# DESIGNING JTRS CORE FRAMEWORKS FOR BATTERY-POWERED PLATFORMS: 10 TECHNIQUES FOR SUCCESS

Charles A. Linn (Harris Corporation, RF Communications Division, Rochester, NY, USA.
clinn@harris.com

## ABSTRACT

The recent emergence of several key technologies in the "embedded" realm has resulted in Software-defined radio (SDR) systems "coming of age" in offering contemporary radio communications solutions. As one of the most developed and evaluated architectural specification efforts in this area, the US Government Joint Tactical Radio System (JTRS) Software Communications Architecture (SCA) specifies an open, standardized architecture for government software-defined radios. Developing an SCA-compliant JTRS core framework for small, battery-operated platforms can pose a significant challenge, however, due to the constraints of these platforms coupled with high user performance expectations. As part of Harris' work in validating the SCA for battery-powered platforms, we have identified a number of considerations specific to these small platforms. This paper presents ten such techniques that can be used to develop a "lightweight core framework" that succeeds in this most challenging area.

## 1. INTRODUCTION

Of the architectures that have been explored for standardizing software-based radios, no other has received as much attention, nor been as completely developed as the US Government's Joint Tactical Radio System (JTRS) Software Communications Architecture (SCA) [1]. The SCA specifies the interfaces and rules for the following elements:

- *Operating System / ORB* – This base environment, on which all other JTRS elements reside consists of a POSIX operating system and a CORBA ORB that provides distributed processing services
- *Core Framework* (CF) – a CF provides a number of components with standardized interfaces that are responsible for installing, instantiating, managing and tearing down JTRS applications. A CF running on top of an OS/ORB is sometimes referred to as a JTRS *Operating Environment* (OE).
- *JTRS applications* – applications are typically communications waveforms that are installed and run on a JTRS platform. Applications run "on top" of the JTRS OE.

The SCA targets a vast expanse of radio platforms, including large fixed mount platforms, shipboard, airborne, ground/vehicular as well as small, battery-powered manpack and handheld devices.

Any architecture that targets this wide range of platforms must make engineering tradeoffs between the capabilities required (and supported in hardware) by the large platforms and the limited capacities of the smaller platforms. The SCA employs a very reasonable set of these tradeoffs, but it remains a formidable challenge to implement both applications and core frameworks that work well in the smaller, battery-powered platforms.

Harris Corporation has been a leader in the development of software-based tactical and strategic radios since the introduction of the all-software RF-5000 vehicular radio in 1988. More recently Harris has been performing work for the US Joint Program Office under a " JTRS Step 2B" contract to validate the SCA on manpack and handheld platforms. This work has included the development of a JTRS manpack form-factor evaluation platform, as well as development of core frameworks, a software re-programmable cryptographic subsystem, and several test applications.

In this paper we will focus techniques for developing core frameworks that are specifically optimized to target small, battery powered platforms such as government / military manpack and handheld radios while retaining SCA compliance.

## 2. CORE FRAMEWORK STRUCTURE

Although it is not within the scope of this paper to describe the detailed structure and requirements for SCA core frameworks, a high-level class structure is depicted in Figure 1. This figure shows all of the major instantiated classes defined in the SCA, as well as the principal interactions between them. For the purposes of this paper, two core framework subsets are defined. The first we refer to as the *Upper Core Framework*. The upper core framework is resident on exactly one CORBA-capable GPP in the radio. This subset of the SCA-defined elements provides radio-wide services not associated with any particular CORBA node. The *Lower Core Framework* consists of a set of classes that provide node (processor) specific services. There are typically as many lower core framework subsystems as there are CORBA capable processors in the radio.

The primary client of the core framework is the Human-machine Interface (HMI). This package, which not only includes the user interface itself, but also includes any other platform code that falls outside the SCA Core Framework or Operating System.

