7

Software Architecture and Components

In previous chapters we have focused on system-level design, hardware selection, and functional partitioning. These topics currently dominate the early design stages of a software defined radio project. This is particularly true for 3G cellular mobile radio, because state-of-the-art hardware is required to meet demanding performance specifications. Ultimately it is the software that provides the functionality, and the software architecture must include characteristics and mechanisms that allow for an efficient utilization of the underlying hardware platform.

7.1 Introduction

Significant effort is being invested in developing open software architectures and interfaces; this work aims to foster software reuse, portability, and compatibility. Many in the SDR industry are hoping that an open source paradigm will produce many of the benefits similarly experienced by users of the LINUX operating system. This chapter introduces two organizations that are helping to define the open software radio platform and follows up with some of the technologies, including operating systems and software languages, that must be chosen during the project's detailed design phase.

7.2 Major Software Architectural Choices

7.2.1 Hardware-Specific Software Architecture

In Chapter 1 we introduced the concept of the ideal software defined radio (see Figure 1.2) and proposed a layered abstracted software architecture. This concept allows the application software to be independent of an underlying standardized hardware platform. The aim of this approach is that ultimately any investment in the development of application software is maintained when ported to new (and presumably better) hardware platforms that comply with the standard.

Commercially designed cellular mobile radio equipment (terminals and base stations) has traditionally been developed as a black box. A software-level interface is not provided; only high-level functional and physical interfaces are exposed (e.g., mobile phone serial interface or BTS Abis interface [1]). In most cases radio equipment (1G and 2G) from different vendors will have incompatible software architectures, where the driving requirements have been to support legacy hardware or software or both.

The degree to which these traditional developments have utilized common interfaces and object-oriented (OO) design is difficult to gauge, because details of the developments are most often kept in-house.

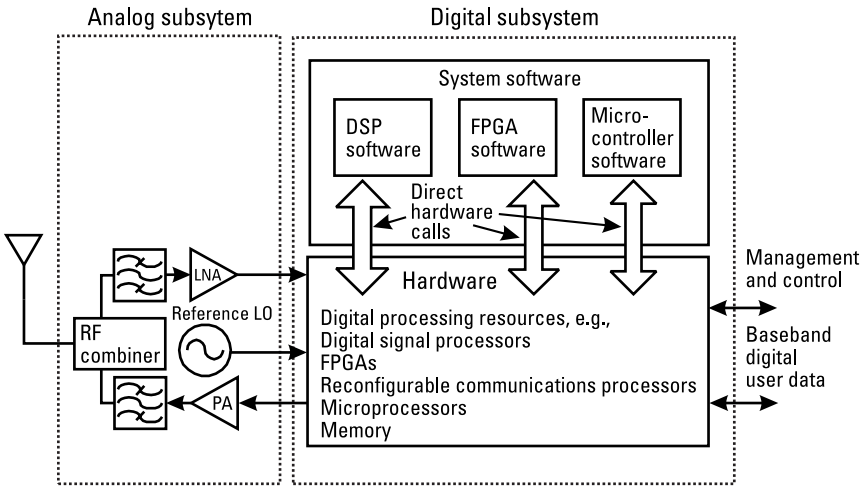

Figure 7.1 Hardware specific software defined radio architecture.

The system architect could choose a hardware-specific software architecture (see Figure 7.1) and still meet the requirement of implementing a software defined radio. For this case the system software is developed in a native language (to the processor), and the software makes direct calls to the hardware resources (e.g., direct manipulation of registers or I/O). This approach is used in conjunction with a structured design method, such as data flow diagrams, to capture the software design. The resultant architecture is not object oriented and certainly not portable.

Therefore, the ideal SDR, as detailed in Chapter 1, should include an emphasis on object orientation (i.e., the radio will be under the control of software and designed to use as much object orientation and software reuse as possible).

7.2.2 Abstracted Open Software Architecture

The term "bloatware" is part of the PC software lexicon as a result of inefficient object-oriented implementations. These poorly designed applications and operating systems are characterized by a tendency, following upgrade, to consume far more hardware resources (RAM, disk space, and CPU cycles) than the previous version for questionable improvements in functionality. As an example, the Microsoft NT operating system reportedly expanded from 16 million lines of code in version 4 to approximately 40 million lines of code in version 5.

The question then becomes: Can a real-time system such as a 3G software radio afford to become as fully object oriented as a PC application? Also, can the abstraction extend from the system's general-purpose microprocessors (e.g., Intel, Motorola 82xx), where the control and signaling software is often hosted to flow across the radios' embedded processors (e.g., DSP, FPGA, RCP, *u*P)? Lynes [2] recognizes this question by stating that traditionally there has been difficulty in providing a similar level of abstraction in the more dedicated resources used to process the physical layer signals." Figure 7.2 illustrates Lynes's [2] approach for a 3G SDR, where a "thin" layer of abstraction is spread across the layer one processing resources.

Figure 7.2 presents both a functional view of the radio's layer one processing and a software architectural view. The shaded areas [e.g., DSP (AP) and FPGA (AP)] make up the application software and the other blocks (e.g., PDC/PUC Lib) constitute the middleware. The level of abstraction provided by this middleware becomes thinner for the highest-rate signal

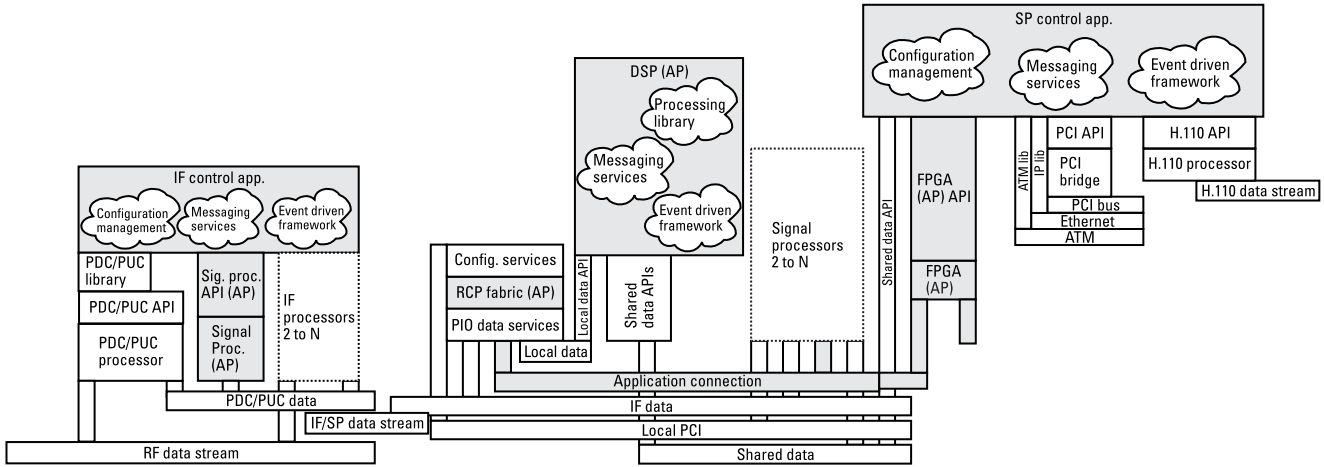

Figure 7.2 Layer one abstraction. (*Source:* ACT Australia, 2001. Reprinted with permission.)

processing elements, since efficiency and channel density are still major objectives. The IF processor library includes standard definitions for all common air interface standards; however, the control application, which is responsible for monitoring gain control, timing offsets, and synchronizing parameter changes, is application dependent and must be developed. The framework (middleware and hardware) provides an event-driven control structure, where the triggering events, much like a sensitivity list, can be modified to suit the application. APIs are provided to access all measurable and controllable parameters.

A similar control framework is provided for the baseband processor. All local and board to board interprocessor communications are handled by middleware. Application code is also needed for the processing element RCP and DSP devices. Both these devices are supported by application libraries and code framework that take care of all the data transfer and code loading functions. The application exists in a generic buffer-in, process, buffer-out environment.

7.3 Software Standards for Software Radio

There are two key organizations pushing forward the adoption of software standards for the development of software defined radios.

The Software Communications Architecture Specification (SCAS) [3] is published by the United States Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This defense program has set goals for future communications systems (i.e., to increase flexibility and interoperability; ease up grade ability; and reduce acquisition, operation, and support costs). The JTRS states that the SCAS is not a system specification but a set of rules that constrain the design of systems to achieve these objectives. The U.S. government expects the basic SCAS to become an approved commercial standard through the Object Management Group (OMG) and has designed the specification to meet commercial as well as military application requirements. The OMG is becoming more involved in software radio specification and has created a special interest group, see Section 7.3.3 for more information.

The other effort is led by the SDR Forum's Mobile Working Group, which has developed a Distributed Object Computing Software Radio Architecture (DOCSRA) [4]. The development describes a software framework for open, distributed, object-oriented, software-programmable radio architectures. The Mobile Working Group developed the DOCSRA using the IEEE's recommended methodology [5] for architectural descriptions.

The SCAS and DOCSRA have a similar heritage with some common terms (e.g., *FileManager* and *DomainManager*); we cover both proposals due to their importance and potential to become industry standards. The presented architectures are object oriented and the designs are captured using Unified Modeling Language (UML) [6] notation. The standardization and open architecture approach is already attracting attention, with early signs of references in commercial product [7].

7.3.1 JTRS Software Communications Architecture Specification

The software framework of the SCAS defines the operating environment (OE) and specifies the services and interfaces that applications use from that environment. The OE is comprised of the following entities:

- A core framework (CF);
- CORBA middleware;
- A POSIX-based operating system (OS) with associated board support packages.

7.3.1.1 Architecture Overview

A graphical depiction of the relationships between the OE (CF and COTS) and SDR noncore components is provided in Figure 7.3; this details the key elements and IDL interfaces for the CF.

The SDR software structure shows that the noncore components are abstracted away from the underlying hardware, and all entities are connected via a logical software bus by using CORBA. Adapters are provided to allow non-CORBA modem, security, and I/O components to interface with the CORBA components via the CORBA bus.

The software architecture is capable of operating using COTS hardware bus architectures (e.g., VME, cPCI, and so on); however, the actual implementation will be determined by the derived performance requirements of the noncore components (e.g., data bandwidth and timing).

