Extension for component portability for Specialized Hardware Processors (SHP)

to the

JTRS Software Communication Architecture (SCA) Specification

[AKA CP 289]
JTRS-xxxx yy
V3.1x
Updated 18 June 2009

Prepared by the
Joint Tactical Radio System (JTRS) Joint Program Office

Contributions made through
Workshop on SCA Extensions
To support Signal Processing Subsystem
29-30 April 2004

Revision Summary

0.7	Draft for limited initial review.
0.8	Incorporated various review comments, completed FPGA interfaces
0.9	Final CPs and edits prior to CP submission to workshop
0.93	Addressed comments/issues based on the workshop 1/25/05. In particular, changed "agent" to "generic proxy", and removed specialized "Agent" interface, and added discussion of device-specific development process assumptions.
0.96	Addressed a variety of issues listed in a separate document (CP-289-4.doc), mostly editorial, with a few clarifications that tighten up explicit requirements.
0.97	Addressed many clarification, editorial, terminology issues, the major change is moving nearly all mention of custom proxies into appendix B.
0.99	Major update to RCC profile to accept many ideas and inputs. Add optional RCC profile enhancement for threads and blocking. Decoupling and simplifying RPL profile from RCC profile, allow worker ports as slaves. Clarifications for collocation optimizations. Additions to worker interfaces to improve alignment with CF::Resource (e.g. stat/stop).
1.0	To TAG. Minor fixes to make sure all 3 RCC examples can compile (gcc -ansi -pedantic -Wall). Some cleanup to custom proxy section. Small fix to sequence diagrams.
1.01	Update based on editorial and functional issues based on implementation experience, RCC. Mercury internal as of 1/20/09.
1.02	Final update changes based on review comments.
1.03	Minor updates based on implementation changes for 1.01 and 1.02. RPL not updated

Changes from the previous revision, other than some editorial corrections, are marked with change bars in the margins.

Table of Contents

Table of Requirements

Table of Figures

1 INTRODUCTION

This document defines a set of requirements that can be placed on certain JTRS waveform components to maximize their portability. Acquisitions may or may not require this level of portability. It addresses components that cannot use the portability requirements defined in the section 3.2 of the SCA core specification (SCACS), since that section addresses components written for feature-rich general purpose software environments that assume 1) the availability of CORBA, 2) significant amounts of the POSIX OS standard, and 3) general purpose object oriented programming languages mapped to CORBA. The software components addressed in section 3.2 of the SCACS are hereafter termed GPP components, for General Purpose Processing (that is fully CORBA-enabled).

In contrast to the discussion of GPP component requirements in section 3.2 of the SCACS, this document defines portability requirements for waveform components targeting more specialized and resource-constrained environments typically represented by DSP, FPGA and ASIC environments. Although some DSP environments are rich enough to be treated as general-purpose environments as addressed in SCACS section 3.2, this document addresses more limited and more typical resource-constrained C-programmable environments typical of DSPs. The components addressed in this section are all heretofore termed SHP components, for Specialized Hardware Processing. The scope of SHP could naturally include resource-constrained processors such as microcontrollers, and RTL-programmable logic processors not fully captured by "FPGA" or "ASIC" terminology.

Thus the purpose of this document is to enable portability requirements to be placed on such environments in a way consistent with, and analogous to, the portability requirements that SCACS section 3.2 defines. This increases the portability of waveforms in general since it increases the functionality and source code that can be subjected to portability mandates in procurements. A related purpose of this document is to facilitate the replacement or migration of components in waveforms among the different types of processing environments without affecting the waveform as a whole, or other components in the waveform. This reduces the effort of technology insertion and migration when deployment requirements change or new technology emerges: it enables multiple compatible, replaceable implementations of the same component functionality, targeting either GPPs or SHPs.

While addressing SHP component portability in general, this document addresses resource-constrained C-programmable (RCC) components (e.g. simple DSP) separately from RTL-programmable logic (RPL) components (e.g. FPGAs and ASICs) due to differences in their common source languages (C, VHDL/Verilog etc.), operating environments and development processes. For brevity and technology independence, the term RPL is hereafter used to refer to include FPGA and ASIC components. There is significant commonality in concept and architecture between the treatment of RCC and RPL components, and many requirements termed SHP requirements apply equally to both. SHP = RCC + RPL. This new terminology is intended to emphasize that "DSP", "FPGA", and "ASIC" are current technologies, whereas the categories represented by "RCC" and "RPL" are more general and will likely include new technologies not currently used in software-defined radios.

This document purposely does not define any interoperability facilities like CORBA's GIOP. Providing interoperable connections between processing devices is considered the integrator's

responsibility. The reason for this is to enable preexisting hardware and software interprocessor messaging mechanisms to be leveraged while complying with this specification.

