Software Defined Radio Forum (SDRF)

SDR FORUM

TECHNICAL REPORT 2.1

Architecture and Elements of Software Defined Radio Systems as Related to Standards

SDR Forum
PO Box 1236
Rome New York 13442-1236

email: info@sdrforum.org

NOTICE AND WARNING TO MEMBERS REGARDING AREAS OF DISCUSSION AMONG MEMBERS

The SDR Forum is an organization whose members include direct competitors, Government and private industry, and suppliers and purchasers of goods and services. Certain communications between and among such parties could give rise to allegations of anti-competitive conduct under US antitrust laws. A meeting such as this will present many situations where such contacts or communications could arise, even if unintentional. Because the opportunity for such anti-competitive conduct is presented, it is important for all members to avoid even the appearance of such conduct.

Therefore, it is the policy of the SDR Forum to prohibit any discussion of:

- pricing (including discounts or terms and conditions),
- market shares of individual competitors,
- exclusion of any competitors,
- cross licensing,
- marketing policies or practices, particularly restrictions on customers, territories or markets,
- preferential pricing or sales terms,
- contract bidding,
- any particular job, bid, contract, or competitive situation,

or other potentially anti-competitive subject matter, during or in connection with member meetings, steering committee meetings, or any SDR Forum activities. Do not initiate any such discussion. If someone does initiate such discussion in your presence, refuse to participate, and stop the discussion.

The following is a description of topics that should not give rise to the foregoing concerns. This is not intended to be an exhaustive list, but a guide to permissible areas of discussion. It is permissible to discuss overall market size and conditions; the identity and characteristics of participants in markets; standards, specifications, and technical matters relating to the industry or the technology; the economic aspects of standards or technologies, including the effects on the market of adopting a standard or competitive considerations with respect to other technologies or standards; lobbying and promotion of the technology; and the business of the SDR Forum.

If you have any questions about the Forum’s policy or about the prohibited topics of discussion, please contact the SDR Forum Chair who will authorize contact with the Forum’s counsel.

Copyright November 1999 SDR Forum

Page

Table of Contents i
Version Status xi
1.0 Introduction 1
2.0 Wireless Services and Applications Overview 1
2.1 Issues Facing the Wireless Industry 1
2.1.1 User's Problem 1
2.1.2 Commercial Carrier's Problem 1
2.1.3 Civil Government's Problem 2
2.1.4 Military's Problem 2
2.1.5 Manufacturers' Problem 2
2.1.6 Regulatory Agency's Problem 3
2.1.7 Semiconductor Vendor's Problem 4
2.1.8 Context Diagrams 5
2.2 Current Operational Environment 8
2.2.1 Terms Definitions 9
2.2.2 Service Parameter Tables 10
2.2.3 Requirements 16
2.2.3.1 Handheld Requirements 16
2.2.3.2 Mobile System Applications and Requirements 16
3.0 SDRF System Architecture 1
3.1 Architecture Frameworks 2
3.1.1 Functional Model 4
3.1.2 Interaction Diagram 8
3.2 Implementation Models 13
3.2.1 Handheld Models 14
3.2.2 Mobile Models 23
3.2.3 Base Station / Satellite Models 28
3.2.3.1 Candidate high level Use Cases 28
3.2.3.2 Smart Antenna and Base station 28
3.2.4 Switcher Downloader 30
3.2.5 Smart Antenna Definitions 33
3.3 SDRF-Compliant Interface Use 36
3.3.1 Summary 36
3.3.2 Why Do Users Need The SDRF Solution? 36
3.3.3 How to add functionality to an existing system 37
3.3.3.1 Design a whole new system 37
3.3.3.2 Modify an existing system in the laboratory 37
3.3.3.3 Provide field-installable new modules 38
3.3.4 The Compatibility Domain 38
3.3.5 Benefits 41
4.0 Application Program Interface (API) Design Guidelines 1
4.1 Structure for Development 1
4.1.1 Background 1
4.1.1.1 Specifying the system 1
4.1.1.2 Application Program Interface (API): 2
4.1.1.3 Software and Hardware Modules: 2
4.1.1.4 Visual Representation 3
4.1.2 What interfaces are required? 4
4.1.2.1 Tier 0 Architectural 4
4.1.2.2 Tier 1 Functional 7
4.1.2.3 Tier 2 Transport and Communication 8

4.1.2.4 Tier 3 Physical Factors 8
4.1.2.5 What do the Individual Tiers Offer? 8
4.1.3 What Makes a Good Interface? 9
4.1.3.1 A Functional Level Interface or API 9
4.1.3.2 A Transport Level Interface 9
4.1.3.3 A Physical Level Interface 9
4.1.3.4 What Must an Interface or API Not Do? 9
4.1.3.5 What an API should do? 10
4.1.4 Capability Exchange 14
4.1.4.1 Capability Exchange Implementations 14
4.1.5 Resource Management 16
4.1.5.1 Identifying the Need for Resource Management 17
4.1.6 Managing Multiple I/O 18
4.1.7 API Design Process 21
4.1.7.1 Introduction 21
4.1.7.2 Overview 21
4.1.7.3 The SDRF API Definition Process 22
4.1.7.4. The SDRF API Approval Process 24
4.1.8 References 24
4.2 Relationships to non-compliant Processes 25
4.2.1 API Relationship Diagram 25
4.2.1.1 SDRF APIs 25
4.2.1.2 Legacy APIs 26
4.2.1.3 Product Specifications 26
4.2.2 The wrapper 27
4.2.3 Conversion techniques 27
4.2.3.1 Translate 28
4.2.3.2 Simulate 29
4.2.3.3 Integrate 30
4.2.4 Trade-offs 31
4.2.5 Conclusions 32
4.3 Distributed Processing Environment 33
4.4. Message description for APIs 33
4.4.1 Introduction 33
4.4.2 Background 33
4.4.3.Messages 33
4.4.3.1 Message acknowledgment 35
4.4.4 Message Definitions 36
4.4.4.1 Message format 36
4.4.4.2 Parameter format 36
4.4.4.3 enable (*destination*) ← (*status*) 37
4.4.4.4 disable (*destination*) ← (*status*) 38
4.4.4.5 reset (*destination*) ← (*status*) 39
4.4.4.6 reset_to_default(*destination*) ← (*status*) 40
4.4.4.7 start (*destination*) ← (*status*) 41
4.4.4.8 stop (*destination*) ← (*status*) 42
4.4.4.9 set_config (destination, table_config_id, parametersÖ) ← (status) 43
4.4.4.10 select_config (*destination, table_config_id*) ← (status) 44
4.4.4.11 get_config (destination, table_config_id)← (status, parameters) 45
4.4.4.12 capability_exchange(destination, max_capability) ← (status, parameters) 46
4.4.4.13 get_status(*destination*) ← (*status*) 47
4.4.4.14 place_module (destination, info) ← (status, parameters) 48
4.4.5 Examples 50
4.4.5.1 The relationship between enable/disable and start/stop 50

4.4.5.2 Module initialization and disabling through the API 52
4.4.5.3 Module replacement through the API 53
5.0 Frameworks and Design Patterns 1
5.1 Handheld Framework Examples 5
5.2 Mobile Framework Examples 6
5.2.1 An Object-Oriented Framework 6
5.2.1.1 Overview 6
5.2.1.2 The SDRF Framework 7
5.2.1.3 Startup 8
5.2.1.4 Mobile System Application of SDRF Framework 9
5.2.1.5 Summary 10
5.2.2 Object Orientation and CORBA Illustration 11
5.2.2.1 Overview 11
5.2.2.2 Legacy Systems 11
5.2.2.3 Object Oriented Software 12
5.2.2.4 Interface Definition Language 15
5.2.2.5 Summary 17
5.2.3 Mobile Framework 19
5.2.3.1 Definitions and Guidelines 19
5.2.4 Software Architecture View 25
5.2.4.1 Introduction 25
5.2.4.2. Software Architecture Rationale 26
5.2.4.3. Middleware Selection Rationale 26
5.2.4.4 CORBA Timing Studies 28
5.2.4.5 Operating Environment Rationale 30
5.2.4.6 Core Framework Rationale 30
5.2.5 Functional View 31
5.2.5.1 SDR Software Reference Model 31
5.2.5.2 Transition from Functional Model to OO Model 31
5.2.6 Structural View 44
5.2.6.1 Open Multi-layered Structural Architecture 44
5.2.7 Logical View 49
5.2.7.1 Core Framework 49
5.2.8 Use Case View 82
5.2.8.1 Boot Up and Initialize Use Case 84
5.2.8.2. Send and Receive Communication Traffic Use Case 85
5.2.9 Summary of Core Framework Operations 89
5.2.10 Core Framework IDL 95
5.2.10.1 Core Framework IDL Listing 96
5.2.11 Other Reference Sources 126
5.3 Base Station Framework Examples 127
5.4 Satellite Framework Examples 128
6.0 Implementation Recommendation 1
6.1 Software Download 1
6.1.1 Introduction 1
6.1.2 Software Download Overview 1
6.1.2.1 Definition of Software Download 1
6.1.2.2 Areas of Application 1
6.1.2.3 Requirements for Software Download 2
6.1.2.4 Methods of Downloading Software 4
6.1.2.5 Download Implementation Issues 4
6.1.2.6 Standardization Issues 4
6.1.2.7 Regulation and Certification Issues 5
6.1.3 Software Download Scenarios 5

6.1.3.1 Handheld Architecture Download Scenarios 6
6.1.3.2 Mobile Architecture Download Scenarios 17
6.1.4 Preliminary API Messaging Requirements 19
6.1.4.1 Objectives 20
6.1.4.2 Download API Context 20
6.1.4.3 The Download Protocol Framework 21
6.1.4.4 API Framework 23
6.1.4.5 API Implications 35
6.1.4.6 New API Messages for Download 36
6.2 Mode switcher API example specification 37
6.2.1 Introduction 37
6.2.2 Requirements 37
6.2.3 Levels of detail 38
6.2.3.1 Regulatory 38
6.2.3.2 Non-regulatory 38
6.2.3.3 Detailed 39
6.2.4 Describing the configuration 39
6.2.5 API definitions 41
6.2.6 Regulatory capability exchange 42
6.2.7 Regulatory configuration 44
6.2.7.1 Approval coding 44
6.2.8 Non-regulatory capability exchange 46
6.2.9 Non-regulatory configuration 47
6.2.10 Detailed capability exchange 48
6.2.11 Detailed configuration 51
6.2.12 Mode Switcher Scenario Flow Charts 55
6.2.13 Power-on 55
6.2.13.1 Power-on Start 56
6.2.13.2 Is There a Default? 56
6.2.13.3 Standard Mode Switcher Set Up 56
6.2.13.4 Is There a Service? 56
6.2.13.5 Can The Configuration Be Changed? 57
6.2.13.6 Change Configuration 57
6.2.13.7 Choose Another Service? 57
6.2.13.8 Capability Exchange 57
6.2.13.9 Select Service 57
6.2.14 Download and Installation 58
6.2.15 Cross-technology Roaming 59
7.0 Form Factor 1
7.1 Handheld Form Factor 1
7.2 Mobile Form Factor 1
7.3 Interconnect Options 1
8.0 Plan for Future Work 1
8.1 Mobile Working Group Work Plan - 2000 3
8.2 Base Station Working Group Work plan - 2000 3
8.3 Handheld Working Group Work Plan – 2000 4
8.4 Switcher / Download Working Group Work Plan - 2000 4
8.5 Antenna API Task Group 4
9.0 Glossary 1
Appendix A. The SDRF Charter 1
Appendix B. List of Chairs and Co-chairs 1
Appendix C. Other Organizations Contacted by SDR Forum 1
Appendix D. Bus/Interconnect and Form Factor Technologies 1
D.1 EISA 1

D.2 PCI Local Bus 1
D.3 PERSONAL COMPUTER MEMORY CARD INDUSTRY ASSOCIATION (PCMCIA) 6
D.4 The VMEBUS Backplane 6
D.5 VME64 8
D.6 VME64 Extensions 9
D.7 VME320 10
D.8 SEM-E on VME 10
D.9 VMEbus P2 Sub-bus Data Transfer Architectures 11
D.9.1 RACEway 11
D.9.2 VSB 11
D.9.3 Skychannel 12
D.9.4 SCSA 12
D.10 Backbone or System Bus Structures 13
D.10.1 MIL-STD-1553B 13
D.10.2 MIL-STD-1773A 14
D.10.3 FDDI 14
D.10.4 N-ISDN 14
D.10.5 ATM 15
D.10.6 FIBREChannel 16
D.10.7 FireWire 16
Appendix E. Members of the Software Defined Radio Forum 1

LIST OF FIGURES .... PAGE

FIGURE 1.0-1 SDR FORUM STANDARDS DEVELOPMENT PROCESS FLOW 2
FIGURE 1.0-2 GENERIC REPRESENTATION OF FUNCTIONAL MODULAR LEVEL STANDARDIZATION 3
FIGURE 1.0-3 SOME REPRESENTATIVE SERVICES FOR CONSIDERATION FOR INCLUSION IN SDRF OPEN ARCHITECTURE 6
FIGURE 2.1-1 NETWORK CONTEXT DIAGRAM: CELLULAR/PCS 5
FIGURE 2.1-2 NETWORK CONTEXT DIAGRAM: CIVIL/AVIATION 6
FIGURE 2.1-3 NETWORK CONTEXT DIAGRAM: DEFENSE APPLICATIONS 7
FIGURE 2.2.2-1 U.S. SPECTRUM ALLOCATION 10
FIGURE 2.2.2-2 JAPAN SPECTRUM ALLOCATION 11
FIGURE 2.2.2-3 SAMPLE EUROPEAN SPECTRUM ALLOCATION 11
FIGURE 3.0-1 SCOPE OF SDRF FORUM ARCHITECTURE WORK 1
FIGURE 3.1.1-1 SDRF HIGH-LEVEL FUNCTION MODEL 4
FIGURE 3.1.1-2 SDRF ARCHITECTURE EVOLUTION PROCESS 6
FIGURE 3.1.1-3 ONE COMMON SDRF FUNCTIONAL ARCHITECTURE MAPS TO HANDHELD, MOBILE, AND BASESTATION RADIO CONFIGURATIONS 6
FIGURE 3.1.1-4 AN EXAMPLE IMPLEMENTATION OF SDRF SOFTWARE AND HARDWARE OPEN ARCHITECTURE 7
FIGURE 3.1.1-5 SDRF FUNCTIONAL INTERFACE DIAGRAM 8
FIGURE 3.1.2-1 INTERFACE/INTERACTION DIAGRAM 10
FIGURE 3.2.1-1 SINGLE-BAND, SINGLE-MODE HANDHELD FUNCTIONAL MODEL 14
FIGURE 3.2.1-2 SDRF MAPPING INTO SINGLE-MODE, SINGLE-BAND HANDHELD FUNCTIONAL MODEL 15
FIGURE 3.2.1-3 MULTIMODE, MULTIBAND SOLUTION USING MULTIPLE SINGLE STANDARD DEVICES 16
FIGURE 3.2.1-4 MULTIBAND, MULTIMODE HANDHELD FUNCTIONAL MODEL 16
FIGURE 3.2.1-5 GENERIC PC HARDWARE/SOFTWARE ARCHITECTURE 17
FIGURE 3.2.1-6 HANDHELD MULTIPLE SERVICE MODEL 17
FIGURE 3.2.1-7 HANDHELD MULTIPLE SERVICE MODEL WITH PDA EXTENSION 18
FIGURE 3.2.1-8 WEARABLE MULTIPLE SERVICE MODEL WITH PDA EXTENSIONS 19
FIGURE 3.2.2-1 INFORMATION TRANSFER THREAD 25
FIGURE 3.2.2-2 MOBILE INFORMATION TRANSFER SYSTEM LOGICAL STRUCTURE 26
FIGURE 3.2.2-3 MULTIPLE INSTANTIATIONS OF EACH FUNCTION OF MODULAR, MULTIMODE OPERATION27
FIGURE 3.2.2-4 SDRF FUNCTIONAL MODEL MAPPED INTO A JOINT MARITIME COMMUNICATION STRATEGY (JMCOMS) APPLICATION 27
FIGURE 3.2.3-1 SIGNALING STRATEGY 32
FIGURE 3.2.3-2 CROSS STANDARDS HANDOFF 33
FIGURE 3.2.5-1. TYPE IA ANTENNA. 34
FIGURE 3.2.5-2. TYPE IB ANTENNA 34
FIGURE 3.2.5-3. TYPE II ANTENNA. 35
FIGURE 3.2.5-4. TYPE III ANTENNA. 35
FIGURE 3.3.4-1 THE COMPATIBILITY DOMAIN 38
FIGURE 3.3.4-2 PAST SYSTEMS 39
FIGURE 3.3.4-3 THE SDRF CONTRIBUTION 40
FIGURE 3.3.4-4 FUTURE SYSTEMS 41
FIGURE 4.1.1-1 THE CORRECT USE OF THE SYMBOLS FOR SOFTWARE INTERFACE AND MODULE IN A LAYER DIAGRAM 3
FIGURE 4.1.1-2 WRONG USE OF THE SOFTWARE INTERFACE AND MODULE SYMBOLS IN A LAYER DIAGRAM 4
FIGURE 4.1.2-1 SDRF FUNCTIONAL INTERFACE DIAGRAM 5
FIGURE 4.1.2-2 SDRF EXPANDING A TIER 0 MODULE TO CREATE THE TIER 1 FUNCTIONS FOR INFORMATION AND CONTROL 6
FIGURE 4.1.2.-3 SDRF ARCHITECTURE AND INTERFACE REFINEMENT USING TIERS 7
FIGURE 4.1.3-1 EXPANDING THE API GRANULARITY 11
FIGURE 4.1.3-2 USING INCREASED GRANULARITY TO ACCESS MULTIPLE TECHNOLOGY SOURCES 12

FIGURE 4.1.3-3 COMBINING MODULES AND REMOVING INTERFACES TO PROVIDE DIFFERENT SOLUTIONS. THE MESSAGE DIRECTIONS HAVE BEEN OMITTED FOR SIMPLICITY. 13
FIGURE 4.1.4-1 CAPABILITY EXCHANGE AND NEGOTIATION USING SPEC ID NUMBERS 15
FIGURE 4.1.5-1 THE TWO-INTO-ONE CONFLICT SCENARIO. NOTE THAT THE UPWARD FLOW HAS NOT BEEN SHOWN IN THE DIAGRAM 17
FIGURE 4.1.6-1 THE MULTIPLE CONTROL PATH PROBLEM. NOTE THAT THE UPWARD FLOW HAS NOT BEEN SHOWN IN THE DIAGRAM 19
FIGURE 4.1.6-2 USING A USER INTERFACE MANAGER. NOTE THAT THE UPWARD FLOW HAS NOT BEEN SHOWN IN THE DIAGRAM 20
FIGURE 4.1.7.2-1. CONTEXT FOR THE API DEVELOPMENT PROCESS 21
FIGURE 4.1.7.3-1 22
FIGURE 4.2.1-1 THE API RELATIONSHIP DIAGRAM 25
FIGURE 4.2.2-1 THE WRAPPER BETWEEN LEGACY AND SDRF APIS 27
FIGURE 4.2.3.1-1 TRANSLATE 28
FIGURE 4.2.3.2-1 SIMULATE 29
FIGURE 4.2.4-1 WRAPPER TRADE-OFFS 31
FIGURE 4.4.3-1 STATE LADDER DIAGRAMS SHOWING THE RELATIONSHIP BETWEEN STATUS WORDS RETURNED AS A RESULT OF RECEIVING A MESSAGE 35
FIGURE 4.4.4-1 MESSAGE DEFINITION STRUCTURE 36
FIGURE 4.4.5-1 THE RELATIONSHIP BETWEEN ENABLE/DISABLE/START/STOP MESSAGES 50
FIGURE 4.4.5-2 THE SCOPE OF THE SET_CONFIG AND SELECT_CONFIG MESSAGES 51
FIGURE 4.4.5-3 MODULE INITIALIZATION AND DISABLING THROUGH THE API 52
FIGURE 4.4.5-4 MODULE REPLACEMENT THROUGH THE API 54
FIGURE 5.0-1 USE OF VIEWS TO DESCRIBE A SYSTEM 2
FIGURE 5.0- 2 FRAMEWORK INTERACTION WITH VIEWS 4
FIGURE 5.2.1.2-1. SDRF FRAMEWORK OBJECTS 7
FIGURE 5.2.1.4-1. FRAMEWORK CONTROL AND APPLICATION OBJECTS 9
FIGURE 5.2.1.4-2. SDRF REFERENCE MODEL 10
FIGURE 5.2.2.2-1. MESSAGE PASSING 11
FIGURE 5.2.2.3-1. THE CORBA ENVIRONMENT 12
FIGURE 5.2.2.3-2. REMOTE METHOD INVOCATION 13
FIGURE 5.2.2.3-3. WRAPPER FOR LEGACY CODE 14
FIGURE 5.2.2.3-4. OBJECT RELOCATION 14
FIGURE 5.2.2.4-1 MESSAGE STRUCTURE EXAMPLE 15
FIGURE 5.2.3.1.1-1. THE SDR CORE FRAMEWORK (CF) 21
FIGURE 5.2.3.1.1-2. THE SDR SOFTWARE ARCHITECTURE 22
FIGURE 5.2.3.1.2-1. THE SDR OPERATING ENVIRONMENT (OE) 24
FIGURE 5.2.5.1-1 SDR SOFTWARE REFERENCE MODEL 32
FIGURE 5.2.5.2-1 CONCEPTUAL MODEL OF SDR NON-CORE APPLICATIONS 32
FIGURE 5.2.5.2-2. CONCEPTUAL MODEL OF SDR RESOURCES 34
FIGURE 5.2.5.2.1-1 CONCEPTUAL MODEL OF THE CORE FRAMEWORK (CF) 36
FIGURE 5.2.5.2.2-1 CONCEPTUAL MODEL OF SDR MODEM RESOURCES 37
FIGURE 5.2.5.2.3-1 CONCEPTUAL MODEL OF SDR NETWORKING RESOURCES 39
FIGURE 5.2.5.2.4-1 CONCEPTUAL MODEL OF SDR ACCESS RESOURCES 40
FIGURE 5.2.5.2.5-1 CONCEPTUAL MODEL OF SDR SECURITY RESOURCES 41
FIGURE 5.2.5.2.6-1 CONCEPTUAL MODEL OF SDR UTILITY RESOURCES 43
FIGURE 5.2.6.1-1. SDR SOFTWARE STRUCTURE 45
FIGURE 5.2.7.1-1. SDR CORE FRAMEWORK (CF) RELATIONSHIPS 50
FIGURE 5.2.7.1.1.1-1. *LIFECYCLE* RELATIONSHIPS 52
FIGURE 5.2.7.1.1.2-1 *STATEMANAGEMENT* RELATIONSHIPS 54
FIGURE 5.2.7.1.1.3-1 EXAMPLE OF CHAINED *RESOURCES* 56
FIGURE 5.2.7.1.1.3-2. *MESSAGEREGISTRATION* RELATIONSHIPS 57
FIGURE 5.2.7.1.1.4-1. *MESSAGE* RELATIONSHIPS 59
FIGURE 5.2.7.1.1.5-1. *RESOURCE* RELATIONSHIPS 62

FIGURE 5.2.7.1.2.1-1. DOMAIN MANAGEMENT 64
FIGURE 5.2.7.1.2.1-2. LAYERED *RESOURCE* ALLOCATION 64
FIGURE 5.2.7.1.2.1-3. *DOMAINMANAGER* RELATIONSHIPS 67
FIGURE 5.2.3.3.5-1. *RESOURCEMANAGERS* REPORT DEVICE PROPERTIES 69
FIGURE 5.2.7.1.2.2-2. *RESOURCEMANAGER* RELATIONSHIPS 70
FIGURE 5.2.7.1.3.1-1. *FILE* RELATIONSHIPS 72
FIGURE 5.27.1.3.2-1. CONCEPTUAL *FILESYSTEM* RELATIONSHIPS 73
FIGURE 5.2.7.1.3.2-2. *FILESYSTEM* RELATIONSHIPS 73
FIGURE 5.2.7.1.3.3-1. FILE MANAGEMENT 75
FIGURE 5.2.7.1.3.3-2. *FILEMANAGER* RELATIONSHIPS 76
FIGURE 5.2.7.1.3.5-1. *LOGGER* RELATIONSHIPS 78
FIGURE 5.2.7.1.4.1-1. *FACTORY* RELATIONSHIPS 81
FIGURE 5.2.7.1.4.2-1. EXAMPLE MESSAGE FLOWS WITH AND WITHOUT ADAPTERS 83
FIGURE 5.2.8-1. SDR USE CASES 83
FIGURE 5.2.8.1.1-1. CF POWER UP AND INITIALIZATION EXAMPLE SCENARIO 85
FIGURE 5.2.8.2.1-1. RECEIVE COMMUNICATIONS EXAMPLE SCENARIO 87
FIGURE 5.2.8.2.2-1. TRANSMIT COMMUNICATIONS EXAMPLE SCENARIO 88
FIGURE 5.2.10-1. CF CORBA MODULE 95
FIGURE 6.1.3-1: SOFTWARE DOWNLOAD FROM A SMARTCARD 10
FIGURE 6.1.3-2: OVER THE AIR SOFTWARE DOWNLOAD OF A SINGLE MODULE UPDATE 13
FIGURE 6.1.3-3: OVER THE AIR SOFTWARE DOWNLOAD OF A SET OF CONTROL, FUNCTIONAL, AND/OR PROTOCOL ENTITIES 16
FIGURE 6.1.3-4: OVER THE AIR SOFTWARE DOWNLOAD OF A SINGLE MODULE UPDATE (MOBILE) 19
FIGURE 6.1.4.2-1 LOCATION OF THE DOWNLOAD API 20
FIGURE 6.1.4.3-1: DOWNLOAD PROTOCOL 22
FIGURE 6.1.4.4.1-1: DOWNLOAD FLOWCHART EXAMPLE WITH EXAMPLE API MESSAGE 25
FIGURE 6.1.4.4.2-1: MESSAGING TO INITIATE DOWNLOAD 26
FIGURE 6.1.4.4.3-1 MESSAGING REQUIRED FOR AUTHENTICATION 27
FIGURE 6.1.4.4.4-1 MESSAGING REQUIRED FOR ENCRYPTION 29
FIGURE 6.1.4.4.6-1 DOWNLOAD ACCEPTANCE EXCHANGE MESSAGING 31
FIGURE 6.1.4.4.7-1 DOWNLOAD MESSAGING 33
FIGURE 6.1.4.5.1-1 HIERARCHICAL CAPABILITY TABLES 35
FIGURE 6.2.3-1 LEVELS OF DETAIL FOR SDRF DEVICE CAPABILITY AND CONFIGURATION TABLES 39
FIGURE 6.2.13-1 THE POWER-ON FLOW CHART 55
FIGURE 6.2.14-1 THE DOWNLOAD SCENARIO 58
FIGURE 6.2.15-1 THE ROAMING SCENARIO 59
FIGURE 8.0-1 STANDARDS RECOMMENDATIONS DEVELOPMENT OVERVIEW 1
FIGURE 8.0-2 RELATIONSHIP OF SDRF VERTICAL WORKING GROUPS TO HORIZONTAL TASK GROUPS 2

LIST OF TABLES .... PAGE

TABLE 1.0-1 AN OVERVIEW OF SDRF ROLE ……………………………………………………………………………. 1-5
TABLE 2.2.1-1 TERMS AND DEFINITIONS …………………………………………………………………………….. 2-9
TABLE 2.2.2-1 REPRESENTATIVE COMMERCIAL WIRELESS STANDARDS AND PARAMETERS …………… 2-13
TABLE 2.2.2-2 REPRESENTATIVE CIVIL WIRELESS STANDARDS AND PARAMETERS …………………………….. 2-14
TABLE 2.2.2-3 REPRESENTATIVE MILITARY WIRELESS STANDARDS AND PARAMETERS ……………………… 2-15
TABLE 3.1.1-1 SCOPE OF THE SDRF APPROACH TO OPEN SYSTEM STANDARDS RECOMMENDATIONS ……. 3-2
TABLE 3.1.2-1 INTERFACE MATRIX ………………………………………………………………………………………… 3-9
TABLE 3.1.2-2 INTERFACE/INTERACTION DIAGRAM INTERFACES AND EXAMPLE CONTENT ……………… 3-10
TABLE 3.2-1 SDRF DIFFERENCES BETWEEN HANDHELD AND MOBILE/STATIONARY SYSTEMS ……………. 3-13
TABLE 3.2.1-1 EXAMPLE FUNCTIONS IN HANDHELD FUNCTIONAL MODEL SUBSYSTEMS ……………………..3-21
TABLE D-1 SMALL PCI AND CARD BUS COMPARISON ………………………………………………………………… D-4
TABLE D-2 COMPARISON OF EXAMPLE PCI OPTIONS ………………………………………………………………… D-5

DISCLAIMER

This document is published by the SDR Forum, Inc. to provide information to the industry and to organizations involved in wireless communications and to open dialog and discussion to solicit information. The SDR Forum reserves the right at its sole discretion to revise this document for any reason.

The SDR Forum makes no representation or warranty, express or implied, with respect to the completeness, accuracy, or utility of the document or any information or opinion contained therein. Any use or reliance on the information or opinion is at the risk of the user, and the SDR Forum shall not be liable for any damage or injury incurred by any person arising out of the completeness, accuracy, or utility of any information or opinion contained in the document.

Nothing contained herein shall be construed to confer any license or right to any intellectual property, whether or not the use of any information herein necessarily utilizes such intellectual property.

This document does not constitute an endorsement of any product or company.

The SDR Forum, Inc. 1999
All Rights Reserved.
Printed November 1999

Version Status

TR V1- July 1997

The SDR Forum Technical Report Version 1 was published in July 1997. It contained a description of the concepts and basic architecture for software defined radio along with a description of the internal interfaces for such a radio.

TR V1.1- January 1998

Version 1.1 of the technical report includes the following modifications:
- Changes to the original architecture in order to better support security concerns,
- A description of the software download process,
- A definition of the software interfaces between modules—the APIs.

TR V1.2- July 1998

Version 1.2 of the technical report includes the following modifications:
- A description of the messages associates with APIs,
- A definition of the API messages needed for the software download process.

TR V2.0- December 1998

Version 2.0. of the technical report restructures the document to make it easier to navigate and understand. It adds more detail on API's, downloads, mobile and handset architecture, as well as initiating work in Base Station and Satellite areas.

TR V2.1- November 1999

Version 2.1 of the technical report adds sections on the mobile framework, base station architechture and smart antenna architecture.

1.0 Introduction

What is the SDR Forum?

The Software Defined Radio Forum (known as the Modular Multifunction Information Transfer System Forum prior to December 1998) is an open, non-profit corporation dedicated to supporting the development, deployment, and use of open architectures for advanced wireless systems.

Primary objectives of the Forum are:

- To enable seamless integration of capabilities across diverse networks, in an environment of multiple standards and solutions,
- To accelerate proliferation of software-definable radio systems,
- To advance adoption of open architectures for wireless systems,
- To promote "multiple capability and multiple mission" system flexibility, and
- To ensure accommodation of current and future user needs in the areas of voice, data, messaging, image, multimedia, etc.

Current Forum membership comprises an international mix of business and technical decision makers, planners, policy makers, and program managers from a broad range of organizations sharing a common view of advanced wireless networking systems evolution, including:

- Service Providers,
- Equipment Manufacturers,
- Component Manufacturers/Providers,
- Hardware and Software Developers,
- System Integrators,
- Government and Military,
- Standards Development Organizations,
- Industry Associations/Forums, and
- Academic and Research Organizations.

The SDR Forum Charter presented in Appendix A, delineates the vision, definition, and mission of the organization.

Standards Requirements/Recommendations Approach

The SDR Forum is pursuing its goals through the efforts of two core committees, the Markets Committee and the Technical Committee. The Markets Committee is chartered with promoting Forum activities and development of SDR-concept-based market forecast material. The Technical Committee is chartered with

specifications development. In addition, a Regulatory Advisory Committee is chartered with tracking and coordination concerning regulations relevant to SDR-concept-based system and product deployment.

The Forum seeks global harmonization of SDR concepts. Efforts are being coordinated with international industry associations, forums, and standards development organizations (SDOs). The Forum operates under a requirements, rather than technology-driven philosophy, although it is understood that the two areas must be considered jointly. Two broad categories of wireless issues, generally defined as end-user need and technical concern, are recognized. Software-definable radios (SDRs), which utilize conventional and innovative approaches to high-speed digital processing, are the underlying technology holding promise for addressing both issues.

The basic process followed by the Forum is to translate the end-user need into standards requirements/recommendations for action by SDOs. If the Forum cannot identify an SDO for the issue, the Forum will develop standards recommendations for release to the industry.

Figure 1.0-1 shows the basic process flow.

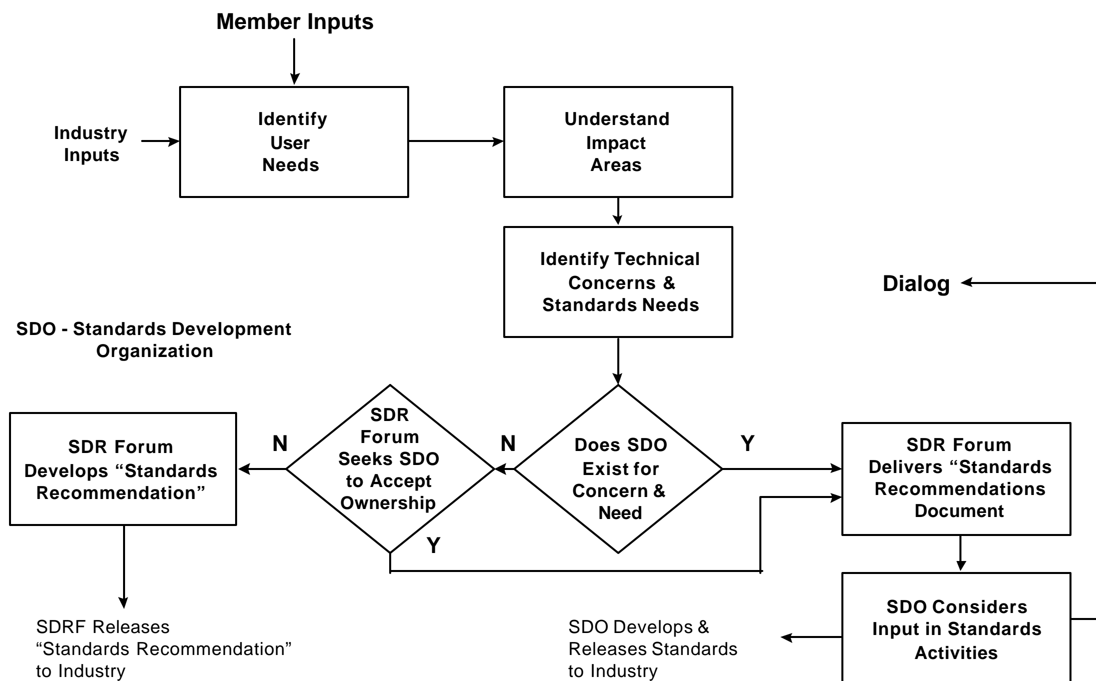


Figure 1.0-1 SDR Forum Standards Development Process Flow

Structure of this Document

Section 2 (Services and Applications) builds the background, top level conceptual descriptions, and functional requirements that form the basis for these standards requirements/ recommendations. Key to this approach is the modularization of the functions that reflect a balance between a coarse division of functional modularity that may be too general to achieve an open architecture and too granular a specification where every function is described.

Section 3 (Architecture) develops the general reference model for the SDRF architecture. The primary functional modules are the RF section, the modem, antenna, infosec, I/O, environment adaptation, and control. These functional modules may be interconnected to collectively function as a whole wireless unit. The standards requirements/recommendations that follow, take the form of specific requirements for the interface between functional modules and a description of the transfer characteristics of each module.

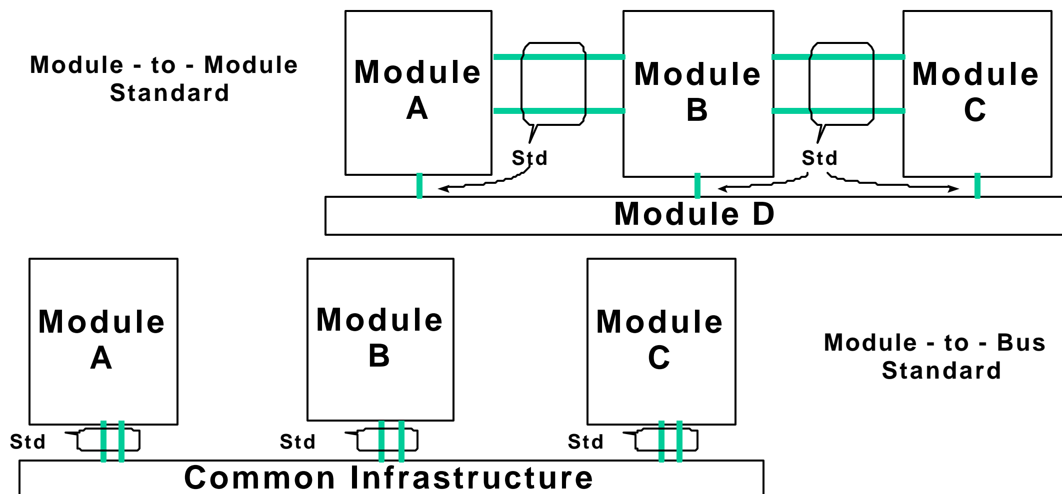


Figure 1.0-2 Generic Representation of Functional Modular Level Standardization

Figure 1.0-2 is a generic representation of a functional modular level standardization approach. The solution may either take the form of a module-to-module (hardware or software) interface, or a module-to-bus standard. The goal is to provide a common interface between modules without restricting and inhibiting the innovation that can be achieved within them. It is necessary that mandatory functions are provided and interface requirements met.

Section 3 also describes the interfaces for a module, separated into an information interface and a control interface. In both cases, these interfaces are bi-directional in nature. There is a separate standard for each of these two interfaces. An interface matrix, developed in a later section of this report identifies the modules and module-to-module elements that require some level of description or standardization. The modules themselves require a description of the functionality that must take place within that module but without specifying what methodology or technology must be used to accomplish it as long as there is compliance with the interface requirements. The module-to-module interfaces will require a standard format for the exchange of information as well as a standard format for the exchange of control.

Section 4 (Application Program Interface {API] Design Guidelines) addresses the development of API's to define SDRF standard devices and information concerning control messages. It provides a design guide, defines a set of generic control messages, and provides a generic framework for the definition of such API's. It also explores how legacy API's and SDRF API's work together.

Additionally, it provides an example of how control messages are used within the SDRF environment and gives specific examples for currently identified control messages. It also identifies some of the key problems, proposes some solutions, and describes the control messages that are needed to control and configure a module via its API.

Section 5 (Frameworks and Design Patterns) provides handheld, mobile, base station and satellite framework examples. . This section is new and will be further developed with future versions of this document. It describes the Software Defined Radio Forum architecture implemented with an object oriented approach, and in the base station section introduces a mechanism for an overall system design using system views as a basis for the framework.

Section 6 (Implementation Recommendation) addresses standards requirements/recommendations for software programmable, open architecture wireless hardware modules, and software download. It presents an overview of software download in the context of SDRF handheld and mobile devices and provides various download scenarios. It also addresses issues surrounding software download. It builds on the API design guide and the example APIs to provide more detailed implementation for a mode switcher function for a multi-function SDRF.

Preparing standards for the individual interfaces shown in Table 1.0-1 is much more complex than it appears. The intent of the SDRF is to specify an open architecture that will allow and support a wide range of services and protocols. As part of this effort, existing standards that require some extension to allow software programmable wireless modules to achieve full multi-band, multi-mode operation will be identified.

Table 1.0-1. An Overview of SDRF Role

Standards Requirement / Recommendation Type	SDRF Role
Air Interface	Support identified standards through common architectural partitioning Identify extensions to accommodate new SDRF capabilities
Internetworking	Support identified standards through common architectural partitioning Identify extensions to accommodate SDRF capabilities
API	Define
Software download	Define
Physical Interfaces	Select from existing open standards
Analog/RF Interconnects	Identify applicable standards and approaches and develop standards recommendations as appropriate
User Interface	None

SDRF will develop selected standards requirements/recommendations for software programmable wireless systems functional modules.

Figure 1.0-3 shows domestic and international private services, the mobile military systems, and civil government and aviation systems. It is the goal of this Forum to specify the module interfaces so they reflect the ability to be configurable and adaptable to many of these varied services.

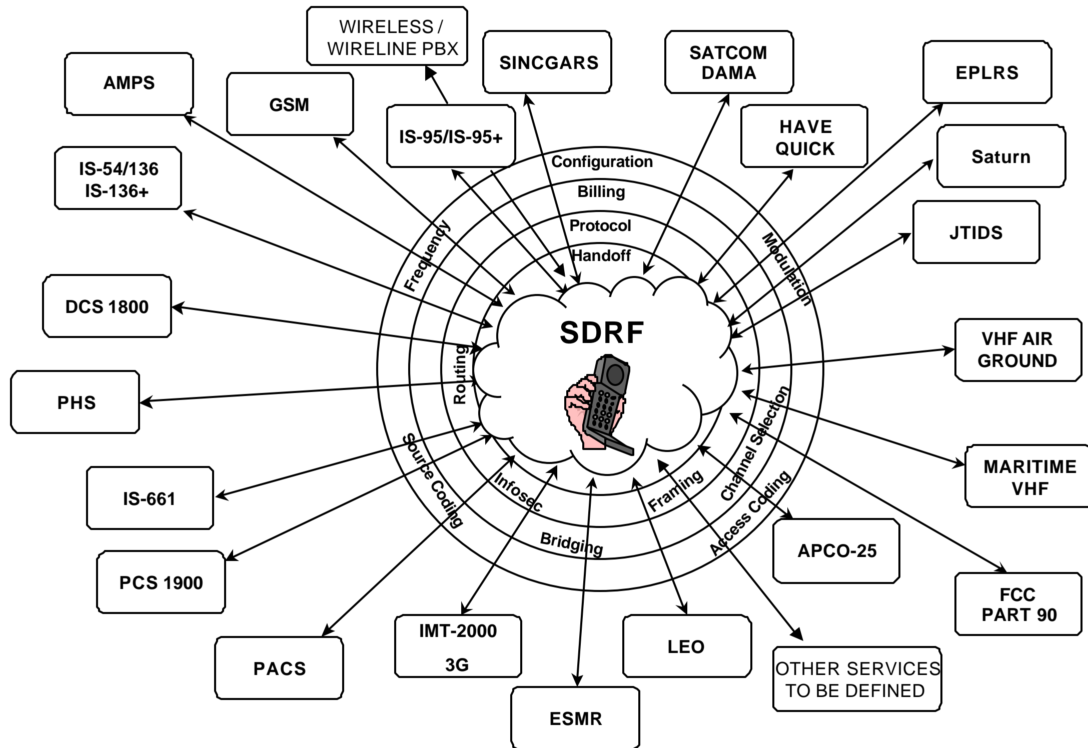


Figure 1.0-3 Some Representative Services for Consideration for Inclusion in SDRF Open Architecture

Section 7 addresses form factor.

2.0 Wireless Services and Applications Overview

This section reviews the typical usage application environments that provide the background for the technical analysis and standards recommendations that follow. In order to understand the scope and limits of SDR Forum standards recommendations focus, system context diagrams are presented that depict this information. The type of technical parameters that must be addressed when developing architectures are presented in the service parameter tables.

To help understand the range of applications that face SDRF, it is useful to look at some examples of problems faced by the various wireless communications users and suppliers in order to provide the motivation for a universal, software defined device to allow seamless use of a range of services.

2.1 Issues Facing the Wireless Industry

2.1.1 User's Problem

The user's problem is one of connectivity and information filtering. At their desks, users have email, telephones, personal computers, and wideband connectivity to internal backbones and external services. As they leave their offices, they have to rely on pagers for notification and cellular or PCS phones for contact. But both of these devices have limited access areas and specific protocols.

Users also have separate palmtop devices for information storage and display, devices incorporating substantial computational power. Software programmable radio technology offers an opportunity for users at home, in their offices, or on the road with an opportunity to have seamless connectivity with their data sources, and filtering capability so that they receive the information they need but are not overwhelmed by broadband data when operating in a narrowband environment.

2.1.2 Commercial Carrier's Problem

The general commercial problem is the need to integrate service portfolios. Carriers with multiple service types and multiple standards want to be able to integrate their service portfolio. Carriers with a single service, single technology strategy fear "bet the business technology decisions" and are being forced into multiple service portfolios by 1) Mergers and acquisitions, 2) International operations, and 3) International roaming. Similar problems exist in the Far East. In addition, in Europe and Japan, capacity problems are creating the need for multimode, multiband solutions.

North American PCS carriers have a unique historical problem: 1) PCS carriers cannot provide coverage in their own license areas initially, therefore, they must rely on AMPS for fill-in, 2) to provide nationwide roaming, PCS must have handsets for a wide range of PCS and cellular standards, and 3)

high volume PCS sign-ups depend on the availability of a single device that allows shrink wrap type distribution.

2.1.3 Civil Government's Problem

There is a need for emergency service agencies and law enforcement agencies to intercommunicate. Currently, city, state, and national agencies supporting a national emergency have multiple services and systems that cannot intercommunicate readily. Interagency communications usually requires an exchange of assets to support these temporary situations. Likewise, civil aviation requires a wide variety of communications and information transfer to support safe air travel and airport management. The civil aviation authorities would like to be able to upgrade systems in the field as new air interfaces, etc. are developed. This fosters the need for long life cycles of systems as well as "future proofing" to facilitate the expansion of the systems to take advantage of new technologies as they become available and to upgrade at a reasonable cost. The need for reducing the number and types of devices by utilizing adaptive technologies also exists.

2.1.4 Military's Problem

The different branches of the military, each with its own service, need to intercommunicate in a transparent way to realize the implementation of the future electronic battlefield. Currently, information transfer and communication compatibility between services is very limited and more than 200 defense radio procurement programs exist. Economies can be realized in the reduction of these radio procurement programs. The integration of information and communication systems could result in significant increases in battlefield efficiencies, the reduction of unnecessary personnel, and reduced budgets.

In addition, there is a need to communicate through public service infrastructures given the current geopolitical situations where the military could be required to operate in any foreign location in a time efficient way. The ability to mobilize and establish a communication environment requires that existing public services be utilized in a time expedient manner.

2.1.5 Manufacturers' Problem

Manufacturers are seeking ways to improve time to market, increase flexibility to add new services and features, reduce the number of fundamental designs, increase the production volume per design, simplify testing, and allow for upgradeability in the field. The flexibility associated with software defined radios and well structured interfaces that anticipate interface features required for new applications overlaid onto existing services, allow the equipment vendor to support customer feature requests for equipment that is already fielded. A reduced number of fundamental designs allows the production volume of each fundamental design to be higher, allows larger component volume purchases, and therefore leads to

more cost effective production techniques, and allows insertion of new features during production. Simplified testing/validation arises with fewer designs, which must be validated.

2.1.6 Regulatory Agency's Problem

The most pressing regulatory issue is how to meet the demand from the communications industry for additional spectrum that is currently unavailable. The governments of the industrialized nations in general have established regulatory agencies to manage the electromagnetic spectrum. They accomplish this task through rule-making proceedings that classify services, provide RF spectrum for various types of communications links, and specify technical parameters for operation. Historically, as communications needs increase, the need for additional spectrum has been met by opening up new bands at higher and higher frequencies, as technology is developed to utilize these higher bands.

Today, this concept is no longer applicable because rapid industrial growth has generated an overwhelming demand for new communications services. This demand has resulted in several stopgap techniques that have been employed by the regulatory community. For example, in the United States, techniques such as reassignment of government frequencies to the civilian sector, splitting frequency channels in half (FCC refarming docket), interspersing land mobile channels into unused television broadcast channels, etc., have been used. Each of these techniques carries a price tag; depleting government frequencies will likely create future insufficient spectrum capacity for their use, refarming has resulted in compatibility problems between older equipment and the newer technology used to access the additional channels, and interspersing has resulted in increased interference to public broadcast services.

In order to provide for the increasing demand for communications services, current spectrum management policy has been geared to auction spectrum licenses to the maximum extent possible. Tied to this policy is the concept of minimal in-band technical requirements. This allows the license holder complete freedom to utilize the most flexible and best technology available to maximize the communications links at his disposal. This regulatory policy is designed to generate a favorable climate for technology development by maximizing the communications link/dollar cost formula and result in increased efficiency of spectrum utilization.

In summary, due to the lack of available spectrum for communications purposes , the most critical issue for regulatory agencies in the industrialized nations is to develop policies that increase the efficiency of spectrum usage while reducing mutual interference and increasing the ease of frequency refarming.. The only way to meet the ever-increasing demand for communications links is through technology breakthroughs that can economically increase the efficiency of spectrum utilization. These breakthroughs are likely to occur through reconfigurable radios that maintain electromagnetic compatibility with existing systems, permit frequency reuse, and allow flexibility for future technology upgrades.

2.1.7 Semiconductor Vendor's Problem

As semiconductor vendors view the silicon chip opportunity space, the one thing evident to all players in the market is the question, "how will we keep our fabrications facility full." Coupled with this question is the subtle paradigm change in the industry. So far, it is claimed that it has been silicon chip development driven by the requirements of the systems. The silicon chip was viewed as a way to lower cost, to enhance further integration, and to drive products into new markets. In the current era, as less than 0.25 micron process technologies become commonplace, the paradigm is "systems because of silicon chips." Systems that were considered inconceivable two years ago are now commercial, single-chip products today.

These two issues together are putting tremendous business pressures on semiconductor vendors. To keep fabrications full, they focus on industries, which require very large silicon chip volumes. For this reason, the wireless industry is the focus of almost every major semiconductor vendor. . The wireless industry offers tremendous volumes, and increasing digital CMOS silicon chip content in the forms of DSP and microprocessor cores coupled with embedded logic. This picture simply whets the appetite of every silicon chip vendor hungry to move high-margin CMOS silicon wafers. Moreover, the profit margins per wafer are becoming more and more important. This value is derived from the intellectual property in the form of hardware and software.

Playing directly against this is the fact that to succeed in the wireless industry, and recognize the full advantage of the "volumes of silicon chips," semiconductor vendors must be able to offer silicon chip solutions that support the multitude of standards across different consumer markets and geographical regions. These solutions must appear on the market at particular power, performance, and price points dictated by the end-product market dynamics. This requires the semiconductor vendor to develop application-specific signal processing solutions for every standard, ranging across IS-136, GSM, DECT, PHS, PDC, IS-95 CDMA, and ISM-band cordless systems. This is an expensive proposition today. The cost is dominated by the need to create a customized semiconductor solutions from scratch for every standard. Moreover, this fixed cost creates a limit on the number of design starts that the vendor can support, irrespective of fabrication facility capacity.

The availability of a software-defined transceiver can fundamentally change this design problem, and, as a result, the business equation. The ability to reduce time on the "design start" process, reduce the number of fundamental designs, significantly reduce silicon manufacturing and test costs, and significantly increase the wafer production volumes per design start creates a business environment where semiconductor vendors can address multiple standards in an economically feasible manner. It is recognized that software-configurability will be a key enabler for this
capability.

2.1.8 Context Diagrams

Context diagrams are used here to define the elements of the communication system network that the SDRF-compliant radio is part of and the interfaces of the SDRF-compliant radio to other elements of the system. Context diagrams provide a graphical method to define the scope of what communication system elements that this document addresses. This section will provide context diagrams for representative systems of the market segments: Commercial, Civil Government, and Defense.

Commercial wireless markets are growing rapidly and include services such as cellular, PCS, paging, wireless data services (e.g., packet radio), cordless phones, wireless local area networks (LANs), satellite, etc. Users value integrated operations in combinations such as cellular, paging, wireless data, and even cordless. Within services, multiple standards exist. The current proliferation of incompatible digital cellular and PCS standards is creating a market environment that will demand software radio technology and standards to facilitate roaming. Figure 2.1-1 provides a representative context diagram for commercial cellular systems.

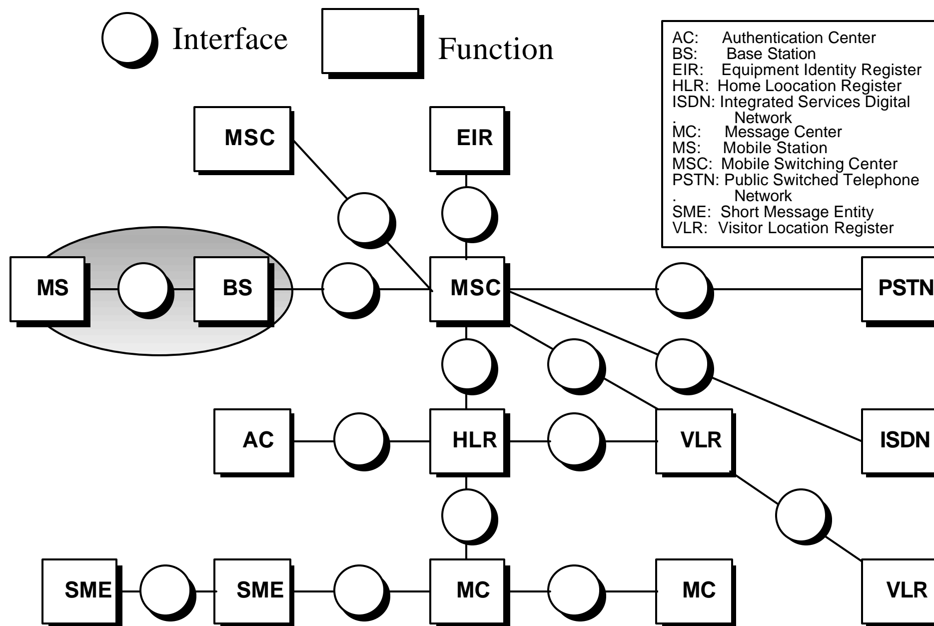


Figure 2.1-1 Network Context Diagram: Cellular/PCS

It is recognized that "software-defined radio" in the SDRF context goes beyond the bounds of a traditional radio and extends from the radio terminal of the subscriber or user, through and beyond the network infrastructures and supporting subsystems and systems. The focus of the SDRF activity summarized in this report addresses the architecture and elements within the shaded area in the context

diagrams. Consideration must also be given to impacts on other elements in the network context diagrams.

Civil wireless encompasses aviation, law enforcement, emergency preparedness, and related applications. These applications address air traffic control and dispatch operations. They are legacy wireless services representing years of operation and installed systems infrastructure equipment. The commercial aviation industry has identified needs for more voice channels and an expansion to include data capability. The traditional 25 kHz channel spacing in Europe is evolving to 8.33 kHz. An international standard for a VHF Digital Data Link (VDL) is currently near finalization. The navigation system is evolving to include the Global Positioning System (GPS). The General Aviation community will maintain legacy radios and systems for years and must be supported.

The public safety, emergency preparedness, and related applications are evolving from 25/30 kHz channel spacing to equipment that supports narrow channel spacing and more channels. Digital operations are envisioned. Interoperation among various agencies and services has usually not been possible and is an identified priority. Civil wireless users desire wireless equipment that supports deployed legacy and more capable emerging radio technologies that software radio technologies can provide. Figure 2.1-2 is a context diagram for a representative civil aviation system.

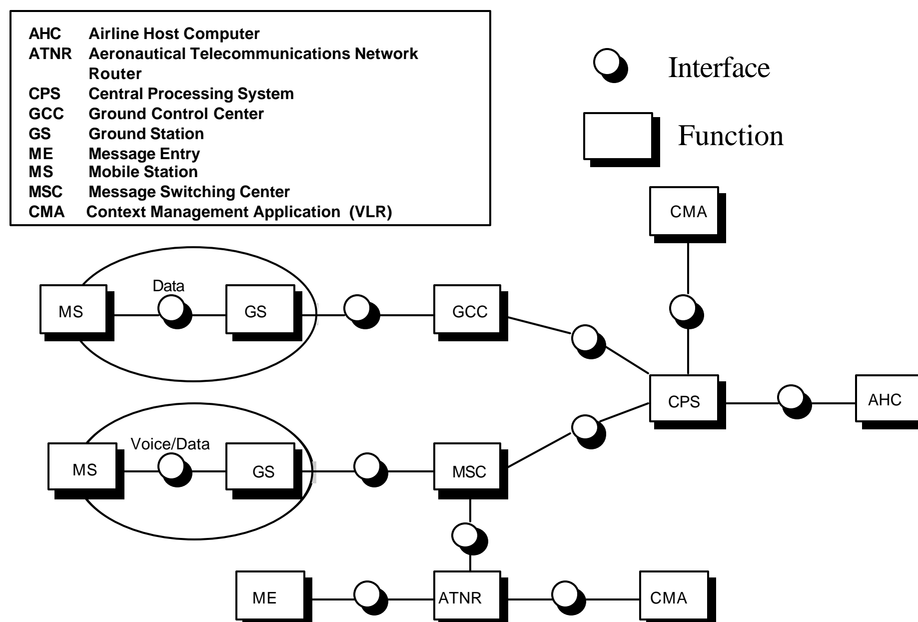


Figure 2.1-2 Network Context Diagram: Civil/Aviation

Current military wireless communication strategies envision integrated operations of land, sea, and air. Highly mobile operation is envisioned with integrated wireless communications extending to even the individual soldiers. Spread spectrum modulation will be employed for multiple access and low probability of intercept (LPI) or detection (LPD) and jamming immunity and also addresses the multipath fading impairments. Additionally, the typical fixed network infrastructure may be non-existent, thereby

necessitating the deployment of transportable and/or highly mobile networks into the field. Thus adaptive network (re)configurations will be required. Multimedia data consisting of data, voice, graphics, and video will be widely available, even in a time varying limited manner to the foot soldier. The goal will be to provide rapid data collection and dissemination in anticipated limited war scenarios within urban areas, mountainous areas, and other geographically restricted areas. Integrated operations will be facilitated by advanced emerging spread spectrum technologies and by flexible software radio that can interoperate with most varieties of narrowband legacy wireless waveforms and equipment. The context diagram for a representative military wireless system is presented in Figure 2.1-3.

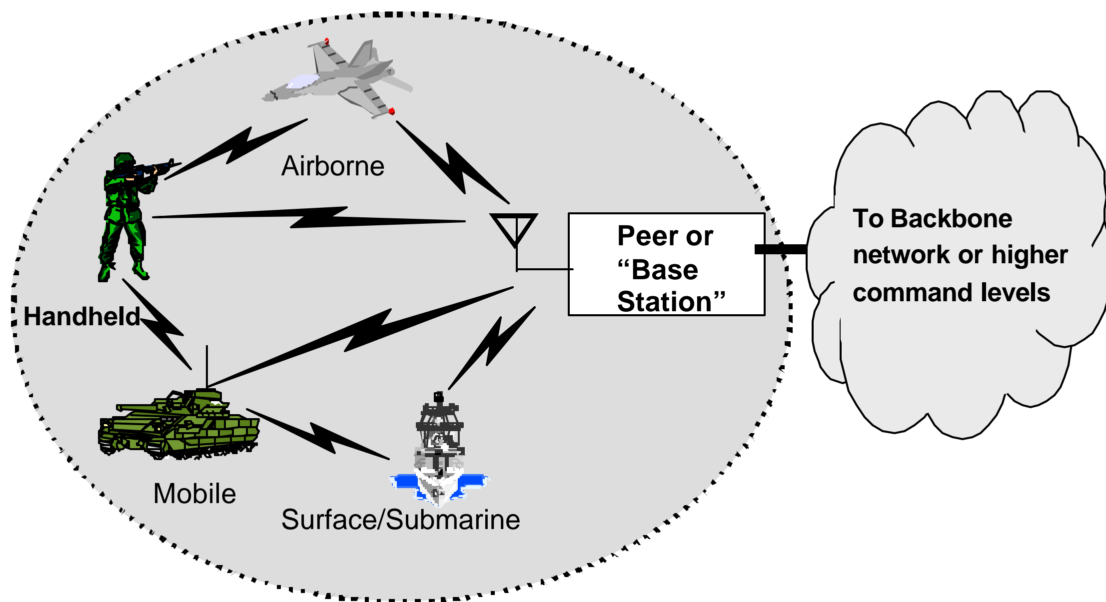


Figure 2.1-3 Network Context Diagram: Defense Applications

2.2 Current Operational Environment

This section provides an overview of the wireless services, supporting standards, critical defining parameters, and required protocols that the SDRF standards recommendations development activities will use as a requirements base.

The goal of this section is to provide the data needed to identify wireless services and standards recommendations according to anticipated capabilities of various classes of software radio platforms.

The Terms and Definition Table defines the different categories and types of applications. These include terms to be defined within a framework where these guidelines may serve both a current vision of SDRF as a software radio, as well as for future developments. Targeted future possibilities include devices and systems which are functionally defined through the digital signal processing and software that these devices feature within an architecture that allows for multiple standards and information transfer services. Near-term applications may include combinations of cellular, pagers, cordless telephones, and possibly GPS. The ability to also include a low earth orbiting satellite (LEO) and International Maritime Satellite (INMARSAT)-type satellite radio function, along with the cellular and other services, could also be envisioned. Further growth to include data networking, and the possibility of other services, open up more applications than can be currently categorized usefully. Thus, the categories of applications provide some flexibility and abstraction, so that further applications may be defined, while fitting within the same overall framework.

The terms “applications,” “services,” and “functions” are used here. Their relationship is best seen by reference to the terms definition table. In order of generality, “applications” is the most general, “services” refers to what technology provides and a user receives (e.g., cellular), and “standard” to the specific standards (as defined by standards organizations) provided (e.g., IS-95). Other definitions are included in the Terms and Definition Table.

Examples of actual applications recommendations are in a set of tables defined by the standards they fit in Section 2.2. Other items in the Terms and Definition Table (e.g., simultaneity) are of interest for SDRF applications. Although the SDR Forum does not intend to recommend standards in these areas, the definitions serve as points of reference for future application descriptions in SDRF. Further, work in identifying SDRF applications will be carried out as part of the technical working groups.

The recommendations include applications and the functional parameters associated with those applications. In many cases parameters are included by reference to the standard (or other reference) they are associated with. The tables are organized around the major categories, and the markets. A set of tables, for each case, provides the parameters, applicable standards recommendations, and standards references.

2.2.1 Terms Definitions

Table 2.2.1-1 below defines the terms used. Recommendations are defined by class and type. Examples are shown only to clarify their use, and in no way imply that these are the only types to be defined, nor necessarily the most important ones. Recommendations are categorized as those pertaining to applications, interfaces, integration, and form factor. These represent only guidance to the Architecture Subcommittee for the standards recommendations setting process.

Table 2.2.1-1 Terms and Definitions

Category	Type of Recommendations	Definition of Term	Examples
Applications			
	Service	Information transfer capability provided	1. Cellular 2. Mobile Satellite Voice
	Standard	Specific type and protocol of air and user interface, defined by standards organizations	1. AMPS, 2. GSM, 3. GPS
	Standard Parameters	Technical parameters associated with specific standard	1. AMPS Channel Bandwidth: 30 kHz; 2. GSM Channel Bandwidth: 200 kHz
Application Features			
	Simultaneity	Simultaneous standards	1. GSM and GPS
	Reconfigurability	Method for changing standards	1. User Selectable, 2. Automatic for Best BER
	Environment	Specific environment factors such as RF or Mobility	1. User mobility up to 100 MPH 2. Fading in urban terrain
Interfaces			
	User Interface	Type of user interface	1. Portable voice handset
	Service Interface	Type of interface with service provider, often included by reference to standard air interface	1. RF interface into Base Station
	Applications Program Interfaces (API's)	API's Allowed or Disallowed; if blank, none are specifically Disallowed	1. Optional encryption and/or authentication API's
Integration			
	Interworking	Type and mode of interface to other open architecture systems	1. Seamless interface to PC for email messaging 2. Interoperability with existing systems
Form Factor			
	Size	length*width*height	
	Weight	weight	

Category	Type of Recommendations	Definition of Term	Examples
	Power	-total power consumption, -type of power supply, -length of time with power supply without recharge/service, -type of recharge/service	
Other			

2.2.2 Service Parameter Tables

The commercial, Civil Government, and defense spectrum allocations in the US are presented in Figure 2.2.2-1. An overview of multiband requirements is illustrated in the example of the U.S. spectrum allocations shown in the figure. Similar situations exist in other countries and regions. Figure 2.2.2-2 is an example of the Japanese spectrum allocation and Figure 2.2.2-3 is a European example.

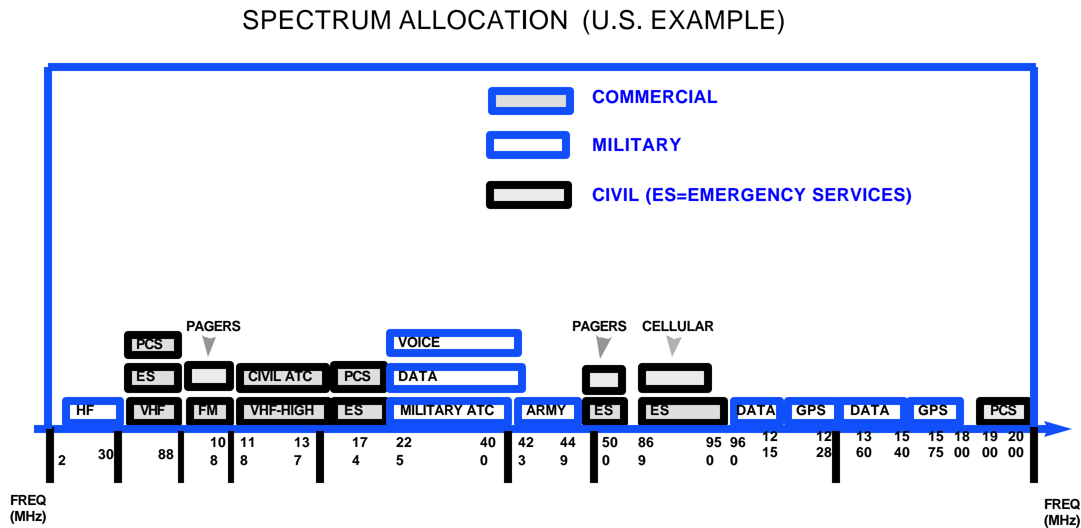


Figure 2.2.2-1 U.S. Spectrum Allocation

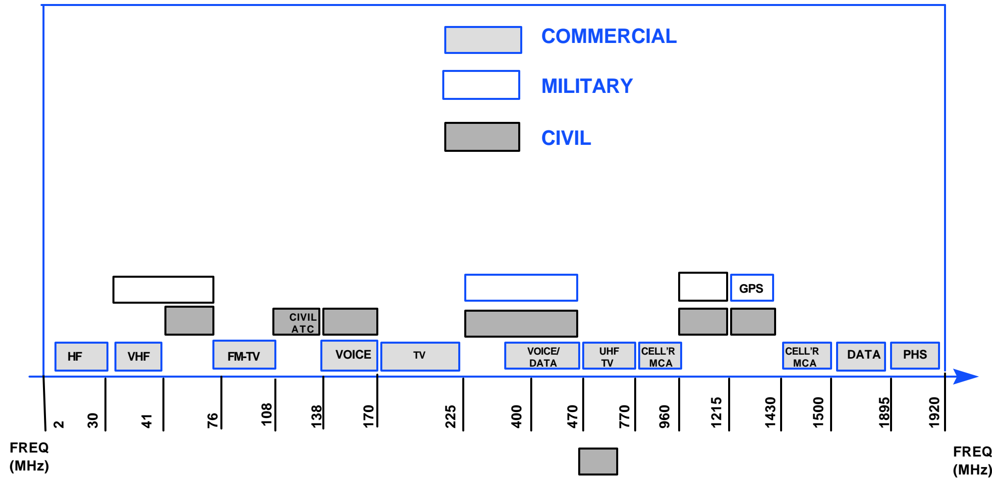


Figure 2.2.2-2 Japan Spectrum Allocation

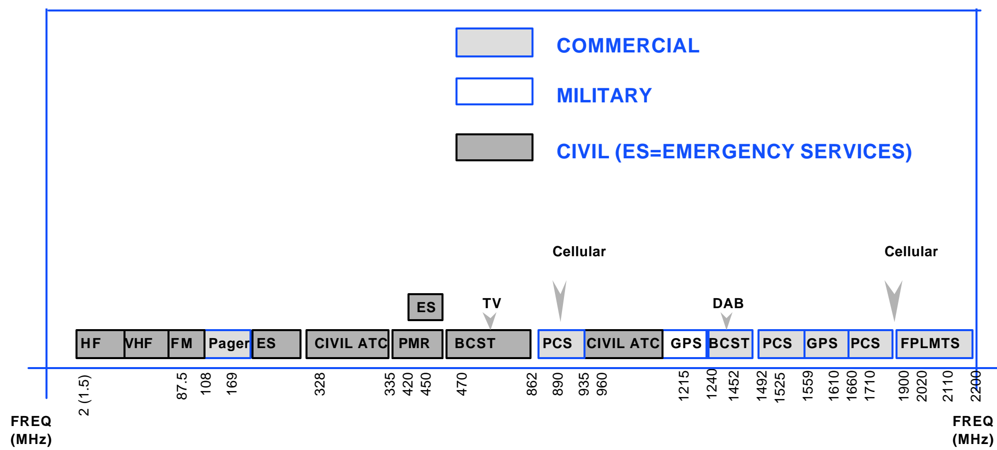


Figure 2.2.2-3 Sample European Spectrum Allocation

Examples of the Service/Standards and the critical parameters are defined in the following tables:

- Table 2.2.2-1, Representative Commercial Wireless Standards and Parameters
- Table 2.2.2-2, Representative Civil Wireless Standards and Parameters
- Table 2.2.2-3, Representative Military Wireless Standards and Parameters

Table 2.2.2-1 Representative Commercial Wireless Standards and Parameters

STANDARDS	Freq. (MHz)	Channel Bandwidth	Raw Data Rate	Modulation Format	Voice Coding	Multiple Access	Duplex	Tx Power
AMPS	TX 824 - 849 RX 869 - 894	60/30 kHz	Analog	FM	Analog	FDMA	FDD	Handset: 600 mw
IS-54/136	TX 824 - 849 RX 869 - 894	60/30 kHz	48.6 kbps	DQPSK	VSELP/8kbps ACELP/ 9.4kbps	TDMA	FDD	Handset: 600 mw
GSM	Tx 880 - 915 Rx 869 - 894	200 kHz	270.833 kbps	GMSK	RPE-LTP 13 kbps	TDMA	FDD	Handset: 2 W
IS-95	TX 824 - 849 RX 869 - 894	1.25 MHz	1.2288 Mcps/1.2-14.4 kbps	OQPSK	QCELP 13.2 kbps	CDMA	FDD	Handset: 200 mw
CT-2	864 - 868	100 kHz	32 kbps	GFSK	ADPCM 32 kbps	FDMA	TDD	
POCSAG	929 -932	25 kHz	2.4 kbps	FSK		TDMA	FDD	
Reflex (Narrowband PCS)	TX 901 - 902 Rx 929 - 932 Rx 940 - 941	Rx 25 / 50 kHz Tx 12.5 kHz	Rx 12 / 24 kbps Tx 9.6 kbps	4FSK		TDMA	FDD	
RAM	Tx 935 - 941 Tx 896 - 901	12.5 kHz	8 kbps	GMSK		FDM	FDD	3 W
ARDIS	Tx 851 - 866 Rx 806 -826	30 kHz	4.8 to 19.2 kbps	proprietary		FDM	FDD	
ISM Band (U.S.)	902 - 928 MHz 2.4 - 2.485 GHz 5.75-5.85 GHz	wide variety No Standard	WLAN, WPBX cordless phone DS-1 links	FCC Part 15 Spread Spectrum DS & FH		Typically FDMA	Typically FDD	1 W (USA)
DECT	1880-1900	1.726 MHz	1.152 Mbps	GFSK	ADPCM 32 kbps	TDMA	TDD	250 mW
DCS 1800	Tx 1805 - 1880 Rx 1710 - 1785	200 kHz	270.833 kbps	GMSK	RPE-LTP 13 kbps	TDMA	FDD	1W
PCS 1900	1800 - 1950	200 kHz	270.832 kbps	GMSK	CELP 13 kbps	TDMA	FDD	Handset: 600 mw
IS-136+	1800 - 1950	6030 kHz	48.6 kbps	DQPSK	ACELP 7.4 kbps	TDMA	FDD	Handset: 600 mw
IS-95+	1800 - 1950	1.25 MHz	1.2288 Mbps	OQPSK		CDMA	FDD	Handset: 200

STANDARDS	Freq. (MHz)	Channel Bandwidth	Raw Data Rate	Modulation Format	Voice Coding	Multiple Access	Duplex	Tx Power
								mw
IS-661 Omnipoint	1800 - 1950	2.5 MHz				FDMA/TDMA/ CDMA	TDD	Handset: 600 mW
PACS	1930-1990 1850-1910	300 kHz	64 kbps	π/4 OQPSK	ADPCM 32 kbps	TDMA	FDD	
PDC	Tx 925-956, 1477-1501 Rx 810-818, 870-883, 1429-1453	50/25 kHz	42 kbps	π/4 OQPSK	PSI-CELP 3.45 kbps	TDMA	FDD	Handset: 600 mW
PHS	1895-1918	300 kHz	384 kbps	π/4 OQPSK	ADPCM 32 kbps	TDMA	TDD	Handset: 10 mW (Avg.) 80 mW (Burst)
IRIDIUM (mobile user segment)	1616 - 1626.5	3,840 channels total (48 cells per satellite, 80 channels per cell on average)	50 Kbps burst to provide voice at 4.8 Kbps and data at 2.4 Kbps	QPSK		FDMA/ TDMA	FD	

Table 2.2.2-2 Representative Civil Wireless Standards and Parameters

Standards	Freq. (MHz)	Channel Spacing	Raw Data Rate	Modulation Format	Multiple Access	Duplex	Tx Power
VHF Digital Link	117.975-137	25 kHz	37.5 kHz	D8PSK	TDMA	HD	
VHF Air/Ground	117.975 - 137.	8.33 kHz/25 kHz	Analog	AM-DSB	FDM	HD	
VHF Air/Ground	108.0 - 117.975	25 kHz	Analog	AM-DSB	FDM	Simplex	
Maritime VHF	156-165(US) 174(EU)	5 kHz	Analog	FM +/-5kHz	FDM	HD	
APCO-25	FCC Part 90	12.5/6.25 kHz	9.6 kbps	APSK/C4FM	FDM	FD	

Table 2.2.2-3 Representative Military Wireless Standards and Parameters

STANDARDS	**Freq. (MHz)**	**Channel Spacing**	**Raw Data Rate**	**Modulation Format**	**Multiple Access**	**Duplex**	**Tx Power**
UHF Voice/Data 188-243	225-400	25 kHz	16 kbps	AM/FM	FDM	HD	
SATURN	225-400	25 kHz	16 kbps	CPFSK	FDM	HD	
Have Quick	225 - 400	25 kHz	16 kbps	AM-DSB/ASK	FDM	HD	
SINCGARS	30 - 88	25 kHz	16 kbps	CPFSK	FDM	HD/FD	
Satcom/DAMA	225 - 400	25 kHz	19.2 kbps burst	various	TDMA - DAMA	HD/FD	
HF Analog 188-141A	1.5 - 30	3 kHz	Analog	SSB, ISB	FDM	HD	
HF Data Modems 188-110A	2 -30	3 kHz	9600 bps	Various	FDM	HD	
HF ALE 188-141A	2 -30	3kHz	75 bps	FSK	FDM	HD	
EPLRS	423 - 449	3 MHz	56kbps	MSK/CPSK	TDMA	HD	
JTIDS	960-1215	3 MHz	384 kbps	MSK/CCSK	TDMA	HD	
VRC-99	1350 - 1850	5 MHz	10 Mbps	QPSK	TDMA/FDM	HD	
NTDR	225 - 450	2.4 MHz	500kbps	QBL-MSK/ CQPSK	CSMA/CA/ FDM	HD	
DWTS	L - Band		2 Mbps Max		FDM/ Trunking	FD	

2.2.3 Requirements

2.2.3.1 Handheld Requirements

Handheld system solutions are driven by a set of requirements, which differentiate them from mobile and fixed systems. The most notable are power, cost, volume, and weight. Handheld solutions have to be in a form factor that is convenient for a person to hold and carry and to have the longest possible battery life. They are typically battery powered using transmit power ranging from 1 mW to 3 W (limited by health concerns). Typical commercial cellular and PCS single-mode single-band handsets today deliver in the range of from 9 to 150 hours of standby time. A recent solicitation from the US military for special unit operations sought 48 hours of "use" on a single battery charge. Another factor related to power management and form factor is heat dissipation. There is no space for cooling devices such as fans and not enough battery power available to afford solutions that generate large amounts of heat. Another aspect of handheld systems is the focus on cost. Since the number of units fielded is relatively large in proportion to other network components, the focus on cost minimization is significant.

Traditional handset form factors are fairly well established. A series of new form/function extensions of handhelds are appearing. A class of small portable personal devices is emerging. They are variously described as "organizers", "Personal Digital Assistants (PDA's)", etc. There is work underway in some quarters to combine the functions of a handset and a PDA. Another class of extensions have to do with adding multimedia capability to a traditional handsets. Examples include the addition of miniature imaging devices and cameras to conventional handsets. These two development vectors are coming together with additional requirements in a class of extensions generically called "wearables". Wearables may combine LAN & WAN wireless capabilities with hands free user interfaces and sensors to support teams working in medical, industrial, and military environments. They may move into general commercial usage, but the early drivers are military, industrial, and medical.

This handheld architecture is based on these requirements.

2.2.3.2 Mobile System Applications and Requirements

The following sections outline a series of operational requirements and applications for mobile units. These mobile units are applied in the military, Civil Government, and private land mobile environments.

Applications

Five mobile information transfer system models will be considered:

- Military land vehicle,
- Aeronautical,
- Naval shipboard,
- Manpack, and
- Automotive information transfer systems.

Common characteristics

The following common features are often included for mobile information transfer systems deployed in the five application environments:

- Operation in a frequency range of nominally 2 MHz (1.5 MHz for some European military service) to 2 GHz,
- Exciter power of 2 watts, into a power amplifier,
- Multi-channel operation,
- Consideration of co-site performance,
- Bridging capability within the system,
- A user interface to control each channel,
- Functionally controlled by software so that the waveform executed by each channel is determined by the software loaded.

MILITARY LAND VEHICLES

Environment

Implementation of the digital battlespace requires communication between force elements on the move and in fixed positions. Military land vehicles include tanks and other vehicles participating in the battle and supporting it. These vehicles need to participate in nets that provide command, control, situational awareness, sensor data, and processed intelligence to all levels of command. Manpack systems used by individual warfighters although participants in the same nets, are treated in a separate model due to their differing form factor characteristics.

Description of the system

The mobile information transfer system will be installed in a vehicle, and will receive power from the vehicle electrical system or from an auxiliary power system. Antennas will be located either on the vehicle or a short distance from it.

For operation in a tank, the user interface of the information transfer system will connect into the vehicle intercommunication system, and will normally be controlled by the tank commander. It will receive situational data from tanks in the same unit and from central resources for display. It will transmit sensor data, including GPS position data and vehicle operational data to higher level units. It will transmit commander’s orders to subordinates as either voice or display data in real time to support unit maneuvers.

Installed in a High Mobility Multi-purpose Wheeled Vehicle (HMMWV), the information transfer system will typically support a tactical operations center or other field headquarters. While on the move personnel can communicate on voice and data channels. At the site, connection will be made to the headquarters local area network, making radio channels available to the commander and personnel

manning the center. In order to provide dispersion, the unit will operate at a distance of 25 to 100 meters from the central facility.

Operation of the system

The mobile radio in a land vehicle operates in a wide variety of ways, each of which includes the following characteristics:

- The system operates with a variety legacy waveforms that may include the following as examples:
    - Global Positioning System (GPS)
    - Joint Tactical Information Distribution System (JTIDS)
    - Enhanced Position Location Reporting System (EPLRS)
    - Packet Radio (VRC-99)
    - Single-Channel Ground and Airborne Radio System (SINCGARS/SIP)
    - High Frequency (HF)
    - UHF Air-ground voice radio (Have Quick Saturn)
    - Trunked radio 10 Mbps
    - Near Term Digital Radio (NTDR)
    - Cellular, PCS

- Typically the interface to the user is modular and includes provisions for internetworking functions like the following:
    - FDDI
    - Ethernet 100 Mbps
    - TCP/IP
    - RS232
    - RS422

- The radio incorporates appropriate INFOSEC and Transient Electromagnetic Pulse Standard (TEMPEST) controls

- Front panel fill and over the air rekeying integrate with the INFOSEC system

- Participation in and adaptive hand-off between operational clusters

Support for division level networks of up to 5,000 nodes

Functions of the information transfer system

The information transfer system will be the backbone of digital battlespace operations wherever landlines are not available. Functionality will vary from replicating the capability of a simple radio for voice contact to acting as a small digital switch.

With its multi-mode capability, the information transfer system will act as a repeater so that an individual equipped with a legacy radio such as SINCGARS can talk directly with support aircraft using UHF Have Quick through an information transfer system.

Implications of mobility on the information transfer system

Mobility implies the following:

- An ability to move rapidly from one operating location to another while maintaining the full capability to communicate
- Operate without dependence on a power line
- Mitigate co-site self jamming
- Modular extension and replacement
- A form factor smaller and lighter than the equipment replaced

Important parameters

Key operational parameters are:

- Frequency
- Modulation type
- Timing
- Orderwire
- Power level
- Keys and hopsets

AERONAUTICAL

Environment

As part of the communications, navigation, and IFF (Communication, Navigation, and Identification [CNI]) system of both military and commercial aircraft, a number of different types of equipment have been traditionally installed in each tail number. When an aircraft is designated for a specific mission or to fly a route in a specific portion of the world, there is a planning element to ensure that it can communicate with other stations as necessary. With a programmable information transfer system, any aircraft equipped with suitable antennas and external RF equipment could be reprogrammed to interoperate with designated waveforms and protocols.

Description of the system

Installation of the information transfer system technology into an aircraft is largely a matter of form factor. If implemented on suitable circuit cards, the system resources can be directly installed in the necessary ATR of SEM-E package for the aircraft type. The internetworking function will necessarily need to match with the aircraft CNI system.

Operation of the system

The information transfer system will operate in a manner similar to other radio equipment installed on the aircraft. The operator's panel will appear as an additional display in the cockpit display, and presets will appear on the channelized control display.

Functions of the information transfer system

The information display system permits the aircraft crew to communicate on any channel for which appropriate software has been loaded into the system archive. Over the air software upload is also possible. Then the crew can talk with ground stations, other aircraft, and satellites. Aircraft sensor data can be downloaded, and graphic data uploaded to the aircraft.

Implications of mobility on the information transfer system

Use in an aircraft involves conformance with the designed form factor, and the replacement must weigh less than previously installed equipment. Power is derived from the aircraft power bus.

Important parameters

Key operational parameters include:

- Frequency
- Modulation type
- Timing
- Power level
- Keys and hopsets

NAVAL SHIPBOARD

Environment

The area of shipboard communications includes all information transfer between naval vessels and an external entity. This includes ship-to-ship communications, ship-to-shore communications, ship-to-aircraft communications, and ship-to-satellite communications. Both commercial shipping and defense naval communications are included.

Description of system

The information transfer system installation is in racks in interior compartments of a ship where it becomes a part of the shipboard communications facilities. The user interface will be used primarily by

radiomen to establish presets available from other stations around the ship. Power is from the ship's generators.

The nature of shipborne communications requirements places particular strains of the modularity aspects of a SDRF radio. Some interesting considerations include:

- Extensibility from as few as five channels per platform on a small ship to over 100 channels per platform on a larger ship.
- Adaptive bandwidth variability from 3 kHz to 3 GHz.
- Multi-media communications for voice, data, video, facsimile, and message signals.
- Security including red and black throughput, a range of crypotographic functions and standards, and external cryptographic devices.
- Demand-assigned or dedicated communications channels accessible by a single user or user group.
- Guaranteed quality of service describing priority, bandwidth, and reliability.
- Variable ranges from line-of-sight communications to 11,000 km.
- Co-site interference control mechanisms to manage interference between collocated receiver and transmitters.
- Remotely configurable through a standard network management protocol such as SNMP.
- Adjacent channel interference control.

Operation of system

The system will provide communications channels to support diverse services. Examples of current services are listed here, primarily emphasizing US Navy communications services.

- Tactical Group Communications circuits for battle group maneuvering, urgent tactical communications, intelligence information, operations and administrative communications, and communications with units deployed to join the battle group.
- Anti-Submarine Warfare (ASW) Communications circuits for ship-to-ship and ship-to-air monitoring of submarine activities.
- Anti-Surface Warfare (ASuW) Communications circuits exchange tactical and air, surface and sonar information between ships.
- Anti-Air Warfare (AAW) Communications circuits for the dissemination of information for aircraft control and air raid reporting between ships.
- Electronic Warfare (EW) Communications circuits to control jamming, search and direction finding, and to exchange information.
- Air Operations Communications circuits for aircraft and helicopter control.
- Tactical Air Communications circuits for control of aircraft engaged in operations.
- Amphibious Communications circuits to communicate between offshore platforms, landing force elements and beach headquarters.
- Naval Gunfire Communications circuits to conduct, coordinate, and control naval fire activities.
- Submarine Communications circuits to communicate between submarines, submarines and ships, and submarines and submarine operating authorities.
- Data links to transfer digital signals between ships, ships and aircraft and ships and shore stations.
- Distress Communications circuits for civilian and military search and rescue operations.

- Mine Countermeasure (MCM) Communications circuits to exchange mine countermeasure maneuvering and tactical information.
- Harbor Communications circuits used for civilian and military navigation.
- UHF Fleet Satellite circuits for long-haul communications between ships and shore facilities.
- Long-Haul HF Communications circuits used on many platforms as the primary ship-to-shore communications medium. Allied forces (NATO) use HF and UHF circuits.
- Strategic Submarine Warfare Communications circuits to meet the special communications needs of ballistic missile submarines and attack submarines.
- Navigation circuits to give position, velocity, and time information to vessels at sea.
- Telephone circuits from ships at sea to shore via (commercial) satellite communications.

In the case of the US Navy, the system will interface with network management systems such as Automated Digital Network System (ADNS) to provide automated management of communications resources and automated dissemination and support of fleet communications planning.

Functions of information transfer system

Functional units of the information transfer system must be consistent with the modular concepts developed in the SDRF architecture. RF capabilities range from VLF (3 kHz - 30 kHz) to UHF (300 MHz - 3 GHz). RF also will provide adaptive functionality, including Link Quality Analysis and Automatic Link Establishment and TRANSEC techniques to prevent signal detection and jamming. The modem converts analog and digital radio traffic into a standard waveform. This includes modulation, interleaving, forward error correction (FEC) and multiplexing with various access techniques including frequency division multiple access (FDMA), time division multiple access (TDMA), and code division multiple access (CDMA). INFOSEC uses various cryptographic techniques and/or interface with external cryptographic equipment to insure secure voice and data communications. INFOSEC also supports special cryptographic encoding of DAMA orderwire and other channel control messages to secure the identity of the channel controller. Message Processing includes functions for LAN communications, data compression, and vocoder functions.

Implications of mobility on the information transfer system

As with other forms of wireless mobile communications, loss of signal is a problem. Most ships move relatively slowly. Once communications are established, loss of signal tends to be related more to atmospheric conditions and time of day than to range of motion. Sea state is an important environmental factor. Careful placement of antennas is required to insure a continuous signal despite severe pitch and roll. In addition, external radio modules must be able to withstand harsh environmental conditions and internal information transfer system modules must be ruggedized and securely mounted.

Important parameters/range

The operational parameters include operating frequency, operating time period, data type, security label, quality of information transfer, and participants. Where possible, these parameters will be negotiated between systems and will be transparent to the end users.

MANPACK

Environment

The current trend in battlespace doctrine calls for electronic connection to each warfighter. Sensor data including video and laser ranging information is fed back to intelligence centers where the situation analysis is kept current. Then the location of friendly and enemy elements can be fed back to individuals and elements for their tactical use. Command and control information also is fed forward.

DARPA, the US ARMY, and the US Marine Corps are conducting experiments such as Small Unit Operations, Sea Dragon, and Extended Littoral Battlespace. These experiments will form the basis for the future doctrine of land forces in the information age.

Description of the system

The manpack communications system will have to be lightweight; have low power consumption; have a simple, user friendly human interface; possess low probability of intercept and low probability of detection in order to enhance operator survivability, and be rugged enough to withstand the environmental rigors of a combat situation.

Operation of the system

The communication system will be multifunctional (voice, data, imagery, and video), multimode (legacy and new waveforms), provide high-resolution location of the operator; and be capable of secure operation.

Functions of the information transfer system

In order to communicate with current combat net radios, the system would have a VHF mode and with current Air Force equipment, would have to have a UHF mode. A cellular capability is envisioned and could operate with a mobile base station from an airborne platform like an unmanned aerial vehicle (UAV). Location could be provided by the Global Positioning System or in the case of dense foliage or inside buildings, could have a Loran or TDOA type capability. The capability to broadcast VHF (ground) and UHF (air) simultaneously to call for timely artillery, mortar, and rocket fire support and call for air support would be required for small units to fight and survive. A capability to receive imagery and video from the global broadcast system is envisioned for map updates, enemy positions, non-combatants locations, etc. Over-the-air keying of crypto would also be required. A paging capability is

envisioned to alert individuals scattered over a large area of events such as impending nuclear, biological, or chemical attacks.

Implications of mobility on the information transfer system

The most obvious implications are varying propagation effects and non-disruptive handoff as the unit moves. Waveform and protocol optimizations also become issues.

Important parameters

The frequencies of operation are important and should cover at least 30 MHz to 2 GHz. Receive frequencies of wideband information could be much higher but schemes such as receiving the GBS broadcast on a UAV and retransmit a portion of the broadcast in L-Band have been investigated. Bandwidth estimates are 10 MHz per channel. Data rate estimates are 1.5 Mbps for receiving wideband but much lower transmit estimates due to power limitations. Range estimates are 10-15 KM ground-to-ground and many miles ground-to-air.

AUTOMOTIVE INFORMATION TRANSFER SYSTEMS

Environment

The time period from 1980 to 1995 saw the introduction of a large number of microprocessors and controllers dedicated to such functions as engine control, automatic braking systems, transmission control, sound system, GPS, cellular, and route display. With the introduction of the Intelligent Transport System, a number of these functions can be combined with wireless communications. As they are brought together, the flexibility of an information transfer system offers the opportunity of upgrading system functionality by adding software to the existing hardware.

Description of the System

A single unit in the vehicle, using the information transfer system architecture, provides RF receive and transmit channels, audio sound, information display, voice synthesis, and processing. Power is received from the automobile electrical system.

Operation of the system

An increasing number of services are available to motorists on highways using RF links. The Intelligent Transfer System provides a capability to utilize these services. It also permits the user to adapt easily to new services as they become available by loading new communications configurations into the system.

Entertainment is provided by the system through audio amplifiers and loudspeakers. Access to a number of voice communications facilities is provided through a single handset. By receiving the GPS waveform,

vehicle location, direction, speed, and the time are available. That information can be used in conjunction with stored map and route information to provide guidance to the desired location. As intelligent highway services are offered, they can be monitored by this system.

Functions of the information transfer system

- AM radio
- FM radio
- CD player
- Stereo amplifier
- GPS position location
- Cellular phone
- PCS services
- Paging
- Amateur radio
- Citizen's band
- Talk between cars
- Traffic information
- Route display
- Voice announcement
- Vehicle Coordination:
  - Congestion rerouting
  - Platooning
  - etc.

Implications of mobility on the information transfer system

As with any cellular or PCS service, the motion of the car introduces problems with Rayleigh and Ricean fading that will have to be overcome to provide continuity of service. Extreme temperature excursions must be accommodated.

Important parameters

The system must accommodate the operational parameters of the services to be provided.

3.0 SDRF System Architecture

This section focuses upon the Software Defined Radio System (SDRS) architecture. The architecture is a representation of a SDRF system that rationalizes, arranges, and connects components to produce the desired functionality. This architecture is intended to form the basis for specific implementations of a system that meets functional SDRF requirements and also provides upgrade paths for handling enhanced, evolving, and new requirements. This feature of "future proof" architectures is a fundamental goal and challenge for the SDR Forum.

Figure 3.0-1 illustrates the scope of SDRF architectural discussions covered within this report. Section 3.1 discusses the SDRF architecture framework. It includes high-level models, functional interface diagrams, and interface interaction diagrams/tables. Section 3.2 presents specific examples of SDRF architectural models. Included in this report are models and discussions for handheld, mobile applications, and cross standards extensions representing the first three specific architectural work items undertaken by the Forum.

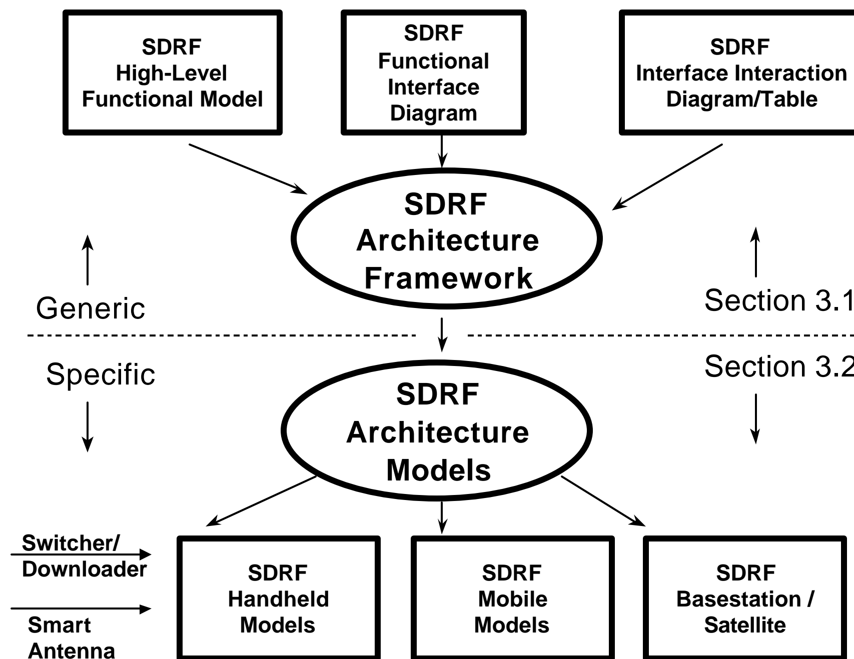


Figure 3.0-1 Scope of SDRF Forum Architecture Work

3.1 Architecture Frameworks

The strategy for meeting the "future proof" goal for a SDRF architecture is to provide high-level functional models that are capable of being mapped into specific software-defined information transfer devices such as handheld, mobile, and base station applications. Articulating the high-level architecture is key to establishing consistency among the specific architecture models to follow. The SDRF architecture framework addresses the higher level architectural aspects for software defined information transfer devices allowing latitude for a variety of specific implementations.