COTS operating systems (OSs) with real-time embedded capabilities such as preemptive multitasking are expected to be suitable for SDR. The OS is also assumed to be portable operating system interface (POSIX) compliant and the SCAS recommends the use of POSIX 1003.13 [8]. POSIX was first published by the IEEE in 1990 and now consists of more than 30 standards [9], ranging from basic OS definitions through to those with advanced real-time extensions. These extensions include such features as

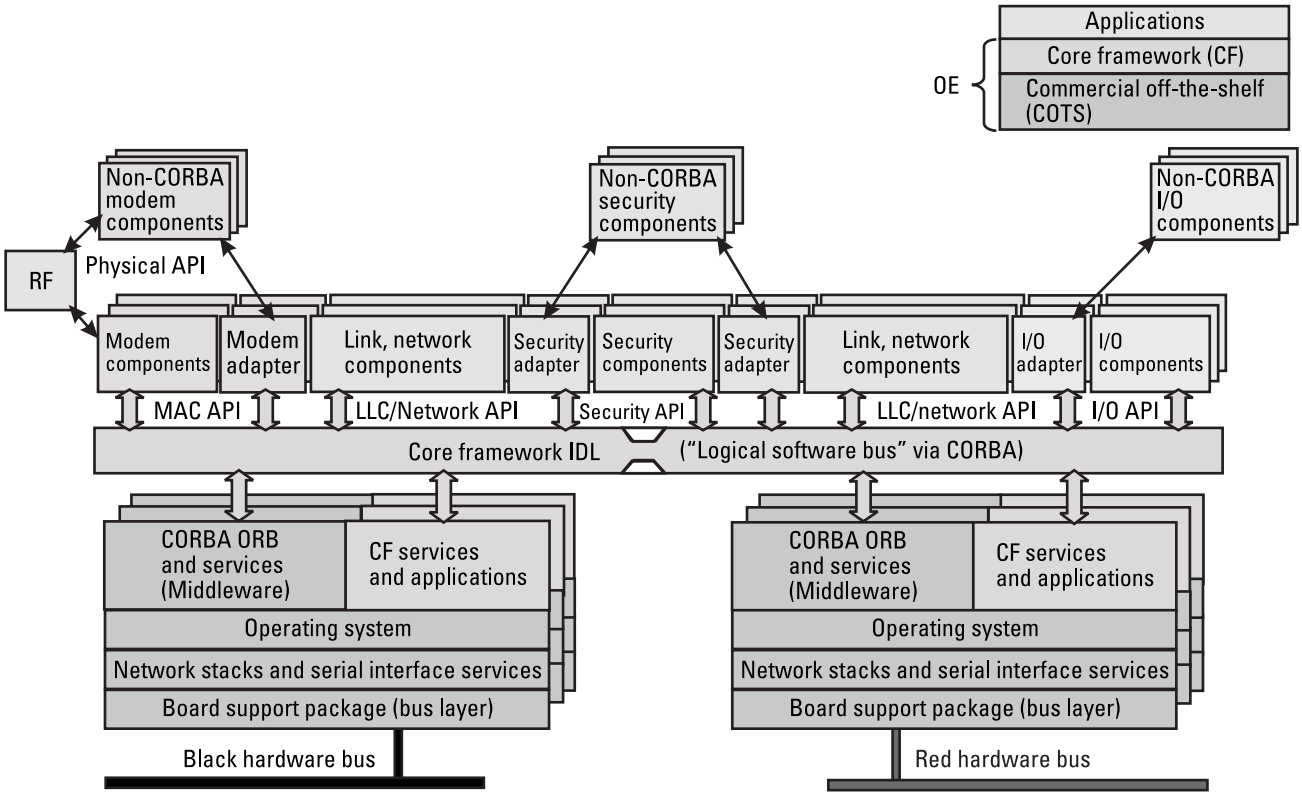

Figure 7.3 Software structure. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

threads, priority scheduling, and semaphores. POSIX 1003.13 does not contain any additional features; instead, it groups the functions of existing POSIX standards into units of functionality.

The operating environment is the integration into an SDR implementation of the CF services and COTS infrastructure (e.g., OSs, bus support packages, and CORBA middleware services). The SCAS emphasises open standards, multiple vendor available commercial (COTS) elements, and higher-order software languages.

The board support package (BSP) dual-connection to a black secure bus and a red nonsecure bus would only be implemented for defense type applications and is not expected to be necessary for civilian and 3G mobile cellular applications.

7.3.1.2 Functional View

The functional view starts with a traditional description of the system with data flow and control paths. The data flow follows the convention adopted in this book (e.g., Figures 1.2, 2.3, and 7.1), where the air interface is on the left and the network interface on the right.

This traditional data flow view is captured as a software reference model, depicted in Figure 7.4. The model is based upon the programmable modular communications system reference model [10]. It serves to introduce the various functional roles performed by the SDR software entities (without dictating a structural model), as well as the control and traffic data interfaces between functional software entities.

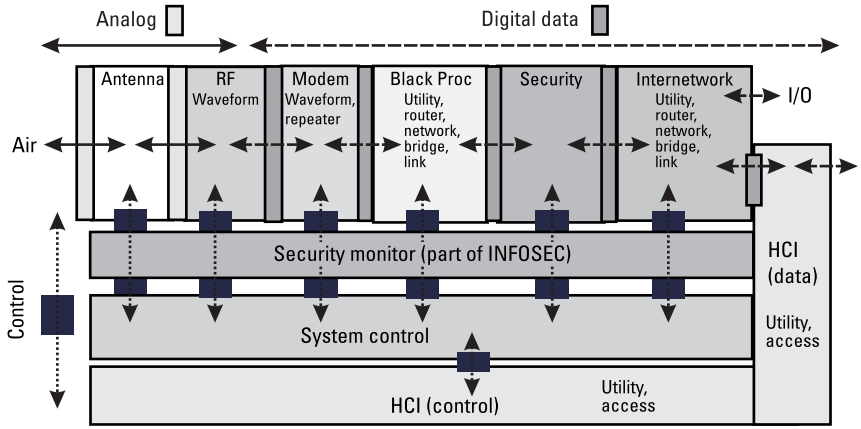

Figure 7.4 Software reference model. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

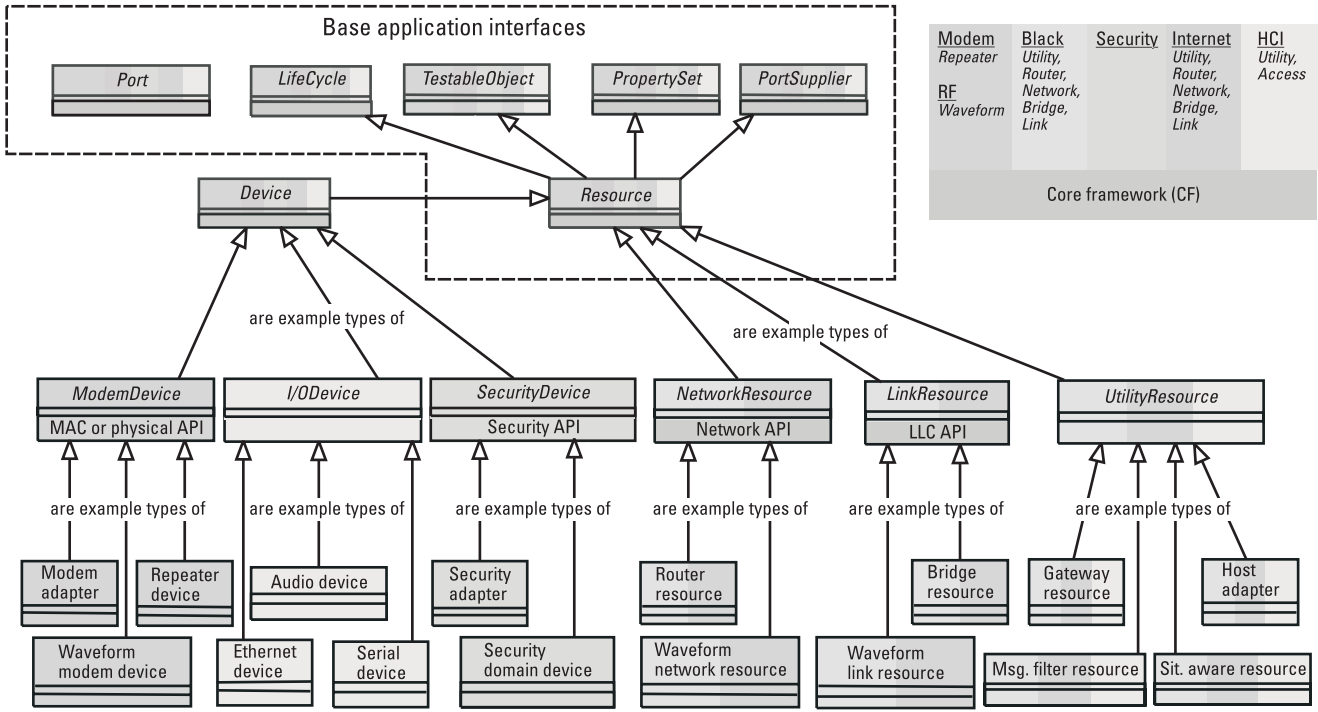

Figure 7.5 Conceptual model of Resources. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