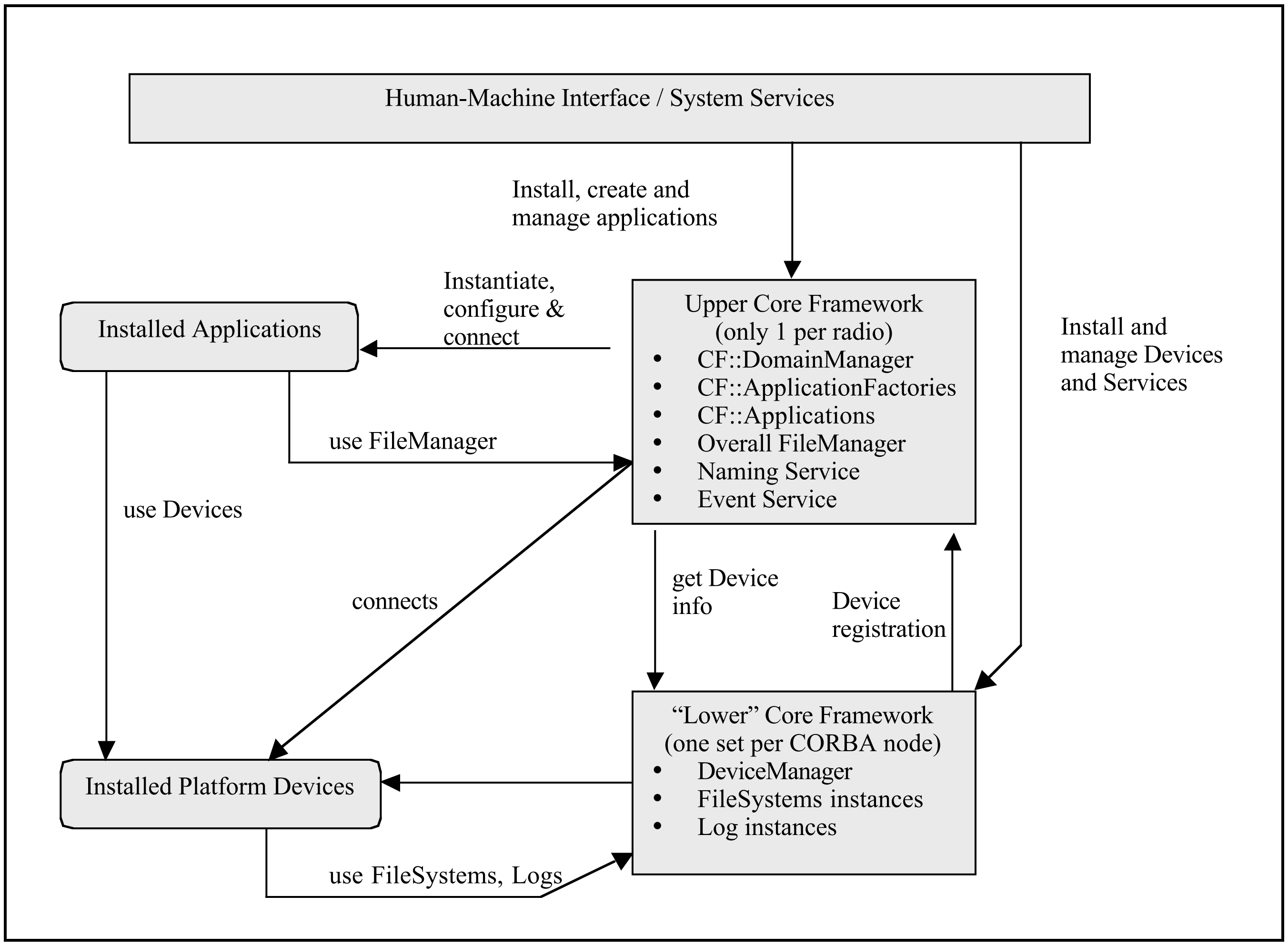
The DomainManager, a part of the upper core framework, forms the primary contact point for the HMI. It, along with the other upper core framework elements, is responsible for creating and managing applications, as well as providing (and controlling) access to the lower core framework services.

