```
5. DomainManager [1144002430084] registerDeviceManager: DeviceManager
   node1DeviceManager has registered succesfully
6. DomainManager [1144002430300]
   registerService:[DomainManager:registerService] Service 'node1Logger1'
   has registered successfully
7. DomainManager registerDevice:[DomainManager:registerDevice] Device
   DCE:72f795a2-5f6a-41cf-8d3a-9beaeef0ac7d has successfulyregistered
8. node1ExecutableDevice Connection node1ExecutableDeviceToLog is
   established. node1ExecutableDevice has been connected to theLog.
9. node1ExecutableDevice [1144002423525]
   [ExecutableDeviceOperationsImpl:finalizeConstruction] Device
   node1ExecutableDevice WARNING: homeFileSystem directory not empty!
10. DomainManager registerDevice:[DomainManager:registerDevice] Device
   DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888 has successfulyregistered
11. RFDeviceImpl_node1RFDeviceImpl Connection node1RFDeviceImplToLog is
   established. RFDeviceImpl_node1RFDeviceImpl has been connected to the Log.
12. DomainManager registerDevice:[DomainManager:registerDevice] Device
   DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB has successfulyregistered
13. RFDeviceImpl_node1RFDeviceImpl
   [AggregateDeviceOperationsImpl:addDevice] DCE:A68A5812-6BE7-4920-9A29-
   A7C013734FAB was succesfully added
14. DomainManager SCA.CF.PortPackage.InvalidPort:
   IDL:CF/Port/InvalidPort:1.0 [ConnectionsHandler:isConnectable]
   'IDMChannelEventPort' is an invalid port name for source component in
   connection id=node1AudioDeviceToIDM_Channel_2
15. [ConnectionsHandler:isConnectable] 'IDMChannelEventPort' is an invalid
   port name for source component in connection
   id=node1AudioDeviceToIDM_Channel_2
16. AudioDevice_node1AudioDevice Connection node1AudioDeviceToLog is
   established. AudioDevice_node1AudioDevice has been connected to the
   Log.
17. AudioDevice_node1AudioDevice [1144002433284] [AudioDevice] Port
   'AudioInLeftDouble' should only be used with a stereo AUDIO_MODE
18. AudioDevice_node1AudioDevice [1144002433286] [AudioDevice] Port
   'AudioInRightDouble' should only be used with a stereo AUDIO_MODE
```

Figure 17.21. (Continued)

This logger output covers the boot up of the system including the re-installation of two application factories, AudioEffect0 and AnalogFM (lines 3 and 4). Notice the increasing time count in brackets. This LogTime type employs the commonly used POSIX timespec structure which contains two fields one for nanoseconds and one for seconds. The SCA does not specify the time base other than to say 'current local time'. What is observed in the time output is that time is not necessarily monotonically increasing. In going from Log Record 6 to 9 time goes backwards. This is not unlike most log services in that a process may be blocked – or the log resource otherwise locked – during the time between when the time tag is grabbed from the system and when it is output to the log. Depending on when the time tag gets generated relative to the actual production of log output it might be necessary to re-sort log output by time tag to see the true sequence of events.

Looking through the rest of the log output we see a few non-standard messages – that is messages that aren't SCA mandatory messages required of certain Core Framework events. These non-standard log messages include (line 9), home file system not empty and (line 15), invalid port name. The 'invalid port name' message gives rise to some concern and warrants further investigation. The error messages put unto the log service in lines 14 and 15 indicate that the 'invalid port name' occurs when the Node 1 devices are being deployed during startup of the domain. The error specifically occurs when attempting to connect the Audio Device to the IDM Event Channel. The error is specifically complaining about an invalid port name. Yet examination of the DCD file shows that the name 'IDMChannelEventPort' is completely consistent with the description of the Audio Device's ports as found in its SCD. It turns out there is a minor coding error in the SCARI Core Framework in line 693 of `$SCA_HOME/demosources/devices/AudioDevice.java`. The code should be modified to read:

```
//portObject = super.getPort(name); // <-- This is in error
portObject = super.getPort(portName);
```

It was previously mentioned that StateChange events were not being observed on the IDM event channel while the system was shutting down (see page 347). Unfortunately fixing this bug, which inhibited the Audio Device from connecting to the IDM event channel, does not eliminate this problem.

17.3.1 Core Framework Usage of the Log Service

The Core Framework's required use of the Log Service is nearly identical to that of the required use of the Event Service. A certain difference is that the SCA requires log entries to be tagged as `ADMINISTRATIVE` or as `FAILURE`s (Table 17.1). The Event Service possesses this notion of level or priority of event as does the Log Service.

Table 17.1 shows that the only difference of Log Service over Event Service is that any time a client accesses the list of deviceManagers, applications, applicationFactories, or File Manager a log message is posted with no corresponding event. Additionally when a Device

Table 17.1. Core Framework required use of Log Service

Operation	Log Level	Event Channel also
`ApplicationFactory.create()`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`DomainManager.get_deviceManagers`	`ADMINISTRATIVE`	
`DomainManager.get_applications`	`ADMINISTRATIVE`	
`DomainManager.get_applicationFactories`	`ADMINISTRATIVE`	
`DomainManager.get_fileMgr`	`ADMINISTRATIVE`	
`DomainManager.registerDeviceManager`	`FAILURE` if not successful Missing in SCA if successful	No

Table 17.1. Continued

Operation	Log Level	Event Channel also
`DomainManager.registerDevice`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`DomainManager.installApplication`	`ADMINISTRATIVE`if successful `FAILURE` if not successful	Yes
`DomainManager.unregisterDeviceManager`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`DomainManager.unregisterDevice`	`ADMINISTRATIVE`if successful `FAILURE` if not successful	Yes
`DomainManager.uninstallApplication`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`DomainManager.registerService`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`DomainManager.unregisterService`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes
`AggregateDevice.addDevice`	`FAILURE` if notsuccessful	
`AggregateDevice.removeDevice`	`FAILURE` if not successful	
`DeviceManager.registerDevice`	`FAILURE` if not successful	
`DeviceManager.unregisterDevice`	`FAILURE` if not successful	
`DeviceManager.registerService`	`FAILURE` if not successful	
`DeviceManager.unregisterService`	`FAILURE` if not successful	

Manager installs successfully no log record need be written; only an ‘ObjectAdded’ event needs to be sent to the ODM event channel. This is probably an oversight on the part of the SCA authors. Finally we see that when Devices and Services are registered and unregistered on a Device Manager only failures are required to be logged.

17.3.2 Resource Usage of the Log Service

*Device*s and *Application*s have only one required use of the Log Service (Table 17.2). Depending on the Core Framework implementation this operation might or might not need to be overloaded by the application or device programmer.

Table 17.2. Resource required use of Log Service

Operation	Log Level	Event Channel also
`Application.releaseObject()`	`ADMINISTRATIVE` if successful `FAILURE` if not successful	Yes

18

Exploring the Domain

With the component names provided by the Name Service it is now possible to retrieve the object references of all artifacts made visible by the Core Framework. This brings up the question of granularity. Should an application consist of four visible components or forty? The SCA is silent on the subject. People talk about the spirit of the SCA; that is having a multitude of relocate-able hardware independent CORBA components. Although an implementation consisting of a myriad of objects spread across a universe of processors is clearly supported by the SCA, the realities of a real-time software radio having limited computational resources tends to push the implementation more towards the hardware. This is especially true if the waveforms to be hosted are bandwidth intensive and/or the radio platform is constrained by size, weight, and power.

There is a waveform that was done in the spirit of the SCA. It was composed of 33 CORBA components, 400+ ports, and over 200 000 lines of XML. This waveform was to be ported to a variety of platforms and, needless to say, memory footprint, CPU utilization, and time to instantiate became a stumbling block. The fact is that both commerce and the military want a radio with giga-bit per-second throughput and that runs on a hearing-aid battery, but the spirit of the SCA is somewhat contrary to these objectives. The software radio concept works nicely with relatively narrow band waveforms – consider the military and public safety domains where backwards compatibility is the precedent – but falls apart on bandwidth, size, weight, and power. It is possible to construct platform independent code that handles 25 year old modulation techniques but such code is not an efficient user of clock cycles or bandwidth and does not scale well in to, say, the handheld domain.

So now we see the wisdom of the authors of the SCA in their reluctance to specify granularity. On the one hand the SCA supports a waveform implemented entirely in FPGA and on the other a waveform with hundred of thousands of lines of high level code running on big, beefy multiple processors. Additionally there is also no guidance on the selection of Properties. The SCA offers configure and query methods and that is all. What needs to be configured and queried is platform- as well as application-dependent. Other than mandating the tracking of capacity on devices that have a *kindtype* attribute ‘allocation’, the SCA does not specify a vocabulary or semantics for capacity tracking. Thus we see that SCA by itself does not specifically enable portability of applications among platforms. That is, looking down into the radio *Device*s there is no guaranteed set of configurable parameters.

The *PropertySet* interface is inherited by all *Resource* objects and thus, by inheritance, all *Device*s and *Application*s. The PropertySet interface is also inherited by *DeviceManager*s and the *DomainManager*.

The Name Service provides the object reference of the *DomainManager* through the `CosNaming_NamingContext_resolve()` operation. In a sense having the *DomainManager* object reference is like having the keys to the kingdom. Among other things the *DomainManager* also acts like a Name Service: that is, the *DomainManager* maintains data structures that contain object references to device managers, application factories, and applications. Through device managers one can gain access to the object references of all registered devices. Surprisingly there is no way to access the object references of application components through the DomainManager. Only the Name Service or an *Application* object can provide object references to application components. We will present code segments in the form of subroutines to access all attributes; first application factory objects, second applications, and then device managers and devices. The final segment presented is the entry point into the hierarchy – the main program – which provides access to the *DomainManager*.

18.1 Application Factory Attributes

The function shown in Figure 18.1 accepts an object reference to an application factory and then print outs all accessible information.

```
void queryAppFact(SCA_CF_ApplicationFactory anAppFact) {
  CORBA_Environment ev;
  CORBA_string aString;

  CORBA_exception_init(&ev);

  aString = SCA_CF_ApplicationFactory__get_name (anAppFact,&ev);
  printf(" ApplicationFactory name: %s\n",aString);
  CORBA_free(aString);
  aString = SCA_CF_ApplicationFactory__get_identifier(anAppFact,&ev);
  printf(" ApplicationFactory identifier: %s\n",aString);
  CORBA_free(aString);
  aString =
  SCA_CF_ApplicationFactory__get_softwareProfile(anAppFact,&ev);
  printf(" ApplicationFactory softwareProfile: %s\n\n",aString);
  CORBA_free(aString);
}
```

Figure 18.1. Query Application Factory function

It's worthwhile to note at this time that a preamble introduces all function calls to the Core Framework. This preamble is `SCA_CF_`. The 'C' language mapping for CORBA handles the nesting of modules and interfaces by concatenating the name of the parent module together with the name of the child module separated by an underscore. Our code examples use IDL files that came from the SCARI software tree. The SCARI developers gathered all of the SCA modules – CF, PortTypes, LogService, PushPorts, PullPorts, and StandardEvents – into a single module called SCA. Any Core Framework implementation that doesn't perform this nesting will have different names for Core Framework methods. Most likely this would be `'CF_'` instead of `'SCA_CF_'`. In order to port our code examples it might be required to perform some global search and replaces.

We will examine the output later but essentially there are only three accessible attributes on an application factory: the string name of the application that is manufactured by the factory; a unique identifier for this application factory instance; and the name of the SAD XML file that is parsed when one invokes the `create()` method. There is nothing spectacular with this code segment, just invocations of `__get` methods to retrieve string parameters. We are careful to return allocated memory to the heap with `CORBA_free()`. The allocation is performed by the callee and it's the responsibility of the caller to free the memory in order to avoid memory leaks. Responsibility for allocating and freeing memory when making CORBA-client calls with the 'C' language has been defined elsewhere [13, pages 1–24]. Memory leaks when using CORBA is assured if a methodical memory management policy is not followed by the programmer.

The output shown in Figure 18.2 is produced when the Audio Application Factory is queried on the domain. Note that an application must be 'installed' within the Domain for its object reference to be available to the *DomainManager*. In our example two applications have been installed.

```
ApplicationFactory name: AudioEffect0
ApplicationFactory identifier: DCE:9601C10A-249F-48B0-8C5A-BE61545BB101
ApplicationFactory softwareProfile: <profile

filename="/AudioEffect0/AudioEffectApplication0.sad.xml" type="DMD"/>

ApplicationFactory name: AnalogFM
ApplicationFactory identifier: DCE:d58c8931-7372-473c-b9be-8f15
c995b75f
ApplicationFactory softwareProfile: <profile
            filename="/AnalogFM/AnalogFM.sad.xml" type="DMD"/>
```

Figure 18.2. Application Factory outptut

The identifier of an *ApplicationFactory* object accessible in the run-time is exactly the same as the `Id` attribute of the `softwareassembly` element taken from the application's SAD file – see SCA specifications, paragraph 3.1.3.2.2.4.3. An *ApplicationFactory*'s name 'shall contain the type of Application that can be instantiated' by the factory. There is no guidance offered on the contents of this name attribute and furthermore there is no explicit mapping to the Domain Profile. The content of this field is most likely the `name` attribute of the `softwareassembly` element from the application's SAD file. Finally the software profile attribute contains the filename of the SAD that is parsed when the `create()` method is invoked on the factory. The end of the string, `type="DMD"`, is hard-coded in the SCARI software and really has no significance.

It is good to have unique identifiers for all artifacts within our SCA-compliant radio but it makes for very poor reading in the course of our book. Instead of printing out 128-bit numbers in future examples we will printout ‘DCE:’ followed by a unique string that is more meaningful to the reader. This makes it easier to see how UUIDs are cross-referenced within the run-time and within the Domain Profile.

18.2 Application Attributes

The code segment in Figure 18.3 accepts an object reference to an *Application* and then retrieves all available run-time attributes of the application. There are a total of seven retrievable attributes on each application registered within the domain: the filename of the application’s SAD; a name given to the instantiation by the ApplicationFactory when the application is created; four unbounded sequences that describe components and dependencies of the Application; and finally, since *Application* inherits from *Resource*, it also has an `identifier` attribute.

```
void queryApp(SCA_CF_Application anApp) {

  CORBA_Environment ev;
  int i;
  SCA_CF_Application_ComponentElementSequence* cesSeq;
  SCA_CF_Application_ComponentElementType aCES; /* string componentId;
                                                 CORBA_string elementId; */
  SCA_CF_Application_ComponentProcessIdSequence* cpidSeq;
  SCA_CF_Application_ComponentProcessIdType aCPID; /*string componentId
                                               CORBA_unsigned_long processId */
   SCA_CF_DeviceAssignmentSequence* dasSeq;
   SCA_CF_DeviceAssignmentType aDAS; /* CORBA_string componentId;
      CORBA_string assignedDeviceId; */
   CORBA_string aString;

   CORBA_exception_init(&ev);

   cesSeq =
   SCA_CF_Application__get_componentNamingContexts(anApp,&ev);
   printf(" There are %d NamingContexts\n",(*cesSeq)._length);
   for (i=0; i<(*cesSeq)._length; ++i) {
     aCES = (SCA_CF_Application_ComponentElementType)
                                                *((*cesSeq)._buffer+i);
     printf(" %s %s\",aCES.componentId,aCES.elementId);
   }

```

Figure 18.3. Query Application function

```
25.  cpidSeq = SCA_CF_Application__get_componentProcessIds(anApp,&ev);
26.  printf(" There are %d component process ID's\n",
                                              (*cpidSeq)._length);
27.  for (i=0; i<(*cpidSeq)._length; ++i) {
28.   aCPID = (SCA_CF_Application_ComponentProcessIdType)
                                                *((*cpidSeq)._buffer+i);
29.   printf(" %s %d\",aCPID.componentId,aCPID.processId);
30.  }
31.
32.  dasSeq = SCA_CF_Application__get_componentDevices(anApp,&ev);
33.  printf(" There are %d device assignments\n",(*dasSeq)._length);
34.  for (i=0; i<(*dasSeq)._length; ++i) {
35.   aDAS = (SCA_CF_DeviceAssignmentType) *((*dasSeq)._buffer+i);
36.   printf("  %s %s\",aDAS.componentId,aDAS.assignedDeviceId);
37.  }
38.
39.  cesSeq =
    SCA_CF_Application__get_componentImplementations(anApp,&ev);
40.  printf(" There are %d component implementations\n",
                                                (*cesSeq)._length);
41.  for (i=0; i<(*cesSeq)._length; ++i) {
42.   aCES = (SCA_CF_Application_ComponentElementType)
                                                *((*cesSeq)._buffer+i);
43.   printf(" %s %s\n",aCES.componentId,aCES.elementId);
44.  }
45.
46.  aString = SCA_CF_Application__get_profile(anApp, &ev);
47.   printf(" profile: %s\n",aString);
48.  aString = SCA_CF_Application__get_name(anApp, &ev);
49.   printf(" name: %s\n",aString);
50.
51.  /* Now the attributes inherited from Resource */
52.  aString = SCA_CF_Resource__get_identifier(anApp, &ev);
53.   printf(" Resource_id: %s\n",aString);
54.
55. }
```

Figure 18.3. (Continued)

The code segment to query an Application object is somewhat more complicated than that of the application factory. Most of this is due to the unique data structures embedded in the *Application* IDL. Variables for these data structures are found in lines 6, 9, and 12. They are data structures for `ComponentElementType, ComponentProcessIdType`, and `DeviceAssignmentType`. As is very common in the Core Framework IDL we encounter unbounded sequences of these data structures. These three data structures are used to hold an Application's attributes for four reasons:

1. a list of components that are registered with the Name Service;
2. a list of process identifiers;

3. a list of devices used by the application;
4. a list of implementation SPDs used to define each component.

It is important to recall the difference between instantiations of a component and the component implementation. A single implementation, as identified by an SPD, can be instantiated multiple times. There can also be multiple implementations identified in the SPD. For instance there might be one implementation for an x86 and another for a PPC. The SAD will provide UUIDs that uniquely identify each and every instance and each and every implementation.

Pointers to CORBA unbounded sequences are defined in lines 5, 8, and 11. The data structures themselves are quite simple: a couple of `CORBA_string` components and a `CORBA_unsigned_long` component for *processId*. Lines 18, 25, 32, and 39 use `'__get()'` methods to retrieve pointers to the four unbounded sequences of structures. For brevity's sake calls to `CORBA_free()` are left out of the code example so be advised that, as printed, this code segment will leak memory like a sieve. With a pointer to an unbounded sequence it is simple to access the `_length` of the sequence; lines 19, 26, 33, and 40. Finally, subsequent code blocks iterate the sequence, retrieve individual structured variables, and print the contents. The output produced for our running AudioEffect application is given in Figure 18.4.

```
There are 3 NamingContexts
  DCE:Echo[0] /SCARI_DM/a2/EchoResource_DCE:Echo[0]
  DCE:Chorus[0] /SCARI_DM/a2/ChorusResource_DCE:Chorus[0]
  DCE:Controller[0] /SCARI_DM/a2/AudioEffectController_DCE:
       Controller[0]
There are 3 component process IDs
  DCE:Echo[0] 1
  DCE:Chorus[0] 2
  DCE:Controller[0] 3
There are 3 device assignments
  DCE:Echo[0] DCE:node1_executableDevice[0]
  DCE:Chorus[0] DCE:node1_executableDevice[0]
  DCE:Controller[0] DCE:node1_executableDevice[0]
There are 3 component implementations
  DCE:Echo[0] DCE:AudioResource_x86impl
  DCE:Chorus[0] DCE:AudioResource_x86impl
  DCE:Controller[0] DCE:EffectController_x86impl
profile:<"/AudioEffect0/AudioEffectApplication0.sad.xml" type="DMD"/>
name: a2
Resource_id: AudioEffect0_DCE:SAD_softwareassembly_id
```

Figure 18.4. Application Attributes output

Again we've replaced the un-interesting UUIDs with something a little more readable: The first unbounded sequence is `componentNamingContexts`. This is essentially a mapping between each uniquely identified waveform component and its full Name Service name.