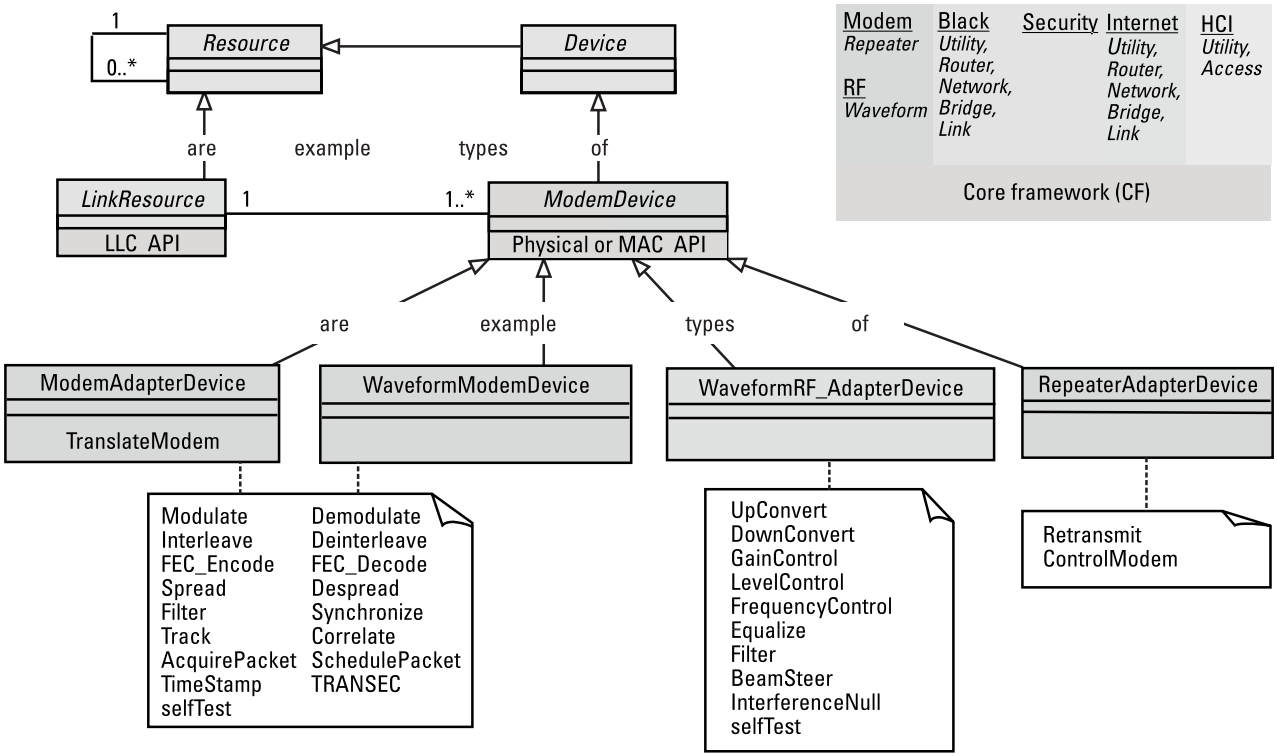

Figure 7.6 ModemDevice conceptual class diagram. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

The remainder of the SCA specification extends this functional view by capturing the software architecture in object-oriented models. The SCAS realizes the software reference model by defining a standard unit of functionality called a *Resource*. All applications are comprised of *Resources* and *Devices* (e.g., the *ModemDevice* includes the antenna, RF, and modem entities) that are types of Resources used as software proxies for actual hardware devices. Figure 7.5 shows examples of implementation classes for *Resources*.

The parent *Resource* is extended by its children in the following ways: *ModemDevice* adds physical devices common to all modems, *LinkDevice* adds link layer interfaces, *NetworkResource* adds network layer interfaces, *I/ODevice* adds a set of input/output devices such as Ethernet and serial, *SecurityDevice* adds security devices, and the *UtilityResource* adds utilities such as message filtering and a network gateway interface. The base application interfaces encapsulated by the *Resource* class provides a mechanism for pushing or pulling messages (*Port*) between *Resources* and *Devices*.

A conceptual class diagram for a *ModemDevice* is provided in Figure 7.6. The *ModemDevice* provides a standard for the control and interface of a modem and encapsulates diverse implementations of smart antenna, RF, and modem functions. Figure 7.6 also illustrates that a parent resource can be extended by adding a child (e.g., *WaveformModemDevice*) with more functionality.

7.3.1.3 Networking Overview

The SCAS includes specification of the external protocols that define communication between an SCAS-compliant software radio and its peer systems. Although the SCAS references many military protocols, it also considers the popular IS-95A CDMA mobile cellular standard as an example.

Figure 7.7 illustrates how the SCAS APIs map onto the OSI seven layer networking model. This mapping is not dissimilar to that used in the cellular mobile world, where the UMTS (3GPP) and CDMA2000 (3GPP2) specifications concentrate on the physical (layer one), link (layer two), and networking (layer three) layers.

7.3.1.4 Core Framework

The core framework (Figure 7.8) is part of the OE and is the essential core set of open application layer interfaces and services to provide an abstraction of the underlying software and hardware layers for software application designers. The CF consists of the following:

- Base application interfaces (*Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory,* and *Resource*), which can be used by all software applications
- Framework control interfaces (*DomainManager, DeviceManager, Application, ApplicationFactory, Device, LoadableDevice, ExecutableDevice,* and *Aggregate*), which provide control of the SDR
- Framework services interfaces, which support both core and non-core applications (*FileSystem, File, FileManager,* and *Timer*)
- A domain profile, which describes the properties of hardware devices (device profile) and software components (software profile) in the SDR

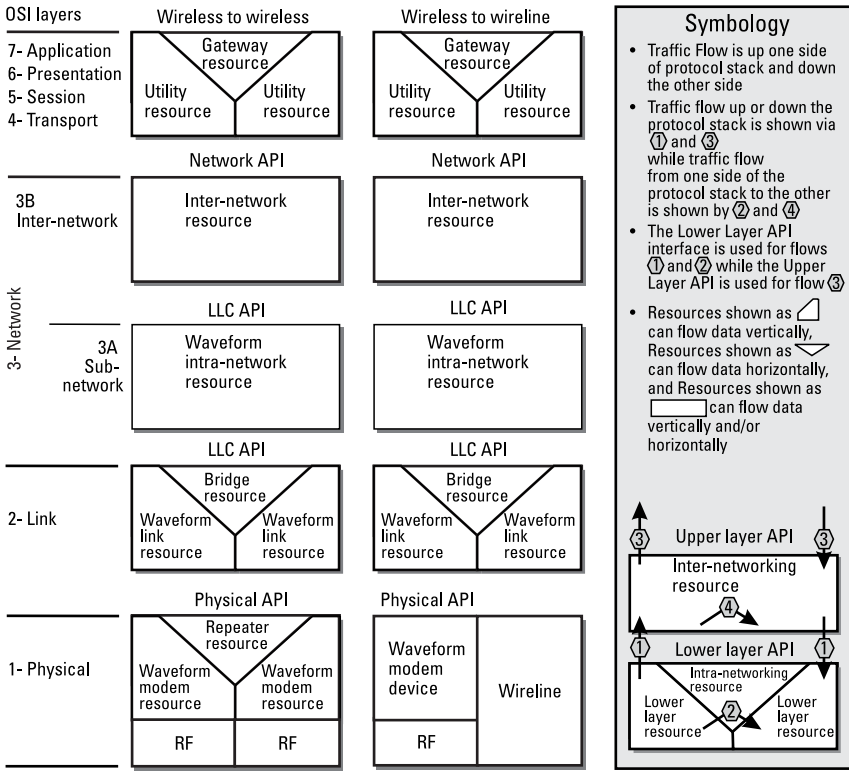

Figure 7.7 Networking mapped to OSI network model. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

As in a UML class diagram, the rectangles in Figure 7.8 represent classes, and the lines connecting classes represent associations. Associations are assumed to be one to one unless annotated as one to many (1 . . . 1 . . .*) or many to many (1 . . . * 1 . . . *); in our example, the *DomainManager* oversees one or many *DeviceManagers,* where the many set includes the number 1. Association lines can have an end adornment; a hollow diamond indicates an aggregation type of relationship (e.g., *FileSystem* is part of *DeviceManager,* and *DeviceManager* is part of *DomainManager*). A triangle at the end of an association line represents a generalization or parent/child relationship. As an example, an *Application* is a type of *Resource,* and as such the *Application* inherits its behavior and interfaces from its parent *Resource*.

The SCAS provides a detailed description of the core framework interfaces and operations. The OMG-developed Interface Definition Language (IDL) is standardized by the International Organization for Standardization as ISO/IEC 14750 [11] and is used to describe these interfaces. According to the CORBA specification [12] an interface is a description of the set of possible operations a client may request of an object. As per the OMG's objectives, the IDL language is independent of compiler, software language, or operating system.

The *DomainManager* (as depicted in Figure 7.9) is a key CF application; its interfaces can be logically grouped into the categories of human computer interface (HCI), registration, and CF administration. The HCI operations are used to configure the domain, get the domain's capabilities (devices, services, and applications), and initiate maintenance functions. Host operations are performed by an HCI client capable of interfacing to the *DomainManager*. The registration operations are used to register/unregister *DeviceManagers*, *DeviceManager's Devices*, *DeviceManager's Services,* and *Applications* at startup or during run time for dynamic device, service, and application extraction and insertion. The administration operations are used to access the interfaces of registered *DeviceManagers* and *DomainManager's FileManager*.

The sequence of operations involved in a *DeviceManager* registering with the *DomainManager* is depicted in Figure 7.10. The sequence starts with the registerDeviceManager operation from the *DeviceManager* to the *DomainManager* and completes with the writeRecord operation from the *DomainManager* to the *Log*.