Readers of this document are assumed to be familiar with the core SCA specification, CORBA, IDL, and how GPP component-based SCA applications are deployed using the SCA Core Framework to execute components on devices.

1.1 SCOPE.

This document specifies requirements on the source code, metadata and operating environments to achieve SHP component portability, and cross-processing-type replace-ability. In the present version it also contains an appendix of change proposals (enhancement recommendations) for other parts of the SCA specification that are required to meet the goals of this document.

This document identifies several classes of SHP components and defines separate compliance points for each, including compliance points for component developers as well as compliance points for suppliers of operating environments for those components. It also includes the definition of a compliance point for environments that integrate multiple classes of processing (multiple GPP and/or SHP OEs).

1.2 COMPLIANCE

The specifications in this document are based on the core SCA version 2.2.1, with its compliance statements intact.

This document *defines* 5 new compliance points:

1. Requirements for SCA RPL component authors
2. Requirements for SCA RCC component authors (not written as GPP)
3. Requirements for SCA RPL device/container suppliers
4. Requirements for SCA RCC device/container suppliers
5. Requirements for platform integrators including devices for SHP (RCC and RPL) components.

Any of these compliance points may be referenced by acquisition documents.

1.2.1 RPL Component Compliance

This compliance point requires that component authors meet requirements x,y,z;

1.2.2 RCC Component Compliance

This compliance point requires that component authors meet requirements x,y,z;

1.2.3 RPL Device Compliance

This compliance point requires that component authors meet requirements x,y,z;

1.2.4 RCC Device Compliance

This compliance point requires that component authors meet requirements x,y,z;

1.2.5 Mixed Device platform compliance

This compliance point requires that component authors meet requirements x,y,z;

1.3 DOCUMENT CONVENTIONS, TERMINOLOGY, AND DEFINITIONS

1.3.1 Conventions and Terminology

This document uses the conventions listed in section 1.3 of the core SCA specification.

1.3.2 Definitions.

Definitions beyond those defined in Appendix A of the core SCA specification are included in Appendix A to this document.

1.4 DOCUMENT CONTENT

Section 2 of this document provides an overview of the problem space and purpose of this document, and introduces all the concepts used in the following sections. Section 3 elaborates the architectural refinements to fully explain the basis for the requirements in later sections. Sections 4-6 contain normative requirements for component authors and container implementers, broken in to sections for all SHP components(4), RCC components(5) and RPL components(6). Section 7 contains normative requirements for platforms including GPP and SHP operating environments.

1.5 APPLICABLE DOCUMENTS

The following documents are applicable to this document either by direct reference or as foundation for the architecture definition.

1.5.1 Government Documents.

SCA Communications Architecture Specification

1.5.2 Commercial Documents.

C Standard: Programming languages – C, ISO/IEC 9899:1990.

POSIX: http://www.unix.org/version3/ieee_std.html

IDL: http://www.omg.org/spec/CORBAe/1.0/PDF, Chapter 6

CORBA: http://www.omg.org/technology/documents/corba_spec_catalog.htm

OCP 2.1: http://www.ocpip.org/socket/ocpspec/

CORBA/E: http://www.omg.org/spec/CORBAe/1.0/PDF

CDR: http://www.omg.org/spec/CORBAe/1.0/PDF, Chapter 17

GIOP: http://www.omg.org/spec/CORBAe/1.0/PDF, Chapter 18

1.5.3 Normative references

1.5.4 Other references

2 OVERVIEW

This section introduces the concepts and goals to provide background and context for the architectural refinements in section 3, and the specific requirements specified in sections 4, 5 and 6.

The subsection below defines the key terms used in this document, and states the goals of the specification.

2.1 KEY TERMS AND CONCEPTS.

To enable even the following goals section, some terms are clarified for use in this document:

2.1.1 ***Component***

The SCACS uses, but does not define, "application software component", "software application component", "functional components of applications". It extols "the broadest reusability and portability of components". However, it uses the term "component" in the general sense of any part of the application, infrastructure, environment, or hardware system. The term "resource" is used as the general term for infrastructure or application software modules (i.e. a source of functionality rather than a source of supply or capacity). This SHP portability specification uses the term "component" more narrowly, as an *independently defined unit of functionality of an application or waveform*, separate from the operating environment or any other infrastructure elements. This is consistent with, but more specific than, the SCACS's use of the term. "Component" applies here equally to units of functionality implemented for ASICs, FPGAs, DSPs or GPPs.

2.1.2 ***Application***

The SCACS uses this term very generally, but this specification uses the term narrowly to designate a configured and interconnected set of components as indicated by the SCA Assembly, but not including any infrastructure (CF, devices or services).

2.1.3 ***Class***

This term is used to indicate which general category of processing device, container, or component: GPP, RCC, or RPL. The SCACS describes portability and operating environment requirements for the GPP class of components and operating environments. This document extends portability to SHP (RCC and RPL) component source code, and thus discusses the RCC and RPL classes, collectively termed the SHP class (vs. the GPP class).