The DeviceManager primarily represents the lower core framework. This component, of which there is one per CORBA node, is responsible for creating and managing all of the platform Devices, FileSystems and Logs for a given node. It registers itself (and in turn its components) with the DomainManager.

## 3. SMALL PLATFORM CHARACTERISTICS

To understand these challenges and in turn address the means to overcome them, let us first examine the characteristics of small, battery powered platform by examining a representative single channel handheld platform employing a cryptographic subsystem. When designed as a JTRS-compliant platform, such a platform will typically employ two general-purpose processors (GPP) capable of supporting CORBA traffic, a programmable cryptographic subsystem (CSS), and one or more digital-signal processors (DSP) and / or Field-programmable Gate Arrays (FPGAs). The two CORBA-hosting GPPS will be distributed on the two sides of the CSS—one of the unencrypted (Red) side, and the other on the encrypted (Black) side.

In a handheld platform, each of the GPPs will have the processing power ranging from a higher-end Personal Digital Assistant (PDA) class processor to that of a low-end laptop processor (25 – 250 MHz). Main execution memory for each processor will be between 16 MB to 64 MB, and program storage typically will be FLASH based. DSP and FPGA components will in handhelds be employed to accomplish mathematically intensive low-level modulation functions and critical (sub-millisecond) timing functions. All modulation and demodulation through Intermediate-Frequency (IF) will be accomplished with programmable (software) components.


Human-Machine Interface / System Services
Install, create and manage applications
Instantiate, configure & connect
Installed Applications
Upper Core Framework (only 1 per radio)
CF::DomainManager
CF::ApplicationFactories
CF::Applications
Overall FileManager
Naming Service
Event Service
Install and manage Devices and Services
use FileManager
use Devices
connects
get Device info
Device registration
"Lower" Core Framework (one set per CORBA node)
DeviceManager
FileSystems instances
Log instances
Installed Platform Devices
use FileSystems, Logs

In comparison with current (non-JTRS) handheld platforms, this represents both a tremendous increase in processing and program storage capacity as well as a very challenging packaging challenge. On the other hand, even when several 200 MHz processors are employed it still has a fraction of the processor capability that a ground/vehicular or shipboard JTRS platform.

Ironically, at the same time small platforms are limited in processing capability, they often have stricter requirements for platform and application responsiveness. Since they are battery powered, and operational time is always at a premium, it is typical for radio users to turn the radio on, use it for a brief period of time, then turn it off again. This usage scenario puts a premium on minimizing radio power-up time, which is expected to be less than 10 seconds – considerably less than the time most "standard" JTRS systems need to power-up. In a similar manner, users expect handhelds and manpacks to be able to change channels and waveforms more quickly (in several seconds at most) than fixed-side equipment, as they view manpacks and handhelds as *actual radios*, whereas the fixed-side equipment is typically viewed through a PC-like interface, and hence subject to (less responsive) PC expectations.

The combination of these factors makes JTRS manpack and handheld application and core framework development a challenging task indeed. In this paper we will address ten "techniques for success" that we have found are key in developing lightweight core frameworks optimized for these small platforms.

## 3. TEN TECHNIQUES FOR SUCCESS

### 3.1 Optimize while maintaining SCA compliance

In most cases a core framework that works well on small platforms can be produced by merely *optimizing* the implementation, rather than changing the SCA specification or producing a non-compliant implementation. There are several ways that can be employed to effectively extend the SCA to meet a core framework's needs.

The SCA specifies specific interfaces and behaviors that must be supported by the CF elements, and to be a compliant implementation, these interfaces must be implemented. However, in most cases CF implementation details are intentionally left unspecified, leaving considerable freedom to the implementer.