The nomenclature used to indicate an instantiation UUID is, for example, `DCE:Echo[0]`. Other instantiations of the Echo *Resource* might be `DCE:Echo[1], DCE:Echo[2]`, etc. The SCARI Application Manager queries the user to provide a name for the application. In this example we called the waveform instantiation ‘a2’. This name appears as part of the ‘path’ for the Name Service name. The reader will note that the contents of the `componentNamingContexts` is exactly the same as was discovered by traversing the name tree in Section 17.1. In that previous example we called the waveform instantiation ‘AudioApp1’. The mapping offered by `componentNamingContexts` is somewhat pointless because

1. the component UUIDs, and more importantly their object references, are already available through the name Service;
2. the mapping offered is only that of an instantiation UUID to that same UUID pre-pended by Naming Context ‘path’ information;
3. all the components associated with an application are already located in a separate naming context so it’s easy to tell which components are associated with which applications.

The next unbounded sequence available on the *Application* object is `componentProcessIds`. This unbounded sequence maps application component UUIDs to their operating system process Ids. Our output from the SCARI Core Framework shows processIds that number 1 through 3. These are evidently not the process Ids assigned by the operating system. Normally we would expect this sequence to map component UUIDs to operating system process Ids. A subtle guidance could be inferred here in that the granularity of components could be that of an operating system process. This is kind of at odds with an SCA POSIX profile that is thread, not process, oriented. Ultimately the SCA does not mandate that every process (or thread) should have a uniquely identifiable, one-to-one mapping between component identifiers and process Ids. In the pathological case an application, perhaps with many components, would be identified with a single SCA component and process Id. This notion of a single visible process is re-enforced by the fact that threads are not provided with a unique process number by the operating system. A multi-threaded application would normally have a single process Id. A possible use for `componentProcessIds` would be to attach a debugger or examine file system usage, IP ports, etc.

The next unbounded sequence available on the *Application* object is `componentDevices`. This unbounded sequence associates an application component (instantiation UUID) to a device. The device or rather instantiation of a device is identified in the DCD XML which identifies all devices reporting to a particular device manager. The devices are exactly identified within a DCD by the componentinstantiation element and Id attribute. In our example each application component is associated with an instance of a generic ExecutableDevice. In the SCARI example the ExecutableDevice can be traced through the Node1 DCD to its Device Package Descriptor (DPD) which identifies the device as a GPP.

The final unbounded sequence maps instances of application components to their implementations. We see that the Echo and Chorus resources are both mapped to the x86 implementation of an Audio resource. Examination of the XML indicates otherwise. Implementation UUIDs are found in a component’s SPD file. If we compare AudioEchoResource.spd.xml and AudioChorusResource.spd.xml we see that the implementations are associated with separate 'jar' (Java archive) files. Since they are from separate implementation files they should identified by separate UUIDs. This disconnect is probably due to the UUID being cut and pasted from one SPD to the other. In the case of the EffectController component there is a separately identified UUID given for a separate implementation file – AudioEffectController.jar.

The three final retrievable pieces of information on an *Application* object are the relative pathname of the SAD file, an arbitrary name issued to the application instance, and finally the UUID corresponding to the softwareassembly element, Id attribute taken from the SAD. The utility of all these *Application* attributes will be summarized at the end of the chapter.

18.3 DeviceManager Attributes

The next code segment (Figure 18.5) accepts an object reference to a *DeviceManager* object and then retrieves all attributes on the instance.

```
void queryDevMgr(SCA_CF_DeviceManager aDevMgr) {
  CORBA_Environment ev;
  CORBA_string aString;
  SCA_CF_FileSystem aFileSys;

  /* just a pointer to sequences of CORBA_Objects */
  SCA_CF_DeviceSequence* pSeq;
  SCA_CF_Device aDevice;

  /* a pointer to sequences of ServiceTypes */
  SCA_CF_DeviceManager_ServiceSequence* sSeq;
  SCA_CF_DeviceManager_ServiceType aService; /* Obj serviceObject */
                                   /* CORBA_string serviceName; */
  int i;
  CORBA_exception_init(&ev);

  aString =
SCA_CF_DeviceManager__get_deviceConfigurationProfile(aDevMgr,&ev);
  printf(" DeviceManager configProfile: %s\n",aString);

  aFileSys = SCA_CF_DeviceManager__get_fileSys(aDevMgr,&ev);
  if (CORBA_Object_is_nil(aFileSys,&ev))
   printf(" FileSystem is NIL\n");
  else printf(" FileSystem has valid IOR\n");

  aString = SCA_CF_DeviceManager__get_identifier(aDevMgr,&ev);
  printf(" DeviceManager ident: %s\n",aString);
```

Figure 18.5. Query Device Manager function

```
27.
28.  aString = SCA_CF_DeviceManager__get_label(aDevMgr,&ev);
29.  printf(" DeviceManager label: %s\n",aString);
30.
31.  pSeq = SCA_CF_DeviceManager__get_registeredDevices(aDevMgr,&ev);
32.  printf(" There are %d registeredDevices\n",(*pSeq)._length);
33.  for (i=0; i<(*pSeq)._length; ++i) {
34.   aDevice = (SCA_CF_Device) *((*pSeq)._buffer+i);
35.   queryDevice(aDevice);
36.  }
37.
38.  sSeq = SCA_CF_DeviceManager__get_registeredServices(aDevMgr,&ev);
39.  printf(" There are %d registeredServices\n",(*sSeq)._length);
40.  for (i=0; i<(*sSeq)._length; ++i) {
41.    aService = (SCA_CF_DeviceManager_ServiceType)
                  *((*sSeq)._buffer+i);
42.   printf(" %s\n",aService.serviceName);
43.   if (CORBA_Object_is_nil(aService.serviceObject,&ev))
44.       printf(" Service is NIL\n");
45.   else printf("  Service has valid IOR\n");
46.  }
47. }
```

Figure 18.5. (Continued)

Line 17 retrieves the filename of the DCD. This element of the domain profile contains XML data describing all *Device* objects started by this device manager. Line 20 retrieves the object reference to the *FileSystem* object running under this device manager. A device manager can be associated with only ONE *FileSystem*, however, many device managers could be setup to refer to the same *FileSystem*. Lines 25 through 29 retrieve and print the contents of a couple of string variables. The DeviceManager's identifier (UUID) is the *deviceconfiguration* element, *Id* attribute from the DCD. The DeviceManager's label is a 'user-friendly' appellation provided by the deviceconfiguration element, name attribute from the DCD.

Finally, in lines 31 through 45 we get to the meat of a DeviceManager object and that is object references to all of its registered *Device*s and Service objects. Please note that Service is not italicized. This is done specifically to indicate that a Service object is not a defined Core Framework interface. Unlike a Core Framework *Device* which has defined *start()* and *stop()* methods, etc., from the perspective of the SCA, a Service object is simply a CORBA object with unknown attributes and methods. All a *DeviceManager* gives us is a string name and an object reference. The output given in Figure 18.6 is produced by querying the SCARI Node1 device manager.

Not a lot of information is found in the DeviceManager itself. The real value is in obtaining object references to the devices themselves. The next section follows each registered *Device*'s object reference down into the implementation to see what attributes are available on it.

```
DeviceManager configProfile: <profile filename="/Node1.dcd.xml"
   type="DCD"/>
FileSystem has valid IOR
DeviceManager ident: DCE:18dd6458-494e-43cd-b823-07778bb9ca51
DeviceManager label: node1DeviceManager
There are 3 registeredDevices
There are 1 registeredServices
  node1Logger1 Service has valid IOR
```

Figure 18.6. Device Manager Attribute output

18.4 Device Attributes

The code segment of Figure 18.7 accepts an object reference to a *Device* object and then retrieves all attributes on the instance.

```
static char* usageStateString[ ] = {
  "IDLE",
  "ACTIVE",
  "BUSY"
};
static char* adminString[ ] = {
  "LOCKED",
  "SHUTTING_DOWN",
  "UNLOCKED"
};
static char* stateString[ ] = {
  "ENABLED",
  "DISABLED"
};

void queryDevice(SCA_CF_Device aDev, int nested) {

  CORBA_Environment ev;
  CORBA_string aString,bString,cString;
  int aUsageNum,anAdminNum,aState;
  CORBA_Object anObj;

  CORBA_exception_init(&ev);

  aString = SCA_CF_Device__get_softwareProfile(aDev,&ev);
  bString = SCA_CF_Device__get_label(aDev,&ev);
  cString = SCA_CF_Device__get_identifier(aDev,&ev); /* inherited */
  printf("SPD = %s\n",aString);
```

Figure 18.7. Device Query function

```
29.  printf("label = %s\n",bString);
30.  printf("ident = %s\n",cString);
31.
32.  aUsageNum = SCA_CF_Device__get_usageState(aDev,&ev);
33.  anAdminNum = SCA_CF_Device__get_adminState(aDev,&ev);
34.  aState = SCA_CF_Device__get_operationalState(aDev,&ev);
35.
36.  printf("      %s %s %s \n", usageStateString[aUsageNum],
37.       adminString[anAdminNum], stateString[aState]);
38.
39.  anObj = (SCA_CF_AggregateDevice)
40.    SCA_CF_Device__get_compositeDevice(aDev,&ev);
41.  if (CORBA_Object_is_nil(anObj,&ev))
42.   printf("compositeDevice is NIL\n");
43.  else {
44.   printf("Found a compositeDevice\n");
45.   if (!nested)
46.     queryAggDev(anObj);
47.  }
48. }
```

Figure 18.7. Device Query function

The first attributes are the now familiar SPD filename, UUID, and label (lines 25–30). All of these parameters come from the DCD part of the Domain Profile. The SPD filename is the *localfile* element, *name* attribute. The UUID and label come from the *componentinstantiation* element, *Id* and *usagename* attributes, respectively.

The next attribute available on any *Device* is its state information. There are three independent states available on each and every Device object: adminState, usageState, and operationalState. For two of the three devices, AudioDevice and RFDevice, in the SCARI example the state information is as follows:

usageState = IDLE, adminState = UNLOCKED, operationalState = ENABLED

For the node1ExecutableDevice (the GPP) the usageState is ACTIVE. Lines 32–36 from the code example provide this output. The enumerated types are mapped unto string variables at the head of the code example (lines 1–14). Finally, each *Device* can in fact be part of an aggregation of *Device*s. This is discernable in the run-time by examining the *compositeDevice* attribute. Running the `__get()` method on the *compositeDevice* attribute returns an object reference to an *AggregateDevice* if, in fact, the *Device* is part of an aggregation. If the *Device* is not associated with other devices, a NIL pointer is returned. Before examining the output from our SCARI example, we provide a code example in the next section that retrieves attributes on an AggregateDevice.

18.5 AggregateDevice Attributes

The only retrievable attribute on an AggregateDevice object is an unbounded sequence of pointers to *Device*s. The code given in Figure 18.8 will retrieve those Devicess one by one and call the `queryDevice()` routine on each.

```
1. void queryAggDev(SCA_CF_AggregateDevice anAggDev) {
2.
3.   SCA_CF_DeviceSequence* pSeq;
4.   SCA_CF_Device aDevice;
5.   CORBA_Environment ev;
6.   int i;
7.   CORBA_exception_init(&ev);
8.
9.   pSeq = SCA_CF_AggregateDevice__get_devices(anAggDev,&ev);
10.   printf("      There are %d members of the
        AggregateDevice\n",(*pSeq)._length);
11.  for (i=0; i<(*pSeq)._length; ++i) {
12.    aDevice = (SCA_CF_Device) *((*pSeq)._buffer+i);
13.    queryDevice(aDevice,1);
14.   }
15.
16. }
17.
```

Figure 18.8. Aggregate Device Query function

In working with the Core Framework we've had lots of examples of unbounded sequences so the code in lines 10–12 should be quite familiar. The output of Figure 18.9 is generated by running `queryDevice()` with calls to `queryAggDev()` as necessary:

The Node1 device manager lists three registered devices: node1ExecutableDevice, node1RFDevice, and node1AudioDevice. Examining this output yields a few notable observations. The only *Device* mapped to our running audio application is the node1ExecutableDevice. The UUID of this device is cross-referenced from the *componentDevices* attribute of our Audio *Application* object. In fact the audio application consists of three components: `Echo`, `Chorus`, and `Controller` all of which are associated with the node1ExecutableDevice which is a GPP. There is nothing available in the run-time domain that expresses a relationship between the audio application and the audio device. The only place to find this relationship is in the SAD. Upon examining the AudioEffect0 SAD, the relationship between audio application components and the audio device is found in the form of 'uses' and 'provides' ports. Unfortunately, there can be nothing garnered from the run-time with respect to port connections.

The last of the three *Device*s listed is the node1AudioDevice. When the `__get()` method is invoked on the *compositeDevice* attribute a non-nil object reference to an *AggregateDevice* object is returned. This object reference is then passed to the `queryAggDev()` function which interrogates the unbounded sequence of constituent *Device*s (line 10) to find that there

```
SPD = <profile filename="/ExecutableDevice.spd.xml" type="SPD"/>
label = node1ExecutableDevice
ident = DCE: node1_executableDevice[0]
    ACTIVE UNLOCKED ENABLED
compositeDevice is NIL

SPD = <profile filename="/RFDeviceImpl.spd.xml" type="SPD"/>
label = node1RFDeviceImpl
ident = DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888
  IDLE UNLOCKED ENABLED
compositeDevice is NIL

SPD = <profile filename="/AudioDevice.spd.xml" type="SPD"/>
label = node1AudioDevice
ident = DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB
  IDLE UNLOCKED ENABLED
Found a compositeDevice
  There are 1 members of the AggregateDevice
  SPD = <profile filename="/AudioDevice.spd.xml" type="SPD"/>
label = node1AudioDevice
ident = DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB
  IDLE UNLOCKED ENABLED
Found a compositeDevice
```

Figure 18.9. Device Attributes output

is one device in the aggregation. For each *Device* in the sequence the `queryDevice()` method is invoked to discern the attributes of the *Device*. By definition any *Device* which has a non-nil *AggregateDevice* cross-reference will be listed in the sequence of *Device*s within the AggregateDevice object. That *Device* will have a non-nil *AggregateDevice* object reference which calls the `queryAggDev()` method and so on and so forth *ad infinitum*. To break the recursion we have a 'nesting' parameter passed into the `queryDevice()` function which prevents a nested call to `queryAggDev()`.

18.6 DomainManager Attributes

The entry point into the domain is the object reference to the Domain Manager. This object reference is resolved through the Name Service. Depending on how one interprets SCA specifications section 3.1.3.2.3.5, there is no standardized name for the Domain Manager. In our SCARI example the Name Service name is 'DomainManager' which is found in the Name Service context `'SCARI_DM'`. If one interprets specification section 3.1.3.2.3.5 literally then domain managers everywhere are called 'DomainManager' within the context 'DomainName'. As always the Name Service `name.kind` component shall be set to the null string – ''.

The complete specification of the Domain Manager's Name Service name can always be found in any Device Manager's DCD. The element `domainmanager`, sub-element `namingservice`, attribute `name` is a #REQUIRED character data field in all DCDs.

Once the object reference to the *DomainManager* object is resolved the code segment in Figure 18.10 will query all of its attributes.

```
main( int argc, char * argv[] )
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CosNaming_NamingContext myNC=CORBA_OBJECT_NIL;
  CosNaming_Name myBindingName;

  SCA_CF_DomainManager dmObj=CORBA_OBJECT_NIL;
  CORBA_string aString;

   /* this is just a pointer to a sequence of CORBA_Objects */
   SCA_CF_DomainManager_DeviceManagerSequence *pSeq;

   /* these are just CORBA_Objects */
   SCA_CF_DeviceManager aDevMgr;
   SCA_CF_ApplicationFactory anAppFact;
   SCA_CF_Application anApp;
   CosNaming_NameComponent
        path[2]={{"SCARI_DM",""},{"DomainManager",""}};
   int i;

   CORBA_exception_init(&ev);
   startOrb(&argc,argv,&orb,&ev);

   myNC = CORBA_ORB_resolve_initial_references(
                        orb,"NameService",&ev);

   myBindingName._length=2;
   myBindingName._buffer= path;

   dmObj=(SCA_CF_DomainManager)
             CosNaming_NamingContext_resolve(myNC,
                  (const CosNaming_Name *)&myBindingName, &ev);

   aString = SCA_CF_DomainManager__get_identifier(dmObj,&ev);
   printf("\nDomain Manager ident: %s\n",aString);

   aString =
    SCA_CF_DomainManager__get_domainManagerProfile(dmObj,&ev);
   printf("Domain Manager profile: %s\n",aString);

   pSeq = SCA_CF_DomainManager__get_deviceManagers(dmObj, &ev);
```

Figure 18.10. Domain Manager Query function

```
41.   printf("\nThere are %d deviceManagers\n",(*pSeq)._length);
42.   aDevMgr = (SCA_CF_DeviceManager) *((*pSeq)._buffer+0);
43.   queryDevMgr(aDevMgr);
44.
45.   pSeq = SCA_CF_DomainManager__get_applicationFactories(dmObj, &ev);
46.   printf("\nThere are %d applicationFactories\n",(*pSeq)._length);
47.   for (i=0; i<(*pSeq)._length; ++i) {
48.    anAppFact = (SCA_CF_ApplicationFactory) *((*pSeq)._buffer+i);
49.    queryAppFact(anAppFact);
50.   }
51.
52.   pSeq = SCA_CF_DomainManager__get_applications(dmObj, &ev);
53.   printf("There are %d applications\n",(*pSeq)._length);
54.   for (i=0; i<(*pSeq)._length; ++i) {
55.    anApp = (SCA_CF_Application) *((*pSeq)._buffer+0);
56.    queryApp(anApp);
57.   }
58.
59.   anApp = SCA_CF_DomainManager__get_fileMgr(dmObj, &ev);
60.   if (CORBA_Object_is_nil(anApp,&ev))
61.    printf("\nFileManager is NIL\n");
62.   else printf("\nFileManager has valid IOR\n");
63.
64.   return 0;
65. }
```

Again the code example leaks memory by neglecting to free strings allocated by the many calls to `__get()` methods that return a pointer to strings. After resolving the initial context of the Name Service – its corbaloc or IOR must be passed on the command line – the Domain Manager object reference is resolved in line 30. There are two string field attributes associated with the *DomainManager* object; line 34 queries the UUID and line 37 the filename of the DomainManager's DMD. The remainder of the code traverses the following unbounded sequences: i) device managers; ii) application factories; and iii) applications. The final attribute on the *DomainManager* object is, line 59, an object reference to the *FileManager* object. Within a 'domain' the SCA specifies a cardinality of one for the Domain Manager, one for the FileManager, and at least one for DeviceManager. All other elements are optional. Running the program for our SCARI example yields the output of Figure 18.11.

The output generated by nested calls to `queryApp()`, `queryAppFact()` and `queryDevMgr()` is suppressed in the output having already appeared in the previous sub-sections.

18.7 Properties

Of significant importance within our run-time domain is the presence of the *PropertySet* interface which can be queried and configured. The PropertySet interface is inherited by

```
Domain Manager ident: DCE:305CCD21-C9S9-4738-9C81-BA0B29745CEW
Domain Manager profile:<profile
filename="/DomainManager.dmd.xml"type="DMD"/>
There are 1 deviceManagers
There are 2 applicationFactories
There are 1 applications
FileManager has valid IOR
```

Figure 18.11. Domain Manager Attributes output

Resource, DeviceManager, and *DomainManager*. Since *Resource* is inherited by the *Device* interface and *Application* interface every object reference available in the run-time with the exception of *ApplicationFactory* objects has properties that can be queried and/or configured. Present initial values of properties can be ascertained from a component's PRF file but this necessarily involves fishing through a lot of XML. It's easier to get property settings from the run-time.