2.1.4 ***Container***

A container is the immediate runtime environment in which a component instance executes. Containers provide to component instances the execution environment, and a standardized set of local API-based services. This definition is as used in literature for most component software systems, including the CORBA component model on which the SCA was partially and loosely based. The SCACS has no such term, but covers the concepts by referring to "executable devices" and "resource factories" as places in which software components may be instantiated and run. The container concept is narrower and more precise: it refers to the immediately surrounding execution environment that 1) invokes and controls the component and 2) provides the local services and APIs the component invokes during execution. "Containment" means that the component only interacts directly with the container, and interacts with external entities only

indirectly, via its container. Containers are sometimes processes, sometimes threads, and sometimes use a pool or group of threads. These are all implementation variations that support the container concept. Containment is logical, not physical, meaning that it may be "optimized away" in cases when multiple component implementations are located in the same container. Thus containment does not always imply any overhead. There is no particular relationship between thread, process and container.

In summary, applications consisting of components are executed by instantiating the components for execution in appropriate containers executing on processing devices (GPP or SHP). Component source code is *authored* for a given *class* of container and processing device. Component source code is *compiled and linked* for a specific container implementation (e.g. for a specific operating system, compiler, ORB).

The SCA concept of a device is hardware, and the SCA concept of a Logical Device is a software proxy for hardware. Containers exist (execute) on the hardware processing device, and thus have a “runs on” relationship with hardware devices, and an “is controlled by” relationship with Logical Devices. There can be multiple containers per SCA device, since components deployed on devices might require different execution environment attributes (e.g. security or priority), which would require them to run in different containers on the same processing device. The Logical Device proxy for a processing device must be a GPP software infrastructure module (as opposed to application component) that executes on some processing device that supports GPP software. A GPP processing device may host its own Logical Device proxy.

2.1.5 ***Platform***

A platform is a collection of (GPP and SHP) processing devices and associated containers that together support the execution of component-based applications. In particular, a set of containers that are enabled to communicate with each other such that components running in different containers can communicate with each other without knowledge of the type or class of container housing the “other component”. An SCA core framework DomainManager manages the platform. The supplier and integrator of a platform must assemble, install and configure the collection of processors, LogicalDevices and associated containers such that they support interoperation of components. The components (source code, compiled code, or metadata) have no knowledge of the class of the container or component at the “other end” of intercomponent communication.

The diagram below depicts the simple hierarchy of execution environment relationships without showing the control infrastructure (core framework elements) or non-programmable devices. The components shown are, strictly speaking, instances of component implementations for components indicated in the application’s assembly descriptor (SAD).

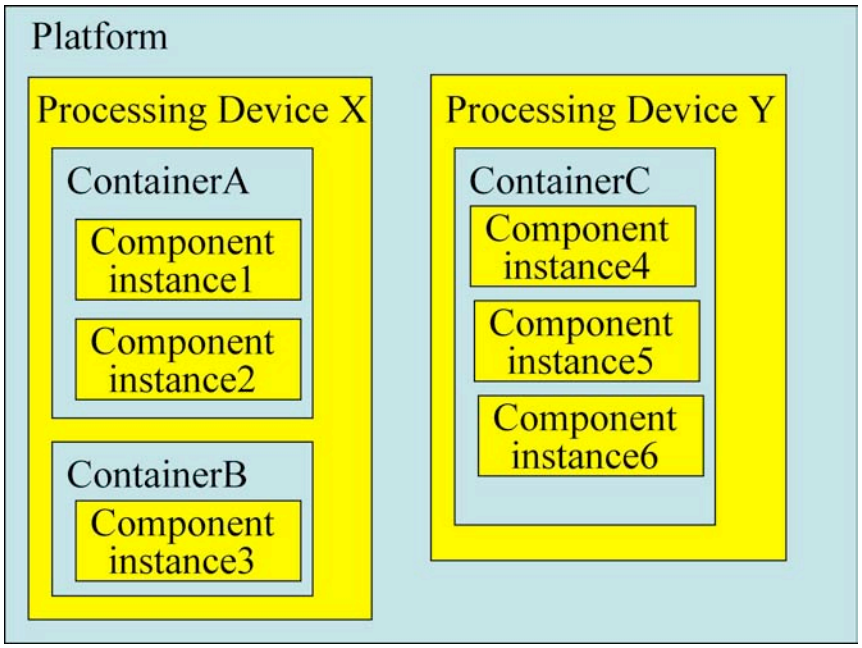

Figure 1 General Architecture of Platform

2.2 GOALS OF THIS SPECIFICATION

This section lists the goals that this specification is intended to achieve. Later sections of this document are written to achieve the goals listed here. These goals are "the requirements of the requirements". Thus the specific normative sections should be seen to meet these goals and can be evaluated against these goals.