The other major CF interface is the *DeviceManager*; its job is to manage a set of logical devices and services. Upon startup the *DeviceManager* registers itself with the *DomainManager* and uses a profile (deviceConfigurationProfile)

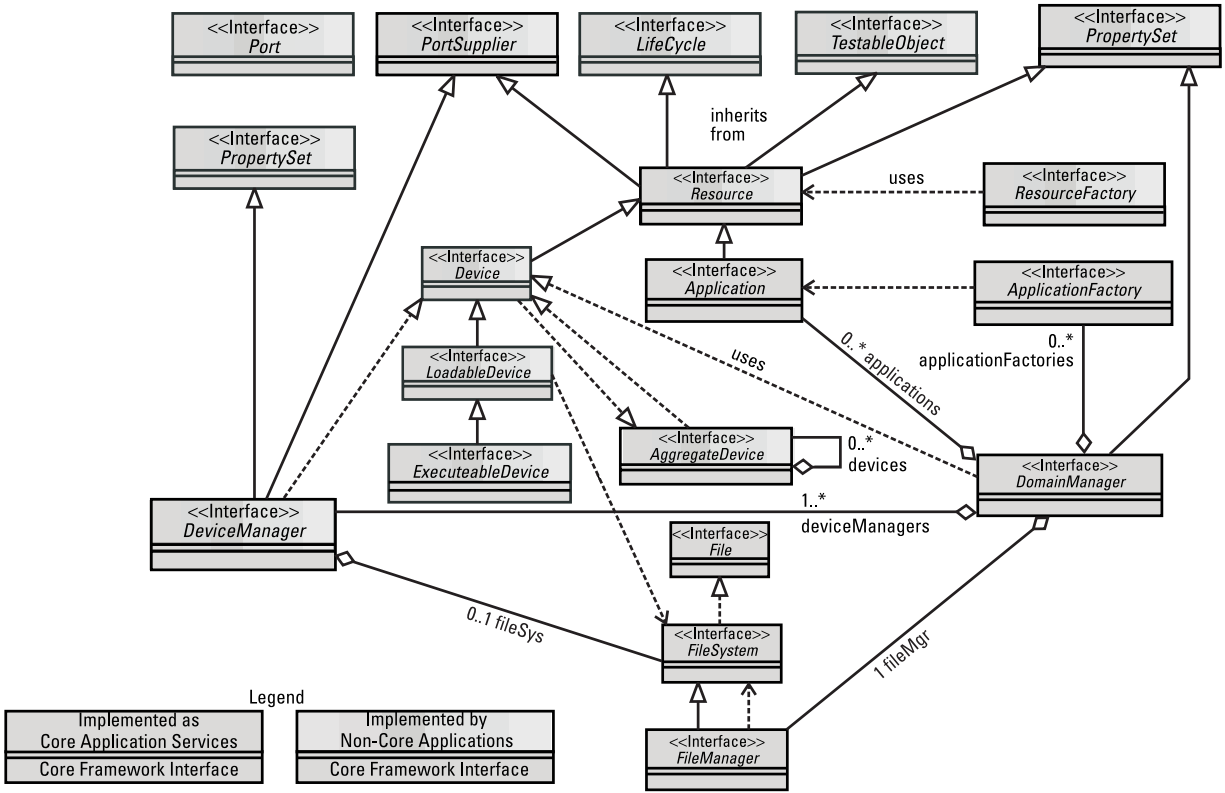

Figure 7.8 Core framework key elements. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

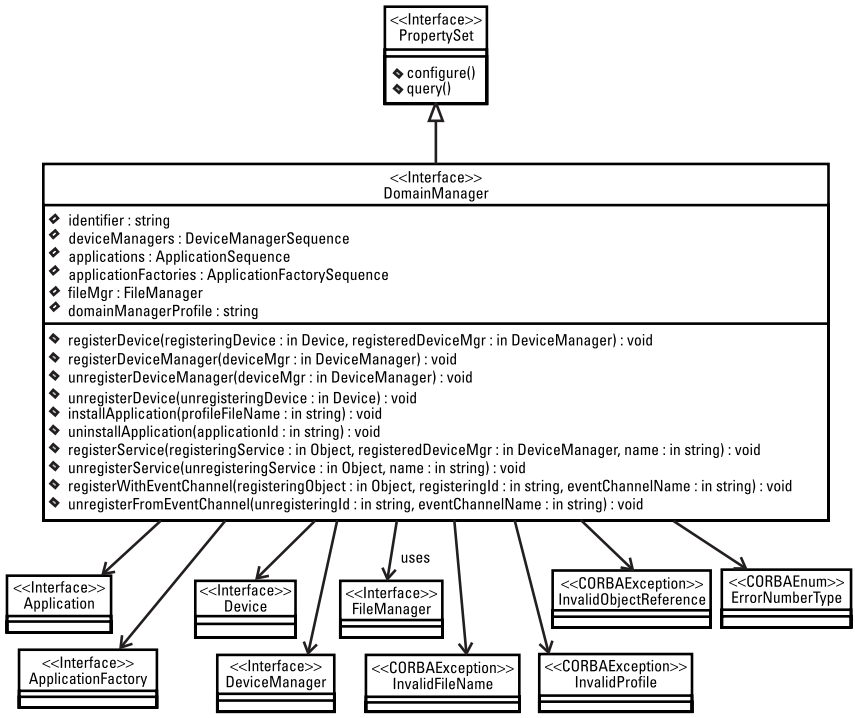

Figure 7.9 DomainManager interface UML. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

attribute for determining the services to be deployed (e.g., *log*[s]), the *Devices* to be created and deployed, and the mount point names for *FileSystems*. It creates *FileSystem* components implementing the file system interface for each OS file in the system. The *DeviceManager* also initializes, configures, and starts logical *Devices* registered to itself. (See Figure 7.11.)

7.3.1.5 Hardware Architecture Definition

The SCAS provides guidance on partitioning the SDR hardware using an object-oriented (OO) approach. The OO method describes a hierarchy of hardware class and subclass objects that represent the architecture. Class structure is a hierarchy that depicts how object-oriented classes and subclasses are related. The class structure in the SCAS identifies functional elements that are used in the creation of physical system elements or hardware devices. As per the OO approach, devices inherit from their parents and share common physical and interface attributes; theoretically, this should make it easier to identify and compare device interchangeability.

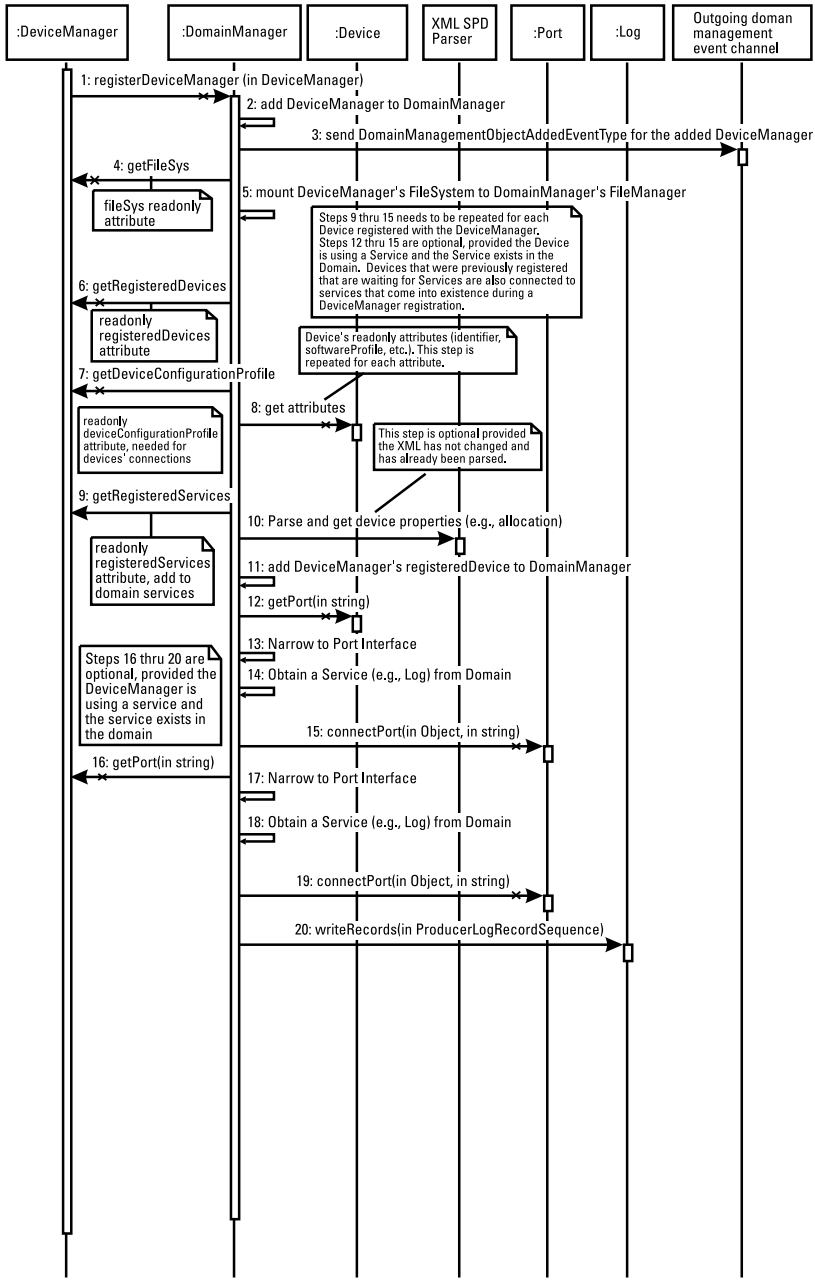

Figure 7.10 Example DomainManager sequence diagram. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

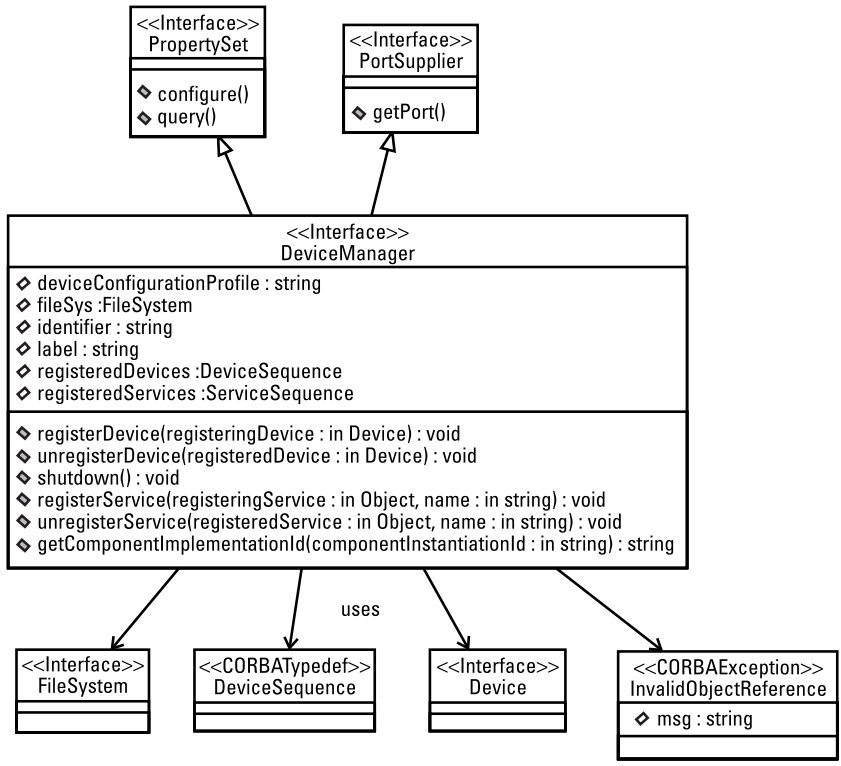

Figure 7.11 DeviceManager interface UML. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

The overall hardware parent is the *SCA-Compliant Hardware* class; it defines attributes such as maintainability, availability, physical, environmental, and device registration parameters. *SCA-Compliant Hardware* has two child classes: *Chassis* and *HW Modules*. The *Chassis* subclass includes the attributes of module slots, form factor, back plane type, platform environmental, power, and cooling requirements. *HW Modules* is the parent to all module subclasses (e.g., *RF*, *Power Supply*, *Modem*, *GPS*, *Processor*, *Reference Standard*, and *I/O*).