The SDRF open architecture is based upon a high-level generic functional model with functional blocks connected via open interface standards recommendations. The goal of the SDRF functional partitioning is to define an architectural framework that can be applied to specific implementation domains. Examples of these implementation domains are handheld, mobile, and fixed site or base station. The SDRF approach to standards recommendations is outlined in Table 3.1.1-1.

Table 3.1.1-1 Scope of the SDRF Approach to Open System Standards Recommendations

Standard Type	**SDRF Role**	**SDRF Approach**
Air Interface	Support identified standards through common architectural partitioning Identify extensions to accommodate new SDRF capabilities	SDRF will identify and recommend extensions to the appropriate standards body
Internetworking	Support identified standards through common architectural partitioning Identify extensions to accommodate new SDRF capabilities	SDRF will identify and recommend extensions to the appropriate standards body
API	Define	Definition based on SDRF functional model partitioning
Physical Interfaces	Select from existing open standards	Selections based on SDRF functional model partitioning and existing interconnect, backplane, and form factor standards
Analog/RF Interconnects	Identify applicable standards and approaches	SDRF will recommend where standards are lacking
User interface	None	Product dependent

Realization of SDRF architecture has several characteristics:

Flexible - the ability, through band and mode selection, to access a desired part of the electromagnetic spectrum and to construct and decode desired waveforms or protocols through readily achievable reconfiguration.

Upgradeable - the ability to get more or better performance from the SDRF device through the insertion of improved hardware and software technologies. The architecture should provide for this in such a way as to localize the impact to the affected modules or components.

Scaleable - the ability to extend the functionality and capacity of the SDRF device to include multiple channels and networking, additional local connectivity and processing, or new evolving wireless services. Scalability relates to the ability to support the addition of existing functions – quantitative growth.

Extensible - the ability to readily permit an addition of a new element, function, control, or capability within the existing framework. Extensibility pertains to the ability to support new functions – qualitative growth.

Development of a system architecture requires the establishment of three viewpoints of a system; the user/owner, the designer, and the developer. The user/owner is concerned with the operational and business attributes of the system, the system architecture designer concentrates upon identification of interconnection and communication of specific building blocks or functional modules, and the developer focuses upon specific implementation of the chosen functional modules, the technical architecture.

Standards recommendations arise from two sources. De facto standards are the result of wide acceptance and use in the marketplace. These standards recommendations typically emerge from proprietary work and are the intellectual property of the developer. If they are made available to third parties for development then they can be considered "open." These standards recommendations frequently evolve rapidly as the developer makes enhancements and adapts to emerging technological developments.

De jure standards are those established by a central body, and are issued in accordance with the guidelines established by that body for standardization. Although this process is inherently slower than de facto standardization, it has the advantage that original acceptance and subsequent changes take place through an established process. They are slower to become accepted, changes are well publicized in advance, and implement negotiated specifications. Implementers can develop products or systems to the specification with assurance that it will be stable.

SDRF will use both of these types of standards. De facto standards, such as bus architectures, permit access to a wide range of commercially available hardware and software items that can lead to convenient interaction and economies of scale. But the purpose of the SDR Forum is to establish

standards recommendations for key elements of target systems as a basis for standardization where open COTS resources are not available.

The SDR Forum provides a mechanism whereby issues specific to software programmable radios can be resolved with an expedited standardization process. This approach permits systems using advanced technology to be fielded expeditiously with competition based on value added for customers and users rather than unproductive battles between essentially equivalent but different implementations.

Modularity is the key to successful implementation of open systems. Between modules are defined interfaces that are subject to standardization. Within a module the developer is free to implement functionality in the most effective way.

3.1.1 Functional Model

Open systems are those that contain open and standard internal interfaces between modules and open and standard external interfaces with other information systems. Open systems are defined by employing commercially successful non-proprietary interfaces, communications protocols, and application program interfaces.

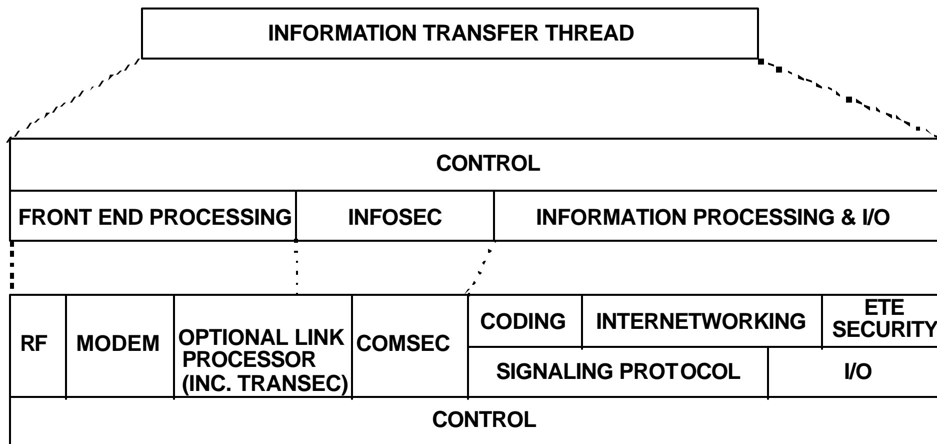


Figure 3.1.1-1 SDRF High-Level Function Model

Figure 3.1.1-1 is a high-level hierarchical functional model for a software defined radio system (SDR). Three views of increasing complexity are presented. The top level view is a simple representation of an entire information transfer thread. The left side interface is the air interface. The right side interface is the wire side and user interface. The next level view identifies a fundamental ordered functional flow of four significant and necessary functional areas; (1) front end processing, (2) information security, (3) information processing, and (4) control. It is noted that diagrams and processes discussed within this document, unless otherwise specified, are two-way devices (send and receive). Note that the functional model as shown in this figure is not intended to show data or signal flow.

Front end processing is that functional area of the end user device that consists generically of the physical air (or propagation medium) interface, the front-end radio frequency processing, and any frequency up and down conversion that is necessary. Also, modulation/demodulation processing is contained in this functional block area.

Information security (INFOSEC) is employed for the purpose of providing user privacy, authentication, and information protection. INFOSEC, within the SDRF model, consists of two fundamental processes: transmission security (TRANSEC) and communications security (COMSEC). TRANSEC includes those processes such as frequency hopping or direct spread spectrum or other signal variation coding; and communications security. COMSEC is the algorithmic encryption and decryption of the digital or digitized analog information. Another primary function is the management of INFOSEC, key management. In the commercial environment, this protection is specified by the underlying service standard while in the defense environment, this protection is of a nature that must be consistent with the various Governmental doctrines and policies in effect.

Content or information processing is for the purpose of decomposing or recovering the imbedded information containing data, control, and timing. Content processing and I/O functions map into path selection (including bridging, routing, and gateway), multiplexing, source coding (including vocoding, and video compression/expansion), signaling protocol, and I/O functions.

Figure 3.1.1-2 demonstrates that the SDRF Architecture features two important attributes: scalability and extensibility. The advantage of this architectural approach is that development can proceed asynchronously in different parts of the system. In other words it supports an evolving design process. The high-level architecture presented in Figure 3.1.1-1 is scaled in one part of the figure to show a “better modem” and “better TRANSEC.” These improved features could mean increased processing capability, lower power operation, smaller size, etc. The importance of the scalability attribute is that the SDRF architecture accepts modular improvements in a seamless and transparent fashion. The figure also demonstrates how the architecture may be extended to show a multiple channel configuration.

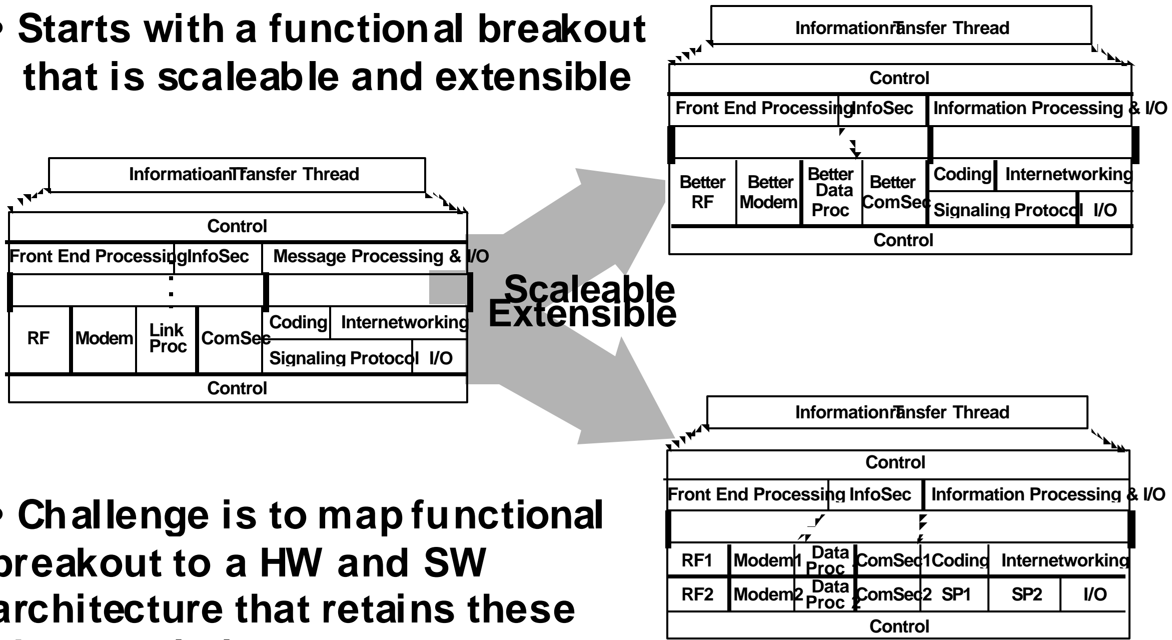


Figure 3.1.1-2 SDRF Architecture Evolution Process

Figure 3.1.1-3 illustrates that the SDRF functional model maps to three specific applications: a handheld unit, a mobile system, and a basestation. The SDRF functional model is common to each of the implementations with more detailed descriptions provided in Section 3.2.1 for the handheld model and Section 3.2.2 for the mobile model. The basestation model is similar to the mobile model.

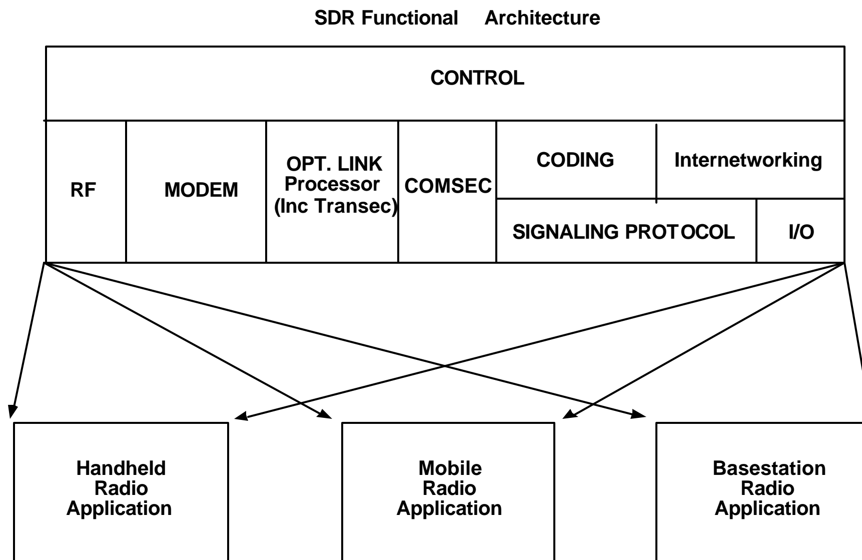


Figure 3.1.1-3 One Common SDRF Functional Architecture Maps to Handheld, Mobile, and Basestation Radio Configurations

The SDRF architecture consists of functions connected through open interfaces, and procedures for adding software specific tasks to each of the functional areas. The software necessary to operate is referred to as a software application. Figure 3.1.1-4 is a diagram of the SDRF open architecture showing six independent subsystems interconnected by open interfaces. In this view the generalized SDRF functional architecture has been particularized by equating a subsystem definition to each functional area. In general this is not the case; subsystems will be determined by implementation considerations. Interfaces exist for linking software application specific modules into each subsystem. Each subsystem contains hardware, firmware, an operating system, and software modules that may be common to more than one application. The application layer is modular, flexible, and software specific. The common software API layer, inferred in Figure 3.1.1-4, is standardized with common functions having open and published interfaces. Peer-to-peer interfaces are neither required nor proscribed.

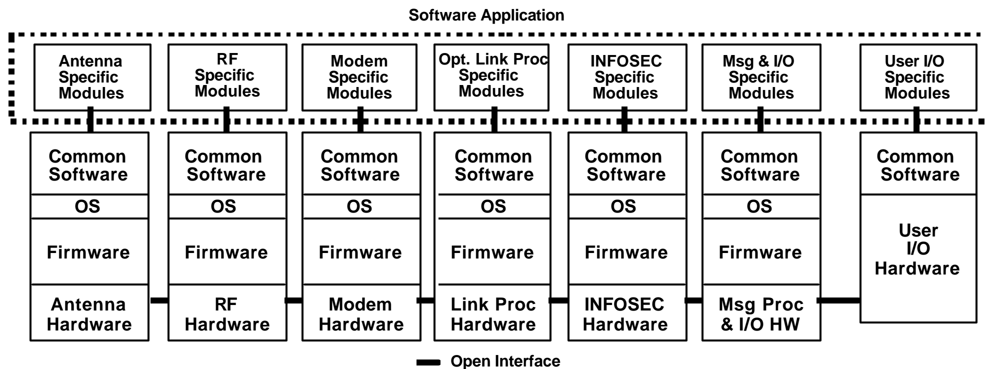


Figure 3.1.1-4 An Example Implementation of SDRF Software and Hardware Open Architecture

Figure 3.1.1.5 presents the SDRF functional interface diagram and demonstrates how the SDRF Architecture extends to the definition of functional interfaces. A representative information flow format is provided at the top of the diagram. Actual representations will be implementation dependent. Interfaces are identified for information and control. For example, information transfer is effected throughout the functional flow within the SDRF architecture to/from antenna-RF, RF-modem, modem-INFOSEC, and INFOSEC-Message Processing interfaces. Control and status is effected between the same interfaces as information and, in addition, control is effected between each functional module and one or more control points and interfaces. Auxiliary interfaces are also allowed, as shown on the diagram.

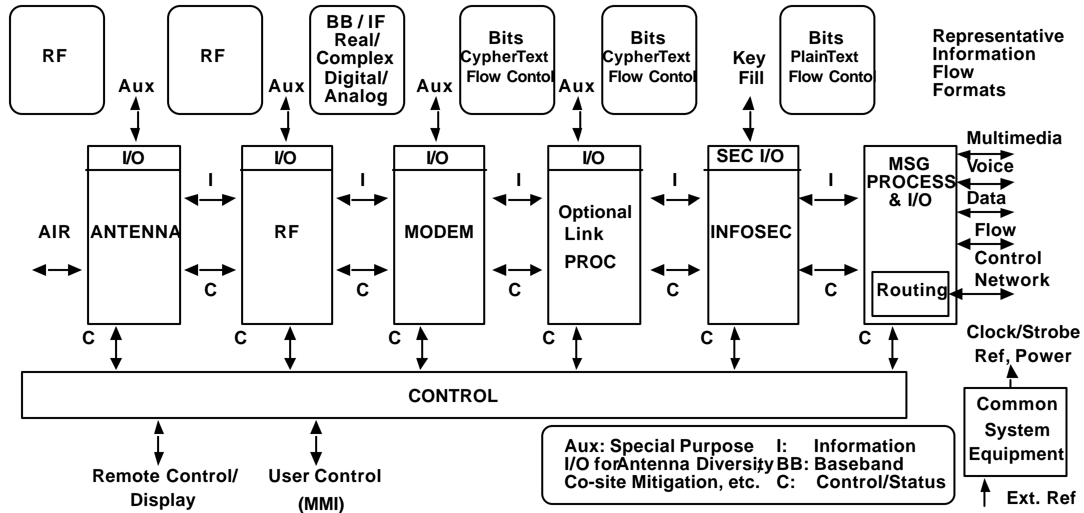


Figure 3.1.1-5 SDRF Functional Interface Diagram

3.1.2 Interaction Diagram

Table 3.1.2-1 supports the functional interface diagrams by employing a matrix which plots information; [I] and control/status; [C] as a function of the appropriate SDRF interface. For example, the RF-Antenna interface contains information as well as control/status whereas the Air-Antenna interface contains information only. The matrix also identifies specific auxiliary interfaces for the purpose of transferring information among multichannels of a particular system or between systems in support of multichannel processing algorithms. Typical external interfaces are also identified within the matrix. The keys for the interaction diagram are:

- I: Information Flow Interface, i.e., information to be transferred over the communication link and information embedded in the signal-in-space waveform (e.g., training symbols, spread spectrum symbols).
- C: Control/Status Interface, i.e., information transferred for the purpose of controlling other functional blocks or for generic radio control functions.
- Aux: External interface to a similar block (e.g., antenna to antenna interface for co-site mitigation) on the same or other radio channel.

Table 3.1.2-1 Interface Matrix

N^2 INTERFACE	AIR	ANT	RF	MODEM	INFOSEC	I/O	SEC. I/O	CONTROL	USER (MMI)
AIR		I							
ANT	I		I C	C				C	
RF		I C		I C	I C			C	
MODEM		C	I C		I C	I C		I C	
INFOSEC			I C	I C		I C	I C	I C	
SEC. I/O					I C				I C
I/O				I C	I C		I C	I C	I C
CONTROL		C	C	I C	I C	I C			I C
USER (MMI)						I C	I C	I C	
AUX		YES	YES	YES				YES	YES
WIRE SIDE I/O						YES			
FILL DEVICE							YES		
REMOTE CONTROL								YES	

This table gives the general view of the candidate interfaces under consideration for SDRF standards recommendations. Note that each module in the first column will require a detailed functional description to classify the functions that must be accomplished within that module but without specifying how those functions will be implemented. The internal requirements of the modules are only limited by the compliance with the input and output standardized characteristics as prescribed by SDRF.

Figure 3.1.2-1 is a graphical depiction of the interface matrix in the form of an NxN interface/interaction diagram. In Figure 3.1.2-1, each of the interfaces shown on each of the modules represent a potential for standards recommendations to establish an open architecture.

Table 3.1.2-2 describes the interface/interaction diagram interfaces with representative example information and/or control and status content. Table 3.1.2-2 offers a finer decomposition of those interfaces and examples of the content that are associated with each. The content of each information and control interface will necessarily need to be further developed, described, and bounded.

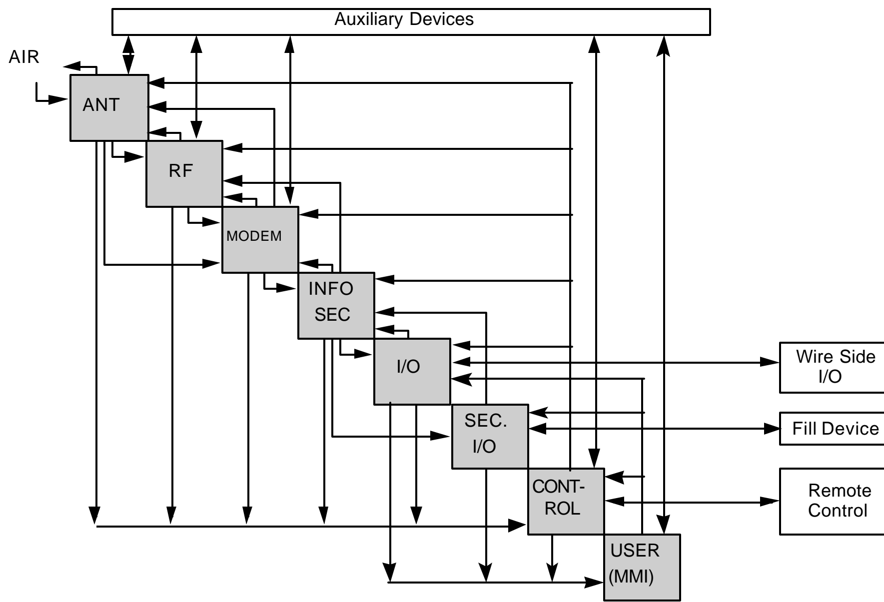


Figure 3.1.2-1 Interface/Interaction Diagram

Table 3.1.2-2 Interface/Interaction Diagram Interfaces and Example Content

Interface	Transfer Type	Example Content
Air to Antenna	I	Information flow is defined by the air interface standard
Antenna to RF	I	Information flow in the RF signal
Antenna to RF	C	RF/Antenna status interfaces (beam steering, etc.)
Antenna to Modem	C	Status information interface for beam steering, etc.
Antenna to Control	C	Control status interface
Antenna to Environment Adaptation	C	Antenna status information for the purpose of adaptation algorithms
RF to Antenna	I	Information flow in the RF signal
RF to Antenna	C	RF/Antenna control interfaces (beam steering, etc.)
RF to Modem	I	Information flow in the RF/IF/Baseband signal
RF to Modem	C	Status interface for AGC, etc.
RF to Control	C	Control status interface
RF to Environment Adaptation	C	RF status information for the purpose of adaptation algorithms
Modem to Antenna	C	Modem control of antenna for beam steering, etc.
Modem to RF	I	Information flow in the RF/IF/Baseband signal
Modem to RF	C	RF control such as frequency control and AGC
Modem to INFOSEC	I	Information (cipher text) flow within the received bits
Modem to INFOSEC	C	Modem status used by the INFOSEC function

Interface	**Transfer Type**	**Example Content**
		(transmit/receive, etc.)
Modem to Control	I	Information retrieved from the communication link data stream by the modem for control purposes.
Modem to Control	C	Modem status information
Modem to Environment Adaptation	C	Modem status information for the purpose of adaptation algorithms
INFOSEC to RF	I	TRANSEC information for waveform parameter variation
INFOSEC to RF	C	Mode control information such as disabling an RF function due to another mode being performed by another RF function such as an LPI channel
INFOSEC to Modem	I	Information such as mode, preambles, key transfers
INFOSEC to Modem	C	TRANSEC information for waveform parameter variation, encrypted digital bits, flow control
INFOSEC to IO	I	Unencrypted bits
INFOSEC to IO	C	Mode switches, flow control
INFOSEC to Secure IO	I	Keys
INFOSEC to Secure IO	C	Key status, parity, alarms
INFOSEC to Control	I	Recovered information used in radio control algorithms
INFOSEC to Control	C	Status
INFOSEC to Environment Adaptation	C	COMSEC acquisition status
Secure IO to INFOSEC	I	Keys
Secure IO to INFOSEC	C	Control parameters
Secure IO to User	I	Front panel display, key status
Secure IO to User	C	Front panel keypad
IO to Modem	I	Bits when INFOSEC function not present
IO to Modem	C	IO derived modem control information, flow control
IO to INFOSEC	I	Bits
IO to INFOSEC	C	IO derived control information, flow control
IO to Control	I	Information parsed from the received bit stream
IO to Control	C	IO derived control information
IO to Environment Adaptation	C	Receive statistics
IO to User	I	Multimedia information
IO to User	C	Flow control
Control to Antenna	C	Antenna control parameters
Control to RF	C	RF control parameters
Control to Modem	I	Control information for the Modem to insert into the information flow
Control to Modem	C	Modem control parameters
Control to INFOSEC	I	Information for the INFOSEC function to insert into the information flow
Control to INFOSEC	C	INFOSEC control parameters
Control to IO	I	Control information for the IO function to insert into the information flow
Control to IO	C	IO control parameters
Control to Environment Adaptation	I	Control status, statistics
Control to Environment Adaptation	C	Environment adaptation control parameters
Control to User	I	Information parsed from the received information stream
Control to User	C	Display of operating status
Control to Remote Control	C	Status

Interface	Transfer Type	Example Content
Environment Adaptation to Antenna	C	Antenna control parameters
Environment Adaptation to RF	C	RF control parameters
Environment Adaptation to Modem	C	Modem control parameters
Environment Adaptation to INFOSEC	C	INFOSEC control parameters
Environment Adaptation to IO	C	IO control parameters
Environment Adaptation to Control	I	Information for the Control function to pass along to other functions to insert into the information stream
Environment Adaptation to Control	C	Control parameter inputs
Environment Adaptation to User	I	Statistics
Environment Adaptation to User	C	Status
User to IO	I	Multimedia information
User to IO	C	Flow control
User to Secure IO	I	Keys
User to Secure IO	C	Secure control parameters, front panel keyboard
User to Control	I	Information to insert into the information stream
User to Control	C	Keypad, Radio control parameters
User to Environment Adaptation	I	Information to insert into the information stream
User to Environment Adaptation	C	Environment adaptation parameters
Antenna	Au	Interface to share information between antenna functions for co-site interference mitigation, etc.
RF	Au	Interface to share information between RF functions for coordination of multi-channel operation
Modem	Au	Interface to share information between Modem functions for coordination of multi-channel operation
Control	Au	Interface to share information between Control functions for coordination of multiple radio system operation
Environment Adaptation	Au	Interface to share information between environment adaptation functions for coordination of multi-channel operation
User	Au	Interface to accommodate multiple user control (local, remote)
Wireside to IO		Standard wire side interfaces for data, voice, LAN, and multimedia
Fill Device to Secure IO		Standard fill device interface
Remote Control to Control	C	Control parameters

3.2 Implementation Models

An architecture is the basis for the design, construction, modification, and operation of a product. It is derived from the design principles and it affects the physical configuration, functional organization, operational procedures, and data formats. This section is divided into three separate subsections. It will develop the mapping and modeling from the generalized SDRF architecture to the next level of definition for handheld and mobile units, identify the existing standards that would be affected by the SDRF approach, and offer suggested extensions to the existing standards for accommodating the operation within those environments.

Table 3.2-1 displays the differences between "Handheld" and "Mobile" systems. Besides form factor and power/performance constraint differences, the most striking difference is that, with minor exceptions, handheld systems support a single standard, single session at a time, while mobile systems tend to have requirements for supporting multiple sessions and sometimes multiple standards simultaneously. An exception in the single service/standard scenario for a handheld is the combination of a discrete paging standard and a voice standard in a single handheld.

Table 3.2-1 SDRF Differences between Handheld and Mobile/Stationary Systems

CRITERIA APPLICATIONS	Handheld Single standard environment	Mobile/ Stationary [*] Multiple standard environment
Prerequisites:		
Power Consumption	Minimum	Moderate
Weight	Minimum	Moderate
Volume	Minimum	Moderate
Price	Minimum	Moderate
Resulting in:		
Receive Elements		
receiver dynamic range	Moderate	Maximum
noise level	Moderate	Minimum
sensitivity	Moderate	Maximum
out-of-band undesired signal behavior	Moderate	Minimum
in-band undesired signal behavior	Moderate	Minimum
Transmit Elements:		
amplifier nonlinearities	Moderate	Minimum
backdoor intermodulation	Moderate	Minimum

[*] Definition of Mobile/Stationary: Any multiple standard/terminal installation in a transportable mobile or stationary application, e.g., van, ship, aircraft, shelters, ground station, and headquarters.

3.2.1 Handheld Models

Figure 3.2.1-1 is one model that can be used to describe the functional units in a handheld unit. This model has evolved from early-dedicated analog baseband implementations to today's digital implementations and reflected common practice in dividing functions into subsystems. In single-mode, single-band implementations those subsystems are dedicated to support single modulation techniques, protocols, data representations, etc. For an example of functions typically found in the different subsystems, please refer to Table 3.2.1-1 later in this section.

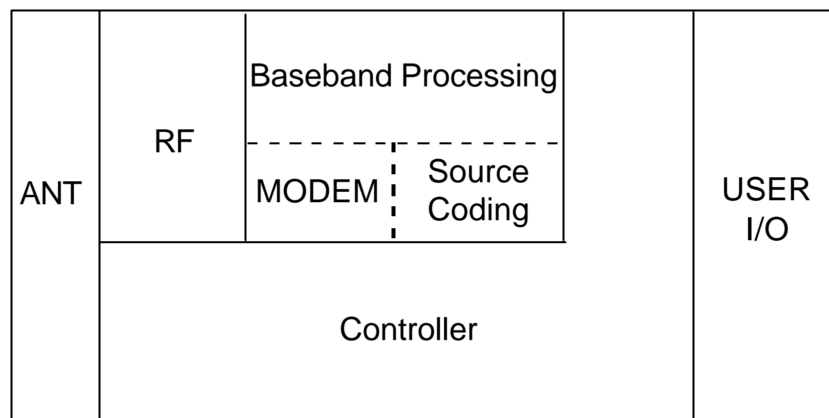


Figure 3.2.1-1 Single-Band, Single-Mode Handheld Functional Model

Figure 3.2.1-2 is an example of mapping the SDRF high-level functional model described in Section 3.1 to a typical single-mode, single-band handheld functional model. A subscriber identification module (SIM), derived from the GSM, may be included for security or privacy functionality.

The example maps a SDRF reference model into a grouping of functions. The top level view provides a handset context diagram. The bottom view shows how functionality may be hosted by the use of personality modules. Data is uploaded or downloaded through control interfaces.

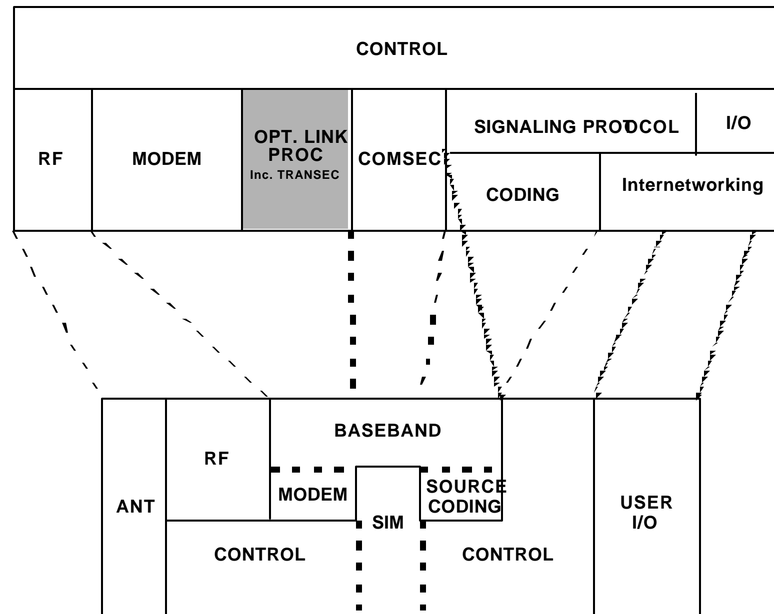


Figure 3.2.1-2 SDRF Mapping into Single-Mode, Single-Band Handheld Functional Model

Figure 3.2.1-3 shows a first iteration of how a typical single band, single standard model can be extended to cover multiple standards and bands using multiple devices. This view is burdened by the dedicated function approach typical of previous single standard single band implementations. An evolutionary view is shown in Figure 3.2.1-4 where the multiple standards and bands are integrated. The user interface, in this figure, is shown as two types, a human input interface and machine interface, typically a data terminal.

In looking for a more helpful model of a software defined radio used in handheld applications, it is useful to look at a generic computer model. Figure 3.2.1-5 shows a generic computer hardware/software model.

Applying this hardware/software model to the multimode, multiband extension model yields Figure 3.2.1-6. The handheld multiple service model, Figure 3.1.2-6, takes the generic handset mapping diagram, adds another level of detail, and converts it into a representation that is more computer-centric; at the bottom is a hardware layer, then a system software layer, and finally a service software layer.

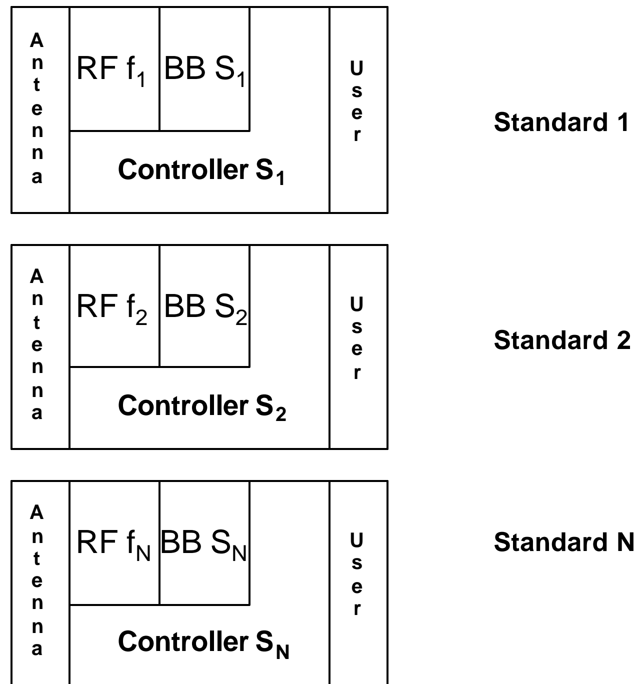


Figure 3.2.1-3 Multimode, Multiband Solution Using Multiple Single Standard Devices

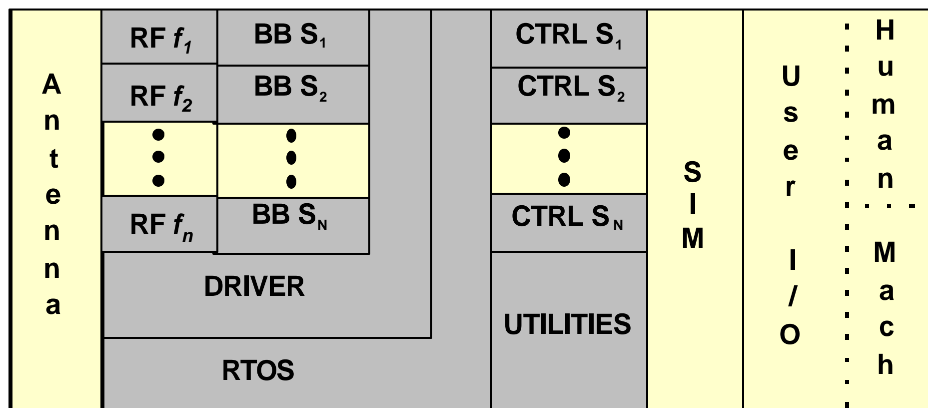


Figure 3.2.1-4 Multiband, Multimode Handheld Functional Model

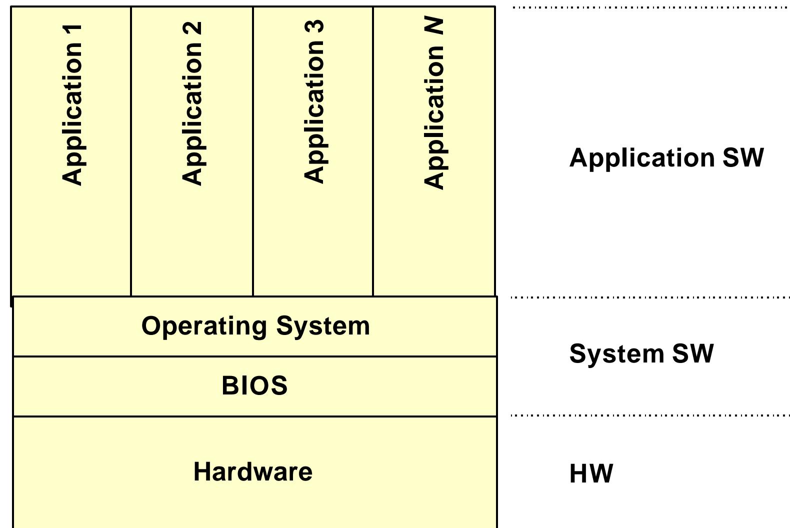


Figure 3.2.1-5 Generic PC Hardware/Software Architecture

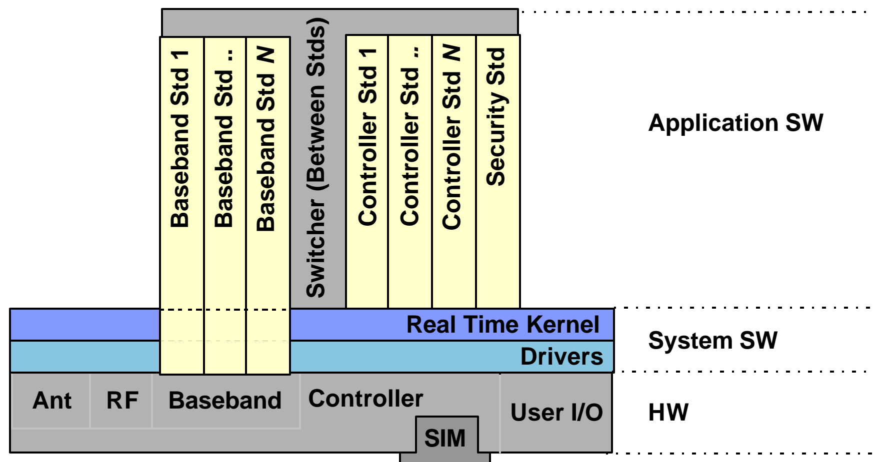


Figure 3.2.1-6 Handheld Multiple Service Model

The baseband implementations for each service are shown as cutting through the system software layer and directly interfacing the hardware layer because of the stringent performance constraints on execution speed and power consumption. A variety of technology approaches are being pursued depending on the constraints of the particular application. Battery power, size, weight, and cost requirements typically push the state-of-the-art in handheld units. In order to achieve processing speed and efficiency, the majority of baseband implementation are programmed very close to the underlying hardware or logic, using low-level languages such as microcode or assembly code. The task of switching between multiple bands using the same or different RF hardware is managed by a combination of the service switcher and the controller services for each individual operational mode.

Executing on the real time kernel (RTK) are two special service software modules: the service switcher and security services. The service switcher coordinates the selection and execution of the appropriate baseband service and controller service. It is both a peer and a master of the baseband service and controller service modules. As a master it supervises their execution. As a peer, it depends on them for support and for providing control. The security services module monitors and manages the COMSEC and TRANSEC security resources of the system. Security services use security configuration information contained in the SIM to enable or disable various security services. COMSEC security processing would require a routing of the data path between the source coding and channel coding functions in the baseband module through a COMSEC processing function, as pictured in Figure 3.2.1-2.

If the basic wireless communications system is combined with machine intelligence to make a portable information appliance, it is sometimes called a PDA (Personal Digital Assistant) or HPC (Handheld PC). It may be desired to combine some of the communications processing requirements with some of the information processing requirements and execute them on the shared system resources. Figure 3.2.1-7 shows how this can be accommodated in the handheld multiple service model.

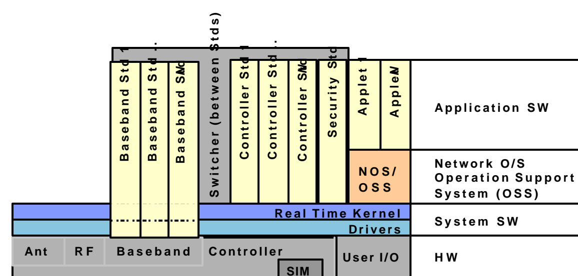


Figure 3.2.1-7 Handheld Multiple Service Model With PDA Extension

NOS/OSS (network operating system/operating support system) is a ‘shell’ that executes on top of the RTK. It has different service requirements than the controllers. For example, it may allow more liberal interrupt policies, etc. The NOS/OSS must have a SDK (software development kit) that allows users, carriers and manufacturers as well as software developers to easily develop, field, and support

applications. At this time there are two notable models for this element: Netscape/Java and Microsoft Exchange/"Active X." There is a high rate of innovation in this area in the industry at this time and there may be other possible solutions for this element.

Applets provide functionality that can be resident on the handheld device, in the network, at a remote site or some combination of the above. Applets provide the user with such functionality as computer applications, computer assisted communications, intelligent agents, etc.

Multimedia handhelds and wearables can be supported by adding a resource manager for the user interface as shown in Figure 3.2.1-8. This resource manager mediates between the basic communications functionality and two types of interfaces. The first is an array of physically attached ('Local') interfaces that support both human users and attachment to other machine intelligence. The second is an array of interfaces distributed around the user's body and connected to the basic communications functionality by a Personal Area Network (PAN).

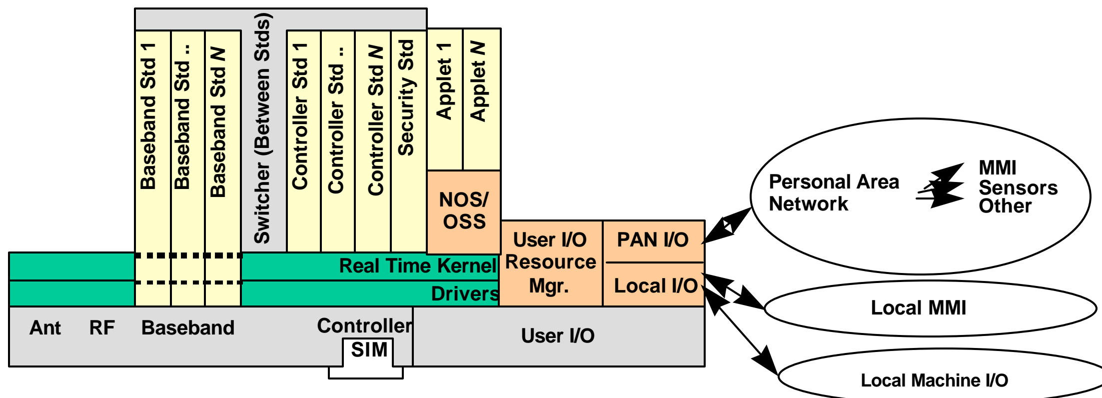


Figure 3.2.1-8 Wearable Multiple Service Model With PDA Extensions

The handheld multiple service model can be considered a combination of logical and physical architecture representations. Due to continuous innovation, technology evolution and the small form factor driven by the various applications for handheld units, the number of stable physical interfaces are very limited. Most of the interfaces are only stable on a logical or API level.

Potential API's could be

- RTK/Service Switch
- RTK/Security Service
- Service Switch/Baseband
- Service Switch/Controller
- RTK/Driver
- Controller/RTK

Potential physical interfaces identified:
- Antenna (passive & active) to RF
- RF to baseband
- User I/O to local machine
- Battery
- SIM

Software Download
A major requirement of a software-defined/adaptable handset or terminal will be the ability to reprogram the device, via download of new software and/or parameter data to the terminal. This will permit for example:
- Download of a new *user application*;
- Download of a new *graphical user interface* (GUI) to change or improve the 'look and feel';
- Download of a protocol stack, physical-layer configuration software, and control software to implement a *different air-interface* standard;
- Incremental download of new software and/or parameters to *improve performance* (for example, a modified source codec);
- Download of *software bug-fixes* (both applications and physical-layer/control software)

This feature is a key differentiator of software defined handsets from traditional single-standard implementations, and offers significant advantages of flexibility to manufacturers, service providers, and users.

Methods by which software may be downloaded include:
- Installation of new software from a new SIM card;
- From another computer, via for example a PC-card (PCMCIA) link;
- From a networked terminal;
- Software download over-the-air.

Each download method raises issues to be addressed in compiling standardization recommendations. For example:
- Security of downloaded code;
- Integrity of downloaded code;
- Standard API for establishing a download link to the handheld terminal;
- Billing.

More general standards issues relating to 'application and configuration' software download might be:
- Choice of terminal-independent language for 'programming' the terminal (e.g., A language such as JAVA may be used for higher-level applications which are sufficiently abstracted from

specific hardware features, but would be less suited to re-configuring the radio link, which may employ specific proprietary hardware features);

- Ownership and licensing of downloaded software;
- Type approval of both terminals and associated software applications making use of operators' resources;
- Matching new features and applications to the 'capabilities' of the terminal at which they are targeted. This could lead to classification of handheld terminals and applications, such that an application may be downloaded and executed *only if* the underlying terminal capabilities can guarantee that the application can be supported (e.g., processing power, memory, and deterministic task execution).

Software-defined reconfigurable terminals can potentially reduce the requirements for *de jure* standardization within the terminal, allowing functionality to develop in unison with terminal technology developments. This can be the case only if the means of programming the terminal and of ensuring its compliance subsequent to programming, can be precisely defined and managed: this will be a major work area for SDRF in producing standardization recommendations.

Table 3.2.1-1 provides examples of the decomposition of each of the modules in the handheld architecture. The breakdown is intended to provide a reasonable, comprehensive list of functions and subfunctions that are typically associated with each of the modules.

Table 3.2.1-1 Example Functions in Handheld Functional Model Subsystems

Category	Function	Sub Function	Notes
Antenna			
	Transducer		
RF			
	Frequency conversion		
	Linearization		Predistortion
	Amplification/Attenuation		
	Frequency selection		
	Frequency de/spreading		
	Pulse shaping		Equalizer, Filter
	Diversity processing		Rake receiver
	Modulator/Demodulator		
	Energy measurement		
	Antenna control		
	Spur management		
Baseband			
	Frequency conversion		
	Frequency selection		
	Frequency de/spreading		
	Pulse shaping		Equalizer, Filter
	Diversity processing		
	Modulation/Demodulation		
	Energy measurement		
	Antenna control		

Category	Function	Sub Function	Notes
	Spur management		
	Media Access Coding		Walsh Coding
	Channel coding		
		Forward Error Correction	
		Framing	
		Multiplexing	
		Interleaving	
	Channel estimation		
	Acquisition		
	Tracking		Freq./Phase/Code, Time
	Linearization		Predistortion
	Source Coding		
		Speech	
Baseband		Voice Activity Detection	
		Data in Voice	
		Still Image	slow & full motion
		Video	
		Data	
		Audio	
		Telephony Signaling	
Controller			
	Network Adaptation		
		Bridging	
		Routing	
		Repeating	
	Network Control		
		Spectrum Sharing Management	
		Registration	
		Mobility Management	
		Media Access Control	
		Link Control	
		Service Switcher	Service Detection, Service Selection, Cross Service Handoff
	Information Security		
		User Authentication	
		Traffic Encryption	payload
		Network Encryption	preamble, …
		Transmission Security	Transec
		Key Management	
		Node Authentication	
User I/O			
	MMI		
		Speech Recognition	
		Handwriting Recognition	
		Image recognition	
		Biometric recognition (Speech, eyeball, handwriting, keyboard,	

Category	Function	Sub Function	Notes
		pointer)	
		Image scanning	
		Speech synthesis	
		Display management	
		Audio management	

3.2.2 Mobile Models

Information Transfer System

Historically, mobile systems have been called "radios," and have been used primarily for voice communication. With increasing need for both voice and data, and with the greatly increased capability brought about by the use of digital data services the term radio has become overly limited. In this context, we refer to them as "information transfer systems" to reflect this additional capability.

The essence of mobile information transfer systems is their use of radio frequency circuits to permit operation from other than a fixed location, independent of a ground-based infrastructure. They may be capable of being transported from one operational site to another, or they may be capable of operation while in motion. They do not have permanent connections to land line networks or power grids, but may take advantage of those support resources when the resources are available.

Mobile information transfer systems are differentiated from fixed systems by their ability to move. They are differentiated from subscriber handheld units by their scale. They are physically larger and heavier, and function with more extensive capability, approaching that of permanent sites. Typical requirements have more extensive network interconnection than handheld units, and may offer more RF channels. For example, a typical cellular PCS handset supports one standard at a time where a mobile unit will encompass supporting multiple simultaneous services.

Critical Factors for Mobile Radios

This section describes factors that are especially important for mobile radio systems but which are not particularly important for hand-held radios. Many of these factors also apply to fixed base station implementations.

Scalability. Mobile implementations will span a wide array of possible platforms. Maritime requirements range from one or two circuits per platform to as many as tens of circuits per platform; cargo ships require one or two simultaneous channels, passenger ships require support of multiple simultaneous telephone calls, and aircraft carriers or other naval command ships require several tens of simultaneous circuits or services. Extensibility implies modular software implementations that allow replication of functionality to support multiple simultaneous instantiations of services; to at least a few hundred replicas, and perhaps to numbers limited only by word lengths, memory size, or other hardware factors. It implies hardware modules that can be replicated as needed on a supporting bus structure. It

implies chassis design that can be flexible—designed for few modules where few are needed, but capable of implementation in larger configurations for more demanding applications. It implies an I/O structure that can be sized to meet platform needs without modifying the basic architecture or implementation concept.

Upgradeable. It should be possible to upgrade the mobile radio without replacing the entire radio. This of course includes software upgrades, which are a central feature of “software” radios. For mobile units, it applies to the hardware as well. This means the potential to replace modules with new, more capable modules. It may mean in some way expanding the chassis to accommodate additional modules. Hardware upgrade for the mobile radio is in contrast to highly integrated and compact hand-held units where hardware upgrades are largely a matter of 100 percent replacement.

Higher-level Control Interface. In larger installations, control of the radio system may not be self-contained. That is, radios may be considered part of a larger electromagnetic systems suite, with control of the entire suite residing outside the confines or domain of the “radio” system. In a military shipboard environment, radio systems must co-exist with radar systems, electronic countermeasures systems, identification systems, and others. In the US Navy, these systems are under cognizance of a Command Control Warfare Commander, who has his own set of management tools. The modular software radio must provide an interface to allow control interaction (control acceptance, status reporting) with some higher-level control system.

Co-site Operation. The mobile radio system must be able to support multiple services, ranging from a few to a few hundred simultaneous circuits, very many of which may be physically distinct (as opposed to virtual) circuits. Naval ship installations often include one hundred or more antennas, with perhaps a dozen or more individual channels (frequencies) multicoupled onto a single antenna. Radio systems must operate in proximity to very high power radars, with instrument landing systems and TACAN, with IFF systems, with navigation receivers, and numerous other electronic systems. Aircraft and land vehicles may have fewer simultaneous operating electromagnetic requirements but also have much less space for antennas and other equipment. The modular software radio must be able to operate in, and in fact be designed to mitigate, difficult co-site interference environments.

Form Factor (Affordability). The form factor selected for the mobile modular software radio must balance performance and cost. Production volume is a major cost factor. Is it possible to identify a form factor that is suitable for high-volume, minimum cost commercial applications, while being adaptable to the more environmentally stressing and lower volume military applications? Can the adaptation also be accomplished without dramatically increasing costs?

Backfittable. In some cases, particularly for aircraft radios, the space available for a new radio is the space occupied by the old radio. This constrains the selection of physical module form factors. The selected implementation for the mobile modular software radio should be implementable within the constraints of existing overall aircraft radio sizes.

Distributed Implementation. Radios for mobile applications will frequently be implemented in a distributed fashion. User interfaces will be conveniently placed for the user(s). The antenna(s) will be located on the outside surface of a vehicle. And other parts of the radio will be located to meet other criteria such as available space, environmental conditions, or transmission line losses. Modules, chassis design, and other form factor issues will need to accommodate desired distributed implementation flexibility.

Figure 3.2.2-1 shows a generalized information transfer thread. It shows how an information transfer system connects an information source to an information sink with a transformation in the center. The normal functionality is an RF channel on the left side and either a handset or a network connection on the right side.

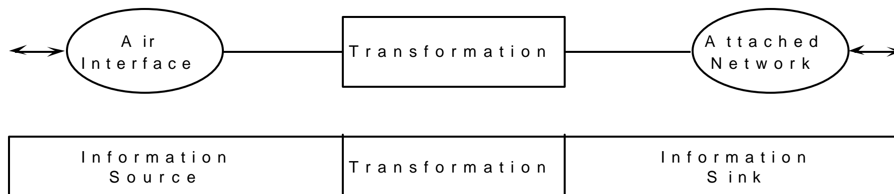


Figure 3.2.2-1 Information Transfer Thread

Anticipated future SDRF mobile systems tend to be differentiated from handheld systems by their availability to support multiple simultaneous information transfer threads. These simultaneous information threads may be multiple functional instantiations on a single physical platform or with multiple physical platforms for each function.

Figure 3.2.2-2 shows the configuration of information transfer threads in a typical mobile information transfer system.

The left side has one or more air standards, each utilizing resources assigned to that standard. Each channel has an independent operation and each standard may have more than one instantiation. The right side has one or more internetworking connections that serve to deliver voice, data, video, or facsimile to a local or remote information user. The internal processing and transfer function provides the detailed actions of baseband, RF, and control services. The control services set up connections between these elements, and operate under direction from the user interface. Bridging and routing is accomplished by connecting two or more left side or right side elements to each other.

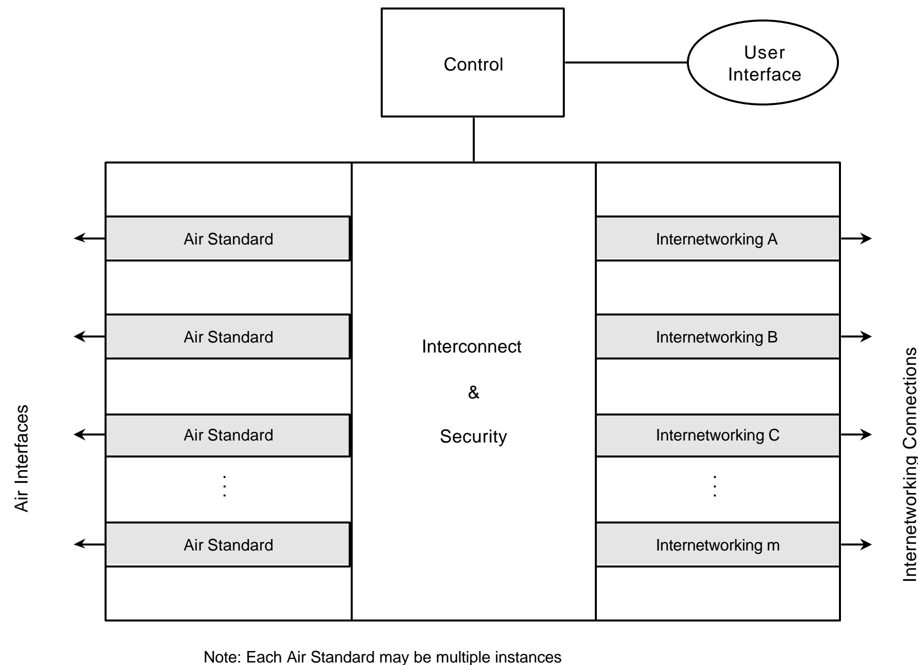


Figure 3.2.2-2 Mobile Information Transfer System Logical Structure

Mobile systems typically require modular, multimode, and simultaneous operation. As shown in Figure 3.2.2-3, this implies multiple instantiations of each function within the context of a multiple input, multiple output system. These multiple functions may occur as physical replications of the function or simply as multiple software instantiations operating on a single processing platform. The information flow among the functional modules is under the control of the distributed control environment.

The modular nature of a mobile SDRF radio allows individual functions to be accomplished internal or external to the SDRF structure. For example, routing and COMSEC functions can be performed external to the SDRF without loss of generality. In other cases functions may be replicated, such as information security which may be included as part of the message process flow between routing and a user interface. The intention of the SDR Forum standards recommendations is that all modules have defined interfaces and control processes so that "plug and play" of the employed modules operates effectively.

Figure 3.2.2-4 relates the US Navy's Joint Maritime Communications Strategy (JMCOMS) architecture to the SDRF high-level functional model as an example of a mobile information transfer system structure with external functional access.

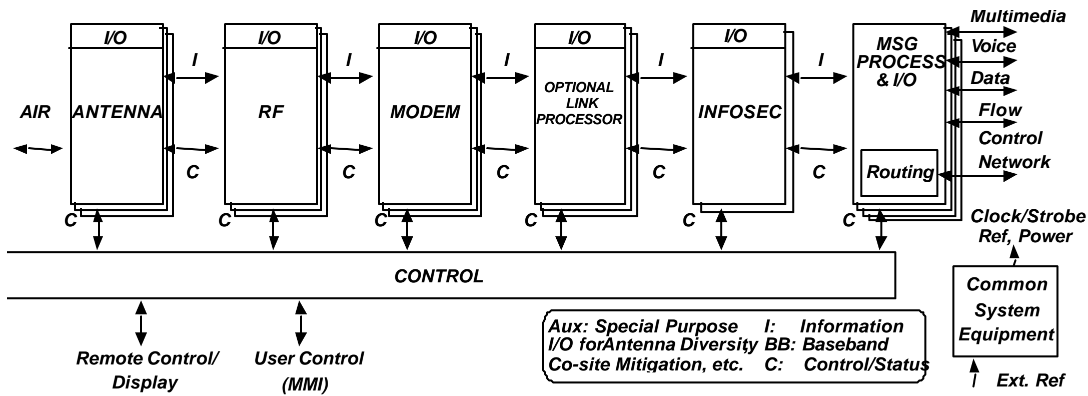


Figure 3.2.2-3 Multiple Instantiations of Each Function of Modular, Multimode Operation

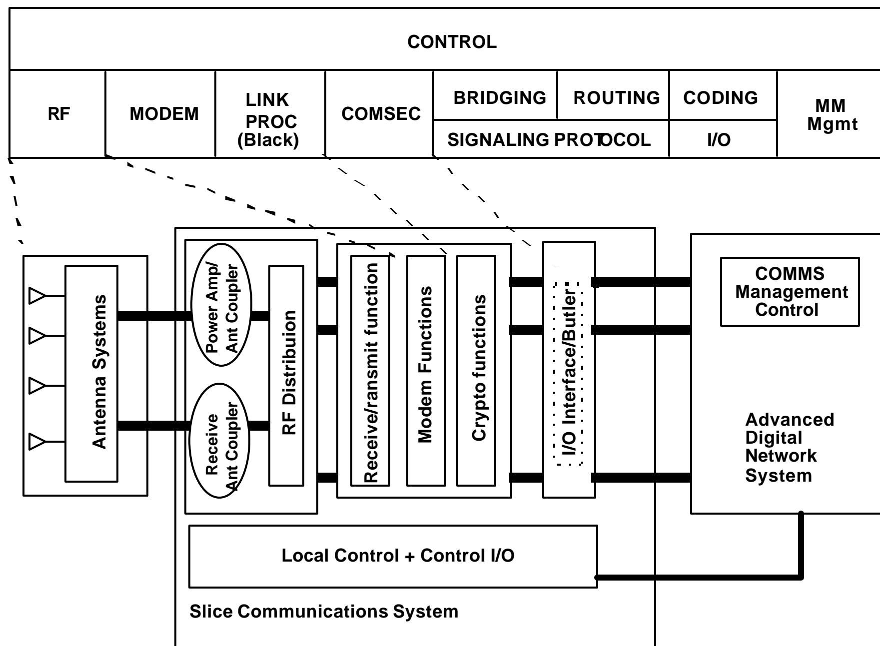


Figure 3.2.2-4 SDRF Functional Model Mapped into a Joint Maritime Communication Strategy (JMCOMS) Application

3.2.3 Base Station / Satellite Models

3.2.3.1 Candidate high level Use Cases

3.2.3.1.1 Base Station application

3.2.3.1.1.1 Base Station arbitration. The Base Station System and user or network control processor conduct arbitration to determine the capabilities available to each prior to the initiation of a change in service or level of functionality.
Some examples of the subjects to be arbitrated include:

- supported air interface standard, the supported minimum and maximum data rates
- capability to transition between service types (voice/data)
- capability to download or upload different service options
- capability to perform interference cancellation
- capability to perform modulation type changes
- level of hardware and software complexity among other thing

3.2.3.1.1.2 Time sensitive services. The Base Station co-ordinates the offering of time sensitive services that are offered to the mobile only during a specified time window.
Some examples of time sensitive services include: the offering of certain capabilities for trial use for reduced or no fee during a special promotional period, or the offering of these same services during the specified user session.

3.2.3.2 Smart Antenna and Base station

3.2.3.2.1 Base Station service change voice/data. The Base Station converts from one type of communication service to another, this function is valid for all air interface standards.
Some examples of the service change include; conversion from IS-136 voice service to IS-136 low rate data service, conversion from high rate data services (video) back to voice service only among other things.

3.2.3.2.2 Quality of service management. The Base Station co-ordinates the reconfiguration of the receive function, the transmit function, or both receive and transmit functions, to maintain service at current data rates under adverse conditions without service deterioration.
Some examples of the reconfiguration include:

- upgrade from non-adaptive to adaptive signal processing to enable continued quality of service (e.g. BER) in the presence of co-channel interference
- upgrade in the modulation scheme chosen to enable higher transmission data rates in the same channel bandwidth among other things.

3.2.3.2.3 The Base Station co-ordinates service offerings to mobiles. The base station would offer a service, or combination of services, to the mobile, which the mobile would accept or reject based on the mobile's capability to accept the service and the mobile's desire for the service.
An example of the co-ordination shows the base station offering video based services at a certain cost per megabit to a mobile that is capable of receiving video data, perhaps in a stationary mode, but accepting that the mobile refuses the service based on the cost offered.

3.2.3.2.4 The Base Station co-ordinates a system upgrade. The base station is upgraded to a new level of performance on the supported standards.
Some examples of a new level of performance on the supported standards include:

- download of new software to repair deficiencies
- installation of a new vocoder
- installation of adaptive signal processing to include Multiple Access Interference cancellation, the addition of antenna elements among other things.

3.2.3.2.5 The Base Station co-ordinates a service change. The modification of base station software and/or hardware to support operation for a new service standard not previously available.
An example of this co-ordination includes the addition of GSM service support to a base station that could previously support only IS-136.

3.2.3.2.6 Base Station reconfiguration. The Base Station co-ordinates the adaptability and reconfigure-ability of the base station to modify any aspect of the system (including both an upgrade and/or downgrade of system performance).
Some examples of this co-ordination include: a change in system bandwidth, a change in space-time processing, a change in software programmability, a deletion of or addition to system hardware configuration, and the addition of more processing elements in either S/W or H/W, among others.

3.2.3.2.7 Base Station repartition. The Base Station co-ordinates the repartitioning of system H/W to support either different standards or new functionality.
An example of this repartitioning might be the reconfiguration of certain processing nodes (microprocessors or gate arrays etc.) or hardware components to support alternate programming or hardware functionality.

3.2.4 Switcher Downloader

Modular, multimode terminals can be fielded in a totally terminal-centric fashion in which the network has no knowledge of the fact that the terminal can change from one configuration to another. The terminal can initiate a session with network type A. When it comes to the border of that network technology, the handset can take down the session with network A and initiate a session with network B. It is also possible to have some intelligence outside the terminal to help coordinate this process. But to achieve the full functional advantages inherent in SDRF mechanisms, facilities are needed for cooperation and hand-off between terminals and networks using different modes and different bands. These mechanisms should support terminal directed hand-offs, network directed hand-offs, and combinations of the two.

Other standards bodies are responsible for maintaining the standards that govern each specific service (single mode/band). It is SDRF's intention to work with these other bodies to develop an umbrella model and recommendations for message types and protocol extensions. Then these other bodies can use, each in their own domain, the model and recommendations to develop terminals and networks that cooperate across modes and bands. This section will present some examples of the requirements from several perspectives and an approach to meeting them.

Commercial cellular/ PCS users need hand-offs between service types and service providers. This requirement is derived from a combination of economic factors, limited spectral resources, legacy systems, geopolitical forces, etc. Examples include CDMA/TDMA/AMPS hand-offs as well as cordless, wireless LANs, CSMA, etc.

Military users require interoperability among systems. Both within a sovereign service branch and within multinational, multiservice branch operations, peacekeeping policy, and direct assistance type operations, there is a requirement for secure use of commercial infrastructures. Interworking and translation are required, especially for legacy systems. Defense users typically are confronted with many different radio systems.

Civilian/aviation needs simultaneous operation across several modes/bands. For example, interoperability requirements among different branches/standards include the provision of different aviation services for an analog voice, MSK, TDMA, CSMA, etc. Today, hand-offs are often done by voice command.

Emergency service coordination requires the support of many standards with significant interworking/translation requirements.

An example of one approach to addressing these requirements to support handoffs between CDMA cellular/PCS and TDMA cellular/PCS in a US standards environment is using AMPS as a bridge. In US cellular standards environments, CDMA and TDMA handsets are required to also support AMPS

mode. Figure 3.2.3-1 shows how AMPS can be used to carry signaling information between the mobile unit and the infrastructure.

Another approach is to provide extensions to the message formats in both environments that allow the mobile unit and the infrastructure to negotiate the desired service type to use for the session or session continuation. Figures 3.2.3-1 and 3.2.3-2 illustrate this approach.

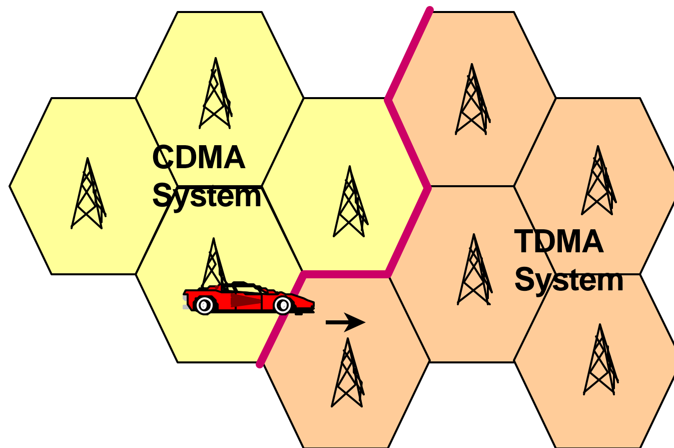
For a fully capable SDRF implementation, there should also be message formats to trigger and deliver from the infrastructure, if necessary, software modules to allow the mobile unit to configure itself to support the negotiated service.

Similar scenarios can be constructed for GSM/DECT, PDC/PHS, terrestrial cellular/LEOs, etc. Furthermore, third generation standards that are emerging are pointing to multiple, non-compatible services with similar handoff requirements. There is also consideration of support for legacy services within third generation systems, which also would encompass similar handoff requirements.

In general, consideration for extensions to each of the other services standards need to take into account the following protocol groups, as applicable:

- Handoff protocol extensions,
- Signaling protocol extensions,
- Key distribution protocol extensions,
- Channel selection protocol extensions,
- Routing protocol extensions,
- Configuration protocol extensions,
- Timing protocol extensions,
- Billing protocol extensions, and
- Administration protocol extensions

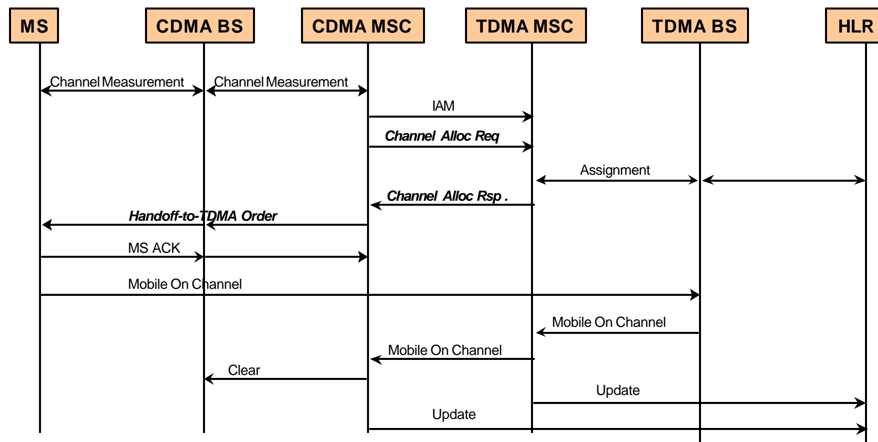
This list will necessarily be expanded as each service type is analyzed in light of the SDRF open architecture.



Signaling Strategy:
CDMA DS/MSC notify TDMA MSC of hand-off
CDMA and TDMA MSCs notify HLR
CDMA BS notifies MS of change in mode after hand-off
Alternately: MS switches to AMPS after crossing, then negotiate for TDMA

MSC: Mobile Switching Center
HLR: Home Location Register
BS: Base Station
AMPS: Advanced Mobile Phone System

Figure 3.2.3-1 Signaling Strategy



Almost identical to inter-MSC TDMA to TDMA handoff
Hand-off will be “hard” hand-off
Requires *new signals*
May have interference problems at boundary

Figure 3.2.3-2 Cross Standards Handoff

3.2.5 Smart Antenna Definitions

A sub-System which includes the antenna (and possibly other classes) that uses the spatial domain in combination with decision based signal processing to improve link performance and enable other value added services. It consists of both the software and the hardware objects associated with the additional processing capability.*

*Smart Antenna processing can be thought of as a basic capability that can be broadly applied to any time division, frequency division, or code division multiple access air interface standard in a similar, but not necessarily identical, way.

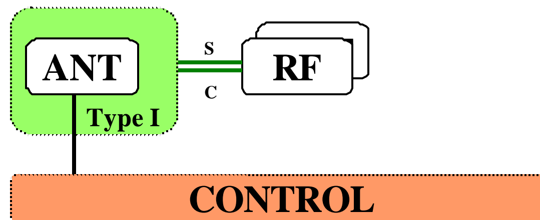
The Working Group will consider several classes of antennas, which may include adaptive processing. Figure 3.2.5-1 shows Type Ia, the simplest of these classes. The Type Ia antenna consists of a single antenna with a single feed, which may be diplexed for transmission and reception. This type of antenna may be rotateable, and may include yagi, log periodic, loop, or other designs. The Type Ia antenna interfaces with the RF and control functions.

The Type Ib antenna system (Figure 3.2.5-2) also interfaces with the RF and control functions. The Type Ib extends the Type Ia class by adding multiple antenna elements, which may include RF combining within the antenna.

The Type II antenna system (Figure 3.2.5-3) extends the Type I antenna system by means of processing within the RF function, which may include RF or IF combining.

The Type III antenna system (Figure 3.2.5-4) further extends the Type II antenna system by means of processing within the baseband processing (modem) function, which may include baseband combining.

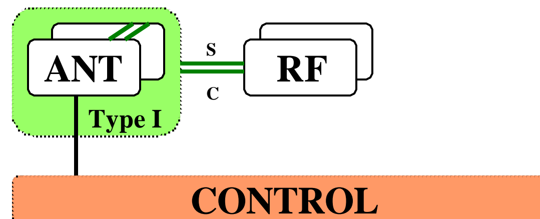
Type Ia Antenna System



•Single antenna
–May be LPA, yagi
•May have rotator, antenna tuner, diplexer
•Single feed

Figure 3.2.5-1. Type Ia Antenna.

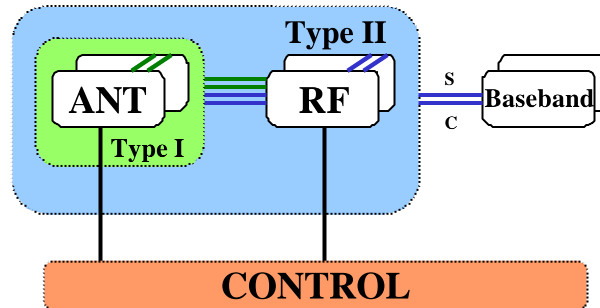
Type Ib Antenna System



- Multiple element antenna
- May have rotators, antenna tuner, diplexer
- May have RF combiner (e.g. Butler matrix)
- May have single or multiple feeds

Figure 3.2.5-2. Type Ib Antenna

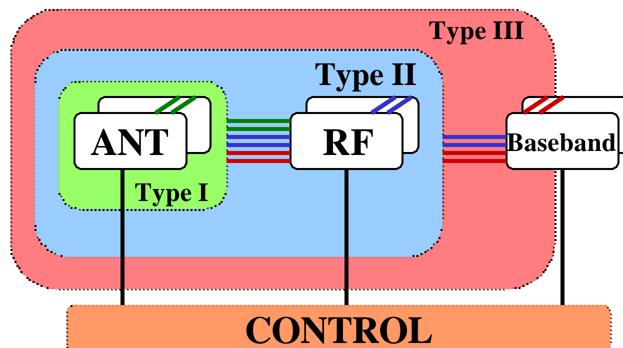
Type II Antenna System



•Type I antenna system plus:
•One or more RF
•RF may have RF combiner (e.g. beamformer)

Figure 3.2.5-3. Type II Antenna.

Type III Antenna System



- Type II antenna system plus:
- One or more baseband processors (e.g. modems)
- May have baseband combiner (e.g. beamformer)

Figure 3.2.5-4. Type III Antenna.

3.3 SDRF-Compliant Interface Use

3.3.1 Summary

Users, operators, and vendors want to improve the flexibility and capability of their wireless communications systems. Software programmable radio system technology offers the potential of substantially improved operational capability at a lower cost then a collection of point solutions.

Development of technical standards by the SDR Forum opens the way for open standards that encourage development of new functionality for existing equipment, aid in introduction of new applications, and facilitate introduction of new technology as it emerges. Substantial life cycle cost reduction is made possible by the combined impact of lower non-recurring expense (NRE) and economies of scale due to common hardware.

To provide new functionality to the field is accomplished by one of the following means:

- Design a whole new system
- Modify an existing system in the laboratory
- Provide field-installable new modules

The SDR Forum technical work enhances each of these approaches. A product life cycle time domain is presented to assist in understanding how the SDRF work fits into the application environment.

3.3.2 Why Do Users Need The SDRF Solution?

The users of wireless communication environment need communications capability to support execution of their responsibilities and attainment of their objectives. The users we refer to span a wide range of endeavors, including individuals in business, the military, public safety, civil government, and consumers. Their communication equipment may be cellular telephones, radios installed in vehicles, ships, or aircraft, or equipment for portable or fixed site operations.

Users want to add new features to existing equipment, implement new applications, roam or use the same hardware in different areas or with different systems, or take advantage of new technology. Often they want to expand the existing capability, but hesitate to invest further in equipment implemented with old technology.

The SDRF technical work is striving to provide solutions to these problems by defining open architectures that can be used to provide a common structure across many product implementations, to provide new capability by incremental upgrades of fielded equipment, and to reduce both product cost and NRE.

3.3.3 How to add functionality to an existing system

The following options are available to users that need to improve their existing capability.

3.3.3.1 Design a whole new system

This is the historical way of approaching the problem. Combining the experience from previous systems with the currently available technology, a completely new system was devised. In many cases, particularly military radios, the new products did not interface with the currently fielded equipment. The current hodgepodge of point solutions restricts operational flexibility and incurs excessive life cycle cost for maintenance. In the cellular market the result is that a single phone cannot operate with all of the fielded cellular and PCS systems.

Although it involves substantial NRE, designing a whole new system does optimize performance and use of technology. It can minimize unit production cost, and if production volume is high, result in low life-cycle cost.

A contribution is made by the SDRF technical work if the new system design is SDRF compliant. That is because the software granularity of the SDRF approach permits substantial reuse of software modules in future reworks of the newly designed system. Further, if the architecture is so structured, the new system will be able to incorporate field upgrades through download of new software. Provision may be made for hardware modules such as PCMCIA cards to introduce new hardware or software functionality.

3.3.3.2 Modify an existing system in the laboratory

When we bring an existing system back into the development environment we open up a number of possibilities for improvement. The system architecture is laid out on the table, and the development team can revisit most of the design tradeoff that were originally made. The approach can range from almost designing a new system to making minor adjustments to incorporate some new software or hardware module.

In the development environment all of the product documentation should be available, and much of the expertise that designed the system in the first place. Existing components can be revised or rebuilt. Software wrappers can be built to interface legacy modules with new ones. Performance enhancements can be developed. Development, test, and debug facilities are available. So much flexibility is available that restraint is often needed to avoid changing more than is absolutely needed.

Of course all of this capability comes at a cost in engineering resources. It should be far less costly than building a whole new system, but more expensive than a capability that can be loaded in the field.

If the project has a requirement to become SDRF compliant during the project then future enhancements will be much easier. Many future enhancements can be inserted in the field. Those that cannot will necessitate another development project, but the SDR imposed granularity and compatibility will work to reduce the amount of work required.

3.3.3.3 Provide field-installable new modules

The most efficient way to provide new capability to an existing system is to do a software download over-the-air (OTA). Close behind an OTA download is a software insertion from a smart card or a PC connection. A description of these capabilities is an important part of the SDRF technical work.

In order for these capabilities to be realized, the original system architecture must have made provision for it. The appropriate APIs must be available for integration of the new modules. And the hardware must be capable of supporting the new capability. Distribution channels must support delivery of the new modules to existing system.

Given that these provisions have been made, field-installable upgrades have substantial potential for flexibility and cost reduction. It can, in fact, provide true "plug and play" capability.

3.3.4 The Compatibility Domain

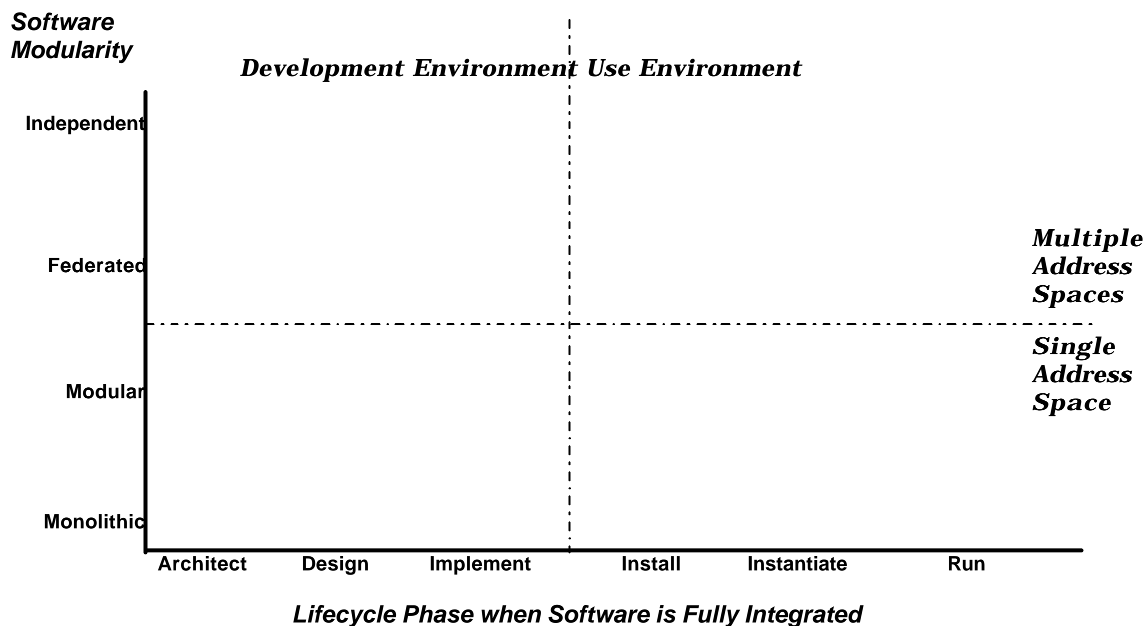


Figure 3.3.4-1 The Compatibility Domain

Figure 3.3.4-1, The Compatibility Domain has two axes. One is Software Modularity, the degree of independence of the modules in the system. Software Modularity is the degree of independence of the software modules in the system.

> **Monolithic** code is typical of legacy systems where a need for efficiency was deemed paramount. It uses branches and local function calls, and is difficult to reuse.

Modular systems have source code arranged in files that can be compiled separately. Within those files the modules refer to each other by function calls, and provide a degree of information hiding. As installed in the system it is typically one large load with function references resolved.

Federated software consists of sets of software that function in separate processing units, but that were intended to work together. They have some coordination and communication mechanism.

Independent software units are developed independently. Any commonality is derived from the architectural level, with facilities to permit them to go through a communication process to exchange capabilities and interact.

The other axis is a representation of most of the product's life cycle. It represents the time in the life of the product when the software for an application is fully in place, loaded, and ready to run. A major break point exists at the point when the product is released from development, put into production, and replicated many times in the use environment.

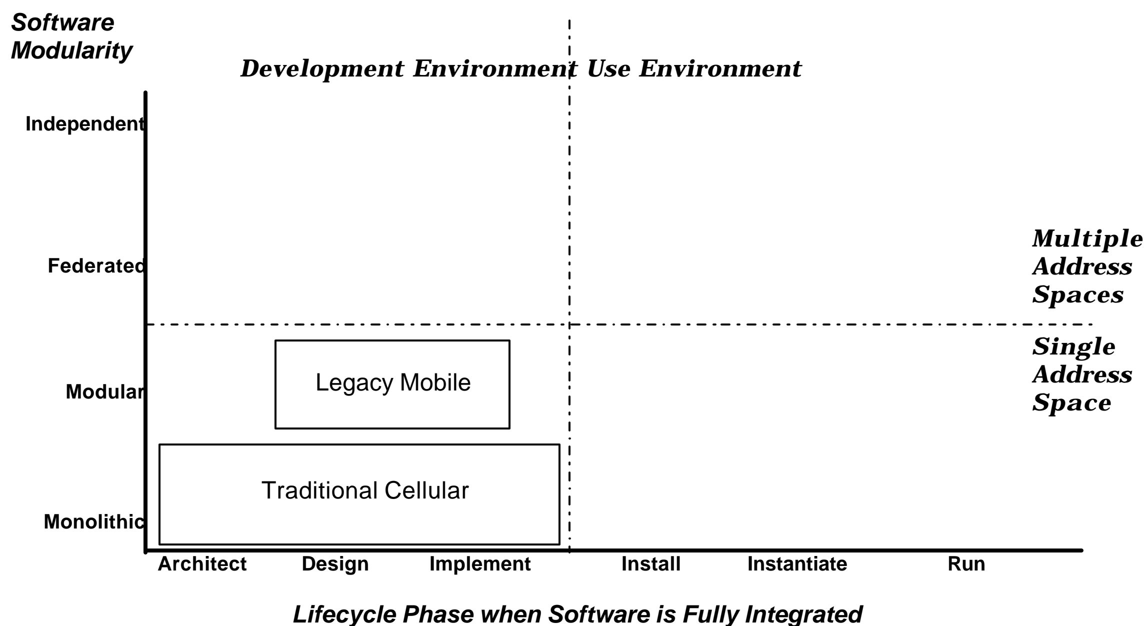


Figure 3.3.4-2 Past Systems

As shown in Figure 3.3.4-2, Past Systems, legacy equipment has often been implemented with a low degree of granularity, and with little provision for adding capability in the field. The analog cellular handset is a good example of such a closed system. When more capability (or just a smaller unit) is desired it is completely replaced. Legacy mobile products have been more modular, but making changes has required redesign. Changes to individual units are occasionally possible with maintenance at depot or factory level.

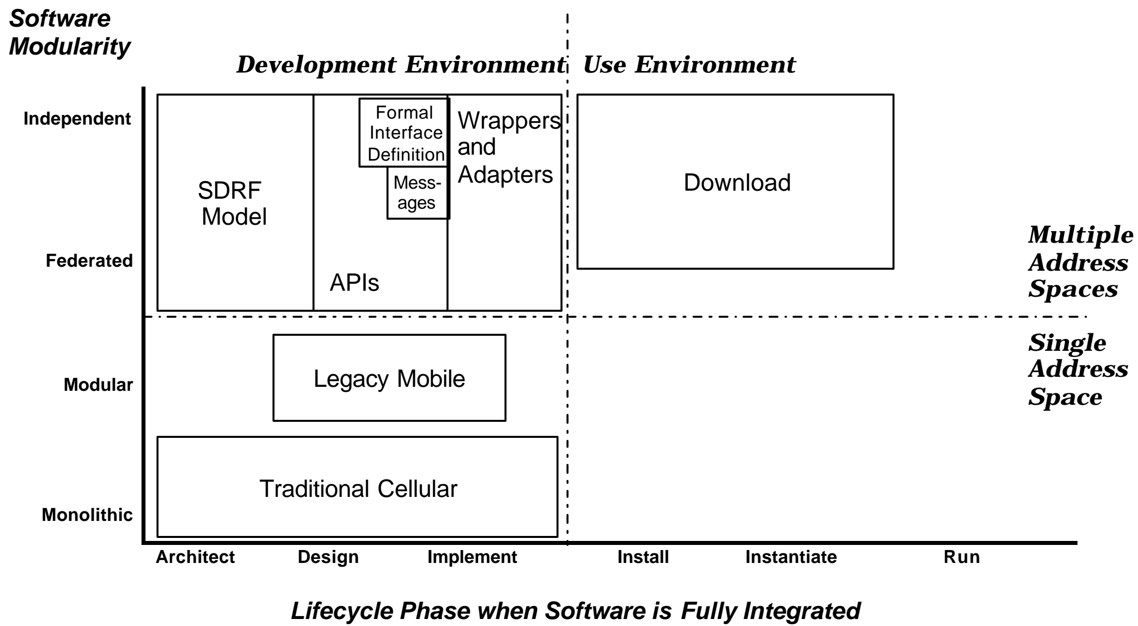


Figure 3.3.4-3 The SDRF Contribution

Figure 3.3.4-3 indicates where the SDRF technical work fits. The SDRF Model is at the architectural level to describe interfaces between major subsystems in the unit. The SDRF APIs are used at design time to establish the details of communication between system components. Then, if needed, wrappers and adapters can be developed to provide interface with legacy systems.

With these facilities in place, the opportunity for future enhancements by the use of the SDRF download capability is available.

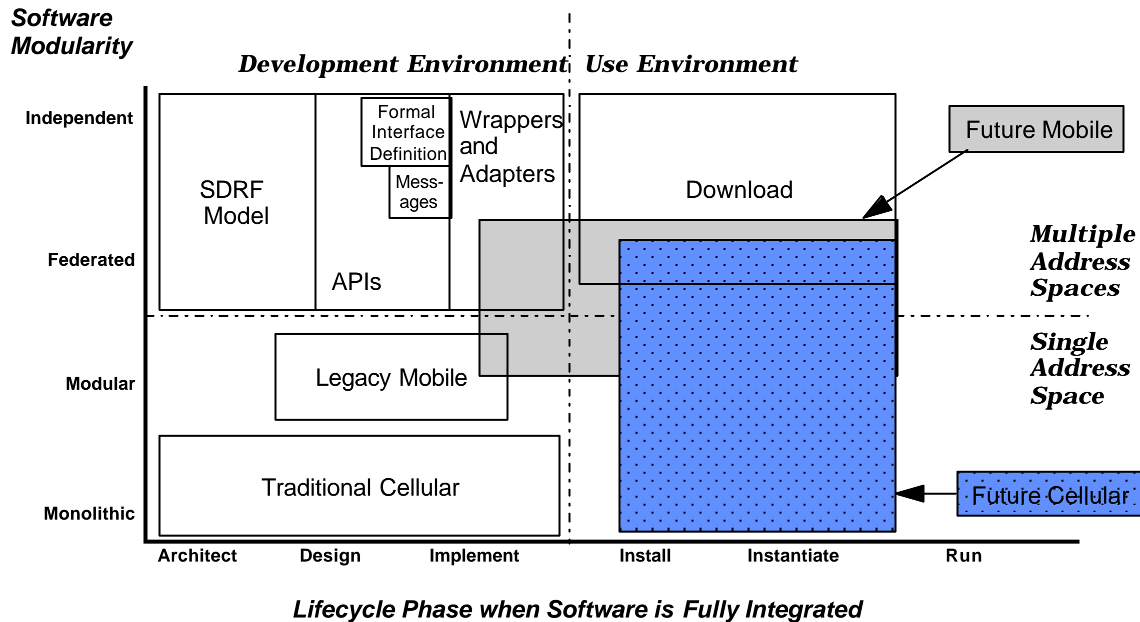


Figure 3.3.4-4 Future Systems

As shown Figure 3.3.4-4 future mobile and cellular systems will provide greatly increased capability for new functionality and flexibility through SDRF compliant download in the field. This approach is the most likely to provide us with universal roaming capability.

3.3.5 Benefits

Software programmability provides a capability to change the functionality of a radio system by activating alternative software applications archived in the equipment. When it is necessary to add new software to that already installed in the system, it can be done in either the development or use environment.

Bringing the system back into a shop or laboratory environment provides many options for modifying or even redesigning the equipment. It is, however, costly.

Software download in the field provides flexibility and convenience, but it is possible only if the original design has provided that capability. Such facilities will become increasingly common in the future.

The SDRF open system definition will assist in changing system functionality under either scenario. By opening up the architecture, application developers in a wide range of fields will be able to take advantage of the advantages of software programmable radio systems.

4.0 Application Program Interface (API) Design Guidelines

4.1 Structure for Development

This section is concerned with the development of APIs to define SDRF standard devices. Section 4.1.1 is a design guide that defines and explains the process that is used to create the APIs that form the core of the standards definition. Section 4.2.2 defines a set of generic control messages that can be used as a template to create specific functions by adding functional specific information. It is envisaged that this template and the functional APIs will create a catalog of functions that SDRF architectures can utilize.

4.1.1 Background

Earlier parts of this document provide high-level descriptions of the SDRF architecture, provides for alternative views of that architecture, and recognizes that the information known about the system may vary over time or that differing perspectives may be needed. The term architecture is defined as the organizational structure of a system, identifying its components, their interfaces, and a concept of execution among them.

- Components are the named "pieces" of design and/or actual entities of the system and are composed of one or more modules of software (SW), firmware (FW), and/or hardware (HW).
- Interfaces are the relationships among component modules in which the modules share, provide, or exchange data.
- Concept of execution represents the dynamic relationships of the modules. It can include such descriptions as flow of execution control, information flow, dynamically controlled sequencing, state transition diagrams, timing diagrams, priorities among units, handling of interrupts, etc.

This section provides a generic framework for the definition of such APIs. The objective is to provide a common set of rules, guidelines, and definitions, which will form the basis for defining software interfaces. The term Application Program Interface (API) is used in this section to mean the interface definition. It provides levels of abstraction to capture and refine the interface design information from general concepts to implementation-specific details.

4.1.1.1 Specifying the system

One of the classic problems faced with any complex project or design is the failure to scope out and understand the complete amount of work that is involved. Too often, there is a gross underestimation of the amount of effort involved – especially in the area of software- and the time taken to specify and integrate the system together. Similar problems can also be encountered when extending or enhancing existing systems

The difficulty facing the industry today is that there is a change-over occurring within radio design that is blurring the system in terms of whether software or hardware is used for the implementation. Although the term API is used to describe these interfaces, it has come from a purely software background and on first inspection is not applicable to other interfaces such as hardware busses and form factors. The problem is that technology that is implemented using programmable devices and software today may be implemented tomorrow in hardware. To allow the successful replacement of modules by different solutions i.e., replacing a software based module by a hardware version, the interfaces which define the boundary of the module must be designed in such a way so that the interface does not imply or exclude a potential implementation. If it does, then its appeal and thus its ability to attract products that conform and thus be reused will be limited.

This means that the development of APIs actually is the development of interfaces, which may or may not be software or hardware only. The term API within this discussion should be interpreted as simply an interface document and therefore can be applied to any part of a system such as a bus, form factor, and so on. It is not simply restricted to software.

The chocolate chip cookie approach[1] to system specification must be avoided. A good way to do this in the initial stages is to create an interface layer diagram that defines the components and how they interface together. It should be remembered that this type of diagram is not a replacement for other modeling techniques such as data flow diagrams or structured analysis but does provide an excellent starting point on which to build. This type of diagram identifies not only the components but the interface mechanisms and definitions that are often overlooked. The model uses only two fundamental terms to explain all the various layers of software and their interfaces.

4.1.1.2 Application Program Interface (API):

As defined earlier, an API is the interface definition, which is a description of the relationships among related software and/or hardware modules, such as the bi-directional flow of data and control information. It is not a lump of code or a program or an application. An API describes the relationships of modules, not the implementation of those relationships. In many ways, this causes a slight problem because an API is an abstraction and not a physical entity.

4.1.1.3 Software and Hardware Modules:

In general a module is the actual implementation that uses the interfaces to receive information, process it and output it. As far as the interfaces go, they should be independent of the implementation, i.e., the module that is sandwiched between them.

---

[1] This is where individual modules -- the chocolate chips -- are identified and are assumed to make up the complete system without taking into account the amount of integration and other system components -- the cookie mixture-- that is needed to complete the system.

A software module is the piece of code that does the work. It uses the interface(s) to communicate with other modules and performs the tasks, the conversions, and so on. Applications, operating systems, device drivers are all software modules.

A hardware module is a piece of hardware that also uses the interfaces to communicate and process information. **It should be interchangeable with a software module that uses the same interfaces**.

These definitions lead to the essential golden rule: as modules and interfaces are layered together to create the overall system structure, they must alternate. Two adjacent module or interface layers are not allowed.

To use a car analogy, the driver and the car are both software modules and they can only work if there is a software interface layered between them. With a car, the interface is the knowledge that the middle pedal is the brake for cars with manual gear boxes, and moving the steering wheel to the right will make the car go to the right. The knowledge or interface does not turn the wheel or move the wheels, that is done by the driver and the car.

It is important that the interfaces are understood and adhered to. If they are not, then the system will not work correctly and it will be difficult to change individual components. Returning to the car analogy, imagine the chaos if the brake and gas pedals were swapped around!

4.1.1.4 Visual Representation

A good way of seeing the interaction between modules and interfaces is through layer diagrams where the various layers are depicted as either interfaces or modules. To be consistent throughout this document, the following symbols are allocated to both interfaces and modules. The first diagram shows the use of the design symbols for module and interface.