This extension causes the portability of non-GPP component implementations to be significantly enhanced. As always, deployed waveform components for real platforms can have specially optimized non-portable component implementations that exploit specific features unique to the containers. However, having a portable implementation appropriate to the class of container (RCC, RPL) will provide a good starting point and reference implementation to significantly reduce porting effort and cost.

Some programmable devices could still be outside the scope of this specification. The term SHP is not meant to apply to them. Also, components whose role is to interface to specific hardware and I/O interfaces attached to certain programmable devices will not be portable in any case. Portability is never absolute. In the SCACS, such components do not have a specific term distinguishing them from programmable processing devices, but in general they are devices without any components "deployed on them".

The specific goals are listed here:

2.2.1 Portability

This goal is to make the source code of waveform components targeting SHP environments to be similar to the level of portability achieved in the core specification for GPP components. Portability is within the same class — "like-for-like".

2.2.2 Consistency

SHP components should be treated as consistently as possible with GPP components, to simplify learning curves, CF implementations, tools, documentation, metadata, and technology migration. In particular, multiple SHP components may be deployed on an SHP device the same way that multiple GPP components may be deployed on a GPP device. GPP and SHP devices are shared by multiple components possibly from multiple applications.

2.2.3 Replace-ability

Different component implementations, *for different classes*, for the same component interface and functionality, should be possible as alternative implementations under the **implementation** element of the Software Package Descriptor (SPD), so that the application may be deployed using any class of processor supported by component implementations. The class of processor for any given component is not "hardwired" in the Software Assembly Descriptor or SCD or SPD. This allows one deployable waveform, as defined by one SAD, to target different scenarios of hardware presence and availability, or to be used in radios with different hardware configurations, further increasing reusability. Replace-ability implies transparency with respect to other components of the waveform.

2.2.4 Separation of concerns

Component functionality, and the role a component plays in an application, should be a separate concern from how the component is implemented; including for which class of processor the component implementation is created.

Component implementation authorship should be concerned with the class of container being targeted and not the class of other containers that house other components that may communicate with it. Implementing portable components for SHP devices should be an independent activity from implementing SHP logical devices or SHP containers.

2.2.5 Resource efficiency and performance

Since SHP execution environments are generally small, lean, and resource-constrained, the other goals should not compromise the typical resource efficiency of SHPs. The "portability tax" should not be more than necessary. When components communicate within or between SHP containers, the latency and throughput characteristics should not necessarily be significantly different than using non-portable approaches.

2.2.6 Minimal Impact to the SCACS

The above goals are achieved with design choices that minimize the necessity to change the SCA Core Specification. The minimized change proposals required are in Appendix A (Section 9).

3 ARCHITECTURE REFINEMENT

This section describes architectural refinements and concepts that provide a foundation for the statement of specific requirements in sections 4-7. The refinements build on the architecture as defined in the SCACS.

The terminology introduced in section 2.1 above described the basic composition of an application (components in the application's assembly) and the hierarchy of execution environments for components (container, processing device, platform). That discussion ignores the functional infrastructure elements that both manage the execution environments and deploy applications into those environments. This is of course the SCA Core Framework. The Core Framework manages the processing devices via Logical Device proxies for those hardware devices.

This section contains restatements of SCACS behavior to help readers understand how this specification builds on and relates to that specification. This material may be removed when this specification is directly attached to the SCACS.

3.1 CONTAINER BEHAVIOR

3.1.1 SCA Core Container Behavior

The Logical Device proxy for a device manages any containers that execute on that device. The SCA CF uses LogicalDevices and does not care about containers. In the simplest GPP case, the Logical Device simply runs executable files as processes of the device's operating system. The component source code is compiled and linked with the CORBA ORB and POSIX libraries into an executable file. In this case the "container" is simply the "main program" that executes in the process. Thus the component and its container are bound together statically at link time. The component interacts with the container (it's immediate environment) via CORBA calls and POSIX library calls.

A more dynamic scenario would be components compiled and linked as dynamic libraries (e.g. DLLs on Windows, shared objects on Unix). In this case the container may be a preexisting process into which the component binaries are dynamically loaded, which would enable multiple components to execute in a single process, probably using multiple threads. In this dynamic case the LogicalDevice would (privately) communicate with the container to load and instantiate components. It is possible that the LogicalDevice object executes in a process that acts as a container for the same device (to conserve resources). Containers may also be created dynamically as they are needed for components being deployed. Containers are not visible to the SCA Core Framework since the Logical Device manages them privately.

Whenever a component communicates with another component via one of its ports, that action is intercepted by the container and routed over the appropriate communication path to reach the connected component, (as indicated in the assembly). When that other component is in the same container (i.e. is *collocated*), then the container can arrange for that communication to be very efficient. When components are collocated, their execution is managed and scheduled by the container, in conjunction with the underlying scheduling capabilities of the local OS. Similarly, containers executing on the same processing device may collaborate to make intercontainer communication more efficient than the general case of containers on different processing devices.