Each of the hardware child classes can be further extended, and examples of the granularity of the extensions are depicted in Figure 7.12 for the *RF* class and Figure 7.13 for the *Modem* class.

The *RF* class is extended by the addition of *Antenna*, *Receiver*, and *Power Amplifier* child classes. The *Receiver* class includes many of the parameters discussed in previous chapters. The *Power Amplifier* child class

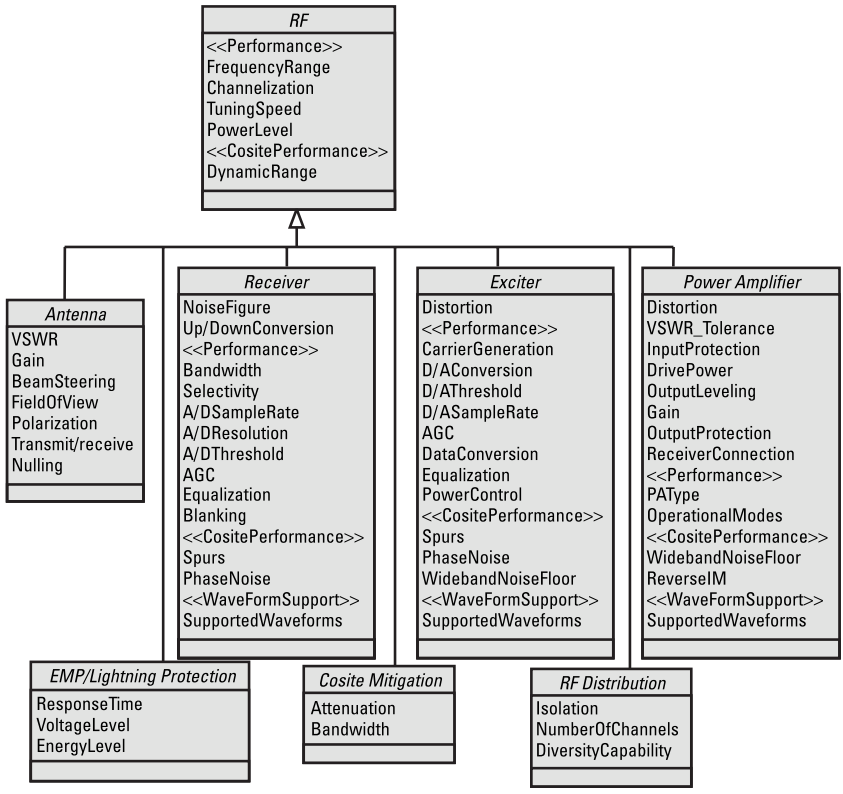

Figure 7.12 RF class extension. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

would need to be extended further for a wideband multicarrier radio and may include additional parameters, such as peak to average power ratio (see Section 3.6.1) or number of carriers. *Cosite Mitigation* implies careful monitoring and control of interference-related parameters; this issue is expected to be a significant one, especially during the deployment of cosited 2G TDMA and 3G TDMA/CDMA systems. The SCAS notes that antennas have been historically passive elements attached to the structure that houses the communications system. In anticipation of technological advances and "smart" antenna (see Chapter 9) functionality, *Antenna* is included as an *RF* subclass with smart antenna parameters, such as BeamSteering and Nulling.

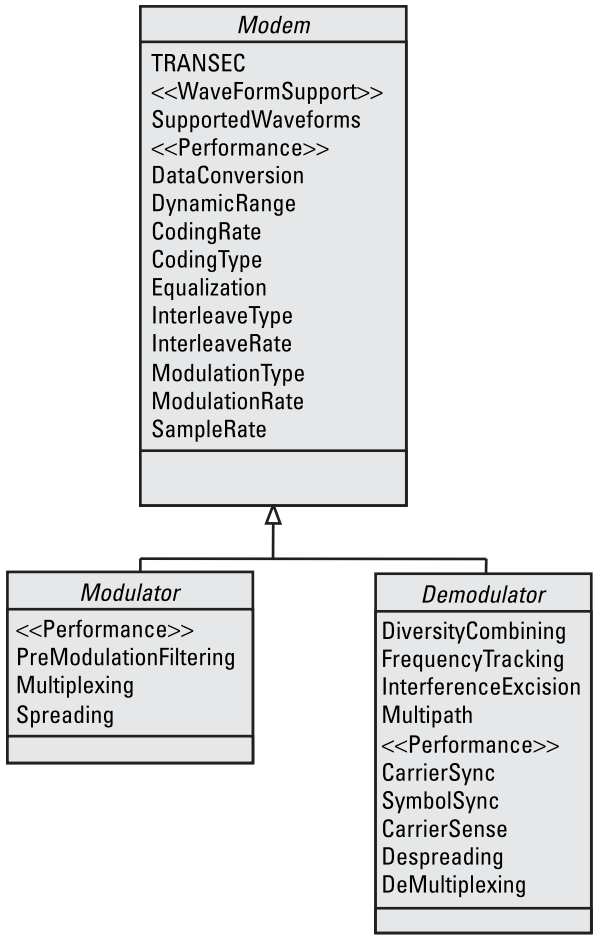

Figure 7.13 Modem class extension. (*Source:* JTRS JPO, 2001. Reprinted with permission.)

The modem class is extended into the transmit and receive subclasses of *Modulator* and *Demodulator*, as illustrated in Figure 7.13. For a multiple air interface mobile cellular radio the SupportedWaveforms attribute may have the valid values of GSM, IS 136, IS-95B, CDMA2000-1xRTT, or UMTS-FDD.

7.3.2 SDRF Distributed Object Computing Software Radio Architecture

7.3.2.1 Architecture Definitions

The architecture is defined by a core framework (CF), operating environment (OE), and rule set.

The core framework is part of the OE and includes a set of core applications, core services, base CORBA interfaces, and an optional core *Factory* interface. The core applications of *DomainManager* and *ResourceManager* are responsible for controlling resources. The core services of *FileManager*, *FileSystem*, *File*, *Logger*, *Installer*, and *Timer* provide support to the core and noncore applications. *Message*, *MessageRegistration*, *StateManagement*, and *Resource* are the CORBA interfaces inherited by the core and noncore software applications. Application life span can be controlled by including the core *Factory* interface.

The operating environment is the integration into an SDR implementation of the CF services and COTS infrastructure (e.g., OSs, bus support packages, and CORBA middleware services). Key to the OE is the inclusion of the software development environment (SDE) used by application developers to develop new capabilities for the SDR.

The rule set is embedded in the behavior, interfaces, and structure defined by the OE and CF. The DOCSRA defines an initial set of rules for the form factor (hardware standard), interfaces, environmental requirements, OSs, and SDE. The emphasis is on open standards, multiple vendor available commercial (COTS) elements, and higher-order software languages within the SDE.

7.3.2.2 Functional View

The functional view starts with a traditional description of the system with data flow and control paths. This traditional data flow view is captured as a software reference model and presented as a means of transitioning to an object-oriented model; it serves to define the functional view and broadly introduce the various functional roles performed by the SDR software entities. The object-oriented model is captured using the structural, logical, and use-case software views, as presented in the following sections.

The software reference model view is not meant to dictate a structural model of the elements. The DOCSRA advises that a software architecture that attempts to force-fit these functions into one place will not provide the flexibility demanded by the various SDR domains, or the reusability desired by software radio vendors and operators [13]. Previous chapters have mostly

left the software architectural issues of an SDR alone. Instead, they have concentrated on defining air interface–specific functionality (antenna, RF, and modem) and performing realistic and efficient partitioning of that functionality considering 3G mobile cellular applications. The DOCSRA advice implies that the software architecture should be flexible enough to allow dynamic functional partitioning and use generic interfaces so that software can be ported from one hardware platform to the next.

An SDR will have several noncore user-oriented (or air interface–specific) software applications. These noncore applications can be considered as resources that inherit common types of behavior and common types of interfaces.

The *Resource* class is extended by its children: *ModemResource* adds physical devices common to all modems, *LinkResource* adds link layer interfaces, *NetworkResource* adds network layer interfaces, *AccessResource* adds a set of multimedia resources, *SecurityResource* adds services such as encryption and authentication, and the *UtilityResource* adds utilities such as message translation and network gateway interface.

The base class interfaces encapsulated by the *Resource* class provide a mechanism for the establishment (*MessageRegistration*) of message paths between resources, the actual path/pipe (*Message*) for communication, and a standard method for managing the resources states (*StateManagement*). Base class interfaces for a given resource (e.g., *ModemResource*) can be overlayed onto an embedded networking architecture by using a networking application program interface (NAPI).

The DOCSRA recognizes that a practical SDR will employ a variety of digital signal processing solutions using hardware (e.g., analog up-/downconverter), programmable hardware (e.g., digital up-/downconverter), and software (e.g., DSP, RCP, *u*P, and so on). Not all of these solutions will be CORBA-capable. To solve this problem the architecture includes a mechanism that allows a CORBA capable resource (e.g., *LinkResource*) to communicate with the noncore applications of the *ModemResource* (e.g., *WaveformModemResource*) via a standard modem NAPI interface. The mechanism that makes this possible is an agent (*ModemAgentResource*), which provides a transparent gateway between CORBA-capable and non-CORBA-capable resources.