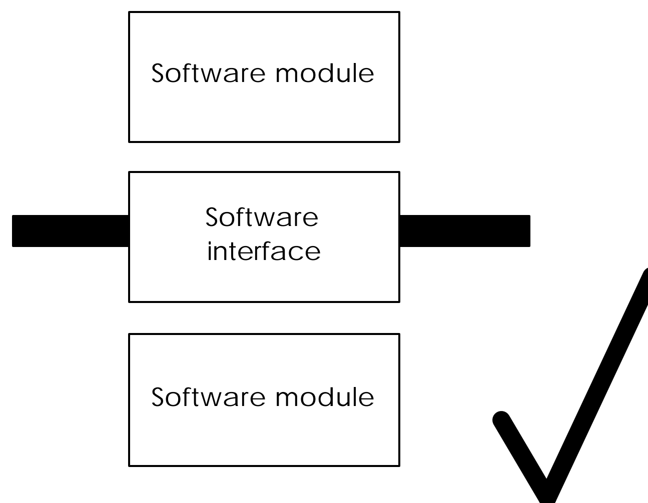


Figure 4.1.1-1 The Correct use of the Symbols for Software Interface and Module in a Layer Diagram

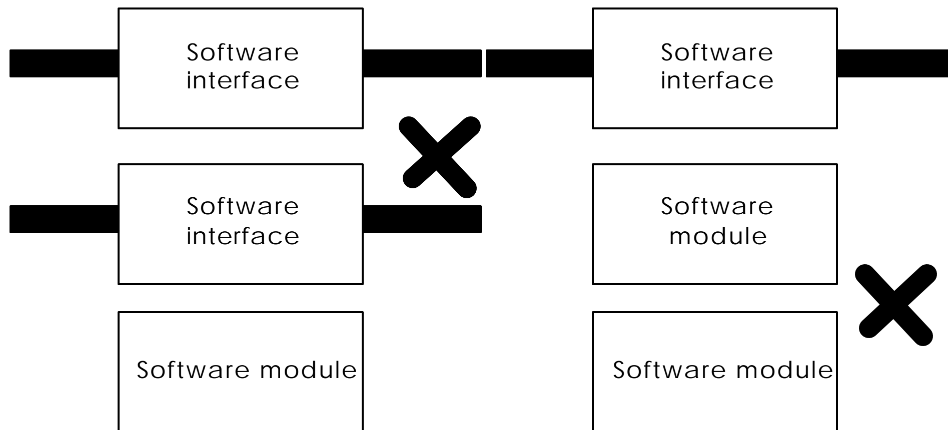


Figure 4.1.1-2 Wrong Use of the Software Interface and Module Symbols in a Layer Diagram

Figure 4.1.1-2 shows two incorrect uses of design symbols where two modules and two interfaces are improperly layered.

4.1.2 What interfaces are required?

It is virtually impossible to define a set of APIs that cover all aspects without causing confusion or precluding new ideas. In practice, a complete set of interfaces needs to define several tiers to provide different levels of compatibility:

4.1.2.1 Tier 0 Architectural

The tier 0 set of interfaces is the highest abstraction level of the API structure and defines the radio architecture that is to be used. This is the starting point for API definitions.

Section 3 of the SDRF Tech Report presents the SDRF function interface diagram and demonstrates how the SDRF Architecture extends to the definition of functional interfaces. This interface diagram is shown here in Figure 4.1.2-1. The interfaces for a module are separated into an information interface and a control interface. These interfaces are bi-directional in nature. A representative information flow format is provided at the top of the diagram. Actual representations will be implementation dependent and specify the control or information that flows in ***and*** out of the modules on both sides of the interface. An information transfer is effected throughout the functional flow within the SDRF architecture to/from antenna-RF, RF-modem, modem-INFOSEC, and INFOSEC-Message Processing interfaces.

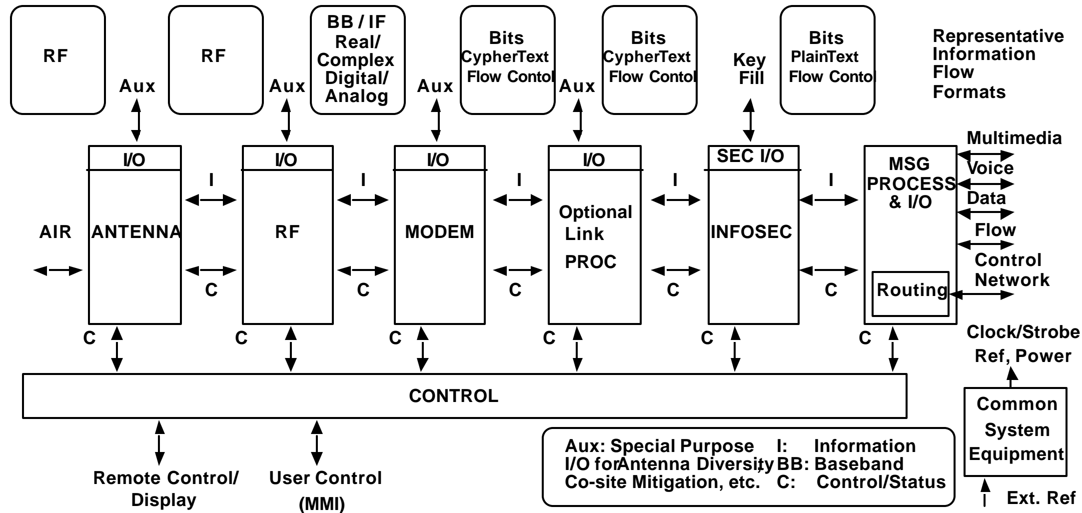


Figure 4.1.2-1 SDRF Functional Interface Diagram

Separate interfaces for information [I] and control/status [C] are shown in the diagram. For example, the RF-Antenna interface contains information as well as control/status whereas the Air-Antenna interface contains information only. The key terms used for the interaction diagram are:

- I: Information Flow Interface, i.e., information to be transferred over the communication link and information embedded in the signal-in-space waveform (e.g., training symbols, spread spectrum symbols).
- C: Control/Status Interface, i.e., information transferred for the purpose of controlling other functional blocks or for generic radio control functions.
- Aux: External interface to a similar block (e.g., antenna-to-antenna interface for co-site mitigation) on the same or other radio channel.

Using Tier 0 as the base, additional tiers, or layers, are added to refine the architecture to represent a specific implementation. These are shown in Figures 4.1.2-2 and 4.1.2-33 and described below in Tiers 1-3.

Each module that is defined in Tier 0 can be expanded to create a Tier 1 model for both control and information flows. This process is expected to be iterative with functions in the Tier 1 model also being expanded until sufficient detail has been identified. These subsequent iterations are also Tier 1 models but with more detail. Figure 4.1.2-2 shows this principle by taking the modem component from the SDRF architecture (Figure 4.1.2-1), the Tier 0 model, and showing how it can be expanded to create the Tier 1 information and control functions. As Tier 0 effectively defines the boundary for the model, all Tier 1 APIs and modules should be related back to the Tier 0 model. In addition to the Tier 1 models, there are additional Tier 2 and 3 models that provide additional information about the transportation of the messages and the physical attributes of the system (Figure 4.1.2-3)

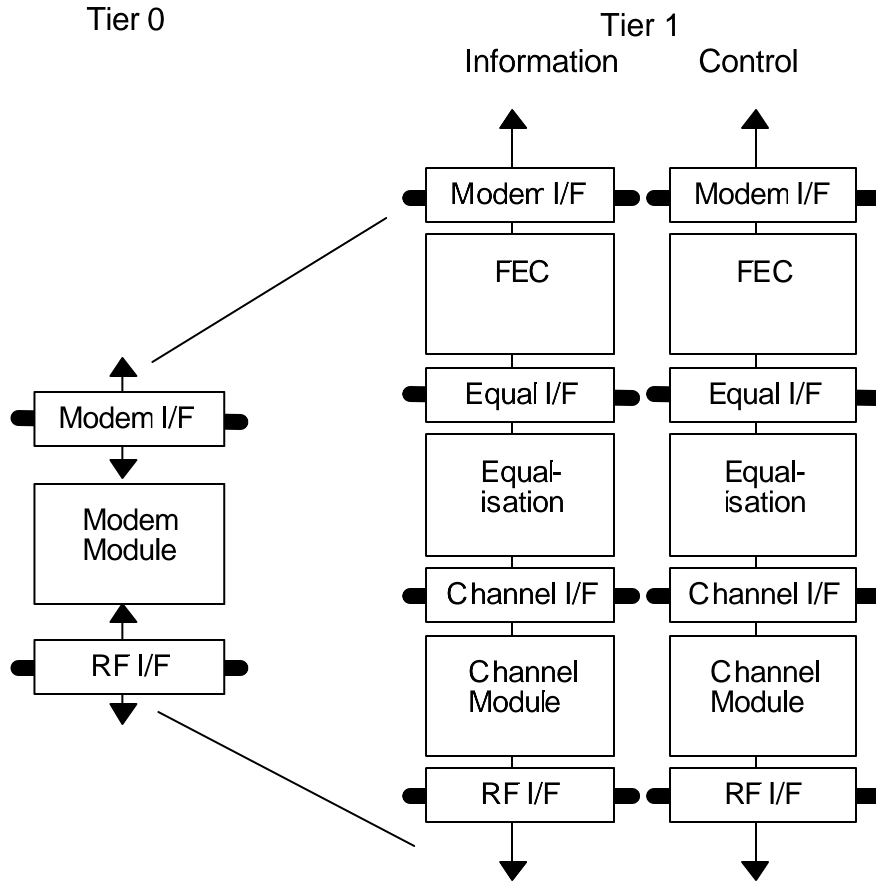


Figure 4.1.2-2 SDRF Expanding a Tier 0 Module to Create the Tier 1 Functions for Information and Control

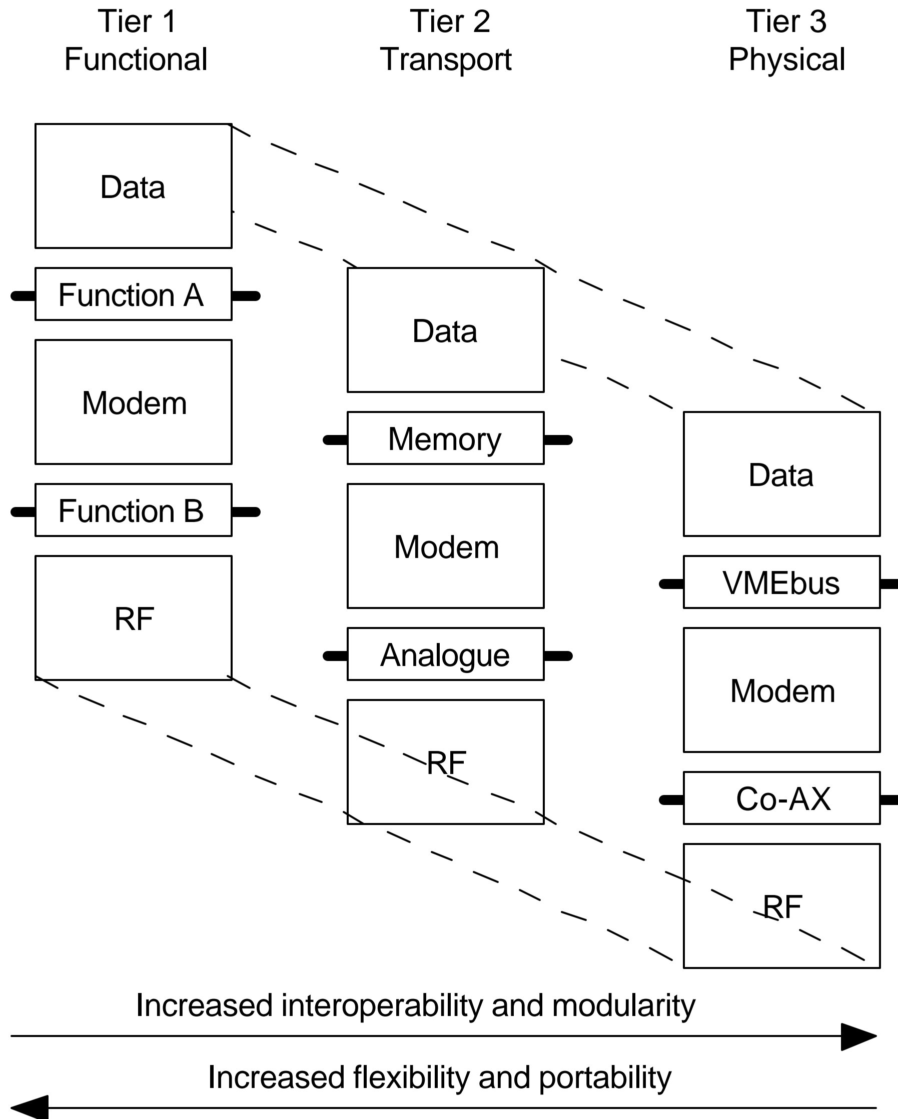


Figure 4.1.2.-3 SDRF Architecture and Interface Refinement Using Tiers

4.1.2.2 Tier 1 Functional

These interfaces describe the various functions that the system has to perform. They describe the boundaries of the various modules that need to be present within the system. They describe the messaging and interpretation. They do not specify how the message is transported. It is at this stage that the separation of the control flow from the information flow is explicitly defined. In most functional specifications, the information flow is treated separately from control. This will be explained in more detail later.

4.1.2.3 Tier 2 Transport and Communication

At this level, the transportation and communication is specified. This tier defines the methods and procedures used by the modules to exchange information. Here, the information content is of no interest because the real focus is in how the messages specified in the functional specifications are transferred.

With some implementations of APIs, the transportation is usually through function calls using common or shared memory e.g., C function calls. This usually assumes that there is a single processor. The same messages could be transferred across a LAN to another processor or via a serial link and so on. In this way, the implementation can use either single or multiple processors as required. It should not be assumed that this approach is restricted to a single processor.

4.1.2.4 Tier 3 Physical Factors

At this level, other factors can be defined such as the physical form factor, plug connectors, and so on. These are not necessarily important for commercial systems where silicon integration may radically change the form factors. However, for military or commercial public or aviation mobile systems that require standard plug in modules, this is essential.

Please note that when the term commercial system is used in this document, it refers only to consumer and other products such as mobile phones and not to aviation and public service radio systems. These latter systems have more similarities with military systems and therefore the term military should be interpreted as including these systems, unless otherwise specified.

4.1.2.5 What do the Individual Tiers Offer?

Essentially they all offer different levels of component reuse. The term component should be interpreted to include hardware, software, or hybrid implementations. By adhering to Tier 1 functional interfaces, components can only be reused if the transportation (Tier 2) and physical form factors (Tier 3) are addressed and re-engineered, as necessary. An example of this would be a software-based channel modem. The functionality would mean that the code would not need to be redesigned but the transportation may require some rework. It may use a memory-to-memory interface in the original design but with the new implementation, this interface is replaced by a serial connection. The level of reuse is still high, but some additional work is needed. For commercial systems such as mobile phones and other consumer products where implementations and form factors change rapidly, this is probably the best that could be hoped for.

With a Tier 1 and Tier 2 compatible module, the functionality and transportation are defined and therefore the only variable is a form factor. Again, for commercial systems, this is not a problem because of the rapid change in search for the smallest/lightest terminal. However, for military mobile systems, the fact that a module is functionally compatible and uses memory-to-memory transportation is not enough if it is based on a PCI card while the rest of the system is based on a VMEbus. This is where the third tier comes into play.

With all three tiers specified, we have true modularity and interoperability where modules can be physically removed and replaced by others that meet all three sets of interfaces without causing any additional re-work with the rest of the system.

It is important to realize that no one tier on its own can address all the software radio market. Different market segments will require different levels but all these levels can fit within a generic API architecture.

4.1.3 What Makes a Good Interface?

Before answering this question, it is important to reinforce the definition of an API or interface. Previously we have defined three tiers of interfaces and these have the following generic requirements:

4.1.3.1 A Functional Level Interface or API

- Describes the message and its contents.
- It does not describe how the message is transported.
- It is only a document!
- It must be unambiguous and have only one interpretation.

4.1.3.2 A Transport Level Interface

- Describes how messages are passed and does not describe the content.

ment!

- Again, it must be unambiguous and have only one interpretation.

4.1.3.3 A Physical Level Interface

- Describes how physical components fit together e.g., form factor, connectors, power supplies, and so on.
- Again it is only a document!
- Again, it must be unambiguous and have only one interpretation.

4.1.3.4 What Must an Interface or API Not Do?

The starting point in answering the question is to define what an interface should not do.

- It should NOT combine information and control flows.

They often need different interfaces and requirements, and combining them can cause conflict and compromise. For example, a RF module may provide a digital control interface but require analogue transmit and receive signals. It is clear in this case that these interfaces should be kept separate. However, if the analogue interface is replaced by a digital one, then the boundary is less clear. The case for keeping them separate is clear: although both interfaces are digital, the control interface is less time critical than for the RX/TX information one and therefore does not need the high-bandwidth interface that the RX/TX information does. Control data is frequently asynchronous in nature and imposing this on the RX/TX information bus can cause the RX/TX information to also become slightly asynchronous. This can be overcome by buffering but this then introduces delay. Then there is the

question of how to route the two signals down the same bus. The end result is a compromised design, which can be avoided by keeping things simple and separate.

In addition to these practical reasons, the explicit separation of control and information flow is normally required for a proper security design.

- It should NOT have two or more APIs or interfaces controlling a single resource without resource management.

This is often a major challenge for any programmable device where a resource such as processing, memory, or an I/O device can be controlled or shared between two independent controls. While not necessarily a problem, this structure should be carefully examined to see if a form of resource management is needed to resolve conflicts.

- It should NOT allow applications to bypass intermediate level APIs directly to lower levels.

The API should be independent of any other APIs below it so that its integrity is not compromised. Preventing the combination of simultaneous upper and lower level access also prevents potential compatibility problems. This is not intended to prevent the successful concatenating of modules together.

- It should NOT be under-specified.

APIs and interfaces need to be designed so that they can support future developments. This can be helped by the provision of extension mechanisms, which allow the orderly expansion of an interface while maintaining compatibility with previous versions.

- It should NOT have knowledge of the implementation or any APIs below it.

The definition of one API should not be dependent on the definition of another API. This is needed to provide a true black-box level of independence below an API or interface.

4.1.3.5 What an API should do?

A good interface will have the following characteristics:

- An API must be written so that it is understandable and testable.

Multiple descriptions or views may be necessary to adequately define an API. For example, graphs, figures, diagrams, and text may be combined to present a complete picture of the API. The API must be testable so that compliance can be verified.

- It will have reasonable granularity within the system partitioning.

This is important when considering re-use and providing an attractive environment for suppliers. The SDRF architectures described in this report provide high-level descriptions of SDR functions, such as modem and RF. However, when these functions are examined in detail, it is clear that they consist of several key technology components. For example, if the API is defined for a modem function as a whole, then the supplier of modem subfunctions such as channel equalization or echo cancellation software cannot supply products to meet the requirements because there is no intermediate API that allows their technology to be treated as a modular component of a modem. As a result, the potential to attract technology is greatly diminished, as the supplier will have to define their own interface. Without a

low enough level of granularity, the opportunities for multiple sources of components that can easily work together will be lost. Therefore, SDRF functions should be expanded and refined to the lowest feasible level of granularity as illustrated in Figure 4.1.3-1. This will support a broader market of module suppliers as shown in Figure 4.1.3-2.

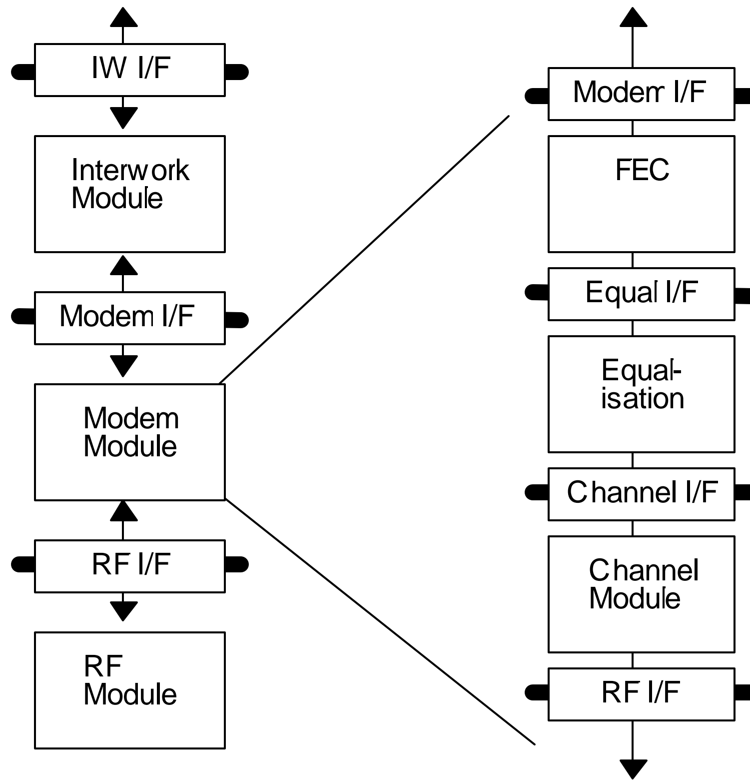


Figure 4.1.3-1 Expanding the API Granularity

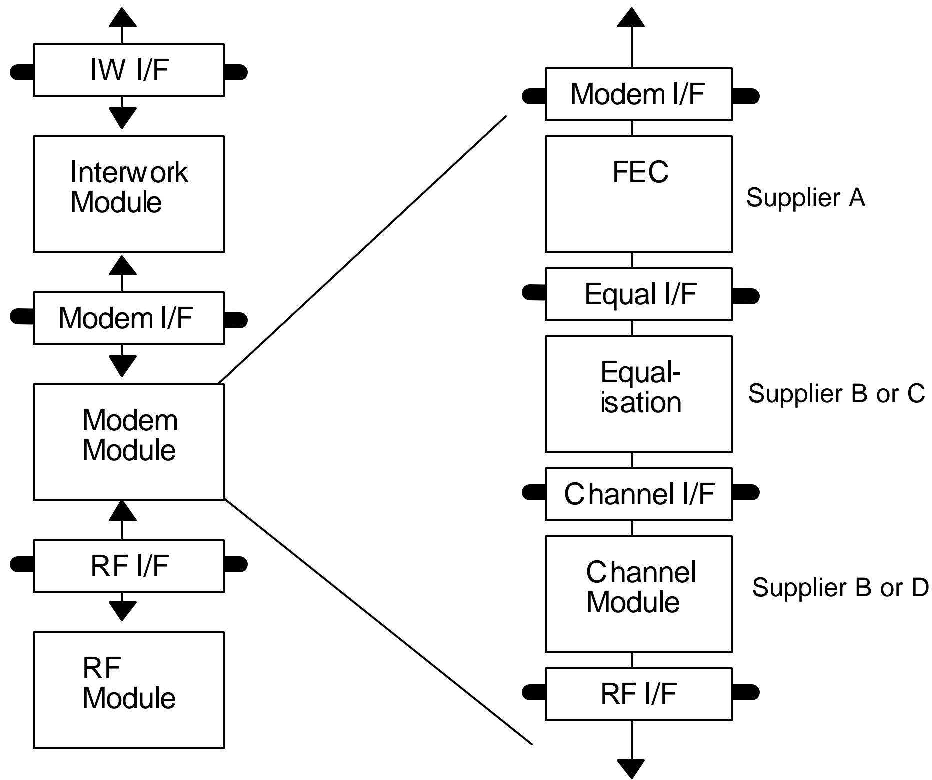


Figure 4.1.3-2 Using Increased Granularity to Access Multiple Technology Sources

This fine-grained approach provides developers with the freedom to mix, match, and combine modules like building blocks. Interfaces can be combined to create a composite of several functions to form a modem block that encompasses internally several APIs. In practice, it does not matter how the internals of the block are designed provided they meet the external interfaces, as shown in Figure 4.2.3-3 below.

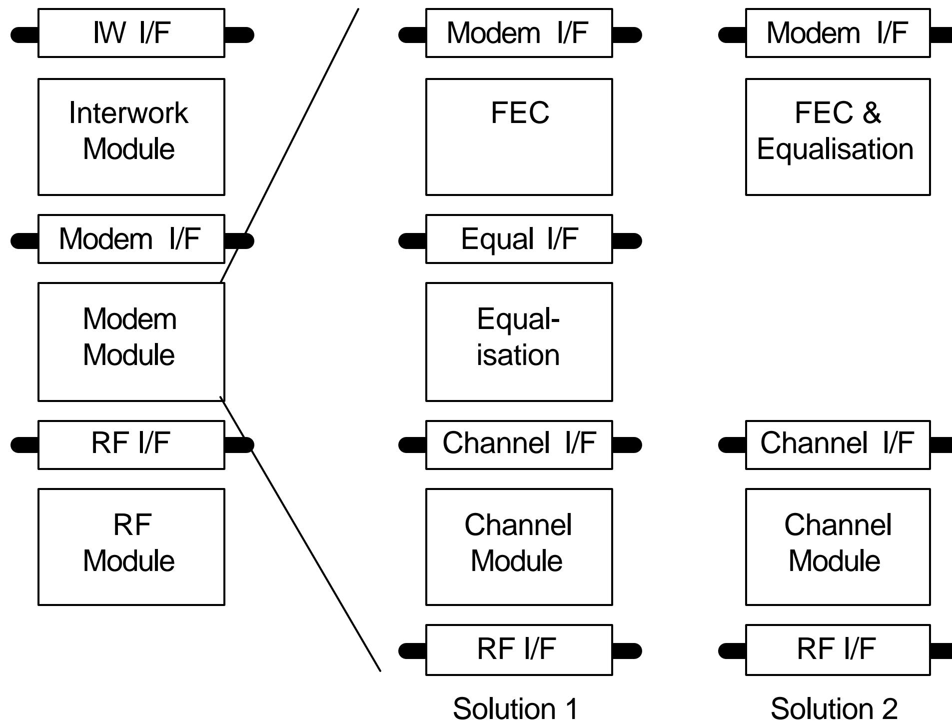


Figure 4.1.3-3 Combining Modules and Removing Interfaces to Provide Different Solutions. The Message Directions Have Been Omitted for Simplicity.

• It can be logically expanded while maintaining capability.

This is necessary to respond to new (and unforeseen) developments within the market place.

• It must address resource management.

This is needed to resolve any conflicts caused by resource sharing.

• It must address security concerns.

Any security constraints must be addressed when an API is developed because it affects design considerations.

• It must address timing requirements.

For proper operation of time-critical systems, information must flow in the correct sequence at defined time intervals. Timing information that must be exchanged between modules must be explicitly defined.

• It will identify and support capability exchange.

This is needed to allow modules to establish a common set of facilities that they can use. It is often incorporated with resource management to declare a capability (and reserve it) as part of the initial communication between two modules via an API or interface.

4.1.4 Capability Exchange

Capability exchange is a well-known technique and is used to allow two modules of differing capabilities to communicate successfully using a single API. It is used by Microsoft in its Telephone API (TAPI) specification [3] and in the H.320 video conferencing standards [4]- It is very closely related to resource management as the declaration of a particular capability, by default, will assume that resources are available.

The first question that typically arises when capability exchange is mentioned is why do you need it if you have a defined interface? There are usually three scenarios that can provide good reasons why an interface should be expandable and therefore require some way of declaring exactly the capabilities of module(s) on either side of an interface.

• Coping with interface revisions and enhancements.

While it is the intention of all concerned to define a single standard, the reality is that this is rarely the case. As new technology becomes available or as errors appear, standards will need revision or enhancement. If this happens, there is a normal requisite for backward compatibility to allow the maximum reuse of legacy components. The problem facing new components is in identifying which revision the component on the other side of the interface is using? This is where the capability exchange is important.

• Allowing the use of proprietary extensions.

It is very common in the PC world for various companies to offer combinations of interfaces or APIs to their products. This allows a user to choose between an open standard or proprietary implementation depending on the differing attributes that the interfaces have. For example, a PC graphics card may reproduce the VGA register set as well as offer a special Windows driver that accesses the hardware differently and gives improved performance.

The proprietary additions may range from simple extensions to a standard interface to a completely different interface. Obviously, using these proprietary additions locks in the owner to a specific implementation but this may be acceptable, and even preferable to the owner. In many cases, especially in the PC world, proprietary standards have quickly become *de facto* industry standards. It is important, however to provide a standardized way of adding these extensions and recognizing when they can successfully be used between two modules using a common API.

• Adjusting support to match the available resources.

In this case, the capability exchange is used to restrict an interface to support those functions for which resources are available. This is necessary when the module that uses the interface shares resources with other modules such as processing time and power, memory, peripherals, and so on.

4.1.4.1 Capability Exchange Implementations

There are typically two types of generic implementations for capability exchanges: at a command level and at an API level. With a command level, the API is designed such that any command which is not supported or not understood either due to the nature of the command or any associated parameters

must be rejected by a unique message that identifies exactly why the rejection has occurred. It is not acceptable to simply state that a command error has occurred because this is too vague to allow the sending module to determine exactly what has gone wrong.

The problem with this mechanism is that it can quickly become quite difficult to determine exactly what is supported and what is not. It becomes difficult to design and test state machines when several different versions need to be supported. As a result, the second approach is more often used.

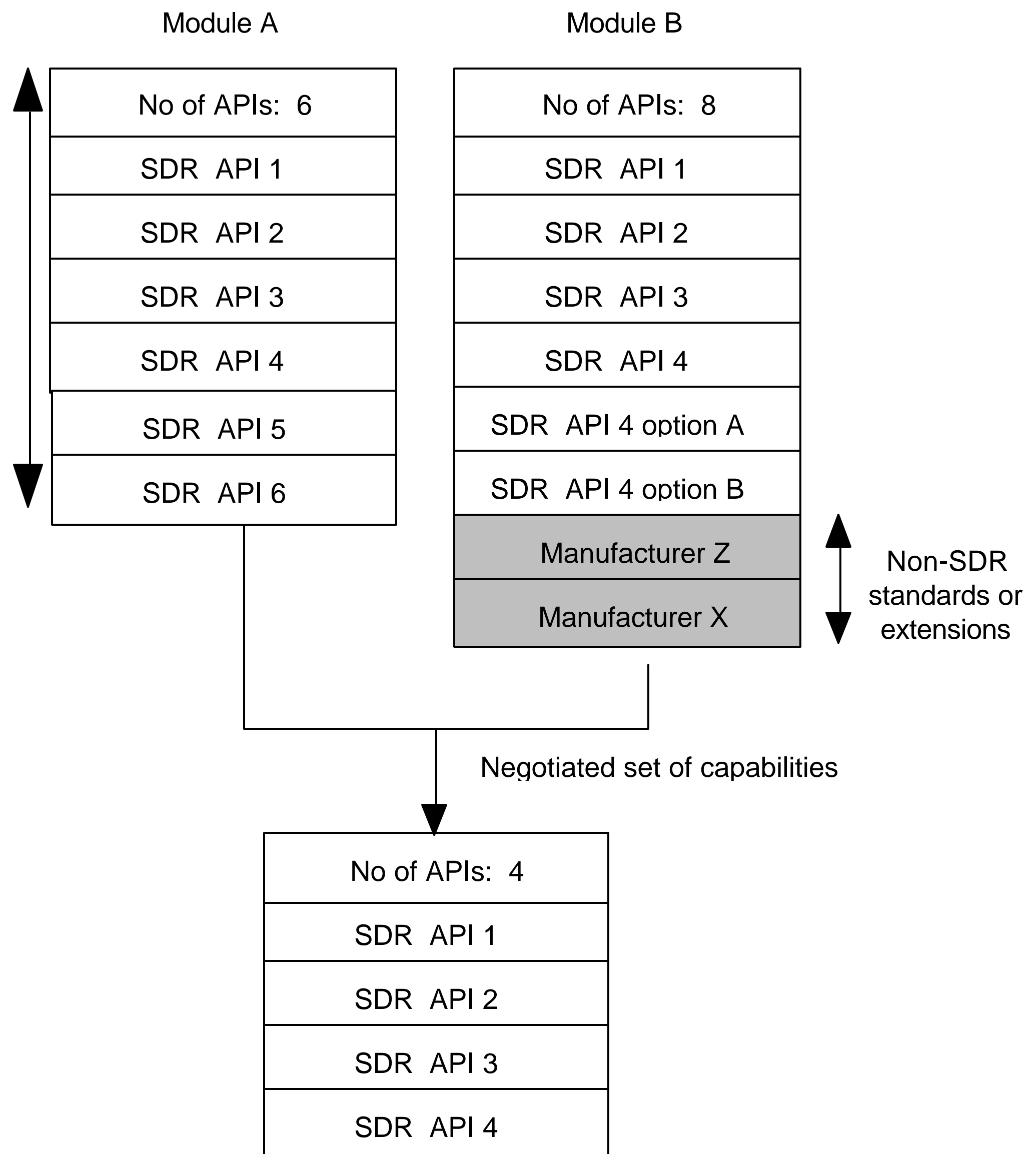


Figure 4.1.4-1 Capability Exchange and Negotiation Using Spec ID Numbers

This provides revision numbers or other system level information to be used to determine at the beginning of any API use, the common standards that can be used. In this way, separate state machines can be used for each standard and/or option without having to make such decisions in mid-protocol.

The diagram shows how this might work: the two modules A and B exchange through their common API a table with the number of API revisions they support. Module A lists six SDRF standards while module B supports six SDRF standards and two non-SDRF standards, Manufacturer Z and Manufacturer X. As part of the negotiation, a common set is derived and returned to both modules. This happens to be the four versions from Module A. This negotiated capability now defines the actual subset of the API that will be used to ensure communication between the two modules.

If module A supported one or more of module B's non-SDRF standards, the negotiation could have resulted in the non-SDRF mode being used without any further use of the standard versions. In this way, both SDRF and other standardized and proprietary modules and interfaces can be supported simultaneously. In practice however, both approaches can be used in combination to determine the common capabilities and the first approach can be used to monitor for any errors or inconsistencies.

4.1.5 Resource Management

Resource management is required to resolve problems caused by the re-use of resources without declaring any potential change in capabilities. In reality, many wireless systems hide the resource management issue because the resource management is handled when the system is designed, integrated, and tested. This static resource management effectively dispenses with any need for its inclusion within an API because the resources are always reserved for the appropriate use.

A good example of this is the GSM handset. It can often support a range of different speech encoders with differing DSP MIPs and memory requirements. When the simplest encoder is used, the additional resources that the most complex encoder needs are not used but are available for use at a moment's notice. If the handset is more sophisticated, the unused resources can be allocated to a different task such as a video decode. However, this reallocation then creates a dilemma: the handset has declared a capability to support a more complex speech encoder but is using the other resources to support another function. Since both parties have declared that they can use Ñ, if the other party decides to change to the complex encoder Ñ the resources required to support the new encoder are not available without stopping the video decoding.

A simple and effective albeit inefficient way of addressing this problem is to adopt a "declare it and reserve it" policy where any declared capability automatically reserves the resources necessary to support it, even though they may not necessarily be used at the time and stand idle.

A more efficient method is to expand the policy to a "declare it, reserve it, negotiate, and re-declare it"policy. In this scheme, unused resources that have been declared and reserved, can be negotiated away to another use providing that the current capabilities are re-exchanged and downgraded to reflect the change in available resources.

Both these techniques are used in the H.320 videoconference [4] standards and the capability exchange mechanism that is described in these documents could form the basis of a key component for the SDRF APIs. They basically work by exchanging tables of information describing spec revisions, options, and

any proprietary extensions that might be available. One party then determines a common subset and this is used to restrict the possible API calls to those that both parties understand.

4.1.5.1 Identifying the Need for Resource Management

Resource management is also needed in other areas apart from the obvious one to determine the exact level of communication through an API. Another common use of the technique is when two modules communicate with a single module in a two-into-one configuration as shown in the diagram. Here the resource management is implicit and not immediately obvious. Modules A and B have independent control flows but both of them interface to the same module C through API C. There is a potential risk of conflict with API C. If this has been designed assuming a single input in, then it cannot cope with two separate flows as shown. It may receive conflicting commands or information. If so, which one does it respond to and when?

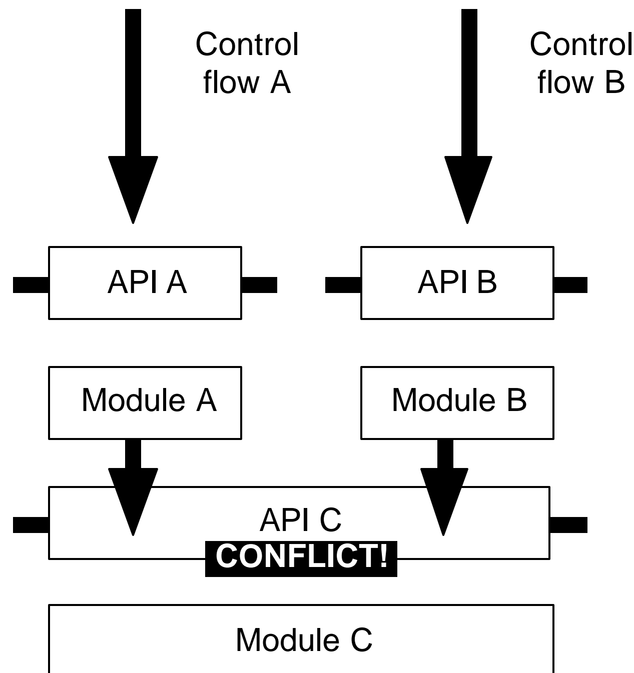


Figure 4.1.5-1 The Two-Into-One Conflict Scenario. Note that the upward flow has not been shown in the diagram

If API C is designed with resource management in mind, this problem can be resolved in several ways:

- Module B could be locked out until module A has completed using module C.

In this case, API C must include some arbitration calls to allow any module to ask for its sole attention. Modules A and B, and their associated APIs A and B, must be designed to cope with delays caused by the arbitration process and the cases when access is denied.

- API C could support multiple access via the use of channels to identify where any data or messaging should be sent.

This now introduces a higher level of complexity into both the API C and module C.

- Simultaneous access could be allowed at this level, but the system design relies on discipline at a higher level to ensure that this never happens.

This is a solution but not a reliable one!

In practice this downward two-into-one architecture is allowable but it should sound alarm bells whenever it appears because of the potential problem with shared resources. There is a reverse condition with the upward data flow case where messages can be sent to multiple locations from a single API. This is less of a problem and in some cases can be quite advantageous. Care must be taken though to ensure that the same information is not sent to the same module via different routes, unless this is actually required and catered for in the design. Message duplication can cause problems that are not often apparent until the system integration phase. In general, the upward two-into-one scenarios can be used with care. If the messages and data are independent then there is no problem. If they are not, care must be exercised to prevent a duplicated message arriving via different routes and being treated as a separate message. This may mean that some form of resource management is needed to remove the conflict by either filtering or re-routing the messages.

4.1.6 Managing Multiple I/O

With many radio systems, particularly with mobile systems, there may be several user interfaces that can have different levels of control over a single radio system. This creates a problem similar to the two-into-one scenario previously mentioned. The diagram in Figure 5.2.6-1 shows such a system with three interfaces providing three separate control paths into the radio. For this system to have any chance to work, the radio system must contain some resource management to reconcile which control path takes command.

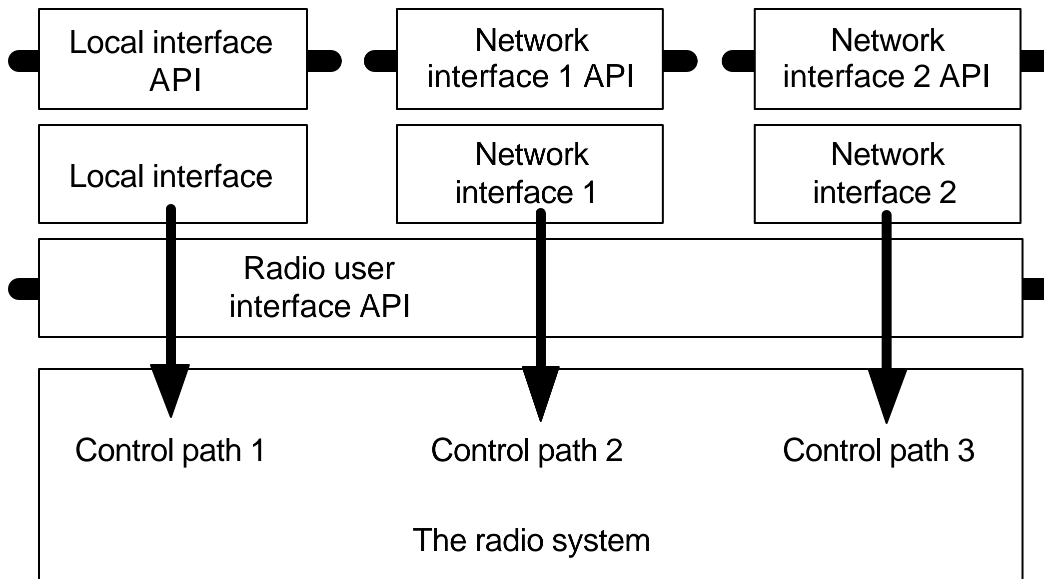


Figure 4.1.6-1 The Multiple Control Path Problem. Note that the upward flow has not been shown in the diagram

This poses a problem for anyone defining an API for the radio system as the user interfaces and control paths are extremely varied. The solution is to bring out the resource management as a separate layer as shown in the following diagram (Figure 4.2.6-2). The three control interfaces are still present and pass commands via the radio user interface almost as before. Almost, because there are also resource management commands associated with the interface as well. These commands use capability exchanges and other information to determine who has actual control of the radio. This could be fixed and hard coded or dynamic and vary with circumstances. By using a common command interface and a capability exchange to restrict each interface to a particular subset, different categories of access can be quickly defined and changed if needed. For example, the network interfaces 1 and 2 could be restricted to receiving and transmitting information while the local interface is the only one allowed to define the communication wavelength, modulation, and encoding.

By being generic in how these interfaces are defined, and allowing extensions to cope with non-generic requirements, it is possible to create a user interface and resource management API that supports almost any variation possible while still retaining and reusing the radio system.

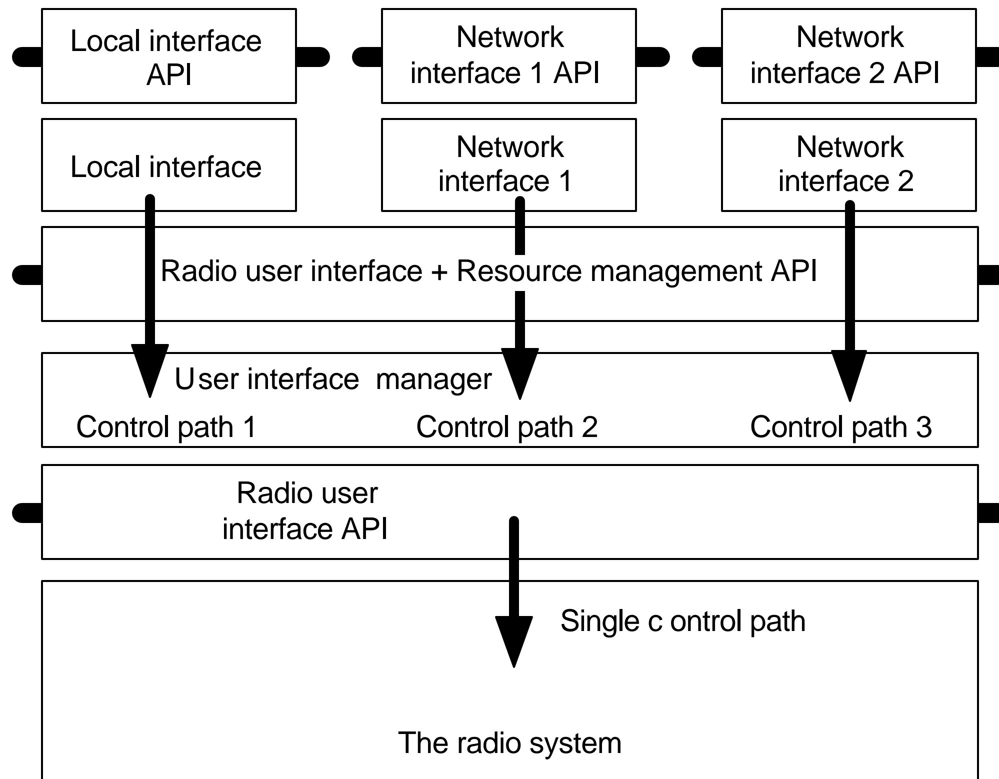


Figure 4.1.6-2 Using a User Interface Manager. Note that the upward flow has not been shown in the diagram

The user interface manager takes the different radio user interface commands from the various external sources and based on capability tables, including priority and a hierarchy of command information, decides how the commands should be combined to create a single user interface control and information flow for the radio. The radio is blissfully unaware that several sources are using the radio simultaneously.

One further aspect is also important. At the radio interface and resource management API level, the interfaces are defined not by their location or communication path but by their capability. This gives far more flexibility because the capabilities can be dynamically changed and overridden. A remote resource could take over local access capabilities and then disable the local control. This could be used in cases where there is unauthorized local access or if the local access is malfunctioning. For sensitive installations, this can be an advantageous feature.
excluding any particular part. The rules, guidelines, and definitions provided in this document should facilitate discussion and development of APIs to support SDRF API's.

4.1.7 API Design Process

4.1.7.1 Introduction

Any design process is cyclical in nature and involves testing a design to identify and remove any errors. The SDRF APIs are no different and are subject to the same constant refinement. This process is important when the SDRF APIs are incorporated into products, especially when they are combined with existing APIs and extensions.

This document describes the process and its elements as used in the development of the APIs and also provides an example process for use in product developments.

4.1.7.2 Overview

The SDRF API Definition process describes how new APIs are proposed, defined, and accepted by the SDR Forum. Figure 4.1.7.2-1. is an overview of the procedure.

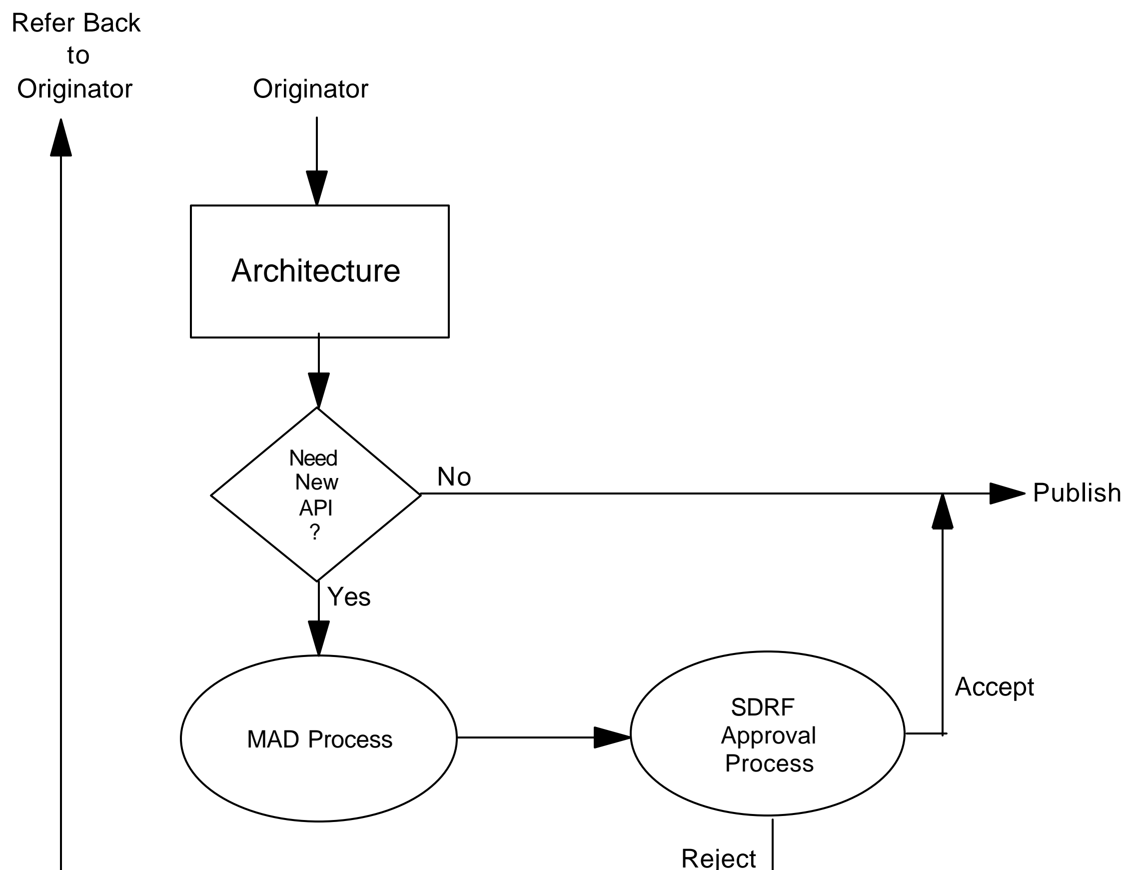


Figure 4.1.7.2-1. Context for the API Development Process

The originator is the person or organization that proposes to add an API to the SDR Forum defined APIs. Two hurdles must be crossed: technical definition and evaluation of the API, and then acceptance for publication by the Forum.

Architecture of an application intended to operate with the SDRF model and claim to be SDRF Conformant is the responsibility of the originator. In the course of defining that architecture, the originator may find that a new API is required. (See Section 4.2, API Relationships.)

The API definition process is the mechanism to define such an API, and the SDRF approval process a means of publishing it, and adding it to the body of SDRF conformant APIs.

4.1.7.3 The SDRF API Definition Process

The API Development process is depicted in Figure 4.1.7.3-1. The need for a new API arises as the result of an application.

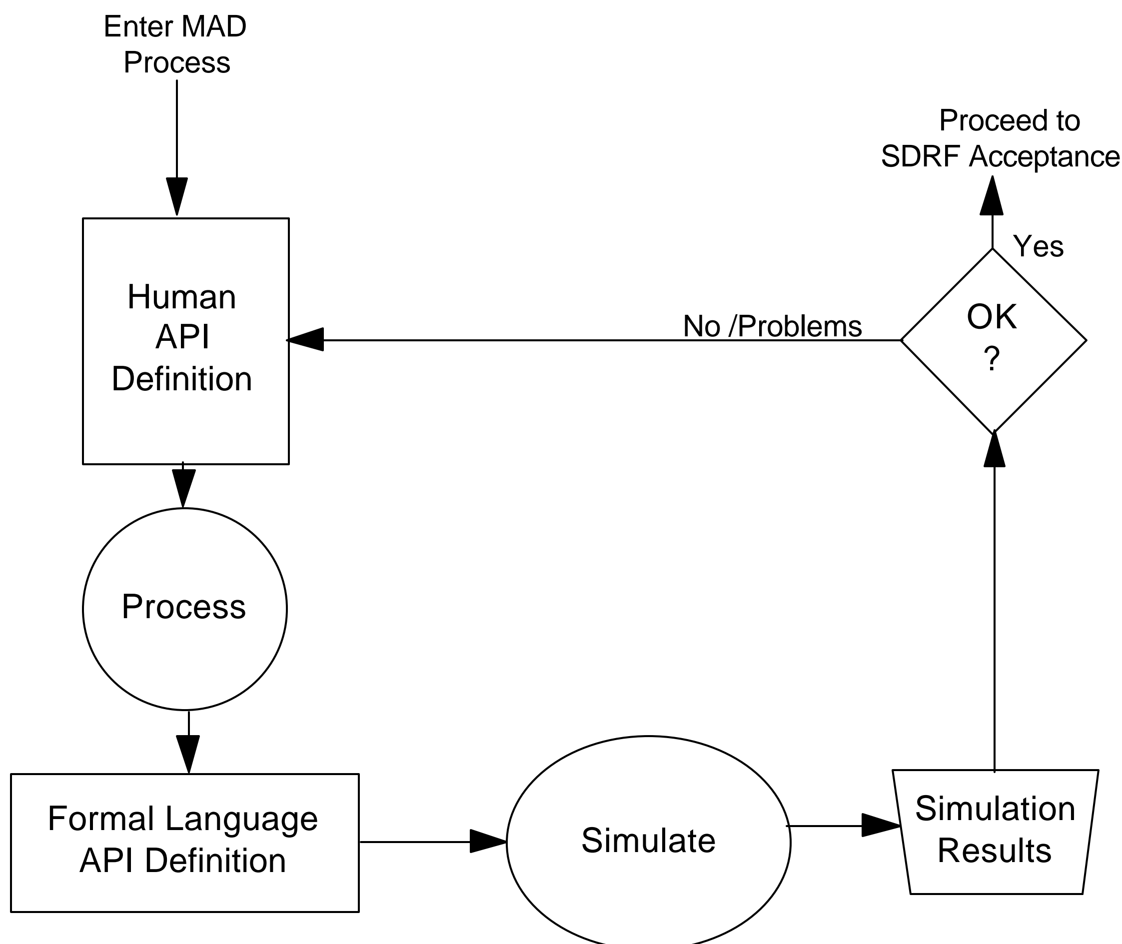


Figure 4.1.7.3-1

4.1.7.3.1 Product architecture

The architecture is based on application requirements and will contain sufficient interface and behavioral information to allow the all API definitions — including any extensions to the SDRF APIs or proprietary

4.1.7.3.2 Human API definition

This is the textual description of the APIs needed to support the product architecture. It is a manual process but can be facilitated in several ways. The SDRF documentation through its API design guide and examples provides detailed guidance on how to define and create APIs.

4.1.7.3.3 Process

This is a manual process but it should only focus on new APIs or extensions. Formal models of existing SDRF APIs can be accessed and re-used in the same way that software libraries are utilized. This reduces the work and focuses it on unsupported APIs and extensions.

4.1.7.3.4 Formal language definition

The output of the previous process will be a formal definition of the interfaces and sufficient behavioral information to define the APIs unambiguously. The current thinking within SDRF is to use SDL and IDL but this does not necessarily preclude the use of other languages. Their use may increase the work required through the lack of re-useable definitions.

4.1.7.3.5 Simulation

The simulation will use tools that can accept the formal definitions. These are used to exercise the elements to confirm consistency.

4.1.7.3.6 Simulation results

The results should identify problems such as signal mismatches, orphan and widow signals and behavioral inconstancies. This information can be also be used to create test patterns sequences for use later in the product integration and test phase(s).

4.1.7.3.7 Results comparison

The simulation results are then compared to the original product architecture and checked to see if there are any problems. This is likely to involve a combination of a design review approach coupled with the use of automated tools to highlight problems.

If the results are acceptable, then the API can proceed to the SDRF acceptance process.

If the errors are caused by implementation issues e.g. a missing signal or message, then the human API definition is then refined to remove the error and the API DEVELOPMENT process repeated.

If the errors are fundamental to the design, then either the requirements or product architecture must be updated before repeating the process. As these are the responsibility of the Originator, the issue is out of the scope of the API DEVELOPMENT Process, and is referred back for resolution.

4.1.7.4. The SDRF API Approval Process

The SDRF API Approval Process has not yet been defined.

4.1.8 References

1. Programmable Modular Communication System Guidance Document, Revision 2, 31 July 97. (http://www.dtic.mil/c3i/pmcs/pmcspage.htm)

2. Microsoft Telephony API specification. Microsoft Windows software development kit.

3. H.320 P.64 narrow band visual telephone systems and terminal equipment specification. ETSI.

4.2 Relationships to non-compliant Processes

A primary objective of the SDR Forum is to establish a set of APIs that will provide an open interface into SDRF-compliant systems. This section explores how legacy APIs and the SDRF APIs work together to define the specific APIs to be included in a new development project.

4.2.1 API Relationship Diagram

It is important to realize that the SDRF APIs for a software definable or configurable radio are not being developed in isolation. Therefore it is also important to include facilities to support both internal and external existing APIs and to provide support for additional facilities necessary to meet the required product specifications. As a result, it is conceivable that there will be many SDRF compliant radios that will have varying proportions of APIs depending on the end requirement. This can range from a radio that is completely SDRF based to one that may have a high level SDRF interface but relies on existing and/or proprietary interfaces. Figure 5.4.1-1 displays these relationships as a Venn diagram.

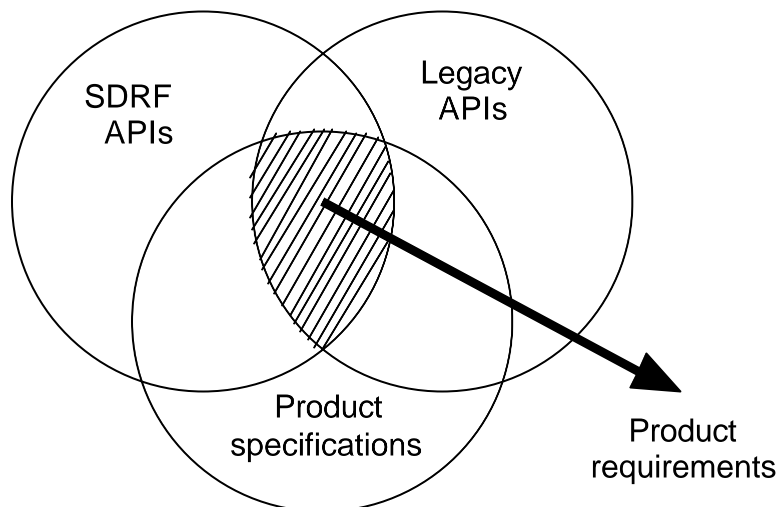


Figure 4.2.1-1 The API Relationship Diagram

4.2.1.1 SDRF APIs

These are the APIs that are defined by SDRF and are used to create a framework. In a complete SDRF system, all the published APIs would be used. In a reduced system, some of the APIs may not be used or required. In a hybrid system, some of the APIs may be replaced by legacy or proprietary APIs. This replacement will require some form of conversion between the two environments. This is performed by the wrapper layer, which is located between the respective SDRF and legacy APIs.

4.2.1.2 Legacy APIs

In an ideal world, there is a strong argument that there is no need to support existing or legacy APIs providing alternatives are available. In a practical world, this may not be achievable and the desire to use or re-use existing sub-systems may have great importance in reducing the design costs. However, this does rely on there being some form of conversion to allow the legacy APIs to communicate with the SDRF environment. It is this conversion that the wrapper performs. Providing the wrapper design and implementation is not prohibitive. It allows SDRF based systems to be developed using existing components.

Supporting legacy APIs also has other benefits in that it can provide an additional source of design knowledge based on practical experience which can be used to test the SDRF APIs and identify omissions and discrepancies.

For all these reasons, legacy API support is an essential and important component of the SDRF environment.

4.2.1.3 Product Specifications

The third input into the relationship is product specifications. These will define requirements that may be beyond the SDRF APIs on first inspection but may still require support from SDRF APIs. The specifications may require software download, Therefore this immediately requires the inclusion of the SDRF software download API . It may also require a sophisticated GUI front end for the end user. This is not directly specified in the SDRF APIs but does rely on SDRF messages and information to allow the connection status and so on to be displayed. In this example, SDRF APIs will be needed to provide the foundations but the designer has the freedom to build whatever support is needed on top of them.

As a result, the proportions of the three rings may and will vary depending on the final product requirements. Where the three rings overlap defines the subset of the requirements for the product. These final proportions when defined effectively create product requirements that can be inputted into the API DEVELOPMENT process to create and test the API design. The technical report provides sufficient detail to allow an initial set of requirements to be inputted into the process.

4.2.2 The wrapper

The idea of the wrapper is to provide a conversion process to allow bi-directional communication between two different APIs. It is a piece of software or hardware that allows modules on either side of it to communicate through their own APIs without knowing that any conversion or modification has taken place.

The wrapper uses the two APIs that it is bridging between as its boundary definitions and then processes the information internally from each API to allow them to work together.

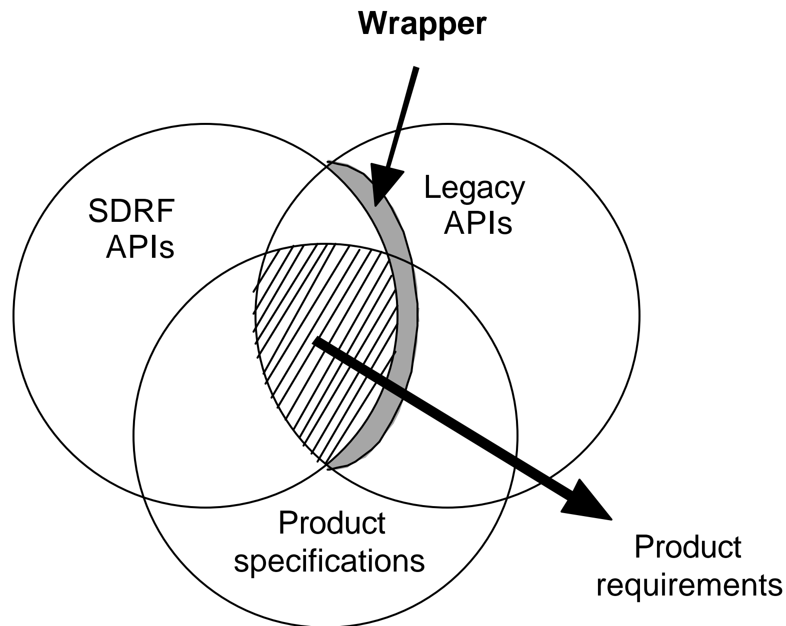


Figure 4.2.2-1 the wrapper between legacy and SDRF APIs

4.2.3 Conversion techniques

The wrapper shown in the diagram uses three basic techniques to provide the communication between the two APIs. These techniques are translate, simulate and integrate.

4.2.3.1 Translate

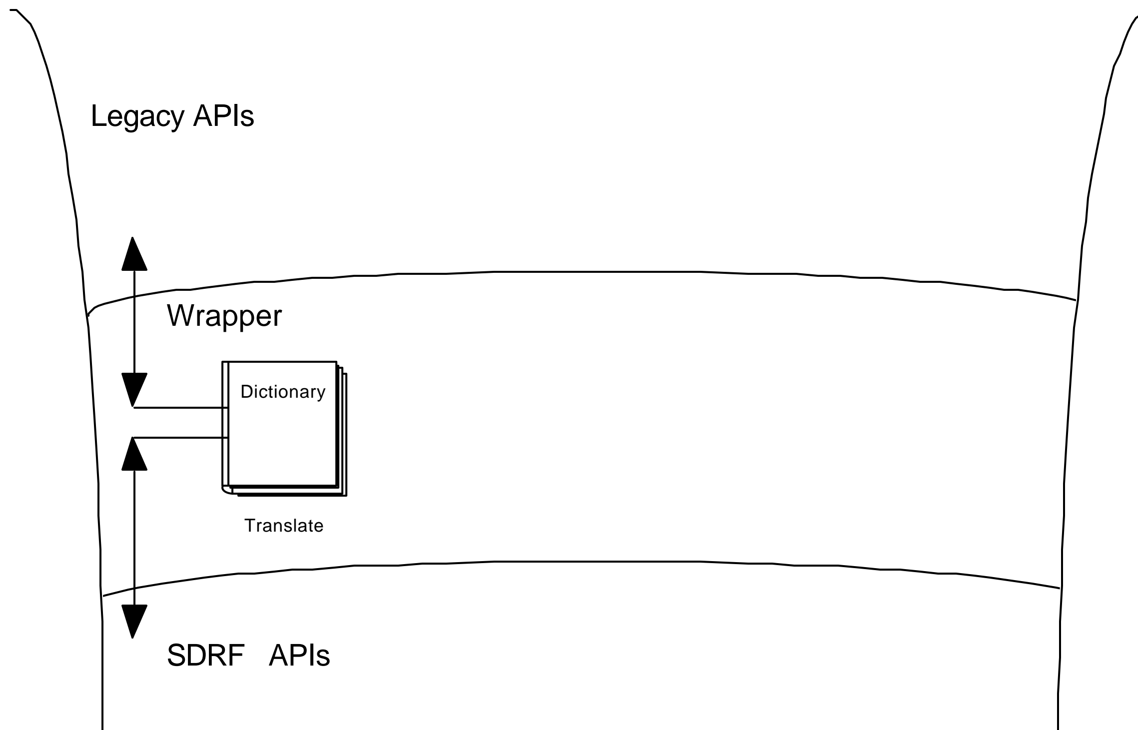


Figure 4.2.3.1-1 Translate

This technique exploits content and/or syntax commonality between the two APIs. It essentially takes each message, looks it up in a dictionary and translates it so that the other API can understand it. The translation process may include some state machine behavior, in which case the technique really falls into the third category – integrate. In general, it should be a one-to-one process and to achieve this requires some commonality between the two APIs.

This in itself is not a problem providing the two APIs are performing similar functions and thus require similar information. In this case, the content is compatible and therefore only the syntax needs to be changed. Examples of this include parameter re-ordering, bit and byte swapping for little and big-endian data organizations.

4.2.3.2 Simulate

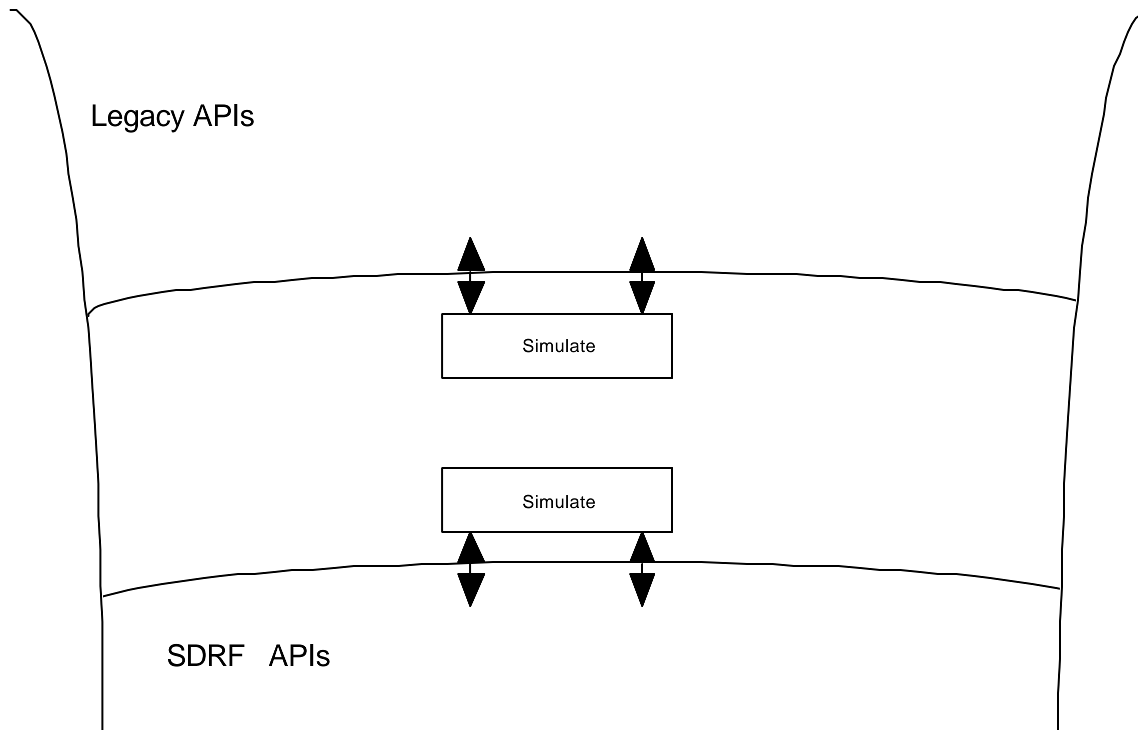


Figure 4.2.3.2-1 Simulate

This technique will simulate messages or behavior to support one API without involving the other API. This can be used where there are messages or behavior that is unique to one API.

4.2.3.3 Integrate

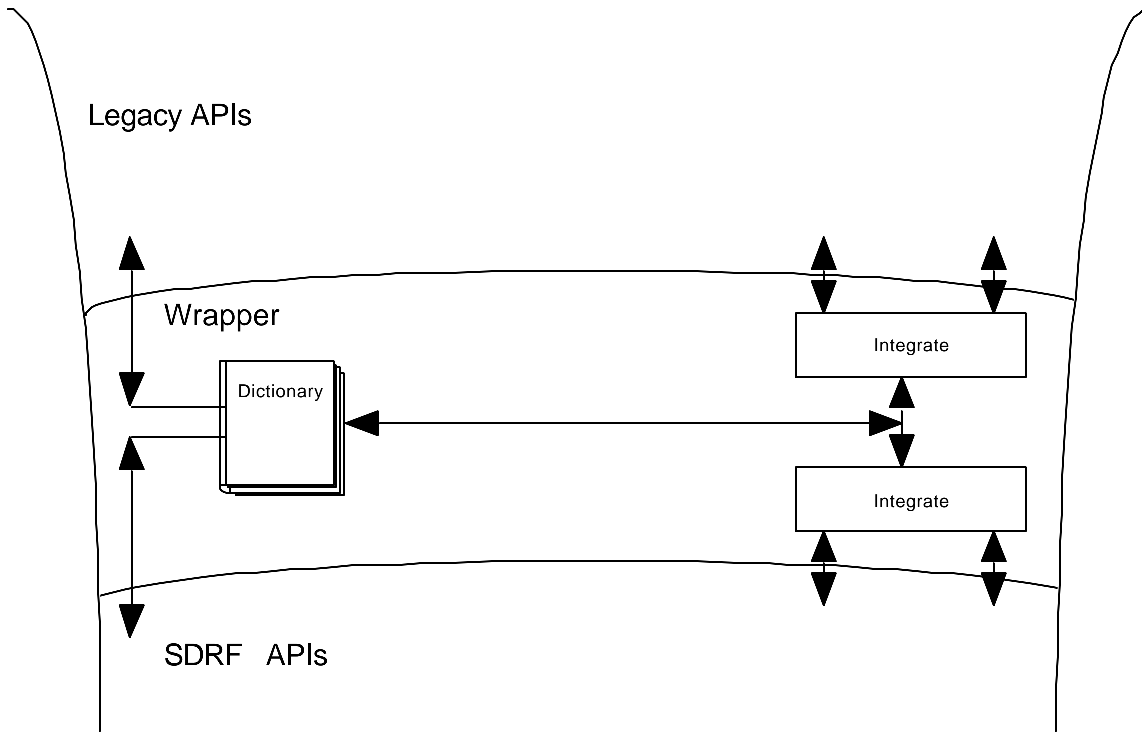


Figure **4.2.3.3-1 Integrate**

This technique is a hybrid between the previous two techniques and integrates both simulation and translation to provide a communication path between the two APIs. It combines translation and simulation to allow very dissimilar APIs to be bridged and thereby to communicate.

4.2.4 Trade-offs

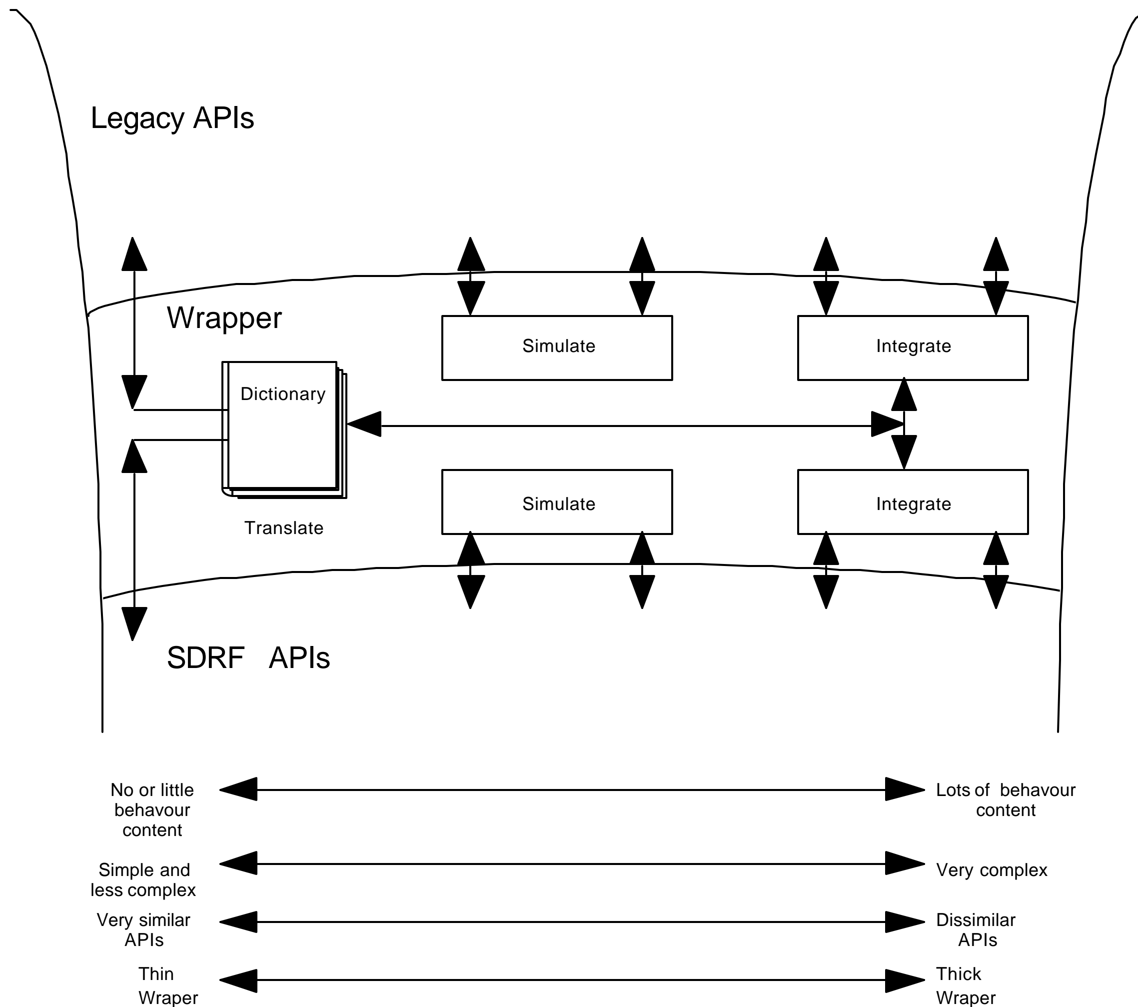


Figure 4.2.4-1 Wrapper Trade-offs

The wrapper diagram also shows some of the trade-offs associated with the various techniques. The translate technique is the simplest to implement because it has little or no behavioral content. It does require that the two APIs are very similar in content and philosophy so that only translation techniques are needed. Typically thin wrappers will predominately use translation techniques, and are simple and easy to create.

At the other extreme, the integrate technique will have a high behavioral content and thus will be far more complex to implement and test. However it will work with very dissimilar APIs. These wrappers are often referred to as thick because of this level of complexity that is needed.

The simulation technique typically sits in the middle in terms of behavioral and complexity.

4.2.5 Conclusions

The SDRF APIs will form a standard that can be used in the design of new systems. They do, in fact, overlap significantly with other API sets serving similar applications. The API definitions for a new system can utilize both legacy and SDRF API sets. In some cases it may be desirable to use wrapper techniques when integrating legacy APIs.

4.3 Distributed Processing Environment

Left blank in this edition.

4.4. Message description for APIs

4.4.1 Introduction

SDRF modules use messages as the primary means of exercising control and exchanging information. Messages are of two types: control messages and information messages. Messages flow in accordance with the guidance of the API design guide in section 4.1. This section provides an example of how control messages are used within the SDRF environment and gives specific examples for currently identified control messages.

Through the set of examples, this section also identifies some of the key problems and proposes some solutions to these issues. These examples reflect a functional perspective but alternatives will be considered. It is anticipated that where ever possible, the final API specifications will utilize current technologies and standards.

4.4.2 Background

For each API in a system, there is a requirement for a set of messages that allow the module associated with the API(s) to be configured and set up correctly. This set of messages are reasonably generic in that all APIs will require them and will only differ in terms of API specific information and parameters.

By using these messages, modules can be initialized, started and stopped, enabled and disabled, capabilities exchanged, configurations adapted and so on. In addition, support for the replacement, augmentation or switching of facilities is also included to provide the foundation support for software download and hot swappable hardware.

The messages can be used in one of two ways:

- To provide the basic control and set-up as previously described.
- To provide support for a switcher module that can intelligently enable/disable modules to radically change the functionality of the overall system. This can be used in multi-band and/or multi-mode systems to select a particular service or waveband and so on.

  The intelligence is contained within the switcher function but it can use the same APIs without having the need to change them.

4.4.3.Messages

This section describes the control messages that are needed to control and configure a module via its API. It is envisaged that these messages will appear in all API definitions along with additional messages

to control the functionality of the API. These messages can be used by a module to implement service switching functions within a multi-band and/or multi-mode phone.

4.4.3.1 Message acknowledgment

Each message is acknowledged by a status word(s) which confirm that the destination has correctly received the message, acted on its contents and replying with the current status. This provides detailed information to not only allow the traditional ack/nack protocol but also for debugging and system integration where a deeper insight is needed. Status words sent in direct reply to a message are referred to as solicited status information (SSI).

In addition, status information can be returned without any initiating message if the module status has changed. For example, if an error is detected this can be reported by sending a suitable status word. These messages are unsolicited and it should not be assumed that they will automatically result in any action from the higher modules and/or APIs. This information is known as unsolicited status information (USI). Indeed, it is recommended that some form of filtering is supported to reduce or expand the reporting back that can be accomplished. Again, this is primarily to help debugging and system integration.

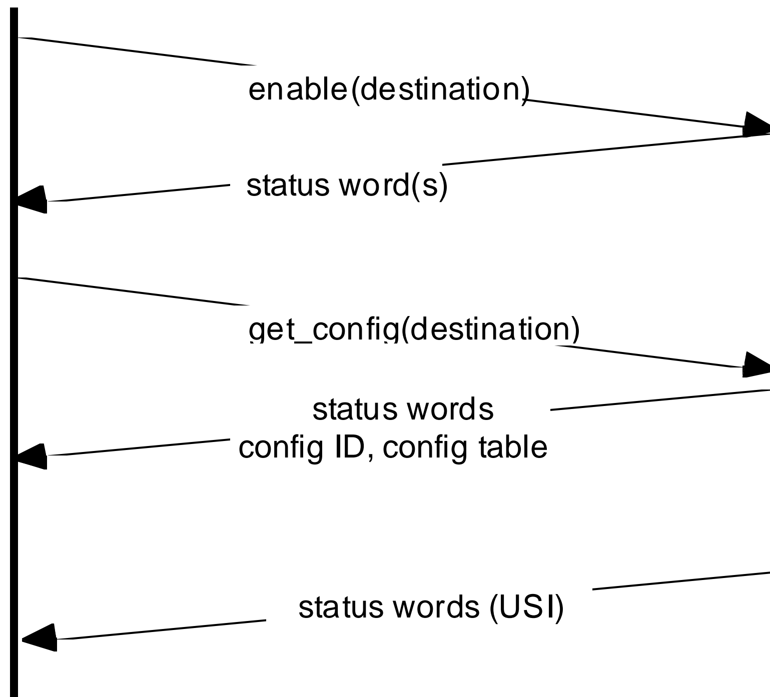


Figure 4.4.3-1 State ladder diagrams showing the relationship between status words returned as a result of receiving a message

These examples assume that all messages come from a single source. Where this is not the case e.g. a multiprocessor system, the message must include data to identify the message source as well as the destination, so that the message flow within the system can be tracked.

4.4.4 Message Definitions

This section describes the message format and the message definitions for the control messages.

4.4.4.1 Message format

The message definitions have three components that define the message name, associated parameters and the solicited status information. The nomenclature used is shown in the diagram. The message name is the first component followed by a list of parameters bounded by a set of brackets. The returned SSI is denoted by an arrow (←).

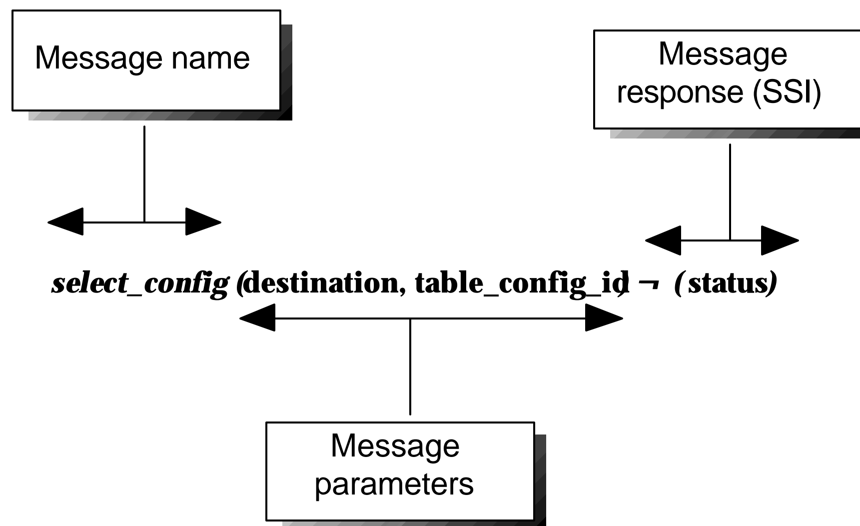


Figure 4.4.4-1 Message definition structure

The SSI will contain an updated version of the status word(s) to reflect the success or otherwise of the originating message. In addition, there may be optional parameters that are returned.
It should be remembered that this document specifies the tier 1 messages and not how the are transported or any other physical parameters or attributes.

These formats could also be used for the tier 2 and tier 3 APIs but this is not obligatory and is optional.

4.4.4.2 Parameter format

Some of the messages have parameters associated with them. The format of how these messages are passed e.g. by using a pointer, mailbox, or as a data structure is not defined in the Tier 1 API but is covered within the associated Tier 2 API for transportation and communication.

<table>
<tr><th colspan="3">4.4.4.3 enable (destination) ¬ (status)</th></tr>
<tr><td>destination
status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.</td></tr>
<tr><td>4.4.4.3.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message will enable the module and does not actually start its functionality. To use the VCR analogy, Enable is the equivalent of powering up the VCR. It does not mean that the VCR should automatically start playing a tape or recording a transmission.

It can be used to trigger several actions within the module that uses the API, such as self test, internal resource allocation and initialization and so on.</td></tr>
<tr><td>4.4.4.3.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently disabled.</td></tr>
<tr><td>4.4.4.3.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to activate and be ready to receive and act on further messages. This may include reserving resources in preparation for this.</td></tr>
<tr><td>4.4.4.3.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should be acted upon.
• The status SSI should indicate that the module is enabled.
• The module should be able to receive further messages but should not process incoming data.</td></tr>
<tr><td></td><td>If the module is currently enabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should continue executing</td></tr>
</table>

4.4.4.4 disable (*destination*) ¬ (*status*)		
Destination *Status*	Destination ID of the module to be enabled through the API. This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.	
4.4.4.4.1	**Description**	
	This message will disable the module and is the equivalent of a "power off" command. It signals that the intention to use the API has gone and that its functionality is not required. To use VCR analogy, Enable is the equivalent of powering off the VCR. It can be used to trigger several actions within the module that uses the API, such as internal resource de-allocation.	
4.4.4.4.2	**Dependencies**	
	This command should only be sent if the module is currently enabled and stopped.	
4.4.4.4.3	**Action**	
	The module should treat this signal as a request to deactivate but still be ready to receive and act on an **enable** messages. This may include releasing resources.	
4.4.4.4.4	**Results**	
	If the module is currently enabled and the message sent:	• The message shuld be acted upon. • The status SSI should indicate that the module is disabled. • The module should be able to receive selected further messages.
	If the module is currently enabled but not stopped and the message sent:	• The message should not be acted upon. • It should be treated as a programming error. • The status maessage should indicate that this is an error. • The module should continue executing.
	If the module is currently disabled and the message sent:	• The message should not be acted upon. • It should be treated as a programming error. • The status message should indicate that this is an error. • The module should stay disabled.

4.4.4.5 reset (*destination*) ¬ (*status*)		
destination *status*	Destination ID of the module to be enabled through the API. This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location. The configuration table ID will be returned in the status word unchanged i.e. it will have the same value as before the message was sent.	
4.4.4.5.1	**Description**	
	This message will reset the module using the current configuration. It is the equivalent of a "warm boot" in a PC where the software will reset itself and start again. It can be used to trigger several actions within the module that uses the API, such as self test, internal resource allocation and initialization and so on.	
4.4.4.5.2	**Dependencies**	
	This command should only be sent if the module is currently enabled and stopped.	
4.4.4.5.3	**Action**	
	The module should treat this signal as a request to reset itself using the current configuration. This may include initializing and/or flushing resources.	
4.4.4.5.4	**Results**	
	If the module is currently enabled and the message sent:	• The message should be acted upon. • The staus SSI should indicate that the module is reset. • The module shuld be able to receive selected further messages.
	If the module is currently enabled but not stopped and the message sent:	• The message should not be acted upon. • It should be treated as a programming error. • The status message should indicate that this is an error.
	If the module is currently disabled and the message sent:	• The message should not be acted upon. • It should be treated as a programming error. • The status message should indicate the this is an error. • The module should stay disabled.

<table>
<tr><th colspan="3">4.4.4.6 reset_to_default(destination) ¬ (status)</th></tr>
<tr><td>destination
status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status work will be the API module ID to confirm that it was sent to the correct location.

The configuration table ID will be set in the status owrd to 0, thereby indicating that the module has reset to the default configuration.</td></tr>
<tr><td>4.4.4.6.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message will reset the module using the default configuration. It is the equivalent of a “cold boot” in a PC where the software will reset itself and start again.

It can be used to trigger several actions within the module that uses the API, such as self test, internal resource allocation and initialization and so on.</td></tr>
<tr><td>4.4.4.6.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently enabled and stopped.</td></tr>
<tr><td>4.4.4.6.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to reset itself using the default configuration. This may include initializing and/or flushing resources.</td></tr>
<tr><td>4.4.4.6.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently enabled and the message sent:</td><td>• The message should be acted upon.
• The status SSI shuld indicate that the module is reset.
• The module should be able to receive selected further messages.</td></tr>
<tr><td></td><td>If the module is currently enabled but not stopped and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message shuld indicate that this is an error.
• The module should continue executing.</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should stay disabled.</td></tr>
</table>

<table>
<tr><th colspan="3">4.4.4.7 start (destination) ¬ (status)</th></tr>
<tr><td>destination
status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.

The start/stop status bit will be returned set to 1.</td></tr>
<tr><td>4.4.4.7.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message will start the module. Data will be accepted and processed in the module only when it has been started. A synchronous start message may be required and has not yet been defined.</td></tr>
<tr><td>4.4.4.7.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently enabled and stopped.</td></tr>
<tr><td>4.4.4.7.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to start processing data.</td></tr>
<tr><td>4.4.4.7.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently enabled but in a stopped state and the message sent:</td><td>• The message should be acted upon.
• The status Ssi should indicate that the module has started.
• The module should be able to receive selected further messages.</td></tr>
<tr><td></td><td>If the module is currently enabled but not stopped and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should continue executing.</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should stay disabled.</td></tr>
</table>

<table>
<tr><th colspan="3">4.4.4.8 stop (destination) ¬ (status)</th></tr>
<tr><td>destination
status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.

The start/stop status bit will be returned set to 0.</td></tr>
<tr><td>4.4.4.8.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message will stop the module via the API from processing any data but it will retain any allocated resources that have been allocated to it. A synchronous stop message may be required and has not yet been defined.</td></tr>
<tr><td>4.4.4.8.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently enabled and started.</td></tr>
<tr><td>4.4.4.8.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to stop processing data.</td></tr>
<tr><td>4.4.4.8.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently enabled but in a started state and the message sent:</td><td>• The message should be acted upon.
• The status SSI should indicate that the module has stopped.
• The module should be able to receive selected further messages.</td></tr>
<tr><td></td><td>If the module is currently enabled but stopped and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should continue executing.</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should stay disabled.</td></tr>
</table>

<table>
<tr><th colspan="3">4.4.4.9 set_config (destination, table_config_id, parametersÖ) ¬ (status)</th></tr>
<tr><td>destination
table_config_id

Parameters

status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This identifies the configuration table that is used to receive the new parameters. The API supports up to n tables with table ID 0 being the default. The total number of tables is defined in the capability exchange information. The default table 0 can either be fixed and not capable of modification or capable of modification. Again, these characteristics are returned in the capability table.
These are the parameters that are sent and stored in the configuration table. The actual format will depend on the API definition for each module and will reflect the functionality controlled by the API.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.</td></tr>
<tr><td>4.4.4.9.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message allows the module configuration via the API to be changed as required. The changes do not imply or demand any activation and therefore the process of changing the functionality should be seen as a two stage process:
• Set-up the new configuration using set_config and one of the available configuration tables.
• Select this table using the select_config message to activate it.</td></tr>
<tr><td>4.4.4.9.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently enabled.</td></tr>
<tr><td>4.4.4.9.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to update a configuration table as selected by table_config_id.</td></tr>
<tr><td>4.4.4.9.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently enabled and the message sent:</td><td>• The message should be acted upon.
• The status SSI should indicate that the message was successful.
• The module should be able to receive selected further messages.</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should stay disabled.</td></tr>
</table>

<table>
<tr><th colspan="3">4.4.4.10 select_config (destination, table_config_id) ¬ (status)</th></tr>
<tr><td>destination
table_config_id

status</td><td colspan="2">Destination ID of the module to be enabled through the API.
This identifies the configuration table to be selected to configure the module. The API supports up to n tables with table ID 0 being the default. The total number of tables is defined in the capability exchange information. The default table 0 can either be fixed and not capable of modification or capable of modification. Again, these characteristics are returned in the capability table.
This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location. The configuration table ID bit will be returned set to the value specified by table_config_id.</td></tr>
<tr><td>4.4.4.10.1</td><td colspan="2">Description</td></tr>
<tr><td></td><td colspan="2">This message instructs the module to use the configuration information stored in the selected configuration table. It is used to signal that this information should now be used. The act of simply writing the information into the table does not imply or be interpreted as requesting an immediate changeover. This select_config message is used to do this as part of a two stage process:
• Set-up the new configuration using set_config and one of the available configuration tables.
• Select this table using the select_config message to activate it.
Note: If the configuration table that is being modified by the set_config command is the current one being used by the module, then any changes made to the table should not change the module's configuration until the select_config message is sent.</td></tr>
<tr><td>4.4.4.10.2</td><td colspan="2">Dependencies</td></tr>
<tr><td></td><td colspan="2">This command should only be sent if the module is currently enabled.</td></tr>
<tr><td>4.4.4.10.3</td><td colspan="2">Action</td></tr>
<tr><td></td><td colspan="2">The module should treat this signal as a request to select a configuration table as selected by table_config_id.</td></tr>
<tr><td>4.4.4.10.4</td><td colspan="2">Results</td></tr>
<tr><td></td><td>If the module is currently enabled and the message sent:</td><td>• The message should be acted upon.
• The status SSI should indicate that the message was successful.
• The module should be able to receive selected further messages.</td></tr>
<tr><td></td><td>If the module is currently disabled and the message sent:</td><td>• The message should not be acted upon.
• It should be treated as a programming error.
• The status message should indicate that this is an error.
• The module should stay disabled.</td></tr>
</table>

4.4.4.11 get_config (destination, table_config_id)¬ (status, parameters)

destination	Destination ID of the module to be enabled through the API.	
table_config_id	This identifies the configuration table that is used to supply the parameters. The API supports up to n tables with table ID 0 being the default. The total number of tables is defined in the capability exchange information. Again, these characteristics are returned in the capability table. The default table 0 can either be fixed and not capable of modification or capable of modification. The currently used table ID is available from the status information.	
Status	This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.	
parameters	These are the parameters that are sent and stored in the configuration table. The actual format will depend on the API definition for each module and will reflect the functionality controlled by the API.	
4.4.4.11.1	**Description**	
	This message gets the configuration information from the selected configuration table. If the ID is the same as that of the currently in use configuration table, the information will define the current configuration of the module. If not, it will represent an alternative configuration that is not active. The current configuration table ID is returned in the status information.	
4.4.4.11.2	**Dependencies**	
	This command should only be sent if the module is currently enabled.	
4.4.4.11.3	**Action**	
	The module should treat this signal as a request to select a configuration table as selected by *table_config_id*, and return its contents .	
4.4.4.11.4	**Results**	
	If the module is currently enabled and the message sent:	• The message should be acted upon. • The status SSI should indicate that the message was successful. • The module should be able to receive selected further messages.
	If the module is currently disabled and the message sent:	• The message should be not acted upon. • It should be treated as a programming error. • The status message should indicate that this is an error. • The module should stay disabled.

4.4.4.12 capability_exchange(destination, max_capability) ¬ (status, parameters)		
destination *max_capability* *status* *parameters*	Destination ID of the module to be enabled through the API. ID of the capability table to be interrogated. Initially only one capability table will be specified and supported and this is the maximum capability of the module which defines the facilities and functional parameters that it supports. This information is used in the capability exchange to determine a common set of messages between two modules. This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location. These are the parameters that make up the capability exchange information. The actual format will depend on the API definition for each module and will reflect the functionality controlled by the API. This should include manufacturer's codes and revision numbers as well as the SDRF API specification revision numbers to allow the API and module to be clearly identified. This can also be expanded to include other data that can be used for certification, type approval and authentication procedures.	
4.4.4.12.1	**Description**	
	This message instructs the module to return its capability information that defines exactly what it can support. This information is related to the configuration information but is not exactly the same, although there are close similarities. The returned parameters define all the capabilities that can be supported by the module through the API and are therefore available to other modules. The information includes options and configuration information etc. that is supported. This table is fixed and cannot be changed unless the module itself is replaced.	
4.4.4.12.2	**Dependencies**	
	This command should only be sent if the module is currently enabled.	
4.4.4.12.3	**Action**	
	The module should process this signal as a request to return the contents of the capability table .	
4.4.4.12.4	**Results**	
	If the module is currently enabled and the message sent:	• The message should be acted upon. • The status SSI should indicate that the message was successful. • The module should be able to receive selected further messages.
	If the module is	• The message should not be acted upon.

	currently disabled and the message sent:	• It should be treated as a programming error. • The status message should indicate that this is an error. • The module should stay disabled.