The code segment (Figure 18.12) examines the Domain Manager, three Devices, and three (application) resources found when running the Audio application in our SCARI example.

For the sake of a simple illustration, the code example in Figure 18.12 does a lot of cheating. For instance all the Name Service names are hard-coded. Repeated runs of the

```
static CosNaming_NameComponent allObjs[7][3] = {
 { {"SCARI_DM",""},{"DomainManager",""},{"",""} },
 { {"node1DeviceManager_DCE:18dd6458-494e-43cd-b823-
 07778bb9ca51",""},{"",""},{"",""} },
 { {"node1RFDeviceImpl_DCE:B6C3F70D-A069-47B5-BCEA-
 708E51C08888",""},{"",""},{"",""} },
 { {"node1AudioDevice_DCE:A68A5812-6BE7-4920-9A29-
 A7C013734FAB",""},{"",""},{"",""} },
 { {"SCARI_DM",""},{"tuesdayAfternoon",""},
          {"AudioEffectController_DCE:7F19A71E-DE41-4BEF-A619-
 9EE5ECCD832C",""} },
 { {"SCARI_DM",""},{"tuesdayAfternoon",""},
          {"EchoResource_DCE:916B1F9F-25BA-43A6-896E-
 5B078D12B727",""} },
 { {"SCARI_DM",""},{"tuesdayAfternoon",""},
          {"ChorusResource_DCE:8FC4B3CF-5203-4874-98D6-
 0FF6E4F2ED5C",""} }
};

int main( int argc, char * argv[] )
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CosNaming_NamingContext myNC=CORBA_OBJECT_NIL;
```

Figure 18.12. Property Set Query function

```
CosNaming_Name myBindingName;
CosNaming_NameComponent myNameComponent;

SCA_CF_Properties propHolder; /* a sequence of DataTypes */
SCA_CF_DataType anElement;
CORBA_Object anObj=CORBA_OBJECT_NIL;
CORBA_TCKind tk;
int i,j;

CORBA_exception_init(&ev);
startOrb(&argc,argv,&orb,&ev);

myNC = CORBA_ORB_resolve_initial_references(
                            orb,"NameService",&ev);

for (j=0; j<7; ++j)
{
 if (j==0) myBindingName._length=2;
 if (j>=1) myBindingName._length=1;
 if (j>=4) myBindingName._length=3;
 myBindingName._buffer= &(allObjs[j][0]);

 anObj=(CORBA_Object)CosNaming_NamingContext_resolve(myNC,
          (const CosNaming_Name *)&myBindingName, &ev);

 propHolder._length=0; /* signal you want all the properties */
 SCA_CF_PropertySet_query((SCA_CF_PropertySet)anObj,
                                     &propHolder, &ev);
myNameComponent =*(myBindingName._buffer+myBindingName._ length-1);
 printf("\n%s Properties\n",myNameComponent.id);

 printf("There are %d elements\n",propHolder._length);
 for (i=0; i<propHolder._length; ++i)
 {
   anElement = *(propHolder._buffer+i);
   tk = CORBA_TypeCode_kind( anElement.value._type,&ev );
   if
((tk==CORBA_tk_struct)||(tk==CORBA_tk_enum)||(tk==CORBA_tk_alias))
          printf("%d %s %s\n", i,anElement.id,
                CORBA_TypeCode_name( anElement.value._type,&ev ) );
   else
   {
          printf("%d %s %s =", i,anElement.id, tkString[tk] );
```

Figure 18.12. (Continued)

SCARI AudioEffect0 application allow us access to componentinstantiation UUIDs so, for the sake of iterating all seven objects that happen to have properties, it is made easier if we

```
61.         if ( tk == CORBA_tk_long )
62.             printf(" %d\n",*(CORBA_long*)anElement.value._value);
63.         if ( tk == CORBA_tk_double )
64.             printf(" %g\n",
                     *(CORBA_double*)anElement.value._value);
65.     }
66.   }
67.
68.  }
69.  return 0;
70. }
```

stick the Name Service names in to an array (see lines 1 through 11). The 'friendly' name given to the application by the user at run-time also happens to be part of the naming context traversed on the way to resolving application components. In previous SCARI example 'runs' we've called the application 'myAudioApp' or 'a2', etc. For this execution example we called the application 'tuesdayAfternoon'.

The code example loops through the seven objects within the domain that have inherited the *PropertySet* interface (line 34). After resolving the object reference, line 41, a call is made to the objects `CF::PropertySet::query()` method. A nice feature of SCA-compliant systems is that if you set the length parameter to zero in the `_query()` call all properties on that object are returned. As opposed to a return value, the caller must provide a pointer to a location for the `_query()` operation to put an unbounded sequence of *DataTypes*. This is one of the three times that the Core Framework makes use of inout parameters. The other two are the query method on a *FileSystem* and to hold the test results of a *TestableObject*.

So the call to `_query()` takes an inout parameter called Properties which is really an unbounded sequence of DataType. DataType is a structure having two fields, a *string* Id and an *any* value. The IDL for these constructs is as follows:

```
struct DataType {
   string id;
   any value;
};
typedef sequence <DataType> Properties;
```

This is our second encounter with CORBA's *any* type which leads to the second form of cheating in our code example. By repeated runs of our code example we can know the types of properties to expect. Line 55 checks for structures or enumerated types and just prints out the name of the structure or enumeration. If the *any* `_type` is not one of these then we know it is either a *long* or a *double* (lines 61 and 63). In reality, code that can de-cipher any *any* `_type` would be quite complex – imagine a structure containing unbounded sequences of structures. Our simple code example shows the heart of de-ciphering the *any* `_ type` and that is using the `_type` information to make the proper type casts of the de-referenced void pointer `_value` (lines 62 and 64). The output in Figure 18.13 is produced with the AudioEffect0 application running.

The SCARI application manager GUI allows the user to modify certain parameters while the application is stopped. Running the PropertySet query program again properly reflects the modified values.

```
DomainManager Properties
There are 1 elements
0 PRODUCER_LOG_LEVEL LogLevelSequenceType

node1DeviceManager_DCE:18dd6458-494e-43cd-b823-07778bb9ca51 Properties
There are 1 elements
0 PRODUCER_LOG_LEVEL LogLevelSequenceType

node1RFDeviceImpl_DCE:B6C3F70D-A069-47B5-BCEA-708E51C08888 Properties
There are 3 elements
0 PRODUCER_LOG_LEVEL LogLevelSequenceType
1 USAGE CORBA_tk_long = 1
2 RF_MODE CORBA_tk_long = 1

node1AudioDevice_DCE:A68A5812-6BE7-4920-9A29-A7C013734FAB Properties
There are 8 elements
0 AUDIO_IN_RIGHT CORBA_tk_long = 1
1 AUDIO_IN_LEFT CORBA_tk_long = 1
2 PRODUCER_LOG_LEVEL LogLevelSequenceType
3 AUDIO_OUT_LEFT CORBA_tk_long = 4
4 AUDIO_BUFFER_SIZE CORBA_tk_long = 8192
5 AUDIO_PROFILE CORBA_tk_long = 0
6 AUDIO_OUT_RIGHT CORBA_tk_long = 4
7 AUDIO_MODE CORBA_tk_long = 6

AudioEffectController_DCE:7F19A71E-DE41-4BEF-A619-9EE5ECCD832C
  Properties
There are 4 elements
0 PRODUCER_LOG_LEVEL LogLevelSequenceType
1 ECHO_DELAY CORBA_tk_double = 0.5
2 NUMBER_OF_VOICES CORBA_tk_long = 5
3 ECHO_GAIN CORBA_tk_double = 0.5
EchoResource_DCE:916B1F9F-25BA-43A6-896E-5B078D12B727 Properties
There are 5 elements
0 SAMPLING_FREQUENCY CORBA_tk_double = 0
1 PRODUCER_LOG_LEVEL LogLevelSequenceType
2 OUTPUT_GAIN CORBA_tk_double = 0
3 ECHO_DELAY CORBA_tk_double = 0.5
4 ECHO_GAIN CORBA_tk_double = 0.5

ChorusResource_DCE:8FC4B3CF-5203-4874-98D6-0FF6E4F2ED5C Properties
There are 3 elements
0 PRODUCER_LOG_LEVEL LogLevelSequenceType
1 OUTPUT_GAIN CORBA_tk_double = 0
2 NUMBER_OF_VOICES CORBA_tk_long = 5
```

Figure 18.13. Resource Properties output

18.8 Manipulating Ports

The port concept enabled by the SCA is quite powerful and flexible. The designers of the SCA wanted users to have maximum flexibility and stopped short of dictating actual push or pull semantics. The transport level details of a port are implementation-dependent. The only operations required by the SCA are a `getPort` operation, which returns a Port object and connect and disconnect operations on the Port object that accepts an object reference as a parameter. The mechanization of ports is simple. The Core Framework reads the port configuration from the SAD. Ports are paired as connections. With `getPort` the Core Framework retrieves object references for ports A and B. Port A provides or implements an interface, for example, `pushOctet`. The Core Framework then calls the connect interface on port B and passes to it the object reference for A. Port B is now fully able to invoke `pushOctet` operations on port A. In this sense the Core Framework acts as a broker.

The power of this connection scenario is that it is script driven. A user can rewire port connections by simply changing the XML within the SAD. Figure 18.14 shows the port configuration for the SCARI Audio Effect application. The Audio device performs `pushPacket` operations on the Echo resource. The Echo resource in turn performs a `pushPacket` operation on the Chorus resource. Finally the Chorus resource performs a `pushPacket` on the Audio Device. The reader is encouraged to rewire this scenario by modifying the SAD. For instance the echo can be removed from the processing chain by changing the `providesname`, line 63 of the AudioEffectApplication0.sad.xml file, from Echo to Chorus. The Echo to Chorus connection can be deleted in its entirety. These modifications must be made in the $SCA_HOME/demosources/Waveforms/AudioEffect0 directory. The reader must then proceed to the $SCA_HOME/lib directory and 'make clean' and then 'make' the executables. The application will need to be re-installed through the `startApplicationManager` application. A simple way to uninstall an application is to remove its entry from the ApplicationFactory.conf file located in $SCA_HOME/demosources/Nodel/profile directory.

An even more radical modification would be to route the AudioOut port directly back in to the AudioIn port of the Audio Device. This exercise is quite instructive in that the concept of packet latency becomes very apparent. Even without the Echo and Chorus resources, there is a delay operation that takes place on the audio stream. This is a product of the design. What constitutes a packet? A tenth of a seconds worth of data, a quarter of a second? Sometimes legacy waveform latency requirements are hard to meet because of this packetization concept. Legacy radios didn't packetize they streamed – there is quite a difference. In a legacy radio when a bit was pulled off the air it was immediately clocked into the baseband terminal device. In a software radio the modem will gather up at least a byte, perhaps hundreds of bytes depending on data rate. The packet is transmitted over some kind of bus, e.g., PCI, VME, to another computer that then clocks it into the baseband terminal device. Because of the packetization there is going to be a latency there that wasn't present in the legacy radio.

18.9 Summary

Our code examples have shown that there is a lot of redundancy in the information that is held by the Core Framework. Several parameters that originate within the Domain Profile are

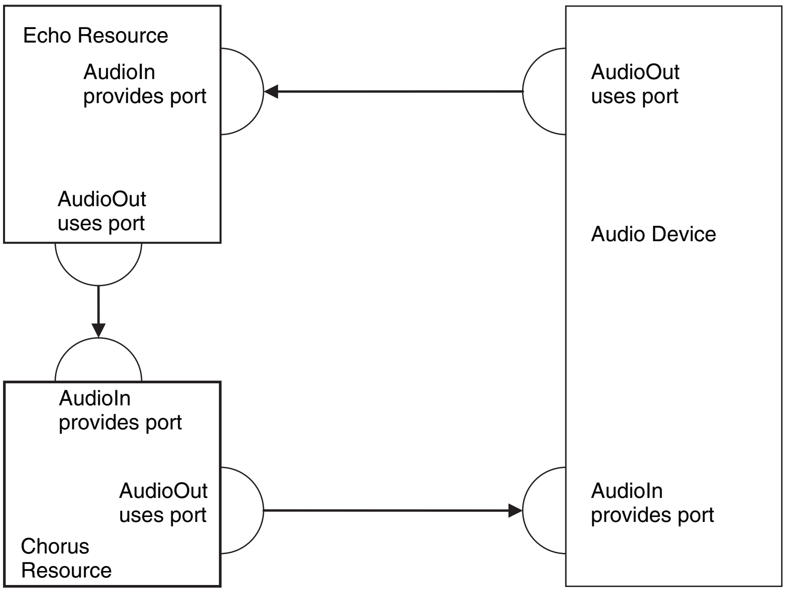


Figure 18.14. SCARI Audio Effect port connections

additionally published within the run-time domain's Core Framework objects. Much of this same data is published by all three CORBA services: Naming, Event, and Log. Of course the persistence of these publications is different. The Core Framework and Name Service provide a current snapshot. The Core Framework additionally provides a list of installed applications – that is given as a sequence of *ApplicationFactory*s. The Name Service lists application components currently instantiated but does not also list the factories. Hence it is impossible to create() an application without going through the Domain Manager to get an Application Factory's object reference.

The Log Service provides a historical record of events and identifications. The SCARI Log Service persists with the same lifetime as the DeviceManager. There is nothing to preclude a Log Service from persisting across power cycles thus providing a life-cycle record of events and identifications. Because of this persistence the Log Service provides an excellent means of catching the output of debug statements. The Domain Manager is required to log `registerDevice()` events as both the Log Service `ADMINISTRATIVE_EVENTS` and on the ODM event channel. If the same data is recorded on both then the log file will contain IORs for all registered devices. Similarly the IORs of application components are also logged when the components are created by an ApplicationFactory.

The shortest window of opportunity available is provided by the Event Service. A user will see events only when his or her IDM and/or ODM push consumer is registered with the DomainManager. This summary section cross-references which items of interest can be found in the domain profile, run-time domain, and the various services. Figure 18.15 provides a summary comparison of the run-time domain and the Name Service.

Figure 18.15 shows that all the object references available through the DomainManager object are also available through the Name Service with the exception of *ApplicationFactory*

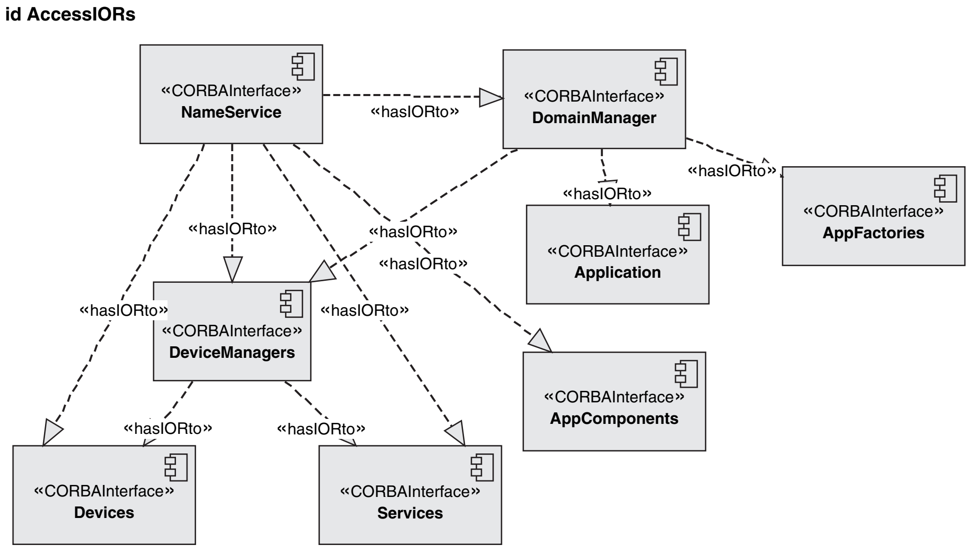

id AccessIORs
«CORBAInterface» NameService
«CORBAInterface» DomainManager
«CORBAInterface» DeviceManagers
«CORBAInterface» Application
«CORBAInterface» AppFactories
«CORBAInterface» AppComponents
«CORBAInterface» Devices
«CORBAInterface» Services
«hasIORto»

Figure 18.15. Accessing object references

objects. The only way to get to an Application Factory say, for example, to run the *create*() method is through the Domain Manager. We are reminded that Application Factories are installed on a domain by calling the *DomainManager::installApplication*() method. There is nothing that disallows the applications programmer from registering an ApplicationFactory with the Name Service. Also, the stringified IOR of an Application Factory is published on the ODM event channel. There are two ways of accessing object references of Devices: through the Naming Service or through a Device Manager. There is only one way to access individual components (*Resource* objects) within an application and that is – surprisingly – through the Name Service.

Access to all of these object references doesn't buy you much if you don't have access to the object's IDL or SCD. This is because SCA requires minimum CORBA which does not have support for the Dynamic Invocation Interface (DII). Thus there is no way to say to an object 'give me all your operations' or 'give me all your attributes'.

Throughout this chapter we have tried to show how parameters accessible in the run-time trace back in to the Domain Profile. Table 18.1 shows a complete summary of all Core Framework attributes accessible in the run-time traced back to their origin in the Domain Profile. Note that for certain attributes the SCA is not explicit. In these instances we provide, as a reference, the choices made by the SCARI implementation. Perhaps future versions of the SCA will nail down these loose ends.

Table 18.1. Core Framework Attributes cross-reference to Domain Profile

CF Object	Attribute	Domain Profile Cross-Reference
Resource	Identifier	SCA – unique identifier (instance) SCARI – DCD.componentinstantiation.id for a Device – SAD.softwareassembly.id for an Application – SAD.componentinstantiation.id for a Resource
DeviceManager	deviceConfigurationProfile	Profile.filename = DCD filename Profile.type (optional)
DeviceManager	Identifier	DCD.deviceconfiguration.id
DeviceManager	Label	DCD.deviceconfiguration.name (section 3.1.3.2.4.5)
DeviceManager	registeredServices.serviceName	SCA – not specified SCARI – DCD.usagename
ResourceFactory	Identifier	SCA – unique instance identifier SCARI – not implemented
Device	softwareProfile	Profile.filename = SPD filename Profile.type (optional)
Device	Label	SCA – meaningful name SCARI – DCD.usagename
DomainManager	domainManagerProfile	Profile.filename = DMD filename Profile.type (optional)
DomainManager	Identifier	DMD.domainmanagerconfiguration.id
Application	componentNamingContexts.componentId	SAD.componentinstantiation.id
Application	componentNamingContexts.elementId	fully-qualified NameService Name
Application	componentProcessIds.componentID	SAD.componentinstantiation.id
Application	componentDevices.componentId	SAD.componentinstantiation.id
Application	componentDevices.assignedDeviceId	DCD.componentinstantiation.id
Application	componentImplementations.componentId	SAD.componentinstantiation.id
Application	componentImplementations.elementId	SPD.implementation.id
Application	Profile	Profile.filename = SAD filename Profile.type (optional)
Application	Name	string passed to ApplicationFactory.create()
ApplicationFactory	Name	SCA – "type of Application" App D, D.6 - SAD.softwareassembly.name
ApplicationFactory	Identifier	SAD.softwareassembly.id
ApplicationFactory	softwareProfile	profile.filename = SAD filename profile.type (optional)

19

An SCA-compliant Application

This chapter starts with a simple non-SCA-compliant Hello World application – consider this a legacy application – which will then be modified to make it SCA-compliant. In order to maximize educational value, we adopt a minimalist approach to SCA-compliance. Recall that for good reason the SCA allows the developer a great deal of latitude in specifying component granularity. The current view of compliance testing is whether your application is 100 % compliant with the 'shall's of the SCA or not. If it is not, then the application becomes SCA-compliant with waivers. There is no graduation of scale – that is, a 9 or a 10 for an application built to the 'spirit' of the SCA versus a 1 or a 2 for an application that has a single CORBA object that manages the entire waveform. The granularity of an approach is typically dictated by the operational requirements of the radio system and not the SCA.