7.3.2.3 Structural View

The structural view details the relationships between the OE (CF and COTS) and SDR noncore applications.

The SDR software structure ensures that the noncore applications are abstracted away from the underlying hardware and all entities are connected via a logical software bus by using CORBA. The recommended rule set emphasizes COTS hardware bus architectures (e.g., VME, cPCI, and so on). Real-time COTS operating systems (OSs) are proposed; these OSs are assumed to be POSIX compliant, with the DOCSRA recommending POSIX 1003.13 [8].

7.3.2.4 Logical View

The logical view of the DOCSRA provides a detailed description of the core framework interfaces and operations. As with the SCAS, IDL is used to describe the core framework interfaces.

The *DomainManager* is a key CF application; it has the job of object managing the software *Resources* and hardware components within the SDR. It is possible for software *Resources* (e.g., *ModemResource*) to directly control hardware components (e.g., DSP, RCP, or FPGA). The *DomainManager* allocates *Resources* to one or more *ResourceManager* objects; this decision is based upon many factors, including the visibility of the hardware to the *ResourceManager* and its availability. A prime goal of the DOCSRA is the use of generic resources and portability. This implies the ability to move hardware and software from one system to another without a change in the core framework; in a multiple air interface 3G BTS this may be interpreted as from one air interface to the next. The *DomainManager* is responsible for allocating hardware and software resources based on the functional requirements of the application or applications (in the case of a multiple air interface 3G radio). The *DomainManager* performs the allocation job by using a *DomainProfile*.

There is at least one *DomainManager* for each system, and the *DomainProfile* is used to store the information about the resources in the system. Physical resources such as digital frequency up-/downconverters and DSPs interact with the *ResourceManager* to report their availability; this information is logged into the *DomainProfile*.

The other CF application is the *ResourceManager*; its job is to boot, initialize, and report the capabilities of the hardware modules. The *deviceProperties*, *deviceList*, and *deviceExists* methods are available to the *ResourceManager* for communication back to the *DomainManager*. To ensure interoperability between modules within the SDR, a core list of properties should be defined; these can then be extended by module developers to increase usability as required. A key function of the ResourceManager is the

capability to load and boot software on the various *Resources* it is responsible for managing.

7.3.2.5 Use-Case View

The final view of the architecture uses the UML use-case paradigm. The technique is designed to capture the totality of a system's actual interfaces and interactions. The design method was developed to reduce lost requirements and misinterpretations that result from informal design processes, where requirements may be discussed but not formally documented. The use-case diagram in Figure 7.14 is adapted from Figure 2.2-30 of the DOCSRA [4]; it is a specific example for a potential 3G SDR BTS and the software development environment used to build it.

Within UML, a scenario [14] refers to a single path through a use case—one that shows a particular combination of conditions within that use case. Using the example of Figure 7.14, several scenarios would be developed for the boot up and initialize use case—one where boot-up runs smoothly, another where boot-up encounters problems, and so on. The use-case view of the DOCSRA follows this scenario convention (e.g., a "receive communications" and "transmit communications" scenario is provided for the equivalent of the "TX and RX control messages and traffic" use case).

7.3.3 The OMG

The nearly 800 member companies of the Object Management Group produce and maintain a suite of specifications that support distributed, heterogeneous software development projects from analysis and design through coding, deployment, run time, and maintenance. The specifications are written and adopted by using a well-defined open process. Any company, institution, or government agency can join the OMG and contribute to or influence the specifications.

The specifications are freely available for download from the OMG Web site (http://www.omg.org), and organizations are free to write software implementations that conform to the specifications and use them, give them away, or sell them. Such activities can be undertaken without OMG membership or license.

The OMG does not produce any software and concentrates purely on specification development. Software products implementing OMG specifications (e.g., UML, CORBA, and IDL) are available from hundreds of

sources, including vendor companies and sources of freeware and open-source software.

The OMG also operates a number of special interest groups and has one group for software defined radio, please see http://swradio-omg.org/.

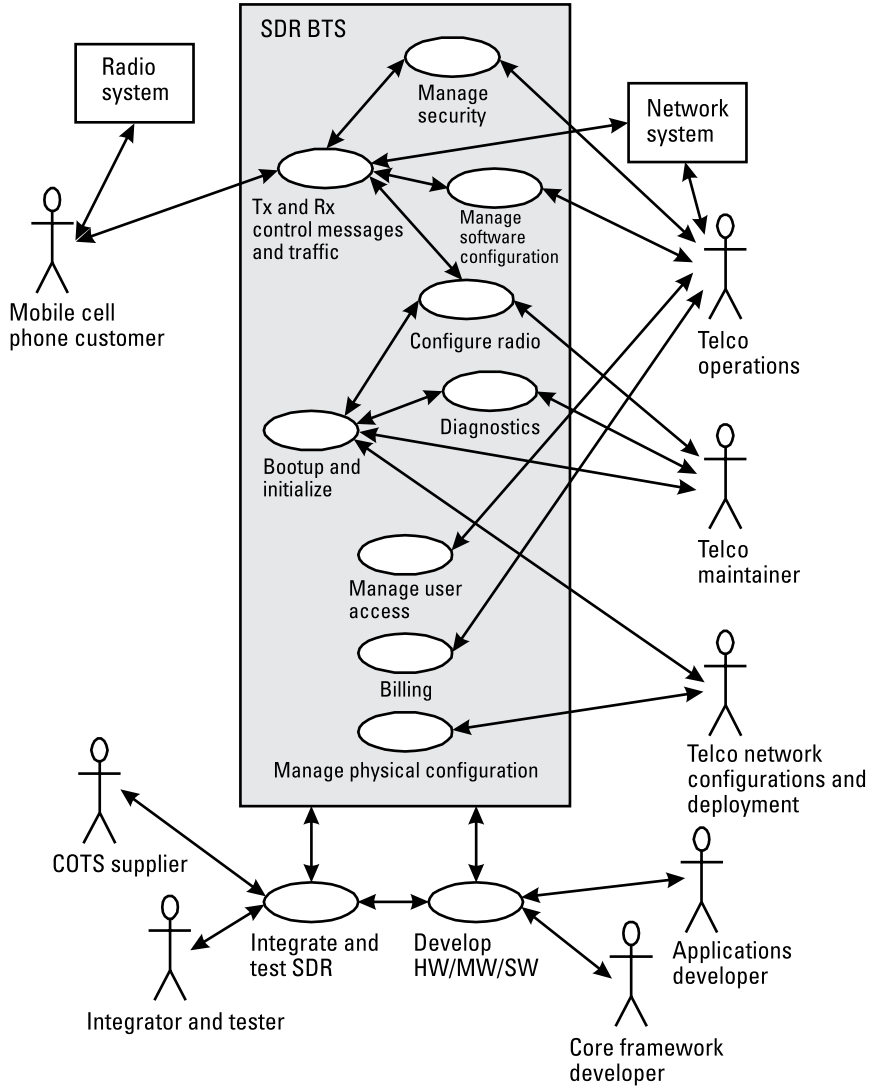

Figure 7.14 3G BTS SDR use cases.

The mission of this group is to:

- Communicate the requirements of the software radio community to the OMG and its various subgroups.
- Promote the use of OMG technologies within the software radio community by direct use of existing and emerging OMG specifications.
- Propose software radio-related extensions to existing and emerging OMG specifications by applying UML modeling techniques.
- Identify and propose new OMG specifications for the software radio community.
- Promote portability, reusability, scalability, and interoperability of software radio-based platforms and applications.
- Form liaisons with related organizations that share common goals and interests.

7.3.3.1 OMG CORBA

The Common Object Request Broker Architecture (CORBA) [12] is defined by the OMG as an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks [15]. By using standard protocols the approach allows CORBA-compliant programs from one vendor to interoperate with those of another vendor, even if the underlying platform (processor, OS, language, and so on) is different. The OMG is responsible for developing and issuing what is now a suite of many documents. The specifications initially concentrated on connecting standalone computers across diverse networks; however, the latest CORBA includes an optional set of extensions called Real-Time CORBA (see Section 24 of [12]).

Each object in the system (e.g., Rake receiver) has its interface defined in Interface Definition Language (IDL). When a client invokes an operation (request) on the object, it must use the IDL interface to specify the operation and to marshal (serialize) the arguments that it sends. The same IDL interface definition is used once the invocation reaches the target object and the arguments are unmarshaled to allow the requested operation to be performed. Figure 7.15 illustrates the communication between software objects using object request brokers (ORBs). The client (e.g., a controller) is invoking different operations (e.g., addUser and setPilot) on two objects (*Rake*

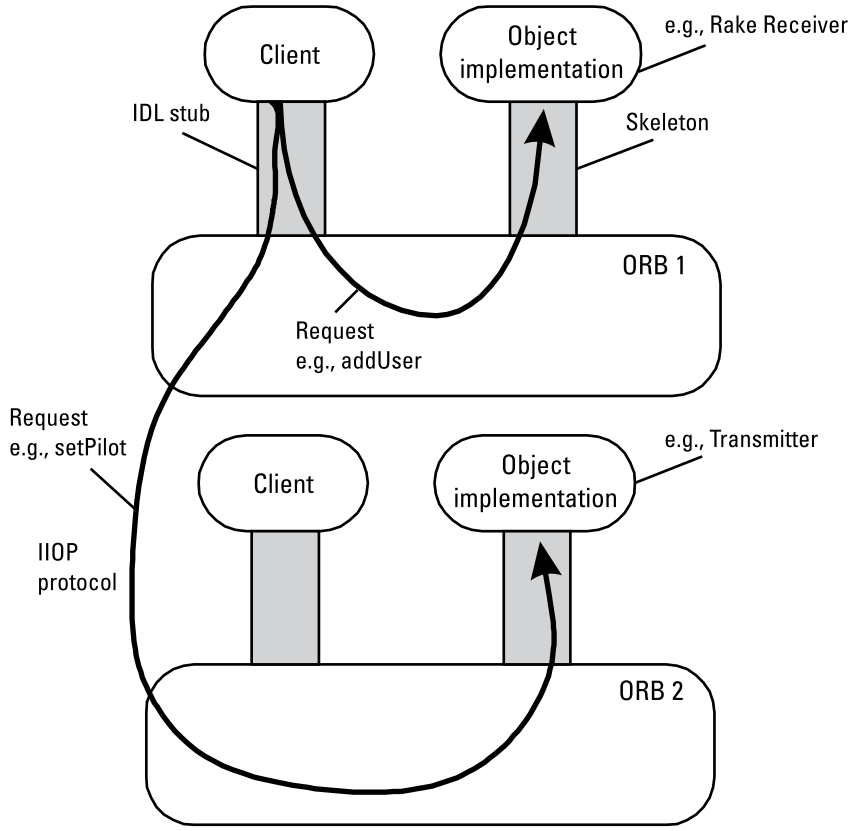

Figure 7.15 Communications via ORBs.

receiver and *Transmitter*) that are connected to different ORBs (ORB1 and ORB2).