First, it is up to the implementation to choose how the CF elements are allocated to different process spaces. In addition, it is assumed that the same manufacturer is creating the DomainManager, ApplicationFactory and Application classes. Since these components can communicate using interfaces in addition to those defined in the SCA, several options exist:

1. If CF elements are located in the same address space, standard native language calls can be employed between the CORBA servant classes.
2. If CF elements are not in the same address space, than their interfaces can be extended with additional operations by inheriting a specialized interface from the SCA CF Interface Design Language (IDL). In this way, the "external world" sees the standard interface, but classes "in the know" can narrow the CORBA object reference to get the richer interface.

This second technique is particularly useful when working in a system with CF components that are not guaranteed to be implemented by the same CF implementer, but perhaps are. This situation arises in several cases, one example would be between a DomainManager and a registering DeviceManager. For example, if, company A's DomainManager sees a DeviceManager registering with it, it can try to narrow the DeviceManager's CORBA object reference to the (expanded) interface provided by company A's DeviceManager. If the narrow succeeds, then the extended interface can be used. If the narrow fails, then the DomainManager must use the standardized interfaces common to all DeviceManagers. In this way compliance is maintained but enhanced operation is still allowed when the circumstances permit.

## 3.2 Parse XML only on installation

The SCA specifies that applications, application components ("Resources"), Devices and Services be all described by XML. The core framework uses the information in the XML to determine (to just name a few) what components need to be created, how to find the created instances, what interfaces these components provide, how they need to be configured, and how they need to be connected. In addition, a number of "dependencies" are specified in XML that indicate what sort of Devices are required by a given application component, or on what type of processor (Device) the component should be deployed.

To efficiently use this information, the XML, which is essentially a text file, must be parsed into a computer readable form. This parsing is typically done using a commercially available XML parser, and due to the amount of XML that has to be parsed, can be a rather lengthy (multi-second) process. The SCA does not specify (either for applications or Devices) when this parsing takes place, and some core frameworks parse all application and device XML each time the radio is turned on. For small platforms it is essential that this parse process take place as part of the installation process or on the next power-up following installation (when the user can tolerate a "one-time" increased boot time). The already-parsed information can then stored to a (non-XML) file for fast restoration on future system boots. Using this technique will shave valuable seconds from the radio boot time.

## 3.3 Use digested profile information

Although the SCA does not specify how XML is to be parsed, one of the more common methods is to employ a commercial parser that creates a "Domain Object Model" (DOM) standard database. Although in larger systems it is often acceptable to use the DOM model on an ongoing basis for CF decisions, this method has some drawbacks for small platforms.

First of all, the generated DOM databases corresponding to applications, components and Devices are often very large. DOMs are constructed to exactly mirror the XML they parse, and in the JTRS system there can be many, many lines of this XML. Based on Harris' experience in our Phase 2B validation contract, where two small applications were developed yielding a total of 4700 lines of XML, and also considering the fields both required and optional from the SCA appendix D, it is easy to believe that 50K to 100K of XML will be typical in radio hosting a full suite of applications and devices. Many of these XML fields (e.g. all fields in the .DPD file) are intended for HMI and / or application documentation, and not used by the Core Framework. When parsed into a DOM model, total database sizes exceeding 10 MB are possible – a considerable fraction of the memory available to a handheld radio! In addition, unless XML is going to be parsed each time on power-up (see technique 3.2), this DOM model will have to be saved to a file so that it can persist between radio sessions, further increasing storage space.

A second reason to use digested information is that it is desirable (to minimize both power-up and application launch time) that as many decisions about "what to do" should be made a install time. If a standard DOM is used without any "digest" done, processing is deferred until the decision is needed, and calculating this decision takes valuable time. For a simple example, consider the process of determining the configuration parameters that an ApplicationFactory will use to configure a newly created Application. These parameters are drawn (in precedence order) from the create() parameters, SAD, SPD implementation properties, SPD level properties and SCD properties. If this "digest" process were performed at install time, a compacted list of parameters could be produced, with all precedence conflicts resolved except the final overlay with the create() parameters. By doing this, not only could valuable processing time be saved later (when time is more critical and other running applications may be more heavily using the CPU), but also the "working" database size could be considerably reduced.