It is now generally recognized that there is little connection between SCA-compliance and portability. Legacy codes originally developed for radio 'X' will generally be very difficult to port to radio 'Y'. An SCA-wrapped version of those codes will still be as difficult to port – perhaps even more difficult on account of the OE. Not all ORBs are the same and neither are real-time OSs claiming to be POSIX-compliant actually POSIX-compliant. Hence, the porting operation invariably involves some re-write of application software to accommodate the new OE. (Another approach is to create an OE abstraction but that's a subject for another book.)

19.1 Hello World Legacy Application

The domain profile SPD's *softpkg* element contains an attribute called *type*. This attribute can be set to `sca_compliant` or `sca_non_compliant` with `sca_compliant` being the default value when not otherwise specified. As a starting point we'll just use the Core Framework to install our legacy application using `DomainManager::installApplication()`. In order to survive the various tests that are mandatorily performed by the installApplication routine we will have to generate a certain amount of XML to describe our application to the domain manager. At a minimum there has to be a SAD file because that filename is expected as a parameter in the call to `DomainManager::installApplication()`. The SAD file in turn requires a file reference to at least one SPD. It is inside this SPD that we are finally able to tell the domain manager that our application is not SCA-compliant. As an `sca_non_compliant` component the domain manager will not look for an SCD. So as far as installation goes the two domain profile files, SAD and SPD, are all that's needed. Thus, minimally, even a legacy `sca_non_compliant` application requires two XML files to describe itself to the domain manager and application factory.

We start by presenting the Hello World 'C' code and its corresponding XML. Then we need some sort of CORBA-based client software to invoke the `DomainManager::installApplication()` operation and the `ApplicationFactory::create()` operation. For many readers the Hello World application has been seen many times and Figure 19.1 shows it once again.

```
#include <stdio.h>
#include <time.h>

static const struct timespec oneSecond = {1,0};

int main() {

    while (1) {
        fprintf(stdout,"Hello World \ n");
        fflush(stdout);
        nanosleep(&oneSecond,NULL);
    }

}
```

Figure 19.1. Hello World – main

The program will simply print 'Hello World' to the standard output device once every second until the process is killed. It is necessary to compile this software and then put the executable in a place where the domain manager can find it. The SCARI software includes a GUI application that copies a Java archive (jar) file from where it was built to where the domain manager can find it. A jar file is Java's version of the Unix tar file or the Windows zip file. It allows one to combine all of the executable 'class' files and other required data files (XML) into one file that is easily copied and then extracted at the new location. After performing the copy and extract, the GUI then invokes the `DomainManager::installApplication()` operation to create an *ApplicationFactory* object. Since that GUI is hard-wired to work with jar files, we won't be able to use it to install our application.

As part of the `installApplication` operation the domain manger will look for a SAD, SPD, and the executable itself in the following location:

```
$SCA_HOME/demosources/Node1/profile/HelloWorld
```

The SCA does not specify pathname naming conventions so the pathnames offered in this example are specific to the SCARI Core Framework. Let's now examine a Document Type Definition (DTD) for a minimum, but still SCA-compliant, SAD.

We will walk through this code example by comparing it to the *softwareassembly* DTD and then to that minimal set required by the SCA – well at least SCARI interpretation of the SCA. We start by creating a minimal *softwareassembly* DTD by taking the full specification (SCA Appendix D.6) and eliminating any element or attribute that is not #REQUIRED or any element that has a cardinality that includes zero. The DTD given in Figure 19.2 represents that minimum.

```
<!ELEMENT softwareassembly
   ( componentfiles
   , partitioning
   , assemblycontroller )>

<!ATTLIST softwareassembly
   id ID #REQUIRED
   name CDATA #IMPLIED> <!- REQUIRED by SCARI -->

<!ELEMENT componentfiles
   ( componentfile+ )>

<!ELEMENT componentfile
   ( localfile )>

<!ATTLIST componentfile
   id ID #REQUIRED>

<!ELEMENT localfile EMPTY>

<!ATTLIST localfile
   name CDATA #REQUIRED>

<!-- DTD says that sub-elements of partitioning are optional -->
<!-- mandatory assmemblycontroller element requires a refid that -->
<!-- must match some componentinstantiation id declared here -->
<!-- therefore at least one sub-element must exist -->

<!ELEMENT partitioning
   ( componentplacement+ )> <!-- modified for single  processor -->

<!ELEMENT componentplacement
   ( componentfileref
   , componentinstantiation+ )>

<!ELEMENT componentfileref EMPTY>

<!ATTLIST componentfileref
```

Figure 19.2. Minimum software assembly descriptor

```
39.   refid CDATA #REQUIRED>
40.
41. <!ELEMENT componentinstantiation EMPTY>
42.
43. <!ATTLIST componentinstantiation
44.   id ID #REQUIRED>
45.
46. <!ELEMENT assemblycontroller
47.   ( componentinstantiationref )>
48.
49. <!ELEMENT componentinstantiationref EMPTY>
50.
51. <!ATTLIST componentinstantiationref
52.   refid CDATA #REQUIRED>
```

Figure 19.2. (Continued)

The DTD given is truly minimal – no ports and only one component instantiation. A brief walk through this DTD will provide context for our Hello World SAD to follow. The *softwareassembly* element must have an *Id* attribute (line 7). Additionally the *softwareassembly* element also has a *name* attribute (line 8). This is used to fill-in the `name` attribute of the ApplicationFactory interface. (SCA itself specifies only that the ApplicationFactory `name` attribute is a 'type of *Application*' but does not specify where it comes from. However paragraph D.6 of SCA Appendix D specifies that this *name* attribute is, in fact, the ApplicationFactory `name` attribute.) After this the softwareassembly is composed of three mandatory elements called *componentfiles*, *partitioning*, and *assemblycontroller*. The order of definition of these elements is as unambiguously stated in the DTD.

The *softwareassembly* next requires definition of a *componentfiles* element (line 10). The presence of this element is mandatory and its cardinality is unity. Now the *componentfiles* element must contain one or more *componentfile* elements (line 11). Each *componentfile* must have an *Id* attribute – this, of course refers to a UUID. Finally, each *componentfile* must have an empty *localfile* sub-element with a single *name* attribute. This *name* is the filename of the component's SPD.

For each of the components previously identified the next element – *partitioning* – describes where to place the components. However for the multi-processor case – consider multiple executable devices – SCA Appendix D does not allow the SAD, by itself, to identify what component goes on what processor. That operation is performed when the application is created. For the minimal SAD, the *partitioning* element does provide a very important UUID that is needed later by the mandatory *assemblycontroller* element. Since *hostcollocation* is somewhat meaningless for a single processor domain it has been removed from the minimal SAD. This now leaves one or more *componentplacement* elements (line 30). Each of these requires a *componentfileref.refid* and a *componentinstantiation.Id*. The *componentfileref.refid* (line 39) must match one of the *componentfile.id*s declared in the previous *componentfiles* section (line 17). Finally, the user provides a UUID for the *componentinstantiation.id* (line 44). One of these components so identified will be required later by the *assemblycontroller* element.

The final mandatory element is identification of the *assemblycontroller*. SCA specification section 3.1.3.2.1.5 requires one and only one point-of-contact between the *Application* interface and the implementation. So an application might have several components but only one – the assembly controller – overloads the *Resource* `start()`, `stop()`, `runTest()`, etc. interfaces. Anyhow the *assemblycontroller* element has a single required sub-element *componentinstatiationref* that has an *Id* attribute (line 52) that must match one of the *componentinstantiation.Id* attributes in the previous described *partitioning* section (line 44).

Our simple SAD makes no provision for connections – the SAD *connections* element is in fact optional in the DTD – but this will be required later as we build an SCA-compliant application.

It's apparent that even the simplest SAD file has element inter-dependencies that must be honored. It is sometimes necessary to go searching in the documentation for these required inter-dependencies. The best places to look are the SCA main document, Appendix D, the DTDs, and maybe even Appendix C IDL. Let's now examine the XML files required for installation. The minimal SAD file given in Figure 19.3 describes our Hello World software assembly.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!-- Manually edit for Wiley book -- Legacy HelloWorld example  -->
3. <!DOCTYPE softwareassembly SYSTEM  "dtd/softwareassembly.2.2.dtd">
4. <softwareassembly id="DCE:a8135383-4ee5-4400-b512-cb69ed4f21ee" name=
   "HelloWorld">
5.  <componentfiles>
6.      <!-- Must be at least 1 component,  "assemblycontroller" -->
7.      <componentfile id="HelloAssemCtrlFile" type="SPD">
8.          <localfile name="HelloAssemCtrl.spd.xml"/>
9.      </componentfile>
10. </componentfiles>
11. <partitioning>
12.     <!-- Must have componentplacement OR  hostcollocation-->
13.     <componentplacement>
14.         <!-- Must have componentfileref one -->
15.         <componentfileref refid="HelloAssemCtrlFile"/>
16.             <!-- Must have at least one componentinstantiation -->
17.             <componentinstantiation  id="DCE:1d9bbb8e-d6ac-4350-b3c6-
   8226ced4e0ef">
18.               </componentinstantiation>
19.             </componentplacement>
20. </partitioning>
21. <assemblycontroller>
22.     <!-- Must match componentinstantiation id of 1 of the  components -->
23.     <componentinstantiationref refid="DCE:1d9bbb8e-d6ac-4350-b3c6-
   8226ced4e0ef"/>
24.    </assemblycontroller>
25. </softwareassembly>
```

Figure 19.3. Hello World software assembly descriptor

Given the rather exhaustive treatment of the underlying minimal DTD it is necessary only to briefly point out the inter-dependencies required by the SCA. After the *softwareassembly* itself is *id*'d and *name*'d the Hello World SAD identifies one component (line 7). The component's SPD file name is given in line 8. This information will be parsed and the existence of the SPD file will be verified during the `install()` operation. The next required use of the SAD is to identify the assembly controller. Here is the dependency path: Line 15 relates *componentfileref* to lines 7 and 8 where the SPD is called out; the *componentinstantiation.Id* associated with line 15 is provided as the UUID of the mandatory assembly controller (line 23).

19.2 Legacy Hello World SPD

In this section we examine the DTD of a minimum SPD. Once again the DTD requires certain elements that are required by the SCA. Figure 19.4 identifies the minimal SPD with respect to the SCA's requirements.

```
<!ELEMENT softpkg
  ( author+
   ,implementation+
  )>

<!ATTLIST softpkg
  id ID #REQUIRED
  name CDATA #REQUIRED
  type (sca_compliant | sca_non_compliant)"sca_compliant"
  version CDATA #IMPLIED >

<!ELEMENT author EMPTY>

<!ELEMENT implementation
  ( code
  , runtime?
  , ( os | processor | dependency )+
  )> <!-- all three os, proc and depend are used -->

<!ATTLIST implementation
  id ID #REQUIRED>

<!ELEMENT code
  ( localfile )>

<!ELEMENT localfile EMPTY>

<!ATTLIST localfile
  name CDATA #REQUIRED>
```

Figure 19.4. Minimum Software Packager Descriptor (no SCD)

```
30.
31. <!ELEMENT os EMPTY>
32.
33. <!ATTLIST os
34.   name CDATA #REQUIRED>
35.
36. <!ELEMENT processor EMPTY>
37.
38. <!ATTLIST processor
39.   name CDATA #REQUIRED>
40.
41. <!ELEMENT dependency
42.   ( propertyref )>
43.
44. <!ATTLIST dependency
45.   type CDATA #REQUIRED> <!-- mips_allocation or
46.                             memory_allocation -->
47.
48. <!ELEMENT propertyref EMPTY>
49.
50. <!ATTLIST propertyref
51.  refid CDATA #REQUIRED
52.  value CDATA #REQUIRED> <!-- allocation property, UUID and value -->
53.
54. <!ELEMENT runtime EMPTY>
55.
56. <!ATTLIST runtime
57.   name CDATA #REQUIRED> <!-- required by SCARI -->
```

Figure 19.4. (Continued)

There are at least two elements required for a SCA non-compliant component. The strict significance of SCA non-compliance is the lack of an SCD. The component is likely to be not even CORBA-capable so most of the content within the SCD is not relevant. Despite being a minimal SPD there is still quite a bit of content required. A complete SPD first off requires an *author* element (line 2). The oddity is that the *author* element has no required sub-elements or attributes. The DTD requirement is satisfied by inserting an empty author element in to the XML (line 12). Finally, a complete SPD requires one or more *implementation* elements. Each implementation is identified by a combination of *os*, *processor*, and *dependency* elements (line 17). Literally the DTD specifies at least one of the three elements, but in practice all three are used to identify a *code* element uniquely which is the binary file image (lines 15, 23, and 28). Two allocation parameters figure prominently in components that are to be loaded and executed on an *ExecutableDevice*: `mips_allocation` and `memory_allocation` (line 45). The UUIDs for these dependency properties (line 51) must match the corresponding capacity properties given in the Properties File (PRF) of the *ExecutableDevice*. When the application is created these values are checked against the *ExecutableDevice*'s current capacity to determine if sufficient capacity exists on the *ExecutableDevice* to accommodate the component. These allocation versus capacity comparisons apply to all components, SCA-compliant or not.

The minimum SPD is not as bad as it looks in the DTD. Figure 19.5 provides the SPD XML necessary to get the SCARI Core Framework to recognize and accommodate our legacy Hello World application.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE softpkg SYSTEM "dtd/softpkg.2.2.dtd">
3. <softpkg id="DCE:5975bdaa-a9ff-44f3-b9ca-7b12272ffbf5"
4. name="demoHelloResource" type="sca_non_compliant">
5. <!-- required -->
6.    <author>
7.    </author>
8.    <!-- required -->
9.    <implementation id="DCE:16dff434-67d4-4654-a7f8-d609851df7ac">
10.       <code type="Executable">
11.           <!-- required -->
12.           <localfile name="/helloWorld"/>
13.       </code>
14.       <os name="Linux"/>
15.       <processor name="x86"/>
16.       <runtime name="Xterm" version="1"/>
17.
18.       <!-- at least one of os OR processor OR dependency -->
19.       <!-- Must match UUID in ExecutableDevice MIPS property -->
20.       <dependency type="mips_allocation">
21.          <propertyref refid="DCE:F364A630-5F0E-11d4-8164-
                              00508B6A52E6" value="2"/>
22.       </dependency>
23.       <dependency type="memory_allocation">
24.           <propertyref refid="DCE:F364A630-5F0E-11d4-8164-
                              00508B6A52E6" value="200"/>
25.           </dependency>
26.  </implementation>
27. </softpkg>
```

Figure 19.5. Legacy Hello World SPD

The SPD XML has comments to help guide the reader but a few key elements are worth noting. Line 12 is the filename of the executable – notice that pathname is not included. Again the SCA provides little guidance on pathnames other than to say that the files to be loaded and the corresponding XML should all be in the same place. (More guidance is offered by the SCA with respect to naming contexts which could be extended to cover pathnames.) The file image identified by line 12 is associated with the implementation, *os.name* = `Linux`, *processor.name* = `x86` (lines 14 and 15). These names are standardized by the SCA specifications, section 2.2.1 in Attachment 2 to Appendix D. Unfortunately the attachment specifies the compliant spelling for Linux as 'Linix'. An additional element called *runtime* (line 16) is used by the SCARI Core Framework to indicate whether an implementation runs in the `java` runtime or not. We will use this parameter to get the SCARI Core Framework to recognize runtimes other than java. In our example we specify the runtime name attribute as `Xterm` (line 16). Recall that an SPD can identify numerous implementations, so the same SPD could also be used to refer to a vxWorks/PPC image. The Core Framework will select the appropriate image file for the node.

Finally, lines 20 through 25 identify mandatory allocations required of components that depend on an *ExecutableDevice*. Although the SCA does not attempt to standardize these critical allocation properties they are sufficiently documented in the *ExecutableDevice's* PRF file. It is mandatory that the UUIDs in lines 21 and 24 match UUIDs attached to the corresponding capacity properties in the *ExecutableDevice's* PRF file. The SCARI Executable Device is identified in line 17 in the top level `Node1.dcd.xml,` path=`$SCA_HOME/demosources/Node1/profile`. This is a list of all devices started on the node when the Device Manager/Domain Manager is instantiated. The SPD associated with the SCARI Executable Device is `ExecutableDevice.spd.xml`. Line 11 of the SPD refers to the SPD level Properties File: `ExecutableDevice_SPDLevel.prf.xml`. The device's capacity is 'uniquely' identified in lines 5 and 10. As seen in previous SCARI examples, these UUIDs appear to be cut and pasted and hence are not really unique.

19.3 HMI Applications

In addition to the main program and the two XML files we need a couple of ancillary applications in order to install and create our Hello World application. First of all we need to pre-install Hello World by putting the executable, Hello World, in the file system where the Core Framework expects to find it. For the SCARI Core Framework, the legacy executable, SAD, and SPD need to reside in the `$SCA_HOME/demosources/Node1/profile/HelloWorld` directory.

The ancillary applications can run from anywhere – and this means anywhere, as long as one can get to the Name Service. Figure 19.6 shows a short code segment that is used to invoke the `DomainManager::installApplication()` operation.

```
#include <orbit/orbit.h>
#include <ORBitservices/CosNaming.h>
#include "stubs/SCA.h"

void startOrb(int* argc, char** argv,
        CORBA_ORB *orb, CORBA_Environment *ev)
{
  *orb = CORBA_ORB_init(argc, argv, "orbit-local-orb", ev);
  printf("ORB initialized\n");
}

```

Figure 19.6. Hello World Install program

```
39. static CosNaming_NameComponent allObjs[1][3] = {
40.     {{"SCARI_DM",""},{"DomainManager",""},{"",""} },
41. };
42.
43. int main( int argc, char * argv[] )
44. {
45.     CORBA_Environment ev;
46.     CORBA_ORB orb;
47.     CosNaming_NamingContext myNC=CORBA_OBJECT_NIL;
48.     CosNaming_Name myBindingName;
49.
50.     SCA_CF_DomainManager dmObj=CORBA_OBJECT_NIL;
51.     CORBA_string aString="/HelloWorld/HelloWorldApp.sad.xml";
52.
53.     CORBA_exception_init(&ev);
54.     startOrb(&argc,argv,&orb,&ev);
55.
56.     myNC = CORBA_ORB_resolve_initial_references(
57.                             orb,"NameService",&ev);
58.     myBindingName._length=2;
59.     myBindingName._buffer= &(allObjs[0][0]);
60.     dmObj=(SCA_CF_DomainManager)
61.                 CosNaming_NamingContext_resolve(myNC,
62.                     (const CosNaming_Name *)&myBindingName, &ev);
63.
64.     SCA_CF_DomainManager_installApplication(dmObj,aString,&ev);
65.
66.     return 0;
67. }
```

Figure 19.6. (Continued)

For clarity, the standard headers and exception handlers have been left out of the code segment and the cosNaming name of the Domain Manager has been hard-coded (lines 39 and 40). The user should save this program as installHW.c, short for ‘installHello World’. In building this application the user will need to include, for linking, the SCA object files (stubs and common) as well as the Name Service libraries. As before, the user is required to fire up the Java Naming Service and Core Framework and launch the Node1 Devices. The user then provides a reference to the Name Service on the command line, for example:

```
./installHW -ORBInitRef NameService=corbaloc::1.2@127.0.0.1:1050/
  NameService
```

The Name Service is used to retrieve the Domain Manger object reference (line 60) and the application is installed in line 64. The single parameter required of the `installApplication()` operation is the qualified name of the SAD file. The string name of our SAD file is found in line 51. If the three files (executable, SAD, and SPD) are located in the right place and consistently named the `install()` operation should run without incident. The reader can run the previously provided queryDomainManager program and witness the presence of the new ApplicationFactory object. This `install()` operation need only be run once because the SCA requires core frameworks to remember which applications are installed across power cycles. SCARI performs this function with a file called `ApplicationFactory.conf` that is read on Core Framework startup. Essentially the application factories are re-installed from scratch every time.