Much of the work of a GPP container is carried out by the CORBA ORB. The "interception" of intercomponent communication is carried out by the CORBA IDL-generated "stubs", and the "intercontainer routing" is done by the ORBs transport system (e.g. GIOP/TCP/IIOP between processors, or possible shared memory between processes). However, CORBA ORBs do not do all the work necessary to act as a true container for components. There must be a "main program". There must be code to load the component binaries (in the DLL case), and there must be code to instantiate components. There must be libraries to supply the local OS services (i.e. POSIX AEP for GPPs). All this together comprises the container.

In general the communication between a LogicalDevice and containers for that device are not, and do not need to be, standardized since there are many implementation scenarios for them.

3.1.2 SHP Container Behavior

Since functionality can be flexibly divided between the LogicalDevice and the container, there is no standard interface between them for any class. Logically, the SHP container performs functions analogous to GPP containers. The main difference is in the local (inside the container) interfaces between components and containers. These interfaces are defined below for SHP containers. They do not use CORBA, and they are more constrained and simplified than those for GPP components. They are specialized differently for both RCC and RPL classes.

This SCA extension does not mandate any intercontainer message format, but only mandates that the containers in a platform cooperate to deliver and accept messages according to the local communication interfaces defined for that class of container. Messages created in one container (based on an API between component and container) must be delivered properly formatted to a component in any other container.

Note that the HAL-C proposal could enhance portability of container and logical device software since it proposes a common socket-like API for inter-device messaging. The portability of such software is out of scope of this extension and does not address interoperability of components between containers.

3.2 SHP COMPONENT IMPLEMENTATIONS

Components are defined (in the SCD) to have a 1) *specific* (control) interface, 2) *port* (intercomponent) interfaces, and 3) use local AEP services.

The term "worker" will be used for the *instantiated SHP component implementation* that runs directly on the SHP devices. Since workers do not use CORBA, and SCA application components for GPPs *are* required to use CORBA, SHP components are implemented in two parts, with an optional third:

1. The *worker*, a functional component instantiation executing in the SHP container on the SHP device. Multiple workers can execute in the same container, even from different applications (as can GPP/CORBA component instantiations).
2. The *generic proxy*, a specialization of the SCA concept of an "adapter", is an object created locally by the LogicalDevice *for each worker* executing on the corresponding physical device. It acts as an insulator between custom proxy and worker, and also enables the portability of the custom proxy (see below). The generic proxy specifically implements the *CF::Resource* interface. It is termed *generic* because its implementation is not specific to any component, custom proxy or worker. It is

written by the author of the logical device software, not by the waveform component author.

3. An optional *custom proxy*, executing in some GPP container (on a GPP device), implements the component's *specific* interface (as indicated in the SAD, which must support *CF::Resource),* when the generic CF::Resource behavior of the generic proxy is somehow insufficient to reference directly from the SAD. This enables broader replace-ability.

These three elements, interact to simultaneously support the existing SCA CF model of application components and the goals specified above for SHP components. The custom proxy is a notional rather than a required element. How it may be used is discussed in detail in Appendix B

3.2.1 SHP Workers

SHP component implementations are the functional codes written in the source language of the SHP class. The binary/compiled code executes directly on the SHP device inside the SHP container. Its control interface (controlled via the generic proxy's implementation of the *CF::Resource* interface) uses a fixed API (with configuration properties), specific to the SHP class, which is a rough functional subset of the Resource interface (start, stop, configure etc.). The executing instantiation of the SHP component implementation is the worker.

Worker ports operate according to interfaces defined by a pure IDL subset as indicated in the SCD, and use a fixed container API to send and receive interoperable messages whose structure is derived from the IDL. The worker knows nothing about any communication mechanisms used on the platform or between any containers, or even inside a container. It only knows how to interact locally and directly with its container. It does not need or use CORBA.

When multiple workers are collocated in the same container, the container may arrange for them to communicate directly with each other, as an optimization. This does not mean that the workers are aware of this, but that the container may optimize the local connectivity to minimize any overhead. In some technologies, a set of collocated, interconnected workers may be statically bound together in an optimized configuration, while still preserving the portability of their source code.

3.2.2 SHP Generic Proxy

The generic proxy is a specialization of the SCA adapter concept. The term “generic proxy” is used here to avoid confusion with LogicalDevices or SHP custom proxies (described in Appendix B). SHP generic proxies are CORBA objects that are created internally by the SHP LogicalDevice per "execute" operation (thus per worker). This is analogous to how an CF::Application object is created internally by the ApplicationFactory per *create* operation, and how the ApplicationFactory object is created internally by the DomainManager per *install* operation. When the SHP LogicalDevice is asked to instantiate an SHP component, using the *load* and *execute* operations, it creates this local CORBA object to represent that instantiation in the SHP container on the SHP device. This generic proxy object has a fixed interface (*CF::Resource*) that is not specialized by the definition of the SHP component (i.e. it's implementation does not depend on the component’s SCD or IDL).