### 3.4 Don't write more XML than is necessary

The SCA Appendix D specifies XML data type descriptions that describe applications, application components, Devices and Services, as well as various core framework components. This XML is intended to support multiple clients of whom only one is the core framework. Other fields are intended for HMI, test tools and, in some cases the application components themselves, and can be deleted if these clients do not have need of the information. For example, consider once again configuration properties. A given application (e.g. SINCGARS) may have dozens of user configuration parameters, yet only the parameters that need to be initialized by the CF itself (i.e. have initial values) need to be specified in XML. Omitting the optional information saves XML development time, file storage space and helps minimize the size of the domain profile database.

### 3.5 Use threads vs. processes when possible

A typical full-up JTRS platform involves an HMI, over a dozen core framework class instances, dozens of application components, dozens of Devices as well as the ORB and underlying operating system. Providing that operating system supports the concept of separate address spaces (i.e. processes), choices need to be made on which elements share process spaces (either as shared utility classes or as separate threads with the same address space) and which are created as separate processes.

The time spent communicating between components using CORBA will vary enormously based on these decisions. Using the Objective Interface Systems ORBexpress v2.5.0 ORB running on the QNX operating system, the ratio of inter-process to intra-process times for a simple 2-way CORBA call passing a long unsigned and returning a long unsigned is over 2000 to 1! Although this measurement employed a standard TCP/IP transport instead of a shared memory transport, it is unlikely that will ever be less than 200 to 1, with inter-process CORBA calls taking in the millisecond range on typical platforms.

Considering this, when developing both core frameworks and applications, it is important to minimize the number of processes that must be traversed, while still meeting the isolation requirements mandated by the SCA security supplement [2]. When possible, the DomainManager, ApplicationFactory and any of their shared classes and domain profile information should be process co-located. Similarly, the DeviceManager, its device profile information, FileSystems and Log should be process co-located if this is practical. Since there is a reasonable amount of communication between devices and their DeviceManager, it is desirable if these too could be process co-located. For security supplement reasons each Application should be in a different process space than the core framework, but *within* an application components can be created using a ResourceFactory to achieve process co-location.

### 3.6 Consider partially linking multiple components

When one looks at the time distribution for either radio power up or application launch time in a JTRS radio, one major contributor is the time it takes to launch and load core framework, Devices or Applications components. Whereas the SCA may lead one to believe that every component must be separately loadable, this is not true for at least the core framework and its Devices. In many cases a single, partially linked shared-library or executable file can be generated that contains multiple components. True, each of these individual components will have their own individual XML files, but these files can cite a common loadable / executable file.

For an example, in a handheld radio GPP, the program code for the DomainManager, ApplicationFactory, Application, DeviceManager, Log and various file components could be linked in a common file that is loaded on power up. Note that this does not mean that the instances cannot be dynamically created or deployed (e.g. the DeviceManager creating a number of fileSystems) since loading a file that is already loaded generally has no effect beyond incrementing a reference count.

### 3.7 Minimize individual CORBA interactions

All communications between the SCA-defined Core Framework, Application and Device interfaces are intended to employ CORBA. Without "bypassing" CORBA, there are still a number of techniques to minimize the sheer number of calls that must be performed, as well as implementation details that are not required (or expected) to employ CORBA.

For a graphic example, consider a DeviceManager that has created 10 devices, two file systems and a log registering with the DomainManager. If the DomainManager needs to examine each device and its three XML files (if nothing else to see if it has changed), there could easily be over 100 CORBA calls.

To minimize this interaction several techniques can be used. First, as mentioned before, parse XML on install. For each XML file examined there is a minimum of 3 CORBA calls (open the file, read, close the file), and ten devices represent 30 XML files. Second, consider adding some non-primitive attributes (beyond the required SCA interfaces) to the DeviceManager. As an example, an attribute parallel to the DeviceManager::Devices could be devised which returned a list of Devices and their identifiers instead of the standard list of devices. Using this attribute would make it unnecessary to query the Device itself to find its identifier. As detailed in section 3.1 this technique is fully SCA compliant, as the standard interfaces are maintained.