4.4.4.13 get_status(*destination*) ¬ (*status*)	
destination *status*	Destination ID of the module to be enabled through the API. This is returned to indicate the success or otherwise of the message and the current status of the module. Included in the status word will be the API module ID to confirm that it was sent to the correct location.
The status word will contain the following information:	
Message ID	This allows the status information to be associated with a message or identified as unsolicited.
Destination ID	This allows the source of the status information to be identified Again this is provided to track status information in relation to messages.
Sequence number	This is provided to allow the correct sequencing of the status information to be obtained in the event that status information arrives out-of-order.
Current configuration ID	This identifies which configuration table is used. If set to 0, this indicates that the default is being used.
Module state	This includes flags for start/stop, enable/disable and so on.
Status code	This encompasses both error and good codes and defines either a response to a message or some change in the module that should be notified to the rest of the system.
Status code filter	This defines the state of the filter for USI messages. This is used to control or limit the number or type of USI messages that are passed across the API.
4.4.4.13.1	**Description**
	This message instructs the module to return its status information that defines exactly its current state. This information is provided in addition to the configuration information but is concerned with short term status.
4.4.4.13.2	**Dependencies**
	This command can be sent at any time. This is necessary to identify the status of an unknown module and to establish exactly how to bring the module up to normal operation. The information is a snapshot of the system and it should be remembered that it is constantly changing. As a result, the actual physical status may be different from that indicated by the received status information. Any differences, should result is further updated status information to be sent via unsolicited status information.
4.4.4.13.3	**Action**
	The module should treat this signal as a request to return the status information .
4.4.4.13.4	**Results**

	• The message should be always be acted upon. • The status SSI should indicate that the message was successful. • The module should be able to receive selected further messages.

4.4.4.14 place_module (destination, info) ¬ (status, parameters)		
destination *Info* *parameters*	Destination ID of the module to be replaced, switched or augmented through the API. Information facilitating the replacement, switching or augmentation of the module. This will be specific to the module and the implementation. To be defined.	
4.4.4.14.1	**Description**	
	This message is used to change (replace, modify or augment) the module that is accessed via the API. It is assumed that the replacement is local and accessible. To get a module from a remote source will be achieved in two stages: the first is to obtain the module and store it locally and the second is to use this command to actually use this new version. The command does not imply or infer that the original module is replaced or whether it is used to replace or switch between software or hardware specific components.	
4.4.4.14.2	**Dependencies**	
	This command should only be sent if the module is currently stopped and disabled.	
4.4.4.14.3	**Action**	
	The module should treat this signal as a request to replace/augment or modify a module or part of a module that is accessed via the API. After this command, the module should go through the normal enable/start sequence, including capability exchange. This is similar to the process used during module power up.	
4.4.4.14.4	**Results**	
	If the module is currently enabled and stopped, and the message sent:	• The message should be acted upon. • The status SSI should indicate that the message was successful. • The module should be able to receive selected further messages.
	If the module is currently disabled and the message sent:	• The message should not be acted upon. • It should be treated as a programming error. • The status message should indicate that this is an error. • The module should stay disabled.

4.4.5 Examples

4.4.5.1 The relationship between enable/disable and start/stop

The relationship between enable, disable, start and stop is shown in the diagram below. The enable/disable and start/stop message pairs provide two levels of operation: enabled-but-stopped (EBS) and enabled-and-started (EAS). This is important to allow the fine control over the module and effectively correspond to an on-line and off line state where changes can be made in either state but their effect on the rest of the system is different. In the enabled state, the module does not process data and therefore the effect of any changes will not necessarily be made visible to other modules. If the visibility is through the processed data, then clearly this will mean that the as far as the rest of the system is concerned, the module is not doing anything. In practice, control and status messages are still flowing and so, some visibility can be obtained, but for all intents and purposes, this is not common or available system wide.

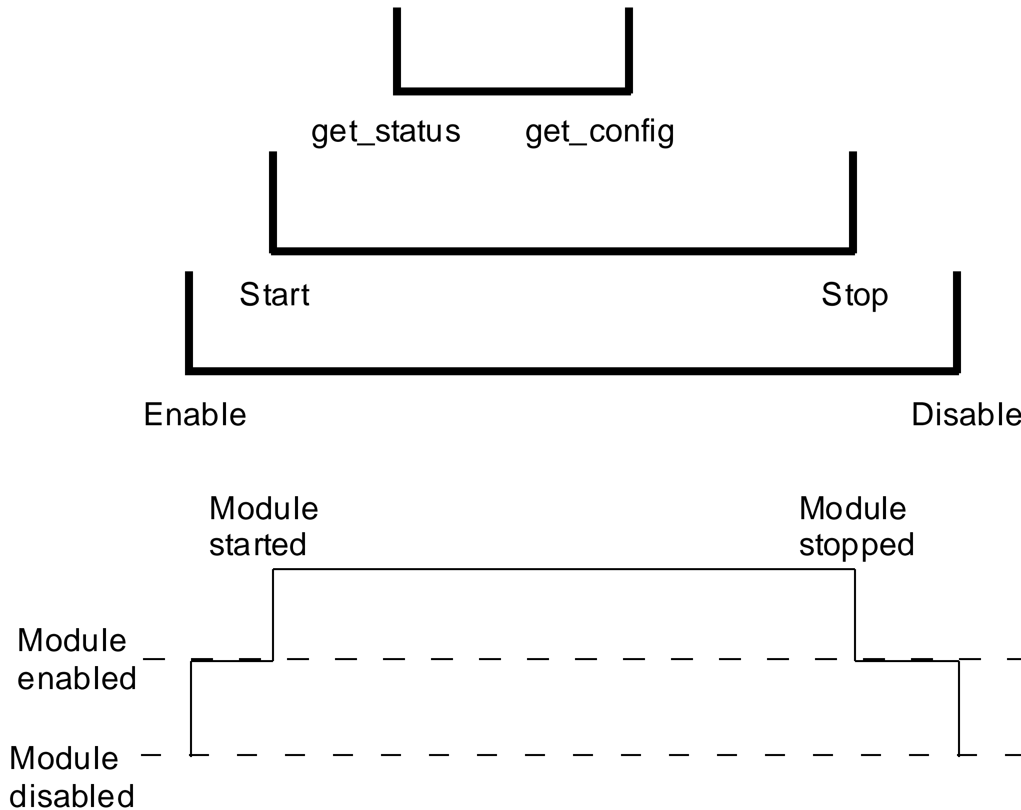


Figure 4.4.5-1 The relationship between enable/disable/start/stop messages

When the module is started, data is processed and thus any changes will become apparent to the rest of the system. It is important therefore to ensure that any changes that are made to the module's configuration are down in a specific way. If the change is minor or expected as part of the normal

operation of the current configuration then it can be done while the module is in the enabled and started state. If the change is radical e.g. changing the modulation scheme to a non-standard one then this change may need to be done only when the module is enabled but stopped to prevent other parts of the system from incorrectly interpreting data or control messages.

As a result, the *set_config/select_config* message pair can operate generically in either of the module's two basic operating states as shown in figure 5.2.4-2 but, depending on the configuration data being changed, it may be recommended to be in one or other states before completing the sequence by sending the *select_config* message.

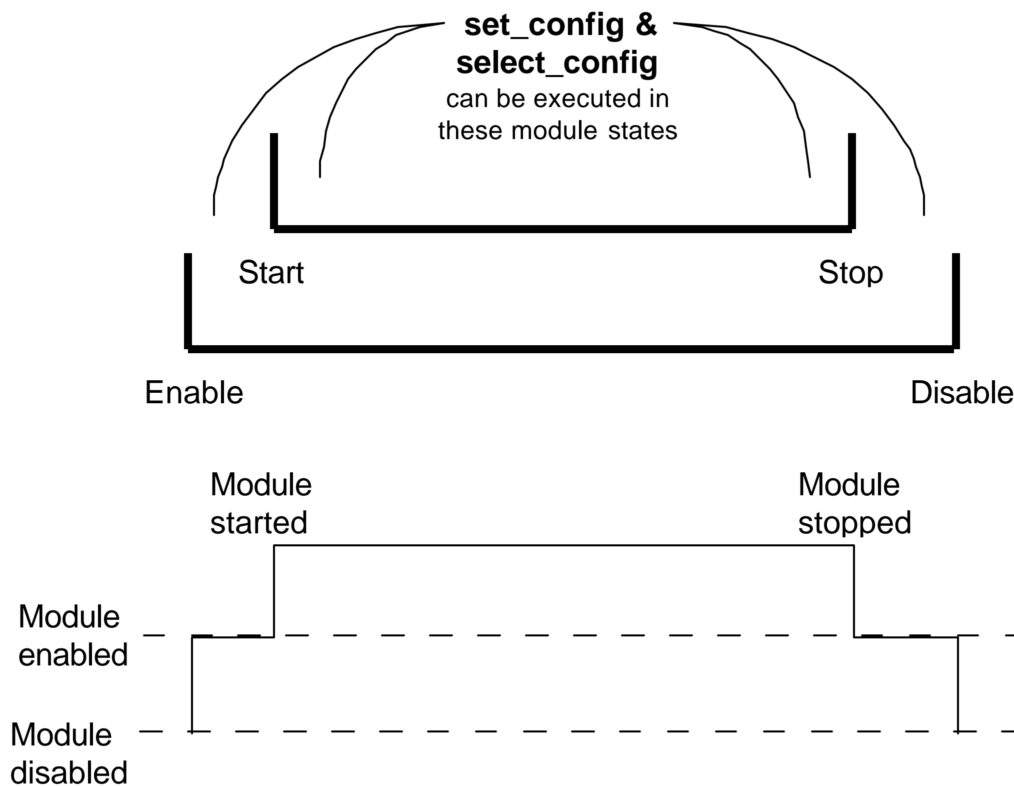


Figure 4.4.5-2 The scope of the set_config and select_config messages

A synchronous start/stop message may be required to simultaneously start/stop multiple modules but has not yet been defined.

4.4.5.2 Module initialization and disabling through the API

With this example, the process of bringing up a module is described. The example consists of two modules (A and B) that communicate via the API. Module A has to initialize module B and it already knows what the module is and that it is ready to proceed. If this was not the case, module A could issue a *get_status* message prior to this process. The option of using the *place_module* message to download or switch to a new implementation is also not included.

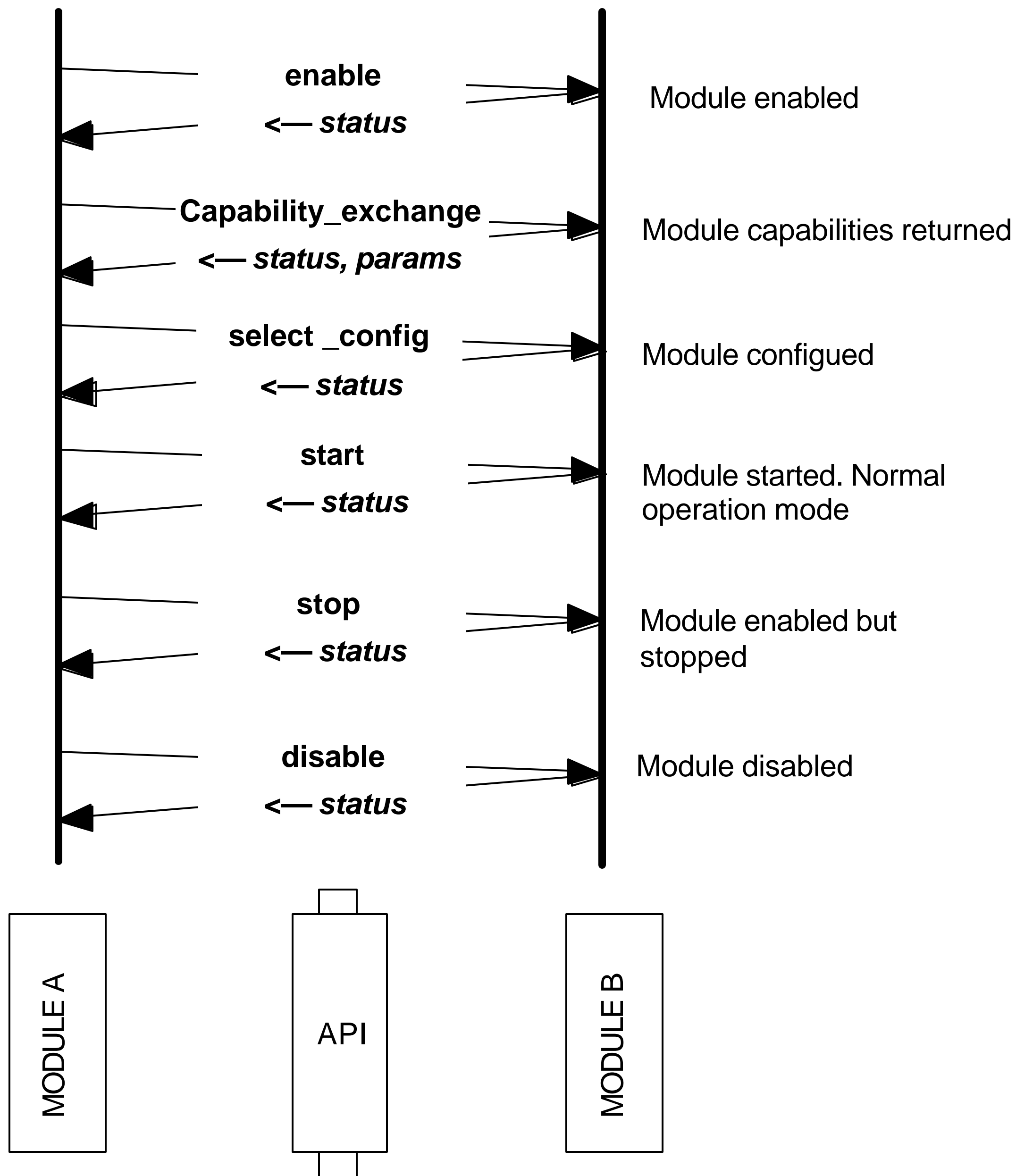


Figure 4.4.5-3 Module initialization and disabling through the API

The process is shown in the ladder diagram. The module status is shown on the right hand side. The process is fairly self-explanatory: the module is first enabled and then interrogated to get its capabilities. These are returned and by using this information, the module can be configured by module A to meets its requirements. In the example, this has been done by selecting the appropriate configuration table. This has assumed that the one of the default tables contains the correct configuration. If this is not the case, then the table must be updated by using the *set_config* message, prior to the *select_config* message. The next stage is to start the module. After the successful completion of this, the module is in a normal operating state and the initialization is complete.

The last two messages are the sequence to stop and disable the module as part of a shutdown process. This may be used when the system is shutdown or prior to replacing the module implementation.

4.4.5.3 Module replacement through the API

With this example, the process replacing or augmenting a module is described. The example consists of two modules (A and B) that communicate via the API. Module B is already operational and is in the enabled-but-started mode. To replace or augment module B, module A must bring module B into the disabled state. This is down by the stop and disable messages. The module is then enabled before issuing the place-module message that instructs the module to replace or augment itself in some way. This is very implementation dependent and could mean simply loading or using a different set of software, switching to a different piece of hardware and so on.

Once this has been completed the new capabilities can be obtained, the module configured and then brought up into the enabled -and-started (EAS) mode.

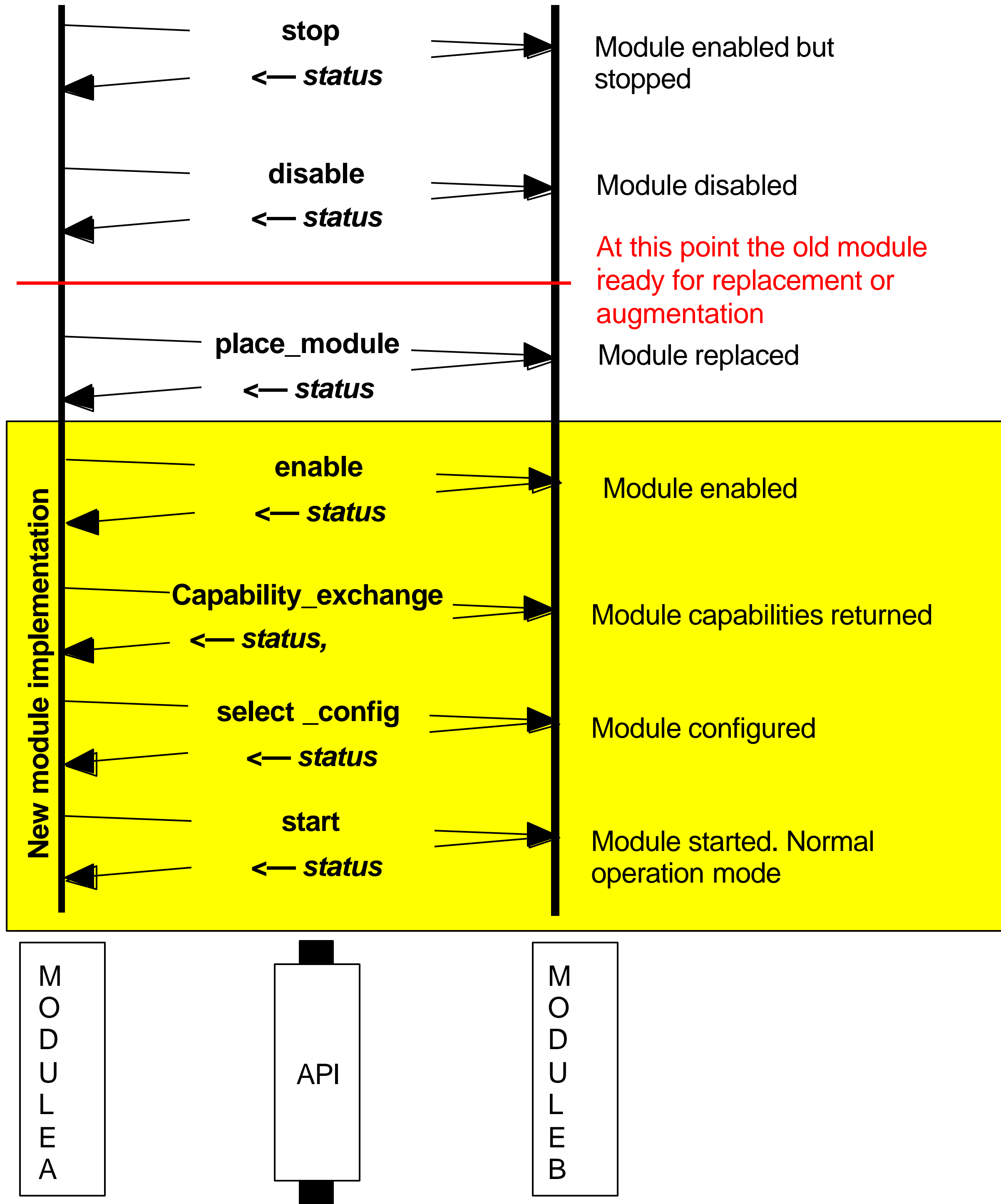


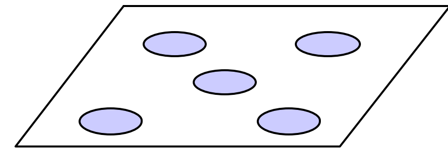
Figure 4.4.5-4 Module replacement through the API

5.0 Frameworks and Design Patterns

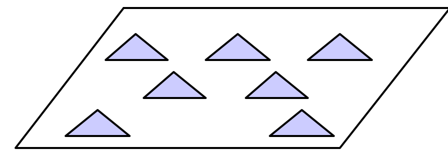
This section introduces a mechanism for overall system design that follows a process based on the use of different views of the system to completely describe its operation on several different levels. Successful implementation of this method enables the designer to describe and manage the relationships between the different system views. The system views are defined as follows : Use Case view, Logical view, Component view, and Deployment view. Descriptors used to specify system operation include: use cases, actors, classes, objects, states, relationships, and interactions. Diagrams are used to delineate the relationships between views and relationships among the elements within each view. Each view describes the way that the different elements of the view relate to one another, this is called a Framework for the design. Taken together, these views and descriptions provide a means of visualizing and manipulating the model's items and their properties to ensure a complete description of the desired system operation is specified at each level. There is a many to many relationship between the elements of each view and the elements of the other views. The relationships of the elements of one view as related to the other views is important but not easy to describe.

Views

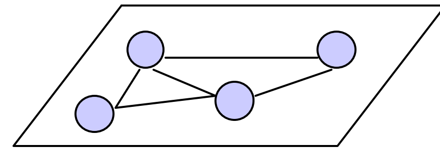
- Determined by the different ways the system is to be used:
  - Use Cases, Actors:
    - Use cases are a basis for incremental evolutionary development
  - Functionality
    - Functional structure & behavior
    - Objects, classes, states, relationships, interactions
  - Implementation
    - Partitioning of functionality into implementations
    - Components, interfaces
    - The way these operate together defines a Framework
  - Physical Elements
    - Handheld, mobile, basestation, satcom
    - General Purpose Processors, DSPs, RF Hardware, BUS structure
    - A particular implementation



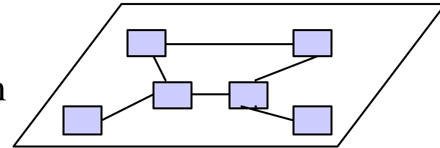
Use Case View



Logical View



Component View



Deployment View

Figure 5.0-1 Use of Views to describe a system

The Use Case View expresses the requirements of a system (or a subsystem) in terms of scenarios, use cases, in which the system interacts with its environment, the actors. A scenario is flow or sequence of interactions between the system and its environment.

The Logical View identifies the conceptual entities of the system, usually as objects and classes with attributes and operations. It also describes the relationships between these entities, and their dynamic behavior in terms of states, state transitions, scenarios etc.

The Component View describes the implementation units of a system. Components may be hardware or software or combinations thereof. Components have defined interfaces, which are described in the component view as well.

The Deployment View describes a specific configuration, in terms of processors, devices and communication mechanisms. This view also allocates the Components from the Component View to its nodes (processors) for a specific instantiation of a system.

What is a Framework

The four views identify the elements that are necessary to build the desired functionality of a system, based on the requirements at the highest level. However, these elements do not operate in a stand-alone fashion, it is necessary to make them cooperate. In order to do this, two additional pieces are needed to complete the design:

1) The components must be designed in a consistent way, in order to allow interoperability. We can call this a Design Pattern for components.

2) The actual interoperability mechanisms must be in place. This includes features like: How components find each other in a distributed environment; What are the mechanisms with which the components exchange data and control information; How are components activated, deactivated, loaded, unloaded. These mechanisms may include standards, design patterns, and special components dedicated to the task of making the other components work together. We can call this structure a *framework*.

Even though components and design patterns may be reused between vastly different environments, only the implementation of the framework (a particular deployment for example) is expected to vary depending on constraints such as memory, power, size, processing power and speed requirements. Therefore, it may be necessary to provide several examples of framework configurations, in which the components may fit for different purposes. The final system depends on the properties of the actual components and the way in which components are combined and configured into a framework. Figure 5.0-2 depicts the framework interaction with views.

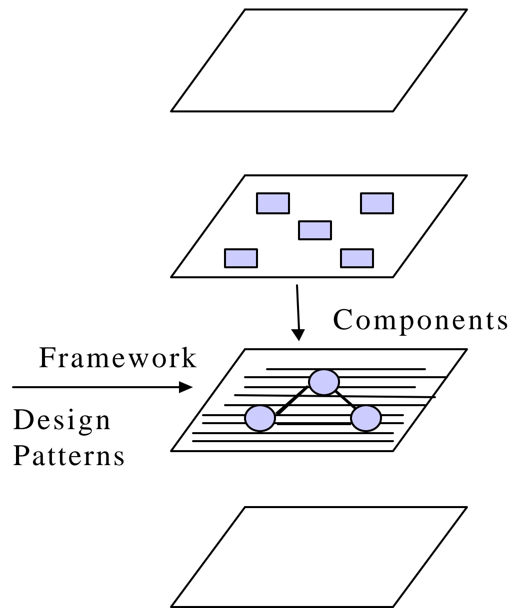


Figure 5.0- 2 Framework Interaction with Views

5.1 Handheld Framework Examples

Intentionally left Blank in this revision.

5.2 Mobile Framework Examples

Below is a description of one method for implementing the mobile SDRF architecture module consistent with the view structure. It is based on an object oriented approach. In the future other approaches may be detailed.

5.2.1 An Object-Oriented Framework

5.2.1.1 Overview

This section describes the Software Defined Radio Forum architecture implemented with an object oriented approach. This design enables objects in a multiprocessor environment to interact under real-time constraints across boundaries imposed by different processor architectures and programming languages. Use of object oriented technology permits development of software with a high degree of encapsulation, and protects individual modules from perturbations in other parts of the system. It facilitates redistribution of objects to processor resources in the system. It also provides a truly open system, facilitating independent third party application development.

Software developed in this environment has inherent design for reuse. Interfaces are described in a formal language so they can be readily published to make the system open for third party developers.

5.2.1.2 The SDRF Framework

The SDRF Framework is an implementation of the Tier 2 interface level (see Section 4.1.2.3).

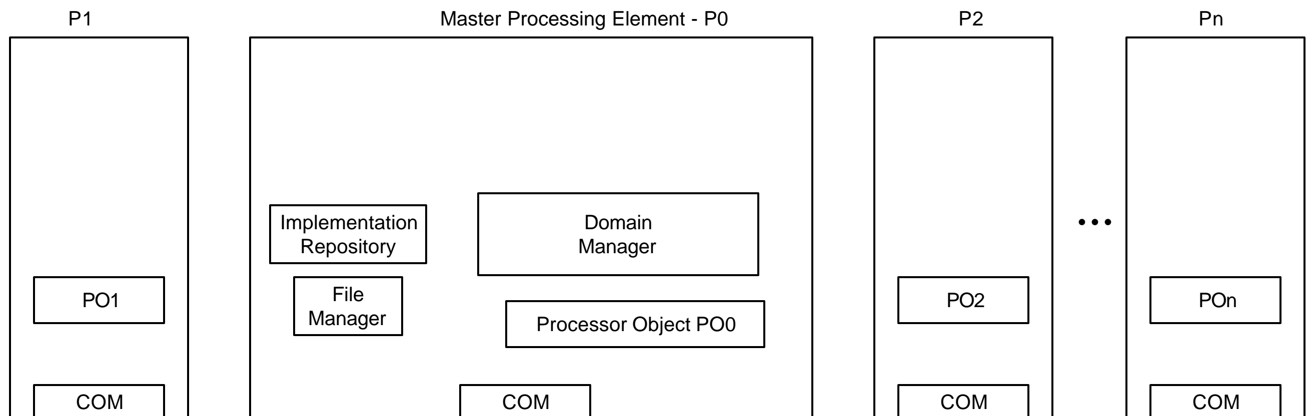


Figure 5.2.1.2-1. SDRF Framework Objects

Figure 5.2.1.2-1. shows the framework components in a multiprocessor environment. Each processing element, n, has a Processor Object (POn) that is responsible for instantiating other objects, loading code, tracking status of available resources, and other housekeeping tasks that may be needed. A communication module (COM) is also positioned in each processor to support inter-processor communication.

One processing element is designated the Master by virtue of having a Domain Manager object resident. The Domain Manager has a well-known address so the Processor Objects can register with it. The Domain Manager maintains a registry of all of the Processor Objects, and communicates with them to load class code, instantiate objects, and commit resources to applications.

The Implementation Repository is the system facility that provides code and structural information needed to instantiate objects, and populate them in the various processors to provide resources for a specific application. A File Manager is the storage facility used to provide persistence for the working data of specific applications. These facilities are shown associated with the Master Processing Element, but they could be located elsewhere, even in a remote system accessed by communication links.

5.2.1.3 Startup

Figure 5.2.1.3-1. shows the SDRF Framework components. The communication function is performed by an object request broker (ORB), or other inter-object communication function and a transport layer for controlling inter-processor messaging.

In addition to the framework objects, the transport layer and Framework Control function are shown. They are the elements that have a role to play in the startup process.

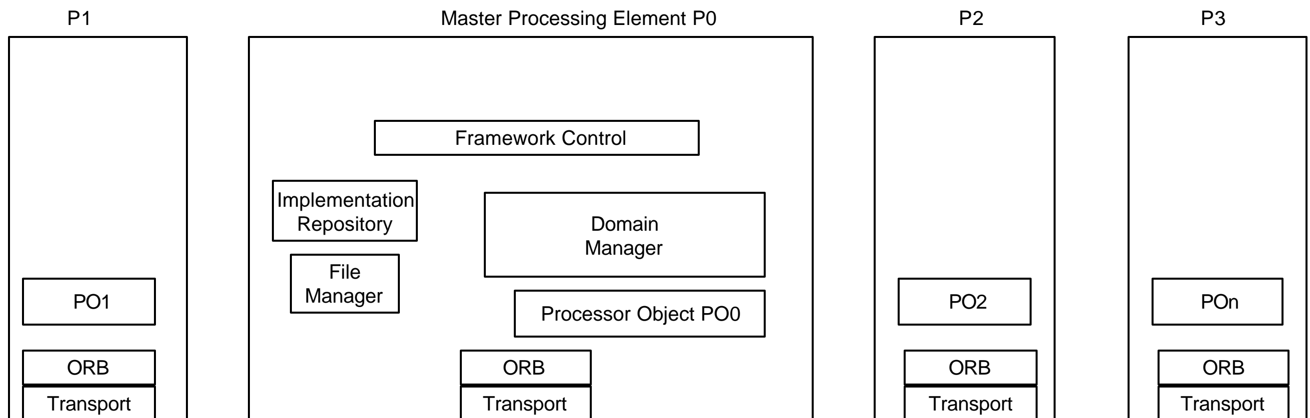


Figure 5.2.1.3-1. SDRF Framework

When power is applied, each processor goes through a boot sequence, which includes loading and instantiation of its Processor Object. In the Master Processing Element, the Domain Manager is also instantiated, followed by the File Manager and Implementation Repository. Each processor object does the necessary startup housecleaning and initiation, and then goes out over the bus through the transport layer to the Domain Manager to register. Processors then retrieve the other code needed for startup.

The system is now ready for an application to request resources, and execute. The system may come up in its last known state, the NONE state (no active application) or by having an application designated for automatic startup as part of the power-on sequence.

All of these objects are persistent for an epoch, the time from power up until system shut-down or reset. Other objects will come and go as various applications are executed. Note that at this level of abstraction there is nothing to differentiate the application - it could be a Software Defined Radio, a bank of elevators, or a factory floor management system. In the next section we will discuss an SDRF application.

5.2.1.4 Mobile System Application of SDRF Framework

Figure 5.2.1.4-1. shows the system configured for an application. The Control function accesses Framework Control Scripts to provide specific details of what objects are needed for each of the processing elements in the system to implement the required applications. Those objects are beyond the scope of the SDRF Framework, and will vary according to the waveform applications being executed, so their characteristics are not specified in this section.

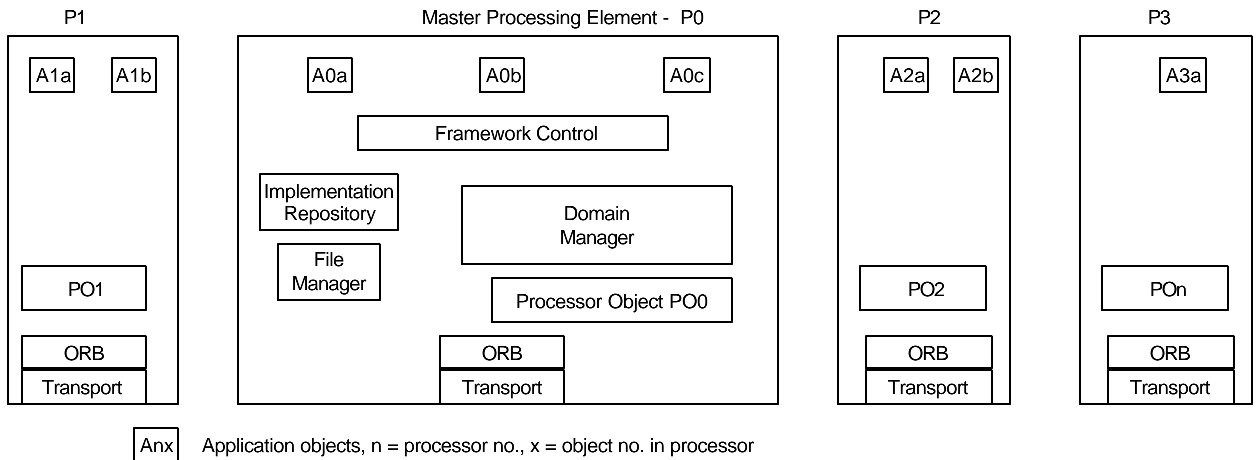


Figure 5.2.1.4-1. Framework Control and Application Objects

To start an application, Framework Control interacts with the Domain Manager to locate and allocate the resources needed for channel setup and waveform instantiation. Then Framework Control orders Processor Objects to obtain the objects they need and instantiate them. After the application objects are in place control is passed to the application.

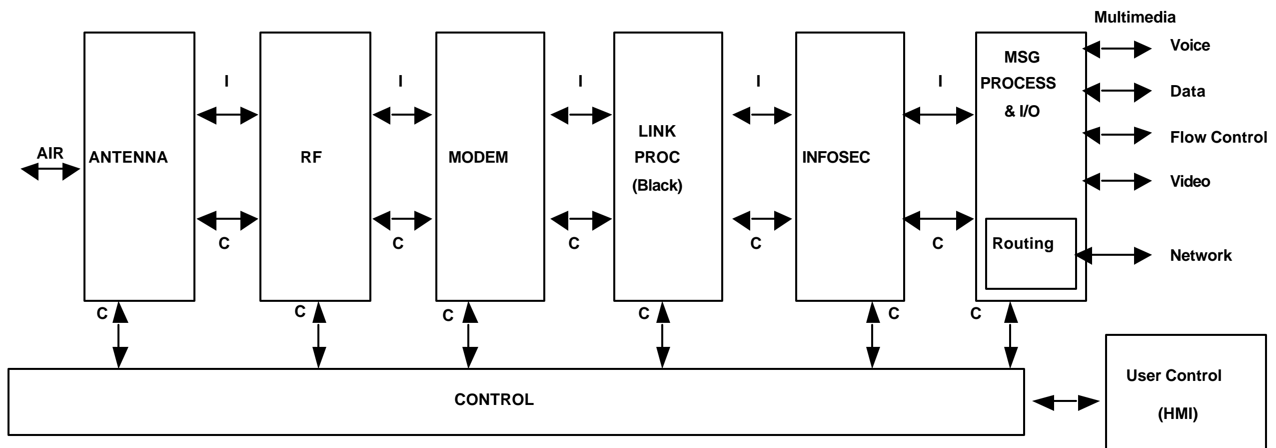


Figure 5.2.1.4-2. SDRF Reference Model

Figure 5.2.1.4-2., derived from the SDRF Reference Model, shows a structure for a typical SDR. At a coarse level of granularity, it specifies the functions allocated to each of the component blocks. The I interfaces are those that carry time-critical data. They must operate with latencies that are sufficiently low to meet the Quality of Service requirements of the system, in effect, real-time. The C interfaces indicate control flows. The SDRF Framework provides different mechanisms for connecting application objects to satisfy the needs of these two types of interfaces.

Control interactions normally use the interobject communication method, and are bi-directional. The calling object blocks until the called method completes providing its service, and returns. This is typical client-server operation.

The I interfaces, however, operate as set of transformations in series. After they are instantiated, control establishes a reference to the transformation method of the next object in the sequence. When each object has completed its task, it passes the data packet on to the next transformation in line through a direct invocation of that object's reference, bypassing the normal interobject lookup. Control information may be included in the I communication paths, in which case it is afforded the same real-time performance as information data.

5.2.1.5 Summary

This description of the SDRF Framework demonstrates how an object-oriented approach implements the SDRF architecture, provides system openness and realizes the benefits of

5.2.2 Object Orientation and CORBA Illustration

5.2.2.1 Overview

As discussed above, object orientation is a software engineering approach that encapsulates data and the code to operate on that data in a package called an object. This section discusses the Common Object Request Broker Architecture (CORBA), a specific approach to an object-oriented software framework appropriate for use in SDRs.

The CORBA specification has been published by Object Management Group, Inc. (OMG), an organization established to define standards in distributed object computing. It is middleware that enables objects in a multiprocessor environment to operate across boundaries imposed by different processor architectures and programming languages. By abstracting the details of inter-object communication out of application objects, it facilitates development of software with a high degree of encapsulation, protecting individual modules from perturbations in other parts of the system.

Software developed for use with CORBA in an object oriented environment has inherent design for reuse. Interfaces are described in the CORBA Interface Definition Language (IDL) so they can be readily published to make the system open for third party developers. Communication between objects on different processors uses the CORBA Common Data Representation (CDR) over the system bus so that independently developed modules can be brought together to provide "plug and play" capability. Object Request Brokers (ORBs) abstract the details of message passing from the application objects, and the General Inter Orb Protocol (GIOP) provides communication between ORBs.

5.2.2.2 Legacy Systems

In the past, systems have been developed using the techniques of structured analysis and structured design. This approach leads to early functional decomposition of the system under development, often before the system requirements are completely understood. Once the subsystems have been defined, however, it is difficult to move modules from one to another.

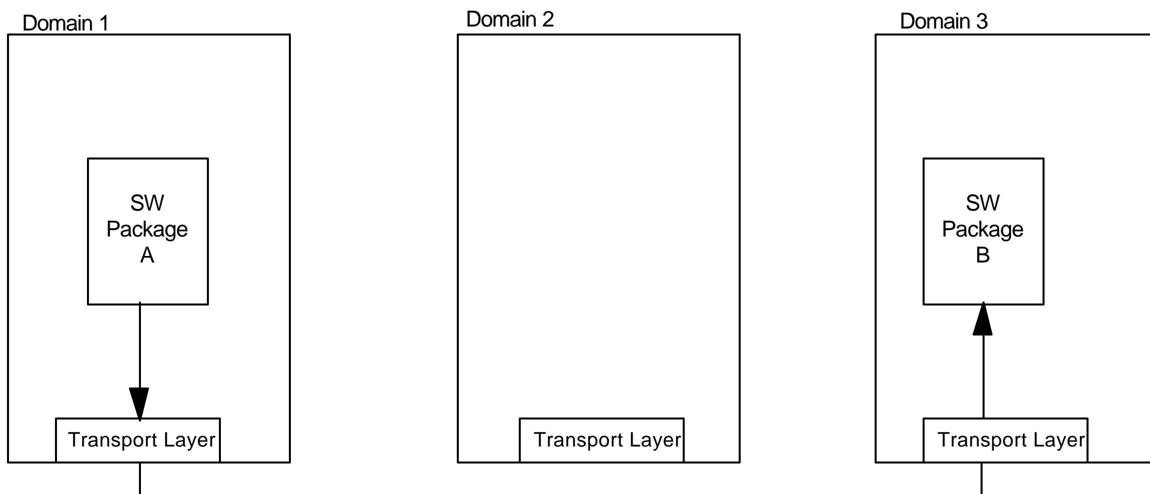


Figure 5.2.2.2-1. Message Passing

Figure 5.2.2.2-1. shows a message passing system. Package A interacts with package B by sending messages through the transport layer and over the system interconnection. Package A contains the information needed to access the transport layer API. It also knows the specific details of where package B is, what messages B knows about, and the data formats to be used in preparing the message.

If a decision is made to move package B from Domain 3 to Domain 2, the developers of all three domains have to coordinate to make the change. Changes will be required in all three domains.

In this type of system it is necessary to have unique message identifiers to avoid conflicts, so a central authority is needed to control them. That complicates independent development of software modules to operate on the system.

5.2.2.3 Object Oriented Software

Object-oriented (OO) software design has been in existence as a computer science concept for a number of years. With the acceptance of languages such as C^{++} and Java that support OO constructs, it has become a mainstream software engineering approach.

An object is a set of data with a set of methods, or computational procedures, to operate on that data. Public methods are invoked by other objects requiring services. Private methods and data are hidden to avoid unnecessary interaction and so they may be changed without unwanted side effects.

When objects on different processors are to interoperate in a CORBA environment, they use the services of an object request broker (ORB).

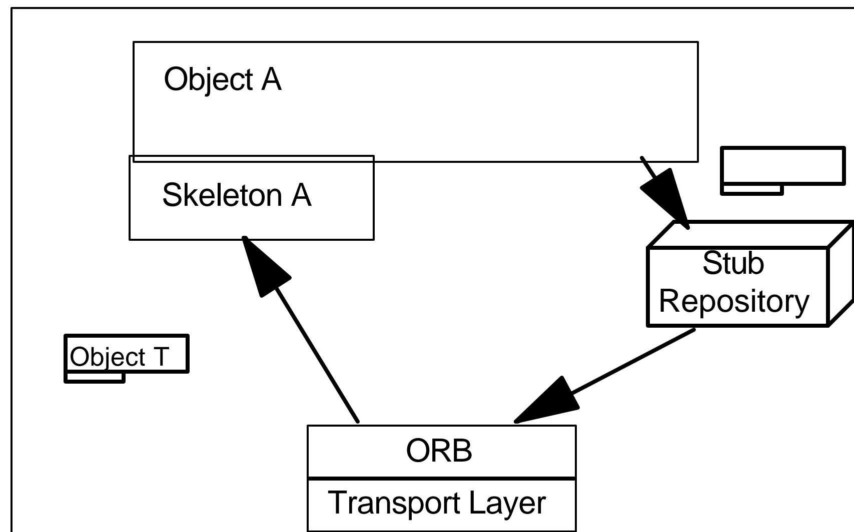


Figure 5.2.2.3-1. The CORBA environment

The CORBA ORB defined by the Object Management Group is illustrated in Figure 5.2.2.3-1. It defines a stub as a proxy for a remote object, and a skeleton to receive method invocations from other

objects. The intent is to abstract the code needed for communication between objects into a common layer rather than requiring that it be built into each one.

Object A can receive invocations from other objects. If the object is on the same processor the invocation uses the normal method invocation or function call mechanism of the language in use. Object T, for instance, can invoke services from Object A directly. If the client object is on another processor, the request comes in through the ORB and the skeleton.

Object A can request services from other objects. Again, local servers do not need to invoke the ORB. To access a remote object, A does so through the remote servers stub.

ORBs operate in pairs to support a given client-server interaction. An ORB is an interface between the operating conventions of the processor on which it resides and the common world of CORBA. Each ORB executes on the same processor as the objects it supports, and is compatible with the language in which they are written. ORBs communicate with each other over some transport mechanism available in the system using an Inter-Orb Protocol (IOP). Data on the bus is transferred according to the rules of the Common Data Representation (CDR) so that the receiving ORB can convert data into the local native format.

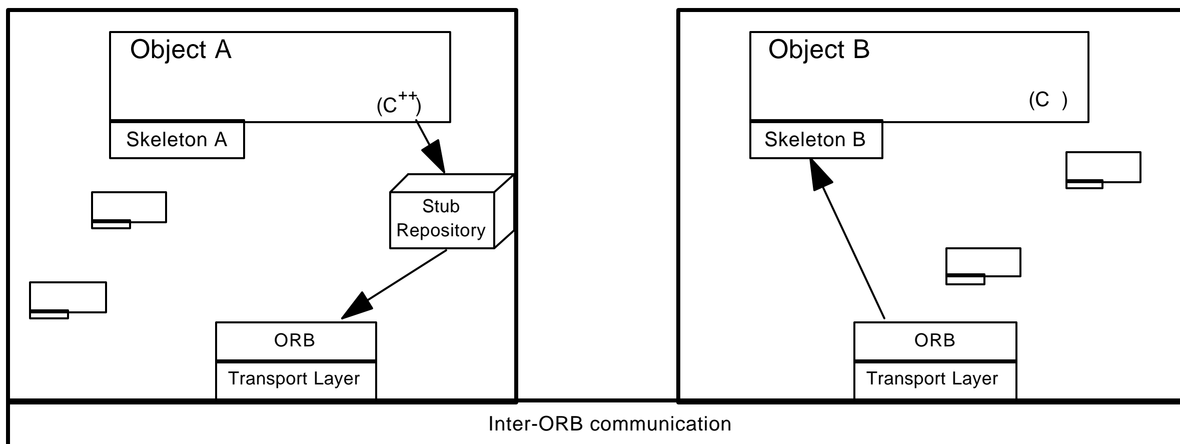


Figure 5.2.2.3-2. Remote method invocation

Figure 5.2.2.3-2 shows a client-server pair. Object A, the client, needs the services of Object B. A invokes the stub for B from the local stub repository, without knowing where B is located, just as if B were a C^{++} object on the same processor. The ORBs handle all the details of message passing, including marshaling the arguments of the call, and converting them into the CDR to operate over the bus. The local ORB works with the remote ORB to perform data conversion, unmarshal the arguments, and pass the invocation to the skeleton for B. That skeleton, written in the C language, makes a function call to B to request the services, and passes the result back to A through the same route. There is no need for either of the objects to know where the other is running, what kind of processor it is on, or what language it is written in.

This high level of abstraction simplifies system integration, and builds reusability into the objects. It does require development of an ORB for each different CPU type used in a system.

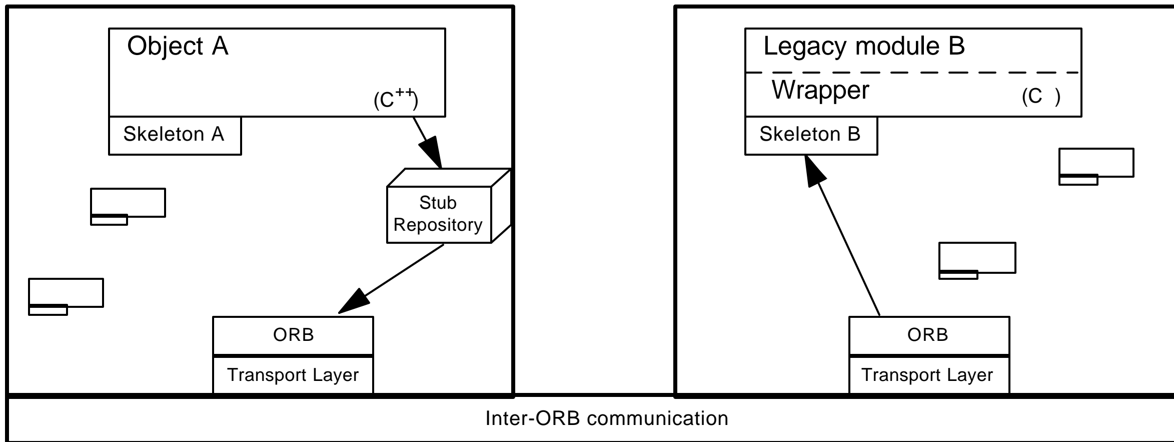


Figure 5.2.2.3-3. Wrapper for legacy code

Except for performance issues, these concepts are not constrained to operation over the system bus. Object B could be on a communication link thousands of miles away. Further, as shown in Figure 5.2.2.3-3, object B may be a legacy module or a whole legacy system adapted to the CORBA architecture with a wrapper.

Historically system design has made use of coarse-grained subsystems that communicate with carefully defined messages. Modifications to one subsystem frequently have repercussions in a number of other parts of the system. The CORBA approach provides a fine-grained structure, populated with objects that are internal surrogates for the real-world system components.

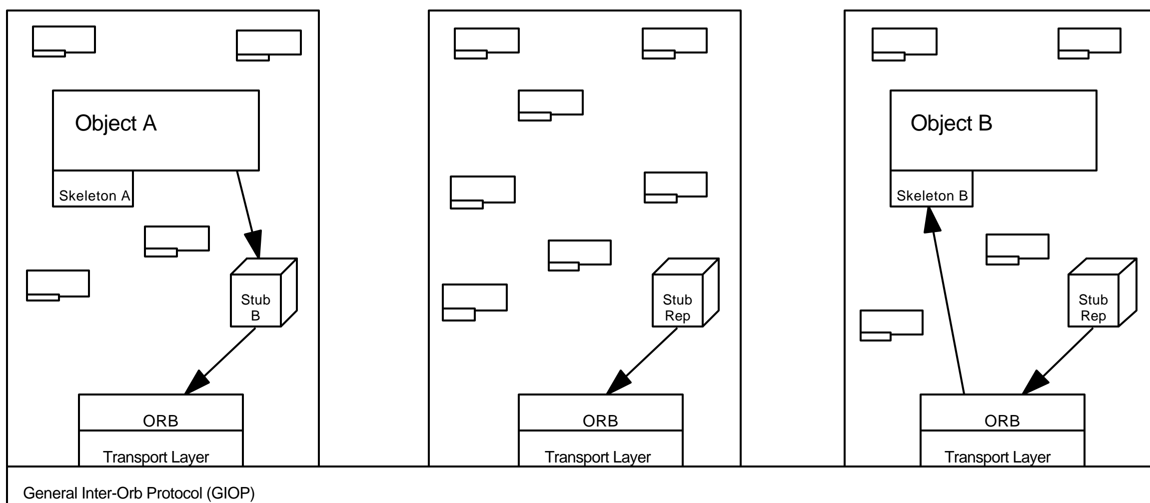


Figure 5.2.2.3-4. Object relocation

In addition, the allocation of objects to processors can be easily changed for load balancing or system modification. Moving object B to the central domain requires only installation of the code and publishing the new location. Object A does not know that a change has taken place.

5.2.2.4 Interface Definition Language

Definition of the SDRF model requires a set of standard interfaces for communication between modules in an open architecture for a Software Defined Radio (SDR). There is a need to specify those open application programming interfaces (APIs) in a fashion that is both useful and unambiguous.

One such technique is to define the exact structure of the message that is passed across the interface. Such a message might be defined as depicted in Figure 5.2.2.4-1.

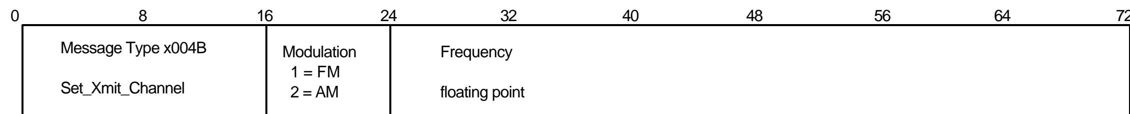


Figure 5.2.2.4-1 Message Structure Example

This message is used to set the transmit channel frequency and modulation type, designated by its message type (hexadecimal 004B). The modulation type is indicated by a code in the third byte. The operating frequency is specified as a floating point number.

Although this is a very clear expression of the message contents, it is not in a form where it can be used with automated tools. Each module designer has to include code to build the message if it is to be transmitted, or parse it into its component parts if it is to be received. Confusion may be introduced when there are many pages of messages in a system definition.

One of the first areas of standardization OMG established was the Interface Definition Language (IDL), a means of describing the public portion of an object. Following the model established by C^{++}, IDL defines a syntax for classes of data, and identifies methods to work on that data. But IDL is purely declarative, it does not describe the work performed by the interfaces it defines. That information is presented in other documentation accompanying the IDL.

IDL has seen enough use so that software tools are available to work with it. In particular, there are cross compilers for a number of languages that take IDL as input, and develop the declaration section of the module in C, C^{++}, Smalltalk, Ada, Java, etc. Because of this cross-language capability IDL is an appropriate way to describe the details of inter-object interfaces after the modules have been identified and the interface connections described at a high level.

The structure of IDL is described in the following pseudo-coded module. Language keywords are in *italics.* Square brackets [...] indicate optional elements. Angle brackets indicate identifiers with their modifiers. Curly brackets {...}, parentheses (...) and semi-colons ";" are part of the are part of the

language grammar, and must be present as delimiters in the positions given. Comments are in sans-serif font.

```
module <identifier>                    -Module provides scoping for related interfaces
// Comment 1.            -Any characters on a line after the // symbol are comments
// Comment 2.     -Comments are treated as white space, to be ignored by a compiler
{
     <types>;                          -Various declarations with module-wide scope
     <constants>;
     <exceptions>;

     interface <identifier> [:inheritance]           -Equivalent to a class
          {
               <types>;             -Declarations with scope limited to this interface
               <constants>;
               <attributes>;
               <exceptions>;

                         -Operation, equivalent to a class member function or method
               <return_type> <identifier> ([<direction parameter>])
                    [raises (exception_name);
          }
}
```

Module. The module is the basic container for IDL. Its purpose is to provide an overall name that can be used as part of a hierarchy to provide unique naming of its internal elements, It also provides a reasonable way to structure multiple interfaces. Modules may be nested.

Comment. Comments can be used in the definition to provide further documentation. Either the C language form /* ...*/ or C++ form //... can be used. They are ignored by a compiler parsing the code.

Module Level Definitions. At the top of the module a variety of definitions can be made. Items declared here have scope throughout the module.

Interface. The interface is the primary working element of IDL. It is similar in nature to a class in C++. In particular, it can inherit from a base class, receiving definitions and operations from the higher level class, and extending them. Multiple inheritance is permissible, but must be free of conflicts.

Interface Level Definitions. Definitions at the interface follow the same syntax as at the module, but have scope local to the interface.

Operations. Operations are the equivalent of methods in C++, and are identified by the parentheses following them. They contain any parameters communicated in and out of the operation, and must be present even if empty.

The following is an example of an IDL description from an information transfer system. The system has the concept of virtual link, a collection of resources that have facilities for Information and Control set up

so that they can operate as a channel. There are different sets of commands needed to initialize the system, set up channels, and operate channels once they are set up. These classes differ in their performance requirements and in the amount of overhead needed. The class of commands used for such operations as changing frequency or switching from receive to transmit is called Link_Command in this example. The operations in the example inherit from the base Link_Command interface and add their specific functionality.

The operator at the user interface has the ability select a frequency, and FM or AM as modulation types. Transmit and receive can be on the same or different frequencies as commanded by invoking the set_xmit_channel and set_rcv_channel operations. The initial condition in receive mode, when the Push To Talk switch is activated the system switches to transmit mode.

```
module AM_FM_Virtual_Link {        // Namespace for FM/AM VHF/UHF

  interface Link_Command {
    attribute float frequency;
  };

  enum ModulationType { AM, FM };

  exception OutOfRange {           // System can't do it
    string errormsg;
  };

  exception IllegalFrequency {     // Frequency not authorized
    string errormsg;
  };

  interface Xmit : Link_Command { // Transmit inherits from Link_Command
    void set_xmit_channel (in float frequency, in ModulationType Modulation)
                          raises (OutOfRange, IllegalFrequency);

    void transmit ();              // PTT is asserted
  };

  interface Rcv : Link_Command {  // Receive inherits from Link_Command
    void set_rcv_channel (in float frequency, in ModulationType Modulation)
                        raises (OutOfRange);
    void receive ();              // Initial condition, PTT has been released
  };
};
```

IDL provides a method for interface description that is both rigorous and practical. Full specification of IDL is found in only 38 pages as Section 3 of the OMG document The Common Object Request Broker, V2.1, dated August, 1997.

5.2.2.5 Summary

CORBA is an effective standard for implementation of SDRs. In this section we have discussed how it abstracts details of communication between objects out of the application code into the common system

framework. Doing so reduces the effort required from application programmers, and improves system openness. We have also described how IDL provides an effective mechanism for documenting interfaces.

5.2.3 Mobile Framework

This section defines the software framework for an open, distributed, object-oriented, software-programmable radio architecture derived by the members of the SDR Forum's Mobile Working Group. The object-oriented methodology used to define the software architecture is based on the Institute of Electrical and Electronic Engineers' Recommended Practice for Architectural Descriptions (draft), IEEE P1471.

5.2.3.1 Definitions and Guidelines

This section defines the key terms used in the description of the software architecture including the Core Framework (CF) and the Operating Environment (OE). Together with a Rule Set (part of the architecture definition), the guidelines for design and implementation are also provided.
The software architecture is composed of the Operating Environment (OE) which includes the Core Framework (CF). The Rule Set for the software architecture comes from the OE and includes the design rules embedded in the attributes (behavior and interfaces) and in the structure defined by the CF and OE.

5.2.3.1.1 Core Framework (CF)

The CF is the essential, "core" set of open interfaces and services that provide an abstraction of the underlying software and hardware layers for "non-core", i.e., non-CF, software radio applications. The CF consists of

- Base CORBA interfaces (*Message*, *MessageRegistration*, *LifeCycle*, *StateManagement*, and *Resource*) that are inherited by core and non-core software applications
- Core applications (*DomainManager* and *ResourceManager*) that provide framework control of resources via CORBA interfaces
- Core services (*Logger*, *Installer*, *Timer*, *FileManager*, *FileSystem*, and *File*) that support both core and non-core applications via CORBA interfaces
- An optional CORBA *Factory* interface for controlling the life span of core and non-core applications
- A *DomainProfile* file that describes the properties of hardware devices and software resources in the radio.

Elements of the CF are colored with aqua shading in the various diagrams throughout this document and are also denoted with *Italics* in the text.

Figure 5.2.3.1.1-1 is a diagram showing the elements of the CF and their relationships.
This figure illustrates the readability of the Unified Modeling Language (UML) diagrams used to define the software architecture. In the figure, boxes represent classes of things (in this case CORBA software objects and interfaces). Lines connecting boxes represent associations between things. An association has two roles (one in each direction). A role can optionally be named with a label. The role from A to B

is nearest B, and vice versa. For example, the roles between *DomainManager* and *ResourceManager* can be read as: "A DomainManager oversees one to many *ResourceManager*(s)" and "one to many *ResourceManager*(s) register with a *DomainManager*". Roles are one-to-one unless otherwise noted. A role can have a multiplicity, e.g., a role marked with a '1..*' is used to denote "many," as in a one-to-many or many-to-many association. A diamond (at the end of an association line) denotes a "part-of" relationship. For example, *Files* are part of a *FileSystem*. A triangle arrowhead (at the end of an association line) is used denote an "inheritance" relationship between a parent class (pointed to) and a child class (or "subclass"). For example, a *Resource* inherits all operations of the *Message*, *MesssageRegistration*, *Life-Cycle*, and *StateManagement* interfaces.

The stick-man figure (referred to as an "actor") represents an external source of command and control (C^2) for the radio. The *DomainManager* provides specific interfaces for controlling and monitoring the resources of the radio. Other (non-core) applications may also provide more specific interfaces for controlling and monitoring their specific behavior.

Next to the block labeled "Non-Core Applications" is a list of the example types of applications that will be hosted on this framework. These applications inherit attributes from the class called *Resource*, which in turn inherits the *Message*, *MessageRegistration*, *LifeCycle*, and *StateManagement* interfaces. Applications are managed by and/or make use of the services provided by the remaining elements of the CF. For example, a non-core application Access *Resource* provides the interface into hardware devices that are instantiated in a specific implementation. This interface provides the connection, for example, to a Hardware Modem or a Security Module. Section 2.2.5.1 discusses the CF in more detail.

A complete conceptual model of the SDR Software Architecture is depicted in Figure 5.2.3.1.1-2. This model provides an informative summary of the key architectural concepts and depicts the inter-relationships of the CF components, non-core application resources, and physical devices.

Example hardware devices are depicted at the bottom of the figure. Example types of non-core application resources, including waveform resources, adapters, access, and utility resources are depicted in lower half of the figure. The CF components are depicted in the upper half of the figure. An "actor", i.e., a human user (or an application acting on the behalf of the user, e.g., a workstation application, or a numeric keypad application) controls the SDR framework.

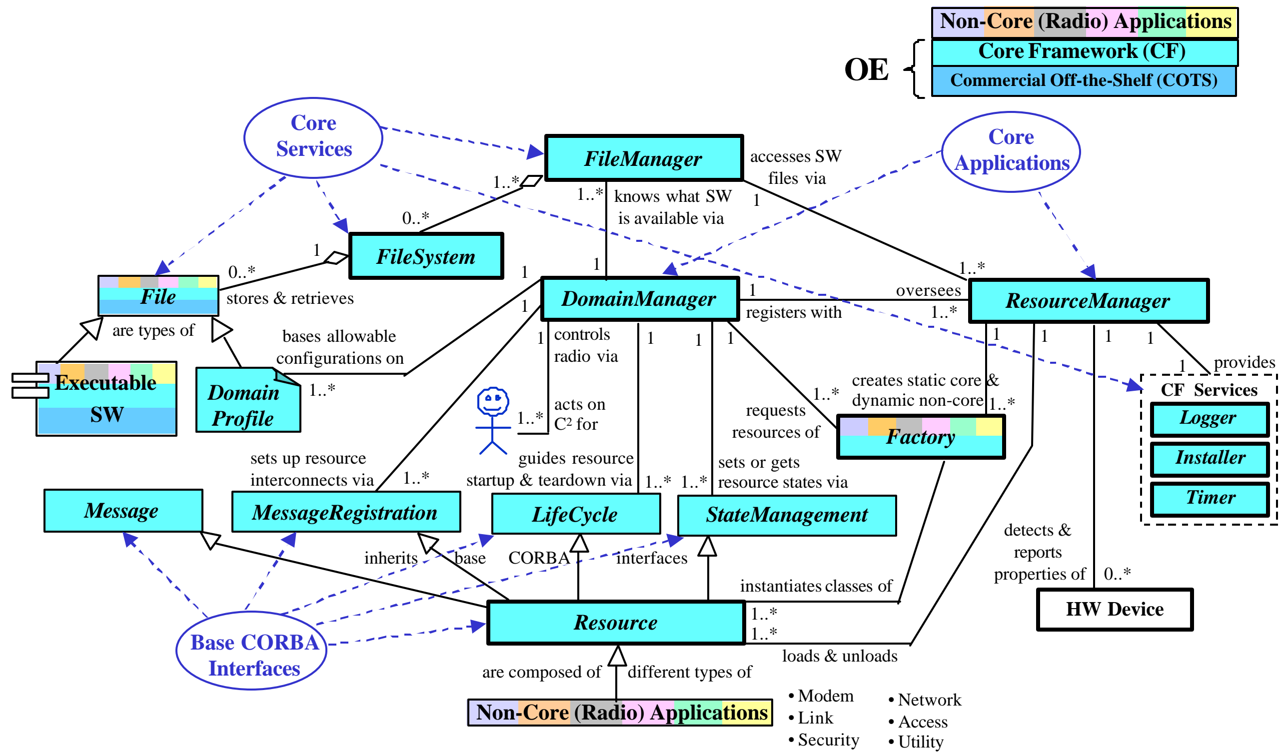


Figure 5.2.3.1.1-1. The SDR Core Framework (CF)

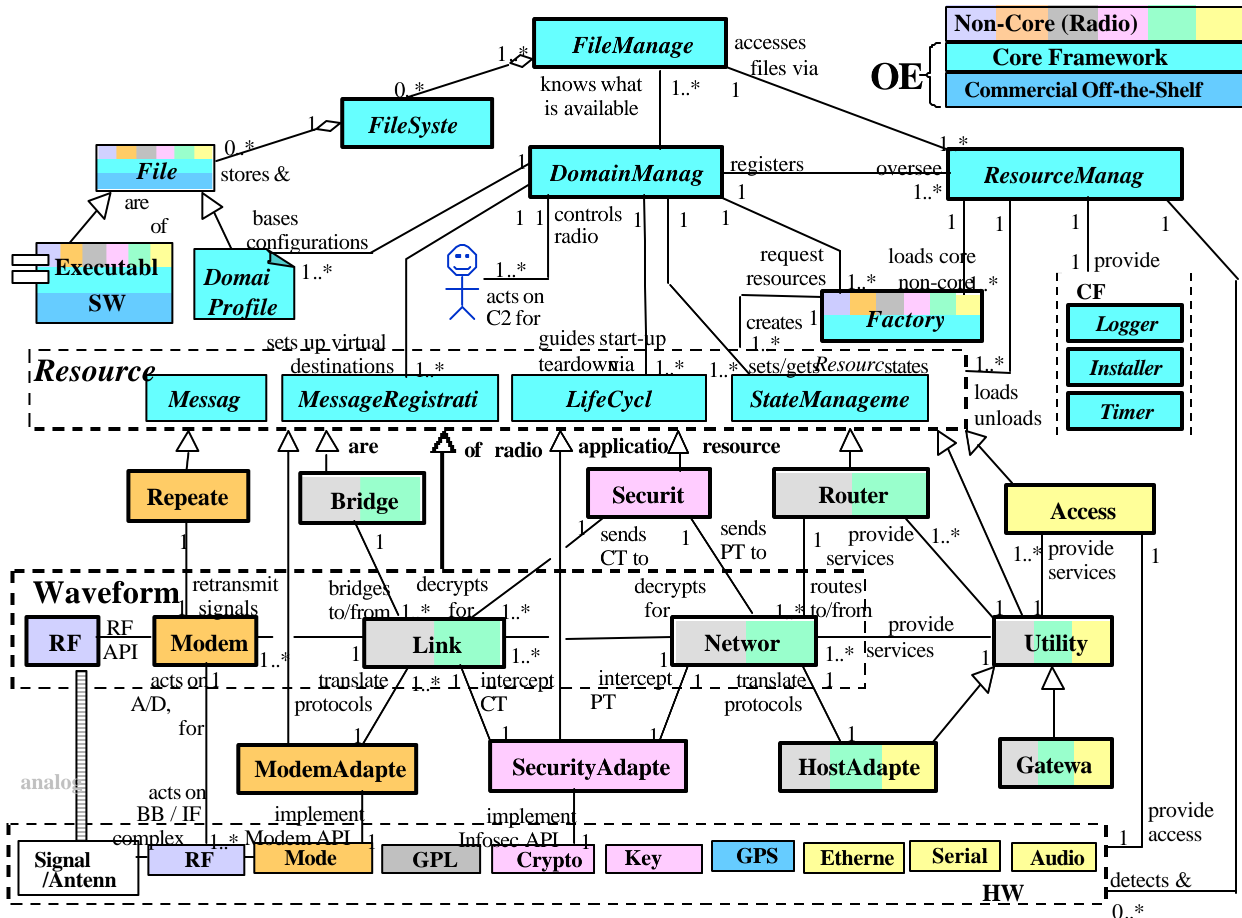


Figure 5.2.3.1.1-2. The SDR Software Architecture

5.2.3.1.2 Operating Environment (OE)

The OE is the combined set of CF services and commercial off-the-shelf (COTS) infrastructure software (e.g., bus support packages, operating system and services, and Common Object Request Broker Architecture (CORBA) Middleware services) integrated together in a SDR implementation. The OE also defines a complete development environment for application software suppliers and reduces the non-recurring engineering cost associated with developing new capability waveforms.

Figure 5.2.3.1.2-1 illustrates the OE and its relationship to the CF. CORBA provides a well-defined structure and logical definition between the SDR software objects.

Radios that do not have security requirements would need neither security software applications nor separate black (secure) and red (non-secure) hardware busses.
This diagram also introduces the concept of "adapters" which are ways to incorporate legacy and non-CORBA compliant elements into the radio. Adapters are further described in section 2.2.5.1.4.2.

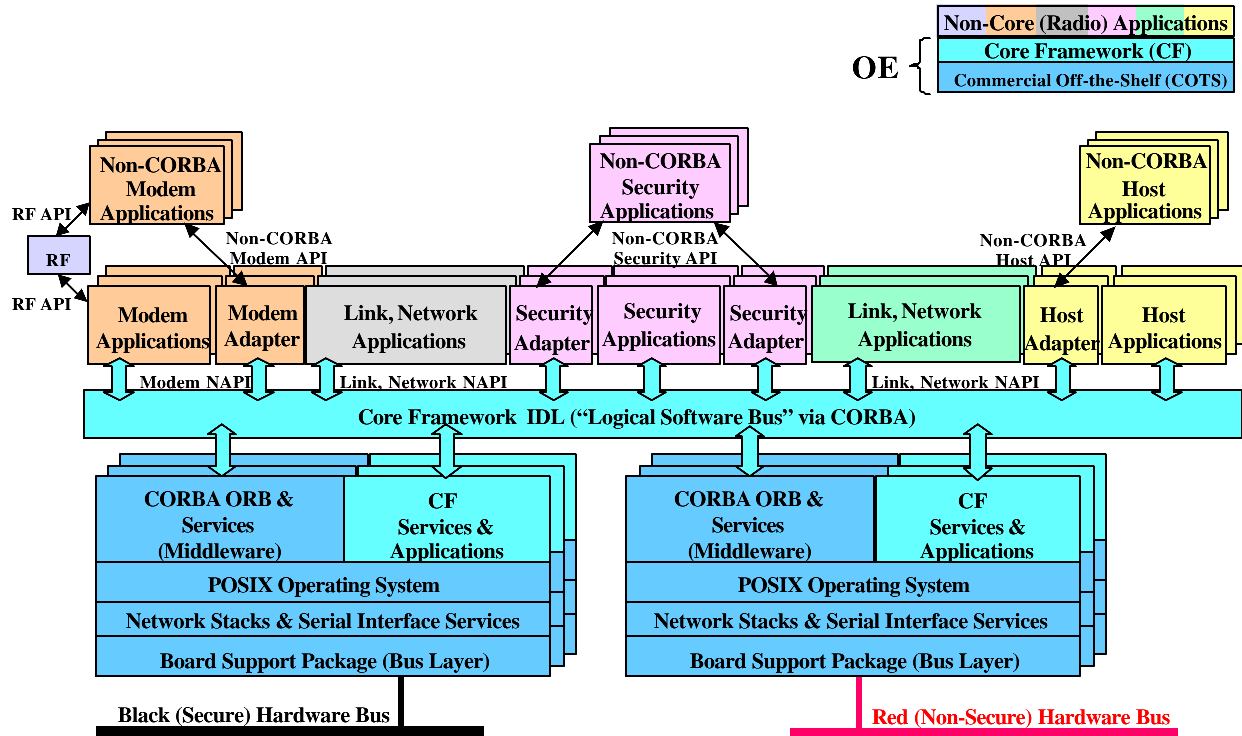


Figure 5.2.3.1.2-1. The SDR Operating Environment (OE)

_0_, which is derived from the SDR functional reference model and broadly categorizes the functional roles that may be performed by and/or controlled by SDR software entities and the interface relationships among these software entities. This view also illustrates the Core Framework (CF) and its role in providing class objects from which user application objects inherit common types of behavior and interfaces.

Structural View – Section 0, which illustrates the multi-layered structure of the Operating Environment (OE), including bus support, COTS software, industry standard protocol stacks, Core Framework (CF) services, and SDR waveform and networking applications.

Logical View – Section 0, which includes more detailed descriptions of the CF as well as the relationships between the various CF objects and interfaces. Extensive use of Unified Modeling Language (UML) diagrams and Interface Definition Language (IDL) of the Common Object Request Broker Architecture (CORBA) support the textual descriptions of the CF.[1]

Use Case View – Section 2.2.6, which contains scenarios that depict examples of how the elements of the Software Architecture, including COTS software, CF services, and SDR applications collaborate or interact with one another to satisfy user requirements.

5.2.4.2. Software Architecture Rationale

This section summarizes the rationale behind the selection of the critical Software Architecture components. The critical software components of the architecture can be grouped into three categories:

1. Middleware Selection
2. Operating Environment
3. Core Framework (CF).

The critical software components support the following SDR goals:

1. Wide Industry Acceptance
2. COTS Availability
3. Distributed-Object Computing Architecture
4. Reuse and Portability
5. Scalability
6. Support for Different Domains
7. Performance
8. Security and Safety.

5.2.4.3. Middleware Selection Rationale

Support for a distributed architecture is a crucial goal of the SDR. Middleware is the software used to transfer messages across a distributed architecture. CORBA was selected as the middleware component based on the following rationale:

1. CORBA is an open standard from the Object Management Group (OMG). Over 800 companies are members of the OMG. CORBA is a widely accepted industry standard and there are many different COTS CORBA vendors. CORBA abstracts the bus hardware under an Object Request Broker (ORB) software bus. Abstraction of the bus hardware allows the applications to be ported over different buses. The CORBA specification calls out minimumCORBA and the Portable Object Adapter (POA). The minimumCORBA

---

[1] CORBA, IDL, and UML are open industry standards defined by the Object Management Group (OMG), a consortium of over 800 worldwide members. The OMG's charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop a heterogeneous computing environment across all major hardware platforms and operating systems. These standards are publicly available on the Internet at http://www.omg.org.

specification calls out the smallest subset of CORBA functionality and the ORB interface. This ability to subset the CORBA functionality supports scalability. The POA standardizes the interface to the ORB on the server side. Applications that are written to sit on a POA and/or minimumCORBA ORB can be more easily ported to a different vendor's ORB.

2. CORBA is language neutral. CORBA uses an Interface Definition Language (IDL) to define the interfaces between distributed objects. There is movement away from programming applications from scratch using low-level protocols. Since the process is automated, errors don't arise from manual handcrafting which speeds up coding, testing, and integration. All this makes the code more reliable and maintainable. Language specific IDL compilers convert the IDL into software headers, packages, etc. Multiple languages could be used across the distributed architecture. Legacy code would not have to be re-written, but would have to be wrapped with the CF.

3. The platforms that communicate using CORBA do not all have to be the same type of platforms. The CORBA framework does not depend on processor type, bus type, or operating system. Big endian processors can communicate with little endian processors. Problems with word size are reduced. Windows applications can even talk to Unix applications.

4. In the CORBA framework, a client and a server can be implemented in different languages but still communicate through the ORB. Different IDL compilers convert IDL into skeletons, stubs and interfaces in different programming languages. The ORB marshals and de-marshals data between local processor and language-based formats and the CORBA a "wire-format."

5. Middleware bridges and Environment Specific Inter-Operable Protocols (ESIOP) can be used to connect to DCE, RPC, DCOM and Java Remote Method Invocation (RMI). Commercial vendors supply RPC for the Texas Instrument (TI) C4x processor. This would allow a CORBA capable module to talk to a digital signal processor (DSP) that uses RPC. The CORBA General Inter-ORB Protocol (GIOP) allows a CORBA implementation to be ported over different transport layers than just the TCP/IP based Internet Inter-ORB Protocol (IIOP).

6. Distributed Common Object Model (DCOM), is Microsoft proprietary More vendors support CORBA and CORBA is supported on more platforms than DCOM. CORBA is more applicable to the real-time domain than DCOM because of scalability and speed related services.

7. CORBA defines several services that support faster than "standard" speed. The CORBAreal-time service can provide thread priorities and present quality of service (QoS) options to the application. The CORBAtelecom service describes the use of a low-latency by-pass of the standard CORBA stack. Control is performed using standard CORBA Interfaces. Data is streamed outside of the CORBA ORB.

8. Timing studies have been made of the data flowing up and down a protocol stack. These studies indicate that less than 20% of the processing time is spent in the CORBA part of the stack. It should be remembered that the majority of the processing performed by CORBA is not unnecessary overhead. The CORBA ORB, stubs, skeletons and helper files "automate"

what would have to be done "by hand" in a traditional message based system. This automatic generation of the code reduces the SLOC count that has to be developed and reduces development time and cost.

5.2.4.4 CORBA Timing Studies

The synchronization of data between two distributed objects, a Producer and a Consumer, can occur in one of four ways:

1. No Synchronization – if the socket is not available, the data is dropped.
2. Synchronization with Transport – the data makes it to the socket but one doesn't know if the Consumer received the data.
3. Synchronization with Socket – the data makes it to the server (Consumer) but the Producer doesn't wait for the Consumer to process the data. This technique means the data is guaranteed to make it to the Consumer.
4. Full Synchronization – the data makes it to the server (Consumer) and the Producer waits for Consumer to process the data.

The implementation of a CORBA operation by an ORB depends on the location of the Consumer and Producer and the capabilities of the ORB. The implementations, listed in order of best timing performances, are:

A. Collocated – where the Consumer and Producer are within the same address (process) space. The collocation call acts as a virtual language Function call and there is no marshalling of data**.** The collocated timing is a maximum of 20 microseconds depending on the ORB and the processor. The SDRF has verified this time on a COTS 200 MHz Pentium processor.

B. Local – where the Consumer and Producer are in separate address space (different processes) but on the same processor. The times for local process to local process CORBA communication varies on the processor speed, ORB capability, and whether two different ORBs are being used for the local communication. When two different ORBs are used, the IIOP communication mechanism is being used. If the same ORB is being used for both the Consumer and Producer objects, then the vendor may have optimized the communication between them by using a local Object Adapter (OA) instead of an IIOP OA, thus no marshalling of the data occurs. This can make the local communication twice as fast as the IIOP transfer. This local OA could be implemented using shared memory or UNIX Domain Protocol for data transport. One CORBA vendor, ORBexpress, provided the following timing information for local communication:

   1. Using a one-way operation passing 64 bytes of data:

      a) 50 microseconds on a Sun UltraSPARC 5 with a 270 MHz UltraSPARC IIi (small cache) running Solaris 2.6.

      b) 40 microseconds on a PC with a 266 Mhz Pentium II processor running Windows NT.

2. Using a two-way operation passing 64 bytes of data:

   a) 310 microseconds on a PC with a 266 Mhz Pentium II processor running Windows NT. The ORB used about 80 microseconds of the total processing time.

   b) 315 microseconds on a Sun UltraSPARC 5 with a 270 MHz UltraSPARC IIi (small cache) running Solaris 2.6. The ORB used about 115 microseconds of the total processing time.

Other vendors show local times from a low of 200 microseconds to a high of 900 microseconds depending on the processor speed and length of the data transfer.

C. Processor-to-Processor – where the Consumer and Producer are on separate processors. The communication conforms to the GIOP/IIOP protocol at the above synchronization levels. The most important features of the ORB with respect to the performance of the request delivery mechanism are the speed of marshalling, the efficiency of the communication mechanism, the speed of dispatching (especially with respect of scalability and demultiplexing in the Object Adapter), and the speed of unmarshalling. These are areas that the CORBA vendors are actively working and competing with another to improve the performance of their products. One CORBA vendor, ORBexpress, provided the following timing information for processor-to-processor communication:

   1. Using a one-way operation passing 64 bytes of data:

      a) 50 microseconds between a Sun UltraSPARC 5 with a 270 MHz UltraSPARC IIi (small cache) running Solaris 2.6, and a PC with a 266 MHz Pentium II processor running Windows NT.

   2. Using a two-way operation passing 64 bytes of data:

      a) 420 microseconds between a Sun UltraSPARC 5 with a 270 MHz UltraSPARC IIi (small cache) running Solaris 2.6, and a PC with a 266 MHz Pentium II processor running Windows NT.

      b) 370 microseconds between two 300 MHz UltraSPARC workstations.

Another test report comparing various ORB vendors using one-way operations for sending data of varying lengths provided additional information. The Visibroker showed the best timing for one-way operations with no parameters of 130 microseconds per call, second was Tao at 180 microseconds. Visibroker's best one-way times were 671 microseconds for 1k array of characters and 6.25 milliseconds for 10k sequence of characters. The Tao was second best at 701 microseconds for a 1k array of characters. The test noted that these results are limited by the network bandwidth. The two-way (synchronization) tests showed the performance was at least twice as slow as a one-way operation. The SDRF has verified some of these two-way times using COTS 200 MHZ pentium processors connected by a 100BaseT Ethernet. For all the tests, the server was running on a Compaq Proliana 333MHz bi-Pentium II machine with 512Mb of RAM and the client was running on a 266MHz Pentium II with 64Mb of RAM. Both were running on Windows NT 4.0. The client and the server machines were communicating thanks to 16Mb Token Ring cards.

The scalability implementation of an ORB impacts the timing performance of an ORB server. Scalability is measured by how well an ORB scales based upon the number of object adapters, the number of

objects within an object adapter, the number of operations in an interface, and the nesting level of interfaces. There are real-time ORBs (Visibroker, ORBexpress, TAO, etc.) currently addressing scalability where no performance degradation is detected, which means the time is basically the same for dispatching a call to the first object as it is for the 1000th object.

5.2.4.5 Operating Environment Rationale

The OE is the infrastructure (hardware and software) upon which an SDR implementation is based. Wide-ranging, domain-specific performance requirements and constraints influence the selection of OE components. The selection of specific OE components is the prerogative of each SDR design. The SDR Forum encourages the selection of OE components that support open, industry standards and are available commercially. At a minimum, the OE must support the CF and software applications that are hosted upon it. Further considerations on the desirable attributes of the OE are discussed in Section 2.2.4 Structural View.

5.2.4.6 Core Framework Rationale

The CF represents a baseline that has been evolved from the SDR Forum's functional reference model. Software modularity, portability, and dynamic instantiation are crucial goals of an SDR implementation. When a radio powers up, the radio does not know what hardware or software is available or needed. The core components of the CF are the *DomainManager*, *ResourceManager* and software *Resource*. This triad provides the ability to investigate the capabilities of a radio domain and implement the requested functionality.

An application in a distributed environment can consist of many different software components (*Resources*). These components can be objects, processes, and/or threads on many different processors. The CORBA *Resource* interface is specific enough to exert the required control but generic enough to support many different applications.

A *ResourceManager* has two purposes. A *ResourceManager* provides the *DomainManager* with a property list of the hardware devices and software resource co-located with the *ResourceManager*. The *ResourceManager* also oversees the operation of the co-located software *Resources*.

The *DomainManager* keeps track of the Domain Profile. The Domain Profile keeps track of the device properties and software resources on the radio domain. The *DomainManager* uses the *ResourceManager* and other CF components to distribute and connect software *Resources*.

The member companies of the SDRF mobile working group have jointly agreed on the architecture. Further industry support is being solicited through other SDRF working groups. Eventually the architecture will be presented to the OMG as a proposed CORBAfacility.

The CF is being offered as an open, non-proprietary architecture. Eventually the CF will be available from multiple vendors. The entire purpose of the CF is to bring up a distributed application in a controlled and secure manner. Because of the use of CORBA and a standardized processor environment the CF supported components should be able to port between different processors, RTOSs, buses and ORBs. The CF is lightweight, and the number of interfaces and operations required by the CF has been kept to a minimum. The CF scales to all SDR domains (mobile, base-station, satellite, and handheld). All SDR domains have the same goals for installing/upgrading multiple

applications, and dynamically allocating these resources to physical assets and linking them to other software *Resources*. The linking of *Resources* to other *Resources* occurs within a processor and across a bus for all the SDR domains. CORBA vendors code (ORB core, and the size of generated stubs and skeletons code form the IDL compiler) are scalable. Some ORBs currently have small footprints around 20 kbytes. The use of CORBA and POSIX provide a basis for security and safety.

5.2.5 Functional View

5.2.5.1 SDR Software Reference Model

The SDR software reference model is depicted in figure 5.2.5.1-1. The key points about the software reference model are:

1. It serves as the basis for defining the functional view of the SDR Software Architecture
2. It broadly introduces the various functional roles performed by SDR software entities without dictating a structural model of these elements
3. It broadly introduces the control and traffic data interfaces between the functional software entities
4. It introduces the color coding of each functional entity used throughout the definition of the SDR Software Architecture.

This functional software reference model is limited in that it cannot form the basis of a distributed object-oriented software architecture. For example, the networking and waveform functions performed by Black-Side Processing and Red-Side Internetworking entities will be, in some cases, completely different functions. In other cases, they will be completely identical functions. A software architecture that attempts to "force fit" these functions into one place will not provide the flexibility demanded by the various SDR domains, nor the reusability desired by software radio vendors and operators. The SDR Forum uses the functional model as a point of transition (or "evolution") into the distributed object-oriented architecture defined by the other three software views (i.e., structural, logical, and use case).

5.2.5.2 Transition from Functional Model to OO Model

SDR software applications will perform user communication functions that include modem-level digital signal processing, link-level protocol processing, network-level protocol processing, internetwork routing, external access, security, and embedded utility behavior. These are user-oriented or "non-core applications", i.e., applications that are not part of the Core Framework (CF). The term "non-core" is not meant to imply that these applications are not important. Without them, a radio does not perform any useful function, so they are extremely important. A conceptual model of SDR non-core applications is depicted in Figure 5.2.5.2-1.

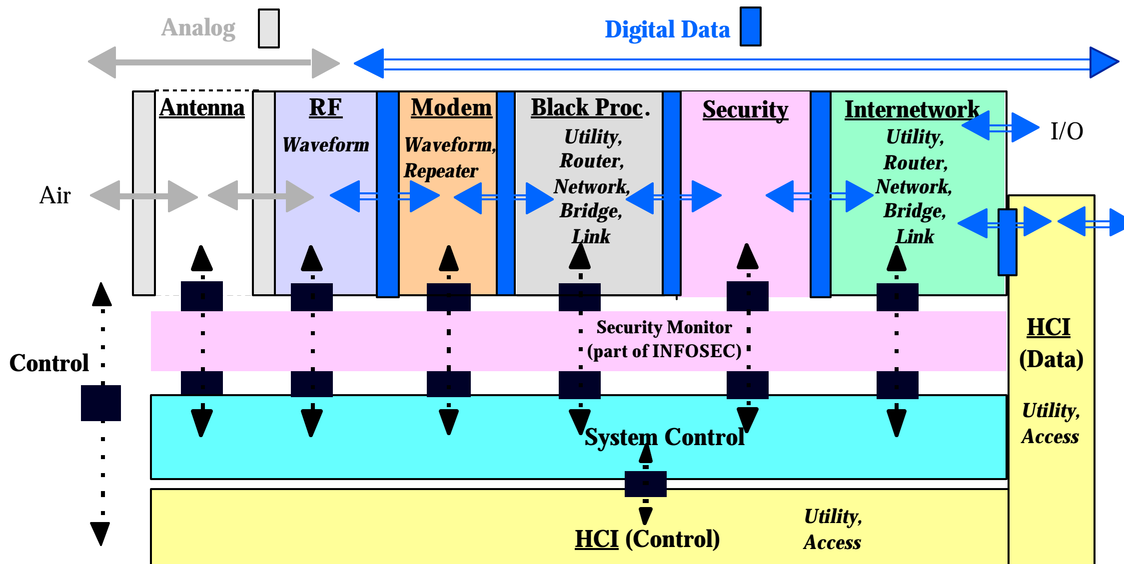


Figure 5.2.5.1-1 SDR Software Reference Model

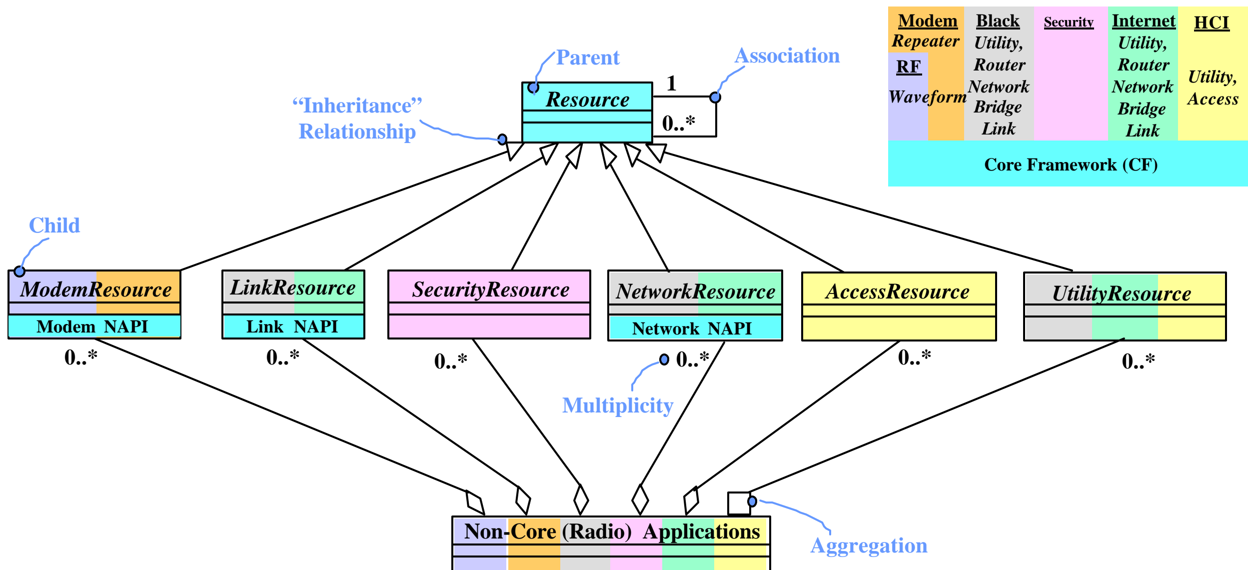


Figure 5.2.5.2-1 Conceptual Model of SDR Non-Core Applications

To help transition from the Functional View to the object-oriented views described in the sections that follow, it is useful to consider non-core applications not as functions but as resources that can inherit common types of behavior and common types of interfaces. The conceptual model of a resource is depicted in Figure 5.2.5.2-2. Notice that a resource encapsulates base class interfaces that support the establishment (registration) of message paths (or "circuits") between resources, provide a "pipe" for message communication between resources, and provide standard methods of managing the states of resources.

Subclasses of a resource extend the base class interfaces to more specific types of resources that implement the non-core application behaviors or "functions". For example, Networking Application Program Interfaces (NAPIs) are the means by which the base class interfaces may be extended to overlay the SDR Software Architecture onto an embedded networking architecture.

The types of *Resources* that are created within a radio domain include but are not limited to:

- Modem Resource – This resource extends the basic resource definition by adding the physical interfaces that are common to all modem devices.
- Link Resource – This resource extends the basic resource definition by adding the link layer interfaces. The Link Resource can be implemented on both sides of the Security boundary as depicted by the Link Resource color in Figure 5.2.5.2-2.
- Network Resource – This resource extends the basic resource definition by adding the network layer interfaces. The Network Resource can be implemented on both sides of the Security boundary as depicted by the Network Resource color in Figure 5.2.5.2-2.
- Access Resource – This resource extends the basic resource definition by adding a set of multi-media resources such as audio, video, serial, Global Positioning System (GPS), and Ethernet. These resources contain the device drivers and the protocol.
- Security Resource – This resource extends the basic resource definition by adding embedded security services such as encryption, decryption, authentication, key management, or other security features.
- Utility Resource – This resource extends the basic resource definition by adding embedded application "utilities" such as situational awareness, message translation, network gateway, and host adapter services.

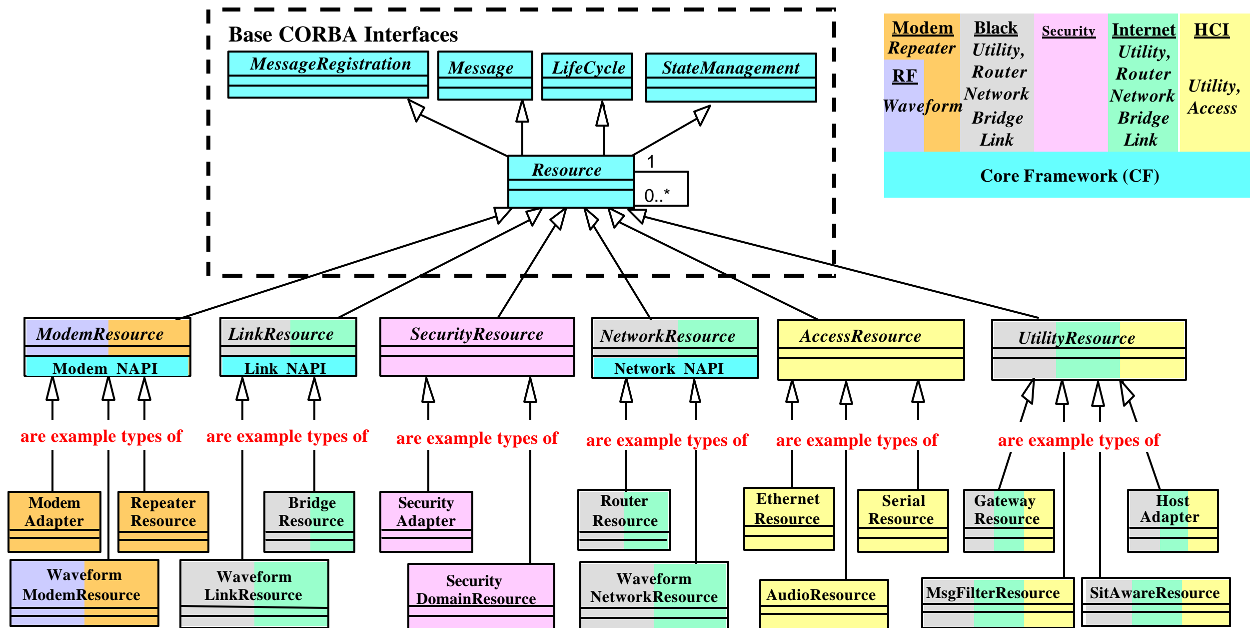


Figure 5.2.5.2-2. Conceptual Model of SDR Resources

Each specialized resource may also be extended by a Waveform resource by adding more functionality as necessary. Each resource can be associated with zero or more other resources. The implementation of a resource determines the relationships it will have with other resources within the radio domain. Non-core applications will provide internal behavior to implement specific waveform, networking, security, user access, and other embedded utility "functionality. This internal behavior is not dictated by the CF. Rather, the CF allows all non-core applications to be derived from the base *Resource* class. Where the hardware or security architecture in a given implementation may prevent a non-core application from being implemented as a *Resource* class CORBA software object, e.g., a time-critical DSP function, or an embedded COMSEC chip control function, the interface to this application will be through a *Resource* class object. Discussions of encapsulated resources and adapter resources are provided in section 0.

5.2.5.2.1 System Control Functions

Core applications, which are a part of the CF, support the non-core applications by providing the necessary function of control as well as standard interface definitions that the non-core applications use to ensure plug and play, ready to execute modularity. This allows industry-wide development of non-core applications to a common standard framework.

Elements of the CF provide the system control functions for managing hardware assets, installing, creating, and managing software resources, managing files, and providing run-time services. These elements are depicted in Figure 5.2.5.2.1-1 and are described in further detail in Section 0 - Logical View.

5.2.5.2.2 Modem Resource Functionality

A conceptual model of SDR Modem Resources is shown in Figure 5.2.5.2.2-1. The high diversity of digital signal processing solutions, both in hardware and software, requires the SDR Software Architecture to be flexible in accommodating a wide range of implementations. From the software architecture perspective, a standard for the control and interface of a modem layer, which encapsulates diverse implementations of smart antenna, RF, and modem functions is a critical concept. The SDR base class interfaces are extended to modem resources through the *Modem NAPI*, which provides a standard interface for control and communication with modem layer operations from a higher (link layer) resource. The inclusion of a modem adapter resource in the architecture provides a transparent gateway for those implementations in which a CORBA capable link resource is communicating with a non-CORBA capable modem resource. The modem adapter provides the translation between the *Modem NAPI* and the API set of the non-CORBA capable modem resource. Using the *Modem NAPI*, the link resource is isolated from this translation. The modem adapter is thus transparent to the link resource, which greatly enhances the reusability of the link resource with multiple modem implementations.

The operations performed by the modem resources will vary depending on waveform requirements as well as hardware/software allocation and are not dictated by the CF. Typical RF and modem operations are depicted within the example subclasses.