The generic proxy is responsible for establishing connections with its worker. Since each worker running on the SHP device has its own ports, the per-worker generic proxy implements the

CF::PortSupplier interface, creates CF::Port objects, and implements the *connectPort* operations for its worker. Since there may be multiple workers based on a single implementation, or multiple workers with common port names may be running on an SHP device, the generic proxy also provides the appropriate scope for each worker's port connectivity.

The generic proxy object has special knowledge and capability to communicate with the SHP container for the purpose of loading, creating, controlling and configuring the workers. It implements a fixed CORBA interface, *CF::Resource*. Having the LogicalDevice create this object locally means that the specialized knowledge and state management of the SHP device and containers is localized within the LogicalDevice implementation. This is analogous to the localized relationships between DomainManager and ApplicationFactory.

Whereas a GPP LogicalDevice's *execute* operation results in a CORBA object reference to the GPP component implementation itself, the SHP LogicalDevice's *execute* operation results in a CORBA object reference to a generic proxy. Thus GPP implementations, and SHP implementations, can coexist in the same SPD, transparent to the SAD.

3.2.3 SHP Component Implementation Summary

The *generic proxies* are typically (but not required to be) process-collocated with the associated LogicalDevice. They hide the mechanisms used to control containers and control workers. An example of this is where the generic proxy might know how to issue a single memory store instruction to a special mapped bus address to change a configuration parameter of a worker. The generic proxy is also responsible for connecting the ports that are implemented by its worker, by implementing the CF::PortSupplier interface.

The *worker* is logically isolated by the container, but is controlled by the *generic proxy*, which performs all control/configuration operations via the container. The container may provide short cuts to allow more direct contact between generic proxy and worker, but this is an implementation detail private to a given LogicalDevice, generic proxy and container implementation to support a specific SHP device. The figures below show the case of an SHP component implementation. The first indicates run time relationships, and the second, the sequence diagram showing component instantiation and connection.