Because CORBA strictly defines interfaces as well as the object implementation (code) and data being hidden (i.e., encapsulated), clients do not need to know where objects are or what the underlying platform for each is. The stubs and skeletons act as proxies for the clients and objects, respectively. Perfect communication between the two is achieved even if proxies have been compiled into different languages or run on different implementations of ORB (e.g., *Rake Receiver* may be a unit of software running on a reconfigurable processor, and *Transmitter* is another unit of software running on a DSP).

7.3.3.2 CORBA Performance

The description of CORBA and Figure 7.15 show that there is some overhead in using the method, since interobject communications are not direct and must flow through at least one ORB. Performance issues were recognized by the SDRF, and the organization undertook some performance measurement prior to CORBA's inclusion in the DOCSRA. These measurements looked at the data flowing up and down the CORBA part of the stack and concluded that less than 20% of the processing time was spent in the CORBA part of the stack [4].

The SDRF also found that timing performance depends upon the location of the objects and the capabilities of the particular ORB. They found that collocated objects, where each object is within the same process or address space, provided the best results. The latency for communications was only 20 μs [4] when using a 200-MHz Pentium processor. The time-consuming process of marshaling and unmarshaling data is not required for the collocated case, and this accounts for the result. The worst timing performance was measured when the objects were on separate processors communicating with a GIOP/IIOP-compliant protocol. While performing a bidirectional 64 bytes of data, 420 μs were required between a Sun UltraSPARC 5 with a 270-MHz UltraSPARC IIi.

A "Real-Time CORBA Trade Study" [16] was undertaken by Boeing for the U.S. Defense Department to benchmark the following CORBA implementations: HARDPack from Lockheed Martin Federal Systems, ORBExpress RT from Objective Interface Systems, and The ACE ORB (TAO) from Washington University and Objective Computing, Inc. The study included a survey of users to gain additional statistics, including aspects of usability and documentation.

For the study Boeing ran the tests on a Motorola PowerPC VME-based processor running version 3 of the LynxOS operating system and a SPARC-based machine running version 2.6 of the Solaris operating system. Each test consisted of seven types of IDL operations with nine scenarios. Scenario 3a has the client and server objects on the one PowerPC device and deals with call and return (CR) operation tests plus one-way (OW) operation tests. Four sets of operations were reported for CR and OW: float operations, aligned records, nonaligned (NA) records, and CORBA Any transfers. Data sizes involved in the transfers ranged from 144 to 24,016 bytes. Results were only published [16] for TAO and ORBExpress, and an example is shown in Table 7.1; results are in milliseconds, and some results were removed for clarity (*).

Table 7.1
Call and Return Average Operation Times (in ms)

		Data Size (bytes) >		
Operation	**CORBA ORB**	**144**	**12,016**	**24,016**
Float	TAO	1.15	2.32	3.45
Record	TAO	1.16	3.09	5.12
NA Record	TAO	1.17	4.54	*
Float	ORBExpress	0.15	0.79	1.48
Record	ORBExpress	0.27	0.99	1.89
NA Record	ORBExpress	0.29	1.89	3.68

A major tradeoff for using the standardized method of communications that CORBA offers is that the amount (SLOC) of new communications and infrastructure software needed to be developed is reduced and made easier.

7.3.3.3 OMG IDL

IDL is part of the CORBA specification (Chapter 3). As illustrated in Figure 7.15, IDL is used to define the interface that sits on the outside of an object's boundary and to control how the object communicates with the outside world. This approach uses the principle of encapsulation, where the internal structure and mechanism of an object are kept inside a boundary where the client (e.g., ORB) is not allowed to penetrate. IDL allows the definition of interfaces that both client and server objects understand and can use regardless of platform, operating system, programming language, and so on. These interfaces specify the allowable operations and the associated input and output parameters with their "types," so that client and server can encode and decode values for travel over the communications medium. IDL enforces object orientation and strong exception handling to reduce the problems associated with incorrect invocations.

The following IDL interface is a very simple example. The object's "type" is transmitter, and it can perform two operations: setPilot and setTrafficChPwrInc. When increasing the traffic channel power for a given user (setTrafficChPwrInc), the transmitter accepts an unsigned integer "walshcode" specifying for which channel the function must be performed. The

return value for setTrafficChPwrInc does not need a name and is also an unsigned integer; this may be used to indicate the success of the operation.

```
interface transmitter {
unsigned_int setPilot ();
unsigned_int setTrafficChPwrInc(in unsigned_int
walshcode);
}
```

7.3.4 Software Design Patterns

Software development has sufficient history behind it so that each new software project is almost certain to include design requirements and problems that have occurred in at least one preceding project. The advent of object-oriented programming practices has propelled the volume of software being developed, and this has helped to foster efforts to recognize common problems and publish well-designed solutions for them. These solutions are known as "design patterns," and the book *Design Patterns Elements of Reusable Object-Oriented Software* [17] details the four elements of a pattern: name, problem, solution, and consequences. Reference [17] details 23 patterns, including factory, singleton, proxy, builder, visitor, and others.

7.4 Component Choices

The first part of this chapter has considered possibilities for an SDR software architecture. These architectures are abstract and by their nature are independent of operating system and language used to develop the application software. This section briefly covers a range of operating systems and languages that can be selected during the detailed design phase for use during implementation.

7.4.1 Real-Time Operating Systems

The choice of an operating system will be driven by many requirements, including support for the target processor, real-time features, development tools, and cost (development and production).

7.4.1.1 LINUX & RT LINUX

LINUX [18] is a free UNIX type of operating system that is made available under the GNU general public license [19]. It was originally developed by Linus Torvlads in 1991, when it was released as version 0.02. The licensing arrangement ensures that users can obtain source code for the operating system. Once a user has the source code, he or she is permitted to create new versions, which they can charge for, but the new source code must be made available back to the community. The open source, low-cost environment has spurred on thousands of developers worldwide to contribute to LINUX, with the result that the operating system is now very widespread and increasing in popularity at an exponential rate.

While LINUX has been particularly popular in the PC world, it is generally not suitable for hard real-time embedded systems. Standard LINUX takes up to 600 μsec to start a handler and can be more than 20 μsec late for a periodic task [20]. The LINUX operating system is optimized for the general case and has some fundamental features that contradict real-time requirements. For example, LINUX will not preempt the execution of the lowest-priority tasks during system calls, synchronization is not fine enough and causes long periods when data is tied up by a non-real-time thread and unavailable to real-time threads, and LINUX will make high-priority tasks wait for low-priority tasks to release resources.

Real-Time LINUX (RT LINUX) has been developed to solve the shortcomings of standard LINUX and is now available to meet the growing need for a low-cost, real-time embedded operating system. RT LINUX treats the LINUX kernel as a task executing under a small, real-time operating system. The design has LINUX as the "idle" task for the real-time OS and only executing when there are no real-time tasks to run. In this mode the LINUX task cannot prevent itself from being preempted or allow the blocking of interrupts. This is achieved by the RT LINUX kernel intercepting LINUX requests to disable interrupts and recording them and then returning to LINUX. If there is a handler for the real-time event, it will be invoked. This ensures that LINUX cannot add latency to the real-time interrupt response time no matter what state LINUX is in.

RT LINUX is distributed by a commercial organization (FSMLabs), which was founded by the original product creators.

7.4.1.2 VxWorks

The VxWorks real-time operating system (RTOS) is a commercially available product from WindRiver. The operating system is a closed proprietary offer-

ing; however, it has grown in popularity and was selected for the high-profile 1997 Mars Pathfinder Lander project. WindRiver recently took over the pSOS operating system, and it is widely expected that VxWorks and pSOS will be merged in the future into a single product.

The operating system is a good choice for systems that only use general-purpose microprocessors (e.g., PowerPC, Intel Pentium, ARM, SPARC, MIPS, and so on). Major disadvantages for VxWorks when considering software radio are that the operating system does not support any digital signal processing devices (e.g., TI or Analog Devices), and it imposes a royalty fee for every deployed instance of the RTOS on a processor.

Features include unlimited multitasking, preemptive scheduling, round-robin scheduling, 256 priority levels, POSIX 1003.1 compatibility, and a good range of diagnostic tools. For larger, embedded systems there is support for networking protocols such as TCP/IP, PPP, FTP, SNTP, and others.

7.4.1.3 OSE

The OSE RTOS by Enea is similar to VxWorks in that it is also a closed proprietary and commercially available system with a similar licensing structure (royalty per instance). The advantages of OSE is that it supports both general-purpose microprocessors and DSPs and is certified for use in systems requiring a level of safety integrity [21].

7.4.1.4 MQX

A commercially available product with a different supply model is the MQX RTOS by Precise. The supplier has chosen a halfway house between LINUX and VxWorks by supplying the source code for MQX and making it royalty free. The RTOS suits mixed processor environments and is available for RISC and CISC microprocessors (e.g., PowerPC, ARC, ARM, MIPS) and DSPs (TIC6x, TIC5x, TIC4x, TIC3x, and ADSP2106x). By being provided with the source code for the operating system, the user has the choice to port to other processors if required.

MQX presents the user with a standard API regardless of the processor used.