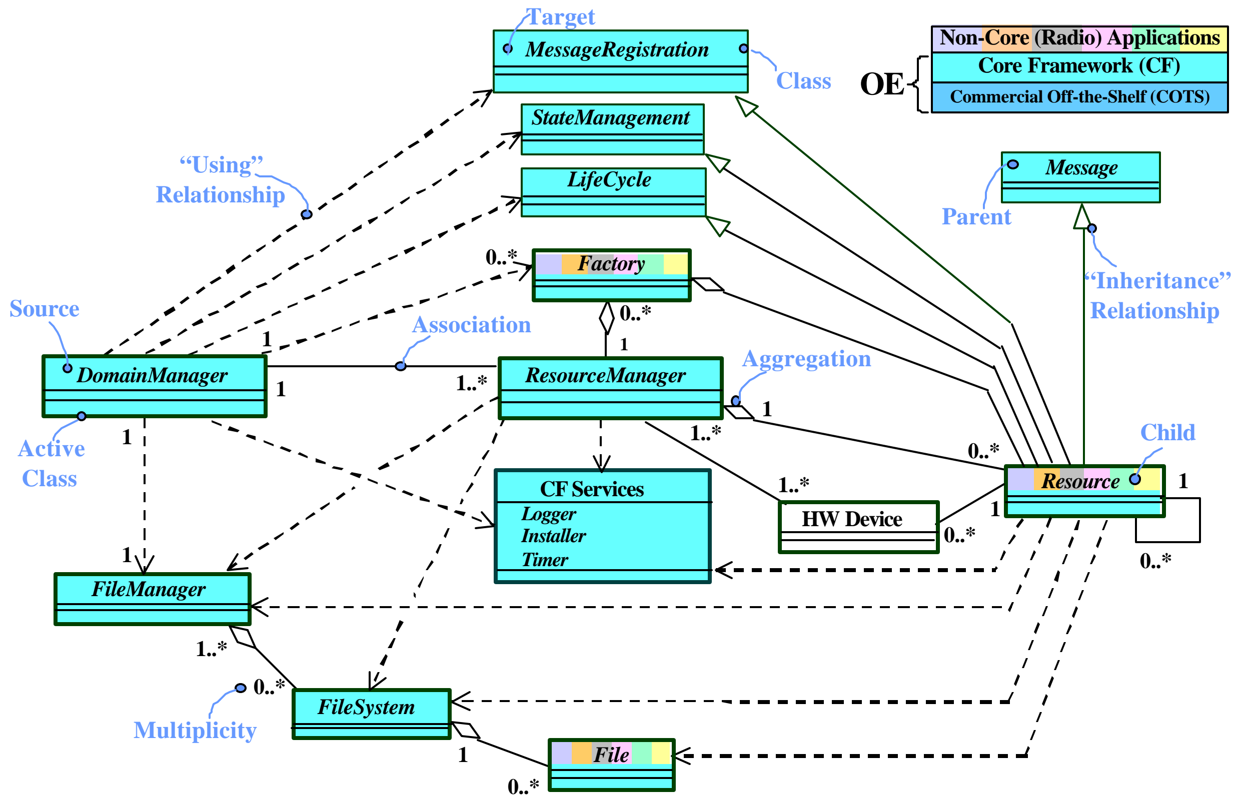


Figure 5.2.5.2.1-1 Conceptual Model of the Core Framework (CF)

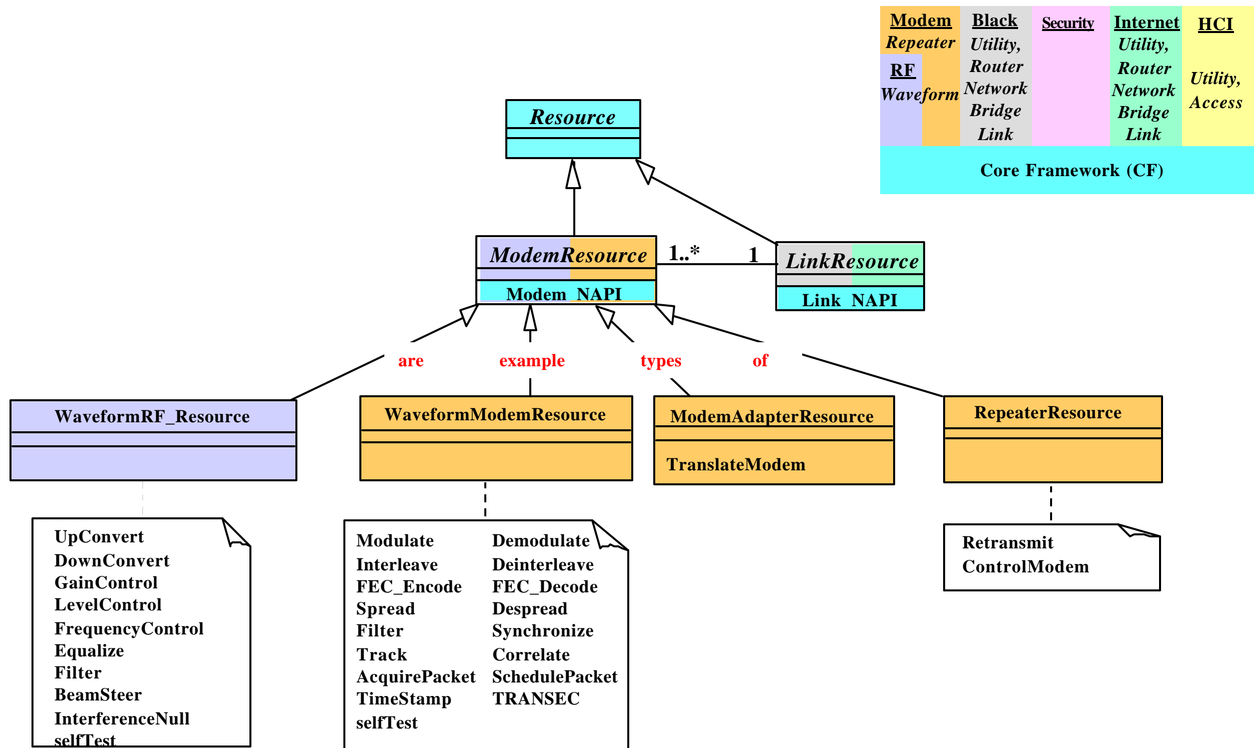


Figure 5.2.5.2.2-1 Conceptual Model of SDR Modem Resources

5.2.5.2.3 Networking Resource Functionality

A conceptual model of SDR networking resources is shown in Figure 5.2.5.2.3-1. The CF base class interfaces are extended to link layer and network layer resources through the *Link NAPI* and *Network NAPI*, which provide a standard interface for control and communication between modem, link, and network layer resources.

The operations performed by the waveform networking and internetworking resources will vary depending on waveform requirements as well as networking requirements and are not dictated by the CF. Resources that provide networking behavior including repeater, link, bridge, network, router, and gateway operations are depicted within the example subclasses.

Note that the Networking Resources may implement wireless IP routing external to the radio and should not be confused with underlying IP stacks that support interprocessor communication within a radio.

5.2.5.2.4 Access Resource Functionality

A conceptual model of SDR access resources is shown in Figure 5.2.5.2.4-1. An access resource provides access to radio hardware devices and external physical interfaces. The operations performed by an access resource will vary depending on the radio hardware assets as well as the physical interfaces to be supported and are not dictated by the CF. Typical access operations are depicted within the example subclasses.

5.2.5.2.5 Security Resource Functionality

A conceptual model of SDR security resources is shown in Figure 5.2.5.2.5-1. Typical security operations are depicted within the example subclasses. The high diversity of security solutions, both in hardware and software, requires the SDR Software Architecture to be flexible in accommodating a wide range of implementations. Transmission security and communications security requirements vary between waveforms. The location of the security boundary with respect to networking requirements also varies between waveforms. A security resource must also provide key fill, key management, programmable security device control and interface, and software integrity and authentication services. The CF base class interfaces are extended by the security resource to provide specific security services within each type of security domain implementation. The inclusion of a security adapter resource in the architecture provides a transparent gateway for those implementations in which other CORBA capable resources, e.g., modem, link, network, and utility resources, are communicating with a non-CORBA capable security resource. The security adpater provides the translation between these CORBA capable resources and the API set of a non-CORBA capable security resource. This isolates these other resources from the security API translation. The security adapter is thus transparent to these other resources, which greatly enhances the reusability of these resources with multiple security implementations. The CF base class interfaces can be extended to include a standard set of security interface functionality for security domains. This would allow the resources to be more plug and play across implementation domains.

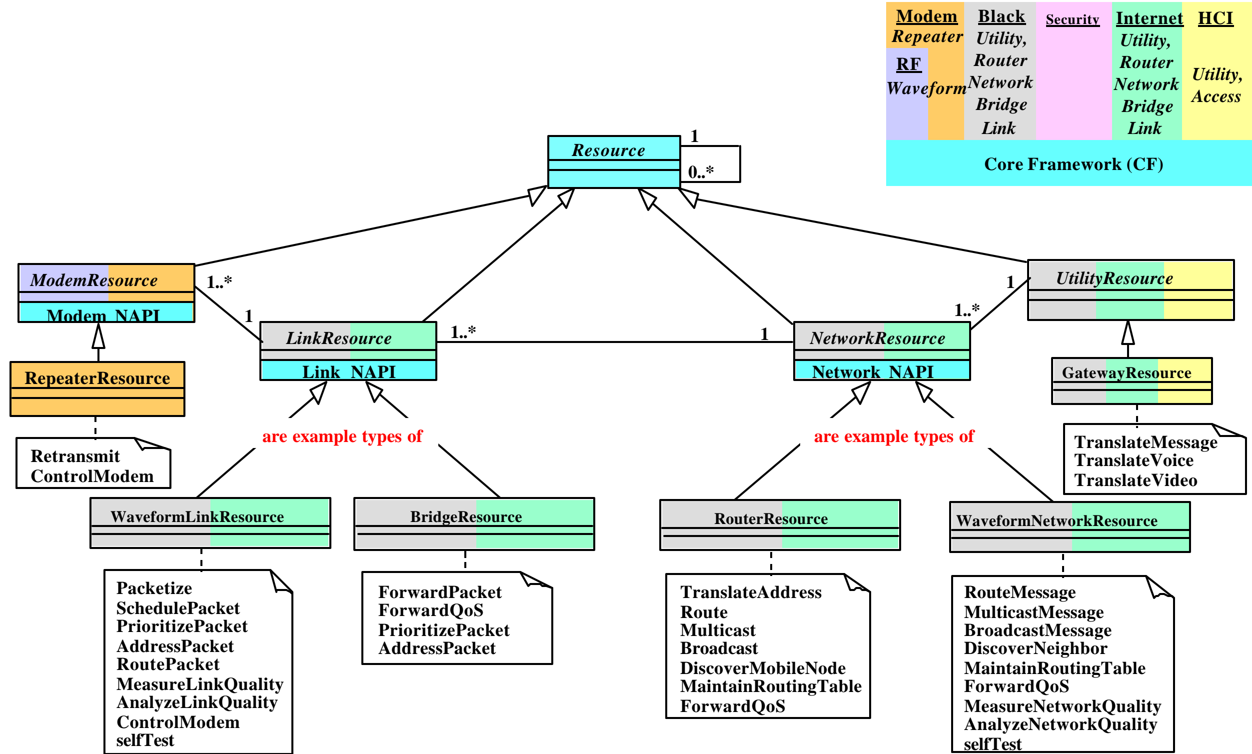


Figure 5.2.5.2.3-1 Conceptual Model of SDR Networking Resources

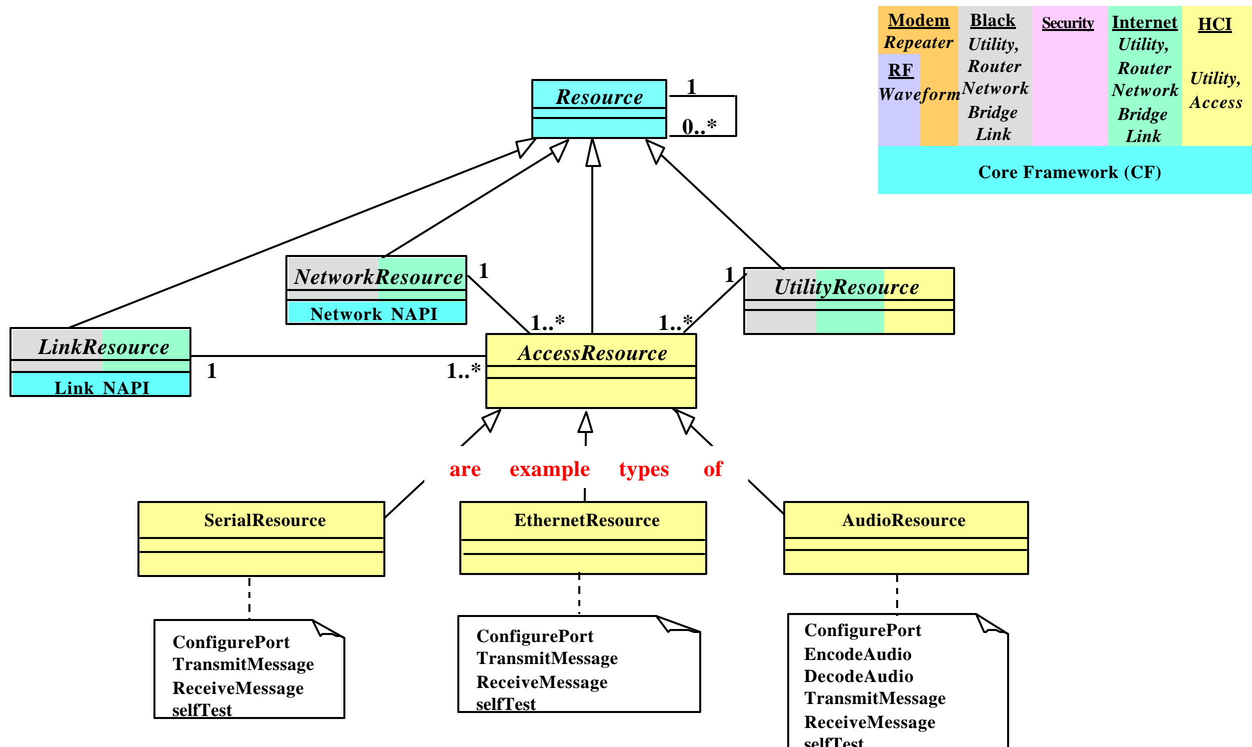


Figure 5.2.5.2.4-1 Conceptual Model of SDR Access Resources

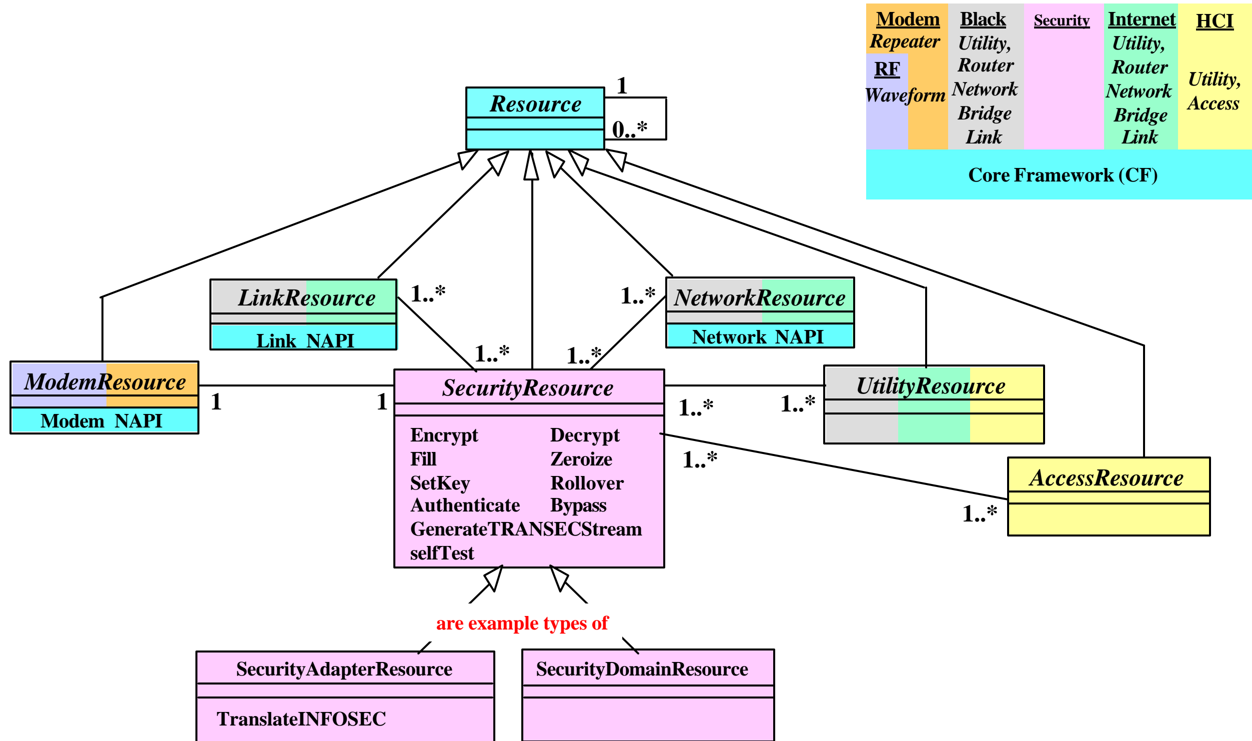


Figure 5.2.5.2.5-1 Conceptual Model of SDR Security Resources

5.2.5.2.6 Utility Resource Functionality

A conceptual model of SDR utility resources is shown in Figure 5.2.5.2.6-1. The operations performed by the utility resources will vary depending on the embedded applications to be supported as well as host interface protocol requirements and are not dictated by the CF. Typical utility operations are depicted within the example subclasses.

The wide range of host system protocols and interfaces that are encountered in SDR implementations requires the SDR Software Architecture to provide support for both CORBA-capable and non-CORBA- capable host system implementations. Many of the CF defined interfaces are designed to be extended "outside the box," for use by CORBA-capable host systems. Where legacy or non-CORBA-capable host systems prevent the direct use of the CF interfaces, the SDR architecture includes a host adapter resource to provide a transparent gateway between the CF interfaces and the non-CORBA-capable host system. The host adapter provides the translation between the CORBA-capable SDR resources and the API set of a non-CORBA-capable host system. This isolates the SDR resources from the Host API translation. The host adapter is thus transparent to the SDR resources, which greatly enhances the reusability of these resources with multiple Host system implementations.

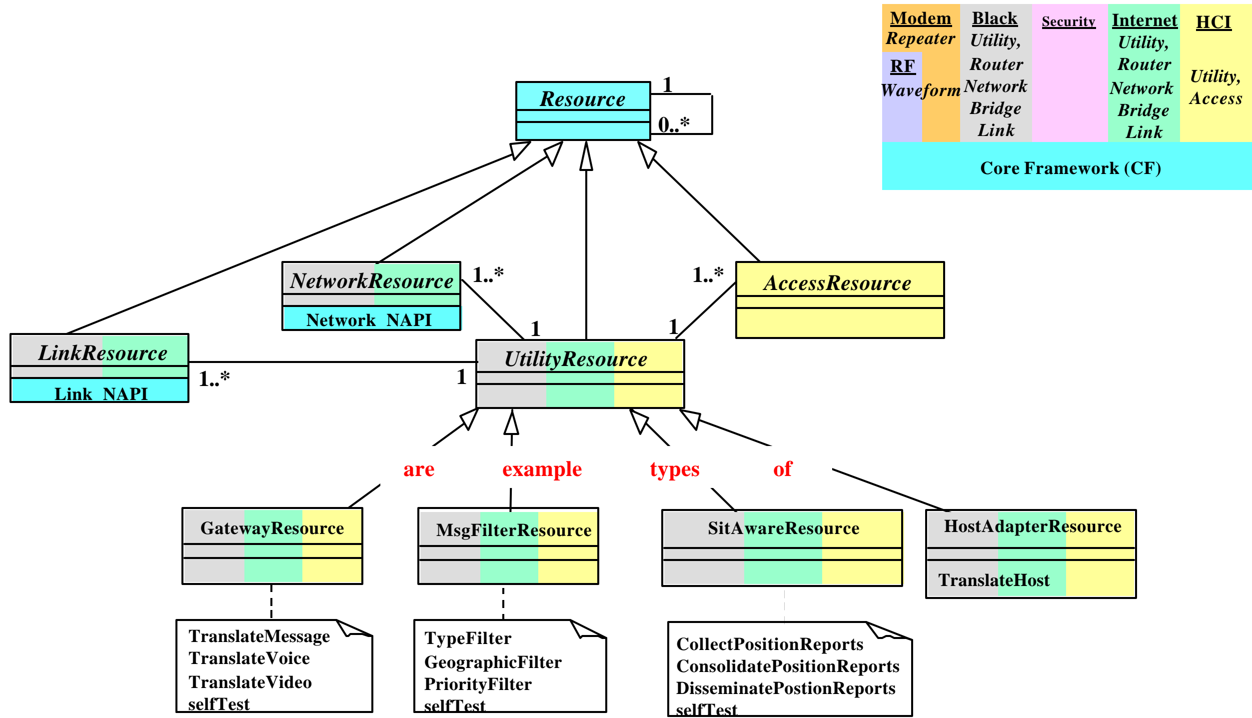


Figure 5.2.5.2.6-1 Conceptual Model of SDR Utility Resources

5.2.6 Structural View

5.2.6.1 Open Multi-layered Structural Architecture

The SDR software structural architecture is shown in Figure 5.2.6.1-1. The structural architecture view provides the best graphical depiction of the OE. The key aspects of the structural architecture are:

1. Maximizes the use of commercial protocols and products,
2. Isolates both core and non-core applications from the underlying hardware through multiple layers of open, commercial software infrastructure, and
3. Provides for a distributed processing environment through the use of CORBA to provide software application portability, reusability, and scalability.

SDR Operating Environment requirements needed to support implementations requiring multiple levels of message security have not yet been established.

5.2.6.1.1 Bus Layer (Board Support Package)

The SDR Software Architecture is capable of operating on most commercial bus architectures. The OE relies on reliable transport mechanisms, which may include error checking and correction at the bus support level. This allows support for VME, PCI, CompactPCI, Firewire (IEEE-1394), Ethernet, and others. The OE does not preclude the use of different bus architectures on the Red and Black subsystems. The choice of bus architecture is driven by the bandwidth and latency requirements of the non-core applications. The core applications and CORBA ORB should be considered, but should not impact the decision.

5.2.6.1.2 Network Stacks & Serial Interface Services

The SDR Software Architecture relies on commercial components to support multiple unique serial and network interfaces. The OE relies on reliable transport mechanisms, which may include error checking and correction at the network and serial interface level. These interfaces can be selected to provide the interfaces necessary to support the platform implementation. Possible serial and network physical interfaces include: RS232, RS422, RS423, RS485, Ethernet, 802.x, and others.

To support these interfaces, various low-level network protocols may be used. They may include PPP, SLIP, CSLIP, LAPx, and others. Using these protocols, other protocols such as IP, TCP/UDP, and X.25 can be added to provide network connectivity. The standard transport mechanism for non-colocated calls using CORBA 2.2 is GIOP on top of TCP/IP. Protocols to support ancillary functionality, e.g., neighbor discovery, address resolution; etc., may also be used.

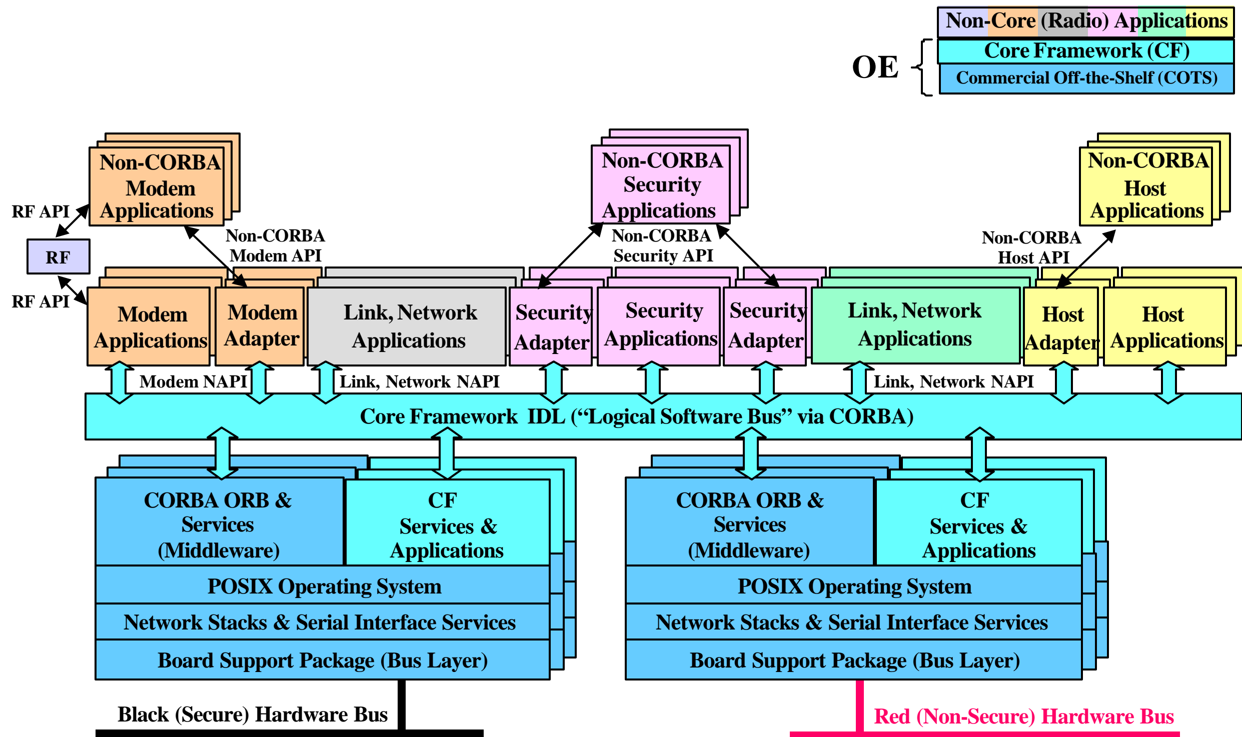


Figure 5.2.6.1-1. SDR Software Structure

5.2.6.1.3 Operating System Layer

The SDR Software Architecture relies on a real-time embedded operating system to provide multi-process, multi-threaded support for core applications (CF applications), as well as non-core waveform and networking applications. A COTS solution is desirable, as well as a standard operating system interface for operating system services in order to facilitate portability of both core and non-core applications.

POSIX is an accepted industry standard that was developed for larger systems as a result of UNIX distributors developing their own flavors of UNIX. POSIX and the real-time extensions are compatible with the requirements to support the OMG's RT CORBA specification. Complete POSIX compliance encompasses more features than are necessary to control a typical SDR implementation. Even though modern systems are more compact, have greater memory and processing speeds, it is still felt that a more streamlined version of POSIX is necessary for the embedded real-time SDR market.

The SDR Software Architecture recommends the use of the POSIX 1003.13 API set. Four profiles have been defined to reflect the wide range of system requirements presented by real-time designs. These four profiles are:

a) **Minimal Real-time Systems Profile (PSE51)** are typically embedded and dedicated to the control of one or more special I/O devices without operator intervention. The hardware model for this profile is a single processor with memory but no memory management support.

b) **Real-time Controller System Profile (PSE52)** is an extension of the Minimal Real-time System Profile, and supports file system interface and asynchronous I/O.

c) **Dedicated Real-time System Profile (PSE53)** is an extension of the Minimal Real-time System Profile, and adds support for multiple processes, a common interface for device drivers and files, and memory locking for management. The hardware model for this profile is one or more processors with or without hardware MMU support.

d) **Multipurpose Real-time System Profile (PSE54)** which combines the functionality of the other three and provides a comprehensive list of functionality, including all of POSIX.1, POSIX.1b, and POSIX.1c. This profile also includes support for POSIX.2 and POSIX.2a. The hardware model for this profile is one or more processors with memory management units, high speed storage devices, special interfaces, network support and display devices.

A minimum requirement that would include one of these profiles or some combination of these profiles for the SDR Operating Environment has not yet been established. The inherent desire to select the fully featured Multipurpose Real-time System Profile (PSE54) would be prohibitive on the resources available in handheld units.

5.2.6.1.3.1 Memory Management

Memory Management is useful in SDR implementations. Not only will memory management be needed to achieve multiple security levels, but it is useful in the development environment as well. During development, common coding errors such as stray pointers and indexing beyond array boundaries can result in one process accidentally overwriting the data space of another. This can cause many wasted

hours of debugging to determine the cause. In an multi-level secure system, memory management protection prevents other processes or threads from reading memory that is outside its allocated area, thus preventing an unclassified process from gaining access to classified data. If a process attempts to access memory that is explicitly declared or allocated for the type of access attempted, the memory management unit (MMU) hardware will notify the operating system (OS) kernel, which can then abort the process immediately at the offending program statement. The kernel can then log information about the process that has caused the access violation for later review. This protects processes from each other, prevents coding errors from damaging memory used by other processes, protects the kernel memory, and supports the SDR security architecture.

The SDR Software Architecture recommends the use of processors equipped with MMUs wherever the COTS OS and CORBA middleware services are hosted.

5.2.6.1.3.2 File System Drivers

An SDR implementation may host many waveforms. These different waveform applications will need to be added, read and distributed, replaced, and removed from a mass storage device. Rotating magnetic media are vulnerable to shock, vibration and acceleration, but could continue to be used in semi-permanent or permanent installations. For mobile platforms solid-state disks (SSD) are the best solution.

Solid-state disks are random access, high-speed storage peripherals that use memory chips such as Flash, EPROM, or SRAM. SSDs give faster and more efficient operation, a longer life span, and a lower risk of breakdown or data loss than their magnetic cousins. Many SSDs incorporate industry standards so the interface is compatible with most operating systems. Also, many of these SSDs follow industry standards for shock and vibration. These characteristics make SSDs an ideal solution for harsh environmental and mission critical applications where availability and reliability is important.

While the SDR Software Architecture does not require the use of SSDs, they are ideally suited to the rugged environments. The SDR Software Architecture recommends the use of POSIX 1003.13 compliant file system services. These POSIX services supported by the OS are extended by the CORBA-based CF definitions of *FileManager*, *FileSystem* and *File* Interfaces for remote and/or distributed Network File System (NFS) type of file access.

5.2.6.1.4 CORBA Middleware

CORBA is a cross-platform framework that can be used to standardize client/server operations when using distributed processing. Distributed processing is a fundamental aspect of the SDR system architecture and CORBA is the mostly widely used “middleware” service for providing distributed processing. A summary of the features and benefits of CORBA are contained in the following paragraphs.

The idea behind the CORBA framework is to replace traditional message passing. As much as possible the CORBA architecture tries to make the exchange of messages look to the client and server software applications like a normal, local function invocation. The CORBA protocol code handles the bit packing and handshaking required for delivering the message.

The IDL is an object-oriented language used to define the interface between a client and a server. This interface definition acts as a **contract** between the client and the server applications. An interface is defined as a collection of methods and object attributes. Methods are also known as messages in

object-speak and correspond to functional operations, function calls and procedure calls. All CF interfaces are defined in IDL, and thus serve as a contract between the SDR applications that use them. The features and benefits of CORBA, including its significant technical advantages over earlier distributed processing techniques, the maturity of the OMG specifications, the ability to define interfaces in IDL, the wide commercial availability of CORBA products, and the wide industry acceptance of CORBA make it ideally suited to the SDR Software Architecture.

5.2.6.1.5 Application Layer

SDR software applications will perform user communication functions that include modem-level digital signal processing, link-level protocol processing, network-level protocol processing, internetwork routing, external access, security, and embedded utility behavior. These are user-oriented or non-core applications, i.e., applications that are not part of the CF. Core applications, which are a part of the CF, support the non-core applications by providing the necessary function of control as well as standard interface definitions that the non-core applications can use. This allows industry-wide development of non-core applications to a common standard framework. Section 0 - Logical View will show the relationship between the CF and non-core applications.

5.2.6.1.5.1 Core Framework

The CF enables the development and use of SDR software applications in a distributed, plug and play context. The CF consists of the following interfaces, core applications, and core services:

- Base CORBA interfaces (*Message*, *MessageRegistration*, *StateManagement*, and *Resource*) that are inherited by core and non-core software applications
- Core applications (*DomainManager* and *ResourceManager*) that provide framework control of resources
- Core services that support both core and non-core applications (*Logger*, *Installer*, *Timer*, *FileManager*, *FileSystem*, and *File*)
- An optional core *Factory* interface for controlling the life span of core and non-core applications.

5.2.6.1.5.2 Non-Core Applications

Non-core applications consist of one or more resources. These resources implement the *Resource* interface or *NAPI* interfaces. The application developers can extend these definitions by creating specialized *Resource* interfaces for the application. At a minimum, the extension has to come from the *Resource* interface. Through the use of the CF, a developer can more easily reuse software developed for an SDR implementation and reduce the NRE to produce new capability. Currently, the SDR Software Architecture defines the following types of non-core applications or resources, but does not preclude the definition of other types:

1. Modem Resource
2. Link Resource
3. Network Resource

4. Access Resource
5. Security Resource
6. Utility Resource.

The internal behavior of a resource is not dictated by the SDR Software Architecture. This is left to the application developer. The interfaces by which a resource is controlled and communicates with other resources are defined by the SDR interfaces and are described in the following section.

5.2.7 Logical View

This section contains the detailed description of the CF interfaces and operations. This includes a detailed description of the purpose of the interface, the purpose of each supported operation within the interface, the IDL for each operation, and interface class diagrams to support these descriptions.

5.2.7.1 Core Framework

Figure 5.2.7.1-1 depicts the key elements of the CF and the relationships between these elements. A *DomainManager* object manages the software *Resources* and hardware assets within the radio. Some of the software *Resources* may directly control the radio's internal hardware assets or interface devices. For example, a *ModemResource* may provide direct control of a modem hardware device such as an FPGA or an ASIC. An *AccessResource* may operate as a device driver to provide external access to the radio. Other software *Resources* have no direct relationship with a hardware device, but perform application services for the user. For example, a *NetworkResource* may perform a network layer function. A *WaveformLinkResource* may perform a waveform specific link layer service. Each *Resource* can potentially communicate with other *Resources*. These *Resources* are allocated to one or more *ResourceManager* objects by the *DomainManager* object based upon various factors including the hardware devices that the *ResourceManager* knows about, the current availability of hardware devices, the behavior rules of a *Resource*, and the loading requirements of the *Resource*.

The *Resources* being managed by the *DomainManager* object are CORBA objects implementing the *Resource* interface. Some *Resources* may be dependent on other *Resources*. This interface provides a consistent way of creating up and tearing down any *Resource* within the radio. These resources can be created by using a *Factory* interface or by the *ResourceManager* interface.

The file services: *FileManager, FileSystem,* and *File* are the interfaces that are used to support installation and removal of application files within the radio, and for loading and unloading application files on the various processors that the *ResourceManagers* execute upon.

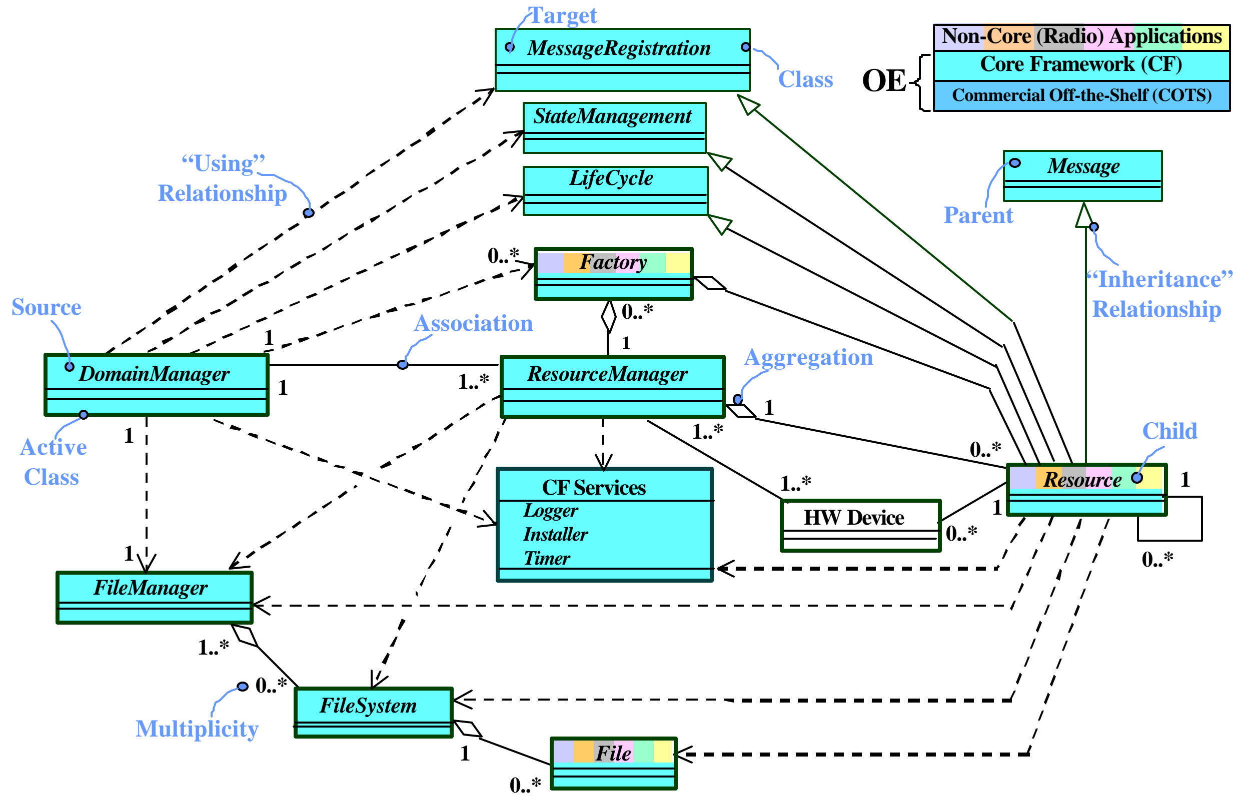


Figure 5.2.7.1-1. SDR Core Framework (CF) Relationships

5.2.7.1.1 Base Class Interfaces

All *Resources* inherit and encapsulate the base CF interfaces of:

- *LifeCycle* – This interface provides operations for managing the start-up and tear-down states of a *Resource*, configuring the properties of a *Resource*, and invoking *Resource* self-test.
- *StateManagement* – This interface provides operations to set and retrieve the Administrative state of a *Resource* and to retrieve the Operational and Usage states of a *Resource*. These states are derived from the ISO/IEC 10164-2 Open Systems Interconnection - Systems Management: State Management Function.
- *MessageRegistration* – This interface provides operations for the creation of virtual circuits, i.e., messaging pathways, among source (producer) and sink (consumer) *Resources* within the radio.
- *Message* – This interface provides messaging operations implemented by a sink (consumer) *Resource* and issued by a source (producer) *Resource*. Messaging operations are provided for all of the CORBA basic types.

5.2.7.1.1.1 LifeCycle

The *LifeCycle* interface defines the generic object operations for:

- Testing.
- Configuring (setting) and querying (retrieving) an object's properties. The parameter type for properties is based upon the CORBA **any** type. This provides the greatest flexibility for developing software by leaving the definition of an object's properties up to the developer not by the core framework definition. The CORBA **any** type is also minimum CORBA compliant.
- Initializing and releasing an object.
- Message processing control operations: start, stop, and pause.

The *DomainManager* object uses these interfaces to start up and tear down resources in a determinate and consistent manner within the radio. The *DomainManager* object performs selfTest, configure, initialize, and start operations for each resource it is responsible for starting up.

LifeCycle Relationships

The definition of the *LifeCycle* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.1.1-1.

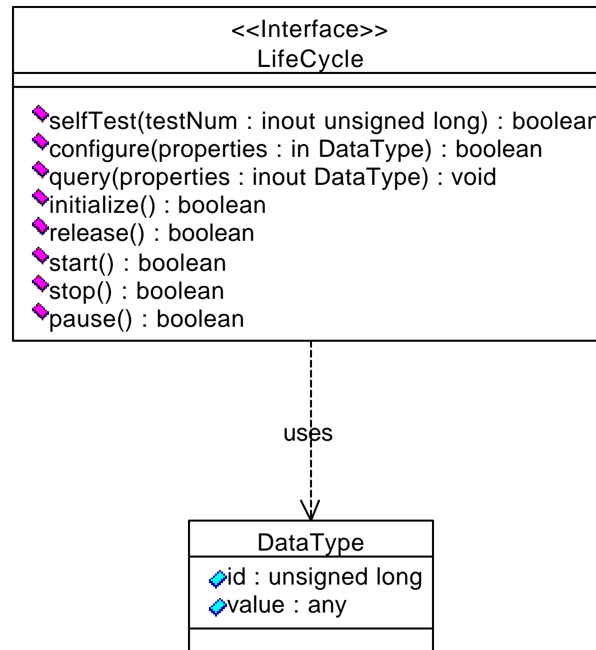


Figure 5.2.7.1.1.1-1. *LifeCycle* Relationships

LifeCycle Interfaces

The IDL for the *LifeCycle* interface produced from the Rational Rose diagrams in Figure is shown below:

interface LifeCycle {

The selfTest operation performs a specific test on an object. True is returned if the test passes, otherwise false is returned. When false is returned, the operation also returns a reason why the test failed.

boolean selfTest(inout unsigned long testNum);

The configure operation sets the object's properties. True is returned if the configure was successful, otherwise False is returned. Any basic CORBA type or static IDL type could be used for the configuration data. An object's ICD indicates the valid configuration values.

boolean configure(in DataType properties);

The query operation retrieves object's properties. Any basic CORBA type or static IDL type could be used for the query. An object's ICD indicates the valid query types. The information retrieved can later be used when an object is recreated, by calling the configure operation.

void query(inout DataType properties);

The initialize operation controls when configuration data is implemented by the resource or initializes the devices being controlled by the resource.

boolean initialize();

The release operation releases itself from the CORBA ORB. When the object's ORB reference count goes go to zero, the object destructor operation will be called.

boolean release();

The start operation starts processing messages that are received from the front end and/or back end of the radio. The object's sink (consumer) objects are enabled for processing messages.

boolean start();

The stop operation stops processing messages that are received from the front end and/or back end of the radio. The object's sink (consumer) objects are disabled from processing messages and the messages are discarded.

boolean stop();

The pause operation queues messages that are received from the front end and/or back end of the radio.

boolean pause();

};

5.2.7.1.1.2 StateManagement

The *StateManagement* interface defines the generic object operations for:

- Retrieving and setting system management state information. The state information is based upon the ISO/IEC 10164-2 Open Systems Interconnection - Systems Management: State Management Function standard. The Administrative, Operational, and Usage states are included in the *StateManagement* interface. The ISO standard identifies additional states that could be used to expand the definition of managed *Resource* states.

The *DomainManager* object uses the *StateManagement* interface to provide the radio operator, administrator, or maintainer with fundamental control and access to system managed states. The purpose of the Administration, Operational, and Usage states are intended for use by the *DomainManager* for high-level, user-oriented, control and system management of an SDR. They allow a system operator to determine the health and status of the SDR. The states are not intended to provide low-level, real-time state management between *Resources.* The following definitions apply to these states.

Administrative State attributes are defined as:

a) Locked = the managed resource is adminstratively prohibited from performing services for its users. This is a "settable" state.
b) Shutting Down = the managed resource is adminstratively permitted to existing instances of use but is otherwise locked to new use. This is a transitional state between Unlocked and Locked.
c) Unlocked = the managed resource is adminstratively permitted to perform services for its users. This is a "settable" state.
d) Admin Not Applicable = the managed resource is not subject to Administrative State control.

Operational State attributes are defined as:

a) Disabled = the managed resource is totally inoperable and unable to provide service to the user.
b) Enabled = the managed resource is partially or fully operable and available for use.

Usage State attributes are defined as:

a) Idle = the managed resource is inactive, i.e., not providing service to the user.
b) Active = the managed resource is actively providing service to the user.
c) Usage Not Applicable = the managed resource is not subject to Usage State reporting.

StateManagement Relationships

The definition of the *StateManagement* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.1.2-1

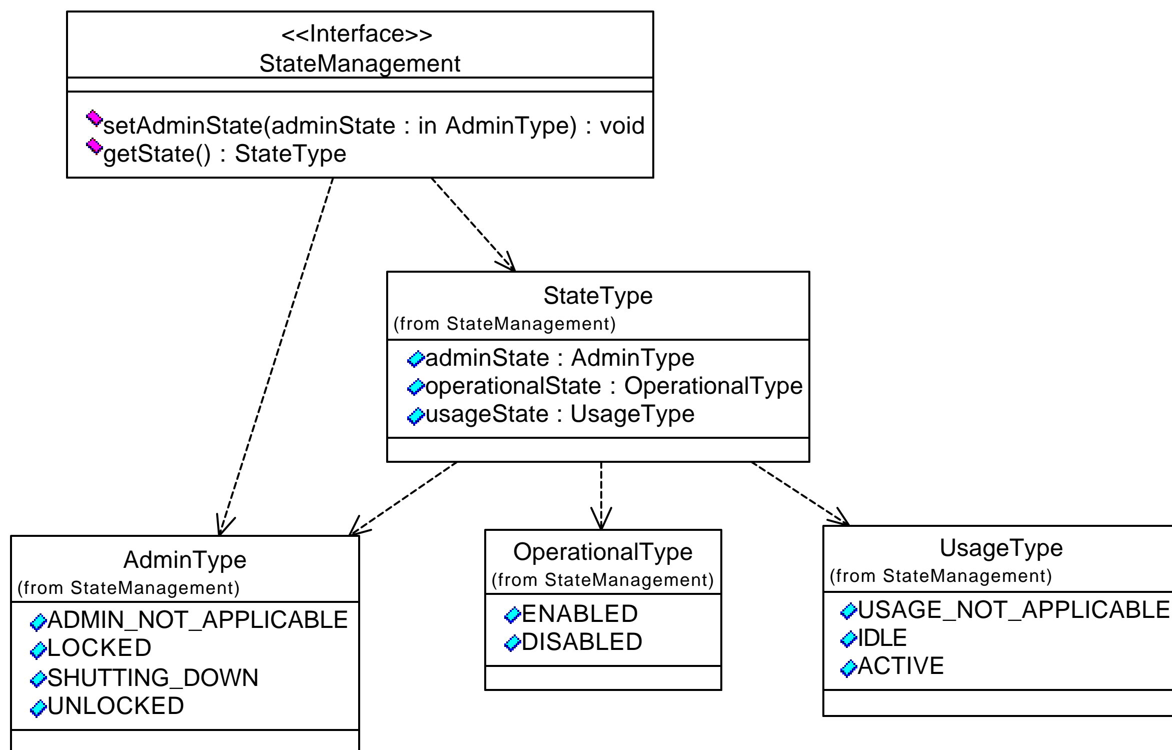


Figure 5.2.7.1.1.2-1 *StateManagement* Relationships

StateManagement Interfaces

The IDL for the *StateManagement* interface produced from the Rational Rose diagrams in Figure 5.2.7.1.1.2-1 is shown below:

interface StateManagement {

The following type is a CORBA IDL enumeratiuon type that defines an object's Administrative states.

```
enum AdminType
   {
    ADMIN_NOT_APPLICABLE,
```

```
      LOCKED,
      SHUTTING_DOWN,
      UNLOCKED
     };
```

The following type is a CORBA IDL enumeration type that defines an object's Operational states.

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

The following type is a CORBA IDL enumeration type that defines the object's Usage states.

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

The following type is a CORBA IDL struct type that contains an object's Admin, Operational, and Usage states.

```
   struct StateType {
      AdminType adminState;
      OperationalType operationalState;
      UsageType usageState;
     };
```

The setAdminState operation sets the Adminstrative State per the specified parameter (Locked or Unlocked).

```
     void setAdminState(in AdminType adminState);
```

The getState operation returns the object's state.

```
     StateType getState();
};
```

5.2.7.1.1.3 MessageRegistration

The *MessageRegistration* interface provides the operations for a Push Data Model and an Observer Design Pattern. The Push Data Model involves a Producer (source) and Consumer (sink), where the Producer pushes data to a Consumer. A Producer may know about a Consumer via the Observer Design Pattern that behaves as a callback where a consumer registers itself with producers for callback to it. The Observer Design Pattern is based on the industry accepted design pattern[2]. Alternatively, the DomainManager may establish the virtual path between a Consumer and Producer by providing the Producer with the Consumer *Resource* object reference. The outcome of using the

---

[2] "Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 293.

MessageRegistration interface is the set up of a dynamic virtual path between two resources within the radio. When an application is started up by the *DomainManager* within the radio, a set of virtual circuits are created among the application resources and other resources (Access, Security, Modem, etc.) as shown in Figure 5.2.7.1.1.3-1

Most of these Resources act as both a Consumer and Producer within the radio depending on the direction (from antenna or to the antenna) of data. The outcome of connecting these dynamic resources together is known as the Chain Of Responsibility design pattern, where each resource processes the data and pushes the data to another resource in the chain who has responsibility for further processing of the data. Only those *Resources* that have a need to process the data need to be included in the virtual path.

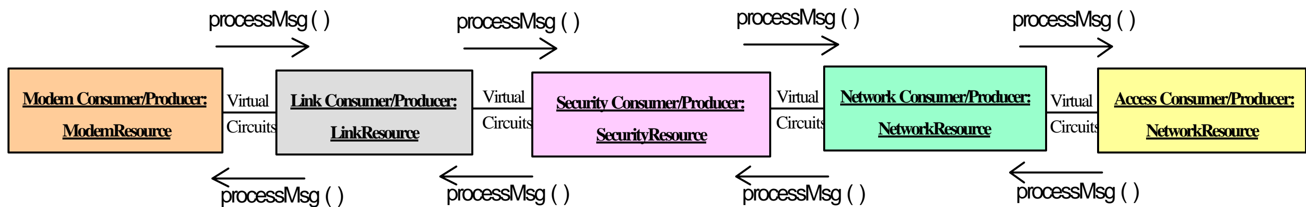


Figure 5.2.7.1.1.3-1 Example of Chained *Resources*

MessageRegistration Relationships

The definition of the *MessageRegistration* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.1.3-2.

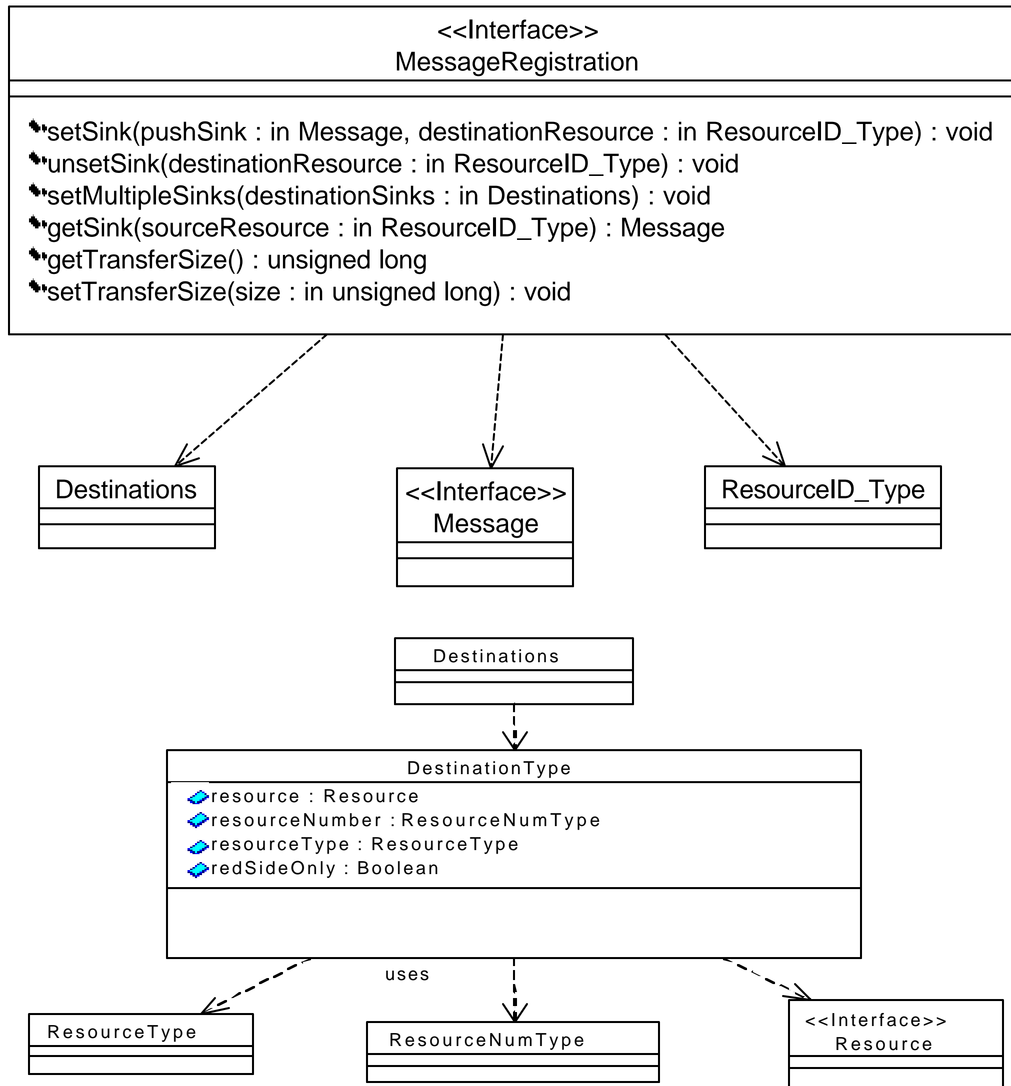


Figure 5.2.7.1.1.3-2. *MessageRegistration* Relationships

MessageRegistration Interfaces

The IDL for *MessageRegistration* interface produced from the Rational Rose diagrams in Figure 5.2.7.1.1.3-2 is shown below:

interface MessageRegistration {

The setSink operation registers a single *Message* sink (Consumer) object for call back by a source (Producer) object. The *Message* sink object reference is added to the source object's list of registered *Message* sinks. When pushing data to this destination the *Message* sink object is used.

void setSink(in Message pushSink, in ResourceID_Type destinationResource);

The unsetSink operation removes a registered *Message* sink (Consumer) resource from a source (Producer) object's registered Message Sinks.

void unsetSink(in ResourceID_Type destinationResource);

The setMultipleSinks operation registers a set of *Message* sink (Consumer) objects for call back by a source (Producer) object.

void setMultipleSinks(in Destinations destinationSinks);

The getSink operation requests the *Message* sink (Consumer) object reference that is responsible for processing data to be received from the requesting source (Producer) object.

Message getSink(in ResourceID_Type sourceResource);

The getTransferSize operation gets the maximum transfer message size.

unsigned long getTransferSize();

The setTransferSize operation sets the suggested transfer message size for the Producer Source.

void setTransferSize(in unsigned long size);

};

5.2.7.1.1.4 Message

The *Message* interface provides multiple operations, based on the CORBA IDL basic types, for pushing data from a Producer to a Consumer. The recommended implementation for *Message* operations is a one way CORBA operation implemented as Synchronization with Socket method or as a co-location call. The co-location call acts as a C language Function call.

Each *Message* operation supports a “message” parameter composed of an unbounded sequence of a specific CORBA IDL basic type and an “options” parameter for describing optional properties or control information to be sent with the “message”. The “options” parameter is of type “Properties”, which is an unbounded sequence of name/value pairs of type DataType.

Message Relationships

The definition of the Message interface captured in Rational Rose using UML notation is as shown in Figure 5.2.7.1.1.4-1.

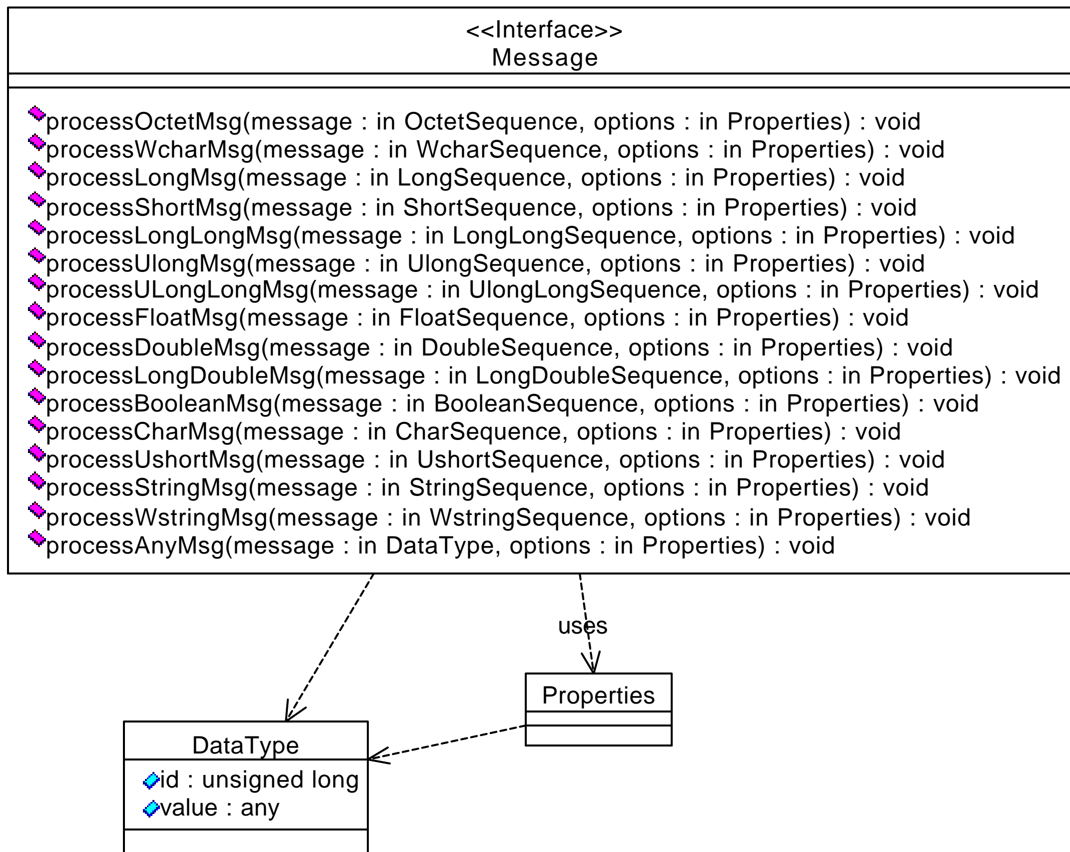


Figure 5.2.7.1.1.4-1. *Message* Relationships

Message Interfaces

The IDL for the *Message* interface produced from the Rational Rose diagrams in Figure 5.2.7.1.1.4-1 is shown below:

interface Message {

Nested Types (unbounded sequences of the CORBA IDL basic types):

This type is an unbounded sequence of octets (8-bit unconverted bytes):

typedef sequence<octet> OctetSequence;

This type is an unbounded sequence of characters:

typedef sequence<char> CharSequence;

This type is an unbounded sequence of signed short (16-bit) integers ($-2^{15}...2^{15}-1$):

typedef sequence<short> ShortSequence;

This type is an unbounded sequence of signed long (32-bit) integers ($-2^{31}...2^{31}-1$):

typedef sequence<long> LongSequence;

This type is an unbounded sequence of signed long long (64-bit) integers ($-2^{63}...2^{63}-1$):

typedef sequence<long long> LongLongSequence;

This type is an unbounded sequence of unsigned short (16-bit) integers ($0...2^{16}-1$):

typedef sequence<unsigned short> UshortSequence;

This type is an unbounded sequence of unsigned long (32-bit) integers ($0...2^{32}-1$):

typedef sequence<unsigned long> UlongSequence;

This type is an unbounded sequence of unsigned long long (64-bit) integers ($0...2^{64}-1$):

typedef sequence<unsigned long long> UlongLongSequence;

This type is an unbounded sequence of IEEE single-precision floating point numbers:

typedef sequence<float> FloatSequence;

This type is an unbounded sequence of IEEE double-precision floating point numbers:

typedef sequence<double> DoubleSequence;

This type is an unbounded sequence of IEEE double-extended floating point numbers:

typedef sequence<long double> LongDoubleSequence;

This type is an unbounded sequence of booleans.

typedef sequence<boolean> BooleanSequence;

This type is an unbounded sequence of wide characters (size is implementation-dependent):

typedef sequence<wchar> WcharSequence;

This type is a CORBA unbounded sequence of strings.

typedef sequence<string> StringSequence;

This type is a CORBA unbounded sequence of wide strings.

typedef sequence<wstring> WstringSequence;

Operations:

The following operations are used to push a sequence of information, formatted according to one of the CORBA IDL basic types listed above, from one *Resource* (a producer or "Push Source") to one or more registered destination *Resources* (consumers or "Push Sinks"). The destination *Resources* are registered using the operations of the *MessageRegistration* interface.

oneway void processOctetMsg(in OctetSequence message, in Properties options);
oneway void processWcharMsg(in WcharSequence message, in Properties options);
oneway void processLongMsg(in LongSequence message, in Properties options);
oneway void processShortMsg(in ShortSequence message, in Properties options);
oneway void processLongLongMsg(in LongLongSequence message, in Properties options);
oneway void processUlongMsg(in UlongSequence message, in Properties options);
oneway void processULongLongMsg(in UlongLongSequence message, in Properties options);
oneway void processFloatMsg(in FloatSequence message, in Properties options);

```
        oneway void processDoubleMsg(in DoubleSequence message, in Properties options);
        oneway void processLongDoubleMsg(in LongDoubleSequence message, in Properties
        options);
        oneway void processBooleanMsg(in BooleanSequence message, in Properties
        options);
        oneway void processCharMsg(in CharSequence message, in Properties options);
        oneway void processUshortMsg(in UshortSequence message, in Properties options);
        oneway void processStringMsg(in StringSequence message, in Properties options);
        void processWstringMsg(in WstringSequence message, in Properties options);
        oneway void processAnyMsg(in DataType message, in Properties options);
};
```

5.2.7.1.1.5 Resource

The *Resource* interface defines the minimal interface for any software resource within the radio. A *Resource* simply inherits and encapsulates the interfaces of *MessageRegistration*, *Message*, *LifeCycle*, and *StateManagement*. This small set of operations is all that a *DomainManager* object will know about for any *Resource* object within the radio. Application *Resources* can, however, extend this basic *Resource* definition and use their extensions among themselves or by their Application GUI, since they are the only ones that know these extensions. The *DomainManager* interface provides the mechanism of retrieving *Resources* that have been created for direct GUI usage.

Resource Relationships

The definition of the *Resource* interface captured in Rational Rose using UML notation is as shown in Figure 5.2.7.1.1.5-1.

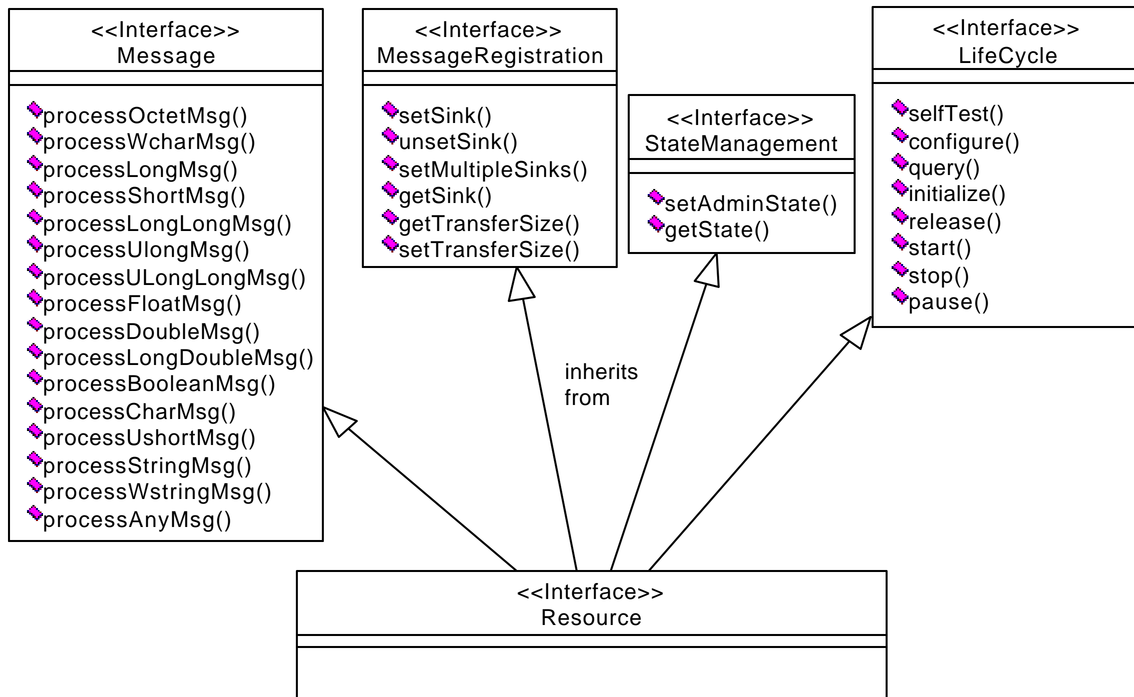


Figure 5.2.7.1.1.5-1. *Resource* Relationships

Resource Interfaces

The IDL for the *Resource* interface produced from the Rational Rose diagrams in Figure 5.2.7.1.1.5-1 is shown below:

```
interface Resource : Message, MessageRegistration, StateManagement, LifeCycle {
};
```

5.2.7.1.2 Framework Control Interfaces

5.2.7.1.2.1 DomainManager

In order to provide for the interoperability of both hardware and software resources within a radio it is necessary to provide for a mechanism within the system to manage resources. Resources need to be treated in a generic manner such that hardware and software may be moved from one system to another. This capability must allow for a module, a waveform application, and other software applications to be updated without requiring code changes to the CF. To assure the ability for the CF to support this functionality, the design includes a *DomainManager* Figure 5.2.7.1.2.1-1.

In a SDR implementation, it is necessary to provide a means to match generic hardware and software resources to the desired user functionality. As shown in Figure 5.2.3.3.5-8 the *DomainManager* is the

CF component responsible for the allocating the Physical Resources in the radio based on the required Functional Applications. The *DomainManager* uses a *Domain Profile* to determine the proper allocation of resources (hardware and software) in the system.
The *DomainManager* and *DomainProfile* are similar concepts to the SDRF Handheld Working Group concepts of a "Switcher" and a "Capability Exchange" respectively.
Figure 5.2.7.1.2.1-2 shows that a SDR Application can be thought of as a collection of Resources connected together in a particular order to provide the desired functionality. Each Resource can be made up of other Resources either software and/or hardware and in turn can require other resources.
There will be at least one Domain Manager in every SDR implmentation. The Domain Manager component can logically be grouped into two categories: Host and Registration. The Host operations are used to configure the radio, manage radio capabilities, manage software resources, and provide radio status information. The Registration operations provide the mechanism for the Resource Managers in the system to acquire and report information about the capabilities of the system.

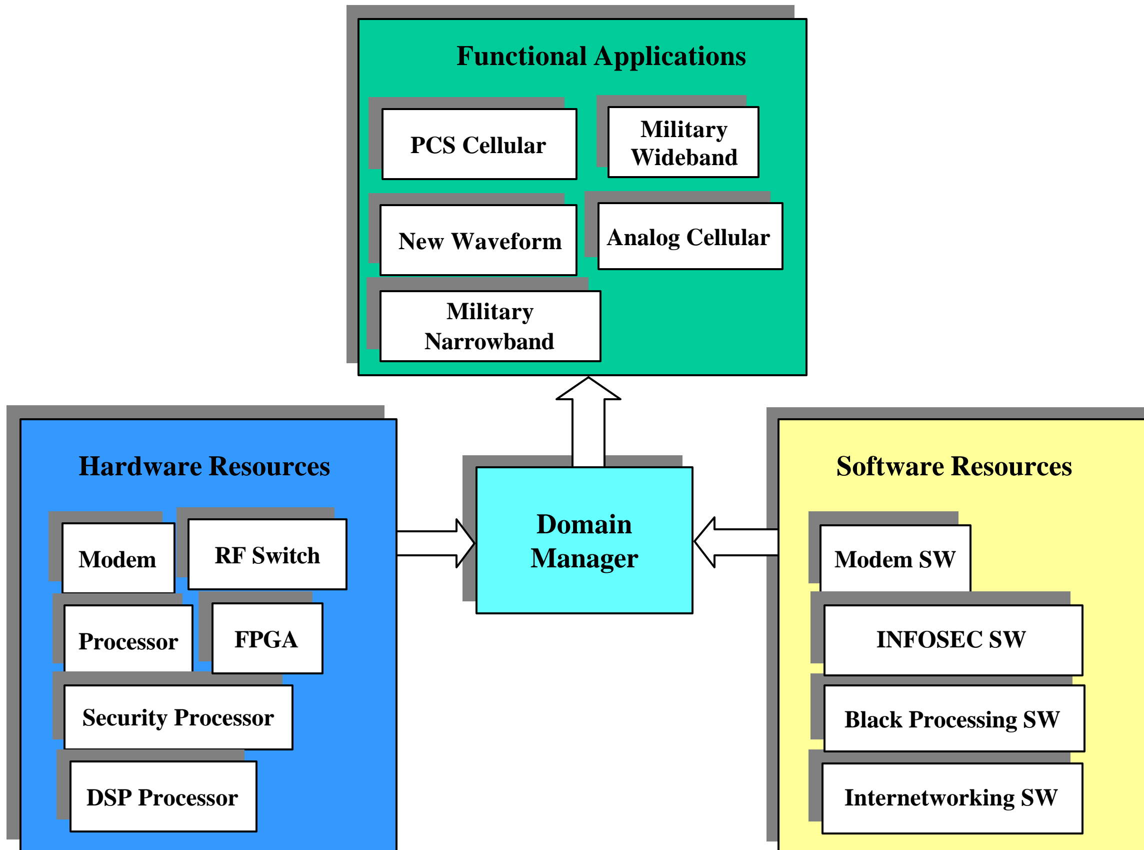


Figure 5.2.7.1.2.1-1. Domain Management

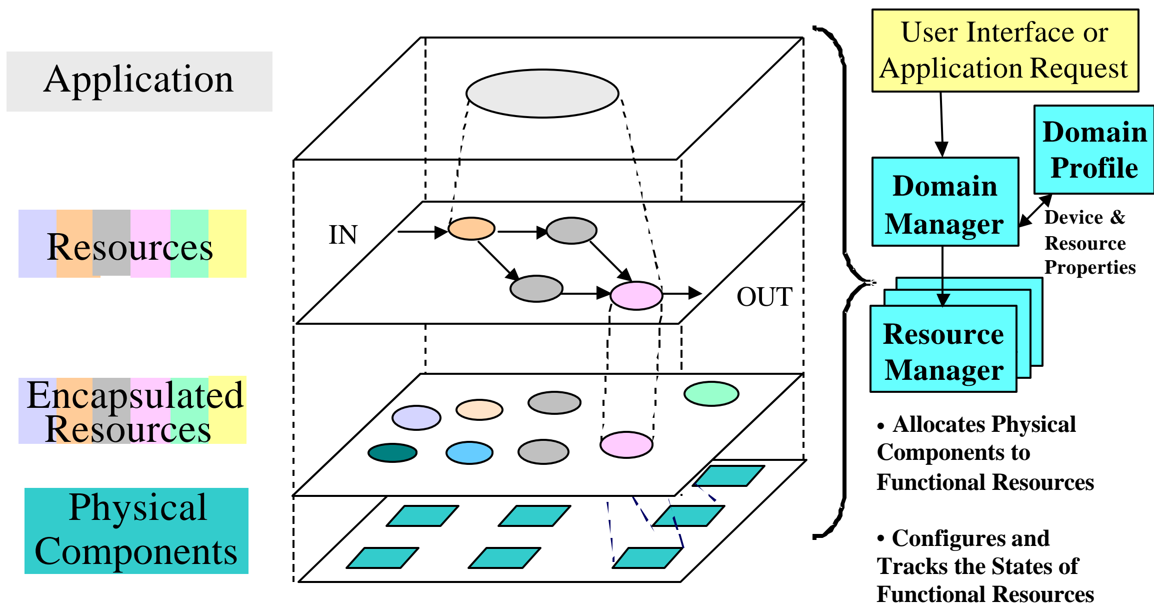


Figure 5.2.7.1.2.1-2. Layered *Resource* Allocation

Profile Domain

The *DomainManager* uses a *Domain Profile* to store the necessary information about the resources in the system to properly allocate and de-allocate hardware and software resources to a required Functional Application. Physical resources such as modems, processors, security processors, Digital Signal Processors (DSP), and Field Programmable Gate Arrays (FPGAs), and other resources each report their presence in the system through the CF component called the *ResourceManager*. As each physical resource in the system reports itself, the data about the resource is stored in the *Domain Profile*. The *DomainManager* will use this information to determine the proper allocation of these resources.

A Domain Profile is the information used by the DomainManager to perform a variety of tasks including Boot up and Initialize of Core Framework Components and Resources, the re-initialization of radio configuration based on the state of the radio at power down, and the allocation and de-allocation of resources. The *Domain Profile* also provides a place for the configuration management of software and hardware resources available to the system. Using the *Install Service*, software modules are loaded into a *File System;* and information such as version, resource requirements, and other information is stored in the *Domain Profile*. The *DomainManager* uses this information to manage system resources to accomplish the required capability of the system. For instance, an algorithm is written and compiled for a particular Digital Signal Processor (DSP) such as the TI-TMS320C6000. Using the *Install Service,* the software module is loaded into the *File System* and information such as version, physical hardware requirements, and other important parameters are stored in the *Domain Profile*. The *DomainManager* uses this information to load the software resource onto the proper allocated physical assets to accomplish the required capability in the system.

The *Domain Profile* also contains information about Functional Applications. Functional Applications are a collection of generic resources that accomplish a specific user desired purpose. Examples of the Functional Applications in a radio will include waveforms, network routing applications, and other high level applications. Application portability across many different versions of an SDR product line reduces development and maintenance costs. In order for an application to be portable, it will be necessary to store information about the required capabilities for the application in the *Domain Profile*. The *Domain Manager* will use this information to determine the requirements of the application when the User makes a request for it.

Application Control

The DomainManager provides interfaces to the User for starting, stopping, configuring, and managing the radio Functional Applications. A User (automated or using an HCI) will request that a specific User Function(s) be provided in the radio. (Using the *CreateVirtualCircuit ()* interface). Based on the Functional Application requested the *DomainManager* uses the information in the *Domain Profile* to determine the devices to be allocated, configured, and used for the desired Function(s). If the required devices are available and operational state is enabled, then the *DomainManager* loads and executes the software resource files necessary to support the mode of operation. Software Resources are loaded on the appropriate processors using a *ResourceManager* interface, and status is returned to identify whether the Functional Application has been created for this request, or not.

The *DomainManager* is also responsible for the transition of *Resource* Objects through their various states using the *StateManagement* interfaces. The *Domain Profile* will indicate which *Resources* to create, and what states to transition *Resources* to. Based upon *Domain Profile*, the *DomainManager* may use Naming or Trading Services to obtain a resource. A *Factory* resource can be obtained from Naming or Trading Services, and may be used for starting other *Resources*. The *DomainManager* is responsible for the setup and control of Resources within a Functional Application. The *DomainManager* assure the *Resources* operations, including health and status, and other pertinent information about the *Resource*. The *DomainManager* also provides a window to the User (HCI) into the state of those *Resources*.
The *DomainManger* is the CF Component that provides the configuration capability that allows for the ultimate flexibility for setting up Application *Resources*. Because of the generic approach to *Resources* in the system, new waveform designs, hardware modules, and new architectural designs may still be supported by the CF now and in the future.

DomainManager Relationships

Below is a Description of the Relationships that the *DomainManager* component has with other CF Components in the system.
<uses> *Factory* – to request a Resource to be instantiated.
<manages> *Resources* – configures, manages the generic Resources in the system
<uses> *FileManager* – to access the necessary files
<allocates resources to> *ResourceManager*

The relationships for this interface are shown in the *DomainManager* Relationships in Figure 5.2.7.1.2.1-3.

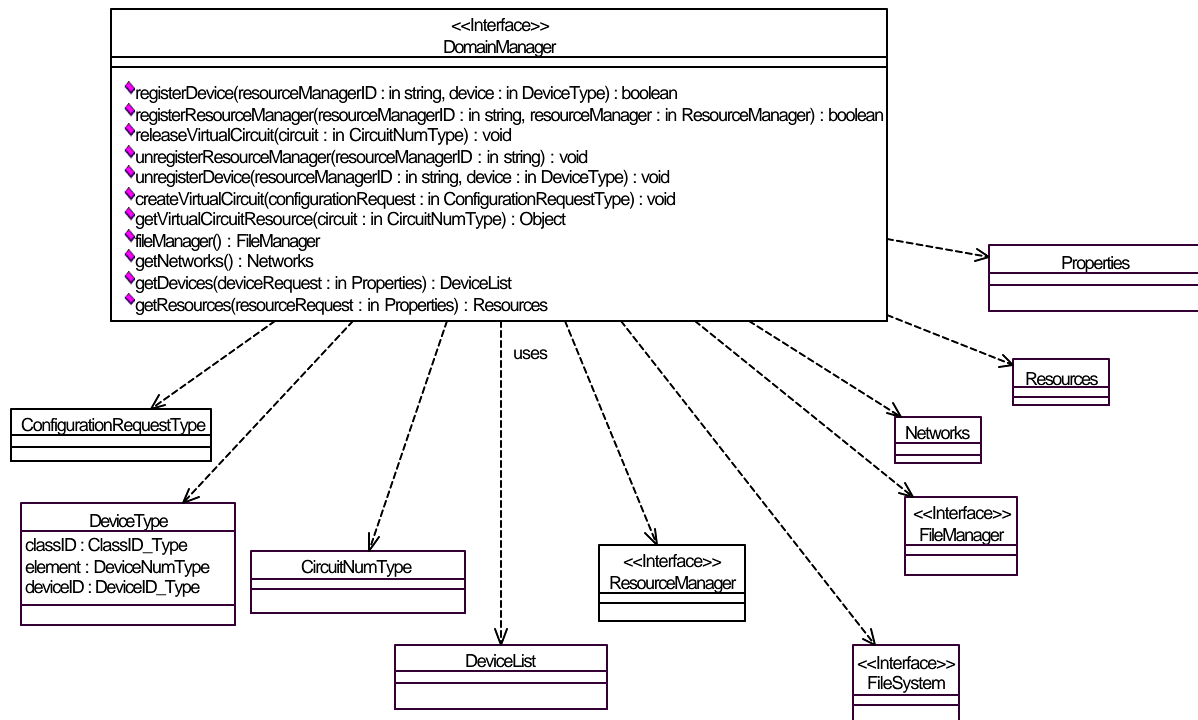


Figure 5.2.7.1.2.1-3. *DomainManager* Relationships

DomainManager Interfaces

Below is the list of interfaces for the *DomainManager* CF component for managing the application setup and teardown capabilities of a radio.

interface DomainManager {

The registerDevice capability adds a device entry into the DomainManager for a specific Resource Manager object.

boolean registerDevice(in string resourceManagerID, in DeviceType device);

The registerResourceManager adds a Resource Manager object entry into the Domain Manager object database.

boolean registerResourceManager(in string resourceManagerID, in ResourceManager resourceManager);

The releaseVirtualCircuit operation releases an active circuit and releases all allocated assets.

void releaseVirtualCircuit(in CircuitNumType circuit);

The unregisterResourceManager capability unregisters a Resource Manager object from the DomainManager.

void unregisterResourceManager(in string resourceManagerID);

The unregisterDevice operation removes a device entry from the Domain Manager object.

void unregisterDevice(in string resourceManagerID, in DeviceType device);

The createVirtualCircuit operation creates a virtual circuit within the radio.

oneway void createVirtualCircuit(in ConfigurationRequestType configurationRequest);

The getVirtualCircuitResource operation returns the object reference for the specified virtual circuit.

Object getVirtualCircuitResource(in CircuitNumType circuit);

The FileMan operation returns a FileManager object reference to the main FileManager repository.

FileManager fileManager();

The getNetworks operation returns network information based upon the input network request.

Networks getNetworks();

The getDevices operation returns devices information based upon the input device request.

DeviceList getDevices(in Properties deviceRequest);

The getResources operation returns the resources information based upon the input resource request.

Resources getResources(in Properties resourceRequest);

};

5.2.7.1.2.2 ResourceManager

The *ResourceManager* is a CF application in a SDR implementation for booting, initializing, and reporting the capabilities of hardware modules. The *ResourceManager* interfaces define the means for communicating with all the devices on a particular module within the radio. Processors with an instantiation of the *ResourceManager* component are responsible for reporting to the *DomainManager* the pertinent information about the hardware devices that it knows about. The *ResourceManager* uses the *deviceProperties*, *deviceList*, and *deviceExists* methods to provide this information to the *DomainManager*. The *DomainManager* uses this information to allocate these devices to specific requested User functions.

Figure 5.2.3.3.5-11 visually demonstrates how *ResourceManagers* report device properties to the *DomainManager*. The *DomainManager* uses the Domain Profile to store the information about the Devices. The *ResourceManager* is responsible to indicate the state of the devices, their capabilities, and other pertinent information about the devices. For proper interoperability, a common set of properties needs to be reported for each module within the radio. These properties should provide the basis for deciding the allocation constraints on the system. Module developers may also provide additional properties for additional usable information.

A *ResourceManager* also provides the capability to load and execute software on *Resources* within its control. The *DomainManager* tells the *ResourceManager* what resources to use, and the *ResourceManager* load and executes the proper software on the given hardware resources. A *ResourceManager* upon startup may create a *Logger, FileManagers, FileSystem,* and other Resources based on the direction of the *DomainManager* using the Domain Profile.

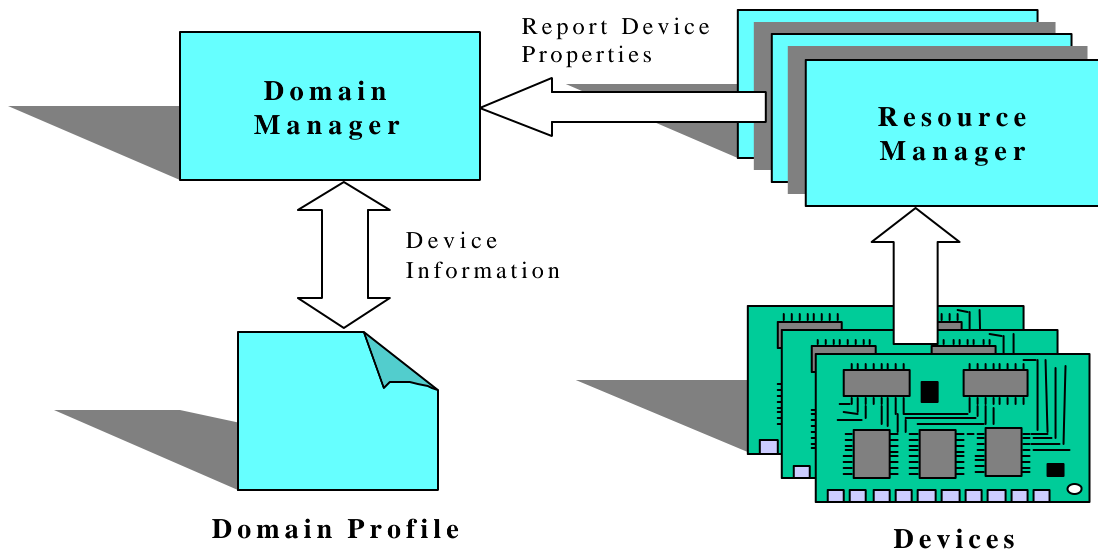


Figure 5.2.3.3.5-1. ***ResourceManagers*** **Report Device Properties**

ResourceManager Relationships

Below is a list of Relationships that the *ResourceManager* component has with other Framework Components in the system.
<Uses> FileSystem – To load and unload software resource files
<Uses> DomainManager – To register itself or register or unregister a device.
<Uses> Logger – To log warnings or information, and alarm conditions

The relationships for this interface are shown in the *ResourceManager* Relationships in Figure 5.2.7.1.2.2-2

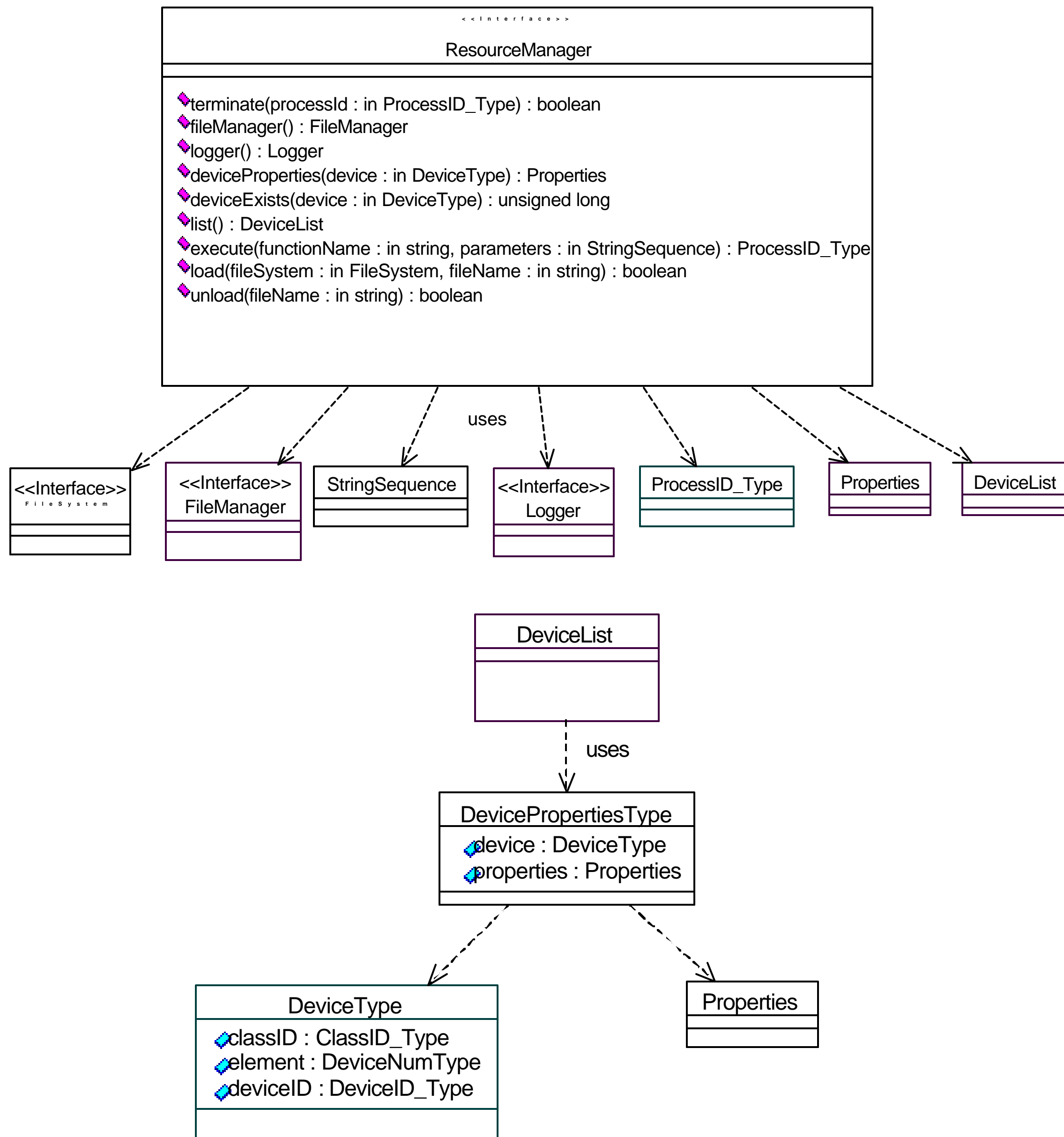


Figure 5.2.7.1.2.2-2. ***ResourceManager*** **Relationships**

ResourceManager Interfaces

Following is the list of Interfaces for the *ResourceManager* CF component for managing the plug and play capabilities of the radio.

interface ResourceManager {

The terminate operation terminates the execution of the function on the device the Resource Manager is managing.

boolean terminate(in ProcessID_Type processId);

The fileManager operation returns the file manager associated with this ResourceManager.

FileManager fileManager();

The logger operation returns the logger associated with this ResourceManager.

Logger logger();

The deviceProperties capability returns the properties for the specified device. If the specified device does not exist, a null Properties set is returned.

Properties deviceProperties(in DeviceType device);

The deviceExists operation returns the number of registered devices based upon the input type.

unsigned long deviceExists(in DeviceType device);

The DeviceList operation provides a list of the hardware devices along with their properties that are currently associated with this *ResourceManager* object.

DeviceList list();

The execute operation executes the given function name using the arguments that have been passed in and returns an ID of the process that has been created.

ProcessID_Type execute(in string functionName, in StringSequence parameters);

The load operation loads a file based on the given fileName using the input *FileSystem* to retrieve it. True is returned if the load was successful, otherwise False is returned.

boolean load(in FileSystem fileSystem, in string fileName);

The unload operation unloads software based on the fileName and returns a success or failure status.

boolean unload(in string fileName);

};

5.2.7.1.3 Framework Services Interfaces

5.2.7.1.3.1 File

The *File* interface provides the basic primitive interfaces for accessing any non-collocated file within an SDR implementation. This interface may be extended for specific application files types.

File Relationships

Below is a list of Relationships that the *File* component has with other Framework Components in the system.
<uses> *Message* Interface
The definition of the *File* interface captured in Rational Rose using UML notation is as shown in Figure 5.2.7.1.3.1-1.

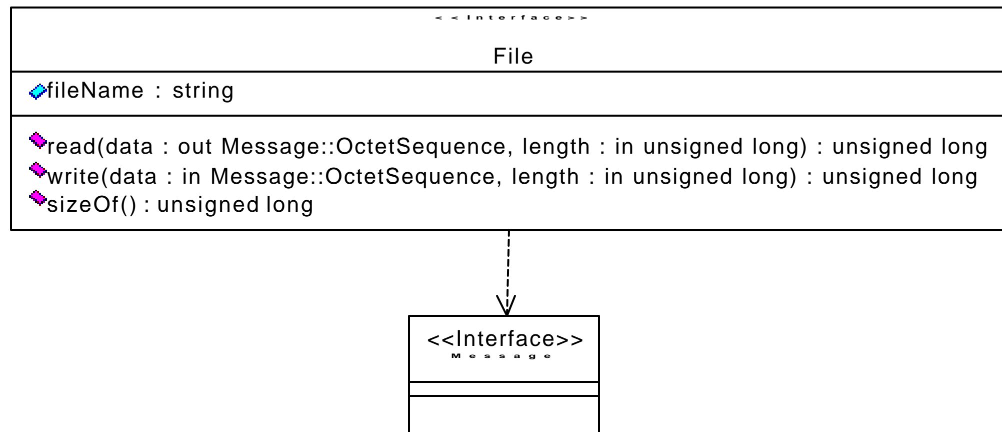


Figure 5.2.7.1.3.1-1. *File* Relationships

File Interfaces

INTERFACE FILE {
This attribute provides read access to the fully qualified name of the file.
readonly attribute string fileName;

The read operation reads data from the file. The read operation returns a True value if the read was successful, otherwise False is returned.
unsigned long read(out Message::OctetSequence data, in unsigned long length);

The write operation writes data to the file. The write operation returns a True value if the write was successful, otherwise False is returned.
unsigned long write(in Message::OctetSequence data, in unsigned long length);

The sizeOf operation returns the current size of the file.
unsigned long sizeOf();
};

5.2.7.1.3.2 FileSystem

The *FileSystem* interface provides basic OS file system operations to access non-collocated files within the radio. The interface also provides the capability for accessing non-collocated *FileSystems* within the radio. The radio may use one to many (1…*) *FileSystems* as depicted in Figure 5.2.7.1.3.2-1.

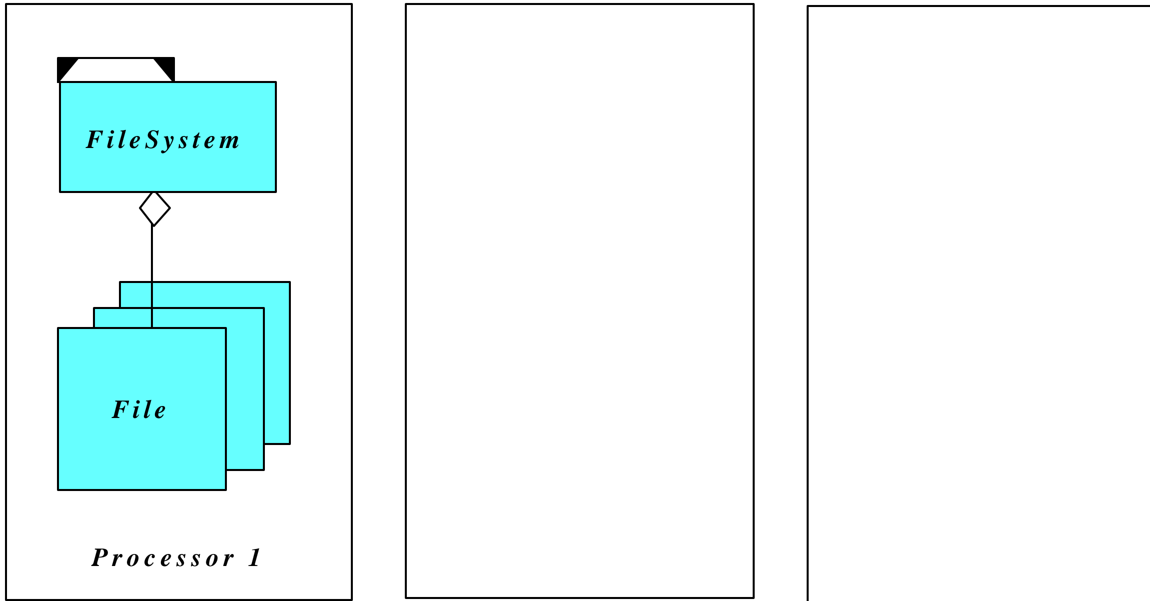


Figure 5.27.1.3.2-1. Conceptual *FileSystem* Relationships

FileSystem Relationships

Below is a list of the relationships that the *FileSystem* component has with other CF components in the system.

<uses> *File* – to open, .delete, and create a file.