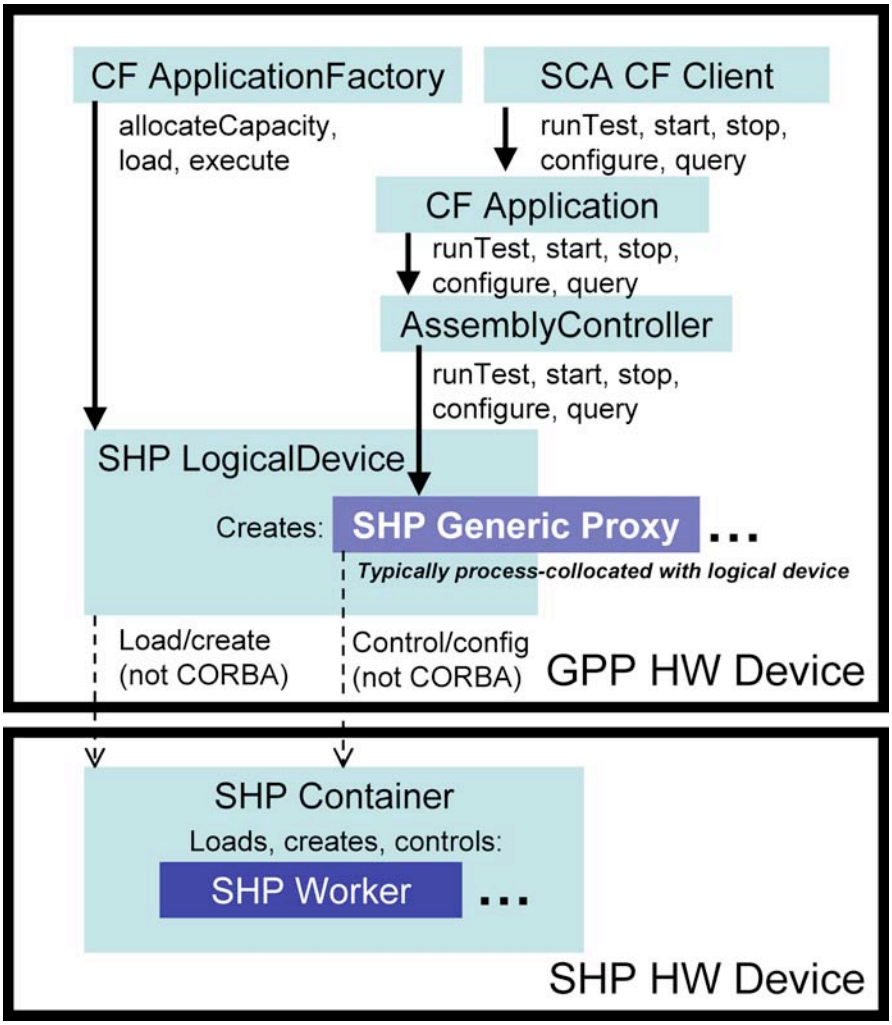

Figure 2 SHP Component Implementation Relationships

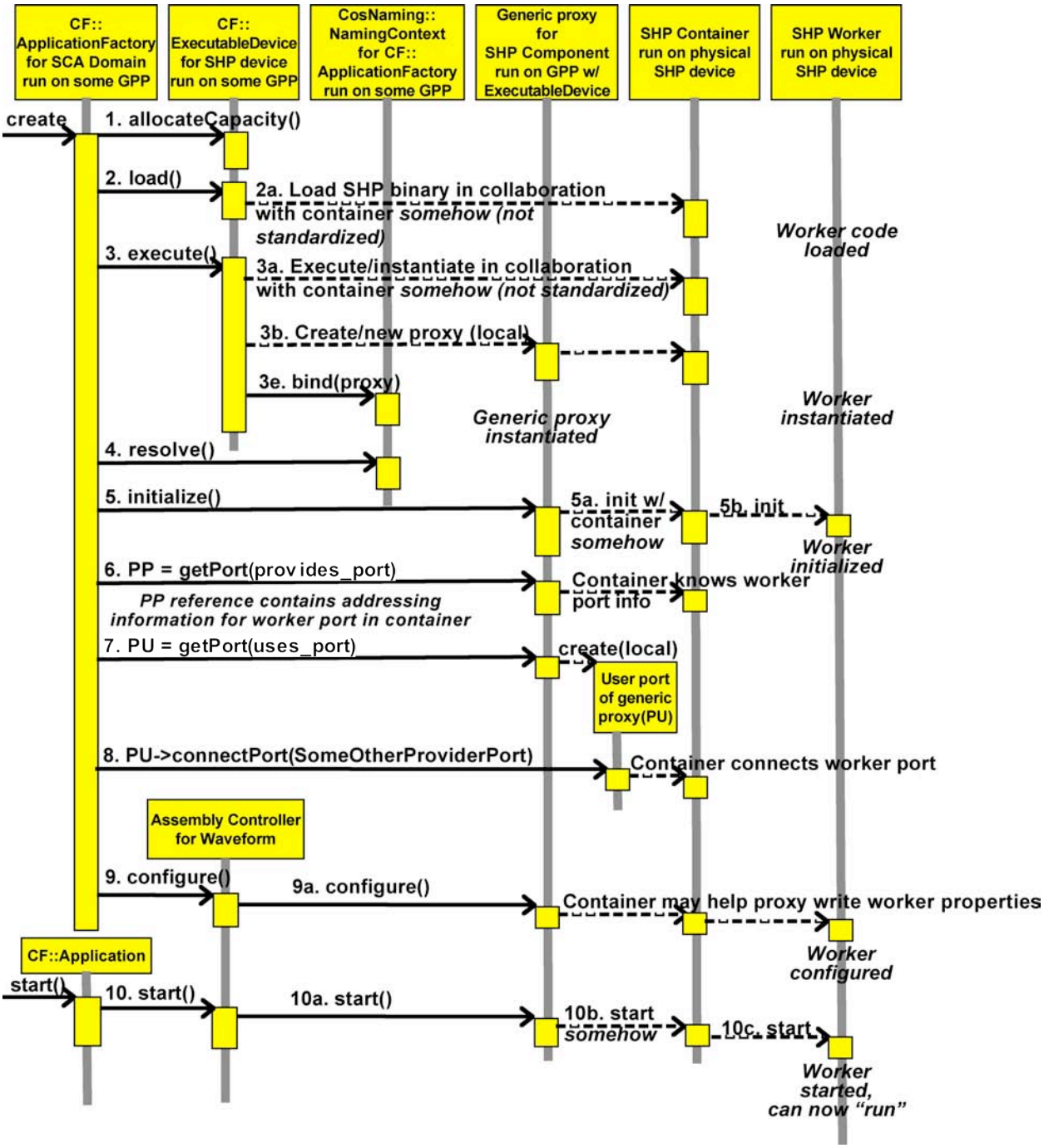

Figure 3 SHP Component Implementation Sequence Diagram

3.2.4 Metadata and Packaging of SHP Component Implementations

Each component in an application described by a Software Assembly Descriptor (SAD) is described by the SPD referenced in the SAD. Each of these SPDs refers to an SCD that describes the interfaces offered by all implementations described in that SPD. The figure below shows the assembly with an SHP implementation.