Before we are able to instantiate or `create()` our Hello World application there are some modifications required to the Core Framework itself. An applications programmer could inherit the *ApplicationFactory* interface and overload the `create()` operation. This is a server side operation, and because the server software is written in Java our overloaded operation would be too. The SCARI Core Framework was principally designed to work with SCA-compliant applications in packaged Java jar files. Our legacy application is simply an executable meant to run from the command line. The SCARI designers actually built in the ability to deal with non-Java applications but a few code modifications are required to make it work.

The first modification is to recognize a non-Java application and form the command line argument.

in $SCA_HOME/demosources/devices
modify ExecutableDeviceOperationsImpl.java as follows:

1. modify line 870 to read:
   from: `if(runtimeName != "")`
   to: `if (runtimeName.compareToIgnoreCase("java") == 0)`
2. make a copy of the entire else block, lines 908–919
3. paste it underneath the orginal else block
4. modify line 908:
   from: `else`
   to: `else if(runtimeName.compareToIgnoreCase("Xterm") == 0)`
5. insert two lines before line 918 to read:
   ```
   execCmdList.add("xterm");
   execCmdList.add("-e");
   ```

Change 1 will distinguish between a Java and non-Java runtime. The SCARI Core Framework uses the SPD element `runtime` to convey this information to the Domain Manager. The SCA does not specify how to convey run-time information so this behavior is likely to be unique to SCARI. Modifications 2–4 add an elseif block appropriate to a runtime of Xterm. When runtime is not 'java' or 'Xterm', then the final else clause executes the filename just as if it were on the command line. Modification 5 adds lines to the Xterm block that prepends the executable filename with the command and parameter necessary for the executable to be run in a newly spawned X terminal. So for runtime = Xterm, the SCARI Core Framework will send the following command line to the OS:

```
xterm -e pathname/appName
```

This will take the name of our executable as identified in the SPD and launch it in an xterm. Upon execution a new window is spawned and our Hello World application begins execution.

Additional modification is required to the SCARI software: Namely, the Core Framework was tested only for SCA-compliant waveforms and not for legacy applications so there are a few loose ends in the software.

in $SCA_HOME/scasources/SCA/CFImpl
modify ApplicationFactoryImpl.java as follows:

6. add line 702:
```
if (assemblyController != null)
```

This simple modification checks for a null `assemblyController` object reference before attempting to run the `configure()` operation on it. Since our application is not CORBA-based it has no object reference. The final modification has to do with the lack of an SCD file.

in $SCA_HOME/scasources/util/domainprofile/launchers
modify SADDeployedComponentLauncher.java as follows:

7. insert before line 476:
```
String scdFileName = "";
if (scdParser != null)
    scdFileName = applicationPath + scdParser.
      getPRFFile();
```
8. Modify line 481:
   from: `applicationPath + scdParser.getPRFFile(),`
   to: `scdFileName,`

Again this simple check will avoid the `getPRFFile()` operation from being performed on a null object reference when the scdParser object does not exist. With these Core Frmework modifications complete, the reader can re-build each of affected sub-directories and then do a '`make clean`' and a '`make`' to re-build the `$SCA_HOME/lib` directory. The reader can now start the Naming Service, and fire up the ORB with `./DemoPlatformNode1Bootup` in the `scari-Open` directory. We are now ready to instantiate and execute our application. Figure 19.7 shows a short code segment that will invoke the `create()` operation on the correct ApplicationFactory interface.

Once the ORB is started it is necessary to create a few empty parameters that will be passed to the `create()` operation. Line 33 allocates memory for a `CF::Properties`

```
1. static CosNaming_NameComponent allObjs[1][3] = {
2.    {{"SCARI_DM",""},{"DomainManager",""},{"",""}},
3. };
4.
5. int main( int argc, char * argv[] )
6. {
7.   CORBA_Environment ev;
8.   CORBA_ORB orb;
9.
10.   /* vars needed to find the DomainManager */
11.   CosNaming_NamingContext myNC=CORBA_OBJECT_NIL;
12.   CosNaming_Name myBindingName;
13.   SCA_CF_DomainManager dmObj=CORBA_OBJECT_NIL;
14.
15.   /* vars needed to find the correct AppFact */
```

Figure 19.7. Hello World Create program

```
CORBA_string aString="HelloWorld"; /* softwareassembly name */
SCA_CF_DomainManager_ApplicationFactorySequence *afSeq;
SCA_CF_ApplicationFactory anAppFact;

/* vars needed for create() */
SCA_CF_Application ourApp; /* return value */
SCA_CF_Properties* initProp;
SCA_CF_DeviceAssignmentSequence* dasSeq;

/* miscellaneous vars */
CORBA_string bString;
int i;

CORBA_exception_init(&ev);
startOrb(&argc,argv,&orb,&ev);

/* Initialize vars needed for create() */
initProp = SCA_CF_Properties__alloc();
(*initProp)._length=0;
dasSeq = SCA_CF_DeviceAssignmentSequence__alloc();
(*dasSeq)._length = 0;

/* get Domain Manager object reference */
myNC = CORBA_ORB_resolve_initial_references(
                     orb,"NameService",&ev);
myBindingName._length=2;
myBindingName._buffer= &(allObjs[0][0]);
dmObj=(SCA_CF_DomainManager)
   CosNaming_NamingContext_resolve(myNC,
      (const CosNaming_Name*)&myBindingName, &ev);

/* Iterate through sequence of AppFactories and
 * locate the correct appFactory */
afSeq = SCA_CF_DomainManager__get_applicationFactories(dmObj, &ev);
for (i=0; i<(*afSeq)._length; ++i) {
 anAppFact = (SCA_CF_ApplicationFactory) *((*afSeq)._buffer+i);
 bString = SCA_CF_ApplicationFactory__get_name(anAppFact,&ev);
 if (strcmp(aString,bString)==0) break;
 CORBA_free(bString);
}
if (i!=(*afSeq)._length) {
   printf("Found %s appFactory\n",bString);
   CORBA_free(bString);
}

/* run the create method */
ourApp = SCA_CF_ApplicationFactory_create(anAppFact, "rtName",
```

Figure 19.7. (Continued)

```
63.             (const SCA_CF_Properties*)initProp,
64.             (const SCA_CF_DeviceAssignmentSequence*)dasSeq,&ev);
65.
66.   return 0;
67. }
```

Figure 19.7. (Continued)

type. Recall that this is just an unbounded sequence of `DataType`. Line 34 sets the length of that unbounded sequence to zero. This parameter is normally used to over-ride `configure()` values found in an application's PRF files. Line 35 allocates an sequence of device assignments. These are devices that would be allocated to the application. There is no run-time information contained in the DeviceAssignment sequence. It is merely an sequence of strings that show how application components (identified as UUID strings) map to devices (identified as UUID strings). The data structure is used in a strictly clerical sense by the SCARI Core Framework. We choose to allocate (line 35), and pass an empty sequence (line 36). Lines 38 through 45 uses the Name Service to locate the Domain Manager:

```
./createHW -ORBInitRef NameService=corbaloc::1.2@127.0.0.1:1050/
  NameService
```

Line 49 will perform a `_get()` operation that returns an unbounded sequence of Application Factories. Line 51 retrieves the object reference of an individual Application Factory. Line 52 performs a `__get_name()` operation on the ApplicationFactory in order to recover its name attribute. When an ApplicationFactory named 'Hello World' is found, the search loop is exited (line 53) and the `ApplicationFactory::create()` operation is exercised on the Hello World ApplicationFactory object (line 62). As required by the SCA, a run-time name is given to the application, `'rtName'`, as a parameter to the `create()` operation. Our helper application should exit successfully and an X-terminal with 'Hello World' running should appear.

19.4 Shutting Down

Typically any kind of distributed application is harder to shutdown than startup. We experience a bit of relief with the SCARI CF in that the existing GUI-based Application Manager can be used to shutdown our Hello World application cleanly. At the $SCA_HOME directory, start the Application Manager by typing

```
./startApplicationManager &
```

A window will open up that shows the installed applications. Click on the Hello World application and the user will see the `'rtName'` instance of the application running. Highlight the name of the instance and click 'Shutdown'. The 'Hello World' X-terminal should disappear and the process should be dropped from the process list.

It should be possible to run multiple instances of the Hello World program by merely providing a unique run-time name to each instance.

19.5 An SCA-compliant Hello World Application

In designing an SCA-compliant application we will divide responsibilities between the application and the device. Our approach will be to create an SCA-compliant terminal device which will be started when the Core Framework and other devices are started on the node. Recall the output from the Name Service after the node is initialized (see Figure 17.3) There are currently two devices, Audio and RF, as well as a Device Manager and a Logger service. We will modify the Device Configuration Descriptor (DCD), `Node1.dcd.xml`, to include a new TerminalDevice. Later an application is provided that will make a Port connection to the TerminalDevice. Once the application is started a string will be sent for display on the terminal. Our Hello World application will have four components: a terminal device; a talk application that generates the 'Hello World!' string; and two Port objects. A uses Port will be associated with the application and a provides Port with the terminal device. For the sake of brevity connections will not be made to the event channels or logger; however, both the device and application will be registered with the Name Service. Normally the IDM event channel is used by devices to announce a change in usageState, adminState, or operationalState. Applications use the ODM event channel to announce the addition or removal of components. Messages are redundantly posted to the logger object. The Core Framework will take care of some of these notifications. Figure 19.8 shows the objects that compose our SCA-compliant Hello World application.

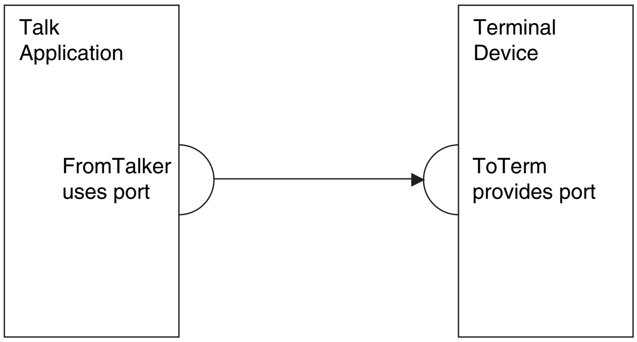


Figure 19.8. SCA-compliant Hello World application

19.5.1 An SCA-compliant Terminal Device

All four components in the Hello World application are CORBA objects that extend Core Framework interfaces. Figure 19.9 provides the very simple IDL for the Terminal object. To avoid contaminating the global namespace, line 6 creates a module called `wileyDevices`. Two interfaces are defined in that module, one for the Port and the other the terminal device itself. No new operations are added to these objects. We intend to use only operations that are inherited from the Core Framework.

```
1. #ifndef _TERMINAL_IDL
2. #define _TERMINAL_IDL
3.
4. #include "SCA.idl"
5.
6. module wileyDevices
7. {
8.   interface termPort : SCA::CF::Port, SCA::PushPorts::
     OctetSeqConsumer
9.  {
10.    // inherits
11.    // oneway void processOctet Msg(in CF::OctetSequencemsg,
12.    //                               in CF::Properties options);
13.   };
14.
15.   interface terminalDevice : SCA::CF::Device {
16.
17.   };
18.
19. };
20.
21. #endif
```

Figure 19.9. Terminal Device IDL

The SCARI implementation extends the SCA by offering Push and Pull Port interfaces. The SCA only defines Port types, whereas the SCARI implementation actually implements push and pull operations on the different types of ports. For example, these SCA port types include sequences of longs, Booleans, floats, etc. For our terminal port we choose the SCA octet sequence. Line 8 shows the termPort interface inheriting both the SCA Port interface and the SCARI OctetSeqConsumer interface. The SCA Port interface provides operations for connect and disconnect while the OctetSeqConsumer provides a processOctetMsg operation. The Talk application will invoke the processOctetMsg interface to send a null-terminated sequence of octets to the terminal device. The terminal implementation of processOctetMsg will simply print the octet sequence to the Xterm. Line 15 shows the interface for the terminalDevice itself. It has no new operations or attributes but will implement only the 19 operations it inherits from the SCA Device interface. These SCA standard interfaces primarily allow the Core Framework to control the life cycle of the terminal device.

When the SCARI Core Framework is started a DeviceManager object reads the `Node1.dcd.xml` file and begins to instantiate devices and then register them with the Domain Manager. We simply modify the XML to point to a new device – that is our terminal device executable – and the Core Framework does the rest. The Core Framework will invoke operations on our device object during this instantiation procedure so, at a minimum, our executable needs to implement these operations. It is first necessary to compile the Terminal IDL into ‘C’ code and then into object code that can be linked in to the executable. Since we are implementing the server side of our object it will be necessary for the IDL compiler to generate skeleton-impl ‘C’ code. The job of implementing the terminal device will be to fill in the skeleton-impl code. The command to compile the IDL is

```
orbit-idl-2 --showcpperrors -I $SCA_HOME/idl --skeleton-impl Terminal.idl
```

The result of the compile operation will be the five files: Terminal.h; Terminal-stubs.c; Terminal-skels.c; Terminal-common.c; and Terminal-skelimpl.c Everything except the skelimpl file can be compiled by adding their source code filenames to the Makefile found in Figure 17.2. Because of the inheritance of the Device interface from the Core Framework IDL the skelimpl contains many more data structures, prototypes, and empty implementations than we will actually need. In fact all of the parent Core Framework objects are also present in the skelimpl – these are simply not needed. It is helpful but not mandatory to create a separate skelimpl file and then cut and paste only the portions of the Terminal-skelimpl.c that we will actually implement. There are five major sections to the skelimpl file: 1) servant structs; 2) prototypes; 3) epv structs; 4) vepv structs; and 5) the empty impl functions. We will extract only what is needed for our terminal implementation. We will need only two structs from the first section, `impl_POA_wileyDevices_termPort` and `impl_POA_wileyDevices_terminalDevice`. The reader will notice that SCA::Device attributes are contained within these data structures, e.g. identifier, compositeDevice, and profile. Proceed to the next section of the skelimpl file by searching for ‘stub proto’. This section contains prototypes for all the operations. Extract any prototype that contains `wileyDevices_termPort` or `wileyDevices_terminalDevice`. Do the same extractions for the ‘epv struct’ and ‘vepv struct’ sections. The final section is found by searching for ‘Stub impl’. These are the empty functions that we will fill in to implement our device. There a total of 20 interfaces that must be implemented on the terminalDevice and five interfaces for the termPort object. Figure 19.10 lists these functions, names.

```
1. impl_wileyDevices_termPort__create()
2. impl_wileyDevices_termPort__destroy()
3. impl_wileyDevices_termPort_connectPort()
4. impl_wileyDevices_termPort_disconnectPort()
5. impl_wileyDevices_termPort_processOctetMsg()
6. impl_wileyDevices_terminalDevice__create()
7. impl_wileyDevices_terminalDevice__destroy()
8. impl_wileyDevices_terminalDevice_initialize()
9. impl_wileyDevices_terminalDevice_releaseObject()
10. impl_wileyDevices_terminalDevice_runTest()
11. impl_wileyDevices_terminalDevice_configure()
12. impl_wileyDevices_terminalDevice_query()
13. impl_wileyDevices_terminalDevice_getPort()
14. impl_wileyDevices_terminalDevice__get_identifier()
15. impl_wileyDevices_terminalDevice_start()
16. impl_wileyDevices_terminalDevice_stop()
17. impl_wileyDevices_terminalDevice__get_usageState()
18. impl_wileyDevices_terminalDevice__get_adminState()
19. impl_wileyDevices_terminalDevice__set_adminState()
```

Figure 19.10. Terminal Device functions requiring implementation

```
20. impl_wileyDevices_terminalDevice__get_operationalState()
21. impl_wileyDevices_terminalDevice__get_softwareProfile()
22. impl_wileyDevices_terminalDevice__get_label()
23. impl_wileyDevices_terminalDevice__get_compositeDevice()
24. impl_wileyDevices_terminalDevice_allocateCapacity()
25. impl_wileyDevices_terminalDevice_deallocateCapacity()
```

Figure 19.10. (Continued)

The `__create` and `__destroy()` functions are essentially constructors and destructors for the objects. We will need to modify the parameter list of the `terminalDevice__create` function to initialize readonly attributes. When the Core Framework synthesizes the command line to start a device certain identifying information is passed in to the executable namely, `DEVICE_ID`, `DEVICE_LABEL`, `PROFILE_NAME`, and `DEVICE_MGR_IOR`. These same parameters will need to get passed to the terminalDevice when it is created. The impl_POA data structure generated by the IDL compiler already contains variables to hold these values: `attr_identifier`, `attr_label`, and `attr_softwareProfile`. They will need to be initialized inside the create operation. For the sake of brevity we will only provide code for five of the 25 operations. For instance it is unlikely that the SCARI Core Framework will invoke the runTest() operation so we can leave that function empty. We know the following operations will get called by the Core Framework over the terminal's lifetime: `get_identifier()`, `initialize()`, `get_softwareProfile()`, `get_label()`, and `releaseObject()`. We can surmise that when we instantiate and run the Talk application there will be calls to `getPort()`, `termPort_connectPort()`, and of course `termPort_processOctetMsg()`. The following code extracts provides the precise implementations.

First of all it is necessary to modify the terminalDevice__create parameter list as follows:

```
static wileyDevices_terminalDevice impl_wileyDevices_terminalDevice__create(
  PortableServer_POA poa, sharedMem *ss, pid_t myPID, /* ← added 2 params */
  CORBA_Environment *ev)
```

where sharedMem is defined as:

```
typedef struct {
     SCA_CF_DeviceManager theDevMgr;
     SCA_CF_Device ourTerminalDevice;
     wileyDevices_termPort ToTerm;
     CosNaming_NamingContext myNC;
     CosNaming_Name *pMyName;
     CORBA_char *devId,*devLbl,*profName;
} sharedMem;
```

Most of these objects will need to be accessible to the terminalDevice servant. For instance devId is the UUID passed in from the Core Framework. It needs to be available also to the terminalDevice servant so it can be supplied as a return value to the `__get_identifier()` operation. A little thought will need to go into the design of our terminal device. But for now let's define the `__create` function and the rest of the skelimpl code.

```
/* ------ init private attributes here ------ */
newservant->attr_identifier = ss->devId;
newservant->attr_label = ss->devLbl;
newservant->attr_softwareProfile = ss->profName;
newservant->privatePID = myPID;
newservant->ToTermPort = CORBA_Object_duplicate(ss->ToTerm,ev);
/* ------ ---------- end ------------ ------ */
```

Essentially we copy parameters that are passed to the `create` function in to the impl_POA structure that is known to the terminalDevice servant object. At the beginning of the skelimpl file we will need to add two members to the default impl_POA structure as follows:

```
pid_t privatePID;
SCA_CF_Port ToTermPort;
```

We now present the implementations one by one:

For `impl_wileyDevices_terminalDevice_releaseObject()` add

```
  kill(servant->privatePID,SIGHUP);
```

This signal will cause the servant object to exit its CORBA_run loop and start to shutdown. No modification is required for `initialize()`. Our implementation has nothing to do. A better implementation might clear the screen or something.