7.4.1.5 DSP/BIOS

DSP/BIOS is a kernel provided by Texas Instruments that provides real-time operating support for DSPs. The kernel supports preemptive multitasking and other services that enable applications to more effectively use event-

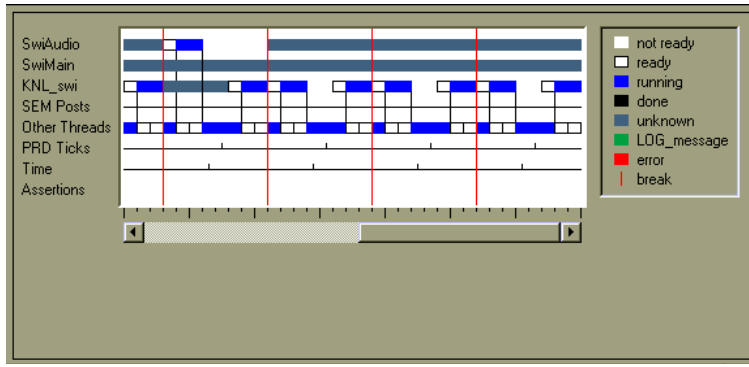

Figure 7.16 Example software and hardware tasks in DSP/BIOS.

driven and interrupt service paradigms. Application software can take advantage of traditional multitasking services, mailboxes, queues, semaphores, and resource protection locks.

Developers have the flexibility of selecting several I/O mechanisms, including data pipes and data stream models. The configuration of the kernel object programming model can be static or dynamic, and the kernel allows run-time memory management, providing dynamic memory allocations and deallocations. Application management and configuration of resources can be performed dynamically, thereby enabling developers to build self-configuring and more complex applications whose mix of functions changes over time.

Figure 7.16 illustrates the results of a running DSP/BIOS program as viewed through the Code Composer execution graph. Event resolution was set to one-millisecond intervals, and the figure shows the operating system switching between the kernel task (KNL_swi, a do nothing idle function), the SwiAudio function (software interrupt controlled), and Other Threads (includes hardware interrupts).

7.4.2 High-Level Software Languages

There are many high-level software languages; however, the most widely used in commercial software defined radio implementations are C and C++.

7.4.2.1 C Code for DSP

Dennis Ritchie of AT&T Bell Laboratories invented the C language in 1972 as a means of implementing the UNIX operating system in a high-level lan-

guage. The design of C and its standardization by ANSI has led to a mass adoption of the language and a high degree of portability across many processors. Therefore, C was an obvious choice when DSP manufacturers first upgraded their assembly language compilers to accommodate a high-level language.

The following simple two-line C program loops through a series of additions, where the loop counter "i" is added to the "result" for each pass through the loop.

```
for (i = 0; i < n; i++) {
     result = result+i;
    }
```

This C program can be written by almost any programmer in a very short period and compiled to run on a general-purpose Intel type processor or a DSP. An example of the assembly code produced by the Code Composer Studio compiler for the Texas Instruments' C62 DSP follows:

```
     ZERO.D2          B4
     STW.D2T2         B4,*+DP[0x11C]
     LDW.D2T2         *+DP[0x11B],B4
     LDW.D2T2         *+DP[0x11C],B5
     NOP              4
     CMPLT.L2         B5,B4,B0
[!B0]B.S1             L2
     NOP              5
     L1:
     LDW.D2T2         *+DP[0x11A],B4
     LDW.D2T2         *+DP[0x11C],B5
     NOP              4
     ADD.D2           B5,B4,B4
     STW.D2T2         B4,*+DP[0x11A]
     NOP              2
     LDW.D2T2         *+DP[0x11B],B5
     ADD.S2           1,B5,B4
     STW.D2T2         B4,*+DP[0x11C]
     NOP              3
     CMPLT.L2         B4,B5,B0
[ B0]B.S1             L1
     NOP              5
     L2:
     ZERO.D1          A4
     B.S2             B3
     NOP              5
```

The high-level language capability of C abstracts the program to such a point that it becomes independent of the target processor that will perform the execution. It becomes the job of the compiler to decide on which hardware-specific instructions to use, and this example illustrates that the result can be a significantly large number (26 in the example) of assembly language mnemonics.

In the past, most mass-manufactured radio products with a DSP software component have used assembly language to achieve the lowest cycle count and highest performance. Today's DSP C compilers are approaching efficiencies obtained by directly written assembly.

7.4.2.2 C++

C++ is an extension of the C language developed in 1983 by Bjarne Stroustrup of Bell Labs. The language was designed for UNIX and aimed to make programming easier and code more portable. It is an OO language and provides full inheritance, polymorphism, and data binding features. A major distinction between OO and structured languages such as C is that data and functions (operations on data) can be combined to create objects.

The language is starting to find its way onto DSP; however, at this stage C remains the dominant DSP language, particularly when code size is an issue. C++ is, however, an excellent choice for implementing the higher-level layers two and three protocol software in a cellular mobile phone or base station.

7.4.3 Hardware Languages

The hardware languages of VHDL and Verilog are widely used to program devices such as FPGAs and RCPs. Both languages specify functionality at various levels of abstraction from behavioral (most abstract) to gate level (most detailed). These languages can, however, be bypassed in favor of programming at the register transfer logic (RTL) level, where the functionality between individual storage elements (or registers) and timing is specified.

These hardware languages continue to be extended (e.g., Verilog 2001) and provide more abstraction and flexibility; however, they may be overtaken by recent efforts to target much higher level languages such as Java to FPGAs. Xilinx now provides Java support for the Virtex II range of FPGAs; the "LavaCore" instruction set implements the Java virtual machine instruction set, as specified in "The Java Virtual Machine Specification" by Lindholm and Yellin.

7.4.3.1 VHDL

VHDL, or VHSIC Hardware Description Language, has been around for many years and was developed as a consequence of the U.S. government program named Very High Speed Integrated Circuits (VHSIC). Conclusions from this program led to development of VHDL, a language for describing the function (or behavior) and structure of integrated circuits. The language is now standardized and available from the IEEE [22].

Complicated behavior in digital circuits cannot always be described by way of structures, inputs, outputs, and Boolean equations. These static methods cannot adequately deal with timing dependencies, and for this reason VHDL is designed as an executable language with similarities to the Ada high-level software language. VHDL is not as fully featured as Ada but includes a range of features, including identifiers, comments, literal numbers, strings, and data types such as integer and so on.

The following VHDL code illustrates a simple "for" loop example:

```
Architecture A of
CONV_INT is
begin
     process(VECTOR)
          variable TMP: integer;
     begin
          TMP := 0;
          for I in 7 downto 0 loop
               if(VECTOR(I)='1')
          then
               TMP := TMP + 2**I;
          endif;
     endloop;

     RESULT <= TMP;
     end process;
end A;
```

7.4.3.2 Verilog

Verilog is also a hardware description language (HDL); it predated VHDL and started out in 1985 as a proprietary language. Competition from the open VHDL language then forced Verilog down the standardization path and it too is available from the IEEE [23]. If VHDL is similar to Ada, then Verilog has structure akin to the C high-level language.

The following Verilog code illustrates a simple "for" loop example:

```
integer list [31:0];
integer index;

initial begin
     for(index = 0; index < 32; index = index +1)
          list[index] = index + index;
     end
```

7.5 Conclusion

This chapter has covered emerging software radio standards and the technologies being used to specify and support them. We have proposed expanding the software radio definition to include the use of object-oriented methodologies. The extent to which OO can be pushed into the core of a real-time software radio will rely on trading off performance and efficiency; however, this can be a staged process providing the software architecture supports it. The scope of the chapter has only allowed a cursory review of a range of software and hardware languages; there are many excellent books available for readers entering the detailed design and implementation stage of a software radio project.

References

[1] ETSI, "Base Station Controller—Base Transceiver Station (BSC-BTS) Interface: General Aspects," GSM 08.51, August 1999.

[2] Lynes, D., "Cellular BTS SDR Framework," *2001 International Conference on Third-Generation Wireless and Beyond*, San Francisco, CA, May 30, 2001.

[3] Joint Tactical Radio System (JTRS) Joint Program Office, "Software Communications Architecture Specification MSRC-5000SCA V2.2," November 17, 2001.

[4] Software Defined Radio Forum, "Distributed Object Computing Software Radio Architecture v1.1," July 2, 1999.

[5] IEEE, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems IEEE Std 1471-2000," October 9, 2000.

[6] Object Management Group, "OMG Unified Modeling Language Specification." Version 1.3, March 2000.

[7] Mercury Computer Systems, "AdapDev SDR Brochure," DS-5C-30, 2000.

[8] IEEE, "1003.13-1998 IEEE Standard for Information Technology—Standardized Application Environment Profile (AEP)—POSIX Real-time Application Support," 1998.

[9] Obenland, K., "The Use of POSIX in Real-Time Systems, Assessing Its Effectiveness and Performance," http://www.mitre.org/support/papers/tech_papers99_00/obenland_posix/obenland_posix.pdf, 2000.

[10] "Programmable Modular Communication System (PMCS) Guidance Document," July 31, 1997.

[11] International Organization for Standardization, "ISO/IEC 14750:1999 Information Technology—Open Distributed Processing—Interface Definition Language," 1999.

[12] Object Management Group, "The Common Object Request Broker: Architecture and Specification CORBA 2.4.2," February 2001.

[13] Software Defined Radio Forum, "Distributed Object Computing Software Radio Architecture Vol. 1.1, July 2, 1999, pp. 2–11.

[14] Fowler, M., and K. Scott, *UML Distilled Applying the Standard Object Modeling Language*, Reading, MA: Addison-Wesley, 1997, p. 50.

[15] http://www.omg.org.

[16] Boeing, "Real-Time CORBA Trade Study," D204-31159-1, January 10, 2000.

[17] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1998.

[18] http://www.linux.org.

[19] http://www.gnu.org/licenses/gpl.html.

[20] Yodaiken, V., "The RT LINUX Manifesto," Department of Computer Science, New Mexico Institute of Technology, http://www.fsmlabs.com/developers/white_papers/rtmanifesto.pdf.

[21] Enea OSE Systems Inc., "Overview of the IEC 61508 Certification of the OSE RTOS," R1.0, 1999.

[22] IEEE, "VHDL IEEE Std 1076-1993," 1993.

[23] IEEE, "Verilog IEEE Std 1364-1995," 1995.