<uses> *Logger* – to log information.

The definition of the *FileSystem* interface captured in Rational Rose using UML notation is as shown in Figure 5.2.7.1.3.2-2.

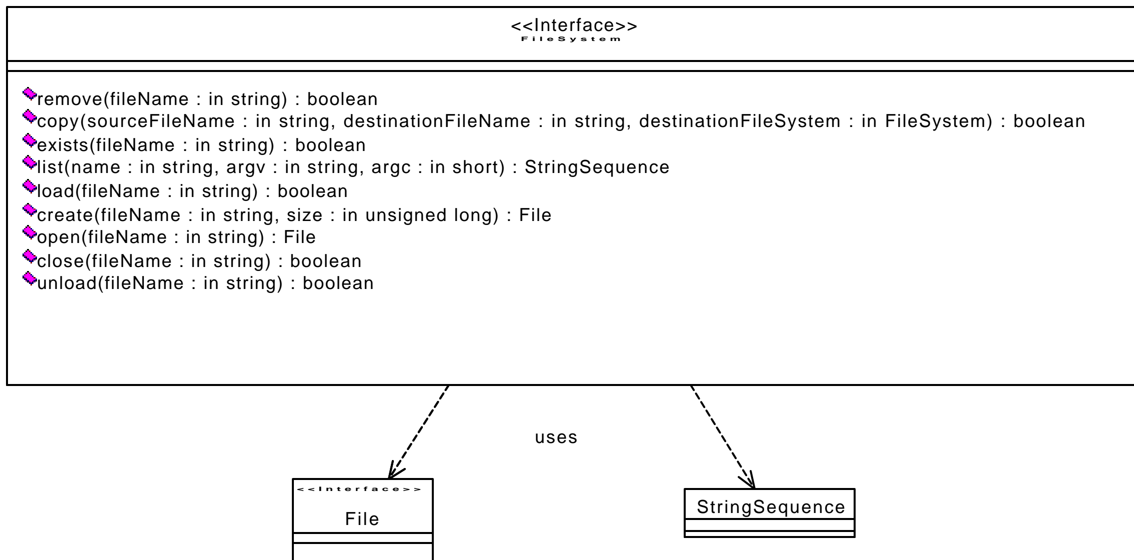


Figure 5.2.7.1.3.2-2. *FileSystem* Relationships

FileSystem Interfaces

interface FileSystem {

The remove operation removes the file with the given name from the file system. The name includes the full path of the file. The operation returns true on success, false on fail.

boolean remove(in string fileName);

The copy operation copies the source file with the specified name to the destination FileSystem. The copy operation returns true on success, false on fail.

boolean copy(in string sourceFileName, in string destinationFileName, in FileSystem destinationFileSystem);

The exists operation checks to see if a file exists based the file name parameter and returns true if found, false otherwise. The file name should include the path where to search for the file.

boolean exists(in string fileName);

The list operation behaves similar to the UNIX "ls" command.

StringSequence list(in string name, in string argv, in short argc);

The load operation loads a file based on the file Name and returns a success or failure status. The load allows a file in the file system to be loaded into RAM without having to open a file and read the file to load the file into RAM.

boolean load(in string fileName);

The create operation creates a new File based upon the input file name. The size is used to determine if the file system has enough space for creating the new file and to verify the file size when closing the file. A null file object reference is returned if the name already exists or size is too large for file system.

File create(in string fileName, in unsigned long size);

The open operation opens a File based upon the input file name. A null File object reference is returned if name does not exist in the file system.

File open(in string fileName);

The close operation releases a File object that has been created and registered with the ORB. A True value is returned upon successful file close, otherwise False is returned.

boolean close(in string fileName);

The unload operation unloads a file based on the fileName and returns a success or failure status. The unload operation unloads the software from RAM.

boolean unload(in string fileName);

};

5.2.7.1.3.3 FileManager

The CF includes the *FileManager* interface for common access to non-collocated *Files* and *FileSystems.* The *FileManager* organizes *FileSystems* within the radio, and makes the various *FileSystems* available to any *Resource* in the system. The *FileManager* is accessible to non-core

application resources as well as CF *Resources* to locate the various *FileSystems* within the radio. The *FileManager* is the top-level access to all the files in the system. Files may be located anywhere within the architecture. Files may be in found in memory, hard-drive space, flash, or other means of storage. The generic *FileManager* interface will allow for multiple FileSystems to be mapped and located using the FileManager interface.

Figure 5.2.7.1.3.3-1 shows a *FileManager* on Processor 2 which provides access to *FileSystem* 1 on Processor 1 and to the local *FileSystem* 2 on its own processor. The second *FileManager* on Processor 3 provides access to the local *FileSystem* 3, and *FileSystem* 2 on Processor 2. Both *FileManagers* have access to the Processor 2 *FileSystem* even though the processors may be of different type or architecture.

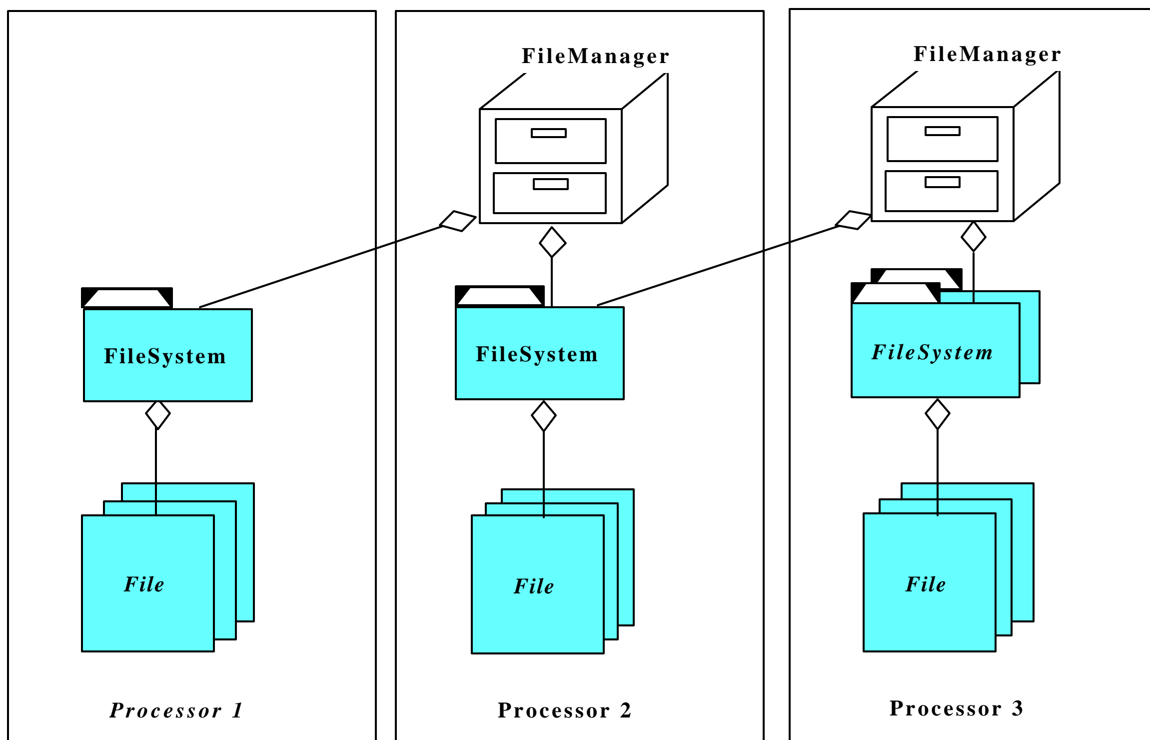


Figure 5.2.7.1.3.3-1. File Management

FileManager Relationships

The definition of the *FileManager* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.3.3-2.

<uses> *FileSystem* – to access and list files in the system

<uses> *Logger* – to log information.

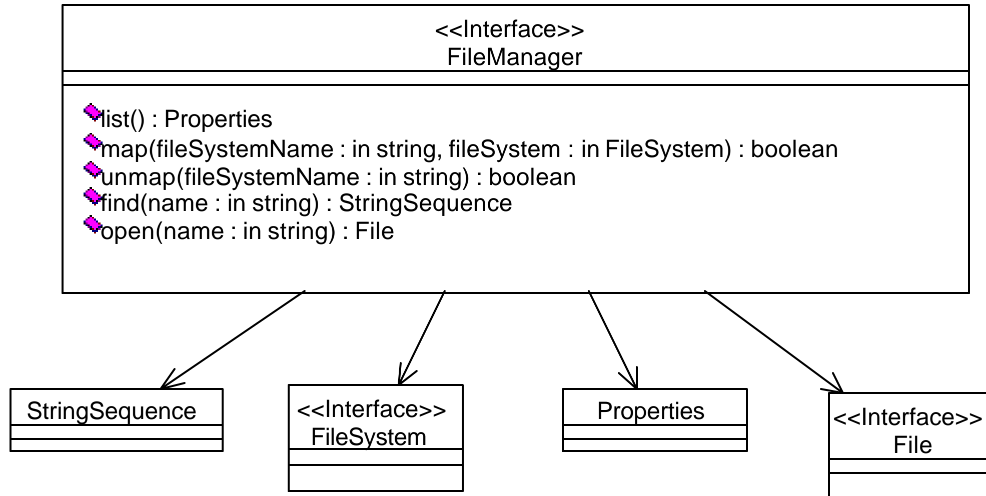


Figure 5.2.7.1.3.3-2. *FileManager* Relationships

FileManager Interfaces

Below is the list of *FileManager* interfaces for managing the *FileSystems* of an SDR implementation.

interface FileManager {

The list operation returns a list for FileSystem object references.

Properties list();

The map operation registers a FileSystem object with the File Manager object. True is returned if the mapping was successful; otherwise false is returned.

boolean map(in string fileSystemName, in FileSystem fileSystem);

The unmap operation removes a FileSystem reference from a File Manager object.

boolean unmap(in string fileSystemName);

The find operation returns a list of Files that are found based upon the input criteria.

StringSequence find(in string name);

File open(in string name);

};

5.2.7.1.3.4 Installer

The *Installer* interface defines a standard mechanism in the SDR architecture for loading, initializing, and reporting the properties of software *Resources* in the system. The *Installer* interface provides the *DomainManager* with information about the software *Resources* by populating the *Domain Profile.* Software *Resources* in an SDR implementation are persistent assets, and remain as usable *Resources* until deleted from the system. Similar to the *ResourceManager,* the *Installer* will provide information to the *Domain Profile*, and by combining both software and hardware resource information, the *DomainManager* is able to determine allocation of resources.

The *Installer* will provide a common interface allowing both HCI and over-the-air-programming (OTAP) components the capability of installing and uninstalling software *Resources* within a radio. The Administrator's ability to access the *Installer* service is verified prior to the *Installer* being invoked. This provides for secure management of the software.
The *Installer* provides the software *Resource* properties and requirements to the *DomainManager* through the population of the *Domain Profile*. The *Resource* requirements consist of the definition of the necessary hardware and software for the operation of the installed software *Resource*. The software properties consist of information needed by the *DomainManager* about the installed modules such as:

- Their location in the *File System*
- Software version identification
- Timestamp
- Size
- Security level.

This information is used to allocate the software *Resources* when a user Functional Application is requested.

5.2.7.1.3.5 Logger

The *Logger* interface is used to capture alarm, warning, and informational messages during the execution of software within the radio. A *Logger* object interfaces with two other types of objects:

- Message Producers – Objects in the system which send messages to be logged by the *Logger*.
- Message Consumers - Objects in the system which register with the *Logger* to receive logged messages of particular log levels.

The *Logger* interface provides operations for both Message Producers and Message Consumers.
The *Logger* uses Log Levels to determine the severity of a message being logged by a Message Producer. Log Levels range in value from LEVEL_14 (alarm) to LEVEL_0 (purely informational) and are provided by Message Producers to the *Logger* along with the message to be logged. The *Logger* determines if any Message Consumers are registered to receive messages at the level provided by the Message Producer and will pass on the message and Log Level to the appropriate Message Consumer(s).
The *Logger* also provides the ability:

- To display the last ***n*** number (where ***n*** is defined by the user) of logged messages to the system console.
- To enable and disable the logging of messages to a file.
- For Message Consumers to filter the types of messages which they are receiving.

Logger Relationships

The definition of the *Logger* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.3.5-1.

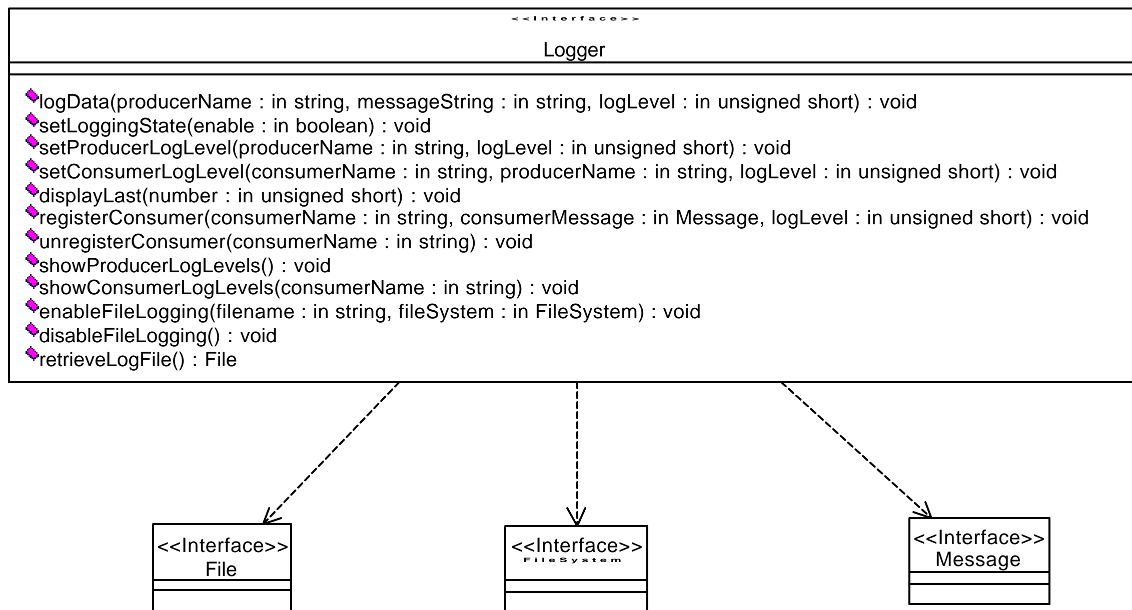


Figure 5.2.7.1.3.5-1. *Logger* Relationships

Logger Interfaces

The IDL for the *Logger* interface produced from the Rational Rose diagram in figure is shown below:

interface Logger {

The logData operation logs a log string and a time stamp to the console depending on the current log level set for the producer object and the log level of the string. It also logs the same information to a file if file logging is enabled for the object. The operation also pushes the data to registered consumers based upon their log levels. The logger log level is automatically assigned to a new producer.

```
oneway void logData(in string producerName,
                    in string messageString,
                    in unsigned short logLevel);
```

This operation enables the logging of all messages at the currently set level for each object, or disables the logging of all messages from all objects, depending on the value of the argument.

```
void setLoggingState(in boolean enable);
```

This operation sets the log level for a producer object. All incoming log strings <= to the currently set level are displayed/saved. The log level is bitmapped 00 00 - 7F FF (hex) with bit 16 being a control bit to allow for log level manipulation.

Examples:

LogLevel = C010 h (1100 0000 0001 0000 b) indicates only levels 14 and 4 are to be displayed.

LogLevel = 000A h indicates levels 10 and below will be displayed, and bits 4-14 are unused.

void setProducerLogLevel(in string producerName,
in unsigned short logLevel);

This operation sets the log level for a consumer object. All incoming log strings <= to the currently set level are displayed/saved. The log level is bitmapped 00 00 - 7F FF (hex) with bit 16 being a control bit to allow for log level manipulation.

Examples:

LogLevel = C010 h (1100 0000 0001 0000 b) indicates only levels 14 and 4 are to be displayed.

LogLevel = 000A h indicates levels 10 and below will be displayed, and bits 4-14 are unused.

void setConsumerLogLevel(in string consumerName,
in string producerName,
in unsigned short logLevel);

This operation displays at the console the last number of log messages stored locally within the logger.

void displayLast(in unsigned short number);

This operation registers a consumer object with the logger. Initially all producers' messages that pass the input logLevel are pushed to the consumer. A consumer can change its filtering by the setConsumerLogLevel operation.

void registerConsumer(in string consumerName,
in Message consumerMessage,
in unsigned short logLevel);

This operation unregisters a consumer object.

void unregisterConsumer(in string consumerName);

This operation displays the current log level for all producer objects.

void showProducerLogLevels();

This operation displays the current log levels for a consumer object.

void showConsumerLogLevels(in string consumerName);

This operation stores to disk the incoming log based on the current log level. It does not affect output to the console.

void enableFileLogging(in string filename,
in FileSystem fileSystem);

This operation disables storage to disk of the incoming log based on the current debug level.

void disableFileLogging();

This operation retrieves the current log file.

File retrieveLogFile();

};

5.2.7.1.3.6 Timer

The CF *Timer* service provides operations for synchronizing time within the radio as well as for creating and managing time-based events.

The *Timer* service is made up of two interfaces, a *Time Service* interface and a *Timer Event* Service interface.

The *Time Service* interface manages the following two types of objects:

- Universal Time Objects (UTO)
- Time Interval Objects (TIO)

The UTOs are used to represent a time and the TIOs are used to represent a time interval. The *Time Service* interface provides operations for creating UTOs or TIOs, as well as operations to create TIOs based upon UTOs and vice versa. The *Time Service* also provides operations for returning the current time and manipulating time formats.

The *Timer Event Service* manages Timer Event Handler objects. To use this service, the CORBA Event Service is used to create an Event Channel. The Timer Event Handler then registers the Event Channel for use. The Timer Event Handler is then used to set up a Timer Event as needed using an UTO.

The CF *Timer* service is based on the CORBA Time Service. Concerns about the real-time nature of current COTS implementations of the CORBA Time Service may preclude its use, however. That being the case, the SDRF recommends the CF Timer service implement the CORBA Time Service. Ultimately, COTS implementations of the CORBA Time Service may replace the CF *Timer* service implementation if deemed acceptable.

5.2.7.1.4 Optional Framework Interfaces

5.2.7.1.4.1 Factory

The *Factory* interface defines a generic interface that can be implemented by any *Factory Resource* within the radio. Each *Factory* object creates a specific type of *Resource* within the radio. The *Factory* interface provides a one-step solution for creating a *Resource*, reducing the overhead of starting up *Resources*. The *Factory* interface is similar to the COM Factory class and is based on the industry accepted Factory design pattern[3]. In CORBA, there are two separate object reference counts. One for the client side and one for the server side. The *Factory* keeps a server-side reference count of the number of clients that have requested the resource. When a client is done with a resource, the client releases the client resource reference and calls **releaseResource** to the *Factory*. When the server-side reference goes to zero, the server resource object is released from the ORB that causes the resource to be destroyed.

---

3 "Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 107.

Factory Relationships

The definition of the *Factory* interface, captured in Rational Rose using UML notation, is as shown in Figure 5.2.7.1.4.1-1.

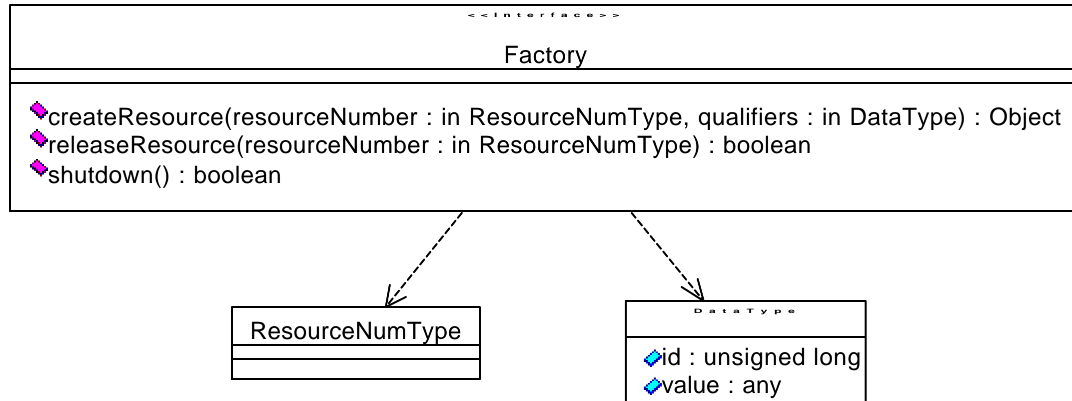


Figure 5.2.7.1.4.1-1. *Factory* Relationships

Factory Interfaces

The IDL for the *Factory* interface produced from the Rational Rose diagram in Figure 5.2.7.1.4.1-1 is shown below:

interface Factory {

The createResource operation returns a resource based upon the input resource number and qualifiers. If the resource does not already exist, then this operation creates the resource, else the operation returns the object already created for that resource number.

Object createResource(in ResourceNumType resourceNumber, in DataType qualifiers);

This operation removes the resource from the *Factory* if no other clients are using the resource. The resource to be released is associated with a specific resource number.

boolean releaseResource(in ResourceNumType resourceNumber);

This operation destroys all resources managed by this factory and terminates the factory object.

boolean shutdown();

};

5.2.7.1.4.2 Adapters

The SDR Software Architecture is easily extended to support non-CORBA-capable processing elements through the use of Adapters. Adapters are inserted into the architecture to provide the translation between non-CORBA-capable *Resources* and CORBA-capable *Resources*. The Adapter

concept is based on the industry accepted Adapter design pattern[4]. Since an Adapter implements the CF CORBA interfaces known to other CORBA-capable *Resources,* the translation service performed by the Adapter is transparent to the CORBA-capable *Resources*. Adapters become particularly useful to support non-CORBA-capable Modem, Security, and Host processing elements. Figure 5.2.7.1.4.2-1 depicts an example of message reception flow through the radio with and without the use of Adapters. Modem, Security, and Host Adapters implement the interfaces marked by the circled letters M, S, and H respectively. Notice that the Waveform Link and Waveform Network *Resources* are unaffected by the inclusion or exclusion of the Adapters. The interface to these *Resources* remains the same in either case.

5.2.8 Use Case View

"No system exists in isolation. Every interesting system interacts with human or automated actors that use that system for some purpose, and those actors expect that system to behave in predictable ways. A use case specifies the behavior of a system or a part of a system and is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor. Use cases serve to help validate the architecture and to verify the system as it evolves during development."[5]

The ability of the SDR Software Architecture definition to meet the needs of users (actors) is contained in the behavioral models specified in this section. Several example scenarios for the use cases depicted in Figure 5.2.8-1 have been modeled using the SDR interface definitions. This is the first step in validating the SDR Software Architecture.

---

4 "Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 139

5 Booch, Rumbaugh, Jacobson – The Unified Modeling Language User Guide, p.219

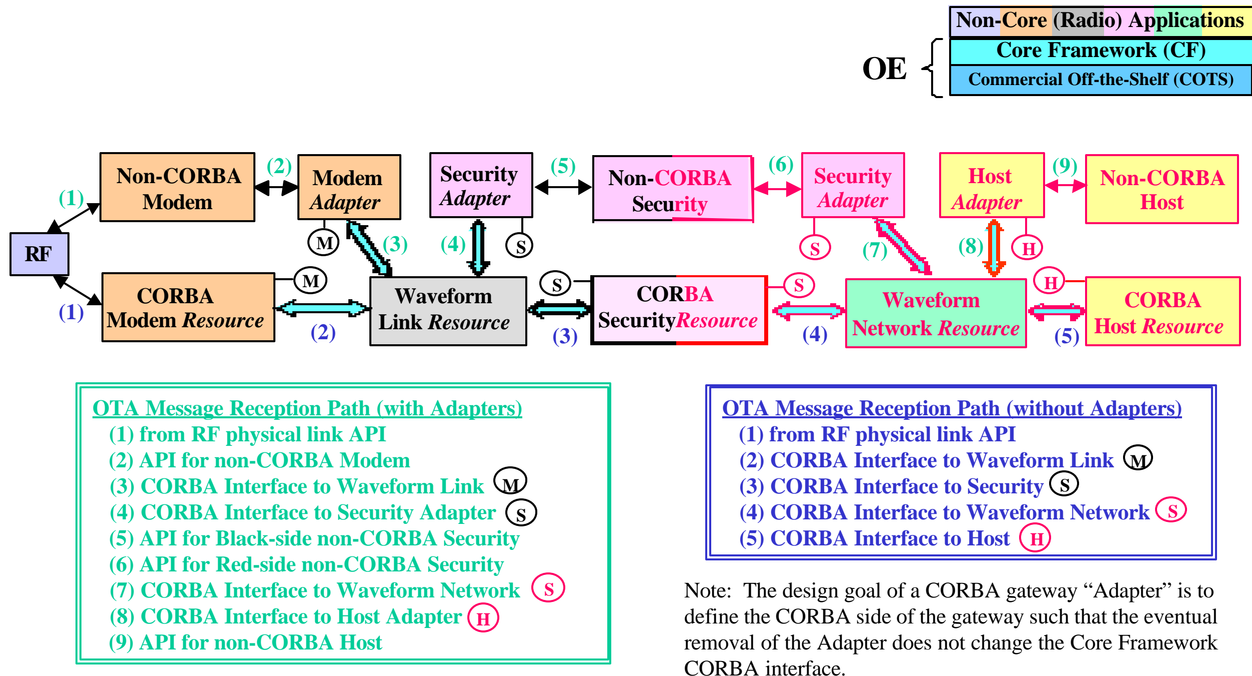


Figure 5.2.7.1.4.2-1. Example Message Flows with and without Adapters

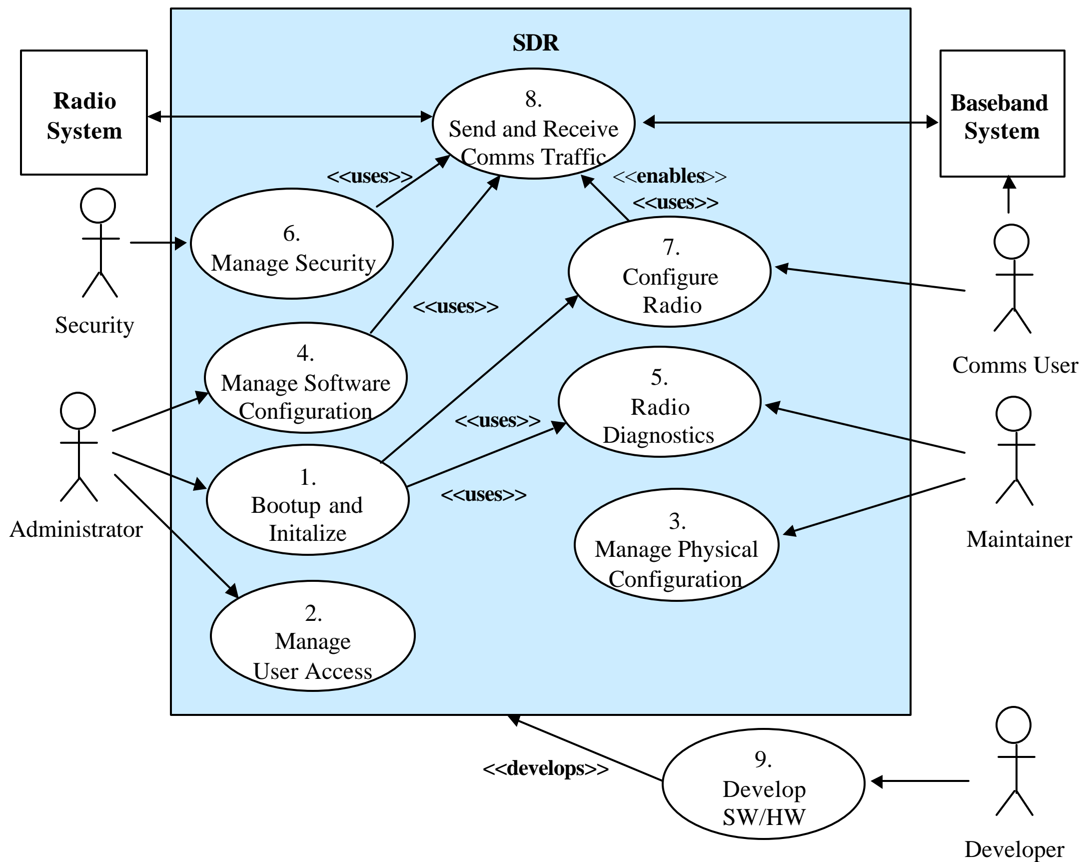


Figure 5.2.8-1. SDR Use Cases

5.2.8.1 Boot Up and Initialize Use Case

5.2.8.1.1 Power Up Scenario

This scenario provides an example of the Power Up and Initialization of the *Core Framework Objects*. The use case begins when the system is booting up. The flow of events is described below and depicted in Figure 5.2.8.1.1-1.

1. The OS start procedure loads and executes a *Factory* on the processor used by the *DomainManager* to create a *FileManager*.
2. The OS start procedure loads and executes a *Factory* on the processor used by the *DomainManager* to create a *FileSystem*.
3. The OS start procedure maps the *FileSystem* to *FileManager*.
4. The OS start procedure executes a *Factory* to create the *Domain Manager* on a designated processor.
5. The *DomainManager* uses the *FileManager* to locate the *DomainManager's DomainProfile*.
6. The *DomainManager* loads the *DomainProfile.*
7. The *DomainManager* locates the startup procedure in the *DomainProfile*.
8. The *DomainManager* loads and executes *Factories* to create core *Resources* (*Logger*, *Factories*, etc.) base on the startup procedure in the *DomainProfile*.
9. The *DomainManager* loads and executes *Factories* on processor(s) to create *ResourceManager(s)* based on the startup procedure in the *DomainProfile*.
10. The *DomainManager* gets the available *Resource(s)* from the *ResourceManager(s).*
11. The *DomainManager* re-initializes the waveform application(s) run at power down.
12. Notification is given to the actor.
13. The *DomainManager* loads and executes *Factories* on processor(s) to create *FileManager(s)* based on the startup procedure in the *DomainProfile*.
14. The *DomainManager* loads and executes *Factories* on processor(s) to create *FileSystem(s)* based on the startup procedure in the *DomainProfile*.
15. The *DomainManager* maps the *FileSystem(s)* to *FileManager(s)* based on the startup procedure in the Domain Profile.

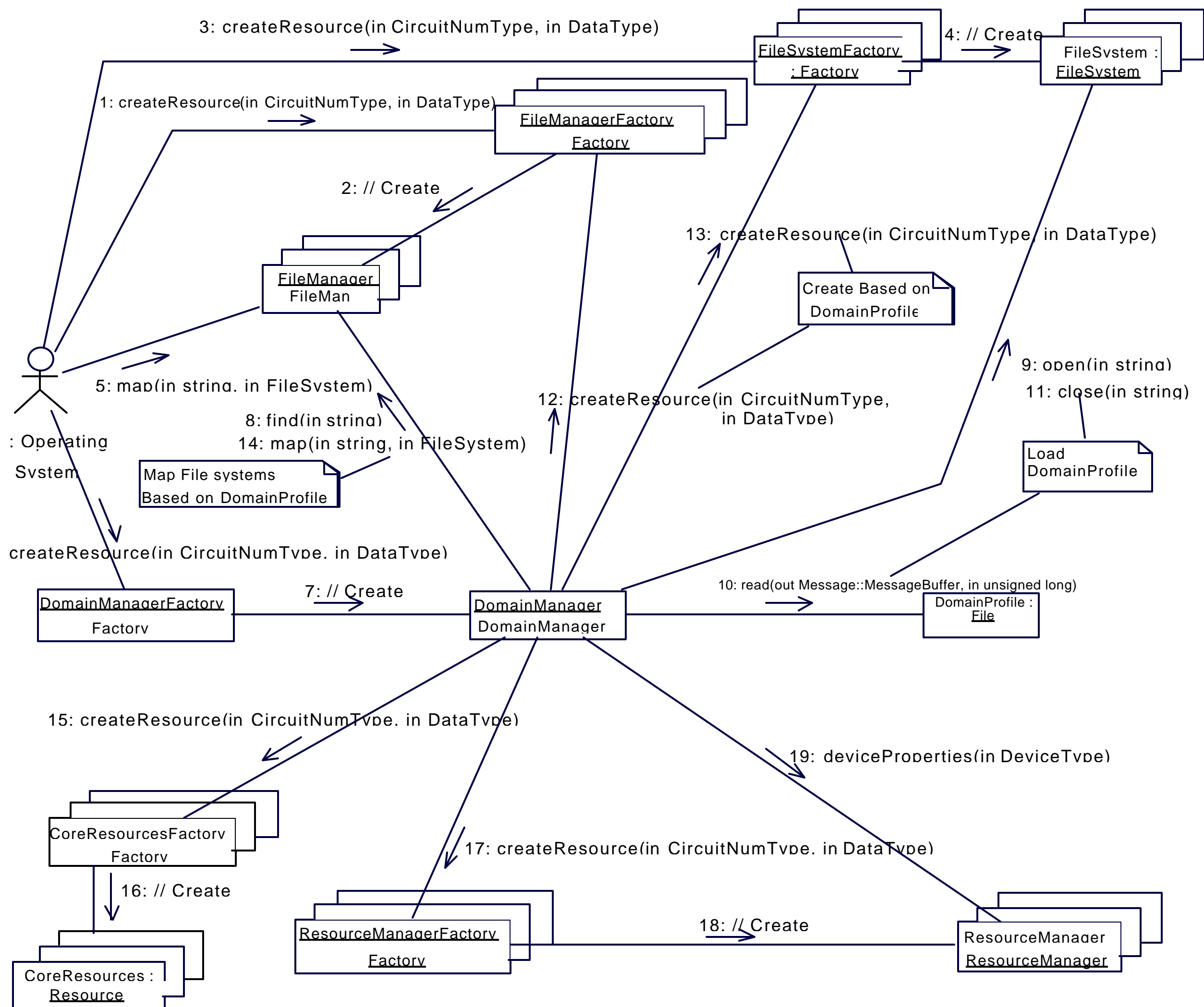


Figure 5.2.8.1.1-1. CF Power Up and Initialization Example Scenario

5.2.8.2. Send and Receive Communication Traffic Use Case

5.2.8.2.1 Receive Communications Scenario

This scenario provides an example of receiving communications traffic. The flow of events is described below and depicted in Figure 5.2.8.2.1-1. The scenario begins after a radio has been powered on and configured to receive. The virtual circuit *Message* paths have been registered. The "**R**" labeled flows denote RF signals. The "**F**" labeled flows denote Frequency Control flows. The "**B**" labeled flows denote Baseband data flows. The "**C**" labeled flows denote Clock recovery flows. **N**" labeled flows denote Network configuration flows.

1. (**R1**) RF signal is received by the antenna and sent to the LNA.

2. (**R2**) LNA amplifies RF signal.
3. (**R3**) Received signal is filtered to reject out-of-band energy.
4. (**F1, F1.1, F1.2**) The synthesizer (part of RF resource) is commanded by waveform control to tune to a user-selected frequency.
5. (**R4**) Signal strength of received signal is adjusted by AGC process.
6. (**R5**) Using the reference frequency provided by the synthesizer,
   (**B1**) the received RF signal is down converted to baseband.
7. (**B2**) Using an error estimate provided by the demodulator, the baseband signal is corrected for any frequency errors.
8. (**B3**) The corrected baseband signal is then demodulated, producing a stream of "raw" data bits.
9. (F2, F2.1) The demodulator also updates the frequency error estimate and the AGC setting at this time.
10. (**C1**) Additionally, the demodulator monitors transitions in the bit stream and provides an indication of these events to the clock generator.
11. (**C1.1, C1.2**) The clock recovery object generates a "receive bit clock" signal to be used as a reference by other receive objects. This clock signal is adjusted regularly to keep it aligned with the "bit transition" indications provided to it by the demodulator.
12. (**B4**) Based on comparisons with expected bit patterns (e.g., a waveform dependant data preamble pattern) the "raw" data stream is converted to a "hard" bit stream (if necessary).
13. (**B5**) The received data stream is then parsed into messages to be decrypted.
14. (**B6**) The Security module decrypts the message and presents the decrypted data to the red-side processor.
15. (**B7**) If the message is user data (such as voice data) it is sent to the voice decoder object.
16. (**N1, N1.1, N1.2**) If the decrypted data is radio configuration data (Op mode change, frequency change, etc.) the new configuration data is sent to the black-side configuration management controller. (Radio configuration is a separate sequence diagram.)
17. (**B7.1**) Finally, the decoded voice data is converted to an analog signal and routed to the enduser.

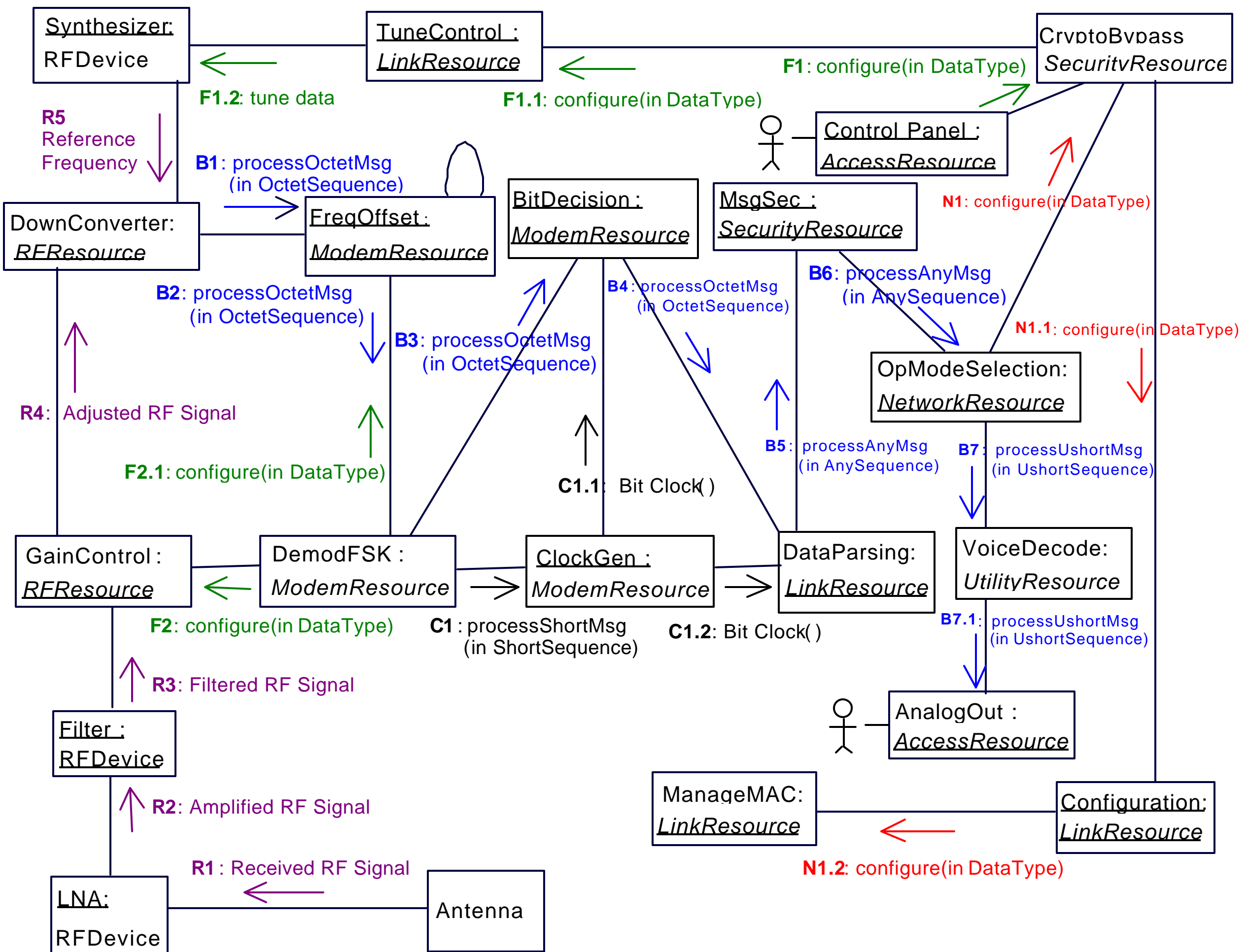


Figure 5.2.8.2.1-1. Receive Communications Example Scenario

Transmit Communications Scenario

This scenario provides an example of transmitting communications traffic. The flow of events is described below and depicted in Figure 5.2.8.2.2-1. The scenario begins after a radio has been powered on and configured to transmit.

1. User indicates that a transmission event is desired. (ex; PTT)
2. The input data (in this example analog voice) is digitized.
3. The digitized voice data is encoded following waveform specific rules. (in this example called "CVSD")
4. The encoded voice message is encrypted.
5. The encrypted data packet is sent over to the black-side where it is processed for transmission. (Interleaving, P-N cover, appended to header information, etc.)
6. The process of preparing the "front-end" of the radio for transmission is then started.

7. The amplifier is enabled.
8. The up-converter is enabled.
9. The packet of data to be transmitted is then passed to the modulator.
10. The modulator translates the data into a base-band signal.
11. Using a signal provided by the carrier generator the base-band signal is up-converted to the transmit frequency.
12. The modulated RF signal is amplified and transmitted.
13. The up-converter is disabled.
14. The amplifier is disabled.
15. Preparation for the next transmission event is begun.

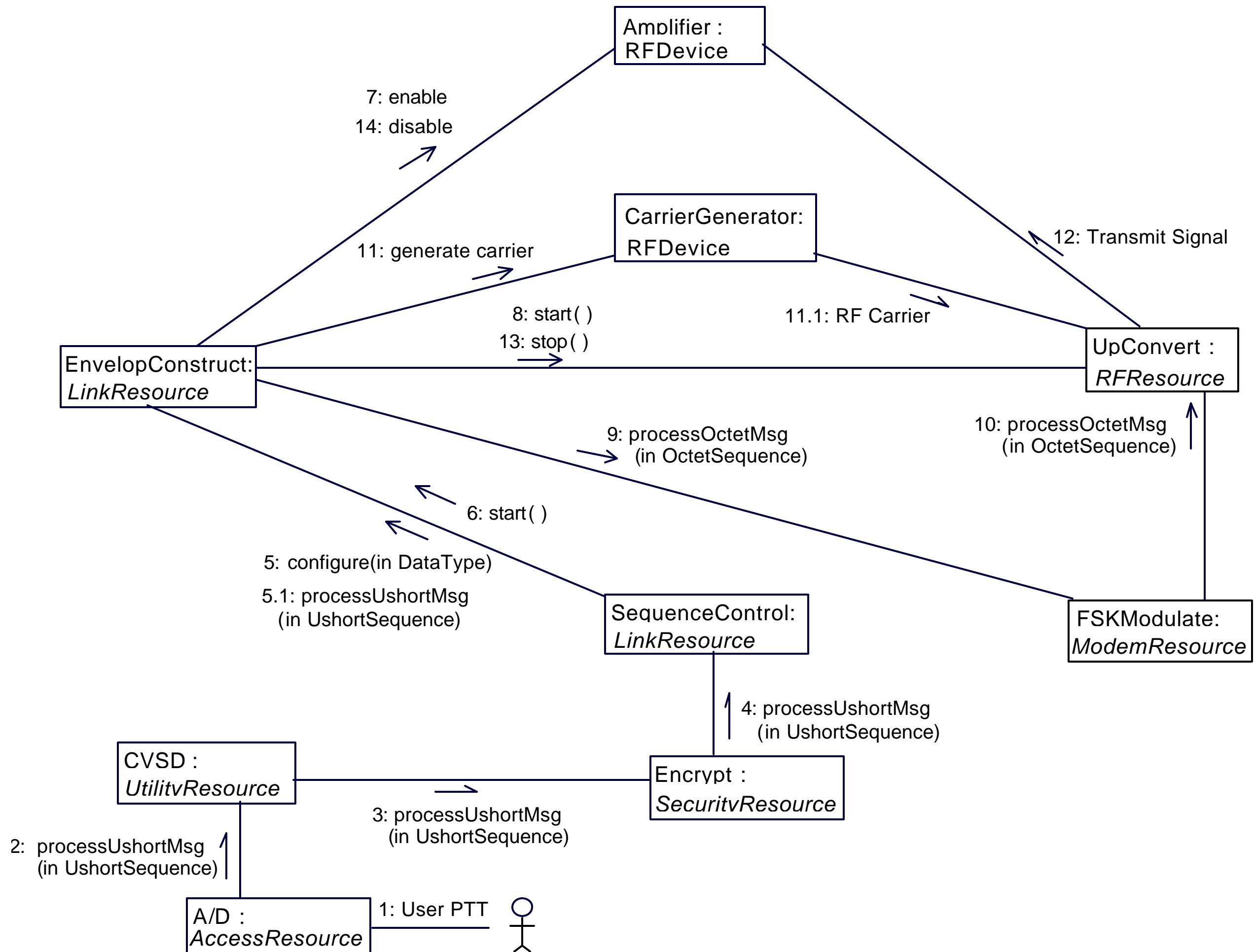


Figure 5.2.8.2.2-1. Transmit Communications Example Scenario

5.2.9 Summary of Core Framework Operations

The following tables provide a summary description of the operations included in the SDR Core Framework (CF).

LifeCycle Interface:

#	Operations	Purpose
1	selfTest	This operation performs a specific test on an object.
2	configure	This operation sets an object's properties.
3	query	This operation retrieves an object's properties.
4	initialize	This operation controls when configuration data is implemented by the resource or initializes the devices being controlled by the resource.
5	release	This operation releases the resource from the CORBA ORB. When the object's ORB reference count goes go to zero, the object's destructor operation will be called.
6	start	This operation starts processing messages that are received from the front end and/or back end of the radio. The object's sink (consumer) objects are enabled for processing messages.
7	stop	This operation stops processing messages that are received from the front end and/or back end of the radio. The object's sink (consumer) objects are disabled from processing messages and the messages are discarded.
8	pause	This operation queues messages that are received from the front end and/or back end of the radio.

StateManagement Interface:

#	Operations	Purpose
9	setAdminState	This operation sets the administrative state of an object managed by a *Resource*. The Administrative state of the object may be set to "Locked" (unusable for service) or "Unlocked" (usable for service) using this operation. (Note: If a request to Lock a resource is made while the resource is busy providing user service (i.e. Usage state is Active), the resource will transition to the "Shutting Down" state. When the user service is completed (i.e. Usage state is Idle) the resource will transition to the requested "Locked" Admin state.
10	getState	This operation returns an object's state. The Administrative (Locked, Unlocked, Shutting Down), Usage (Active, Idle), and Operational

		(Enabled, Disabled) states of an object managed by a *Resource* are returned.

MessageRegistration Interface:

#	Operations	Purpose
11	setSink	This operation registers a single *Message* sink (Consumer) object for call back by a source (Producer) object.
12	unsetSink	This operation removes a registered *Message* sink (Consumer) resource from a source (Producer) object's registered Message Sinks.
13	setMultipleSinks	This operation registers a set of *Message* sink (Consumer) objects for call back by a source (Producer) object.
14	getSink	This operation requests the *Message* sink (Consumer) object reference that is responsible for processing data to be received from the requesting source (Producer) object.
15	getTransferSize	This operation gets the maximum transfer *Message* size.
16	setTransferSize	This operation sets the suggested transfer *Message* size for the source (Producer) object.

Message Interface:

#	Operations	Purpose
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32	processOctetMsg processWcharMsg processLongMsg processShortMsg processLongLongMsg processULongMsg processULongLongMsg processFloatMsg processDoubleMsg processLongDoubleMsg processBooleanMsg processCharMsg processUshortMsg processStringMsg processWstringMsg processAnyMsg	These operations are used to push information to be received or transmitted through the radio from a source object (Producer) to one or more registered destination objects (Consumers). The destination objects are registered using the operations of the *MessageRegistration* interface. The data format is determined by the operation. All of the basic CORBA IDL types are supported.

DomainManager Interface:

#	**Operations**	**Purpose**
33	registerResourceManager	This operation adds a *ResourceManager* object entry into the *DomainManager* Domain Profile. This allows the *DomainManager* to dynamically direct the *ResourceManager* to load and execute software applications.
34	registerDevice	This operation adds a device entry into the *DomainManager* for a specific *ResourceManager* object.
35	unregisterResourceManager	This operation unregisters a *ResourceManager* object from the *DomainManager.*
36	unregisterDevice	The unregisterDevice operation removes a device entry from the *DomainManager* object for a specific *ResourceManager*.
37	getResources	This operation returns resources based upon the input resource request.
38	getDevices	This operation returns device information based upon the input device request.
39	getNetworks	This operation returns network information based upon the input network request.
40	createVirtualCircuit	This operation creates a virtual circuit for an application within the radio.
41	getVirtualCircuitResource	This operation returns the object reference for the specified virtual circuit.
42	releaseVirtualCircuit	This operation changes the state of the associated device entries in the *DomainManager* to be available and releases the application resources in the radio for this virtual circuit number.
43	fileManager	This operation returns a *FileManager* object reference to the main *FileManager* repository.

ResourceManager Interface:

#	Operations	Purpose
44	load	The load operation loads a file based on the given fileName using the input *FileSystem* to retrieve it.
45	unload	This operation unloads software based on the fileName.
46	execute	This operation executes the given function name using the arguments that have been passed in and returns an ID of the process that has been created.
47	terminate	This operation terminates the execution of the function on the device the *ResourceManager* is managing.
48	fileManager	This operation returns the *FileManager* object associated with this *ResourceManager*.
49	deviceProperties	This operation returns the properties for the specified device.
50	deviceExists	This operation returns the number of registered devices based upon the input type.
51	list	This operation provides a list of the hardware devices along with their properties that are currently associated with this *ResourceManager* object.
52	logger	This operation returns the Logger object associated with this *ResourceManager*.

File Interface:

#	Operations	Purpose
53	read	This operation reads data from a file.
54	write	This operation writes data to a file.
55	sizeOf	The sizeOf operation returns the current size of a file.

FileSystem Interface:

#	Operations	Purpose
56	remove	This operation provides a method of removing a file from the local file system.
57	copy	This operation provides a method of copying a file from the local file system to a remote file system.
58	exists	This operation provides a method of determining if a particular file exists in the file system.
59	list	This operation provides a list of the files contained in the file system.
60	load	This operation provides a method for the loading of a file system resident executable file into dynamic memory for subsequent execution.
61	unload	This operation provides a method for the unloading of an executable file from dynamic memory.
62	create	This operation creates a new on the local file system.
63	open	This operation opens a file on the local file system.
64	close	This operation closes a *File* server object that has been created and registered with the ORB.

FileManager Interface:

#	Operations	Purpose
65	list	This operation returns a list for *FileSystem* object references.
66	map	This operation registers a *FileSystem* object with the *FileManager* object.
67	unmap	This operation removes *FileSystem* reference from a *FileManager* object.
68	find	This operation returns a list of files that are found based upon the input criteria.
69	open	This operation opens a file.

Logger Interface:

#	Operations	Purpose
70	logData	This operation logs a log string and a time stamp to the console depending on the current log level set for the producer object and the log level of the string. It also logs the same information to a file if file logging is enabled for the object. The operation also pushes the data to registered consumers based upon their log levels. The logger log level is automatically assigned to a new producer.
71	setLoggingState	This operation enables the logging of all messages at the currently set level for each object, or disables the logging of all messages from all objects, depending on the value of the argument.
72	setProducerLogLevel	This operation sets the log level for a producer object. All incoming log strings <= to the currently set level are displayed/saved.
73	setConsumerLogLevel	This operation sets the log level for a consumer object. All incoming log strings <= to the currently set level are sent to the consumer.
74	displayLast	This operation displays at the console the last number of log messages stored locally within the logger.
75	registerConsumer	This operation registers a consumer object with the logger. Initially all producers' messages that pass the input logLevel are pushed to the consumer. A consumer can change its filtering by the setConsumerLogLevel operation.
76	unregisterConsumer	This operation unregisters a consumer object.
77	showProducerLogLevels	This operation displays the current log level for all producer objects.
78	showConsumerLogLevels	This operation displays the current log level for all consumer objects.
79	enableFileLogging	This operation stores to disk the incoming log based on the current log level.
80	disableFileLogging	This operation disables storage to disk of the incoming log based on the current debug level.
81	retrieveLogFile	This operation retrieves the current log file.

Factory Interface:

#	Operations	Purpose
82	createResource	This operation returns a *Resource* based upon the input resource number and qualifiers. If the *Resource* does not already exist then this operation creates the *Resource*, else the operation returns the object already created for that resource number.
83	releaseResource	This operation removes the *Resource* from the Factory if no other clients are using the *Resource*.
84	shutdown	This operation shuts down the *Factory* and destroys all *Resource*(s) that are being maintained by the *Factory*.

5.2.10 Core Framework IDL

The CF interfaces are expressed in CORBA IDL. The IDL has been generated directly by the Rational Rose UML software modeling tool. This "forward engineering" approach ensures that the IDL accurately reflects the architecture definition as contained in the UML models. Any IDL compiler for the target language of choice may compile the generated IDL. All of the interfaces are contained in the CF CORBA module as depicted in Figure 5.2.10-1. This module is also provided separately in electronic form. A listing of the Core Framework IDL is provided on the following pages.

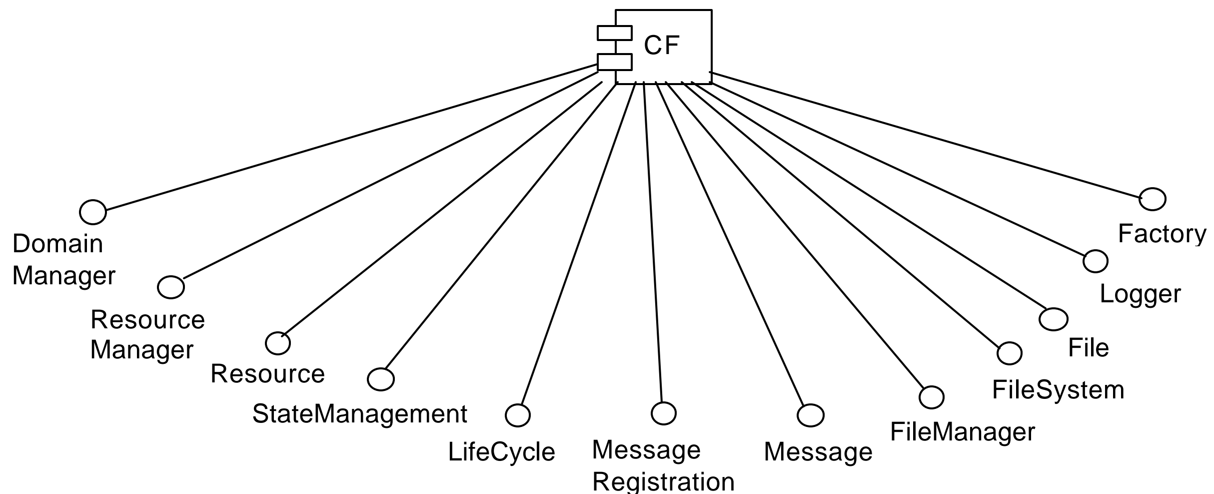


Figure 5.2.10-1. CF CORBA module

5.2.10.1 Core Framework IDL Listing

The following is a complete list of the CF IDL generated from the Rational Rose model.

```
//## Module: CF
//## Subsystem:  CF_IDL_Implementation_Component
//## Source file:  C:\projects\sdr\CF.idl
//## Documentation::
//      This CORBA module defines the SDR CF interfaces and types.
//##begin module.cm preserve=no
//    %X% %Q% %Z% %W%
//##end module.cm

//##begin module.cp preserve=no
//##end module.cp

#ifndef CF_idl
#define CF_idl

//##begin module.additionalIncludes preserve=no
//##end module.additionalIncludes

//##begin module.includes preserve=yes
//##end module.includes

// ===================================================================

interface Resource;

//## DataType Documentation:
//      This type is a CORBA  IDL struct type which can be
//      used to hold any CORBA basic type or static IDL type.
//## Category: CF_IDL_Design_Components

struct DataType {
  //##begin DataType.initialDeclarations preserve=yes
  //##end DataType.initialDeclarations

  // Attributes

  //## Attribute: id
  //## Documentation:
  //      The id attribute indicates the kind of value (e.g.,
  //      frequency, preset, etc.).
  unsigned long id;
  //## Attribute: value
  //## Documentation:
  //      The value attribute can be any static IDL type or
  //      CORBA basic type.
  any value;

  // Relationships
```

```
  // Associations

  //##begin DataType.additionalDeclarations preserve=yes
  //##end DataType.additionalDeclarations

};

//## Properties Documentation:
//      The Properties is a CORBA IDL unbounded sequence of
//      DataType(s), which can be used in defining a sequence
//      of name and value pairs.  The relationships for Properties
//      are shown in the Properties Relationships figure.
//## Category: CF_IDL_Design_Components

typedef sequence <DataType> Properties;

//## CircuitNumType Documentation:
//      This class defines the circuit number values within the radio.
//## Category: CF_IDL_Design_Components

typedef unsigned short CircuitNumType;

//## DirectionType Documentation:
//      This CORBA IDL enumeration type defines the data direction
//      within the radio.
//## Category: CF_IDL_Design_Components

enum DirectionType
{
  FROM_ANTENNA,
  TO_ANTENNA
};

//## ProcessID_Type Documentation:
//      This defines the process number within the radio.
//      Processor number is qualified by the Processor OS
//      that created the process.
//## Category: CF_IDL_Design_Components

typedef unsigned long ProcessID_Type;

//## DeviceNumType Documentation:
//      This type defines the device number values.
//## Category: CF_IDL_Design_Components

typedef unsigned long DeviceNumType;

//## ClassID_Type Documentation:
//      This type defines the device kind number.
//## Category: CF_IDL_Design_Components

typedef unsigned long ClassID_Type;

//## DeviceID_Type Documentation:
```

```
//      This type defines the device identification values.
//## Category: CF_IDL_Design_Components

typedef unsigned long DeviceID_Type;

//## DeviceType Documentation:
//      This CORBA IDL struct type defines the actual physical
//      hardware device.which contains an identification, number,
//      and device identification as shown in the DeviceType
//      Relationships figure.
//## Category: CF_IDL_Design_Components

struct DeviceType {
  //##begin DeviceType.initialDeclarations preserve=yes
  //##end DeviceType.initialDeclarations

  // Attributes

  //## Attribute: classID
  //## Documentation:
  //      This attribute indicates the type of physical device.

  ClassID_Type classID;

  //## Attribute: element
  //## Documentation:
  //      This attribute indicates the  number of the device kind.

  DeviceNumType element;

  //## Attribute: deviceID
  //## Documentation:
  //      This attribute identifies the device within the radio.

  DeviceID_Type deviceID;

  // Relationships

  // Associations

  //##begin DeviceType.additionalDeclarations preserve=yes
  //##end DeviceType.additionalDeclarations

};

//## StringSequence Documentation:
//      This type defines a sequence of strings
//## Category: CF_IDL_Design_Components

typedef sequence <string> StringSequence;

//## DevicePropertiesType Documentation:
//      This type contains the properties for a specific type.
//## Category: CF_IDL_Design_Components
```

```
struct DevicePropertiesType {
  //##begin DevicePropertiesType.initialDeclarations preserve=yes
  //##end DevicePropertiesType.initialDeclarations

  // Attributes

  DeviceType device;
  Properties properties;

  // Relationships

  // Associations

  //##begin DevicePropertiesType.additionalDeclarations preserve=yes
  //##end DevicePropertiesType.additionalDeclarations

};

//## DeviceList Documentation:
//      This type defines an unbounded CORBA IDL sequence of
//      DeviceType as shown in the DeviceList Relationships figure.
//## Category: CF_IDL_Design_Components

typedef sequence<DevicePropertiesType> DeviceList;

//## Message Documentation:
//      The Message  interface provides operations for pushing data
//      to a consumer Message object. This interface is implemented by
//      a consumer that processes messages pushed to it. The relationships
//      for Message are shown in the Message Raltionships figure.
//## Category: CF_IDL_Design_Components

interface Message {
  //##begin Message.initialDeclarations preserve=yes
  //##end Message.initialDeclarations

  // Nested Classes
  //## OctetSequence Documentation:
  //      This type is a CORBA unbounded sequence of octets.

  typedef sequence<octet> OctetSequence;

  //## CharSequence Documentation:
  //      This type is a CORBA unbounded sequence of characters.

  typedef sequence<char> CharSequence;

  //## ShortSequence Documentation:
  //      This type is a CORBA unbounded sequence of short integers.

  typedef sequence<short> ShortSequence;

  //## LongSequence Documentation:
```

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

  typedef sequence<long> LongSequence;

  //## LongLongSequence Documentation:
  //      This type is a CORBA unbounded sequence of long long integers.

  typedef sequence<long long> LongLongSequence;

  //## UshortSequence Documentation:
  //      This type is a CORBA unbounded sequence of unsigned short integers.

  typedef sequence<unsigned short> UshortSequence;

  //## UlongSequence Documentation:
  //      This type is a CORBA unbounded sequence of unsigned long integers.

  typedef sequence<unsigned long> UlongSequence;

  //## UlongLongSequence Documentation:
  //      This type is a CORBA unbounded sequence of unsigned longlong
integers.

  typedef sequence<unsigned long long> UlongLongSequence;

  //## FloatSequence Documentation:
  //      This type is a CORBA unbounded sequence of floats.

  typedef sequence<float> FloatSequence;

  //## DoubleSequence Documentation:
  //      This type is a CORBA unbounded sequence of doubles.

  typedef sequence<double> DoubleSequence;

  //## LongDoubleSequence Documentation:
  //      This type is a CORBA unbounded sequence of long doubles.

  typedef sequence<long double> LongDoubleSequence;

  //## BooleanSequence Documentation:
  //      This type is a CORBA unbounded sequence of booleans.

  typedef sequence<boolean> BooleanSequence;

  //## WcharSequence Documentation:
  //      This type is a CORBA unbounded sequence of wide characters.

  typedef sequence<wchar> WcharSequence;

  //## StringSequence Documentation:
  //      This type is a CORBA unbounded sequence of strings.

  typedef sequence<string> StringSequence;
```

```
  //## WstringSequence Documentation:
  //      This type is a CORBA unbounded sequence of wide strings.

  typedef sequence<wstring> WstringSequence;

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: processOctetMsg
  //## Documentation:
  //      This operation is used to push a sequence of octets.

  oneway void processOctetMsg(in OctetSequence message, in Properties
options);

  //## Operation: processWcharMsg
  //## Documentation:
  //      This operation is used to push a sequence of wide characters.

  oneway void processWcharMsg(in WcharSequence message, in Properties
options);

  //## Operation: processLongMsg
  //## Documentation:
  //      This operation is used to push a sequence of long integers.

  oneway void processLongMsg(in LongSequence message, in Properties options);

  //## Operation: processShortMsg
  //## Documentation:
  //      This operation is used to push a sequence of short integers.

  oneway void processShortMsg(in ShortSequence message, in Properties
options);

  //## Operation: processLongLongMsg
  //## Documentation:
  //      This operation is used to push a sequence of long long integers.

  oneway void processLongLongMsg(in LongLongSequence message, in Properties
                                options);

  //## Operation: processUlongMsg
  //## Documentation:
  //      This operation is used to push a sequence of unsigned long integers.

  oneway void processUlongMsg(in UlongSequence message, in Properties
options);
```

```
  //## Operation: processULongLongMsg
  //## Documentation:
  //      This operation is used to push a sequence of unsigned long long
  //      integers.

  oneway void processULongLongMsg(in UlongLongSequence message, in Properties
                                  options);

  //## Operation: processFloatMsg
  //## Documentation:
  //      This operation is used to push a sequence of floats.

  oneway void processFloatMsg(in FloatSequence message, in Properties
options);

  //## Operation: processDoubleMsg
  //## Documentation:
  //      This operation is used to push a sequence of doubles.

  oneway void processDoubleMsg(in DoubleSequence message, in Properties
                               options);

  //## Operation: processLongDoubleMsg
  //## Documentation:
  //      This operation is used to push a sequence of long doubles.

  oneway void processLongDoubleMsg(in LongDoubleSequence message, in
                                   Properties options);

  //## Operation: processBooleanMsg
  //## Documentation:
  //      This operation is used to push a sequence of booleans.

  oneway void processBooleanMsg(in BooleanSequence message, in Properties
                                options);

  //## Operation: processCharMsg
  //## Documentation:
  //      This operation is used to push a sequence of characters.

  oneway void processCharMsg(in CharSequence message, in Properties options);

  //## Operation: processUshortMsg
  //## Documentation:
  //      This operation is used to push a sequence of unsigned short
integers.

  oneway void processUshortMsg(in UshortSequence message, in Properties
                               options);

  //## Operation: processStringMsg
  //## Documentation:
  //      This operation is used to push a CORBA string
```

```
  oneway void processStringMsg(in StringSequence message, in Properties
                              options);

  //## Operation: processWstringMsg
  //## Documentation:
  //      This operation is used to push a CORBA wide string

  void processWstringMsg(in WstringSequence message, in Properties options);

  //## Operation: processAnyMsg
  //## Documentation:
  //      This operation is used to push a CORBA any type

  oneway void processAnyMsg(in DataType message, in Properties options);

  //##begin Message.additionalDeclarations preserve=yes
  //##end Message.additionalDeclarations

};

//## File Documentation:
//      The File interface defines the CORBA interfaces for manipulating
//      a file within the radio.  The relationships for File are shown in
//      the File Relationships figure.  The File interface emulates the
//      POSIX/C file interface.
//## Category: CF_IDL_Design_Components

interface File {
  //##begin File.initialDeclarations preserve=yes
  //##end File.initialDeclarations

  // Attributes

  //## Attribute: fileName
  //## Documentation:
  //      Theis attribute provides read access to the fully qualified
  //      name of the file.

  readonly attribute string fileName;

  // Relationships

  // Associations

  // Operations

  //## Operation: read
  //## Documentation:
  //      The read operation reads data from the file.  The read operation
  //      returns a True value if the read was successful, otherwise False
  //      is returned.
```

```
  unsigned long read(out Message::OctetSequence data, in unsigned long
length);

  //## Operation: write
  //## Documentation:
  //      The write operation writes data to the file.  The write operation
  //      returns a True value if the write was successful, otherwise False
  //      is returned.

  unsigned long write(in Message::OctetSequence data, in unsigned long
length);

  //## Operation: sizeOf
  //## Documentation:
  //      The sizeOf operation returns the current size of the file.

  unsigned long sizeOf();

  //##begin File.additionalDeclarations preserve=yes
  //##end File.additionalDeclarations

};

//## FileSystem Documentation:
//      The FileSystem interface defines the CORBA interfaces that
//      provide the file I/O manipulation operations for a file system.
//      The FileMan interface provides the flexibility of having
//      multiple file systems within the radio, and of being located
//      anywhere within the radio.  The relationships for FileSystem
//      are shown in the File System Relationships figure.
//## Category: CF_IDL_Design_Components

interface FileSystem {
  //##begin FileSystem.initialDeclarations preserve=yes
  //##end FileSystem.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: remove
  //## Documentation:
  //      The remove operation removes the file with the given name
  //      from the file system.  The name includes the full path of the
  //      file. The operation returns true on success, false on fail.
  boolean remove(in string fileName);

  //## Operation: copy
  //## Documentation:
  //      The copy operation copies the source file with the specified
```

```
//      name to the destination FileSystem.  The copy operation
//      returns true on success, false on fail.
boolean copy(in string sourceFileName, in string destinationFileName, in
             FileSystem destinationFileSystem);

//## Operation: exists
//## Documentation:
//      The exists operation checks to see if a file exists based on
//      the file name parameter and returns true if found, false
//      otherwise.  The file name should include the path where to
//      search for the file.

boolean exists(in string fileName);

//## Operation: list
//## Documentation:
//      The list operation behaves similar to the UNIX "ls" command.

StringSequence list(in string name, in string argv, in short argc);

//## Operation: load
//## Documentation:
//      The load operation loads a file based on the file Name and
//      returns a success or failure status.  The load allows a file
//      in the file system to be loaded into RAM without having to
//      open a file and read the file to load  the file into RAM.

boolean load(in string fileName);

//## Operation: create
//## Documentation:
//      The create operation creates a new File based upon the input
//      file name.  The size is used to determine if the file system
//      has enough space for creating the new file and to verify the
//      file size when closing the file.  A null file object reference
//      is returned if the name already exists or size is too large
//      for the file system.
File create(in string fileName, in unsigned long size);

//## Operation: open
//## Documentation:
//      The open operation opens a File based upon the input file
//      name.  A null File object reference is returned if name does
//      not exist in the file system.

File open(in string fileName);

//## Operation: close
//## Documentation:
//      The close operation releases a File  object that has been
//      created and registered with the ORB.   A True value is
//      returned upon successful file close, otherwise False is returned.

boolean close(in string fileName);
```

```
  //## Operation: unload
  //## Documentation:
  //      The unload operation unloads a file based on the fileName
  //      and returns a success or failure status.  The unload
  //      operation unloads the software from RAM.
  boolean unload(in string fileName);

  //##begin FileSystem.additionalDeclarations preserve=yes
  //##end FileSystem.additionalDeclarations

};

//## Logger Documentation:
//      The Logger interface is used to capture alarms, log warnings and
//      information messages during the execution of software withinn the
//      radio, and pushes messages to registered consumers.  The interface
//      provides operations for both producer and consumer clients.
//## Category: CF_IDL_Design_Components

interface Logger {
  //##begin Logger.initialDeclarations preserve=yes
  //##end Logger.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: logData
  //## Documentation:
  //      This operation logs a log string and a time stamp to the console
  //      depending on the current log level set for the producer object
  //      and the log level of the string.  It also logs the same
  //      information to a file if file logging is enabled for the object.
  //      The operation also pushes the data to registered consumers based
  //      upon their log levels. The logger log level is automatically
  //      assigned to a new producer.

  oneway void logData(in string producerName, in string messageString, in
                      unsigned short logLevel);

  //## Operation: setLoggingState
  //## Documentation:
  //      This operation enables the logging of all messages at the
  //      currently set level for each object, or disables the logging
  //      of all messages from all objects, depending on the value of
  //      the argument.

  void setLoggingState(in boolean enable);
```

```
  //## Operation: setProducerLogLevel
  //## Documentation:
  //      This operation sets the log level for a producer object.  All
  //      incoming log strings <= to the currently set level are
  //      displayed/saved. The log level is bitmapped 00 00 - 7F FF(hex)
  //      with bit 16 being a control bit to allow for log level manipulation.

  //      Examples:
  //      LogLevel = C010 h (1100 0000 0001 0000 b) indicates
  //      only levels 14 and 4 are to be displayed.
  //      LogLevel = 000A h indicates levels 10 and below
  //      will be displayed, and bits 4-14 are unused.

  void setProducerLogLevel(in string producerName, in unsigned short
logLevel);

  //## Operation: setConsumerLogLevel
  //## Documentation:
  //      This operation sets the log level for a consumer object.  All
  //      All incoming log strings <= to the currently set level are sent
  //      to the consumer. The log level is bitmaped 00 00 - 7F FF (hex)
  //      with bit 16 being a control bit to allow for log level manipulation.

  //      Examples:
  //      LogLevel = C010 h (1100 0000 0001 0000 b) indicates
  //      only levels 14 and 4 are to be displayed.
  //      LogLevel = 000A h indicates levels 10 and below
  //      will be displayed, and bits 4-14 are unused.

  void setConsumerLogLevel(in string consumerName, in string producerName, in
                           unsigned short logLevel);

  //## Operation: displayLast
  //## Documentation:
  //      This operation displays at the console the last number of log
  //      messages stored locally within the logger.

  void displayLast(in unsigned short number);

  //## Operation: registerConsumer
  //## Documentation:
  //      This operation registers a consumer object with the logger.
  //      Initially all producers' messages that pass the input logLevel
  //      are pushed to the consumer.  A consumer can change its filtering
  //      by the setConsumerLogLevel operation.

  void registerConsumer(in string consumerName, in Message consumerMessage, in
                        unsigned short logLevel);

  //## Operation: unregisterConsumer
  //## Documentation:
  //      This operation unregisters a consumer object.

  void unregisterConsumer(in string consumerName);
```

```
  //## Operation: showProducerLogLevels
  //## Documentation:
  //      This operation displays the current log level for all producer
  //      objects.

  void showProducerLogLevels();

  //## Operation: showConsumerLogLevels
  //## Documentation:
  //      This operation displays the current log levels for a consumer
object.

  void showConsumerLogLevels(in string consumerName);

  //## Operation: enableFileLogging
  //## Documentation:
  //      This operation stores to disk the incoming log based on the current
  //      log level.  It does not affect output to the console.

  void enableFileLogging(in string filename, in FileSystem fileSystem);

  //## Operation: disableFileLogging
  //## Documentation:
  //      This operation disables storage to disk of the incoming log based
  //      on the current debug level.

  void disableFileLogging();

  //## Operation: retrieveLogFile
  //## Documentation:
  //      This operation retrieves the current log file.

  File retrieveLogFile();

  //##begin Logger.additionalDeclarations preserve=yes
  //##end Logger.additionalDeclarations

};

//## FileManager Documentation:
//      The FileMan interface provides the operations for manipulating a
//      File Manager object.  A File Manager object contains a set of
//      File System object references.  The File Manager interface is
//      is similar to COTS OS file managers (UNIX, NT) capabilities.  The
//      relationships for FileMan are shown in the File Manager
//      Relatinships figure.
//## Category: CF_IDL_Design_Components

interface FileManager {
  //##begin FileManager.initialDeclarations preserve=yes
  //##end FileManager.initialDeclarations

  // Attributes
```

```
  // Relationships

  // Associations

  // Operations

  //## Operation: list
  //## Documentation:
  //      The list operation returns a list for FileSystem object references.
  Properties list();

  //## Operation: map
  //## Documentation:
  //      The map operation registers a FileSystem object with the
  //      File Manager object.  True is returned if the mapping was
  //      successful, otherwise false is returned.

  boolean map(in string fileSystemName, in FileSystem fileSystem);

  //## Operation: unmap
  //## Documentation:
  //      The unmap operation removes FileSystem reference from a
  //      File Manager object.

  boolean unmap(in string fileSystemName);

  //## Operation: find
  //## Documentation:
  //      The find operation returns list of Files that are found based
  //      upon the input criteria.

  StringSequence find(in string name);

  //## Operation: open
  //## Documentation:
  //      This operation opens a file.

  File open(in string name);

  //##begin FileManager.additionalDeclarations preserve=yes
  //##end FileManager.additionalDeclarations

};

//## ResourceManager Documentation:
//      The ResourceManager interface defines the CORBA interfaces for
//      communicating with a device that is CORBA capable.  A Resource
//      Manager object dynamically receives load and execute requests.

//      A Resource Manager upon startup determines its local devices and
//      may create or obtain a Logger and FileSystem objects.  The
//      relatiuonships for this interface are shown in the ResourceManager
//      Relationships figure.
```

```
//## Category: CF_IDL_Design_Components

interface ResourceManager {
  //##begin ResourceManager.initialDeclarations preserve=yes
  //##end ResourceManager.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: terminate
  //## Documentation:
  //      The terminate operation terminates the execution of the function
  //      on the device the Resource Manager is managing and returns a
  //      True value when termination is successful or False if unsuccessful.

  boolean terminate(in ProcessID_Type processId);

  //## Operation: fileManager
  //## Documentation:
  //      The fileManager operation returns the File Manager associated
  //      with this Resource Manager.

  FileManager fileManager();

  //## Operation: logger
  //## Documentation:
  //      The logger operation returns the logger associated with this
  //      Resource Manager.

  Logger logger();

  //## Operation: deviceProperties
  //## Documentation:
  //      The deviceProperties capability returns the properties for the
  //      specified device.  If the specified device does not exist a null
  //      Properties set is returned.

  Properties deviceProperties(in DeviceType device);

  //## Operation: deviceExists
  //## Documentation:
  //      This operation returns the number of registered devices based upon
  //      the input type.

  unsigned long deviceExists(in DeviceType device);

  //## Operation: list
  //## Documentation:
  //      This operation provides a list of the hardware devices along
```

```
  //      with their properties that are currently associated with this
  //      Resource Manager object.

  DeviceList list();

  //## Operation: execute
  //## Documentation:
  //      The execute operation executes the given function name using the
  //      arguments that have been passed in and returns an ID of the process
  //      that has been created.

  ProcessID_Type execute(in string functionName, in StringSequence
parameters);

  //## Operation: load
  //## Documentation:
  //      The load operation loads a file based on the given filename using
  //      the input FileSystem to retrieve it.  True is returned if the load
  //      was successful, otherwise False is returned.

  boolean load(in FileSystem fileSystem, in string fileName);

  //## Operation: unload
  //## Documentation:
  //      The unload operation unloads software based on the fileName and
  //      returns a success or failure status.

  boolean unload(in string fileName);

  //##begin ResourceManager.additionalDeclarations preserve=yes
  //##end ResourceManager.additionalDeclarations

};

//## ResourceType Documentation:
//      This type is used to indicate a type of resource.
//## Category: CF_IDL_Design_Components

typedef unsigned long ResourceType;

//## Category: CF_IDL_Design_Components

typedef unsigned short ResourceNumType;

//## Factory Documentation:
//      The Factory class is the interface for all factories in the Radio.
//      The relationships for Factory are depicted in the Factory Raltionships
//      figure.

//      Each Factory object produces a specific resource within the radio.

//      The Factory Interface provides a one-step solution for creating a
//      resource, reducing the overhead of starting up resources.
```

```
//      The Factory Interface provides a one-step solution for releasing
//      resources, reducing the overhead of releasing resources.

//      The Factory Interface is similar to the COM factory class and is
//      based on the industry accepted Factory design pattern.
//## Category: CF_IDL_Design_Components

interface Factory {
  //##begin Factory.initialDeclarations preserve=yes
  //##end Factory.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: createResource
  //## Documentation:
  //      This operation returns a resource based upon the input resource
  //      number and qualifiers.  If the resource does not already exist
  //      then this operation creates the resource, else the operation
  //      returns the object already created for that resource number.

  Object createResource(in ResourceNumType resourceNumber, in DataType
                        qualifiers);

  //## Operation: releaseResource
  //## Documentation:
  //      This operation removes the resource from the Factory if no
  //      other clients are using the resource.  The resource to be
  //      released is associated with a specific resource number.

  boolean releaseResource(in ResourceNumType resourceNumber);

  //## Operation: shutdown
  //## Documentation:
  //      This operation destroys all resources managed by this factory
  //      and terminates the factory server.

  boolean shutdown();

  //##begin Factory.additionalDeclarations preserve=yes
  //##end Factory.additionalDeclarations

};

//## ResourceID_Type Documentation:
//      This type defines a CORBA IDL struct  type which contains a
//      direction, resource number, and resource type as shown in the
//      ResourceID_Type Relationships figure.
//## Category: CF_IDL_Design_Components
```

```
struct ResourceID_Type {
  //##begin ResourceID_Type.initialDeclarations preserve=yes
  //##end ResourceID_Type.initialDeclarations

  // Attributes

  //## Attribute: direction
  //## Documentation:
  //      This attribute indicates the direction the data is coming
  //      from (to the antenna or from the antenna).

  DirectionType direction;

  //## Attribute: number
  //## Documentation:
  //      This attribute indicates the  number of the resource sink
  //      Consumer) or source (Producer) object.

  ResourceNumType number;

  //## Attribute: resourceType
  //## Documentation:
  //      This attribute indicates the type of resource
  //      (e.g., modem, access, link, etc)
  ResourceType resourceType;

  // Relationships

  // Associations

  //##begin ResourceID_Type.additionalDeclarations preserve=yes
  //##end ResourceID_Type.additionalDeclarations

};

//## StateManagement Documentation:
//      The StateManagement interface defines the state information
//      that is based upon the ISO/IEC 10164-2 Open Systems
//      Interconnection - Systems Management: State Management Function
//      standard.  This standard identifies additional states which
//      could be used to expand the definition of the Resource states.
//## Category: CF_IDL_Design_Components

interface StateManagement {
  //##begin StateManagement.initialDeclarations preserve=yes
  //##end StateManagement.initialDeclarations

  // Nested Classes
  //## AdminType Documentation:
  //      This type is a CORBA IDL enumeratiuon type that defines an
  //      object's administrative states.

  enum AdminType
```

```
{
  ADMIN_NOT_APPLICABLE,
  LOCKED,
  SHUTTING_DOWN,
  UNLOCKED
};

//## UsageType Documentation:
//      This type is a CORBA IDL enumeration type that defines the
//      object's Usage states.

enum UsageType
{
  USAGE_NOT_APPLICABLE,
  IDLE,
  ACTIVE
};

//## OperationalType Documentation:
//      This type is a CORBA IDL enumeration type that defines an
//      object's Operational states.

enum OperationalType
{
  ENABLED,
  DISABLED
};

//## StateType Documentation:
//      This type is a CORBA IDL struct type that contains an object's
//      Admin, Operational, and Usage states.

struct StateType {
  //##begin StateType.initialDeclarations preserve=yes
  //##end StateType.initialDeclarations

  // Attributes

  AdminType adminState;
  OperationalType operationalState;
  UsageType usageState;

  // Relationships

  // Associations

  //##begin StateType.additionalDeclarations preserve=yes
  //##end StateType.additionalDeclarations

};

// Attributes

// Relationships
```

```
  // Associations

  // Operations

  //## Operation: setAdminState
  //## Documentation:
  //      This operation sets the adminstrative state per the specified
  //      parameter.

  void setAdminState(in AdminType adminState);

  //## Operation: getState
  //## Documentation:
  //      This operation returns the object's state.

  StateType getState();

  //##begin StateManagement.additionalDeclarations preserve=yes
  //##end StateManagement.additionalDeclarations

};

//## LifeCycle Documentation:
//      The LifeCycle interface defines the generic object operations
//      for: 1) Testing, 2) Configuring (setting) and querying
//      (retrieving) an object's properties, 3) Initializing and
//      releasing an object, and 4) Messaging control operations:
//      start, stop, and pause.

//      The parameter type for properties is based upon the CORBA "any"
//      type.  This provides the greatest flexibility for developing
//      software by leaving the implementation up to the developer not
//      by the core framework definition.  The CORBA any type is also
//      minimum CORBA compliant.
//## Category: CF_IDL_Design_Components

interface LifeCycle {
  //##begin LifeCycle.initialDeclarations preserve=yes
  //##end LifeCycle.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: selfTest
  //## Documentation:
  //      The selfTest operation performs a specific test on an object.
  //      True is returned if the test passes, otherwise false is
  //      returned.  When false is returned, the operation also returns
```

```
//      a reason why the test failed.

boolean selfTest(inout unsigned long testNum);

//## Operation: configure
//## Documentation:
//      The configure operation sets the object's properties.  True
//      is returned if the configure was successful, otherwise False
//      is returned.   Any basic CORBA type or static IDL type could
//      be used for the configuration data.  An object’s ICD indicates
//      the valid configuration values.

boolean configure(in DataType properties);

//## Operation: query
//## Documentation:
//      The query operation retrieves object's properties.  Any basic
//      CORBA type or static IDL type could be used for the query.
//      An object's ICD indicates the valid query types.  The
//      information retrieved can later be used when an object is
//      recreated, by calling the configure operation.

void query(inout DataType properties);

//## Operation: initialize
//## Documentation:
//      The intialize operation controls when configuration data is
//      implemented by the resource or initializes the devices being
//      controlled by the resource.

boolean initialize();

//## Operation: release
//## Documentation:
//      The release operation releases itself from the CORBA ORB.
//      When the object's  ORB reference count goes to zero, the
//      objects desctructor operation will be called.

boolean release();

//## Operation: start
//## Documentation:
//      The start operation starts processing messages that are
//      received from the front end and/or back end of the radio.
//      The object's sink objects are enabled for processing messages.

boolean start();

//## Operation: stop
//## Documentation:
//      The stop operation stops processing messages that are
//      received from the front end and/or back end of the radio.
//      The object's sink objects are disbaled from processing
//      messages and the messages are discarded.
```

```
  boolean stop();

  //## Operation: pause
  //## Documentation:
  //      The pause operation queues messages that are received from
  //      the front end and/or back end of the radio.

  boolean pause();

  //##begin LifeCycle.additionalDeclarations preserve=yes
  //##end LifeCycle.additionalDeclarations

};

//## ConfigurationStatusType Documentation:
//      This type indicates configuration status values.
//## Category: CF_Domain_Manager_IDL_Design_Components

enum ConfigurationStatusType
{
  Configure_Successful,
  Configure_Failure,
  Configure_NA
};

//## Category: CF_Domain_Manager_IDL_Design_Components

struct Network {
  //##begin Network.initialDeclarations preserve=yes
  //##end Network.initialDeclarations

  // Attributes

  string netType;
  ResourceNumType resourceNumber;
  string alias;
  Properties properties;

  // Relationships

  // Associations

  //##begin Network.additionalDeclarations preserve=yes
  //##end Network.additionalDeclarations

};

//## Category: CF_Domain_Manager_IDL_Design_Components

typedef sequence <Network> Networks;

//## Category: CF_Domain_Manager_IDL_Design_Components
```

```
struct ResourcePropertiesType {
  //##begin ResourcePropertiesType.initialDeclarations preserve=yes
  //##end ResourcePropertiesType.initialDeclarations

  // Attributes

  ResourceID_Type id;
  Properties properties;

  // Relationships

  // Associations

  //##begin ResourcePropertiesType.additionalDeclarations preserve=yes
  //##end ResourcePropertiesType.additionalDeclarations

};

//## Category: CF_Domain_Manager_IDL_Design_Components

typedef sequence <ResourcePropertiesType> Resources;

//## DestinationType Documentation:
//      The Destination type is a CORBA IDL struct type
//      which defines the attributes necessary for setting
//      up a virtual path to another resource in the radio.
//## Category: CF_IDL_Design_Components

struct DestinationType {
  //##begin DestinationType.initialDeclarations preserve=yes
  //##end DestinationType.initialDeclarations

  // Attributes

  //## Attribute: resource
  //## Documentation:
  //      The object attribute indicates the CORBA object that
  //      should be used for pushing data to it.

  //      When the destination is for an I/O physical device and the
  //      source is the red-side waveform object, then the source
  //      obtains the message object from the object using the getSink
  //      operation.  Otherwise the destination needs to be sent to
  //      the black-side waveform for setting up the virtual path.

  //      When the source and destination is for a MODEM and the source
  //      is compatible (e.g., mode, crypto algorithm and key) with the
  //      destination, then the destination information needs to be sent
  //      to the black-side to set up the virtual path on the black-side.
  //      Otherwise, source obtains the message object from the object
  //      using the getSink operation.

  //      When the source is an I/O (audio, serial, ethernet) physical
  //      device, it should call the GetSink operation using the
```

```
  //      resource interface.  For black-side I/O, it needs to obtain
  //      the object reference from the waveform connection factory.

Resource resource;

  //## Attribute: resourceNumber
  //## Documentation:
  //      This attribute describes the identification number
  //      of the resource.

  ResourceNumType resourceNumber;

  //## Attribute: resourceType
  //## Documentation:
  //      This attribute indicates the type of resource
  //      (e.g., modem, access, link, etc.)

  ResourceType resourceType;
  //## Attribute: redSideOnly
  //## Documentation:
  //      This attribute indicates whether or not the resource is on
  //      the red side boundary of the INFOSEC resource.  True
  //      indicates the resource is on the red side of the INFOSEC.

  boolean redSideOnly;

  // Relationships

  // Associations

  //##begin DestinationType.additionalDeclarations preserve=yes
  //##end DestinationType.additionalDeclarations

};

//## Destinations Documentation:
//      The Destinations type defines an unbounded CORBA IDL sequence
//      of Destination(s) as shown in the Destinations figure.  Each
//      Each destination is used to set up a virtual path to a resource.
//## Category: CF_IDL_Design_Components

typedef sequence <DestinationType> Destinations;

//## MessageRegistration Documentation:
//      The MessageRegistration interface provides the operations for
//      an active source (producer) side of a push data transfer.  The
//      interface defines the operations to register and unregister
//      PushSink (consumer) objects to a source (producer) object.
//      The source object pushes data to these sink objects.  The
//      PushSource is implementing the Observer Design Pattern, which
//      behaves as a callback.  This interface supports the Push Model.
//      The relationships for this interface are shown in the
//      MessageRegistration figure.
//## Category: CF_IDL_Design_Components
```

```
interface MessageRegistration {
  //##begin MessageRegistration.initialDeclarations preserve=yes
  //##end MessageRegistration.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //## Operation: setSink
  //## Documentation:
  //      This operation registers a single Message sink (Consumer)
  //      object for call back by a source (Producer) object.  The
  //      Message sink object reference is added to the source object's
  //      list of registered Message sinks.  When pushing data to
  //      this destination the message sink object is used.

  void setSink(in Message pushSink, in ResourceID_Type destinationResource);

  //## Operation: unsetSink
  //## Documentation:
  //      This operation removes a registered Message sink (Consumer)
  //      resource from a source (Producer) object's registered Message Sinks.

  void unsetSink(in ResourceID_Type destinationResource);

  //## Operation: setMultipleSinks
  //## Documentation:
  //      The setMultipleSinks operation registers a set of Message sink
  //      (Consumer) objects for call back by a source (Producer) object.

  void setMultipleSinks(in Destinations destinationSinks);

  //## Operation: getSink
  //## Documentation:
  //      This operation requests the Message sink (Consumer) object
  //      reference that is responsible for processing data to be
  //      received from the requesting source (Producer) object.

  Message getSink(in ResourceID_Type sourceResource);

  //## Operation: getTransferSize
  //## Documentation:
  //      This operation gets the maximum transfer message size.

  unsigned long getTransferSize();

  //## Operation: setTransferSize
  //## Documentation:
  //      This operation sets the suggested transfer message size for
```

```
  //      the Producer Source.

  void setTransferSize(in unsigned long size);

  //##begin MessageRegistration.additionalDeclarations preserve=yes
  //##end MessageRegistration.additionalDeclarations

};

//## Resource Documentation:
//      The Resource interface defines the minimal
//      interface for any software resource created up by a Domain Manager.
//## Category: CF_IDL_Design_Components

interface Resource : Message, MessageRegistration, StateManagement, LifeCycle
{
  //##begin Resource.initialDeclarations preserve=yes
  //##end Resource.initialDeclarations

  // Attributes

  // Relationships

  // Associations

  // Operations

  //##begin Resource.additionalDeclarations preserve=yes
  //##end Resource.additionalDeclarations

};

//## Circuits Documentation:
//      This type defines an unbounded CORBA sequence of
//      DeviceTypes.
//## Category: CF_Domain_Manager_IDL_Design_Components

typedef sequence <CircuitNumType> Circuits;

//## ConfigurationRequestType Documentation:
//      This type defines the configuration request for a physical link
//      in the Radio.
//## Category: CF_Domain_Manager_IDL_Design_Components

struct ConfigurationRequestType {
  //##begin ConfigurationRequestType.initialDeclarations preserve=yes
  //##end ConfigurationRequestType.initialDeclarations

  // Attributes

  //## Attribute: device
  //## Documentation:
  //      This attribute indicates the channel physical link to be
  //      configured within the radio.
```

```
  DeviceType device;

  //## Attribute: operationalMode
  //## Documentation:
  //      This attribute indicates the mode of operation for the channel
  //      physical link.

  string operationalMode;

  //## Attribute: destinationCircuits
  //## Documentation:
  //      This attribute contains a list of channel physical link's
  //      channel/port numbers which indicate the channels that are to
  //      receive the data from this physical link.

  Circuits destinationCircuits;

  //## Attribute: qualifiers
  //## Documentation:
  //      This attribute describes the Configuration Qualifiers for a
  //      PhysicalLinkConfiguration.

  Properties qualifiers;

  // Relationships

  // Associations

  //##begin ConfigurationRequestType.additionalDeclarations preserve=yes
  //##end ConfigurationRequestType.additionalDeclarations

};

//## DomainManager Documentation:
//      The DomainManager interface is used to configure the radio,
//      get the radio's capabilities, software resources, and status.

//      The DomainManager interface can be logically grouped into two
//      catagories: Host and Registration.  The Host operations are
//      used to configure the radio, get radio's capabilities, software
//      resources, and radio status, and to get a channel resource.
//      The Registration operations are used to register and unregister
//      Resource Managers and devices at startup or dynamically for
//      hot swap capability.  The DomainManager Relationships figure
//      depicts the relationships fo DomainManager.
//## Category: CF_Domain_Manager_IDL_Design_Components

interface DomainManager {
  //##begin DomainManager.initialDeclarations preserve=yes
  //##end DomainManager.initialDeclarations

  // Attributes
```

```
// Relationships

// Associations

// Operations

//## Operation: registerDevice
//## Documentation:
//      The registerDevice capability adds a device entry into the
//      DomainManager for a specific ResourceManager object.

boolean registerDevice(in string resourceManagerID, in DeviceType device);

//## Operation: registerResourceManager
//## Documentation:
//      The registerResourceManager adds a ResourceManager object
//      entry into the Domain Manager object database.

boolean registerResourceManager(in string resourceManagerID, in
                                ResourceManager resourceManager);

//## Operation: releaseVirtualCircuit
//## Documentation:
//      The releaseVirtualCircuit operation changes the state of the
//      associated device entries in the DomainManager to be
//      available and releases the resources in the radio for this
//      virtual circuit number.

void releaseVirtualCircuit(in CircuitNumType circuit);

//## Operation: unregisterResourceManager
//## Documentation:
//      The unregisterResourceManager capability unregisters a
//      Resource Manager object from the DomainManager.

void unregisterResourceManager(in string resourceManagerID);

//## Operation: unregisterDevice
//## Documentation:
//      The unregisterDevice operation removes a device entry from
//      the Domain Manager object.

void unregisterDevice(in string resourceManagerID, in DeviceType device);

//## Operation: createVirtualCircuit
//## Documentation:
//      This operation creates a virtual circuit within the radio.
//      The mode of operation (waveform or non-waveform for this
//      virtual circuit is passed in as input to this operation.
//      The virtual circuit end point (Antenna end of the radio)
//      may be specified or determined by the DomainManager.
//      Other parameters passed in to this operation are the
//      destination paths the virtual circuit uses to push data to
//      and the configuration data for the virtual circuit created.
```

```
  //      The createVirtualCircuit operation determines the devices to
  //      be loaded based upon the devices available and the types
  //      devices that are needed by the request.  If  devices are
  //      available and operational state is enable, then this operation
  //      loads and executes the software resource  files based on the
  //      mode of operation on the appropriate processors using the
  //      ResourceManager interface and returns a virtual circuit number
  //      that has been created for this request.