For `impl_wileyDevices_terminalDevice_getPort()`add

```
  retval = CORBA_Object_duplicate(servant->ToTermPort,ev);
```

For `impl_wileyDevices_terminalDevice__get_identifier()`add

```
  retval = CORBA_string_alloc(strlen(servant->attr_identifier));

  strcpy(retval,servant->attr_identifier);
```

For `impl_wileyDevices_terminalDevice__get_label()`add

```
  retval = CORBA_string_alloc(strlen(servant->attr_label));

  strcpy(retval,servant->attr_label);
```

For `impl_wileyDevices_terminalDevice__get_softwareProfile()` add

```
/* ------ insert method code here ------ */
   char *cPtr;
/* The SCA requires the software profile filename to be in the form */
/* of a profile element (Appendix D.9) */
/* As passed in the command line: /WileyTerminal.spd.xml, example */
/* output: <profile filename="/AudioDevice.spd.xml" type="SPD"/> */
static const char beginS[] = "<profile filename=\"";
static const char  endS[] = "\" type=\"SPD\"/>";
retval = CORBA_string_alloc( strlen(servant->attr_softwareProfile) +
                               strlen(beginS) + strlen(endS) );
```

```
strcpy(retval,beginS);
cPtr = retval + strlen(retval); /* address of terminating \0 */
strcpy(cPtr,servant->attr_softwareProfile);
cPtr = retval + strlen(retval); /* address of terminating \0 */
strcpy(cPtr,endS);
/* ------ ---------- end ----------- ------ */
```

As the comment indicates, the softwareProfile returned by the `__get()` operation is in the form of a profile element. This is not the way it was passed to the terminal device when it is executed by the Core Framework so some string manipulation is involved. The SCARI Core Framework will invoke this operation and test the syntax of the return value. So far there is not too much device software that had to be written for SCA compliance. It is more of a book-keeping function. The Core Framework gives the device certain strings from the XML and the device regurgitates the strings when queried by the Core Framework. So let's get to the important code: that is, the code that will actually print to the screen.

For `impl_wileyDevices_termPort_processOctetMsg()` add

```
    printf("%s\n",(char *)msg->_buffer);
    fflush(stdout);
```

Those two lines are the guts of the terminal device code. The remaining operations on the termPort don't really do anything in our implementation and can be left alone. The SCA requires some 45 user-defined exceptions to be posted, for instance, invalidPort. Our sample application is designed for success not failure so we choose not to implement the throwing of the 45 user-defined exceptions. The reader is advised that the JTAP tool which is used for SCA-compliance testing does in fact test for the user-defined exceptions. Now for the main program. Figure 19.11 provides the terminalDevice code that gets executed by the Core Framework.

```
/* normal includes, orbit, CosNaming, etc. */
#include "SCA.h"
#include "Terminal.h"

#include "wileyTerm-skelimpl.c"

static const struct timespec tenthSecond = {0,100000000};

int main( int argc, char * argv[] ) {

    int i;
    CORBA_Environment ev;
    PortableServer_POA root_poa=CORBA_OBJECT_NIL;
    CORBA_char *cPtr;
    static const char under[] = "_";
    sharedMem simpleBlock = {
        CORBA_OBJECT_NIL, CORBA_OBJECT_NIL, CORBA_OBJECT_NIL,
        CORBA_OBJECT_NIL};

```

Figure 19.11. Main program for Terminal Device

```
  CORBA_exception_init(&ev);

  signal(SIGABRT, orderlyShutdown);
  signal(SIGHUP, orderlyShutdown);

  /* The orb will ignore the args it cannot recognize */
  /* and remove those args it understands */
  startOrbPOA(&argc,argv,&global_orb,&root_poa,&ev);

  /* Parse the SCA-required args, will need to pass to
     terminal device when it is created */
   for (i=1; i<argc; ++i) {
     if (strcmp(argv[i],"DEVICE_ID")==0)
        simpleBlock.devId = argv[i+1];
     if (strcmp(argv[i],"DEVICE_LABEL")==0)
        simpleBlock.devLbl = argv[i+1];
     if (strcmp(argv[i],"PROFILE_NAME")==0)
        simpleBlock.profName = argv[i+1];
}

/* Get the Device Manager Object Reference */
 for (i=1; i<argc; ++i)
     if (strcmp(argv[i],"DEVICE_MGR_IOR")==0) break;
 simpleBlock.theDevMgr =
       (SCA_CF_DeviceManager)CORBA_ORB_string_to_object(global_orb,
       (const CORBA_char *)argv[i+1],&ev);

  /* create the Port object */
  simpleBlock.ToTerm =
     impl_wileyDevices_termPort__create(root_poa, &ev);

  /* Create the terminal device */
  /* need to pass a pointer to sharedMem and the process number */
  simpleBlock.ourTerminalDevice =
     impl_wileyDevices_terminalDevice__create(
     root_poa, &simpleBlock, getpid(), &ev);

  /* prepare to bind the terminalDevice with the Naming Service */
  /* Need to get the initial naming context */
  simpleBlock.myNC = CORBA_ORB_resolve_initial_references(
                         global_orb,"NameService",&ev);

  simpleBlock.pMyName = CosNaming_Name__alloc();
  simpleBlock.pMyName->_length = 1;
  simpleBlock.pMyName->_buffer = CosNaming_NameComponent__alloc();

  /* name = label + "_" + uuid */
```

Figure 19.11. (Continued)

```
67.   simpleBlock.pMyName->_buffer->id = CORBA_string_alloc(
68.              strlen(simpleBlock.devLbl)+strlen(simpleBlock.devId)+1);
69.    strcpy(simpleBlock.pMyName->_buffer->id,simpleBlock.devLbl);
70.    cPtr = simpleBlock.pMyName->_buffer->id +
71.                              strlen(simpleBlock.pMyName->_buffer->id);
72.    strcpy(cPtr,under);
73.    cPtr = simpleBlock.pMyName->_buffer->id +
74.                              strlen(simpleBlock.pMyName->_buffer->id);
75.    strcpy(cPtr,simpleBlock.devId);
76.    simpleBlock.pMyName->_buffer->kind = CORBA_string_alloc(0);
77.    *(simpleBlock.pMyName->_buffer->kind) = '\0';
78.
79.    if ( fork()!=0 )
80.    {
81.       CORBA_ORB_run(global_orb,&ev); /* returns on signal */
82.
83.       CosNaming_NamingContext_unbind(simpleBlock.myNC,
84.            (const CosNaming_Name*)simpleBlock.pMyName, &ev);
85.       kill(getppid(),SIGHUP);
86.    }
87.    else
88.       registration (&simpleBlock); /* child process */
89.
90.   return 0;
91. }
```

Figure 19.11. (Continued)

This code segment begins in line 3 by including the skelimpl.c. The reader should insert the definition of the sharedMem data structure before this include. Since this is server side software signal handlers are installed in lines 22 and 23 to allow the user the ability to shut the ORB down cleanly. This signal handler and the startOrbPOA functions are exactly the same as found in Figures 17.18 and 17.15. Lines 31 through 38 recognize the SCA parameters passed in the command line and copy their pointers to the sharedMem variable. Lines 41 through 45 retrieve the Device Manager's stringified object reference and converts it to a true object reference. The Device Manager object reference is provided in that SCA-compliant *Device*s are required to register with it once they are up and running. Now it's time to create our two objects. Line 49 creates the Port object and line 54 the terminalDevice object. A pointer to the sharedMem variable and the current process number is passed to the terminal's `__create` function. Finally in preparation for the Name Service bind() operation, line 59 recovers the object reference to the initial Name Context. Lines 62 through 77 form the Name that will be used to bind to the terminal's object reference. The SCA does not mandate that *Device*s register with the Name Service, only with application components. When it comes to performing Port connections the SCARI Core Framework requires both components to be registered with the Name Service.

As of line 79 we are all ready to register our Device with both the Name Service and Device Manager; however, our Device is not running yet. The POA servant does not accept connections until `CORBA_ORB_run()` gets called and that call will block permanently until a shutdown is signaled. Once registered the SCARI Core Framework will start making calls on our terminal object which isn't running yet. This minor chicken and egg scenario is easily handled by forking off a new process that will test for the existence of the terminal device before registering with the Name Service and Device Manager. We know that `fork()` is not on the list of mandatory POSIX APIs. However that list applies only to *Application* components not *Device*s so we are well within the guidelines of the SCA. The call to `fork()` clones the parent process and creates a child process. The only distinguishable feature between parent and child is that `fork()` returns a value of zero to the child process whereas in the parent the non-zero process number of the child is returned. So the parent will execute line 81 which puts the servant online to accept connections to our Port and terminalDevice objects. The child will execute line 88 and call the registration routine. Note that a pointer to the sharedMem variable is passed. The code for the register routine is provided in Figure 19.12.

```
1. int registration (sharedMem *sBlock)
2. {
3.    CORBA_Environment ev;
4.    CORBA_exception_init(&ev);
5.
6.    /* bind with the Name Service */
7.    CosNaming_NamingContext_bind(sBlock->myNC,
8.                (const CosNaming_Name*)sBlock->pMyName,
9.                (const CORBA_Object)sBlock->ourTerminalDevice, &ev);
10.
11.   /* register with the Device Manager */
12.   SCA_CF_DeviceManager_registerDevice(sBlock->theDevMgr,
13.      (const SCA_CF_Device)sBlock->ourTerminalDevice, &ev);
14.
15.   return 0;
16. }
```

Figure 19.12. Registration function for Terminal Device

Although the term 'shared memory' is used, there are two separate process spaces involved. This 'shared' memory is unilateral from parent to child. When the fork occurs a child is cloned that has an exact copy of the state of the parent. Any variable that changes state in the parent will not be reflected in the child and vice versa. In line 7 the terminal object and Name are bound on the initial Name Context. Finally, line 12 registers the terminal device with the Device Manager. Thus the terminal's object reference is available through the Device Manager or the Name Service. It is likely that in other Core Framework implementations, Devices will only be registered with the Device Manager. With the terminalDevice software now complete it's time to address modifications and additions to the XML.

19.5.2 Domain Profile for Terminal Device

The Device Component Descriptor (DCD) is used by the Device Manager at boot time to start up all the Devices on a particular computational node. Like other files in the

Domain Profile the DCD is broken up into sections. The ones that will get modified are *componentfiles* and *partitioning*. We will not make modifications to *connections* because our *Device* example will not connect to any other *Device* or service. Our single connection between the Talk application component and the Terminal device will be described in the SAD file. The `Node1.dcd.xml` file is located in the directory `$SCA_HOME/demosources/Node1/profile`. Add the following *componentfile* to the *componentfiles* element of the Node1 DCD:

```
<componentfile id="wileyTermFile" type="SPD">
<localfile name="TerminalDevice.spd.xml"/>
</componentfile>
```

This addition provides an Id to the component which will be referred to later in the DCD and, as always, the name of the *Device*'s SPD. We reiterate here that EVERY component – device or application – requires an SPD. SCA-compliant, CORBA-enabled components will also require an SCD. Add the following *componentplacement* sub-element to the *partitioning* element.

```
<componentplacement>
<componentfileref refid="wileyTermFile"/>
<componentinstantiation id="DCE:fe994973-b25a-4aa4-95e6-9b58555c39bc">
<usagename>TerminalWindow</usagename>
</componentinstantiation>
</componentplacement>
```

The `componentfileref.refid` attribute must match the `componentfile.id` from the previous section. These elements link the two sections together. When the terminal device registers with the Name Service it will concatenate *usagename* with *Id* separated by an underscore. The result that will appear on the initial context of the name tree is

```
TerminalWindow_DCE:fe994973-b25a-4aa4-95e6-9b58555c39bc
```

The DCD `componentfile.localfile.name` attribute provides a link to the terminal's SPD which is presented in Figure 19.13.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE softpkg SYSTEM "dtd/softpkg.2.2.dtd">
<softpkg id="DCE:24573c13-7e78-42e0-8a11-23cab6ecc7dd"
name="wileyTerm">
  <author>
  </author>
  <propertyfile type="PRF">
     <localfile name="TerminalDevice_SPDLevel.prf.xml"/>
  </propertyfile>
  <descriptor>
     <localfile name="TerminalDevice.scd.xml"/>
  </descriptor>
  <implementation id="DCE:cc62f6a9-5ffb-4c7a-8031-d07d23a2a478">
      <propertyfile>
         <localfile name="TerminalDevice.prf.xml"/>
```

Figure 19.13. Terminal device SPD

```
15.       </propertyfile>
16.     <code type="Executable">
17.     <localfile name="wileyTerm"/>
18.     </code>
19.     <runtime name="Xterm" version="1"/>
20.     <processor name="x86"/>
21.     <os name="Linux"/>
22.   </implementation>
23. </softpkg>
```

Figure 19.13. (Continued)

This SPD sequence provides several links to further XML files that will provide information required for successful instantiation of the terminalDevice. Line 7 provides a link to a Properties file (PRF) that will contain allocation properties and Log Service settings for this device across all implementations: ppc, x86, linux, windows, etc. Line 10 contains a reference to the terminal's SCD (more about that later). Line 14 refers to a PRF file unique to this particular implementations (that is, x86-Linux-Xterm). This PRF file contains command line exec parameters required by the ORB. Finally, in line 17 an x86-Linux-Xterm implementation component wileyTerm is identified as the name of the executable. When the terminalDevice code is successfully compiled this executable should be copied to `$SCA_HOME/demosources/Node1/profile`. This directory is also home to all the terminal device's XML. Remember this same SPD can be supplemented to refer to other implementations. In this sense the SPD readily supports portability in that the Device Manager will match the correct executable to the processor type/operating system combination.

Next we present the two PRF files. Figure 19.14 shows the PRF file that applies to all implementations and Figure 19.15 shows the PRF unique to this implementation.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE properties SYSTEM "dtd/properties.2.2.dtd">
3. <properties>
4.     <simple id="TERMINAL_TYPE" type="string" mode="readwrite">
5.         <value>Xterm</value>
6.         <kind kindtype="allocation"/>
7.         <action type="eq"/>
8.     </simple>
9.     <simplesequence id="PRODUCER_LOG_LEVEL" type="long"
10.                        name="PRODUCER_LOG_LEVEL">
11.       <kind kindtype="configure"/>
12.    </simplesequence>
13. </properties>
```

Figure 19.14. SPD level PRF file

In lines 4 through 8, the SCA requires that all Devices have at least one allocation property. In lines 9 through 12, the SCA requires that all components have a `PRODUCER_LOG_LEVEL`

```
14. <?xml version="1.0" encoding="UTF-8"?>
15. <!DOCTYPE properties SYSTEM "dtd/properties.2.2.dtd">
16. <properties>
17.     <simple id="dummy0" type="string">
18.         <value>-ORBIIOPIPv4=1</value>
19.          <kind kindtype="execparam"/>
20.     </simple>
21.    <simple id="dummy1" type="string">
22.         <value>-ORBIIOPUNIX=0</value>
23.          <kind kindtype="execparam"/>
24.     </simple>
25. </properties>
```

Figure 19.15. PRF file containing exec parameters

property. These properties apply across all implementations. Now for the execparams unique to our ORBit ORB. Lines 18 and 22 setup two command line options that are required to configure ORBit for IIOP interoperability. Recall that the default mode of connectivity for ORBit is a secure connection based on UNIX domain sockets. This protects the ORB from Denial of Service attacks. We want to interoperate over TCP-IP with the Java ORB so we enable IPv4 and disable UNIX domain sockets. The last bit of XML required is the SCD. In earlier domain profile examples, we specifically examined the absolute minimum amount of XML required to satisfy the DTD. In the case of the Software Component Descriptor (SCD) all elements are required. Even if an object has no ports the componentfeatures must still be present: it will just be empty. Figure 19.16 provides the SCD for our SCA-compliant terminal device.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE softwarecomponent SYSTEM "dtd/softwarecomponent.2.2.dtd">
<softwarecomponent>
  <corbaversion>2.2</corbaversion>
  <componentrepid repid="IDL:CF/Device:1.0"/>
  <componenttype>device</componenttype>
  <componentfeatures>
    <supportsinterface repid="IDL:CF/Device:1.0"
               supportsname="Device"/>
    <supportsinterface repid="IDL:CF/Resource:1.0"
               supportsname="Resource"/>
    <supportsinterface repid="IDL:CF/PropertySet:1.0"
               supportsname="PropertySet"/>
    <supportsinterface repid="IDL:CF/LifeCycle:1.0"
               supportsname="LifeCycle"/>
    <supportsinterface repid="IDL:CF/PortSupplier:1.0"
               supportsname="PortSupplier"/>
```

Figure 19.16. SCD for SCA-compliant Device

```
18.    <supportsinterface repid="IDL:CF/TestableObject:1.0"
19.            supportsname="TestableObject"/>
20.    <ports>
21.      <provides repid="IDL:PushPorts/OctetSeqConsumer:1.0"
22.          providesname="ToTerm">
23.               <porttype type="data"/>
24.      </provides>
25.    </ports>
26.   </componentfeatures>
27.   <interfaces>
28.    <!--Supported Interfaces -->
29. <interface repid="IDL:CF/Device:1.0" name="Device">
30.   <inheritsinterface repid="IDL:CF/Resource:1.0"/>
31. </interface>
32. <interface repid="IDL:CF/Resource:1.0" name="Resource">
33.   <inheritsinterface repid="IDL:CF/PropertySet:1.0"/>
34.   <inheritsinterface repid="IDL:CF/LifeCycle:1.0"/>
35.   <inheritsinterface repid="IDL:CF/PortSupplier:1.0"/>
36.   <inheritsinterface repid="IDL:CF/TestableObject:1.0"/>
37. </interface>
38. <interface repid="IDL:CF/PropertySet:1.0" name="PropertySet"/>
39. <interface repid="IDL:CF/LifeCycle:1.0" name="LifeCycle"/>
40. <interface repid="IDL:CF/PortSupplier:1.0" name="PortSupplier"/>
41. <interface repid="IDL:CF/TestableObject:1.0"
         name="TestableObject"/>
42. <!- Port Interface ->
43. <interface repid="IDL:CF/Port:1.0" name="Port"/>
44. </interfaces>
45.</softwarecomponent>
```

Figure 19.16. (Continued)

This SCD is very much boilerplate for any SCA-compliant device. It basically reiterates the repository ids of the Core Framework interfaces. This is completely pointless in that minimum ORB's do not have an interface repository. Our device inherits all of these interfaces through SCA::Device. A key piece of information required by the Core Framework is found in lines 21 through 24 – this identifies our port interface. This must match information that will be supplied in our Talk application SAD. That is all there is to it. Once the executable and XML are placed in the proper directory, the next time the Core Framework starts up an Xterm window should pop up.

19.5.3 An SCA-compliant Talk Application

The next task at hand is to design and develop an SCA-compliant application that will send messages to our terminal device. As before we will start with the IDL, fill in the skelimpl, write the main program, and finally present the XML. The IDL for our Talk application is given in Figure 19.17.

We will add no attributes or operations – the Talk application will implement only interfaces inherited from the Core Framework IDL. Since both *Application* and *Device* inherit

```
#ifndef _TALK_IDL
#define _TALK_IDL

#include "SCA.idl"

module wileyApps
{
   interface talk : SCA::CF::Application {

   };
};

#endif
```

Figure 19.17. Talk application IDL

from *Resource*, and *Application* offers no new operations, the list of operations is pretty much the same as Figure 19.10. The Application object does have five additional attributes not found in *Resource* but we will find out that the SCARI Core Framework itself implements those `__get()` operations and our server implementation doesn't even get called.

As before, compile the IDL and the generated 'C' files. Again it is recommended to setup a separate skelimpl file and copy only the data structures, prototypes and Stub implementations directly associated with the `wileyApps_talk` interface and the `CF_Port` interface. The following implementations need to be filled in:

For the `impl_POA_SCA_CF_Port` struct add