A third method of minimizing CORBA calls is detailed in the next section.

### 3.8 Speed up local file access

The SCA-specified File interface meets its goal of providing a distributed file system, but its use comes at a great cost in efficiency. Application components must use the standard CF::File interface, since they are limited to using the POSIX subset defined in the SCA Application Environment Profile (AEP) which does not include the native OS file operations. Core framework components and Devices, however are not subject to this limitation, and hence can considerably increase file read speed when the file is located on the same processor as the CF component.

The main problem is how the CF / Device component can obtain the native OS file name it needs, given a reference to a CF::File. To do this, extend the File interface (through IDL inheritance) to add an additional operation, getNativeFilename(), with a processor ID passed as a parameter. When the client CF component gets the CF::File, it tries to narrow this object reference to the more derived type. If this succeeds, it can now use this special knowledge to obtain the native filename, if indeed the file is found to reside on the same processor (and hence reachable via standard POSIX read() operations).

Use of this technique is especially valuable for loadable and executable devices, whose implementations typically *require* native OS access so they can pass the name to the operating system. Without employing this technique, these devices would have no choice but to make a local file copy of known name, *even if the source file was on the same file system*, as there is no way specified in the SCA for determining a CF::File's native file system name.

### 3.9 Minimize use of Device capacities

Although the SCA supports the concept of Device allocation capacities, it does not require their use. Unlike larger platforms, in most cases with smaller platforms a given component will almost always be deployed on the same Device – there are no "pool" of generic processors available to load components on an "as needed" basis. In addition, higher-level mechanisms typically limit the number and types of simultaneously running applications. When these conditions are true, there is no need for the Devices in question to maintain multiple (or even any) capacities. If not capacities are supported, then the ApplicationFactory is not required to perform allocateCapacity() operations on the devices, speeding application launch.

On common misconception is that the SCA requires devices to support capacities: to quote the SCA [1],

> "The registerDevice operation shall raise the CF InvalidProfile exception when: … 2. The Device's SPD does not reference allocation properties."

Here it is important to note that there are two types of allocation properties – "normal" and "external". This requirement can be satisfied by use of a non-external allocation property, which does not involve performing an allocateCapacity() operation on the device, be instead only involves simple string comparisons within the ApplicationFactory.

### 3.10 Consider using parser-free DeviceManagers

Especially in handheld platforms, the versatility of having a fully generic DeviceManager that follows XML script to determine which Devices to launch and how to configure them is not always justified. In these cases a processor-specific DeviceManager should be considered. This component would support all of the SCA required interfaces but be "hard coded" (rather than XML driven) to launch and configure its Devices, Services and FileSystems.

Even if this technique is employed, XML files still need to exist to describe the components and their interfaces. The DomainManager would use this XML, but, as long as it is consistent with the DeviceManager's actions the DeviceManager itself does not need to parse these XML files. As a result, no XML parser would be required by the DeviceManager, potentially saving over 1 MB in code space.

## 4. CONCLUSION

Harris' experience is that the SCA is a valid specification for small platforms, but the implementation of core frameworks, devices and applications is far from straightforward. Every design decision must be with made with efficiency in mind, and one must look beyond the obvious to come up with a system that meets its requirements and user expectations.

The performance of the core framework is central to the performance of the radio as a whole, since the usable platform capacity and system responsiveness are primarily defined by this subsystem. Using the techniques outlined in this paper will help accomplish this goal, leading to not only extensible, but usable radios.

## 5. REFERENCES

[1] MSRC-5000SCA: "Software Communications Architecture", version 2.2, 17 November, 2001

[2] MSRC-5000SEC: "Security Supplement to the Software Communications Architecture Specification", version 1.1, 17 November, 2002.