  //      The operation obtains the resource object references from
  //      Factories or from CORBA Naming Services, and transitions the
  //      resource thru their states using the LifeCycle interface.
  //      The application rule indicates which resources the
  //      DomainManager should control and setup virtual paths between
  //      the resources.

  //      If the device is unavailable or its operational state is
  //      disabled, then a NULL virtual circuit number is returned to caller.

  oneway void createVirtualCircuit(in ConfigurationRequestType
                                   configurationRequest);

  //## Operation: getVirtualCircuitResource
  //## Documentation:
  //      The getVirtualCircuitResource operation returns the
  //      object reference for the specified virtual circuit.

  Object getVirtualCircuitResource(in CircuitNumType circuit);

  //## Operation: fileManager
  //## Documentation:
  //      This operation returns a FileManager object reference to the
  //      main FileManager repository.

  FileManager fileManager();

  Networks getNetworks();

  //## Operation: getDevices
  //## Documentation:
  //      This operation returns device information based upon the input
  //      deviceRequest.

  DeviceList getDevices(in Properties deviceRequest);

  Resources getResources(in Properties resourceRequest);

  //##begin DomainManager.additionalDeclarations preserve=yes
  //##end DomainManager.additionalDeclarations

};

#endif
```

5.2.11 Other Reference Sources

1. The Unified Modeling Language User Guide, Grady Booch, et al, Addison Wesley, 1998
2. Unified Modeling Language Reference Manual, Grady Booch, et al, Addison Wesley, 1998
3. Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, et al, Addison-Wesley, 1995
4. The Common Object Request Broker Architecture and Specification, Version 2.2, OMG, 1 February 1998
5. Naming Service Specification contained in CORBAservices: Common Object Services Specification, 05 July 1998
6. Event Service Specification contained in CORBAservices: Common Object Services Specification, 05 July 1998
7. Transaction Service Specification contained in CORBAservices: Common Object Services Specification, 05 July 1998
8. Time Service Specification contained in CORBAservices: Common Object Services Specification, 05 July 1998
9. Trading Object Services Specification contained in CORBAservices: Common Object Services Specification, 05 July 1998
10. MinimumCORBA, Joint Revised Submission, OMG, August 17, 1998
11. CORBAtelecoms: Teecommunications Domain Specifications, Version 1.0, June 1998
12. "Optimizing a CORBA Inter-ORB Protocol (IIOP) Engine for a Minimal Footprint Embedded Multimedia Systems", Washington University Web Site.
13. Aniruddha Gokhale, Gokhale@research.bell-labs.com, Bell Laboratories, Lucent Technologies
14. Douglas C. Schmidt, Schmidt@cs.wustl.edu, Dept. of Computer Science, Washington University, One Brookings Drive, St. Louis, MO 63130
15. http://www.objenv.com/cetus/oo_object_request_brokers.html#oo_corba_orbs_comparisons. (This internet site provides online CORBA ORB comparisons.)
16. http://www.objenv.com/cetus/software.html. (This internet site provides online object-oriented information such as languages, distributed communicating, modeling, etc)
17. http://www.objenv.com/cetus/oo_corba.html (This internet site provides online CORBA information)
18. A Large Distributed Control System Using Ada in Fusion Research, John P. Woodruff and Paul J. Arsdall, Lawrence Livermore National Laboratory.
19. DoD Joint Technical Architecture Version 3.0 Draft 1 2.2-18, 26 February 1999
20. "Recommendations for Using DCE, DCOM and CORBA Middleware", MITRE Corporation. April 13,1998, DII COE Distributed Applications Series by the Defense Information System Agency (DISA) Joint Interoperability & Engineering Organization (JIEO) Center for Computer Systems Engineering (JEXF)

5.3 Base Station Framework Examples

Intentionally left blank this revision

5.4 Satellite Framework Examples

Intentionally left Blank in this revision

6.0 Implementation Recommendation

6.1 Software Download

6.1.1 Introduction

This section contains progress to date on software download which, following further development within the technical committee, will lead to standards recommendations.

Section 6.1.1 presents an overview of software download in the context of SDRF handheld and mobile devices, with reference to application, requirements, methods and implementations, and presents a list of issues to be resolved, concerning regulation, certification and standardization.

Section 6.1.2 presents a number of download scenarios for both handheld and mobile SDRF devices.

6.1.2 Software Download Overview

This section discusses various issues surrounding software download, and has been used for extracting potential standardization issues, establishing implications on the software and hardware architecture of the SDRF device, and defining working group work packages for issues requiring further study.

6.1.2.1 Definition of Software Download

Software download is the process of introducing new program code to a SDRF device to modify its operation or performance.

6.1.2.2 Areas of Application

A SDRF device potentially offers ultimate reconfigurability, via software download, of all its functions, benefiting

- Manufacturers
- Operators (carriers)
- Third-party software developers
- Subscribers

The SDRF device with software download capability thus further enables the ongoing convergence of personal computing and personal communications, and the convergence of technology for personal and professional applications as bearer services are enhanced.

Downloaded software will fall into distinct categories:

- High-level communications and computing *applications*
- *Protocol entities* for modification or changing of the air interface or the bearer service
- Low-level *signal processing algorithms* for modification or changing of the communication physical layer processing

In turn, download will enable the following areas of application:

- Download of new computing and communication applications
- Download of new user interface (look and feel) and I/O drivers
- Adaptation of air interface to implement a new standard (inter-standard adaptation)
- Adaptation of air interface to implement different features (e.g., increased bearer data rate) specified within a standard (intra-standard adaptation)
- Download of incremental enhancements (module or entity replacement)
- Download of patches for software bug-fixes
- Download of reference material, e.g, locally available services and operators
- Download of activation licenses to activate downloaded applications, upon verified receipt of payment

6.1.2.3 Requirements for Software Download

Many parties potentially benefit from the prospect of software download. Moreover, different parties will have different requirements for software download. Some examples of potential beneficiaries are listed:

- Manufacturers
- Operators (carriers)
- Third party software developers
- Subscribers and users
- Military
- Civil
- Regulators
- Software distribution channels
- Hardware distribution channels

Rather than producing a set of requirements to cover all military, civil, and commercial requirements, a list of relevant factors has been proposed. From these factors, listed below, a set of requirements for the commercial, civil, and military groups may be derived.

- Usability - Who will be the users and how easy will it be for them to download the software
- Implementation complexity
- Protocol support including whether it is recoverable, redefinable, and recognition of the instruction set
- Capability exchange, i.e., negotiation with the network or other device as to the waveform and protocol used for communications
- Incremental upgrade. i.e. does all the software have to be replaced or can just one block be replaced
- Authentication of the software module, the hardware module, the user, and the organization
- Download time
- Roaming support – will the terminal function on other networks with different air interfaces
- Reset and recovery
- Who makes the decision to download the software
- Is plug and play functionality required
- Billing and licensing
- Regulatory – type approval and certification
- Configuration management
- Liability
- Responsibility for maintenance
- Download destination
- Access controls
- Backward compatibility
- Extensible and scaleable
- Energy consumption
- Network capacity – how much software download traffic can the network support
- Memory management
- Compression
- Can the download process be used to deny service
- Low probability of interception and detection

6.1.2.4 Methods of Downloading Software

A number of methods of downloading software to a SDRF device can be envisaged, for example:

- Distribution of new software via *SIM-card or other removable media*
- Downloading software via a modem and fixed network, e.g., telephone or cable service
- From a handheld field device
- From CD-ROM or Internet/Intranet, via a PC;
- From a street side terminal, e.g., download onto SIM card via an Automatic Teller Machine (ATM)
- Download from a Point of Sale terminal in a shop or service center
- Over-the-air reconfiguration by downloading software over the wireless link. In this case, the options of point-to-point and point-to-multiple-point are available, allowing forms of broadcast reconfiguration

6.1.2.5 Download Implementation Issues

With each method of download are an associated set of issues. Some examples:

- Security
- Integrity of downloaded code
- Billing and licensing of downloaded code
- Regulatory Issues: e.g., How will 'type approval' be applied to terminals capable of reconfiguration via software developed by independent third parties? Must all hardware and software be type approved?
- Technical Capability: Does the destination terminal have the technical capability to correctly run the software
- Resilience
- Ownership of the software
- Liability for the software
- Maintenance
- Configuration management
- Usability
- Recovery

6.1.2.6 Standardization Issues

The following standards issues are raised by the preceding discussion areas, and will be appended, expanded, and discussed in future issues of this document. Likelihood and implications of *de-facto* or *de-jure* standardization will also be discussed:

- What is the minimum level of standardization required to ensure that SDRF devices can communicate (download) within a number of radio environments, yet allow manufacturers, operators, service providers, and independent software developers the flexibility to innovate within their field?

- How can the API structure be defined such that code updates are permissible by the operator, service provider, and manufacturer?
- How will a SDRF device detect the local radio environments?
- What modifications will be required to existing standards to permit software download and multistandard roaming?
- Will a classification of the technical performance of the SDRF device (in terms of transceiver bandwidth, processing power benchmarks, memory, etc.) be required? Can this be achieved via a capability exchange as part of the download procedure?
- Will standardization be required to address billing, licensing, ownership, and security issues for downloaded software?

6.1.2.7 Regulation and Certification Issues

Many issues arise regarding certification and type approval. Historically, type approval has been applied to specific terminal equipment containing resident fixed software. When software download is considered, the personality, behavior, and performance of the terminal can be modified. Some issues are listed below:

- Must all software applications and all hardware platforms be separately type-approved
- How can type approval be guaranteed for any type-approved software application running on any type-approved platform
- How should billing, licensing (both time-domain and geographical), and ownership of software be handled
- How can regulation protect against malicious intent (e.g., software viruses)

6.1.3 Software Download Scenarios

An important consideration in designing interfaces and architectures for implementation of a terminal that is software-definable to some degree, is its ability to support software download. Software download can be achieved using a number of mechanisms, for example:

- From a smart-card, which contains the software to be downloaded, and which optionally may contain a pre-paid license for using the software
- From a host computer via a local network
- From a remote host via street-side terminals or modem (e.g., ATM, point-of-sale terminals)
- Over the air, directly from a server, either point-to-point or point-to-multipoint (broadcast). In a commercial situation, the server would be provided by the service provider, utilizing the network operator's air time resources

As a means of identifying the architectural, implementation, regulatory, and standardization issues posed by the requirement to support software download, a number of download scenarios have been developed for both handheld and mobile application areas. The scenarios, detailed in following subsections, will be expanded through future working sessions, to determine:

- Implication on interface (API) design, including capability exchange features;
- Regulatory implications, including network integrity, security, billing, licensing issues;
- Implications on new standards (e.g., 3rd generation cellular) and modifications required to existing standards (e.g., GSM, AMPS) such that over-the-air software download can be accommodated;

The scenarios were developed as exercises within the handheld and mobile subcommittees, and do not represent all options and mechanisms for download: they are point examples. As a result of developing the scenarios, ongoing work within the download working group will determine whether a generic download procedure can be developed, covering handheld and mobile requirements, and not precluding any options. Results will be published in future revisions of this document.

6.1.3.1 Handheld Architecture Download Scenarios

Three scenarios have been developed for handheld (commercial cellular) terminals:

- Download from a smart card
- Over the air download of a single software module (update or bugfix)
- Over the air download of a complete air interface software suite

In each case, the scenario is described as a flowchart indicating the expected information transfer between the software source device (smartcard or service provider server) and the terminal. In each case the source device (acting as master) controls the download procedure, although either source or terminal may initiate download.

Each scenario follows a common general procedure:

1. *Initiation*: download source device or terminal initiates a download request
2. *Mutual Authentication*: between source device and terminal
3. *Capability exchange*: to ensure terminal can be configured to accept, install, and successfully run the downloaded code
4. *Download Acceptance Exchange*: Source provides information on type approval of the code, download procedures and schedules, installation procedures, billing and licensing options, and procedures. Options are selected by the terminal or terminal user, which are validated by the source device
5. *Download/Integrity test*: Code is downloaded from source device to a buffer area within the terminal according to the agreed download schedule. In-line integrity testing of received data is performed with retransmission’s requested where errors are detected. Only 100 percent error-free reception of code is acceptable
6. *Installation*: Code is appropriately compiled and installed within the terminal, providing billing and licensing conditions have been met. Because there may be a time delay between download of code and installation, a capability exchange internal to the SDRF device is required prior to installation, to ensure that the configuration of the terminal at install time remains acceptable for correct operation of the downloaded application/entity

7. *In-situ testing*: This is a provision for testing the downloaded code on the terminal platform. Test vectors downloaded with the program code are used within the test to verify the modified operation. It is desired that in-situ testing be optional
8. *Non-repudiation Exchange*: Response from SDRF device to confirm successful installation, permitting required billing to take place

The desire that in-situ testing remains optional relies upon the following conditions being met:

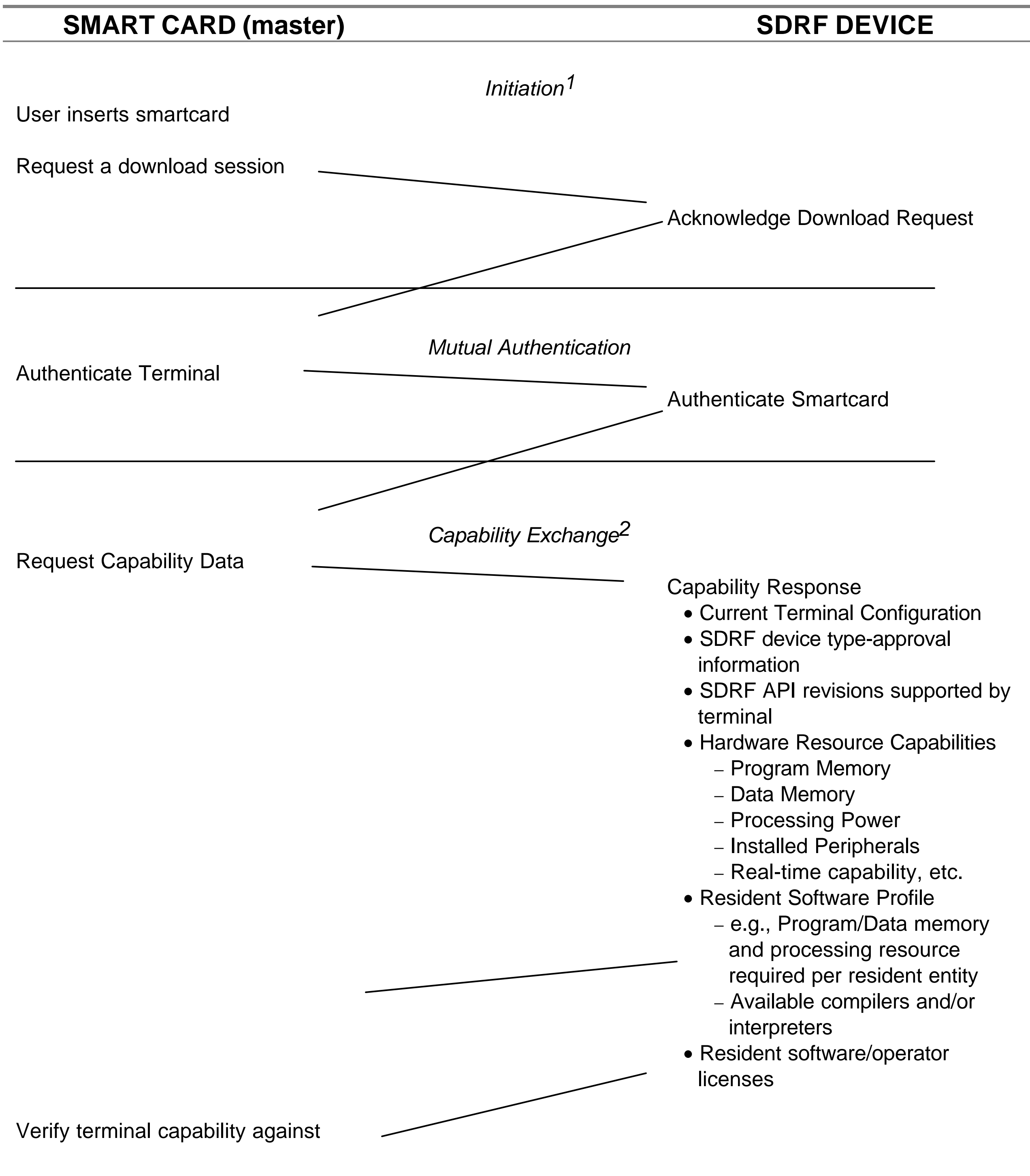
- Downloaded code was delivered to the terminal error-free
- Downloaded code was compiled within supplied critical constraints
- Downloaded code complies with the terminal logical/physical interface structure

and that these represent the full set of conditions that if met, ensure satisfactory operation of the terminal.

The three scenario flowcharts appear below.

6.1.3.1.1 Download from a Smartcard

Figure 6.1.3-1 below describes the communications between smartcard, network, and terminal (SDRF device) to download one or more software modules into the terminal, and to install the software. The scenario assumes that the smartcard contains the new software, and describes how that software might be downloaded and installed in the SDRF device.



[1] Initiation may be triggered by SDR device, or the download source: this diagram does not attempt to reflect all options.

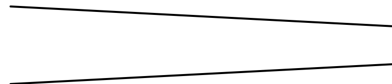
[2] Capability exchange may comprise the exchange of multiple messages

application requirements.
IF application cannot be built to match terminal capabilities, terminate download process
ELSE identify optimum application configuration (using modules resident on the smart-card) for terminal capability

---

Download Acceptance Exchange

Send Download Installation Profile
- Download type (mandatory or optional, etc.)
- Download and installation procedures, e.g.,
  - download complete entity or incrementally
  - download schedule
- Installation options
- Licensing information
- Billing information

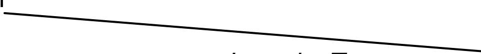


Select installation options. Indicate acceptance/rejection

Validate selected options

Download Software Module[3]

Download Code, including:
- Capability Tables (Resource Requirements)[4]
- A delivery wrapper:
  - Compilation requirements[5]
  - Real-time constraints
  - Installation information



Integrity Test

Test code integrity, e.g., via checksums
Request retransmissions as appropriate
Code considered installable if:
- Error-free (i.e., bit-exact)
- Type approved to current API structure (*could be assumed if*

---

[3] The software module could be a code segment (application, protocol entity or functional entity), or a set of switches or parameters to reconfigure resident software (remote control).

[4] The Resource Requirements must sufficiently describe the requirements of the system configuration to support the module(s) being downloaded, such that local capability exchanges across APIs *within the SDR device* can determine whether the module(s) can be successfully installed

[5] A compiler resident within the SDR device will be required to support platform independent download. The complexity and feasibility of platform independent download and resident compilation is very much dependent upon where the downloaded module resides within the SDR architecture (application, protocol entity, signal processing algorithm).

downloaded over the air by approved operator)

Acknowledge verified receipt

Terminate download procedure

Installation[6]

Internal capability exchanges: to ensure device is able to support the downloaded module[7]

IF module(s) cannot be supported by terminal, terminate.
ELSE request installation key.

Initiate billing/licensing negotiation, if appropriate. Negotiation[8] could be

- between terminal and smartcard, e.g. to check expiration of paid-up license.
- between terminal and network, e.g., if geographical constraints exist

Billing/licensing response, e.g.,

- Acceptance of terms
- On-line payment (if required)

IF unacceptable, deny installation key request
ELSE
send installation key and in-situ test program/data (if appropriate)

Deliver and install code
Update capability descriptor
Signal successful installation

In-situ testing[9] *(if appropriate)*

Figure 6.1.3-1: Software Download from a Smartcard

6 This stage assumes that the new module(s) (and their capability tables) are accessible by the SDR device

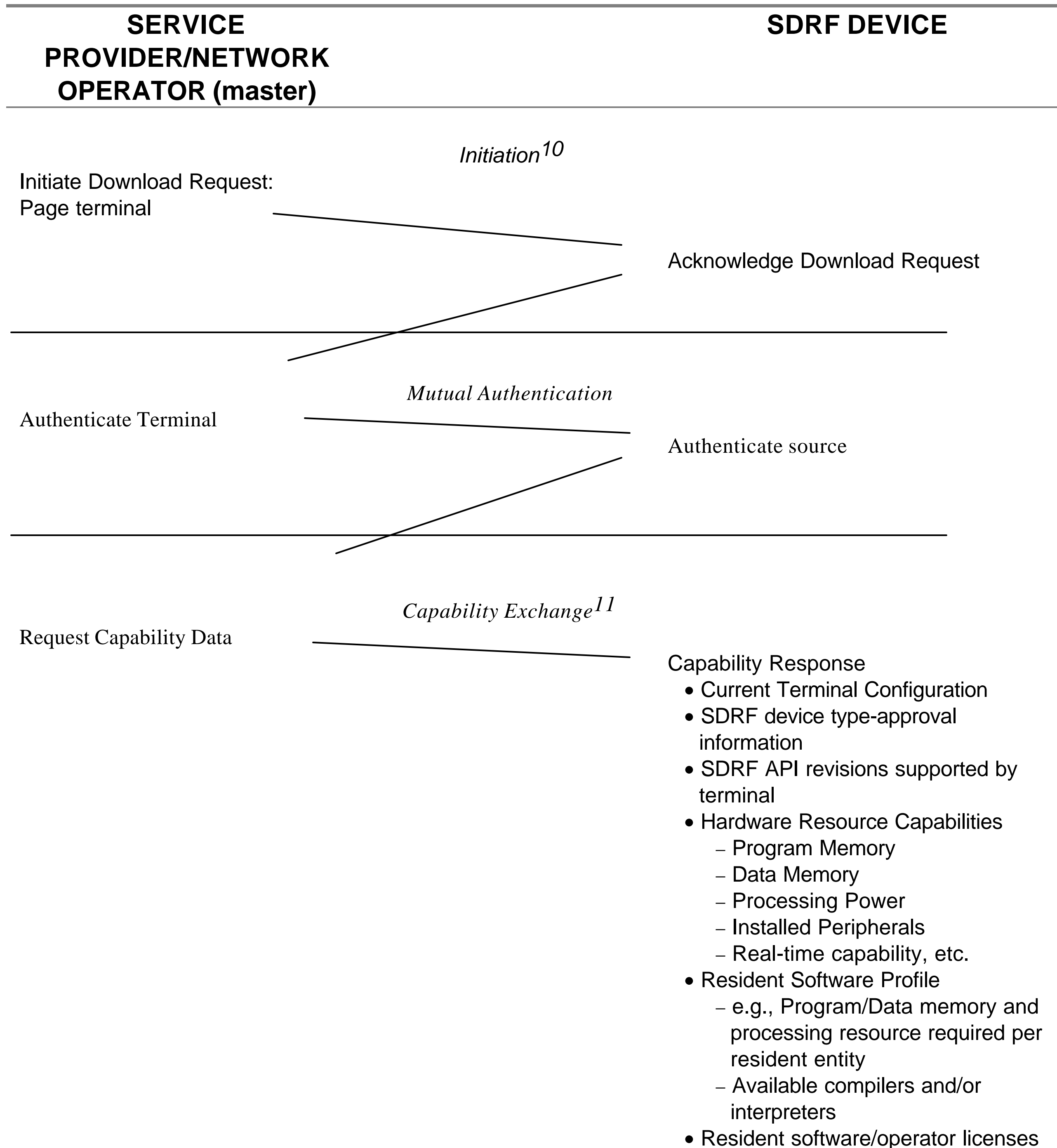
7 See note 4: Further changes to the software and/or hardware configuration of the SDR device may have taken place between download and installation. This capability exchange/resource management operation determines whether the module(s) can still be successfully installed.

8 Licensing issues may include validation of geographical position, time and network operator and/or service provider.

9 Regardless of whether a formal in-situ test is performed, it may be desirable to retain the ability to switch back to the previous software configuration (prior to installation) if the new configuration fails.

6.1.3.1.2 Over the Air Download: Single Module or Entity

Figure 6.1.3-2 below describes the communications between network and terminal to implement *over the air* download of a single functional entity or software module. An example might be the replacement of a software-implemented speech codec to improve speech reproduction quality, or to fix a bug.



[10] Initiation may be triggered by SDR device, or the download source: this diagram does not attempt to reflect all options.

[11] Capability exchange may comprise the exchange of multiple messages

IF an appropriate software module exists, which can be configured to comply with the current terminal capability and configuration: select the appropriate software module and open a download channel.
ELSE Terminate download.

Download Acceptance Exchange

Send Download Installation Profile
- Download type (mandatory or optional, etc.)
- Download and installation procedures, e.g.,
  - download complete entity or incrementally
  - download schedule
- Installation options
- Licensing information
- Billing information

Select installation options. Indicate acceptance/rejection

Validate selected options

Download Software Module[12]

Download Code, including:
- Capability Tables (Resource Requirements)[13]
- A delivery wrapper:
  - Compilation requirements[14]
  - Real-time constraints
  - Installation information

Integrity Test

Test code integrity, e.g., via checksums. Request retransmissions as appropriate. Code considered installable if:
- Error-free (i.e., bit-exact)
- Type approved to current API structure (*could be assumed if downloaded over the air by approved operator*)

Acknowledge verified receipt

[12] The software module could be a code segment (application, protocol entity or functional entity), or a set of switches or parameters to reconfigure resident software (remote control).

[13] The Resource Requirements must sufficiently describe the requirements of the system configuration to support the module(s) being downloaded, such that local capability exchanges across APIs *within the SDR device* can determine whether the module(s) can be successfully installed

[14] A compiler resident within the SDR device will be required to support platform independent download. The complexity and feasibility of platform independent download and resident compilation is very much dependent upon where the downloaded module resides within the SDR architecture (application, protocol entity, signal processing algorithm).

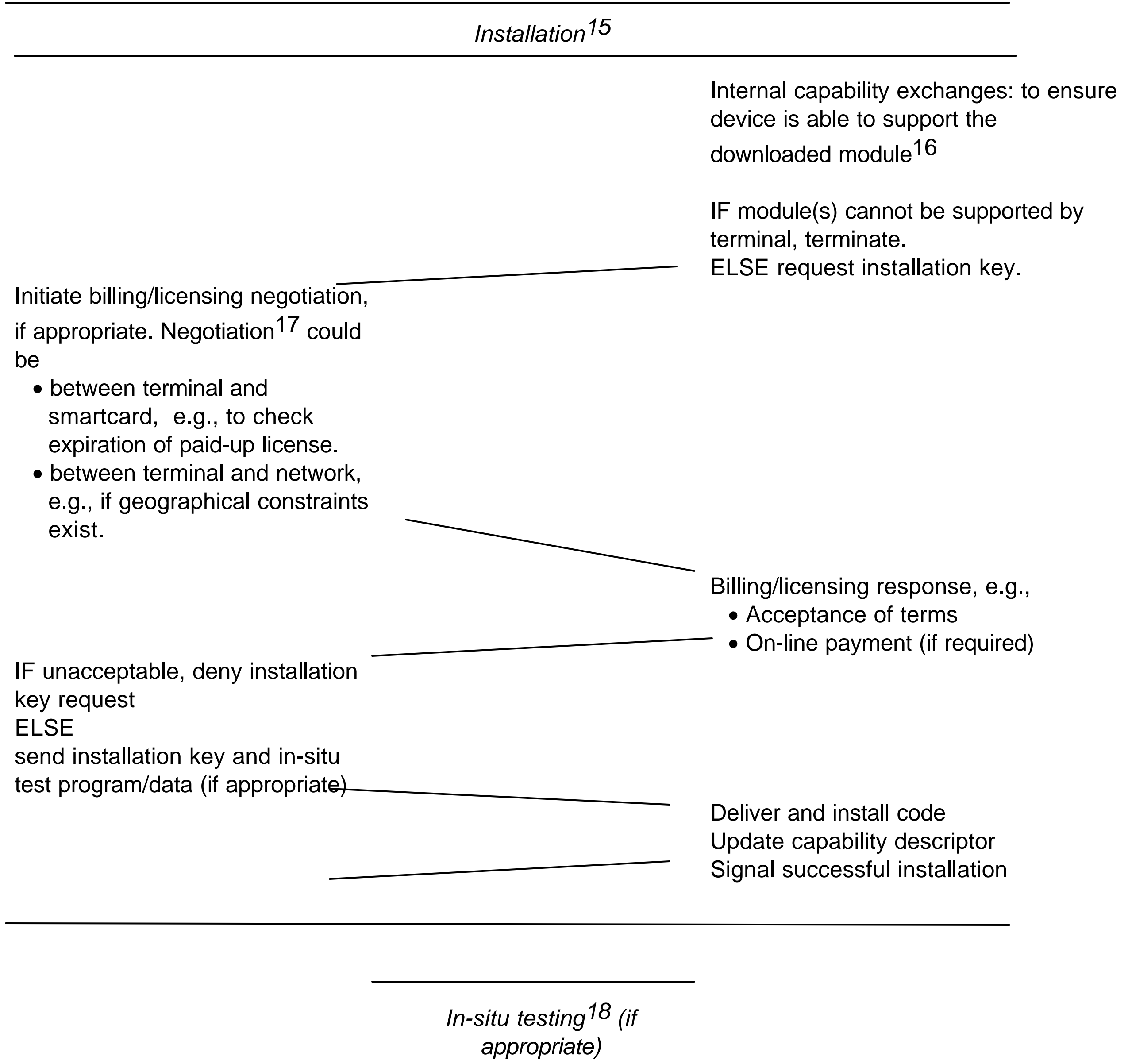


Figure 6.1.3-2: Over the Air Software Download of a Single Module Update

15 This stage assumes that the new module(s) (and their capability tables) are accessible by the SDR device

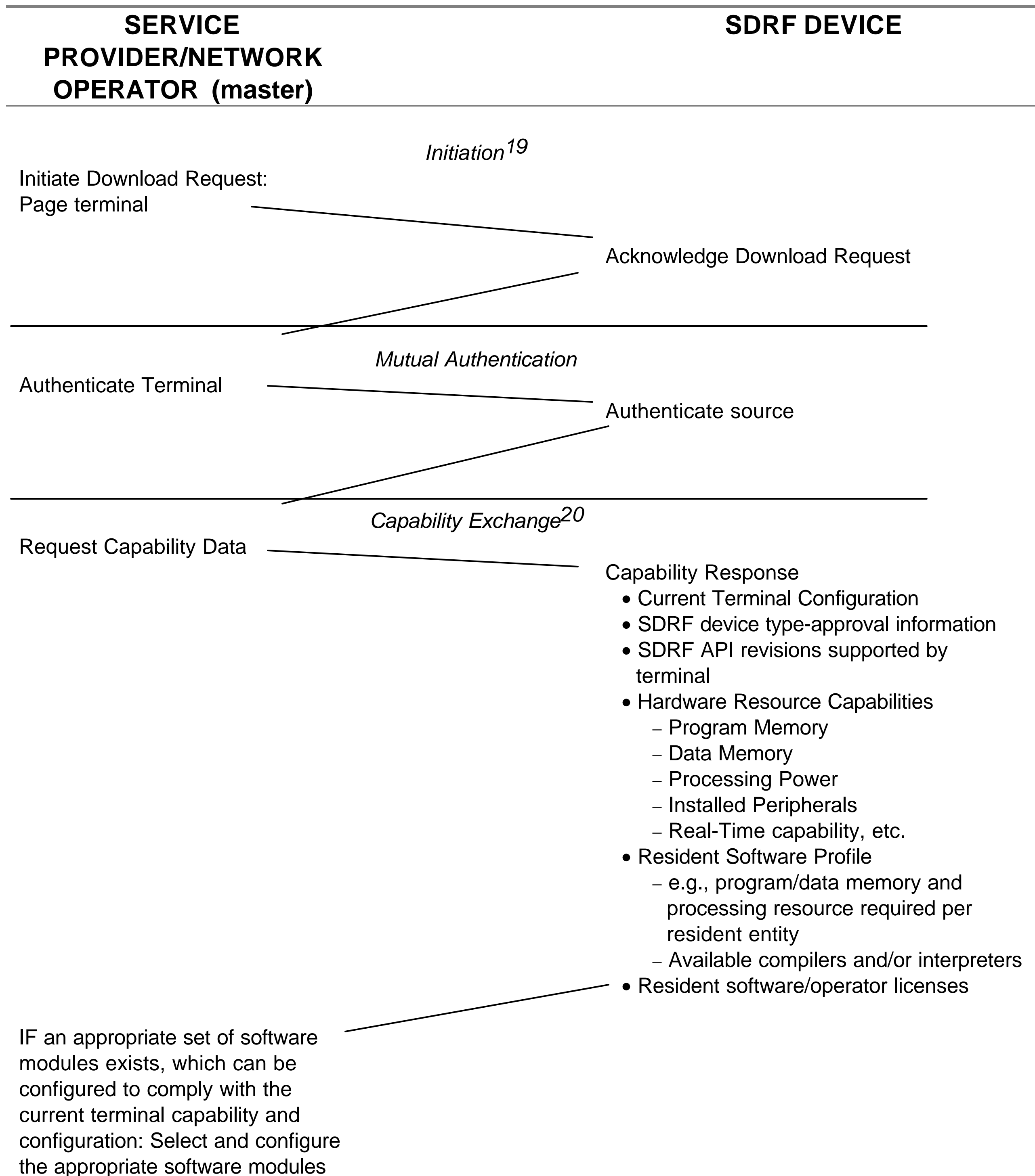
16 See note 13: Further changes to the software and/or hardware configuration of the SDR device may have taken place between download and installation. This capability exchange/resource management operation determines whether the module(s) can still be successfully installed.

17 Licensing issues may include validation of geographical position, time and network operator and/or service provider.

18 Regardless of whether a formal in-situ test is performed, it may be desirable to retain the ability to switch back to the previous software configuration (prior to installation) if the new configuration fails.

6.1.3.1.3 Over the Air Download: Complete Air Interface

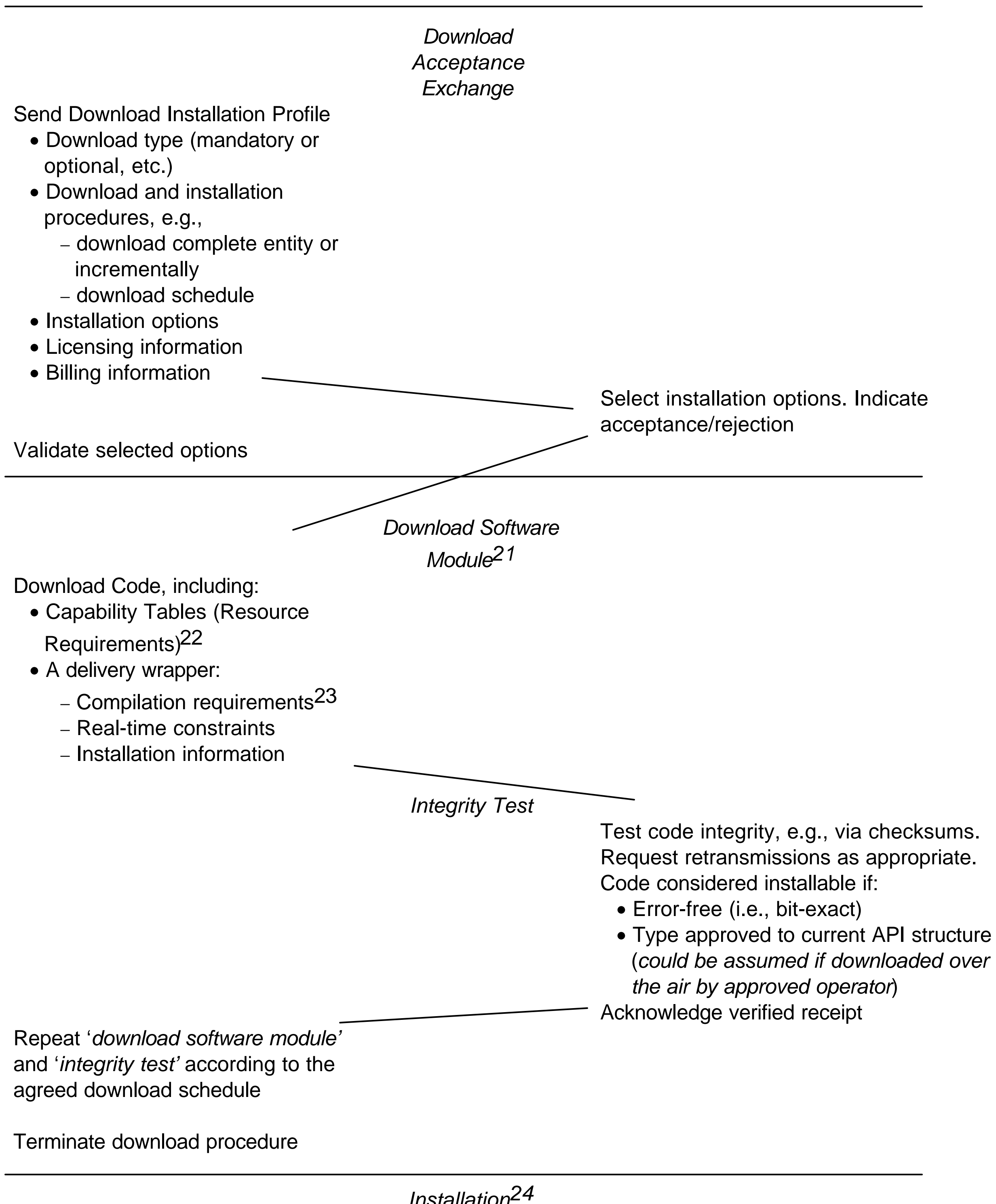
Figure 6.1.3-3 below describes the communications between network and terminal to implement *over the air* download of a complex set of software modules, which might define for example, the complete protocol stack and signal processing software for a new standard.



[19] Initiation may be triggered by SDR device, or the download source: this diagram does not attempt to reflect all options.

[20] Capability exchange may comprise the exchange of multiple messages

and open a download channel.
ELSE Terminate download.



[21] The software module could be a code segment (application, protocol entity or functional entity), or a set of switches or parameters to reconfigure resident software (remote control).

[22] The Resource Requirements must sufficiently describe the requirements of the system configuration to support the module(s) being downloaded, such that local capability exchanges across APIs *within the SDR device* can determine whether the module(s) can be successfully installed

[23] A compiler resident within the SDR device will be required to support platform independent download. The complexity and feasibility of platform independent download and resident compilation is very much dependent upon where the downloaded module resides within the SDR architecture (application, protocol entity, signal processing algorithm).

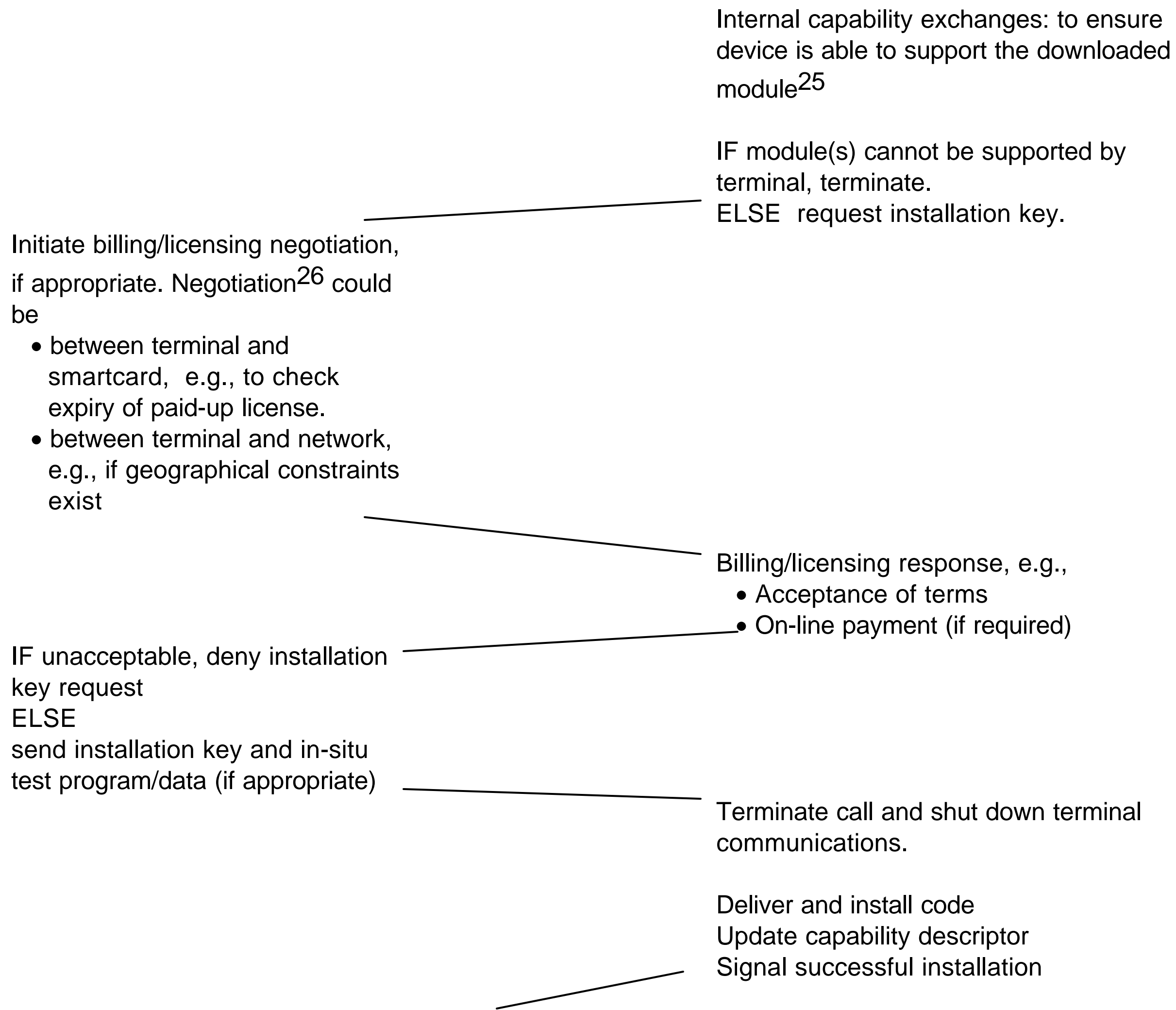


Figure 6.1.3-3: Over the Air Software Download of a Set of Control, Functional, and/or Protocol Entities

[24] This stage assumes that the new module(s) (and their capability tables) are accessible by the SDR device

[25] See note 22: Further changes to the software and/or hardware configuration of the SDR device may have taken place between download and installation. This capability exchange/resource management operation determines whether the module(s) can still be successfully installed.

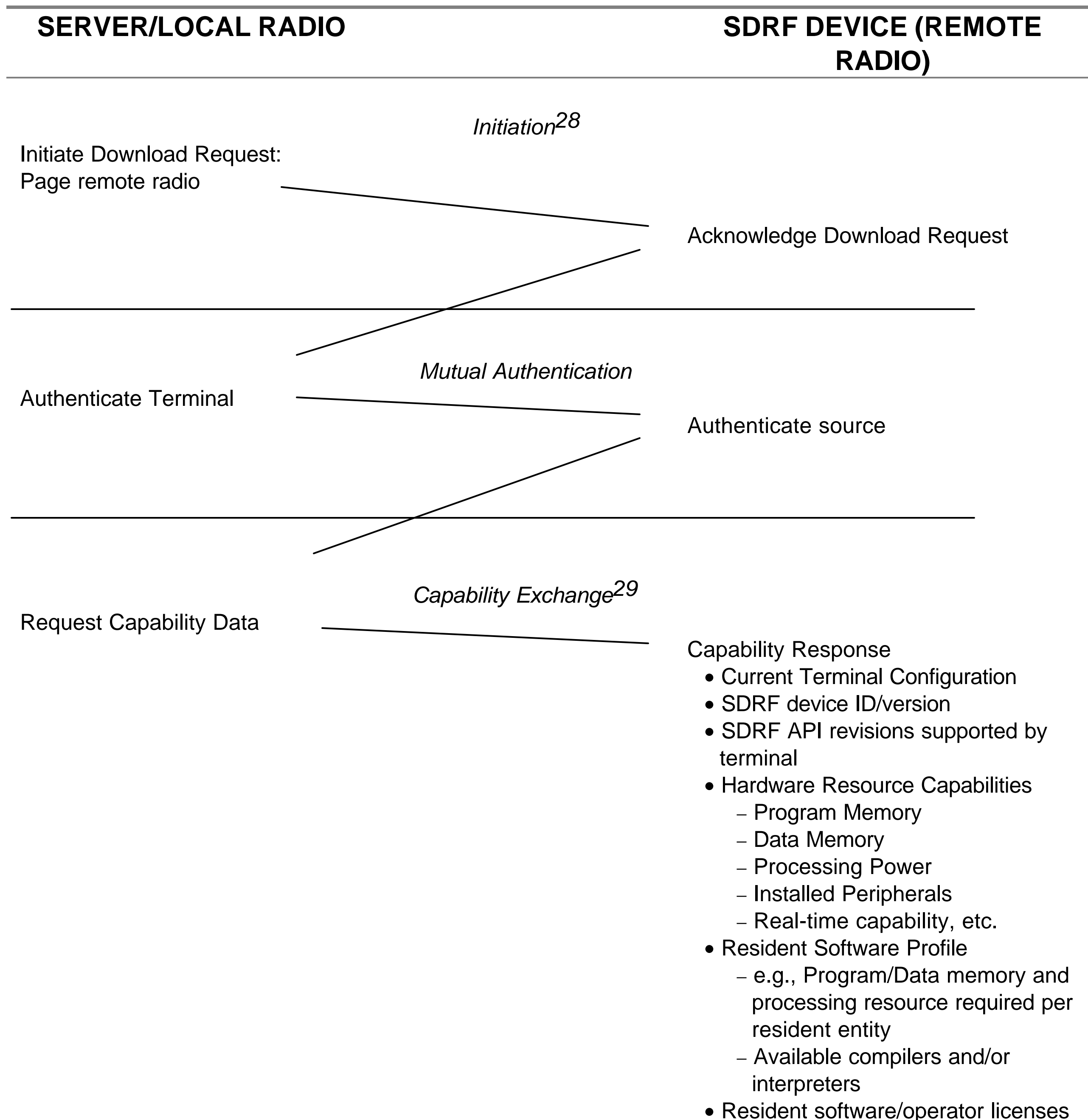
[26] Licensing issues may include validation of geographical position, time and network operator and/or service provider.

[27] Regardless of whether a formal in-situ test is performed, it may be desirable to retain the ability to switch back to the previous software configuration (prior to installation) if the new configuration fails.

6.1.3.2 Mobile Architecture Download Scenarios

6.1.3.2.1 Over the Air Download: Single Module or Entity

Figure 5.1.3-4 below describes the communications between network and terminal to implement *over the air* download of a single functional entity or software module. An example might be the replacement of a software-implemented speech codec to improve speech reproduction quality, or to fix a bug.



[28] Initiation may be triggered by SDR device, or the download source: this diagram does not attempt to reflect all options.

[29] Capability exchange may comprise the exchange of multiple messages

- Server evaluates remote radio needs
- Formulates download actions

IF an appropriate software module exists, which can be configured to comply with the current terminal capability and configuration: Select the appropriate software module and open a download channel. ELSE Terminate download.

Download Acceptance Exchange

- The server seeks authorization to proceed with the download. This authorization process will likely vary from system to system, but may include such factors as cost, capability or capacity, availability, geography/positional status, supporting infrastructure, etc. In other cases, such as when a charge is incurred, further authorization may be required.
- The server proceeds with the download by establishing a download session with the remote reference point. This includes any handshaking, certificate exchange or passing of download key to the remote radio reference point.

- Select installation options. Indicate acceptance/rejection
- Certificate exchange
- Validate download certificates or keys
- Accept download session establishment

Validate selected options

Download Software Module[30]

[30] The software module could be a code segment (application, protocol entity or functional entity), or a set of switches or parameters to reconfigure resident software (remote control).

Download Code, including:
- Capability Tables (Resource Requirements)[31]
- A delivery wrapper:
  - Compilation requirements[32]
  - Real-time constraints
  - Installation information

Integrity Test

Test code integrity, e.g., via checksums. Request retransmissions as appropriate. Code considered installable if:
- Error-free (i.e., bit-exact)
- Type approved to current API structure (*could be assumed if downloaded over the air by approved operator*)

Acknowledge verified receipt

Terminate download procedure

Installation[33]

- Acknowledge receipt of download to server
- Execute internal update process

- Terminate session
- Update configuration files

In-situ testing[34] *(if appropriate)*

Figure 6.1.3-4: Over the Air Software Download of a Single Module Update (Mobile)

6.1.4 Preliminary API Messaging Requirements

This section of the report describes a framework for a tier 1 SDRF API which supports the previously developed download process (section 6.1). Message exchanges between download server and terminal are explored in order to identify whether new control messages are required to implement the download process, in addition to the previously defined generic API messages and formats (section 4.4). Also, necessary refinements to the SDRF API methodology are extracted and discussed.

Detailed message syntax, data structures, passed parameters, status information, capability/configuration table structure and content are not yet defined.

---

[31] The Resource Requirements must sufficiently describe the requirements of the system configuration to support the module(s) being downloaded, such that local capability exchanges across APIs *within the SDR device* can determine whether the module(s) can be successfully installed

[32] A compiler resident within the SDR device will be required to support platform independent download. The complexity and feasibility of platform independent download and resident compilation is very much dependent upon where the downloaded module resides within the SDR architecture (application, protocol entity, signal processing algorithm).

[33] This stage assumes that the new module(s) (and their capability tables) are accessible by the SDR device

[34] Regardless of whether a formal in-situ test is performed, it may be desirable to retain the ability to switch back to the previous software configuration (prior to installation) if the new configuration fails.

This process represents a top-down approach to defining the API, and is likely to require revision as more detail is developed. This section of the report represents the first phase of the 'human API definition' described in the SDRF API definition process.

6.1.4.1 Objectives

The objectives of this piece of work are:

- To establish a framework to support the top-down development of the API;
- To identify any new command messages which may be required in addition to the generic messages previously defined
- To identify any refinements to the process and data structures described in the SDRF API design guide

With these objectives in mind, it should be noted that the ladder diagrams and examples described in this *discussion* are illustrative, and do not necessarily represent the definitive approach.

6.1.4.2 Download API Context

The Download API illustrated in figure 6.1.4.2-1 forms the interface across which all programmable functionality of the terminal is defined: this applies either to functional definition at the time of manufacture, or in the field by an authorized user – for example a service engineer, or indeed the end user downloading new applications or functionality.

In this sense, the Download API forms the highest-level terminal API, and can be visualized as 'surrounding' the terminal.

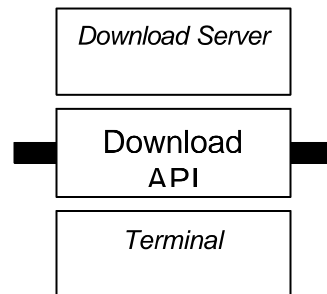


Figure 6.1.4.2-1 Location of the Download API

The three-tier API model [see SDRF API Design Guide] applies to the download API:

Tier 1: Functional (Messaging)

Describes the messages and associated parameters and expected responses required to initiate and effect download from server to terminal. It will also describe all possible solicited status responses and unsolicited status messages.

Tier 2: Transport

Describes the mechanism by which the messages are transferred. In the case of the 'download' message, this would describe the protocol used for the download of code, including scheduling, the error detection/correction scheme and any fail/retry mechanism employed.

Tier 3: Physical

Defines the physical interface between download server and terminal, for example, the connector specification for download from smartcard.

6.1.4.2.1 Assumptions

The download process described herein assumes that a registered connection exists between the 'server' device (supplying the downloaded code) and the device set to receive the downloaded code. In this sense, the tier 1 API control messages described here may be embedded into the information passing along that connection, and will require delimiting such that they are recognized as SDRF messages within that information stream. This might be accomplished, for example, by preceding each message with an escape character, recognized globally as '*SDRF message follows*'.

It is also assumed that the download link is point-to-point. The point-to-multipoint (broadcast) case will be addressed as a future work item.

6.1.4.3 The Download Protocol Framework

Figure 6.1.4.3-1 illustrates the download process developed through the study of various download scenarios. A clear partition exists between the *download process* (initiation, mutual authentication, capability exchange, download acceptance exchange, download) and the *installation process*, the latter assuming that code is locally stored in the terminal and is accessible (but possibly compressed and/or encrypted). There could be a time lapse between completing the download process and initiating the installation process, requiring the need for an internal capability exchange prior to actual installation, to insure that the downloaded module(s) is still compatible with the current terminal configuration and capability.

The download API described in this section of the report defines the interface and messaging requirements for the *download process.* The *installation process* is handled separately by the API through which the downloaded module(s) will be accessed. This installation process will use previously defined generic API messages such as:

```
place_module (dest,info) ← (status, parameters)
```

which replaces, modifies or augments a module accessed by the API with another module that is stored locally.

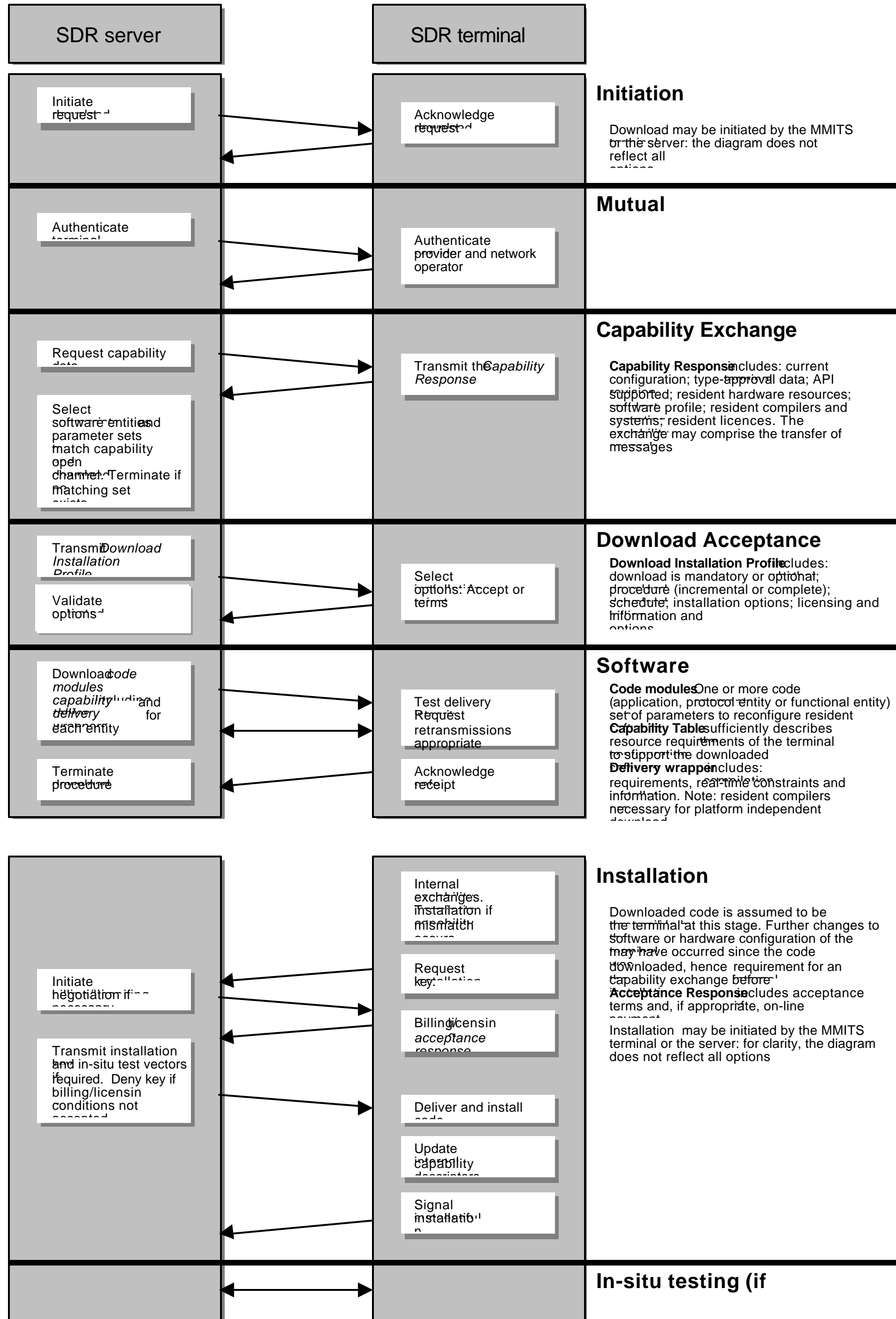


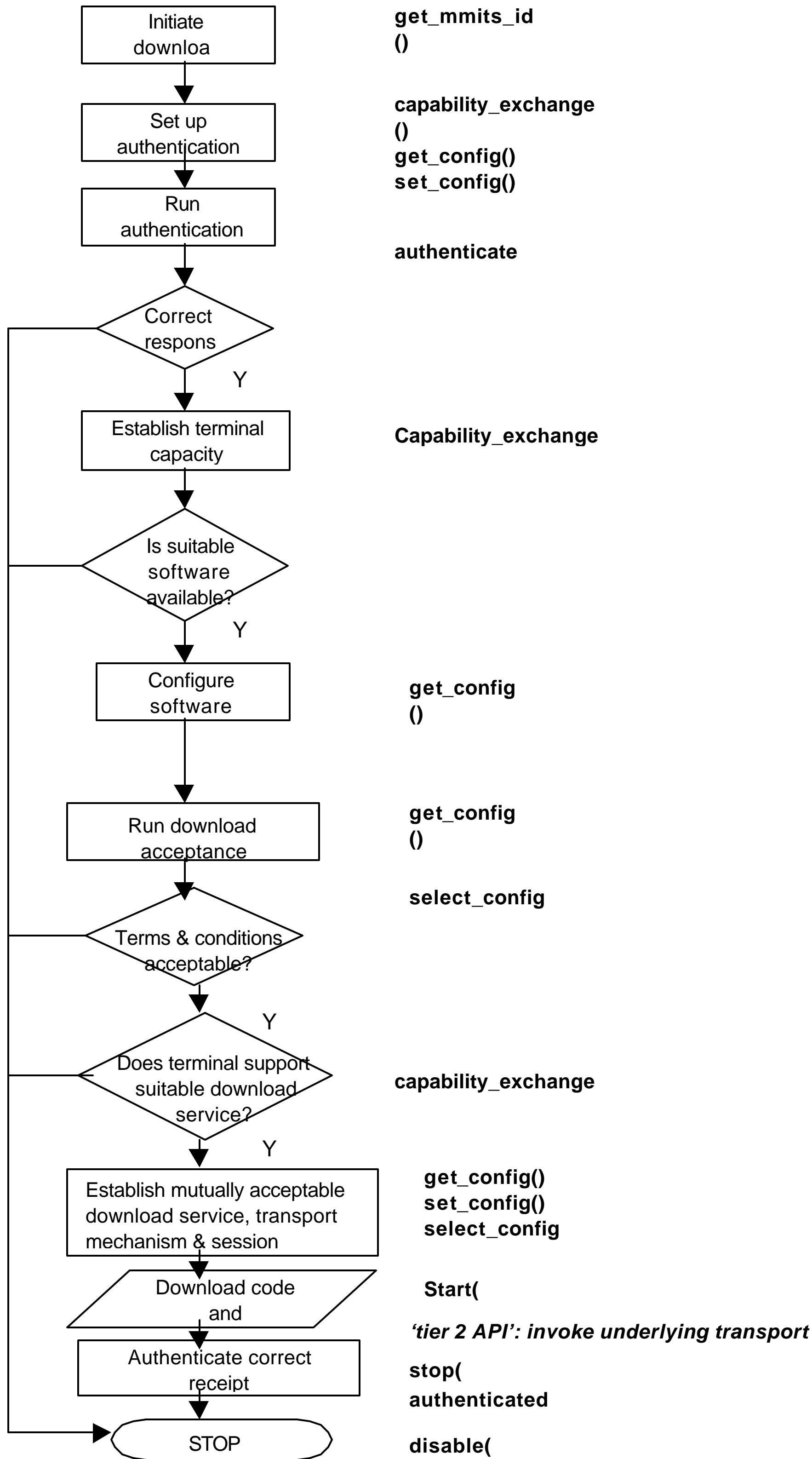
Figure 6.1.4.3-1: Download Protocol

6.1.4.4 API Framework

6.1.4.4.1 Example Download Sequence

In order to relate the download protocol in Figure 6.1.4.3-1 with the tier 1 download API messages, consider figure 6.1.4.4.1-1 where the download sequence is illustrated as a flowchart, with suggested API messages. This represents an initial framework for developing the API. Note that figure 6.1.4.4.1-1 is stylized and does not show messages flowing between the parties – it simply illustrates how an outline sequence of messages begins to implement the protocol described in figure 6.1.4.3-1.

The sections following the flowchart take each main action separately, and begin to develop the messaging requirements in more detail. Detailed syntax, parameter definitions and data structures are to be developed.



get_mmits_id
()

capability_exchange
()
get_config()
set_config()

authenticate

Capability_exchange

get_config
()

get_config
()

select_config

capability_exchange

get_config()
set_config()
select_config

Start(

'tier 2 API': invoke underlying transport

stop(
authenticated

disable(

Figure 6.1.4.4.1-1: Download flowchart example with example API message

6.1.4.4.2 Initiate Download Process

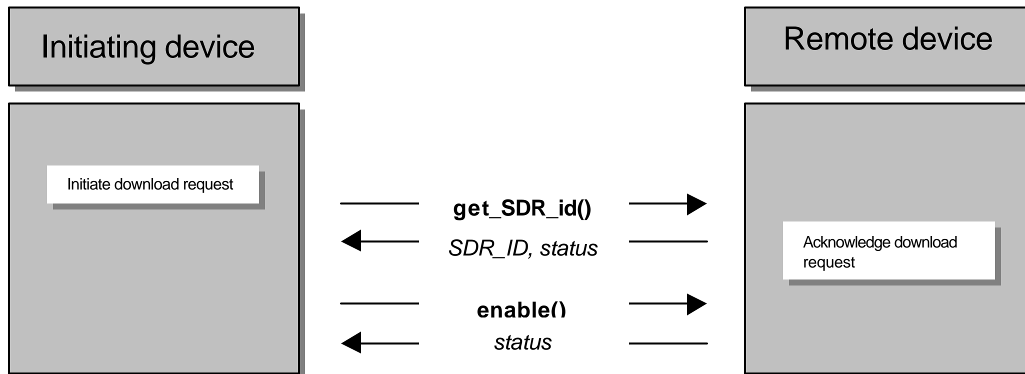


Figure 6.1.4.4.2-1: Messaging to initiate download

The download process could be initiated by either the server party or the client (terminal) party. It is proposed that a new message `get_SDRF_id()` is issued by the initiating party to begin the process, permitting the remote device to identify itself as supporting the SDRF download API.
The response to the `get_SDRF_id()` message will be either:

- The SDRF_ID, which includes a simple go/no-go indication of the other party's ability to support SDRF Download. Status information will also be returned. See section 5.4.6 'New

- No response, in which case a locally employed time-out will be triggered. The assumption would be that the device is not SDRF compliant if, after a number of get_SDRF_id() attempts, no response is received within the time-out period.

The exact format and content of the SDRF_ID has not been defined here. Any regulatory or SDRF_ID maintenance/control issues have not been addressed here.

The enable() message could arguably appear before or after the get_SDRF_id() message: the optimum sequence will be determined when the next level of detail is examined. The suggested sequence is to place the enable() after get_SDRF_id(). In this case, get_SDRF_id() identifies whether the terminal is SDRF download compliant (go/no-go), and the following enable() indicates a request to allocate resources for a download session.

6.1.4.4.3 Authentication Process

A very general framework is described within which many standard authentication schemes may be supported.

A mutual authentication is required prior to download. By virtue of the mechanism described in figure 6.1.4.4.3-1, mutual authentication gives access to each other's detailed capability tables necessary for assessing capabilities to successfully download, install and run the desired new code. Prior to authentication, only a very limited set of information may be accessible through a capability exchange – enough to permit the authentication to take place.

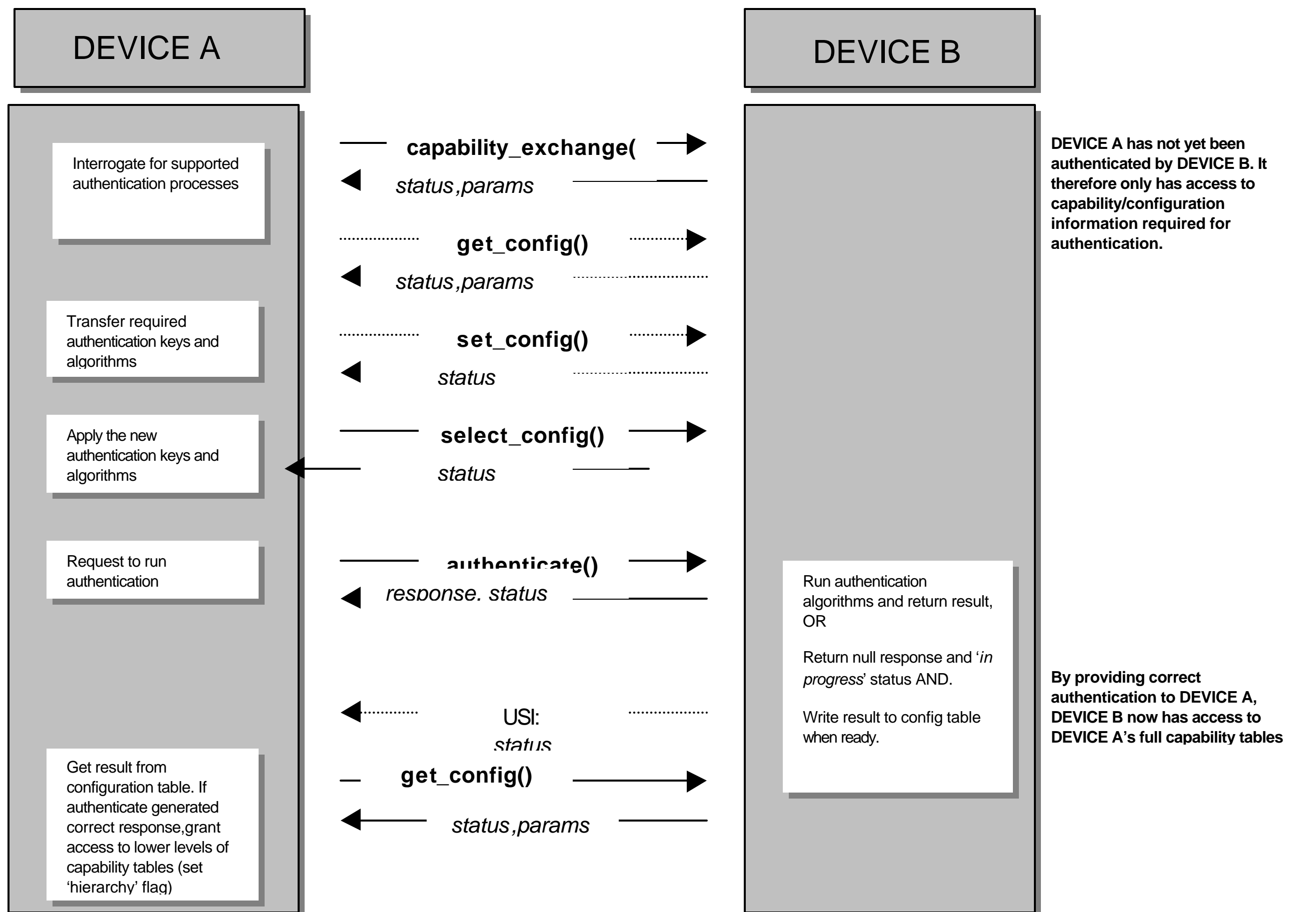


Figure 6.1.4.4.3-1 Messaging required for authentication

It should be noted that the procedure described here is sufficiently generalized to allow either existing standardized authentication mechanisms to be applied, or indeed new or updated mechanisms. Furthermore, authentication of the *user* (SIM card or equivalent) will have already taken place at registration, prior to the download request. This scenario represents an additional mutual authentication of SDRF devices relating to download.

Figure 6.1.4.4.3-1 illustrates the likely process required for one device (DEVICE A) to authenticate the other (DEVICE B). Using the appropriate authentication scheme (algorithmic, biometric, etc.), algorithm(s) and key(s) negotiated during a prior capability and configuration exchange, the newly defined `authenticate()` message causes execution of the algorithm by the remote device, which returns the result as a response to the message.

The `get_config()` and `set_config()` transfers show with dotted arrows (indicating optional) represent the setting up of a new authentication configuration in device B. They are therefore only required if that configuration does not already exist (determined by the preceding capability exchange).

If additional information is required by the authentication procedure specific to the current authentication process (for example, a random data sequence) this could be accommodated by an information field in the `authenticate()` message, thus:

authenticate(destination, info)

Figure 6.1.4.4.3-1 illustrates the option to continue whilst the authentication process completes: in this case the response to the authenticate() message is a null response accompanied by solicited status information indicating ‘in progress’. A USI (unsolicited status information) is returned asynchronously when the authentication has been completed, indicating the ID of the configuration table which contains the result. DEVICE A may then interrogate DEVICE B for the result via a `get_config()` message.

If DEVICE B is successfully authenticated, DEVICE A may permit DEVICE B full access to its capability information if previously restricted to that required for authentication.
The process may be repeated by the other device (DEVICE B) to achieve mutual authentication. The new API message, authenticate() is defined in section 5.4.6 ‘New API Messages for Download’.

Multiple Authentication

The declaration of a specific authentication message allows that message to be invoked at any time if required, even during a download session – the diagrams in this section of the report simply depict examples of when the message is *likely* to be used.
For example, `authenticate()` may be applied (with different parameters) after the download has taken place, to verify that the complete module received was that which was authorized to be received, and that it was received without corruption.

6.1.4.4.4 Encryption

This process requires a negotiation between SDRF devices to establish and set up an agreed set of encryption algorithms and parameters. Encryption then needs to be switched on and off, possibly on a packet-by-packet basis. These tasks can all be achieved using the previously defined generic API messages. This is illustrated in figure 6.1.4.4.4-1 It should be noted that the encryption referred to here pertains to the download sequence only, and may be different than any encryption already applied to other information flowing across the registered connection.

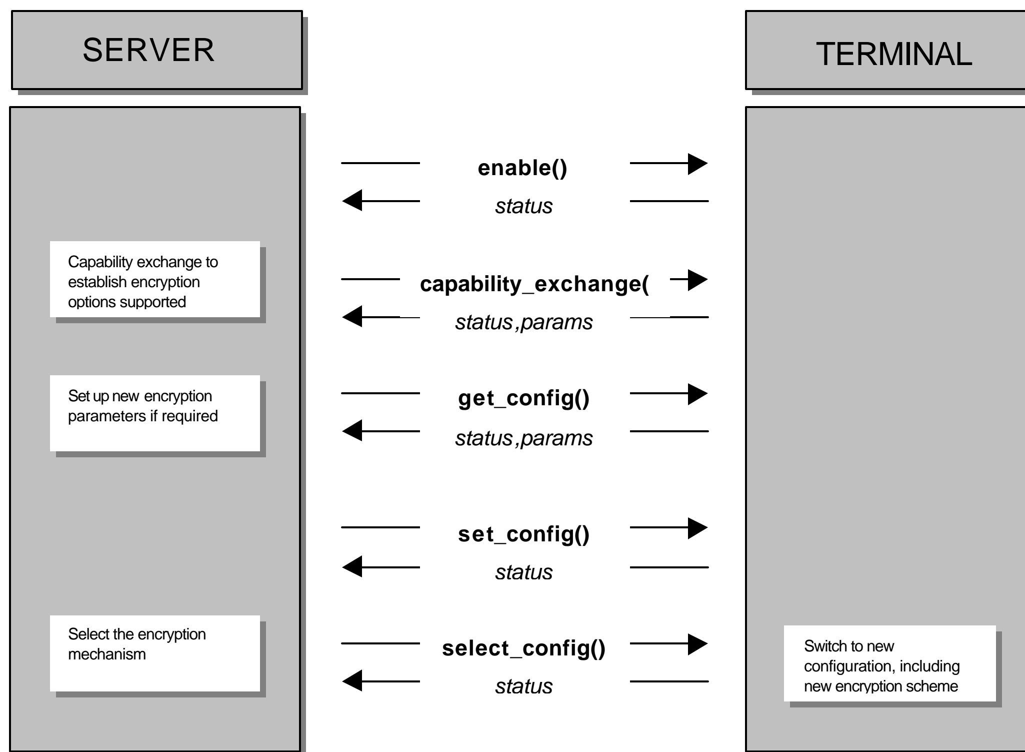


Figure 6.1.4.4.4-1 Messaging required for encryption

Examples

Like the authentication process, this illustrates a general process for setting up and invoking encryption. Exactly how all the requirements will be supported within the framework is yet to be defined.

For example, to apply encryption to selected packets might require definition of a configuration table entry, coupled with an ‘encrypt enable’ flag attached to each packet (defined in a tier 2 API).

Other examples of encryption requirements:

Extensions to standard encryption mechanisms: define through configuration tables

Confirmation of encryption on/off: through SSI status responses

Change of key attributes or resetting/synchronizing key: through configuration tables

It should also be noted that the encryption mechanism (if used) should be selected via `select_config()` *prior* to authentication. This provides some protection against the situation of the download being 'hijacked' by a rogue device after the legitimate device was authenticated.

6.1.4.4.5 Capability Exchanges

The download server must establish the capabilities of the terminal in two senses:

- To determine which software modules and associated parameters to download so as to match the capabilities of the terminal.
- To understand the download modes and associated attributes supported by the terminal, such that a download channel may be successfully established between download server and terminal (see 'Code Download mechanism' below).

Both of these exchanges may be accomplished using the previously defined `capability_exchange()` message.

The chosen download process attributes and parameters may then be set up using the `set_config()` message, and the new information selected using `select_config()`. This process is very similar to the encryption example in figure 6.1.4.4.4-1.

6.1.4.4.6 Download Acceptance Exchange Process

It is suggested that negotiation via capability exchanges and configuration tables be used to implement the download acceptance exchange as illustrated in figure 6.1.4.4.6-1.

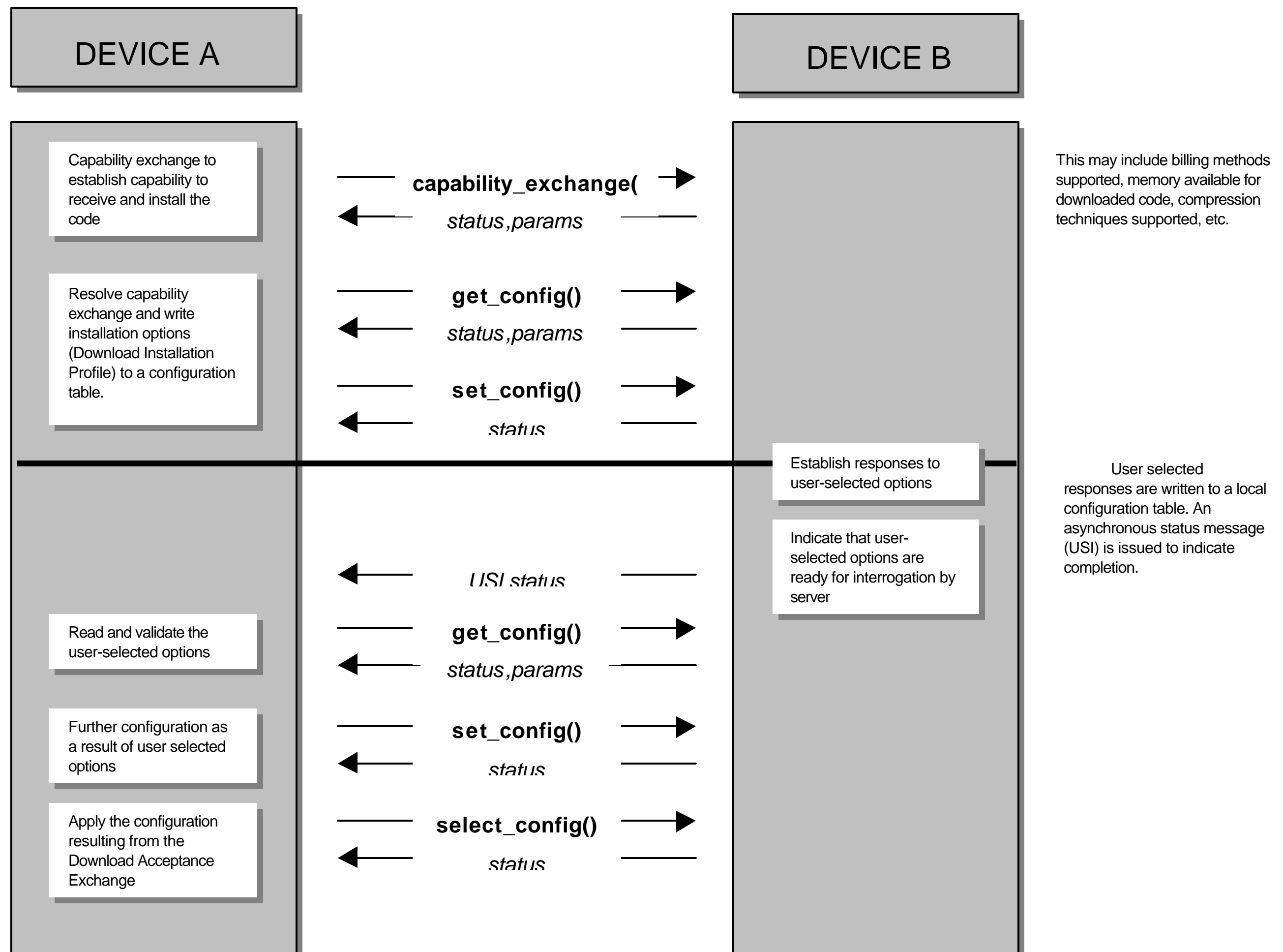


Figure 6.1.4.4.6-1 Download Acceptance Exchange Messaging

From Figure 6.1.4.4.6-1, the download acceptance exchange essentially involves the delivery of:

- Fixed terms and conditions for installation
- An ‘options form’ (Download Installation Profile) from the download server to the terminal, to be ‘filled-out’ and returned to the server.

Examples of the information to be exchanged are:

- Whether the download is mandatory or optional
- A schedule for the download process
- Installation and configuration options
- Terms and conditions, including licensing and billing options

Because this process may require user responses to complete some of the options it may be preferable to allow the server to progress while waiting, implying the use of asynchronous status messages (USIs) to indicate that the response to the download installation profile is ready for interrogation.

6.1.4.4.7 Code Download Process

The download process itself may be considered as the invocation of a transport mechanism to transfer downloaded code from server to terminal. The preparation for opening a download data channel and selecting a download data transfer mechanism and integrity checking mechanism might be accomplished by capability exchange (described above) and then setting parameters in configuration tables. The download would begin upon the issue of a start() message, and successful completion or failure would be signaled by a USI status response. This is a tier 1 API, and is illustrated by the example exchanges in figure 6.1.4.4.7-1

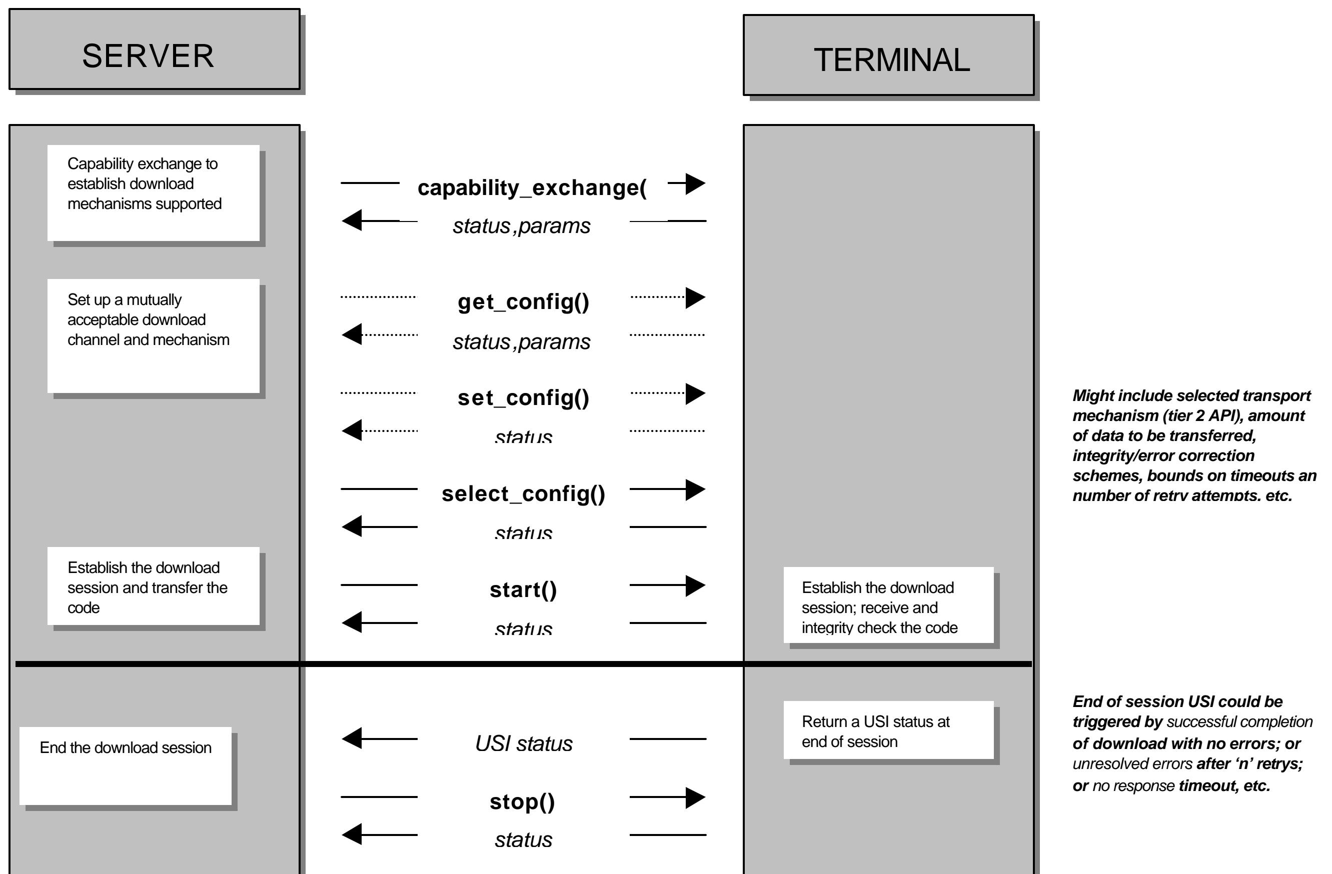


Figure 6.1.4.4.7-1 Download Messaging

The get_config() and set_config() transfers show with dotted arrows (indicating optional) represent the setting up of a new download mechanism. They are therefore only required if that configuration does not already exist (determined by the preceding capability exchange).

The detailed messaging required to implement the transfer, that is the chosen transport mechanism (for example, TCP), is a *tier 2 API* and is not addressed here. The tier 2 API would define messages used by the chosen transport mechanism (send, receive, retry, abort, flush, etc), and any integrity testing employed (error correction codes, check-sums, retransmission requests, etc.). The transport mechanism may describe a point-to-point or broadcast stream transfer, or a packet-transfer.

6.1.4.5 API Implications

6.1.4.5.1 Hierarchical Capability Tables

Working group discussions revealed the need to support hierarchy (nested lists) within capability tables, such that the required information could be accessed economically from the potentially vast lists.

For download, it may also be necessary to limit access to the capability tables until the devices have been mutually authenticated.

Together, these requirements imply the *option* of a hierarchical capability table structure across the download API, whose highest level only contains sufficient capability information to permit SDRF ID exchange and mutual authentication.

One approach is that the response to a capability_exchange() message includes *flags* to indicate whether there is *accessible* hierarchy beneath the values returned, and that the flags are set so as to deny access to any lower levels of hierarchy until mutual authentication is successfully established. This scheme is illustrated in figure 6.1.4.5.1-1

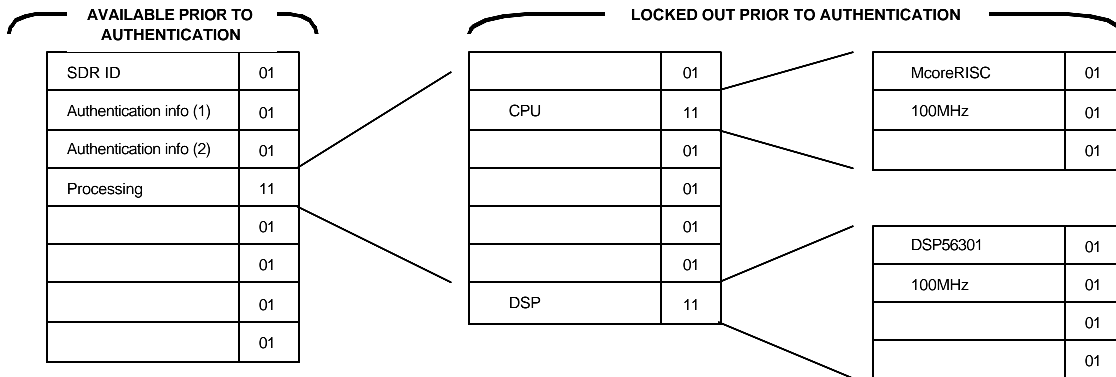


Figure 6.1.4.5.1-1 Hierarchical Capability Tables

Details of the exact structure of the capability table and how the structure is accessed are not described here.

6.1.4.6 New API Messages for Download

This section describes the new messages required by the download API to implement the download protocol described in figure 6.1.4.3-1 This represents an initial set of messages which are likely to require augmentation as the API details are developed. Furthermore, other more generic messages described in section 4.4 of the Technical Report will also be required (for example, start_module, stop_module, enable_module, disable_module, place_module, etc.). Detailed parameter listings for the messages have yet to be developed and are not described here.

get_SDRF_id(destination) ¬ (SDRF_id, status)

destination:	Destination identifier of download target below the download API
SDRF_id:	Header assigned to each SDRF compliant terminal or download server.
Examples:	Version numbers; unique identifier; download capability
status:	This is returned to indicate the success or otherwise of the message and the current status of the device/module. Included in the status word will be the destination module/device ID to indicate that it was sent to the correct location.
Description:	This message is sent to the Download API by the device initiating download (in a cellular system this could be either the basestation or the mobile terminal). The response indicates a simple ability of the target device to send/receive downloaded code by the given means (examples: over the air, smartcard).
Other actions:	If there is no response from the target SDRF device within a time-out period it may be assumed that the device is not SDRF compliant and download cannot take place. It may be preferable to attempt to send this command a number of times before making this assumption.
Issues:	The exact format of the SDRF ID, the information it contains, how it is issued and controlled and associated regulatory issues are yet to be established.

authenticate(*destination, info*) ¬ (*response, status*)

destination: Destination ID of device to be authenticated.

info: Additional information required by the authentication process unique to *this* authentication.
Example: A random number sequence

response: Result from remote execution of authentication algorithm.
Example: This could be a null, accompanied by an 'in progress' status flag, permitting the initiating device to continue whilst the authentication process is completed. In this case a USI would be issued upon completion, indicating 'authenticate complete'.

status: This is returned to indicate the success or otherwise of the message and the current status of the device/module. Included in the status word will be the destination module/device ID to indicate that it was sent to the correct location. An 'in progress' flag may be necessary if the initiating device is to be alerted asynchronously of completion of the authentication.

Description: Using the authentication algorithm and key negotiated during a prior capability exchange, this message causes execution of the algorithm by the remote device, which returns the result as a response to the message.

Issues: If the authenticate() message returned the expected response, access may be granted access to lower levels of hierarchy within the capability tables of the device which issued the message.

6.2 Mode switcher API example specification

6.2.1 Introduction

This section builds on the API design guide and the example APIs to provide a more detail implementation for a mode switcher function for a multi-function i.e. multi-mode and multi-band SDRF device.

6.2.2 Requirements

The requirements for a mode switcher are dependent on the type and design of the phones that are used in the handset. In practice, a multi-function phone can be viewed as a single physical entity containing several virtual phones. This view of the multi-functional provides the widest scope for implementation and introduction of the technology. The virtual phone model can be implemented using several physical

discrete phones — the velcro phone model — or with a single phone with switchable modes and bands. A SDRF device can use either of these techniques in its implementation.

- Capabilities
- Low level control

6.2.3 Levels of detail

If the mode switcher is to have a consistent API, it needs to be able to provide different levels of control over the configuration, depending on how the multi-function SDRF device has been implemented. In this section, three levels are defined and are referred to collectively as levels of detail (LoD). For a simple high level mode switcher that is used to select from a variety of SDRF devices velcro'd together, the level of control and therefore API detail needed will be less than with a true software definable radio where the actual RF waveform, channel coding and power control could be configured.

It is important therefore to have an API that supports these different levels of control without imposing a large overhead or need for detailed knowledge on systems that cannot support it, either because they were never designed to or because the design does not warrant it.

Given that there is a need for different levels of detail within the mode switcher, what else is needed? It should be remembered that the API simply defines the connection information between two entities and does not define how the information is processed within each entity except to supply the responses as defined. In other words, when a command is sent to configure the SDRF device to a specific state, the mode switcher and the rest of the system above it do not know anything about the implementation below it.

6.2.3.1 Regulatory

This is a very high level view of the capabilities of the SDRF device and are in line with a model number or type approval level of functionality as approved and defined by regulatory bodies. In this case, the multiple modes and bands are effectively treated as a set of separate entities as far as the mode switcher is concerned. This greatly simplifies the interface and allows virtually any type of lower level implementation to exist.
It also assumes that all other messaging and control information, including configuration is handled outside the mode switcher API. This is useful for working with existing handsets and thus provides an upgrade path.

6.2.3.2 Non-regulatory

At this level, some high-level configuration information is used to enhance the higher level choices. This may include the choice of voice encoder and other similar information that can configure the SDRF device's capabilities.

6.2.3.3 Detailed

At this level, there is the low level configuration information which can allow the complete set up and definition of a waveform, modem coding and so on. This will need to be extremely detailed to provide the breadth of support that is necessary.

In practice, this level of support is initially of immediate interest to mobile radios rather than commercial handhelds simply because of the technology required to implement this high level of configurablility is not yet available or cost effective for all markets.

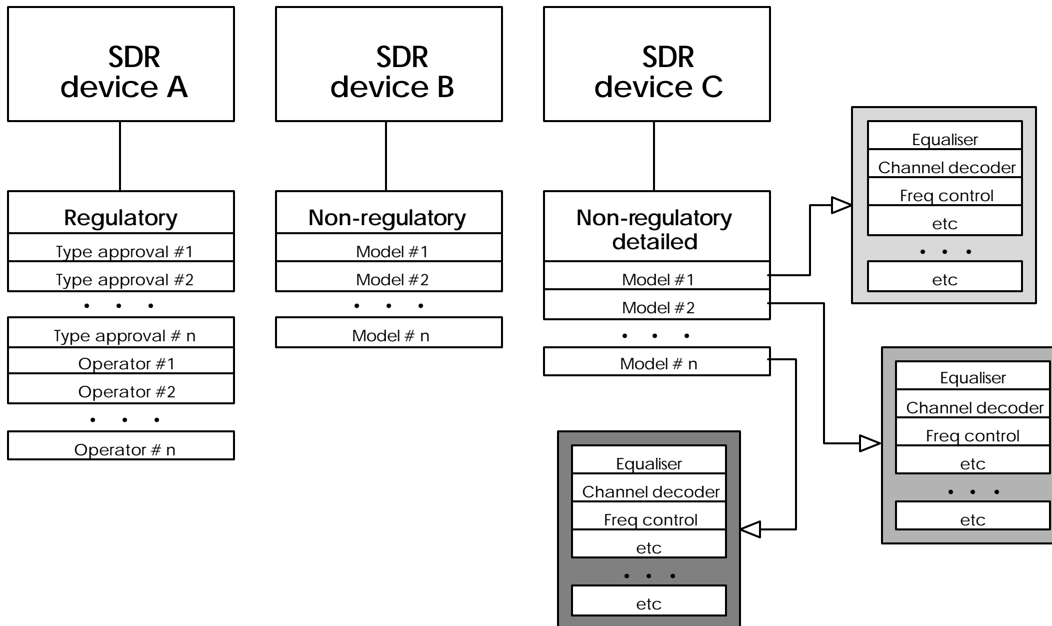


Figure 6.2.3-1 Levels of detail for SDRF device capability and configuration tables

It is possible for a SDRF device to support several different levels simultaneously. For example, it could support a regulatory level and a detailed level. While possible, this may not make much sense because the detailed access could allow changes that break the type approval. This assumes that the regulatory and detailed levels are interlinked and that the subsequent resource management issues are resolved.

It is feasible for a SDRF device to support several different levels with each level controlling a particular aspect of its facilities. For example, cellular handset networks can be accessed via a regulatory level while a detailed level can be used to configure the SDRF device in other parts of the RF spectrum.

6.2.4 Describing the configuration

The API design examples provide a framework for describing this information. It uses two basic concepts: a set of simple commands and the idea of configuration tables. The commands allow the different modes to be selected and the actual configuration is controlled by entries in configuration tables. These tables are available in two forms: one provides the information for capability exchanges and the other for controlling the actual configuration.

By combining these tables with the commands described in the API control message section, a complete API for the mode switcher function can be created to work at the three levels previously defined.

The tables should be assumed to be fixed in size in that there is no provision for their extension directly except by using the *place_module* command to replace or augment a module. This will force a capability exchange and a new configuration to be used. The new capability and configuration tables may be smaller or larger when compared to the original module. Previous services may have been removed to allow the new service to be supported, for example and vice versa.

The table definitions in this section may be expanded to support new revisions and non-standard information, however. This allows backward compatibility as the APIs are further extended and the provision of non-SDRF specified support. Suitable techniques for this are described in section 4.1.2 API design guide.

Resource management is carried out through the use of configuration tables. This means that the capability tables are a source of reference information to allow the configuration tables to be correctly set up and used. Thus the capability tables are shown as read only and cannot be modified through this API, except by using the *place_module* command, as previously mentioned. This facility to directly modify the capability tables may be added in the future — along with a command to make the modifications — to allow a higher level of resource management to be implemented.

6.2.5 API definitions

The rest of this section describes some suggestions for the contents of the capability and configuration tables that are used within the APIs for the three levels earlier identified. The contents are illustrative of the type and detail of information that would needed for the three levels. It should not be assumed that the facilities described in these tables are a definitive representation of the final APIs. They simply provide examples to enable a better understanding of how the concept of capability and configuration tables will work. It should also be remembered that the advantage offered by capability tables is the declaration of what facilities are supported and, perhaps more importantly, what are not supported. For example, an equaliser may not support parameter changes. In this case, the parameter section within the capability table would declare such changes are not supported. This does not mean that the ability to change these parameters are not needed either for future expansion or to support future products. It is clear that additional work will be required to evaluate and define the exact contents of these tables.

These examples will be refined in future document revisions. Some of the issues to be addressed will include:

- Unique identifiers within each structure
- Graphical representations
- Application of this approach to other SDRF models
- Implementation within a formal description language.

6.2.6 Regulatory capability exchange

Parameter	**Definition**	**Read/write status**
Capability ID	A unique value that is used to identify a set of capabilities. The ability to support multiple sets is included for future expansion but it is anticipated that only one set of capability exchange information will be supported currently.	Read only
Capability size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of type approvals	The number of type approvals that the SDRF device can support. It is also used to help decode the tables.	Read only
Type approval #1	The first type approval reference	Read only
Type approval #1 support	The first type approval support definition. Declares whether this mode is an exclusive one or capable of simultaneous operation with another service provided by a different type approval.[1]	Read only
Type approval #*n*	The *n* th type approval reference	Read only
Type approval #*n* support	The *n* th type approval support definition. Declares whether this mode is an exclusive one or capable of simultaneous operation with another service provided by a different type approval.	Read only
Number of operator approvals	The number of operator approvals that the SDRF device can support. It is also used to help decode the tables. This differs from the type approvals in that it provides an additional level of qualification if needed which relates a type approval to a specific operator. This could be used to define additional/optional services by the operator that provides them.	Read only
Operator approval #1	The first operator approval reference	Read only
Operator approval #1 support	The first operator approval support definition. This will declare whether this mode is an exclusive one or capable of simultaneous operation with another service provided by a	Read only

[1] This support may have to be restricted initially because it can rapidly become quite cumbersome to implement and declare when many potential combinations are available. By having a parameter in this set of examples it acts as a reminder that such support may be required in the future. It is likely that the first implementations will declare this parameter to be 'exclusive'.

	different operator approval.	
Operator approval #z	The z th operator approval reference	Read only
Operator approval #z support	The z th operator approval support definition. Declares whether this mode is an exclusive one or capable of simultaneous operation with another service provided by a different operator approval.	Read only

6.2.7 Regulatory configuration

Parameter	**Definition**	**Read/write status**
Configuration ID	A unique value that is used to identify a set of configuration parameters. The ability to support multiple sets is made possible by this parameter.	Read only
Configuration size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of type approvals	The number of type approvals that the SDRF device can support. It is also used to help decode the tables.	Read only
Type approval #1	This is the first type approval reference. This should be the same as the entry in the capability table.	Read only
Type approval #1 status	The first type approval status. Declares whether this mode is enabled or disabled. [2]	Read/write
Type approval #*n*	The *n* th type approval reference	Read only
Type approval #*n* status	The *n* th type approval status Declares whether this mode is enabled or disabled.	Read/write
Number of operator approvals	The number of operator approvals that the SDRF device can support. It is also used to help decode the tables. This differs from the type approvals in that it provides an additional level of qualification if needed which relates a type approval to a specific operator. This could be used to define additional/optional services by the operator that provides them.	Read only
Operator approval #1	The first operator approval reference	Read only
Operator approval #1 status	The first operator approval status. Declares whether this mode is enabled or disabled.	Read/write
Operator approval #*z*	The *z* th operator approval reference	Read only
Operator approval #*z* status	The *z* th operator approval status. Declares whether this mode is enabled or disabled.	Read/write

6.2.7.1 Approval coding

For a simple multi-band SDRF device such as a GSM phone operating at 900 MHz and 1800 MHz, the control structure could be constructed in one of several different ways:

[2] This support may have to be restricted initially because it can rapidly become quite cumbersome to implement and declare when many potential combinations are available. By having a parameter in this set of examples it acts as a reminder that such support may be required in the future. It is likely that the first implementations will declare this parameter to be 'exclusive'.

- It can be defined by using a separate type approval entry for each band.
- It can be defined using a single type approval entry to define the GSM support and two operator entries for the two different bands.
- It could be defined with no type approval entries and two operator entries for the two different bands.

To expand this to support an analog channels as well, an additional type approval and/or operator entry could be added. If the SDRF device could support both channels simultaneously, the entries would need to be marked as simultaneous instead of exclusive.

At some point, recommendations will be needed to clearly define the recommended way of choosing the preferred method of configuring these tables.

This example uses a combination of type and operator approvals to identify exactly the level and type of services that can be supported. It assumes that this information can be derived from a type approval or operator reference and this can be facilitated through the use of coded serial or reference numbers. This would, of course, require the co-operation and agreement of the regulatory bodies and operators. This would at a minimum be at a regional level but ideally should be at an international level.

6.2.8 Non-regulatory capability exchange

Parameter	**Definition**	**Read/write status**
Capability ID	A unique value that is used to identify a set of capabilities. The ability to support multiple sets is included for future expansion but it is anticipated that only one set of capability exchange information will be supported currently.	Read only
Capability size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of model definitions	The number of model definitions that the SDRF device can support. It is also used to help decode the tables.	Read only
Model #1 ID	The first model ID	Read only
Model #1 standard	Contains the communications standard(s) that the SDRF device supports .e.g. GSM or IS-95 or AMPS and so on. This should be a single standard. If the SDRF device supports multiple standards, this should be represented as separate model entries with each model supporting a standard. It should not be assumed that an entry here means that the SDRF device has approval to use the standard.	Read only
Model #1 Voice coding	Defines the different voice coding that can be used. e.g. half rate, full rate, enhanced full rate and so on.	Read only
Model #1 Data support	Defines the data rates and data types that can be supported. e.g. GSM 9600, GSM high speed circuit switched, GPRS and so on.	Read only
Model #1 contents support	Defines the contents support. E.g. email, fax, video, mixed	Read only
	• • •	
Model #n ID	The *n* th model ID	Read only
	These are the model parameters.	Read only
		Read only
	END OF CAPABILITY TABLE	Read only

6.2.9 Non-regulatory configuration

Parameter	**Definition**	**Read/write status**
Configuration ID	A unique value that is used to identify a set of configuration parameters. The ability to support multiple sets is made possible by this parameter.	Read only
Configuration size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of model definitions	The number of model definitions that the SDRF device can support. It is also used to help decode the tables.	Read only
Model #1 ID	The first model ID	Read only
Model #1 standard status	Contains the communications standard(s) that the SDRF device supports .e.g.GSM or IS-95 or AMPS and so on. This should be a single standard. If the SDRF device supports multiple standards, this should be represented as separate model entries with each model supporting a standard.	Read/write
Model #1 Voice coding status	Defines the different voice coding that can be used. e.g. half rate, full rate, enhanced full rate and so on.	Read/write
Model #1 Data support status	Defines the data rates and data types that can be supported. e.g. GSM 9600, GSM high speed circuit switched, GPRS and so on.	Read/write
Model #1 Contents support status	Defines the contents support. e.g., email, fax, voice, video and so on.	Read/write
	• • •	
Model #n ID	The *n* th model ID	Read/write
	These are the model parameters.	Read/write
		Read/write
	END OF CONFIGURATION TABLE	

At this level, the identification method has been expanded to operate at a slightly lower level. The example uses a concept of a SDRF device that allows the system level functions to be configured in terms of several different virtual models. Each model is a different configuration. For example, model #1 could be a GSM phone. Model #2 could also be a CDMA phone and so on. This can be developed further to create models where lower level differences are used: model #2 could be a GSM data phone, model #3 could be GSM voice and so on.

6.2.10 Detailed capability exchange

Parameter	Definition	Read/write status
Capability ID	A unique value that is used to identify a set of capabilities. The ability to support multiple sets is included for future expansion but it is anticipated that only one set of capability exchange information will be supported currently.	Read only
Capability size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of model definitions	The number of model definitions that the SDRF device can support. It is also used to help decode the tables.	Read only
Model #1 ID	The first model ID	Read only
Model #1 coding standard	Contains the communications standard(s) that the SDRF device supports .e.g. GSM or IS-95 or AMPS and so on. This should be a single standard. If the SDRF device supports multiple standards, this should be represented as separate model entries with each model supporting a standard.	Read only
Model #1 Frequency band	Contains the frequency bands that can be supported by the standard.	Read only
Model #1 Voice coding	Defines the different voice coding that can be used. e.g. half rate, full rate, enhanced full rate and so on.	Read only
Model #1 Data support	Defines the data rates and data types that can be supported. e.g. GSM 9600, GSM high speed circuit switched, GPRS and so on.	Read only
Model #1 Messaging support	Defines the messaging support.	Read only
Other parameters	To be defined	
Model #1 Equaliser	The ID to identify the equaliser for the model	Read only
Version number	Used for version control.	Read only
Equaliser scheme	e.g. DFE, MLSE, RAKE	Read only
Correlator parameters	To be defined	Read only
Number of multipaths	Defines the minimum and maximum values.	Read only
PN codes	To be defined.	Read only
Soft/hard decision support	Indicates the if either or both decision schemes are supported.	Read only

Acquisition time	Defines the minimum and maximum values that the model is capable of supporting.	Read only
Training scheme	To be defined.	Read only
Other parameters	To be decided	
Model #1 Modulator	The ID to identify the modulation for the model	Read only
Version number	Used for version control.	Read only
Modulation scheme	Defines the modulation schemes that are supported.	Read only
Modulation parameters	These are to be defined but would encompass the modulation schemes described in the previous parameter.	Read only
Clock frequency	Self-explanatory.	Read only
Bit error rate	To be defined. Some modulation schemes will change depending on the fed back bit error rate.	Read only
Channel framing structure	To be defined. This may need expanding to provide sufficient detail.	Read only
Analog/digital in	Defines whether the input is analog or digital.	Read only
Analog/digital out	Defines whether the output is analog or digital.	Read only
Digital in word size	Defines the supported input word size in the digital domain. If digital input is not supported, this field is blank.	Read only
Digital in sample rate	Defines the supported input sample rate for the digital conversion. If digital input is not supported, this field is blank.	Read only
Digital out word size	Defines the supported output word size in the digital domain. If digital input is not supported, this field is blank.	Read only
Digital out sample rate	Defines the supported output sample rate for the digital conversion. If digital input is not supported, this field is blank.	
Quality	To be defined.	Read only
Resolution	To be defined.	Read only
Channel level	To be defined	Read only
Other parameters	To be defined	
Model #1 Radio Tuning	The ID to identify the radio tuning module for the model.	Read only
Version number	The version number.	Read only
Synthesis parameters	To be defined.	Read only
Frequency	Defines the minimum and maximum frequency that can be used.	Read only
Step size	Defines the frequency step sizes that can be	Read only

	supported. This may require some expansion to cope with different step/frequency combinations.	
Jitter	To be defined.	Read only
Phase noise	To be defined.	Read only
Settling time	To be defined.	Read only
Delay to implement change	To be defined. This may also vary with the frequency or size of change and therefore be a set of values.	
Output power	To be defined. This may also vary with the frequency therefore be a set of values.	Read only
Frequency description	The equivalent to the waveform description.	Read only
Oscillator source	To be defined.	Read only
Synthesiser source	To be defined.	Read only
Other parameters	To be defined	
Model #1 Channel encoder	The ID to identify the channel encoder module for the model.	Read only
Version number	The version number.	Read only
Training sequence	To be defined.	Read only
Tail bit sequence	To be defined.	Read only
Framing structure	To be defined.	Read only
Clock frequency	To be defined.	
Interleaving	To be defined.	Read only
Reed-Solomon parameters	To be defined.	Read only
Viterbi parameters	To be defined.	Read only
Convolutional codes	To be defined.	Read only
Block codes	To be defined.	Read only
Processing delay	To be defined.	Read only
Other parameters	To be defined	
Model #1 Channel decoder	The ID to identify the channel decoder module for the model.	Read only
Version number	The version number.	Read only
Training sequence	To be defined.	Read only
Tail bit sequence	To be defined.	Read only
Framing structure	To be defined.	Read only
Clock frequency	To be defined.	Read only
De-Interleaving	To be defined.	Read only
Convolutional codes	To be defined.	Read only
Block codes	To be defined.	Read only
Processing delay	To be defined.	Read only
Other Parameters	To be defined	

	END OF CAPABILITY TABLE	

6.2.11 Detailed configuration

Parameter	**Definition**	**Read/write status**
Configuration ID	A unique value that is used to identify a set of configuration parameters. The ability to support multiple sets is made possible by this parameter.	Read only
Configuration table size	The number of bytes that is contained in the table and is used to decode the information.	Read only
Number of model definitions	The number of model definitions that the SDRF device can support. It is also used to help decode the tables.	Read only
Model #1 ID	The first model ID	Read only
Model #1 coding standard	Contains the communications standard(s) that the SDRF device supports .e.g. GSM or IS-95 or AMPS and so on. This should be a single standard. If the SDRF device supports multiple standards, this should be represented as separate model entries with each model supporting a standard.	Read/write
Model #1 Frequency band	Contains the frequency bands that can be supported by the standard.	Read/write
Model #1 Voice coding	Defines the different voice coding that can be used. e.g. half rate, full rate, enhanced full rate and so on.	Read/write
Model #1 Data support	Defines the data rates and data types that can be supported. e.g. GSM 9600, GSM high speed circuit switched, GPRS and so on.	Read/write
Model #1 Messaging support	Defines the messaging support.	Read/write
Other parameters	To be defined	
Model #1 Equaliser	The ID to identify the equaliser for the model	Read only
Version number	Used for version control.	Read only
Equaliser scheme	e.g. DFE, MLSE, RAKE	Read/write
Correlator parameters	To be defined	Read/write
Number of multipaths	Defines the minimum and maximum values.	Read/write
PN codes	To be defined.	Read/write
Soft/hard decision support	Indicates the if either or both decision schemes are supported.	Read/write
Acquisition time	Defines the minimum and maximum values that	Read/write

	the model is capable of supporting.	
Training scheme	To be defined.	Read/write
Other parameters	To be defined	
Model #1 Modulator	The ID to identify the modulation for the model	Read only
Version number	Used for version control.	Read/write
Modulation scheme	Defines the current modulation scheme.	Read/write
Modulation parameters	These are to be defined but would are dependent on the current modulation scheme.	Read/write
Clock frequency	Self-explanatory.	Read/write
Bit error rate	To be defined. Some modulation schemes will change depending on the fed back bit error rate.	Read/write
Channel framing structure	To be defined. This may need expanding to provide sufficient detail.	Read/write
Analog/digital in	This defines whether the input is analog or digital.	Read/write
Analog/digital out	This defines whether the output is analog or digital.	Read/write
Digital in word size	This defines the supported input word size in the digital domain. If digital input is not supported, this field is blank.	Read/write
Digital in sample rate	This defines the supported input sample rate for the digital conversion. If digital input is not supported, this field is blank.	Read/write
Digital out word size	This defines the supported output word size in the digital domain. If digital input is not supported, this field is blank.	Read/write
Digital out sample rate	This defines the supported output sample rate for the digital conversion. If digital input is not supported, this field is blank.	Read/write
Quality	To be defined.	Read/write
Resolution	To be defined.	Read/write
Channel level	To be defined	Read/write
Other parameters	To be defined	
Model #1 Radio Tuning	The ID to identify the radio tuning module for the model.	Read only
Version number	The version number.	Read only
Synthesis parameters	To be defined.	Read/write
Frequency	Defines the minimum and maximum frequency that can be used.	Read/write
Step size	Defines the frequency step sizes that can be supported. This may require some expansion to	Read/write

	cope with different step/frequency combinations.	
Jitter	To be defined.	Read/write
Phase noise	To be defined.	Read/write
Settling time	To be defined.	Read/write
Delay to implement change	To be defined. This may also vary with the frequency or size of change and therefore be a set of values.	Read/write
Output power	To be defined. This may also vary with the frequency therefore be a set of values.	Read/write
Frequency description	The equivalent to the waveform description.	Read/write
Oscillator source	To be defined.	Read/write
Synthesiser source	To be defined.	Read/write
Other parameters	To be defined	
Model #1 Channel encoder	The ID to identify the channel encoder module for the model.	Read/write
Version number	The version number.	Read/write
Training sequence	To be defined.	Read/write
Tail bit sequence	To be defined.	Read/write
Framing structure	To be defined.	Read/write
Clock frequency	To be defined.	Read/write
Interleaving	To be defined.	Read/write
Reed-Solomon parameters	To be defined.	Read/write
Viterbi parameters	To be defined.	Read/write
Convolutional codes	To be defined.	Read/write
Block codes	To be defined.	Read/write
Processing delay	To be defined.	Read/write
Other parameters	To be defined	
Model #1 Channel decoder	The ID to identify the channel decoder module for the model.	Read only
Version number	The version number.	Read only
Training sequence	To be defined.	Read/write
Tail bit sequence	To be defined.	Read/write
Framing structure	To be defined.	Read/write
Clock frequency	To be defined.	Read/write
De-Interleaving	To be defined.	Read/write
Convolutional codes	To be defined.	Read/write
Block codes	To be defined.	Read/write
Processing delay	To be defined.	Read/write
Other parameters	To be defined	
	END OF CONFIGURATION TABLE	

6.2.12 Mode Switcher Scenario Flow Charts

The remainder of this section describes three scenarios through the use of flow charts which are examples of the how the Mode switcher API and messages can be used to initiate power-on, module download and cross-technology or air interface roaming.

6.2.13 Power-on

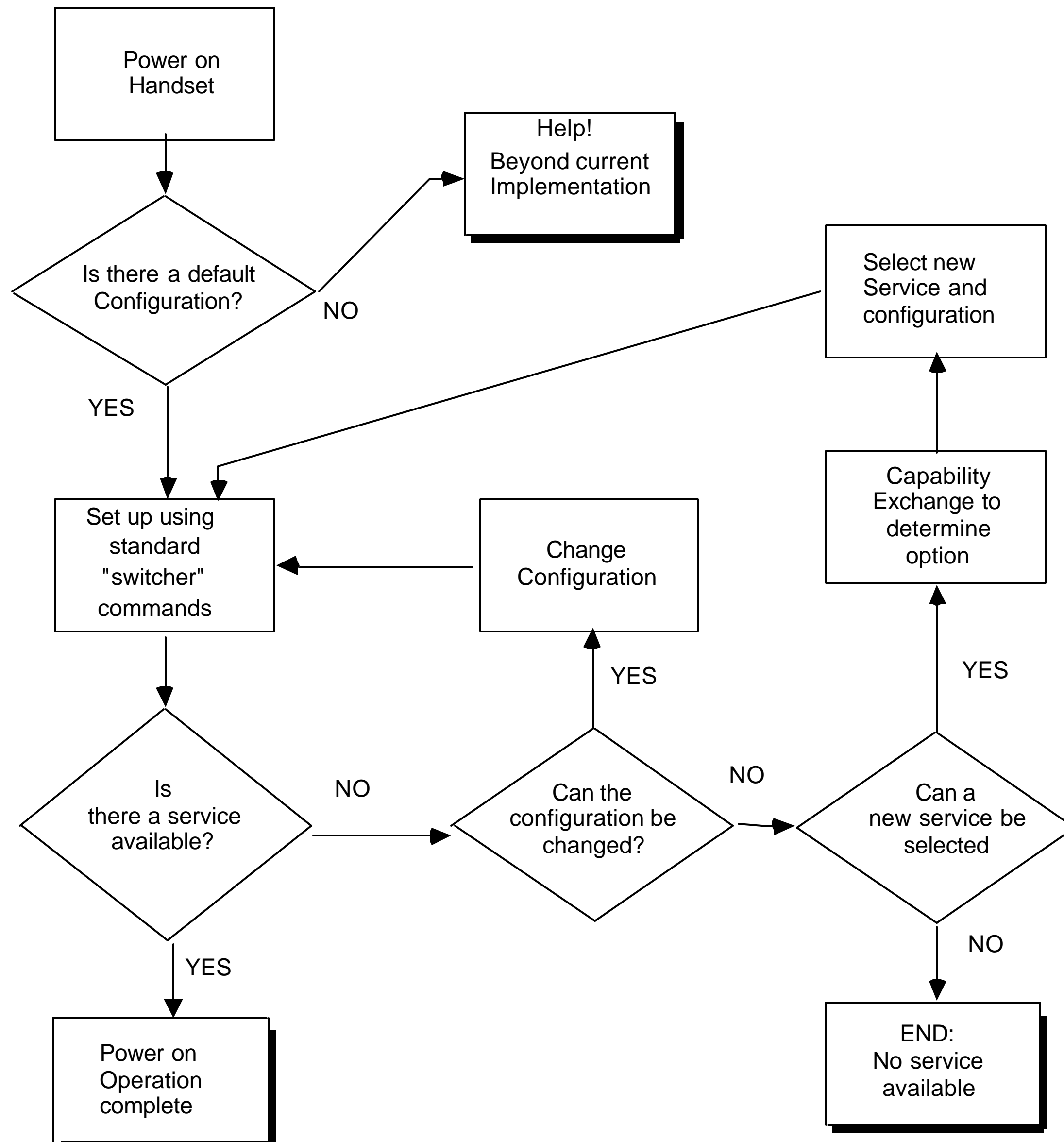


Figure 6.2.13-1 The power-on flow chart

The flow chart essentially consists of two loops that try and register with a service. The first loop will change the configuration parameters for the selected service until either a successful connection has been

made or all the possibilities have been exhausted. At this point an alternative service is selected (if available) and the process repeated. This flow chart does not include the possibility of downloading further software for untried services.

6.2.13.1 Power-on Start

This is the event that brings up the SDRF device from a dormant state. It need not be assumed that it is locally created although this will probably be the main route. Examples include switching the SDRF device on or receiving a remote command to power up from an external source. This can be the result of a warm or cold start. Within the scope of the example, this is not explicitly defined.

6.2.13.2 Is There a Default?

This question asks if there is a default configuration that is known. This data is not part of the Mode switcher capability exchange information and is assumed to be available from elsewhere in the system.

If the answer is yes, this information is used to identify which of the Mode switcher configurations to use as the default. This may involve either selecting a configuration or configuring one as required.

If the answer is no, then some intelligence is required to resolve this situation. This is beyond the scope of this example. Without a default start-up configuration, the system may simply choose one. This could then create difficulties with existing services if the chosen service is unsupported and interferes with the existing services. In this case, listening to a pilot channel or scanning the frequency bands may be the only solution.

6.2.13.3 Standard Mode Switcher Set Up

This is the procedure as defined earlier in the API section that uses the Mode switcher API commands to enable, start and set the configuration of the required virtual capability.
The configuration is selected and set-up based on the information that defines the default.
It may use the regulatory, non-regulatory or detailed level of details

6.2.13.4 Is There a Service?

This checks to see if the set-up procedure described in the previous box has resulted in a service registration and connection

If the answer is yes, then the power on operation has been successful and the operation terminates with a successful completion.

If no, then the next stage is to modify the current configuration to see if this will create a successful connection. This starts a loop process that will cycle through different configurations.

6.2.13.5 Can The Configuration Be Changed?

This reflects the ability of the power-on module to determine if and how the current configuration can be changed.

It will need the virtual capabilities through the capability exchange facility.

If yes, this will cause the current configuration to be modified. If no, then modification of the current configuration is not possible and an alternative virtual capability must be used.

6.2.13.6 Change Configuration

This is where the current configuration is modified on the basis of the changes identified in the previous decision diamond.

The modified configuration information is then fed back into the Mode switcher set-up to create a new configuration and to see if this is successful in gaining a valid connection.

6.2.13.7 Choose Another Service?

This decision is reached if no modification of the current configuration can be done. At this point a decision has to be made concerning which new service is to be tried.

If no, it will terminate.

If yes, then further work to determine which service and configuration will be used next.

6.2.13.8 Capability Exchange

This will interrogate the virtual capabilities via the Mode switcher's capability exchange. This information is needed for the next stage.

6.2.13.9 Select Service

This is where the next service is selected based on the information supplied from the capability exchange.
Again this assumes that a suitable algorithm has been developed for the Power On module.
Once a selection has been made, this information is used to set-up the service (yet again) to try and get a successful connection.

6.2.14 Download and Installation

This scenario describes the basic process for downloading and installing a new module. It has a basic loop that chooses and downloads a module and then adjusts the module configuration till it can fit and run in the system. The download process is described in section 5.2.1 Download API.

The diagram has an additional entry point B and an exit point A. These fit into the roaming scenario described in the next section.

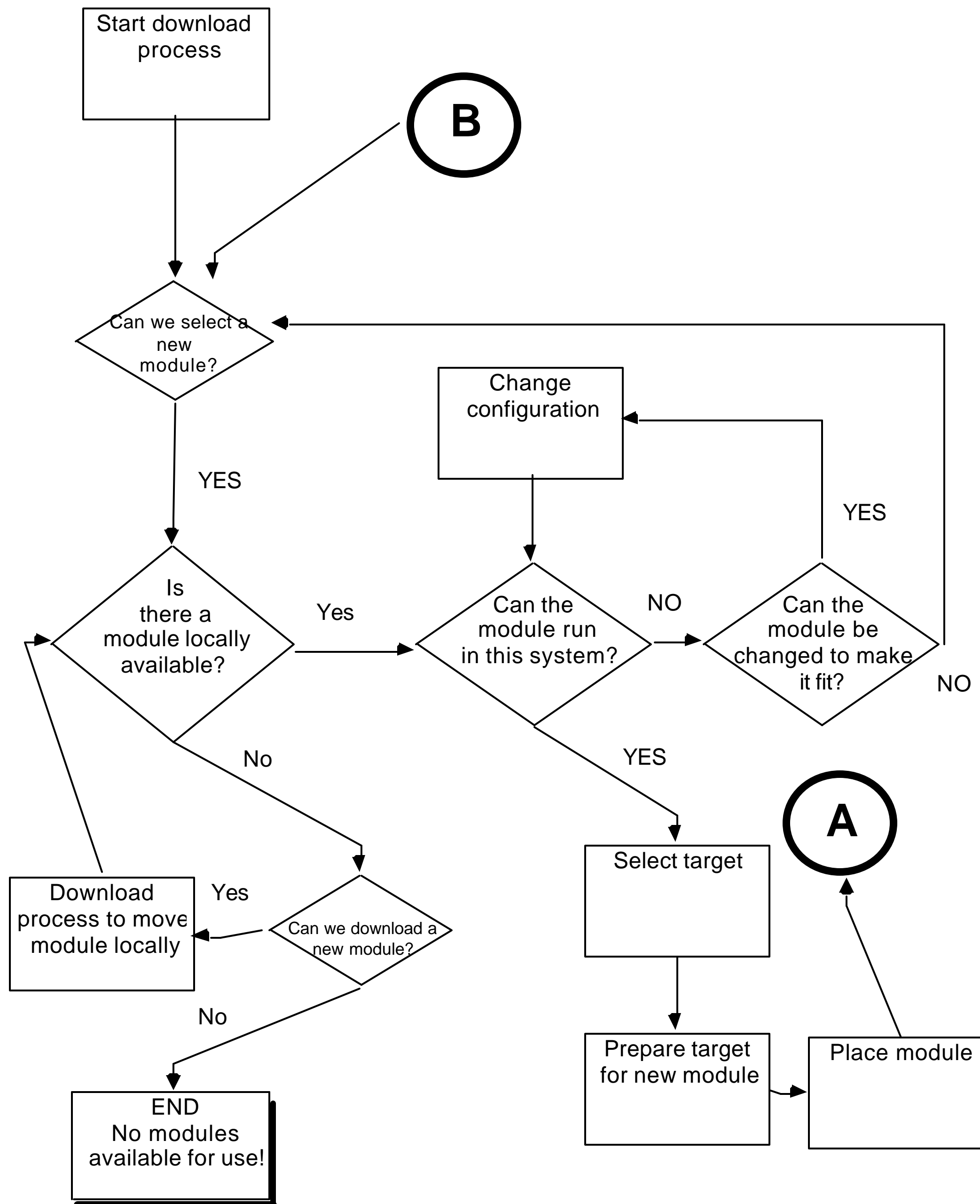


Figure 6.2.14-1 The download scenario

The select target stage is where a decision is made concerning where the module is to be placed, and perhaps more importantly, which module, if any, is replaced.

6.2.15 Cross-technology Roaming

The roaming scenario is started either by detecting that a service is degrading sufficiently to warrant roaming to a replacement or via an instruction from the infrastructure.

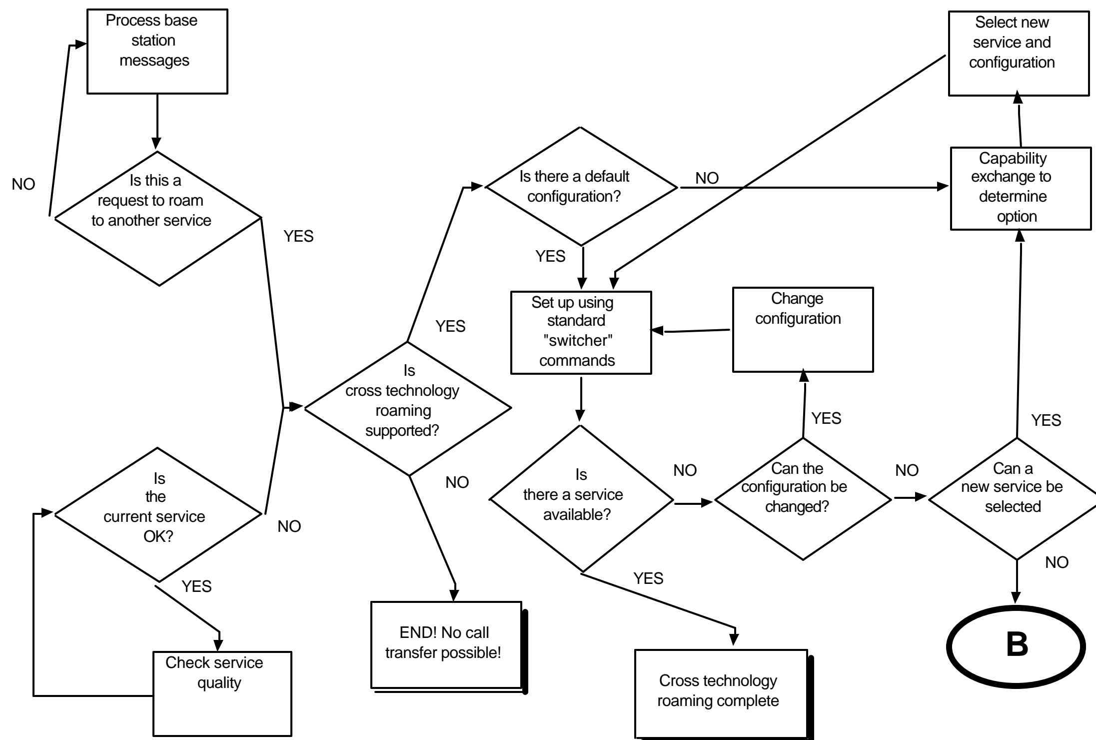


Figure 6.2.15-1 The roaming scenario

In either case, a check is then performed to ensure that cross-roaming is supported. The flow chart then moves into a flow similar to that of the power-on sequence in that different configurations/services are repeatedly tried until a successful replacement is made.

The scenario can be expanded by adding the download support using points A and B to allow modules to be downloaded to access the replacement service if the necessary software to support the connections is not available in the SDRF device.

7.0 Form Factor

7.1 Handheld Form Factor

Specific physical interfaces that could be candidates for handheld form factor recommendations will be compliant with the guidelines for functional interfaces and physical modularity as defined by SDRF approach to open system standards recommendations.

Handheld devices are continuously being aggressively driven towards higher levels of integration and hence smaller form factors by the highly competitive commercial marketplace.

The physical modularity is continuously varying across functional interfaces internal to the handheld device. Physical modularity is most stable at the external interfaces and the following interfaces have been identified as potential candidates for form factor recommendations:

- Antenna to RF
- RF to Modem
- User I/O to locally attached machine
- Battery connection
- SIM connection

7.2 Mobile Form Factor

Specific recommendation for form factor and interconnects are still under development and will be included in a later revision to this document.

7.3 Interconnect Options

Appendix E provides examples of common interconnect standards that are appropriate for consideration in establishing an open architecture for SDRF. Then, based on requirements in the handheld and mobile areas, a subset will be recommended. A process and criteria for selecting this subset will be developed and published.

8.0 Plan for Future Work

It is the intention of SDRF in the process of developing standards recommendations to start with a very broad approach to describing the general architecture of the solution set, the modularization of the solution, and the identification of those items that will be standardized as well as those that will not be standardized. Successive iterations will refine those definitions and will become more focused. Figure 8.0-1 provides a view of the focusing of the standards recommendations development process. It is shown as a funnel to graphically illustrate the intention of the SDRF to start with very broad definitions and architectures and proceed down levels of detail only to the point where the needs of the industry are met with the minimum limitation on innovation. This graphical representation served the SDRF well through the TR 1.X series. The activities of the Forum have become too complex to fit in this graphical representation going forward. Therefore with TR 2.0, Figure 8.0-2 was introduced. It shows the relationship of vertical Working Groups focused on one application area with horizontal Task Groups that cut across all of the application areas. With TR 2.1 a textual description approach is used.

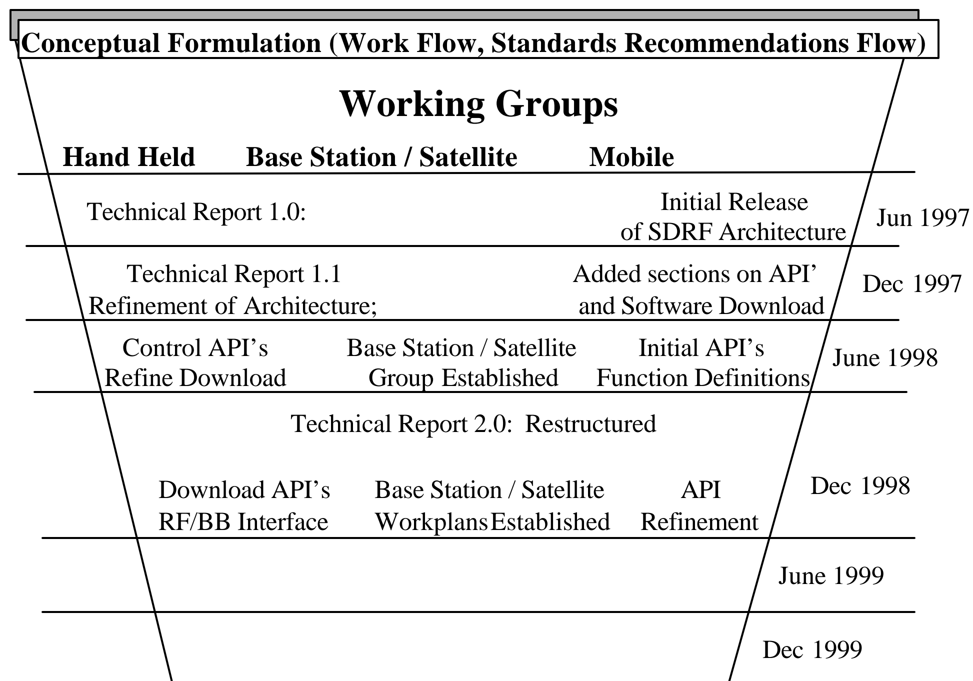


Figure 8.0-1 Standards Recommendations Development Overview

	Handheld Working Group	Base Station Working Group	Mobile Working Group
Switcher / Download Task Group			
API Task Group			
Antenna API Task Group			

Figure 8.0-2 Relationship of SDRF Vertical Working Groups to Horizontal Task Groups

8.1 Mobile Working Group Work Plan - 2000

Goal: To have some member companies implement a SDRF architecture that executes a common software radio application.

1. Define interfaces to the SDR services.
   - Identify and define the API's (Feb 2000)
2. Define the management structures for SDR control (April 2000)
3. Finalize Domain profile (May 2000)
4. Finalize POSIX profiles (Feb 2000)
5. Define at least one common software radio application for the November 2000 test of submitted SDRF architecture implementations. (Sep 2000)
6. Update the SDRF architecture based on lessons learned from the real implementations. (Dec 2000)
7. Finish off addressing the 11 work items from Stockholm. (Jan 15/2000)
8. Post on the web site an overview on the SDRF architecture. (Feb 2000)
9. Add Glossary to define CF objects (Feb 2000)
10. Create a rationale document for CF (May 2000)
11. Socialize the CF to other SDR groups and outside to OMG?
12. Framework accepted by other SDRF groups esp. Base-station. (June 2000)

8.2 Base Station Working Group Work plan - 2000

- By the end of the February meeting the group will finalize the description of Use Cases with priority number 1. It will evaluate the TR 2.1 reference architecture with respect to these Use Cases and consider changes as needed to support them (e.g. time criticality, data path utilization, memory efficiency and organization). It will define and develop a specification (methods and control signals) for a Base station with Type I and II Antenna sub-systems with respect to input and output parameters (gain, frequency etc) in UML or XML. At the April meeting the group will vote Use Cases with priority number 1 with the Technical committee and begin description of Use Cases with priority 2. It will work with the mobile group to recommend changes to the reference architecture for base station application. It will vote the specification for a Type I and II Antenna sub-systems.
- In June the group will vote Use Cases with priority numbers 1 and 2 as ready to deploy in a request for comments form on the Forum web site, and begin description of Use Cases with priority 3. It will vote an update to the reference architecture to support base station application. It will begin an evaluation of the specification for a Base station with Type III and IV Antenna sub-systems with respect to inputs and outputs. At the September meeting the group will vote Use Cases with priority 3 and develop Use Cases with priority 4. It will vote the specification for a

Type III and IV Antenna sub-systems. In November the group will vote Use Cases with priority number 4 with the Technical committee.

8.3 Handheld Working Group Work Plan – 2000

The Handheld Working Group will review the handheld architecture in the context of the mobile framework. It will revise the handheld architecture as appropriate. Then the group will define a subset of interfaces to document. These interfaces will be specified to the level of detail required to be useful to an implementor by June 2000.

8.4 Switcher / Download Working Group Work Plan - 2000

November Meeting Follow-up:
- Complete and post (SDR Forum website) TR2.0, Ch. 6 revisions
- Complete and submit overview document to WAP, MexE

February Meeting:
- Complete voting on TR2.0, Ch. 6.
- Initial Work plans back from WAP and MexE

April Meeting:
- Joint meeting with MexE prior to Korea meeting
- Initiate discussions with SUN on JAVA

June Meeting:
- Joint meeting with WAP prior to Seattle meeting

September Meeting:
- Vote SDR Forum contribution to WAP/MexE out of Forum

November Meeting:
- Download protocol Version 1 completed and introduced in MexE requirements and WAP

8.5 Antenna API Task Group

The Antenna API Task Group will be organized by June of 1999 and it will publish its initial work plan and objectives.

9.0 Glossary

DEFINITIONS

Applets	An applet is a small program that is not intended to be run on its own, but to be embedded inside another application.
Architecture	The design principles, physical configuration, functional organization, operational procedures, and data formats used as the bases for the design, construction, modification, and the operation of a product, process, or element.
Economies of Scale	1) Decreasing unit costs when the scale of operation is increased; and 2) decreasing costs associated with joint production.
Extensibility	The ability to readily permit an addition of a new element, function, control, or capability within the existing framework. In SDRF, this may be new, evolving wireless services.
Feature	A specific element of a service that provides a desirable result. Examples are encryption and authentication.
Function	An operation or algorithm. Examples are down conversion and demodulation.
Functional Partitioning	A logical grouping of functions into identifiable functional blocks for the purpose of implementing a service or mode within an architecture comprised of these functional blocks.
Mode	A specific implementation type of a service. Examples are AMPS, GSM, or GPS.
Module	1) An interchangeable subassembly that constitutes part of, i.e., is integrated into, a larger device or system. 2) In computer programming, a program unit that is discrete and identifiable with respect to compiling, combining with other modules, and loading.
Multi-Function	Capable of operating in a number of different communications services with a single piece of equipment.
Multimode	Support multiple modulation formats and modulation bandwidths QAM, PSK, FSK, MSK, DSSS various bit rates and symbol rates
Multiband	Support Multiple frequency Bands of Operation Cellular 800 MHz PCS 1.9 GHz ISM .9, 2.4, 5.8 GHz Private Land Mobile Radio (PLMR) Multiple bands: 30- 900 MHz
Multi-Standard	Specific Standard Supported Cellular AMPS, IS-54, IS-95 PCS APCO - 25 (Public Safety)

	MIL-STD-188-XYZ
Open system	A system with characteristics that comply with specified, publicly maintained, readily available standards and that therefore can be connected to other systems that comply with these same standards. (Open Architecture)
Paging	A one-way communications service from a base station to mobile or fixed receivers that provide signaling or information transfer by such means as tone, tone-voice, tactile, optical readout, etc.
Refarming	The process of moving incumbent authorized users out of one frequency band into another. It is likely to be extended to the process of moving users from one mode to another mode.
Scalability	The ability to extend the functionality of the SDRF device to include multiple channels and networking or additional local connectivity and processing.
SDRF	Software Defined Radio uses adaptable software and flexible hardware platforms to alter or change its functional characteristics.
Service (OSI)	In the Open Systems Interconnection Reference Model (OSI RM), a capability of a given layer, and the layers below it, that (a) is provided to the entities of the next higher layer and (b) for a given layer, is provided at the interface between the given layer and the next higher layer.
Service (SDRF)	Capability or a defined and interrelated set of capabilities structured to meet a specific requirement.
Service Access	In personal communications service (PCS), the ability for the network to provide user access to features and to accept user service requests specifying the type of bearer services or supplementary service that the users want to receive from the PCS network.
Service Domain	Association of a service to a particular implementation domain. Examples are cellular and satellite voice.
Upgradeability	The ability to get more or better work from the SDRF device through the insertion of improved hardware and software technologies.
User Service	Service designed to meet a user requirement. Examples are voice and data user services.

ACRONYMS

AAW	Anti-Air Warfare
ACELP	Algebraic Code Excited Linear Prediction
ADNS	Automated Digital Network System
ADPCM	Adaptive differential pulse-code modulation: a method of digitally encoding speech signals
AGC	Automatic Gain Control
ALE	Automatic Link Establishment
ALOHA	A simple multiple access protocol invented at the University of Hawaii in which users transmit whenever they have something to send. A variant that offers

	greater throughput is "slotted" ALOHA, in which transmissions are synchronized to a universal clock.
AMPS	Advanced mobile phone system: the American analog cellular telephone system
ANSI	American National Standards Institute
APCO	Associated Public Safety Communications Officers, Inc.
API	Application Program Interface
ARDIS	Motorola wireless two-way data network
ASuW	Anti-Surface Warfare
ASW	Anti-Submarine Warfare
ATM	Asynchronous transfer mode: a packetized digital transfer system, adopted for the B-ISDN. (Beware: the same abbreviation is used for automated teller machines, i.e., "hole-in-the-wall" bank cash machines.)
BB	Base Band
BER	Bit error ratio
B-ISDN	Broadband integrated services digital network
BPSK	Binary phase shift keying
C/I	Carrier-to-interference ratio, usually expressed in dB
CCITT	Comite' Consultatif International de Radio: formerly the ITU body responsible for radio standards (now the responsibility of ITU-R). Comite' Consultatif International Te1egraphique et Te1ephonique: formerly the ITU body responsible for nonradio standards (now the responsibility of ITU-T).
CDCS	Continuous dynamic channel selection: a channel management technique used in DECT
CDF	Cumulative distribution function: the integral of the PDF
CDMA	Code Division Multiple Access
CDPD	Cellular Digital Packet Data
CELP	Code-excited linear prediction
CISC	Complex Instruction Set Computer
CNI	Communication, Navigation, and Identification (CNI)
COMSEC	Communication Security
COTS	Commercial-off-the-shelf Systems (Software)
CSE	Cross Standards Extensions
CSMA	Carrier sense multiple access: a multiple-access protocol that offers improved performance over ALOHA, users being required to listen for a quiet channel before transmitting.
CT2	Second-generation cordless telephone
CT3	Third-generation cordless telephone
CT0, CT1	Early cordless telephone standards
CTR	Common technical regulations: the basis for type-approval of, for example, GSM handsets
CUG	Closed user group
CW	Carrier wave, that is, a constant, unmodulated radio carrier
DAMA	Demand Assignment Multiple Access

DAMPS	Digital AMPS: a digital cellular system having some compatibility with the (analog) AMPS system (U.S.)
DARPA	Defense Advanced Research Project Agency
DCS1800	Digital communication system: a variant of the GSM standard providing for operation in the 1800-MHz band, initially required by the United Kingdom for its PCN service.
DECT	Digital European Cordless Telecommunications: the second-generation cordless system standardized by ETSI
DIN	Deutsche Industrie-Norm(enaussxhuss) (German Industrial Standards Authority, equivalent of EIA, BSA etc.)
DLC	Data link control (layer)
DOD	Department of Defense
DQPSK	Differential Quadrature Phase Shift Keying
DS	Direct sequence: a form of spread-spectrum system, using a pseudorandom binary stream to spread the signal
DTMF	Dual-tone multi-frequency: system of low-speed signaling in telephone systems, for example, for dialing, using paired audio tones
DWTS	Defense Wide Transmission Systems
EISA	Enhanced Industry Standard Architecture
EPLRS	Enhanced Position Location Reporting System
ES	Emergency Service
ESMR	Enhanced Specialized Mobile Radio
ETS	European Telecommunications Standard
ETSI	European Telecommunications Standards Institute
EU	European Union
EW	Electronic Warfare
FCC	Federal Communications Commission: U.S. Government regulatory body
FD	Full Duplex
FDD	Frequency Division Duplexing
FDDI	Fiber distributed data interface: a U.S. standard for high-rate fiber optic token-ring LAN systems
FDMA	Frequency Division Multiple Access
FEC	Forward Error Correction
FH	Frequency hopping: a form of spread-spectrum system
FPLMTS	Future public land mobile telecommunication system: the ITU name for third-generation systems. Since this name is neither memorable nor pronounceable in any language, the name IMT-2000 has been proposed.
FSK	Frequency Shift Keying
GFSK	Gaussian-minimum Frequency Shift Keying
GHz	Gigahertz
GMSK	Gaussian minimum shift keying: a form of constant-envelope binary digital modulation
GoS	Grade of service: in telephony, the probability that a call will not succeed. Note that a high GoS is worse than a low one. Also used loosely to mean service quality.
GPS	Global Positioning System
GSM	Groupe Special Mobile: originally, the CEPT (later, ETSI) committee responsible for the pan-European digital cellular standard. Also, used as the name of the

	system and the service. Global System for Mobile (Communication): the name for the GSM system and service, invented by a group of European operators, to fit the abbreviation "GSM."
HD	Half Duplex
HMI	Human-Machine Interface
HMMWV	High Mobility Multi-purpose Wheeled Vehicle
I/O	Input / Output
IBCN	Integrated broadband communications network
IEEE	Institute of Electrical and Electronic Engineers (U.S.). IEEE-802 is a committee responsible for developing standards for LANs.
IF	Intermediate Frequency
IFF	Identification Friend or Foe
IMT-2000	International mobile telecommunications-2000: a name proposed within ITU for their third-generation concept, otherwise known as FPLMTS
IN	intelligent network
INFOSEC	Information Security
IPR	Intellectual property rights, including patents, trademarks, copyrights
ISA	Industry Standard Architecture
ISDN	Integrated services digital network
ISM	Industrial, Scientific, Medical
ISO	International Standards Organization
ITU	International Telecommunication Union
JMCOMS	Joint Maritime Communications Strategy
JTIDS	Joint Tactical Information Distribution System
kHz	Kilohertz
LAN	Local area network
LED	Light-emitting diode
LEO	Low earth (satellite) orbit
LOS	Line of sight (radio path)
LPD	Low Probability of detection
LPI	Low probability of intercept: a property of spread-spectrum systems
MAC	Medium access control (protocol layer)
MAN	Metropolitan area network
MAP	Mobile application part: an extension of signaling system number 7, providing support of mobile systems.
MBLT	Multiplexed Block Transfer
MCM	Mine Countermeasure
MCN	Microcellular network
MEO	Medium earth (satellite) orbit
MHz	Megahertz
MMI	Man-Machine Interface
MPMLQ	Multipulse maximum likelihood quantized
MSK	Minimum Shift Keying

NATO	North Atlantic Treaty Organization
NET	Norme Europeenne de Te1ecommunications: formerly, the specification for type-approval, produced by CEPT, later replaced by CTRs. Thus, NET-10 was the type-approval spec for GSM, now replaced by CTR-5 and CTR-9.
NMT	Nordic mobile telephone (system): cellular telephone system prevalent in the Nordic countries.
NOS	Network Operating System
NPRM	Notice of Proposed Rule Making: an official pronouncement by the FCC (U.S.)
NTDR	Near-Term Digital Radio
NTT	Nippon Telegraph and Telephone Corporation (Japan)
OFTEL	Office of Telecommunications: U.K. official body created to protect the interests of consumers of telecommunication services.
OKQPSK	Offset-keyed quadrature phase-shift keying: digital modulation system in which in-phase and quadrature components carry bit-streams offset by half a bit, resulting in desirable spectral and envelope characteristics.
OS	Operating System
OSI	Open systems interconnection: the ISO layered protocol model
OSS	Operating Support System
PABX	Private automatic branch exchange
PACS	Personal Access Communications System, Licensed Band
PAD	Packet assembler/disassembler: device used to interface with a packet network
PBX	Private Branch Exchange
PCI	Peripheral Component Interconnect
PCIA	Personal Communications Industry Association (U.S.).
PCMCIA	Personal Computer Memory Card Industry Association
PCN	Personal communications network: used as a general term for such networks, and specifically for the UK systems licensed for operation in the 1800-MHz band and using the DCS 1800 standards
PCS	Personal Communication System
PDA	Personal digital assistant: a name coined to describe a portable, screenbased communication-oriented data terminal or organizer.
PDC	Personal Digital Cellular
PDF	Probability distribution function
PDH	Plesiochronous digital hierarchy: transmission system standard (plesiochronous = near-synchronous)
PHS	Personal handy phone system: a RCR standard for cordless telephony
PICMG	PCI Industrial Manufacturers Group
PIN	Personal identification number: a (typically four-digit) secret number to be input by the user to obtain service
PLMN	Public Land Mobile Network
PMR	Private mobile radio
POCSAG	Post Office Code Standardization Advisory Group
POTS	Plain old telephone service
PSI-CELP	Pitch Synchronous Innovation Code Excited Linear Prediction
PSK	Phase-shift keying
PSPDN	Packet-switched public data network
PSTN	Public-switched telephone network
PTO	Public telecommunications operator
PTT	Post, telephone and telegraph (authorities): an old name for the (usually state-monopoly) operators of telecommunication and postal services
QAM	Quadrature amplitude modulation

QCELP	Qualcomm proprietary CELP
QoS	Quality of service: an ill-defined term covering various measures of "quality" in telecommunication systems
QPSK	Quadrature phase-shift keying
RAM	A wireless mobile data system
RES	Radio equipment and systems: a technical committee of ETSI, responsible for terrestrial radio standards other than GSM
RF	Radio Frequency
RP-CELP	Regular pulse-code excited linear prediction
RPE-LTP	Regular-Pulse Excitation Long-Term Prediction
RTK	Real-Time Kernel
SCI	Scaleable Coherent Interconnect
SCSA	Signal Computing System Architecture
SCSI	Small Computer System Interface
SDH	Synchronous digital hierarchy: a transmission system standard
SDO	Standards development organization
SDR	Software defined radio
SDRF	Software Defined Radio Forum
SEM-E	Standard Electronic Module, Defined by Mil Std 1389 Appendix E
SFH	Slow frequency hopping
SIM	Subscriber Identification Module - derived from GSM
SINCGARS	Single-Channel Ground and Airborne Radio System
SMG	Special Mobile Group: the name adopted by ETSI for the (former) GSM committee, for consistency with other ETSI Technical Committees, which all had English names. There are numerous subgroups: SMGI, SMG2, etc. The SMG5 subgroup has responsibility for work on third-generation systems.
SNR	Signal-to-noise ratio, usually expressed in dB
S-PCN	Satellite personal communications system
SS7	Signaling system number 7: an ITU standard for telecommunications network signaling
T1	U.S. standards committee, active in personal communications area
TACAN	Tactical air navigation
TACS	Total access communication system: an analog cellular phone system based on AMPS, used in the United Kingdom and elsewhere. Technical basis for regulation.
TCM	Trellis coded modulation
TCP/IP	Transmission Control Protocol/Internet Protocol
TDD	Time-division duplex: two-way communication using synchronized alternate transmission on a single carrier
TDHS	Time domain harmonic scaling
TDMA	Time Division Multiple Access
TEMPEST	Transient Electromagnetic Pulse Standard

TETRA	Trans-European trunked radio: second-generation digital PMR system standardized by ETSI
TGMS	Third-generation mobile systems
TIA	Telecommunications Industry Association (U.S.)
TRANSEC	Transmission Security
TTM	Time to Market
UAV	unmanned aerial vehicle
UHF	Ultra high frequency: usually defined as 0.3 to 3 GHz
UMTS	Universal mobile telecommunication system (or service): the concept of third-generation systems developed in Europe, particularly by the RACE program.
UPT	Universal personal telecommunication
VA	Voice activation
VERSA	Motorola defined micro-computer bus (1979)
VLF	Very Low Frequency
VME	VERSA Module Eurocard
VRC-99	Packet Radio
VSB	VME Subsystem Bus
VSELP	Vector Sum Excited Linear Prediction
WAN	Wide area network
WARC	World Administrative Radio Conference
WLLAN	Wireless local area network
WSS	Wide-sense stationary

Appendix A. The SDRF Charter

The following is the SDRF charter. It is also available on the SDRF web site:
URL http://www.sdrforum.org

Version 1.0 5/6/96

SDRF Vision

The SDRF vision is to provide high quality, ubiquitous, competitively priced wireless networking systems equipment and services with advanced capabilities. This vision includes a view of seamlessness across diverse networks and integration of capabilities in an environment of multiple standards and solutions.

Ease of use, mobility, enhanced productivity, and support for lifestyle choices are all wanted by the communications systems users. Convergence among wireless and wired services such as educational, entertainment, and information services requires improved interworking and interoperability.

Consequently, consumers of communications services, communications service providers, equipment suppliers and maintainers can benefit from open architecture coupled with the software definable networking radio systems developments espoused within SDRF. This community of interest not only includes the needs of the general public, but also includes governments, and their requirements for defense, law enforcement, and emergency services, including National Security and Emergency Preparedness.

SDRF Definition

The Software Defined Radio Forum (SDRF) presents an open architecture for wireless networking systems. Major considerations in networking systems include software defined radio waveform hardware and software, security, source coding, and networking protocols.

Software defined radios use adaptable software and flexible hardware platforms to address the problems that arise from the constant evolution and technical innovation in the wireless industry particularly as waveforms, modulation techniques, protocols, services, and standards change.

A software defined radio in the SDRF context goes beyond the bounds of traditional radio and extends from the radio terminal of the subscriber or user, through and beyond the network infrastructures and supporting sub-systems and systems. SDRF is a concept that spans numerous radio network

technologies and services, such as cellular, PCS, mobile data, emergency services, messaging, paging, and military and government communications.

SDR Forum Mission

The mission of the Open Architecture Software Defined Radio Forum (SDRF) is to accelerate development, deployment and use of software definable radio systems consistent with the objectives of the above wireless vision.

The SDR Forum will work toward the adoption of an open architecture for advanced wireless systems that includes the requisite functionality in terminals, networks, and systems to provide multiple capability and multiple mission flexibility for voice, data, messaging, image, multimedia, and future needs.

The SDR Forum shall establish requirements related to the definition of internal and external system interfaces, modules, software, and functionality that the industry can use as guidelines in building modules, products, and systems.

Further, the SDR Forum will promote the development of standards for SDR, including those focused on SDR equipment and those in supporting service application areas, in areas of interoperability and performance, and in underpinning core technologies, either directly or through appropriate liaison to other industry associations and standards bodies. The SDR Forum will pursue industry wide acceptance of these standards.

To assist the wireless and supporting industries in understanding the value and benefit of software definable radio and in particular the SDRF vision, the SDR Forum will also address market requirements, quantify the market, and develop timelines relative to the use of multi-mode, multi-band, and multi-application wireless communications systems.

The SDR Forum membership shall include telecommunications users, equipment suppliers, and developers of technology, products, systems, hardware, and software as well as service providers and system operators or any other individual, organization, or entity who has interest in furthering the objective of SDRF.

Appendix B. List of Chairs and Co-chairs

TECHNICAL COMMITTEE (as of November, 1999)

OFFICE	NAME	PHONE		E-MAIL
Technical Comm. Chair	Vacant			
Technical Comm. Co- Chair	Szelc, Dawn	+1-703-883-7770	The Mitre Corp	dszelc@mitre.org
Technical Comm. Co-Chair	Dr. Kohno, Ryuji	+81-3-5448-4380	Advanced Telecom. Lab., Sony Computer Science Laboratories, Inc.	kohno@csl.sony.co.jp
Handheld WG Chair	Cummings, Mark	+1-408-777-4802	enVia	markcummings@envia.com
Mobile WG Co-Chair	Cook, Pete	+1-602-441-1300	Motorola SSTG	p25359@email.mot.com
Mobile WG Co-Chair	Williams, Larry	+1-219-487-6154	ITT A/CD	ljwillia@itt.com
Mobile WG Co-Chair	Fuchs, Alden	+1-978-256-0052 x1788	Mercury Computer Systems, Inc.	afuchs@mc.com
Basestation / Smart Antenna WG Chair	Murotake, Dave	+1-978-256-0052 x1129	Mercury Computer Systems, Inc.	dmurotak@mc.com
Basestation / Smart Antenna WG Co-Chair	Meyer, Ron	+1-972 344 2367	Raytheon	r-meyer1@collins.rockwell.com
Download Task Group Co-Chair	Ralston, John	+1-408- 369-7227 x3108	Morphics Technology, Inc.	jralston@morphics.com

SDR FORUM OFFICERS and Steering Committee (as of November, 1999)

OFFICE	NAME	PHONE		E-MAIL
Forum Chair	Blust, Stephen	+1-404-249-5058	BellSouth Cellular.	blust.stephen@bwi.bls.com
Vice Chair	Ralston, John	+1-408-369-7227 x 3108	Morphics Technology	jralston@morphics.com
Treasurer	Fette, Bruce	+1-480-441-8392	Motorola SSTG	P11693@email.mot.com
Secretary, Operations Chair	Margulies, Allan	+1-315-336-4966	The MITRE Corp	asm@mitre.org
Steering Committee Chair	Williams, Larry	+1-219-487-6154	ITT Aerospace	ljwillia@itt.com
Technical Committee Chair	(Vacant)			
Markets Committee Chair	Watson, John	+1-408-573-6353	QuickSilver Technology	johnw@qstech.com
Large Company Rep.	Hacker, Henry	+1-219-429-6629	Raytheon	hjhack@ftw.rsc.raytheon.com
Medium Company Rep.	Adams, Mark	+1-703-329-9707	Exigent Int'l	mark_adams@alx.sticomet.com
Small Company Rep.	Cummings, Mark	+1-408-777-4802	enVia	markcummings@envia.com
Non-profit Representative	Szelc, Dawn	+1-703-883-7770	The MITRE Corp	dszelc@mitre.org
At Large Member	Cook, Peter	+1-480-733-8225	Motorola	pgcook@uswest.net
At Large Member	Uhrig, Nalini	+1-973-386-7071	Lucent Technologies	nuhrig@lucent.com

Appendix C. Other Organizations Contacted by SDR Forum

The following is a partial list of organizations, including standards bodies, that have been contacted by the SDR Forum for the purpose of providing an introduction to the objectives of the MMTS program and to solicit cooperative participation as appropriate.

AeroSense '97

CTIA

European Commission DG XIII-B Software Radio Workshop

European Commission DG XIII-B ACTS Concertation Meeting

Federal Telecommunications Standards Committee

GloMo

GSM-MOU Association Third Generation Interest Group

IEEE - Microcomputer Workshop

IEEE Wearable Computer Conference

Interdepartmental Radio Advisory Committee

Multiband, Multimode Terminals Workshop

National Association of Broadcasters

PCIA

T1P1/TR46

Telecommunications Industry Association Mobile Communications Systems Division

WAP

MexE

IEICE SDR Study Group

FCC (USA)

MPT (Japan)

Reg TP (Germany)

ITU

ARIB

Appendix D. Bus/Interconnect and Form Factor Technologies

The following section will briefly discuss the technologies that have been considered for implementation in software radio architectures. There are two terms that we need to define, Backplane and Backbone. Backplane is an embedded data transfer bus where modules plug into the backplane and transfer data between the modules. The system backbone is a larger data transfer pipe and is usually connected to another blackbox or chassis within a weapons platform or in a wide area network.

The next section will discuss the backplanes that industry is using.

D.1 EISA

Enhanced Industry Standard Architecture (EISA) was generated in 1988 by nine companies, AST Research, Compaq Computer Corp., Epson, Hewlett-Packard, NEC, Olivetti, Tandy, Wyse, and Zenith Data Systems. The group was frustrated with the extent of IBM's market share of the PC industry and its attempt to capture an even greater market with its new Microchannel architecture. Compaq originally headed up the effort, however, the standard now resides with BCPR services, which officiates the EISA standard.

EISA was designed as an enhancement to the very popular AT bus standard. EISA is a 32 bit backplane bus architecture that would be the successor to the ISA 16 bit standard. EISA features included new advanced data transfer modes that would trim the number of required clock cycles needed to move each byte of data. The theoretical maximum that the 32 bit bus can reach with a defined 8.2 MHz clock is 33MB/sec. This was a 50 percent increase in the proposed Microchannel backplane bus throughput. EISA, Microchannel, and specific processor local bus structures make up the very large and dynamic PC industry.

D.2 PCI Local Bus

The Peripheral Component Interconnect (PCI) local bus could be called the bus of tomorrow. This was designed by Intel and adopted by a consortium which includes Compaq, IBM, Intel, Apple, Digital, and Motorola to name a few. Most of today's high speed processors have PCI local bus interfaces built into the silicon interface. There are a series of I/O controllers, memory controllers and memory devices that have this PCI local bus built into their silicon. PCI holds the best chance of replacing the ancient Industry Standard Architecture (ISA) bus. PCI local bus is a 32 bit, or 64 bit, bus with multiplexed address and data lines. It is intended for use as an interconnect mechanism between highly integrated peripheral controller components (i.e., Ethernet, Serial, SCSI), processor memory systems, and peripheral add-in boards. The motivation for developing this consortium standard is the new graphics-oriented operating systems such as Windows and OS/2. These new graphic user interfaces

and object oriented operating systems are creating bottlenecks between the processor and its display peripherals in the standard PC architecture.

The new features and benefits of the PCI local bus include higher performance over the processor local bus (132MBps), and lower cost, because it is designed for direct silicon interconnection, requiring no glue logic or electrical drivers. The ease of use enables full auto configuration support of the PCI local bus add-in boards and components. The PCI devices contain registers with device information required for dynamic reconfiguration. Another benefit is longevity, because it is processor independent and can migrate to 64-bit architectures, and it uses the new 3.3V and old +5V signaling voltages providing a smooth transition to the new industry standard.

However, PCI as a backplane will have its limitations. Currently the PCI local bus is able to reach its high performance because it dictates what the loading on the bus can be. This means that for a backplane running at 33MHz we can have a load factor of 10 but for the same bus running at 66MHz we can only support a load factor of four. The average 64-bit processor available today has a load factor of two. There is work being done in standards bodies and trade associations that will soon provide solutions to this loading problem for a backplane solution. However, if the PCI local bus was used on the modules or boards plugging into the backplane of embedded system architecture, this would provide an additional factor of modularity and upgradability. We could then see the use of the IEEE 1396.1 PCI local bus Mezzanine Card standard. This standard can be used on Futurebus+, Multibus II, and VMEbus modules that have a PCI local bus installed on their boards.

The other problem that PCI has is that it was originally designed as a motherboard solution. Multiprocessors and multiprocessing are not clearly defined by the standard or by any implementation of the standard currently available off the shelf.

The variations of the PCI local bus are given below.

PCI

Desktop environment and is built on current EISA 12 inch by 4 inch boards (32 and 64 bit options 33MHz and 66MHz).

Compact PCI

Compact PCI was initially designed by seven companies; AMP, DEC, I-Bus, Gespac, Hybricon, Pro-Log, and Ziatech to improve PCI for industrial applications. Compact PCI combines the 3U x 160 mm and 6U x 160 mm board sizes of VMEbus with a 2 mm pin-and-socket connector and a passive backplane. The high density 2 mm connector provides good signal integrity and minimizes noise. The standard was approved by the PCI Industrial Manufacturers Group (PICMG) in November 1995, and there are now about 100 products on the market.

PC/104

The PC/104 architecture developed by Ampro became an open standard, IEEE-P996.1, in February 1992 and the PC/104 Consortium became its custodian. PC/104 is essentially a stackable PC architecture using the 5MBps Industry Standard Architecture (ISA) bus. The card dimension is 3.775" x 3.55". Its small size makes it useful for embedded applications, and it is compact and rugged.

Enhancement

PC/104-Plus is fully compliant with the PC/104 form factor, but uses the PCI bus taking the backplane throughput from 5 MBps to 133 MBps. ISA and PCI modules can stack together. Currently it is a preliminary specification in the PC/104 Consortium. There are numerous products available for both PC/104 and PC/104-Plus.

High Reliability Enhanced PCI Bus

This bus will be developed by IEEE Project 1996 for transportation, telecommunications, and process control industries to provide high availability, fault tolerant systems that support harsh environments and extended temperature ranges. The module sizes include 6SU (4.53"), 12SU (10.43"), 18SU (16.34"), and 24SU (22.24") in height, and 225mm depth preferred for 6SU and 300mm depth preferred for 12SU as well as 175mm and 250mm. It is a 32 bit and 64 bit bus that accommodates Hot Swap and is self configurable.

CardBus

CardBus, also known as PC Card 32, started out as the PCMCIA card (see section F.3) which became popular for adding peripheral I/O and memory to mobile computers. The PCMCIA card is known as PC Card 16 (or R2, for revision 2), and PC Card 32 is the enhanced version. CardBus adds PCI bus performance to the PC Card 16. The PC Card Standard 95 replaces versions 2.0 and 2.1 of the earlier PCMCIA standard and covers both PC Card 16 and PC Card 32. The standard specifies a 68-pin interface between cards and socket host. There are three different form factors, Type I, Type II, and Type III. The only difference between them is the thickness of the cards with Type I being the thinnest. They all plug into the same socket. Three different pin assignments use the same 68-pin interface: PC Card 16, PC Card 32, and zoomed video (ZV). ZV is a recent addition which allows mobile PCs to deliver full-screen, broadcast quality video directly from a PC Card to a system's VGA controller without using the PCI bus. The PC Card Standard also specifies PCMCIA software interfaces. The software architecture specifies Socket Services (SS) and Card Services (CS) modules.

Small PCI

Small PCI was developed as a small-form-factor implementation of the PCI bus. It implements the same performance and electrical characteristics as standard PCI but only the 32 bit option. Small PCI has the same physical form factor as PC Card and CardBus and connects parallel to the system board via a 108-pin header mounted on the system board. See the chart below for a comparison of the Small PCI and the PC Card bus.

Table D-1 Small PCI and Card bus Comparison

Description	Small PCI	Card bus
Form Factor	PCMCIA type II and III	PCMCIA type I, II, III plus RF extensions
Socket Keying	3.3v, 5v Universal and will reject JEIDA, DRAM and PCMCIA cards	Rejects all PCI & Japanese Electronic Industry Association (JEIDA) DRAM cards
Applications (Market Place)	low profile desk top	SAME
Environment	Under system PnP compatible	External PnP for Dynamic insertion & removal
Market	Built to order PCI only	Dynamic expansion & reconfiguration - bus independent
Cost	Low, direct attached to PCI	Higher cost - PCMCIA/CardBus controller required
Card Cover	Not required	Encapsulated
Connector	108 pin	68 pin
Connector Reliability	100 cycle (tested higher)	10,000 cycles
Performance	33MHz	33 MHz
Topology	Bus oriented	Point to Point
Voltage	3.3V, 5V	3.3V
Power	2.5, 5, 10 Watts	

Table D-2 Comparison of Example PCI Options

	Desktop PCI	Passive Backplane PCI	PMC	Compact PCI	Card Bus	Small-PCI	PC/104-Plus
Dimensions (in.)	Long: 12.3 x 3.9 Short: 6.9 x 3.9	12.3 x 3.9	5.9 x 2.9	6.3 x 3.9	3.4 x 2.1	3.4 x 2.1	3.8 x 3.6
Area (sq in)	Long: 48 Short: 24	48	17	25	7	7	13
Bus Connector	Edge-Card	Edge-Card	Pin & Socket	Pin & Socket	Pin & Socket	Pin & Socket	Pin & Socket
Includes ISA Bus	No	Yes	No	No	No	No	Yes
Installation Plane	Perpendicular	Perpendicular	Parallel	Perpendicular	Parallel	Parallel	Parallel
Expands Without Additional Slots (Self Stacking)	No	No	No	No	No	No	Yes
Positive Retention	No	No	Yes	Yes	No	No	Yes
Standards Body	PCI-SIG	PICMG	IEEE	PICMG	PCMCIA	PCI-SIG	PC/104
Primary Application Area	Desktop: motherboard expansion	Industrial: backplane expansion	Industrial: VME mezzanine	Industrial: backplane expansion	Laptop: end user additions	Laptop: factory options	Embedded: SBC expansion

D.3 PERSONAL COMPUTER MEMORY CARD INDUSTRY ASSOCIATION (PCMCIA)

PCMCIA was driven by the new PC market demands for smaller personal computers. These markets are laptops, Notebooks, and Palmtop computers. Since the late 1980s there have been many attempts to reduce the size and power requirements of Random Access Memory (RAM). In 1987 Mitsubishi had a popular memory card the size of a credit card. However, it used a proprietary 60 pin package. Fujitsu also had a memory card with a proprietary 68 pin package. POQUET, a new company investing in memory cards as an alternative to disk drives, was driving the market for a standard. In June 1988 the PCMCIA was established and work began on a standard memory card. In September 1990 the first release was accepted and products were available to users. In September 1991 release two was accepted as a standard and backwardly compatible to release one. This standard identifies both mechanical and electrical interfaces. It deals with file formats, data structures, and methods through which the card can convey its configuration and capabilities to the host.

PCMCIA type II or release two provide the user with 5.0 mm spacing for the case, a printed circuit card and all the components. This is the price we pay for the convenience of a credit card size memory board. However, PCMCIA has not been limited to memory board applications. PCMCIA is becoming the de facto expansion standard for mobile communications. However, component height, component footprint, component power, and data conversion are still issues that have to be solved when designing a PCMCIA card.

D.4 The VMEBUS Backplane

History

In 1979 Motorola defined a micro-computer bus called the VERSAbus. This bus was designed to build multiprocessing systems using the new Motorola 68000 32 bit processor. In 1981, Motorola, working with Mostek and Signetics modified the VERSAbus specification and called it VERSA Module Eurocard (VME), named from the location of the labs where the specification was modified. These companies released the specification to the world calling it the VMEbus. In 1987, the Institute of Electrical and Electronics Engineers (IEEE) published ANSI (American National Standards Institute) and IEEE 1014-1987, the standard that defined both the electrical and mechanical characteristics of the VMEbus backplane and module.

The VMEbus is defined by ANSI/IEEE 1014-1987 which defines both the electrical and mechanical characteristics of backplane and module. There are two acceptable board sizes 3U x 160mm (3.9 inches x 6.2 inches) with one connector and 6U x 160mm (9.2 inches x 6.2 inches) with two connectors (P1, P2). The standard defines a convection or forced air cooling method of the modules. The theoretical transfer rate of a module over the backplane bus is 40 Megabytes (MBps) per second at 32-bit wide transfers. It is a multiprocessor, asynchronous parallel bus architecture.

There are a number of military and commercial users who have developed their products based on the VMEbus. They range from military avionics and industrial control systems, to medical imaging systems. The spectrum of VMEbus products available today covers a wide variety of environments. Today's military programs will require equipment to meet programmatic concerns as well as performance requirements. The programmatic concerns deal with cost, performance, development time, proof of concept, and development of the application software. There are five equipment styles into which today's VMEbus products can be categorized. These styles are applicable to 90 percent of military application platforms and include commercial, ruggedized air cooled, ruggedized conduction cooled, military air cooled, and military conduction cooled. These equipment styles cover environmental and programmatic requirements from commercial systems to mission-critical applications.

The VMEbus migration with technology

Since 1980 the VMEbus has found its way into many commercial, industrial, and military applications. The VMEbus has become many corporations' internal research and development backplane bus which evolved to actual commercial products based on the VMEbus. The VMEbus is in a wide variety of commercial applications, from billing and controlling systems for the telecommunication industry to medical imaging and manufacturing floor applications. VMEbus technology follows the PC market advances in silicon and Input/Output (I/O) application techniques.

The VMEbus has migrated from single boards acting as processor, memory, and I/O control to single boards containing all three elements. Recently the backplane industry has seen a new trend in multiprocessor technology. Specialty processors like the Intel I860, the Texas Instruments TSM320C040 digital signal processor, and Transputers are showing up 2, 4, 8 processors to a VMEbus board. These processors in combination with the new 32 bit and 64 bit Complex Instruction Set Computer (CISC)- based processors from Intel, Motorola, Hewlett Packard, and Digital provide an unique modular capability for embedded applications.

To meet the data throughput demands of these new processors and I/O controllers, the VMEbus has added new high speed data bus structures to its current VME Subsystem Bus (VSB) P2 alternate data bus. Raceway, SkyChannel, and Signal Computing System Architecture (SCSA) all provide unique solutions to moving data from processor to processor to memory or I/O controller.

With new technologies emerging like Personal Computer Memory Card Industry Association (PCMCIA), Peripheral Component Interconnect (PCI) local bus, and Scaleable Coherent Interconnect (SCI) the VMEbus is providing solutions for these technologies. There are PCMCIA type three memory expansion boards, PCI local bus processors and even mezzanine bus cards, as well as Scaleable Coherent Interconnect I/O controllers.

Despite its age the VMEbus has been able to adapt to the changing commercial marketplace expanding the role it plays in development, research, and production. It is also expanding its market industry by addressing Fault Tolerance, Live Insertion, High Availability, and Testability. Below are sections that discuss some of the enhancements to the VMEbus since 1980.

D.5 VME64

VME64 is an ANSI standard, ANSI/VITA-1 1995, and increases the VMEbus throughput to 80 Mbps from the 32-bit version of 40. VME64 is 100 percent backward compatible to ANSI/IEEE 1014 (32-bit VMEbus). VME64 provides 64/32/24/16/08 data it transfers with 16/32/64 bit addressing. VMEbus 6U x 160mm boards will be capable of 80 Mbytes/sec. 3U x 160mm boards will provide their users with 40 Mbytes/sec. VME64 adds new capabilities such as:

- Rescinding DTACK
- Lock commands
- Retry signal
- Auto slot ID
- Auto system controller
- Control and Status registers

Rescinding DTACK provides users and integrators with quicker and cleaner data transfers. LOCK commands aid the integrator and user in sectioning off specific system resources. The RETRY signal provides a mechanism to retry a data or address transfer if an error or read modify write cycle is in process. Auto Slot ID provides the system integrator with board identification and location in the VMEbus system configuration. Along with the Auto system controller the VMEbus system can now, on power up, identify who the slot one controller is and what other resources are in the system configuration. The Control and Status registers will provide faster access to faults and health monitoring capabilities.

Products Availability

Currently there are four manufacturers of VMEbus Silicon that have VME64 compliant silicon. Newbridge Technology, Force Computers, Motorola, and Cypress Semiconductor. The majority of products that are available with VME64 enhancements are processor boards. I/O and memory boards are still in development with the VME64 enhancements. To find what company is supporting VME64, check the VITA World Wide Web site for their on-line product catalog (http://www.vita.com).

Integration Techniques

The VMEbus boards that do support VME64 enhancements may have little or no effect on the throughput of a system when the new Multiplexed Block Transfer (MBLT) of VME64 is integrated. The reason for this is simply that many I/O and memory boards are still transferring 32 bit data instead of the MBLT 64 bit data transfers. Also some of the processors that offer MBLT 64-bit transfers still have 32-bit local bus structures requiring two local bus accesses for every one VMEbus MBLT 64 bit data transfer. Therefore if you have a VME64 processor who will transfer 32 bit data over the VMEbus at a rate of 10 MBytes a second (BLT) you may find that same processor transferring 64 bit data over the VMEbus at a slower rate of 7 MBps. This is because to the processor's local bus is 32 bits wide and the processor has to move the data into or out of the VMEbus interface chip's internal

File In, File Out (FIFO) or register. Look to the 64-bit processors and their 64-bit local bus structures for maximizing the MBLT feature of the VMEbus. PCI local bus is still currently only 32-bits wide. However it can move data at a very high rate (130MBps). Thirty-two-bit processors like the Power PC may not have the MBLT slow down with this combination local bus and VMEbus interface.

D.6 VME64 Extensions

The VME64 Extensions are a group of optional capabilities or enhancements that will help the integrator tailor a VMEbus-configured system to meet specific needs of his or her application. VME64 Extensions have sought solutions to providing the VMEbus user with additional I/O through the P1 and P2 connectors. They have looked at ways to improve signal integrity and provide higher availability of the configured system through fault management and testability techniques.

Enhancements

The VME64 Extensions provide VMEbus users and system integrators with the hardware and software tools to meet the new generation of applications. VME64 Extensions provide new voltage pins like 3.3V and 48V with additional ground signals to improve signal integrity. VME64 Extensions provides the user with a new high speed serial bus designed to be an alternate data path for the VMEbus. There is a new test and maintenance bus (IEEE 1149.5) defined for use with the VME64 Extensions. Live Insertion pins have been defined to allow hot swap of boards. There are also additional I/O pins on the new connectors and a space defined for an optional P0 connector. The VME64 Extension also defines a new EMI protection front panel, that will reduce the emissions of VMEbus processor boards so that they can meet UL and CE (European) standards.

Products Availability

Currently this standard is in a working committee and has not been released for ANSI accreditation. Therefore, there are no products available with all of the features described in the paragraphs above. However, there are boards that have been produced by major VMEbus board manufactures and Backplane vendors have tested the new connector and backplane solutions.

Integration Techniques

When these products do become available in the VMEbus market, care should be given to configuring systems with VME64 Extension products. Because these enhancements are options within the document not all companies will produce all options. This could cause a problem with interoperability.

D.7 VME320

Enhancements

At a recent press conference, Bustronic Corporation and Arizona Digital announced their development of a new enhancement to the VMEbus, VME320. This is a new type of VME backplane that transfers data at 320 Mbytes/sec. This is 4 times faster than the existing VME64 and unlike the VME64 Extensions, will be completely backwardly compatible with all existing VMEbus boards. It uses the existing 96-pin DIN connectors and is also available in a 160 MBps version, VME160.

Products Availability

Currently this standard is in a working committee and has not been released for ANSI accreditation.

Integration Techniques

Because this new technique provides cleaner transitions than traditional VMEbus backplanes, the integration of products into new or existing systems should provide no change in performance or an improvement in electrical performance as well as data transfer rate.

D.8 SEM-E on VME

Enhancements

SEM-E has been a popular form factor used in the military for a number of years. It is based on conduction cooled technology, a small form factor, and a blade-and-fork style connector that provides high reliability in high shock and vibration environments. This technology under standardization uses the SEM-E form factor with the VMEbus backplane and protocol.

Products Availability
Currently this standard is in a working committee and has not been released for ANSI accreditation.

Integration Techniques

Once products become available using this form factor/backplane combination the utility of SEM-E will be greatly enhanced by the inclusion of a standardized protocol.

D.9 VMEbus P2 Sub-bus Data Transfer Architectures

This section will discuss a sub-bus or sub-backplane data transfer bus. These buses have been added to meet the rising need to move larger amounts of data in greater speeds.

D.9.1 RACEway

The ANSI Board of Standards Review officially approved the VITA 5-1994, RACEway Interlink as a standard on July 31, 1995. RACEway is a crossbar-switch technology integrated onto a P2 backplane for the VMEbus. RACEway provides the VMEbus user with up to six bi-directional data paths with a maximum throughput of 160 MBps.

Enhancements

The development of RACEway Interlink was motivated by the desire to provide high bandwidth communication while maintaining compatibility with the VMEbus. RACEway Interlink is a scaleable interconnection fabric based on a network of crossbar-switch devices that provide 1.6 gigabytes per second (Gbps) throughput. The RACEway Interlink specification was brought to the VSO for standardization by Mercury Computer Systems, Inc..

The RACEway Interlink standard is targeted at multiprocessor and high-speed I/O applications. I/O devices such as A/D, D/A, frame stores, graphic displays, LAN/WAN connections, and storage devices. RACEway Interlink enhances VMEbus performance in existing VMEbus systems by enabling concurrent communication between multiple processors.

Product Availability

Numerous RACEway Interlink design projects are underway in the VMEbus community with developers using the new standard to link I/O subsystems with off-the-shelf VME boards. Also underway is the design of chip-level products to support this standard. Mercury and Cypress Semiconductor Corporation will be working together to develop semiconductors that will help proliferate RACEway technology.

D.9.2 VSB

The VME Subsystem Bus (VSB) is the son of VMX and VMX32 memory expansion buses that were developed by Motorola to move data to and from the processor board's local bus. VMX and VMX32 were local bus expansions in the days when only 256K and 512K DRAM or SRAM fit onto a single board computer. The VME community saw that a separate data transfer bus would increase the system throughput and expanded the VMX concept to include arbitration and interrupts.

Enhancements

The VSB uses all the user defined pins of the VMEbus P2 connector. It defines a complete asynchronous backplane data transfer bus. The VSB is a full 32-bit address, data bus with a single level arbiter and single level interrupt. The VSB supports dynamic bus sizing, read-modify write cycles, and block transfers as does the VMEbus. The throughput of the VSB is currently 30-40 MBps. There is work going on at VITA/VSO that is exploring how to increase the speed and data size of the current VSB standard.

Products

The VSB has been available off-the-shelf since 1987. There are many processors, I/O controllers, and memory cards that support this P2 bus. There are even processors that are available in five environmental build standards that support this 32 bit subsystem bus.

D.9.3 Skychannel

SKYchannel is a 200 MBps packet bus architecture developed by SKY computers. This architecture provides the VMEbus user a higher throughput and alternate data transfer path from the standard VMEbus backplane architecture. SKYchannel connects multiple single board computers on the VMEbus through an active P2 backplane connection. This concept is similar to the RACEway crossbar P2 active backplane solution.

Enhancements

The SKYchannel packet bus architecture provides yet another P2 backplane communications path. This new packet mode architecture uses a series of FIFOs, DMA controllers and packet controllers at each SKYchannel interface to ensure that packets are built continuously and data pipelines are filled. SKYchannel can be a backplane or a point to point connection. The current proposed specification allows the user up to ten ports and five simultaneous 320Mbps per level for an aggregated data bandwidth of 1600Mbps (200MBps).

Products Availability

SKYchannel is still in working group draft. Products are available from SKY, but they do not meet the current configuration of the draft standard.

D.9.4 SCSA

The ANSI Board of Standards Review officially approved the VITA 6-1994, SCSA as a standard on July 24, 1995.

Enhancements

SCSA is a comprehensive, open software and hardware architecture that streamlines the process of building computer telephony systems.

SCSA offers a standard way of dealing with the many levels and inter relationships that come into play when developing and building computer telephony systems. It provides standard interfaces that satisfy the demands of application program developers, hardware component developers, software algorithm developers, platform developers, and end users.

This new ANSI/VITA 6-1994 SCSA standard allows the VMEbus architecture to move firmly into telephony. SCSA enables VMEbus manufacturers to develop products that bring SCSA's high signal capacity and VME performance and robust packaging to a growing, global CTI market.

Product Availability

Currently there are no board level products available with VMEbus and SCSA. However, there are many turnkey systems that integrate VMEbus and SCSA in digital switching systems.

D.10 Backbone or System Bus Structures

In this section we will be discussing current system backbone structures and how they are implemented or going to be implemented in the software radio and to the software radio system architectures.

D.10.1 MIL-STD-1553B

During the late 1960's the military aircraft industry reached a level of performance complexity such that simple point-to-point interconnection was no longer cost effective. Nor did it meet the requirements of the applications. The need to share information and resources grew as size, weight, and space requirements became critical in smaller faster aircraft. In 1968 the SAE A2K Committee in cooperation with the TRI-Services was formed to generate military standard for multiplexing. The Military Standard Aircraft Internal Time Division Command, Response Multiplexed Data Bus was issued in 1973. The current MIL-STD-1553B standard was adopted in 1978. This serial bus had a transfer rate of 1 Mbps in 20-bit frames. However, with the processor overhead and response required throughputs of 7.125 Kbps is what is seen on today's MIL-STD-1553B backbone bus standards. The data bus travels a shielded twisted pair cable. The data bus transmissions are serial time division multiplex messages in pulse code modulation form. The data bus traffic is half duplex, hence it travels in one direction at a time over one of the dual redundant cables. The MIL-STD-1553B data bus network functions in a command, response sequence. Access to the data bus is provided only when a command is received and acknowledged by the data bus controller. The MIL-STD-1553B data bus is found in many commercial and military Avionics system architectures as well as in the military Vetronics industry. We have seen a great push to have the black box chassis connect to this standard serial bus to provide a cost effective mechanism to pass data within our system application.

D.10.2 MIL-STD-1773A

The MIL-STD-1773A is the optical equivalent of the MIL-STD-1553 bus. The MIL-STD-1773 was developed to realize the optical technology advantages for the Avionics community. Optics provide solutions with higher bandwidth, lighter weight, and reduced space. Optics also provides immunity to electromagnetic interference. MIL-STD-1773 is implemented to replace MIL-STD-1553 and therefore the standard is implemented at the same speed of 1 Mbps. Since MIL-STD-1773 deals with optical rather than electrical signals there were changes required in the Manchester II coding. A logical one begins with optical energy present and a logical zero begins with no optical energy present.

D.10.3 FDDI

Fiber Distributed Data Interface (FDDI) is a 100 Mbps Local Area Network (LAN) standard. The topology of the FDDI is a point to point connection of links connected in a logical ring. There are two such rings that circle each other connecting to the same links in the ring. To govern who will be in control the FDDI bus a token is passed between all the nodes on the FDDI bus. This topology lends itself to fault tolerance. In a dual attached LAN a break in the ring will automatically switch the data to the other ring. There can be up to 500 links or stations with the maximum size of the rings or LAN as 200 kilometers. The maximum distance between each link or station can be up to 2 kilometers squared. The Open System Interconnect reference model identifies FDDI as a physical layer or layer one and part of the data link layer two. FDDI and Ethernet (IEEE 802.3) are considered very similar. However, FDDI is more related to the (IEEE 802.5) token ring. The first FDDI standard was available from ANSI X3T9 committee in early 1983. It officially was completed in the year 1988 as the ANSI standard X3.148-1988

D.10.4 N-ISDN

Narrowband Integrated Services Digital Network (N-ISDN) provides end-to-end digital connectivity with access to voice and data services over the same digital transmission and switching facilities. N-ISDN provides two core interfaces Basic Rate Interface (BRI) and Primary Rate Interface (PRI). N-ISDN also provides the user with three channel types B, D, and H. BRI is a set of two bearer channels (B) which carry data or voice. Each B channel is a 64 K bits per second (bps) pipe. This pipe or channel can carry any type of digitized voice, data, and video information. BRI also carries one D channel for signaling and switching data. The D channel is a 16 Kbps digital channel that carries the information for the network switches to set up, connect, monitor, and tear down connections or calls. PRI is a larger set of channels that includes 23 B channels at 64 Kbps and one D channel. The D channel in the PRI mode is different than BRI. The difference is the amount of data required in the D channel for the switching information. This channel is 64 Kbps. In Europe, primarily, the H channel is used to carry user information relating to video teleconferencing, high speed data, and high audio or sound programs and images. The H channel has variable throughput capabilities at 384 Kbps, 1.536 Mbps, or 1.920 Mbps. These throughput capabilities allow video teleconferencing, high-quality audio

and images to be passed between users. It is the signaling channel that allows N-ISDN to carry multiple integrated digital services over a switched circuit.

D.10.5 ATM

Asynchronous Transfer Mode (ATM) is the formal International Telecommunication Union (ITU) standard for cell based voice, data, and multimedia communication in a public network. ATM is a high bandwidth, low delay switching and multiplexing technology that uses a 53 byte cell (one byte is eight bits) for transmitting information. Each cell consists of an information field that is transported transparently by the network (Similar to the signal channel D of N-ISDN) and a header containing routing information. The obvious benefits of ATM include high speed, low latency, and increased network availability through automatic and guaranteed assignment of network bandwidth. This is ideal for time sensitive data like voice and video. For the military it could carry synchronous channels with cryptographic data. Current ATM data rates are 45Mbps, 51Mbps, 100Mbps, 155Mbps, and 622Mbps. ATM is a connection oriented process, although it is designed for either connectionless and connections-oriented services. ATM supports a number of applications using the ATM Adaptation Layers (AAL). ATM is an evolving standard and there are currently three AALs presently defined with a fourth AAL2 under development.

AAL1: Timing required, constant bit rate, connection oriented. This ATM Adaptation Layer provides the details for connection oriented circuit emulation of a point to point line. T1, E1 or T3, E3 leased circuits could be emulated with the use of this AAL.

AAL2: Designed to support variable bit rate applications such as compressed motion video traffic generated by the MPEG, MPEG2 algorithms.

AAL3/4: Timing not required, variable bit rate, connectionless. This supports fast packet services such as Switched Multi-megabit Data Service (SMDS).

AAL 5: Unrestricted (variable bit rate, connection oriented or connectionless), also known as "Class X." This AAL supports a fast packet service such as cell-relay. This AAL is being implemented in the Local Area Network emulation for ATM

ATM information is sent between two points over a media comprised of virtual channels. Each channel can be transmitted over the network in a manner consistent with the needs of the data or subscriber. The user's data is associated with a specified virtual channel. In ATM the virtual channel is used to describe unidirectional transport of ATM cells associated by a common unique identifier value called a virtual channel identifier. A virtual path is used to describe unidirectional transport of ATM cells belonging to virtual channels that are associated by a common identifier value called virtual path identifier. The ATM switch reads the virtual channel identifier and virtual path information from an incoming cell and based on the information makes a routing decision and sends the cell out through the proper switch port.

D.10.6 FIBREChannel

In 1988 the ANSI standards body, X3T9.3 committee, formed a FibreChannel working group to develop a practical, inexpensive yet expandable method for achieving high speed data transfers among workstations, mainframes, supercomputers, desktop computers, storage devices, and display devices. FibreChannel was started to address the need for fast transfers of large volumes of data while at the same time relieving system manufacturers from the burden of supporting the variety of channels and networks currently in place throughout the Information Technology market.

In 1994 ANSI X3.230-1994 was approved for optimizing large volumes of data, not for low-latency dynamic interactive usage. The throughput of the standard allows for multiple architectures, point-to-point and Local Area Network configurations at 265,531 and 1062 MBps to 2 and possibly 4 GBps.

The current market that the FibreChannel solutions appear to be addressing is in the area of mass storage devices. SCSI-3 is a FibreChannel-like serial connection.

D.10.7 FireWire

FireWire, IEEE Std. 1394-1995, was originally intended as an interconnection method between PC peripherals and consumer electronic devices. The throughput speeds are 100 Mbps and 200 Mbps with 400 Mbps in development. The distance over which it operates is bounded at 10s of meters. Some FireWire products are available, and Microsoft has announced its intention to support it in future versions of Windows. FireWire, IEEE Std. 1394-1995, was originally intended as an interconnection method between PC peripherals and consumer electronic devices. The throughput speeds are 100 Mbps and 200 Mbps with 400 Mbps in development. The distance over which it operates is bounded at 10s of meters. Some FireWire products are available, and Microsoft has announced its intention to support it in future versions of Windows. FireWire, IEEE Std. 1394-1995, was originally intended as an interconnection method between PC peripherals and consumer electronic devices. The throughput speeds are 100 Mbps and 200 Mbps with 400 Mbps in development. The distance over which it operates is bounded at 10s of meters. Some FireWire products are available, and Microsoft has announced its intention to support it in future versions of Windows.

Appendix E. Members of the Software Defined Radio Forum

Advanced Communications Technologies
Agilent Technologies
Algorex
AMP - M/A-COM
BellSouth Cellular
Blue Wave Systems
Boeing
CCL/Industrial Technology Research Institute
CommLargo
COMSAT
Raytheon Systems
Ditrans
Department of National Defence/Defence Research Establishment Ottawa
enVia
Ericsson
ETRI, Korea
Exigent International
General Dynamics
IIT Research Institute
ITT Industries
Kokusai Electric Company
Kyocera DDI
LG Corporate Institute of Technology
LOGIC Devices, Inc.
Lucent Technologies
Mercury Computer
The MITRE Corporation (Center for Air Force C2 Systems)
The MITRE Corporation (Washington C3 Operations)
Mitsubishi Electric
Morphics Technology
Motorola SSTG
NEC America
NTT
Nokia Telecommunications, Inc.
Omron
Personal Telecomm Technologies
Quicksilver Technology
Rockwell Collins, Inc.
Roke Manor Research
Samsung
Samuel Neaman Institute (Technion)
Sangikyo
SK Telecom
Sonera
Sony Computer Science Lab
Southwestern Bell Technology Resources
SPAWAR Systems Center
Titan/Linkabit
Toshiba
Triscend
Tropper Technologies
University of Oulu
US Air Force Research Laboratory
US Army - PMTRCS
USWest
Vanu, Inc.
Visteon
Yokohama National University