```
wileyDevices_termPort *pTerminalPort; /* ptr to shared mem*/
int *pIsConnected; /* ptr to shared memory */
```

For the `impl_POA_wileyApps_talk` struct add

```
SCA_CF_Port FromTalkerPort;
int *pIsStarted; /* ptr to shared memory */
```

Modify the parameter list to `impl_SCA_CF_Port__create()` to include

```
wileyDevices_termPort *pTermPort
int *pIsConnect
```

To the body of `impl_SCA_CF_Port__create()` add

```
newservant->pTerminalPort=pTermPort; /* needed by connect */
newservant->pIsConnected=pIsConnect; /* needed by connect */
```

To the body of `impl_SCA_CF_Port_connectPort()` add

```
  *(servant->pTerminalPort) =
             CORBA_Object_duplicate(connection,ev);
  *(servant->pIsConnected) = TRUE;
```

Modify the parameter list of `impl_wileyApps_talk__create()` add

```
  sharedMem *ss,
```

To the body of `impl_wileyApps_talk__create()` add

```
  newservant->FromTalkerPort =
              CORBA_Object_duplicate(ss->FromTalker,ev);
  newservant->pIsStarted = &(ss->isStarted);
```

To `impl_wileyApps_talk_getPort()` add

```
  retval = CORBA_Object_duplicate(servant->FromTalkerPort,ev);
```

To `impl_wileyApps_talk_start()` add

```
  *(servant->pIsStarted) = TRUE;
```

To `impl_wileyApps_talk_stop()` add

```
  *(servant->pIsStarted) = FALSE;
```

These are all of the modifications required of the skelimpl file. We want our application to be SCA-compliant therefore we will employ threads within the same process space. Where the terminal device example used cloned memory our Talk application relies on shared memory. That is a write to shared memory by one thread will need to be read in a different thread. The skelimpl code inserts show data being written to shared memory via pointers. An example of this is the shared memory variable to control the starting and stopping of the application. The sharedMem struct is provided:

```
typedef struct {
  SCA_CF_Application ourApplication; /* needed to bind with Name Service */
  SCA_CF_Port FromTalker;     /* needed for getPort */
  volatile wileyDevices_termPort TerminalPort; /* set by connect */
  CORBA_char *ncsIOR; /* from CF, application name context */
  CORBA_char *Id, *bindName; /* other info from the CF */
  volatile int isCreated;
  volatile int isConnected;
  volatile int isStarted;
  volatile int isShuttingDown;
  int argc;    /* allows threads access to command line params */
  char **argv;
   } sharedMem;
```

This data structure should be defined prior to the #include for the skelimpl file. Some more transactions with the sharedMem structure include a getPort operation which returns a duplicate of Talk's uses port interface. The Core Framework will then perform a connect on that port object. One of the parameters passed in `connect()` is the object reference to the provides port. This is the port object on the terminal device on which we will be performing `processOctetMsg()`. So internally the `connect()` operation duplicates the provides port object reference to a shared memory area where the thread responsible for sending the 'Hello World' can use it. Note that parameters used to communicate changes of state between threads are qualified as volatile. This keyword forces the compiler to re-read the variable from memory every time. An optimizing compiler might try to save clock cycles by caching the variable into a register.

The main program for the Talk application is given in Figure 19.18. Note that initialization of the ORB and POA does not take place in the main thread – this is different to any previous example and will warrant further examination.

```
int main( int argc, char * argv[] ) {

    CORBA_Environment ev;
    sharedMem simpleBlock = {
       CORBA_OBJECT_NIL, CORBA_OBJECT_NIL,CORBA_OBJECT_NIL, NULL };
    CosNaming_Name myBindingName;
    pthread_t serverT;
    CosNaming_NamingContext appNC=CORBA_OBJECT_NIL;
    CosNaming_Name *pAppName;
     const char outString[] = "Hello World!";
     SCA_CF_Properties *initProp=NULL;
     SCA_CF_OctetSequence *aMsg=NULL;

     CORBA_exception_init(&ev);
     simpleBlock.isCreated = simpleBlock.isConnected = FALSE;
     simpleBlock.isStarted = simpleBlock.isShuttingDown = FALSE;

     simpleBlock.argc = argc;
     simpleBlock.argv = argv;

     pthread_create(&serverT,
              (const pthread_attr_t *)NULL,(void *)server,
                     (void*)&simpleBlock);

     signal(SIGABRT, orderlyShutdown);
     signal(SIGHUP, orderlyShutdown);

     /* wait for objects to be created */
     while (!simpleBlock.isCreated)
        nanosleep(&tenthSecond,NULL);

     /* get the Application's Naming Context */
     appNC = (CosNaming_NamingContext)
```

Figure 19.18. Talk application main program

```
34.      CORBA_ORB_string_to_object(global_orb, simpleBlock.ncsIOR, &ev);
35.
36.    /* bind the Application */
37.    pAppName = CosNaming_Name__alloc();
38.    pAppName->_length = 1;
39.    pAppName->_buffer =CosNaming_NameComponent__alloc();
40.
41.    pAppName->_buffer->id = CORBA_string_alloc(
42.                 strlen(simpleBlock.bindName));
43.    strcpy(pAppName->_buffer->id,simpleBlock.bindName);
44.    pAppName->_buffer->kind =CORBA_string_alloc(0);
45.    *(pAppName->_buffer->kind) = '\0';
46.
47.    CosNaming_NamingContext_bind(appNC,
48.                 (const CosNaming_Name*)pAppName,
49.                 (const CORBA_Object)simpleBlock.ourApplication, &ev);
50.
51.    /* create parameters for the processOctetMsg operation*/
52.    aMsg = SCA_CF_OctetSequence__alloc();
53.    aMsg->_length = strlen(outString)+1;
54.    aMsg->_maximum = aMsg->_length;
55.    aMsg->_buffer = CORBA_sequence_CORBA_octet_allocbuf(aMsg->_length);
56.    strcpy((char *)aMsg->_buffer,outString);
57.
58.    initProp = SCA_CF_Properties__alloc();
59.    (*initProp)._length=0;
60.
61.    /* Wait for connect */
62.    while (!simpleBlock.isConnected) {
63.       nanosleep(&tenthSecond,NULL);
64.    }
65.
66.    operateLoop(ss,aMsg,initProp)
67.
68.    pthread_join(serverT,(void **)NULL);
69.    return 0;
70. }
```

Figure 19.18. (Continued)

After initializing a few variables the main program creates a server thread and passes the shared memory block to it. The thread will parse the command line start the ORB and POA, and then create the Port and Talk objects. The main program will block in a loop at line 29 until the thread has finished creating the two objects. The caller – in this case the Core Framework – passes the application's Name Context as an execparam. Lines 33 and 34 convert that object reference from stringified form. Lines 41 through 49 bind the application Name and object reference to that context. Next main synthesizes the parameters for the processOctetMsg operation (lines 51 to 59). At line 62 the main program will check and wait, if necessary, for the port to be connected. The Core Framework actually calls the connect operation as part of parsing the application's SAD file. Once connected the main program can go into its while-forever loop. This loop will return only when the terminalPort object reference is set to NIL. The while-forever loop is given in Figure 19.19.

```
1. operateLoop(sharedMem *ss, SCA_CF_OctetSequence aMsg,
2.                   SCA_CF_Properties initProp) {
3.   int i=0;
4.   CORBA_Environment ev;
5.   CORBA_exception_init(&ev);
6.
7.   /* wait for application to get started or released */
8.   while (1) {
9.     nanosleep(&tenthSecond,NULL);
10.    if (ss->isStarted) {
11.     if ( (++i)%10 == 0 ) { /* send Msg once a second */
12.     if (!CORBA_Object_is_nil((CORBA_Object)ss->TerminalPort,&ev)){
13.             wileyDevices_termPort_processOctetMsg(ss->TerminalPort,
14.                   aMsg, initProp, &ev);
15.       }
16.      }
17.    }
18    if (CORBA_Object_is_nil((CORBA_Object)ss->TerminalPort,&ev)) {
19.          ss->isShuttingDown = TRUE;
20.          break;
21.    }
22.  } /* end while (1) */
23. }
```

Figure 19.19. While-Forever loop for Talk application

This is a loop that runs 10 times a second. If the application is started (line 10) the loop will send a packet to the terminal device every 10th iteration. In line 12 the software tests for a NIL object reference before attempting the processOctetMsg operation. The skelimpl software will set the provides port object reference to NIL as part of releaseObject(). When this happens lines 18 through 20 will notify the other threads and exit the while-forever loop.

19.5.4 Multi-threaded Servant

In order for this application to work the ORB has to be configured to run in multi-threaded mode. The Root POA cannot be configured to do this so it will be necessary to create a child POA that is configured at creation for 'Thread per Request' operation. The code segment of Figure 19.20 provides this startup sequence.

This code sequence was derived from test code that was posted on the ORBit2 website [29]; we are grateful to the authors for their efforts without which our SCA-compliant application would be in a world of hurt. There only a few differences over the basic RootPOA activation found in Figure 17.15. In line 9 the option passed to `ORB_init()` is modified to read

```
1. static void startOrbPOA(int* argc, char** argv,
2.        CORBA_Environment *ev)
3. {
4.   const static int MAX_POLICIES = 1;
5.   PortableServer_POA rootpoa = CORBA_OBJECT_NIL;
6.   PortableServer_POAManager rootpoa_mgr = CORBA_OBJECT_NIL;
7.   CORBA_PolicyList *poa_policies;
8.
9.   global_orb = CORBA_ORB_init(argc, argv, "orbit-local-mt-orb", ev);
10.
11.   rootpoa = (PortableServer_POA)
12.        CORBA_ORB_resolve_initial_references(global_orb,"RootPOA",ev);
13.
14.   rootpoa_mgr =
15.       PortableServer_POA__get_the_POAManager(rootpoa,ev);
16.
17.   poa_policies = CORBA_PolicyList__alloc ();
18.   poa_policies->_maximum = MAX_POLICIES;
19.   poa_policies->_length = MAX_POLICIES;
20.   poa_policies->_buffer = CORBA_PolicyList_allocbuf (MAX_POLICIES);
21.   CORBA_sequence_set_release (poa_policies, CORBA_TRUE);
22.
23.   poa_policies->_buffer[0] = (CORBA_Policy)
24.           PortableServer_POA_create_thread_policy (
25.                   rootpoa,
26.                   PortableServer_ORB_CTRL_MODEL,
27.                   ev);
28.
29.   default_poa = PortableServer_POA_create_POA (rootpoa,
30.                   "Thread Per Request POA",
31.                   rootpoa_mgr, poa_policies, ev);
32.
33.   ORBit_ObjectAdaptor_set_thread_hint (
34.          (ORBit_ObjectAdaptor) default_poa,
35.                      ORBIT_THREAD_HINT_PER_REQUEST);
36.
37.   CORBA_Policy_destroy (poa_policies->_buffer[0], ev);
38.   CORBA_free (poa_policies);
39.
40.   PortableServer_POAManager_activate(rootpoa_mgr,ev);
41.   CORBA_Object_release((CORBA_Object)rootpoa_mgr,ev);
42.
43. }
```

Figure 19.20. Multi–threaded POA activation

`orbit-local-mt-orb`. The 'mt' here denotes a multi-threaded operation. Everything proceeds normally until line 17 where an unbounded sequence of policies is allocated. The policy itself is returned by a call to `create_thread_policy` in line 24. This operation is part of the minimum CORBA spec in 1998 but is then deleted in the 2002 spec. Line

29 actually creates the new POA as a ‘Thread Per Request POA’ under the root. Line 40 activates the root including the newly created threaded POA. This routine is called from the server thread that was created back in the main program. After starting the ORB and POA the server thread has a few tasks to perform before entering the blocking `CORBA_ORB_run()` routine. The server code is provided in Figure 19.21.

```
void server(sharedMem *sBlock)
{
    int i;
    int argc = sBlock->argc;
    char **argv = sBlock->argv;
    CORBA_Environment ev;

    CORBA_exception_init(&ev);

     /* The orb will ignore the args it cannot recognize */
     /* It will remove those args it understands */
     startOrbPOA(&argc,argv,&ev);

     /* Parse the SCA-required args, will need to pass to
        application object when it is created */
     for (i=1; i<argc; ++i) {
        if (strcmp(argv[i],"COMPONENT_IDENTIFIER")==0)
           sBlock->Id = argv[i+1];
        if (strcmp(argv[i],"NAMING_CONTEXT_IOR")==0)
           sBlock->ncsIOR = argv[i+1];
        if (strcmp(argv[i],"NAME_BINDING")==0)
           sBlock->bindName = argv[i+1];
     }

     /* Create the port first, must pass to Application constructor */
     sBlock->FromTalker =
        impl_SCA_CF_Port__create(default_poa,
           &(sBlock->TerminalPort), &(sBlock->isConnected), &ev);

     /* Create the application */
     sBlock->ourApplication =
        impl_wileyApps_talk__create(
        default_poa, sBlock, &ev);

     /* Notify main program to register the Application */
     sBlock->isCreated = TRUE;
     CORBA_ORB_run(global_orb,&ev); /* block until signal */

     /* notify main program that we are shutting down */
     sBlock->TerminalPort =
         CORBA_Object_duplicate(CORBA_OBJECT_NIL,&ev);

}
```

Figure 19.21. Application/Port server thread

The server thread contains some very simple code. After starting the ORB/POA (line 12) the server will parse the command line and populate the sharedMem data structure with pointers to the various SCA-required arguments. In lines 26 through 33 the server creates the Port and Talk objects. The sharedMem's `is Created` flag is updated to let the main program know that it is now safe to bind the application to the Name Service. The blocking call to fire up the ORB is made in line 37 and will not return until a signal is received. A SIGHUP is actually sent by the Core Framework whereas in the terminal device the SIGHUP was generated within the releaseObject() skelimpl. Line 40 sets the TerminalPort object reference to NIL; this will tell the main program to break out of the while-forever loop and shutdown. And that is all the software for the our Talk application. Just two items remain: generation of the XML and some more upgrades to the SCARI Core Framework for minimum CORBA.

19.5.5 Talk Application XML

The centrepiece of the SCA-compliant application is the SAD given in Figure 19.22.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE softwareassembly SYSTEM "dtd/softwareassembly.2.2.dtd">
<softwareassembly id="DCE:2037fd5d-41f5-4707-902d-a6a4210e331c"
                  name="Talk0">
  <componentfiles>
    <componentfile id="TalkAssemblyControllerFile" type="SPD">
      <localfile name="TalkAssemblyController.spd.xml"/>
    </componentfile>
  </componentfiles>
  <partitioning>
    <componentplacement>
      <componentfileref refid="TalkAssemblyControllerFile"/>
      <componentinstantiation
                id="DCE:e9b7420a-5f94-4630-8642-3847bc3e6f91">
        <usagename>TalkController0</usagename>
        <findcomponent>
          <namingservice name="TalkController"/>
        </findcomponent>
      </componentinstantiation>
    </componentplacement>
  </partitioning>
  <assemblycontroller>
    <componentinstantiationref
              refid="DCE:e9b7420a-5f94-4630-8642-3847bc3e6f91"/>
  </assemblycontroller>
  <connections>
    <connectinterface id="TalkerToTerminalConnection">
      <usesport>
       <usesidentifier>FromTalker</usesidentifier>
      <findby>
        <namingservice name=
```

Figure 19.22. Talk application SAD

```
32.          "TalkController_DCE:e9b7420a-5f94-4630-8642-3847bc3e6f91"/>
33.          </findby>
34.        </usesport>
35.        <providesport>
36.          <providesidentifier>ToTerm</providesidentifier>
37.          <findby>
38.          <namingservice name=
39.          "TerminalWindow_DCE:fe994973-b25a-4aa4-95e6-9b58555c39bc"/>
40.          </findby>
41.        </providesport>
42.      </connectinterface>
43.    </connections>
44. </softwareassembly>
```

Figure 19.22. (Continued)

As before the SAD provides a linkage to each component's SPD (line 7). In lines 6 and 12 are a link between the `componentfile` and the `componentinstantiation`. They must match. In lines 14 and 17, the actual name used to bind the application to the real-time Name Context is the concatenation of these two attributes. In line 24 we identify our single component (line 14) as the assembly controller. Finally, we identify our single connection. The connection itself is given a name (line 27). A single uses port (lines 28–34) and provides port (lines 35–41) are identified. The naming convention for these ports is exactly as given in the initial application block diagram (see Figure 19.8). As required by the SCARI Core Framework, both Port object references must be accessible through the Name Service. These names are identified in lines 32 and 39. Since *Device* and *Application* components are registered in different Name Contexts, the Core Framework will need to search the name tree for the desired connection component. Note also that the port name given in line 36 must match the port name given in line 23 of the Terminal device's SCD (see Figure 19.16). Next the mandatory SPD is given in Figure 19.23.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE softpkg SYSTEM "dtd/softpkg.2.2.dtd">
3. <softpkg id="DCE:8b99d9e6-9ea3-426b-9d42-5e8849eea95b"
4.           name="talkerResource" type="sca_compliant">
5. <author></author>
6. <descriptor>
7.    <localfile name="TalkAssemblyController.scd.xml"/>
8. </descriptor>
9. <implementation id="DCE:5ad12078-79f4-4068-a37f-96d4417797b9">
10.   <description>"C" program sends Hello to Terminal</description>
11.   <propertyfile type="PRF">
12.      <localfile name="TalkAssemblyController.prf.xml"/>
13.   </propertyfile>
14.   <code type="Executable">
15.      <localfile name="/wileyApp"/>
16.   </code>
```

Figure 19.23. Talk application SPD

```
17.   <runtime name="ELF_executable" version="1"/>
18.   <os name="Linux"/>
19.   <processor name="x86"/>
20.   <dependency type="mips_allocation">
21.      <propertyref refid="DCE:F364A630-5F0E-11d4-8164-00508B6A52E6"
22.                   value="2"/>
23.   </dependency>
24.   <dependency type="memory_allocation">
25.      <propertyref refid="DCE:F364A630-5F0E-11d4-8164-00508B6A52E6"
26.                   value="200"/>
27.  </dependency>
28. </implementation>
29. </softpkg>
```

Figure 19.23. (Continued)

Two additional XML files are identified in the SPD. These are the SCD (line 7) and the Properties file (line 12). The implementation filename is identified in line 15. Its runtime is identified as an 'ELF_executable'. A previous example of run time was 'Xterm'. An ELF_executable will not have a window associated with it, but will run as a background process. Finally, since the application is required to run on ExecutableDevice the two required dependencies – `mips_allocation` and `memory_allocation` – are identified by UUID in lines 21 and 25. These UUIDs must match the allocation properties of UUIDs given in the ExecutableDevice's PRF file. This is not unlike the same matching that had to be performed for the non-SCA-compliant version of the Hello World program. The PRF file for this implementation is exactly the same as that for the terminal device (see Figure 19.15). These properties are simply command line options that are passed to the ORB. Figure 19.24 presents the application's SCD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE softwarecomponent SYSTEM "dtd/softwarecomponent.2.2.dtd">
<softwarecomponent>
  <corbaversion>2.2</corbaversion>
  <componentrepid repid="IDL:CF/Resource:1.0"/>
  <componenttype>resource</componenttype>
  <componentfeatures>
     <supportsinterface repid="IDL:CF/PortSupplier:1.0"
                        supportsname="PortSupplier"/>
     <supportsinterface repid="IDL:CF/LifeCycle:1.0"
                        supportsname="LifeCycle"/>
     <supportsinterface repid="IDL:CF/PropertySet:1.0"
                        supportsname="PropertySet"/>
     <supportsinterface repid="IDL:CF/TestableObject:1.0"
                        supportsname="TestableObject"/>
     <ports>
        <uses repid="IDL:CF/Port:1.0" usesname="FromTalker">
```

Figure 19.24. Talk application SCD

```
18.          <porttype type="data"/>
19.       </uses>
20.    </ports>
21.  </componentfeatures>
22. <interfaces>
23.    <interface repid="IDL:CF/Port:1.0" name="FromTalker"/>
24.    <interface repid="IDL:CF/Resource:1.0" name="Resource">
25.      <inheritsinterface repid="IDL:CF/PortSupplier:1.0"/>
26.      <inheritsinterface repid="IDL:CF/LifeCycle:1.0"/>
27.      <inheritsinterface repid="IDL:CF/PropertySet:1.0"/>
28.      <inheritsinterface repid="IDL:CF/TestableObject:1.0"/>
29.    </interface>
30.    <interface repid="IDL:CF/PortSupplier:1.0" name="PortSupplier"/>
31.    <interface repid="IDL:CF/LifeCycle:1.0" name="LifeCycle"/>
32.    <interface repid="IDL:CF/PropertySet:1.0" name="PropertySet"/>
33.    <interface repid="IDL:CF/TestableObject:1.0"
34.                name="TestableObject"/>
35. </interfaces>
36.</softwarecomponent>
```

Figure 19.24. (Continued)

The important part of the SCD is the identification of the uses port in lines 16–20. The name *usesname* in line 17 needs to match line 29 of the SAD (Figure 19.22). This completes all the code that is required for the Talk application. Once compiled and linked it will be necessary to create a sub-directory for the application files. This sub-directory needs to be created under `$SCA_HOME/demosources/Node1/profile`. For consistency with HMI code yet to be provided, the directory should be named Talker. The application should be named as specified in line 15 of the SPD; `wileyApp`. We need one final HMI application to allow us to install, create, start, stop and release our application. Figure 19.25 provides such a code.

```
static CosNaming_NameComponent allObjs[7][2] = {
    {"SCARI_DM",""},{"DomainManager",""} };

int main( int argc, char * argv[] )
{
  CORBA_Environment ev;
  CORBA_ORB orb;
  CosNaming_NamingContext myNC=CORBA_OBJECT_NIL;
  CosNaming_Name myBindingName;

   SCA_CF_DomainManager dmObj=CORBA_OBJECT_NIL;

   CORBA_string aString="Talk0", bString;
   CORBA_string cString="/Talker/Talk0.sad.xml";
```

Figure 19.25. Talk application control code

```
CORBA_string dString="rtName";
SCA_CF_DomainManager_ApplicationFactorySequence *afSeq;
SCA_CF_ApplicationFactory anAppFact;

SCA_CF_DomainManager_ApplicationSequence* aSeq;
SCA_CF_Application ourApp;

SCA_CF_DeviceAssignmentSequence* dasSeq;
SCA_CF_Properties* initProp;
int i;

CORBA_exception_init(&ev);
startOrb(&argc,argv,&orb,&ev);

/* initialize some empty sequences for create */
dasSeq = SCA_CF_DeviceAssignmentSequence__alloc();
(*dasSeq)._length = 0;
initProp = SCA_CF_Properties__alloc();
(*initProp)._length=0;

/* Locate the Domain Manager */
myNC = CORBA_ORB_resolve_initial_references(
      orb,"NameService",&ev);

myBindingName._length=2;
myBindingName._buffer= &(allObjs[0][0]);

dmObj=(SCA_CF_DomainManager)
         CosNaming_NamingContext_resolve(myNC,
           (const CosNaming_Name *)&myBindingName, &ev);

/* Parse the command: install, create, release, start, stop */
if (strcmp(argv[1],"create")==0) {

printf("create Application %s, name = %s\n",aString,dString);

 /* Locate the correct appFactory */
  afSeq = SCA_CF_DomainManager__get_applicationFactories
            (dmObj, &ev);
  for (i=0; i<(*afSeq)._length; ++i) {
    anAppFact=(SCA_CF_ApplicationFactory)*((*afSeq)._buffer+i);
    bString = SCA_CF_ApplicationFactory__get_name(anAppFact,&ev);
    if (strcmp(aString,bString)==0) break;
    CORBA_free(bString);
  }
  if (i!=(*afSeq)._length) {
    printf("Found %s appFactory\n",bString);
   CORBA_free(bString);
```

Figure 19.25. (Continued)

```
63.      }
64.      ourApp = SCA_CF_ApplicationFactory_create(anAppFact, dString,
65.               (const SCA_CF_Properties*)initProp,
66.               (const SCA_CF_DeviceAssignmentSequence*)dasSeq, &ev);
67.    return 0;
68.    } else if (strcmp(argv[1],"install")==0) {
69.
70.      printf("install Application, profile = %s\n",cString);
71.      SCA_CF_DomainManager_installApplication(dmObj,cString,&ev);
72.      return 0;
73.
74.    } else { /* remaining commands require the Application IOR */
75.        aSeq = SCA_CF_DomainManager__get_applications(dmObj, &ev);
76.        printf("There are %d applications\n",(*aSeq)._length);
77.        for (i=0; i<(*aSeq)._length; ++i) {
78.           ourApp = (SCA_CF_Application) *((*aSeq)._buffer+i);
79.           bString = SCA_CF_Application__get_name(ourApp, &ev);
80.           if (strcmp(bString,dString)) break;
81.        }
82.        printf("%s application located\n",dString);
83.     }
84.
85.     if (strcmp(argv[1],"release")==0) {
86.       SCA_CF_LifeCycle_releaseObject((SCA_CF_LifeCycle)ourApp,&ev);
87.       printf("releaseObject OK\n");
88.    }
89.    else if (strcmp(argv[1],"start")==0) {
90.       SCA_CF_Resource_start((SCA_CF_Resource)ourApp,&ev);
91.       printf("start() OK\n");
92.    }
93.    else if (strcmp(argv[1],"stop")==0) {
94.       SCA_CF_Resource_stop((SCA_CF_Resource)ourApp,&ev);
95.       printf("stop() OK\n");
96.    }
97.    else {
98.       printf("Unknown command, no action performed\n");
99.    }
100.
101.    return 0;
102. }
```

Figure 19.25. (Continued)

The control application hard codes a couple of names (lines 13–15). The reader might want to make these mutable by making them command line parameters. The first name 'Talk0' is the name of the ApplicationFactory object. Line 14 provides the full pathname to the SAD file relative to the node's base directory. Line 15 is the name we provide in the run-time as a parameter for the `create()` operation. The Core Framework will use this name to create a unique naming context for the application's components. In the command line the user must supply the corbaloc for the Name Service as well as a command. Acceptable commands are install, create, start, stop, or release. The control application uses the Domain Manager to `install()` the ApplicationFactory (lines 68 through 72). The application files including the executable must already be located in the Talker sub-directory because the Core Framework will validate the XML and the presence of the executable as part of the install procedure.

Next an Application can be created. This is done in lines 52 through 67. It is first necessary to find the correct Application Factory. When located it is simple to run the `create()` operation which then returns an Application object. The remaining operations are on the Application itself – these include start, stop, and release – lines 85–95. It is necessary to query the Domain Manager to locate the correct Application object reference. It is not correct to use the Name Service to get to the Application. This is because that object reference refers only to the assembly controller component. There is a part of the Talk application that lives inside the Core Framework: Consider it the parent of the assembly controller. If you send a `releaseObject()` directly to the child assembly controller, who is going to notify the Core Framework to also release? At a minimum the Core Framework has to remove the Application's object reference from the DomainManager's readonly attribute ApplicationSequence. Because of its readonly modifier, this data structure is accessible only from within the Core Framework's Java code. It cannot be accessed through CORBA. Thus all SCA operations on the Application need to go through the Core Framework; a user should never go directly to the assembly controller because it will corrupt state information held internally by the Core Framework.

19.5.6 Modifications for Minimum CORBA Compliance

The reader is now free to install the Talk application. As long as the XML is syntactically correct and files are located where they are supposed to be the installation should be painless. However we will run into problems when we attempt to `create()` our application. Inside the Core Framework there is a module called ConnectionHandler that takes care of calling `getPort()` and `connectPort()`. That Java code verifies the object type of the Port objects using the CORBA call `is_a()`. That call is not supported by minimum CORBA. Not being a compliance junky this would normally not be bothersome. However, the `is_a()` is not supported by ORBit and this could sabotage our efforts. Fortunately we can modify the Java code and just remove the calls. `ConnectionHandler.java` is located in the `$SCA_HOME/scasources/SCA/CFImpl` directory. The following instruction will remove the offending calls:

1. comment out lines 535 to 542;
2. comment out lines 547 to 554;
3. comment out lines 566 to 572;
4. comment out lines 435, 528, and 529.

We are not out of the woods yet. Any calls to `narrow()` also include an embedded call to `is_a()`. Therefore, each call to narrow needs to get commented out so we can substitute an appropriate code segment:

5. comment out line 559 and replace with:

```
if (sourceObj == null)
  sourceComponent = null;
else if (sourceObj instanceof SCA.CF.PortSupplier)
  sourceComponent = (SCA.CF.PortSupplier)sourceObj;
else
{ org.omg.CORBA.portable.Delegate delegate =
     ((org.omg.CORBA.portable.ObjectImpl)sourceObj)._get_delegate();
  SCA.CF._PortSupplierStub PSstub = new SCA.CF._PortSupplierStub();
  PSstub._set_delegate(delegate);
  sourceComponent = (SCA.CF.PortSupplier)PSstub;
}
```

6. with the 11 lines added, comment out line 584 and replace with:

```
if (sourcePortObj == null)
  sourceComponentPort = null;
else if (sourcePortObj instanceof SCA.CF.Port)
  sourceComponentPort = (SCA.CF.Port)sourcePortObj;
else
{ org.omg.CORBA.portable.Delegate delegate =
   ((org.omg.CORBA.portable.ObjectImpl)sourcePortObj)._get_delegate();
  SCA.CF._PortStub stub = new SCA.CF._PortStub();
  stub._set_delegate(delegate);
  sourceComponentPort = (SCA.CF.Port)stub;
}
```

7. with the 22 lines added, comment out line 611 and 612 and replace with:

```
if (destinationComponent == null)
  destinationPortSupplier = null;
else if (destinationComponent instanceof SCA.CF.PortSupplier)
  destinationPortSupplier = (SCA.CF.PortSupplier)destinationComponent;
else
{ org.omg.CORBA.portable.Delegate delegate = ((
org.omg.CORBA.portable.ObjectImpl)destinationComponent)._get_
        delegate();
  SCA.CF._PortSupplierStub stub = new SCA.CF._PortSupplierStub ();
  stub._set_delegate(delegate);
  destinationPortSupplier = (SCA.CF.PortSupplier)stub;
}
```

That's it. Re-make the directory and re-make the $SCA_HOME/libs directory and the new Core Framework should effortlessly create the Talk application. The reader is free to send start, stop, or release commands and enjoy the show.

19.5.7 Concluding Remarks

A simple 'C' Hello World program takes the novice programmer maybe half an hour to get working. For an experienced programmer, it would take five minutes. The SCA-compliant Hello World program took over a month. The experience was enhanced by spending many, many hours getting the ORB and Core Framework to run the application to completion.

For the ORB a single flag needed to be cleared on the command line – the problem was solved in 30 seconds, the research to isolate the problem took three days. The same applied to the Core Framework. The result is a great work that is very stable and only needed minor code patches to run non-Java applications on a minimum ORB. Again the time to isolate the issues was extraordinary compared to the actual time to put in the fixes. In retrospect one-third of the time was writing code and experiencing the hard way that, for instance, all devices must have at least one allocation property. Then two-thirds of the time was chasing issues in the infrastructure and learning about multi-threaded servants.

This experience is not unlike stories heard from the front line – that is, from the companies that are building SCA-compliant radios. Embedded software engineers who know CORBA and XML just didn't need to exist prior to SCA. Now a project cannot be expected to succeed without that talent base. Companies are encouraged to consider sub-contracting SCA work to an experienced vendor. The cost of building such a talent base internally is prohibitive and likely to be a schedule breaker.

Appendix A
Mandatory POSIX Calls

All 256 POSIX system calls deemed as mandatory by the SCA

abort	getcwd	pthread_attr_setstacksize	sem_trywait
abs	gets	pthread_cancel	sem_unlink
access	gmtime	pthread_cleanup_pop	sen_wait
acos	gmtime_r	pthread_cleanup_push	setbuf
aio_cancel	isalnum	pthread_cond_broadcast	setjmp
aio_error	isalpha	pthread_cond_destroy	setlocale
aio_fsync	iscntrl	pthread_cond_init	sigaction
aio_read	isdigit	pthread_cond_signal	sigaddset
aio_return	isgraph	pthread_cond_timedwait	sigdelset
aio_suspend	islower	pthread_cond_wait	sigemptyset
aio_write	isprint	pthread_create	sigfillset
asctime	ispunct	pthread_detach	sigignore
asctime_r	isspace	pthread_equal	siginterrupt
asin	isupper	pthread_exit	sigismember
atan	isxdigit	pthread_getschedparam	signal
atan2	kill	pthread_getspecific	sigpending
atof	ldexp	pthread_join	sigprocmask
atoi	link	pthread_key_create	sigqueue
atol	lio_listio	pthread_key_delete	sigset
bsearch	localtime	pthread_kill	sigstack
calloc	localtime_r	pthread_mutex_destroy	sigsupend
ceil	log	pthread_mutex_getprioceiling	sigtimedwait
chdir	log10	pthread_mutex_init	sigwait
clearerr	longjmp	pthread_mutex_lock	sigwaitinfo
clock	Lseek	pthread_mutex_setprioceiling	sin
clock_getres	malloc	pthread_mutex_trylock	sinh
clock_gettime	mkdir	pthread_mutex_unlock	sprintf
clock_settime	mktime	pthread_mutexattr_destroy	sqrt
close	mlock	pthread_mutexattr_getprioceiling	srand
closedir	mlockall	pthread_mutexattr_getprotocol	sscanf

cos	modf	pthread_mutexattr_init	stat
cosh	mq_close	pthread_mutexattr_ setprioceiling	strcat
creat	mq_getattr	pthread_mutexattr_ setprotocol	strchr
ctime	mq_notify	pthread_once	strcmp
ctime_r	mq_open	pthread_self	strcpy
difftime	mq_receive	pthread_setcancelstate	strcspn
exp	mq_send	pthread_setcanceltype	strftime
fabs	mq_setattr	pthread_setschedparam	strlen
fclose	mq_unlink	pthread_setspecific	strncat
fdopen	munlock	pthread_sigmask	strncmp
feof	munlockall	pthread_testcancel	strncpy
ferror	nanosleep	putc	strpbkr
fflush	open	putchar	strrchr
fgetc	opendir	puts	strspn
fgets	pathconf	qsort	strstr
fileno	pause	raise	strtok
floor	perror	rand	strtok_r
fmod	pow	rand_r	tan
fopen	printf	read	tanh
fpathconf	pthread_attr_destroy	readdir	time
fprintf	pthread_attr_getdetachstate	realloc	time
fputc	pthread_attr_getinheritsched	remove	timer_create
fputs	pthread_attr_getschedparam	rename	timer_delete
fread	pthread_attr_getschedpolicy	rewind	timer_getoverrun
free	pthread_attr_getscope	rewinddir	timer_gettime
freopen	pthread_attr_getstackaddr	rmdir	timer_settime
frexp	pthread_attr_getstacksize	scanf	tmpfile
fscanf	pthread_attr_init	sched_yield	tmpnam
fseek	pthread_attr_setdetachstate	sem_close	tolower
fstat	pthread_attr_setinheritsched	sem_destroy	toupper
ftell	pthread_attr_setschedparam	sem_getvalue	ungetc
fwrite	pthread_attr_setschedpolicy	sem_init	unlink
getc	pthread_attr_setscope	sem_open	utime
getchar	pthread_attr_setstackaddr	sem_post	write

Appendix B
References to Part III

[1] minimumCORBA, OMG TC Document orbos/98-05-13, Object Management Group, http://www.omg.org/, 19 May 1998

[2] CAE Specification System Interfaces and Headers, Issue 5: Volume 1, The Open Group, UK, February 1997

[3] The Single UNIX Specification, Version 3, A White Paper from the Open Group, The Open Group, UK, May 2003

[4] International Standard, Programming languages – C, ISO/IEC 9899:1999, International Organization for Standardization/International Electrotechnical Commision, Switzerland

[5] The Austin Group Home Page, http://www.opengroup.org/austin

[6] POSIX.4: Programming for the Real World, Bill O. Gallmeister, O' Reilly & Associates January 1995, p. 114

[7] Against Priority Inheritance, Victor Yodaiken, Finite State Machine Labs (FSMLabs), 9 July 2002

[8] http://www.software.org/quagmire/descriptions/rtcado-178b.asp

[9] Pthreads Programming, Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, O'Reilly & Associates, 1996, pp. 18–19

[10] Federal Information Processing Standards (FIPS) 151-2, National Institute of Standards (NIST) – ISO/IEC 9945-1:1990, International Organization for Standardization/International Electrotechnical Commision, 1990

[11] POSIX Programmers Guide: Writing Portable UNIX Programs, Donald Lewine, O'Reilly & Associates, 1991, pp. 152, 157, 166, 401, 408

[12] Programming with POSIX Threads, David R. Butenhof, Addison-Wesley, 1997, pp. 163–172

[13] C Language Mapping Specification, OMG TC Document, formal/06-xx-99, Object Management Group, http:// www.omg.org/, June 1999

[14] PIM and PSM for Software Radio Components Final Adopted Specification, dtc/04-05-04, Object Management Group, May 2004

[15] Object Interconnections, An Overview of the OMG CORBA Messaging Quality of Service Framework, Douglas C. Schmidt and Steve Vinoski, *C++ Report* Magazine, March 2000

[16] Minimum CORBA version 1.0, formal/02-08-01, Object Management Group, August 2002

[17] Orbacus™ Users Guide, version 4.3.1, IONA Technologies PLC, 7 Feb 2006, pp. 354–359

[18] Distributed Object Computing for Distributed Real-time and Embedded Systems, http://www.dre.vanderbilt.edu/
[19] Objective Interface Systems Home Page, http://www.ois.com/
[20] ORBit2 Home Page, © GNOME Foundation, http://www.gnome.org/projects/ORBit2/index.html
[21] MICO ORB Home Page, http://www.mico.org/
[22] omniORB Home Page, http://omniorb.sourceforge.net/
[23] Event Service Specification, OMG TC Document, formal/01-03-01: EventService, v1.1, Object Management Group, http://www.omg.org/, March 2001
[24] Lightweight Logging RFC, OMG TC Document, realtime / 02-06-14, Object Management Group, http:// www.omg.org/, June 2002
[25] Lightweight Log Service Specification v 1.0, OMG TC Document, formal/03-11-03, Object Management Group, http://www.omg.org/, November 2003
[26] Advanced CORBA Programming with C++, Michi Henning and Steve Vinoski, Addison-Wesley, 1999, pp. 771–810
[27] Distributed Computing Environment (DCE) Univeral Unique Identifier (UUID) 1.1 Remote Procedure Call, Open Software Foundation, 1994
[28] The Common Object Request Broker: Architecture and Specification, OMG TC Document formal/98-07-01, Object Management Group, http://www.omg.org/, February 1998
[29] ORBit2 Documentation, Ewan Birney, Michael Lausch, Todd Lewis, Stéphane Genaud and Frank Rehberger, http://www.gnome.org/projects/ORBit2/documentation.html

Index