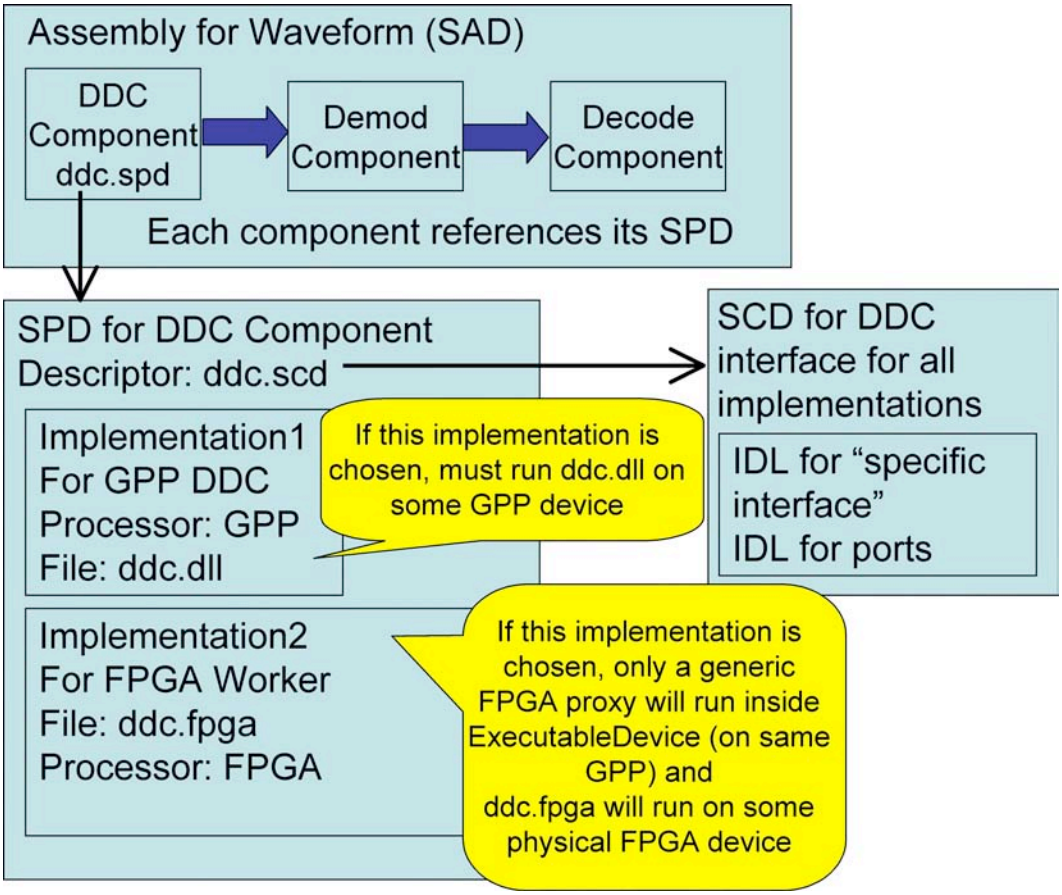

Figure 4 Assembly with DDC Component with SHP implementation

3.3 SHP COMPONENT INTERFACES

3.3.1 SCA Core Interface Model

SCACS section 3.2.2 states that each component's interface is described in a Software Component Descriptor (SCD) file (which references IDL-defined interfaces). SCA Appendix D (Domain Profile) section 4.1.2 describes how a Software Package Descriptor (SPD), when describing a package of implementations of an SCA Component, contains a reference to one SCD file. The SPD describes multiple implementations of the component all of which implement the component interface as described in the single referenced SCD file.

In the SCA, each component has a single "specific" interface, which inherits from the *CF::Resource* interface, and may additionally inherit from one or more "supported" interfaces. This specific interface is the basis for the CORBA object reference for the component as a whole. It is used to initialize, control, configure and connect the component in a way that is generic; all components are controllable and configurable in the same way using the same *CF::Resource* interface. When component designers require more specialized control interfaces beyond what is provided by the *CF::Resource* interface, those customized interfaces still inherit, and all implementations support, the basic *CF::Resource* interface. The CF ApplicationFactory, CF Application and assembly controllers, are the primary clients of this interface.

In addition to the singular “specific” interface, a component can have ports, which allow for intercomponent communication. Each port is also defined by an interface, either as a “user” (client) of that interface or a “provider” (server) of that interface. Thus a component’s interfaces consist of its “specific” interface, and its port interfaces. The specific interface may be thought of as a special “provider” port.

The component’s application interface specification (in IDL and SCD) are used to generate appropriate API-specific interface artifacts that act as “gaskets” between component code and container code. For GPP components these “gaskets” are the stubs, skeletons, and interface header files generated by a CORBA IDL compiler.

IDL implies that messages are sent from client to server for requests, and (for two way operations) messages are sent from server to client for responses. One precise way these messages can be formatted and exchanged is specified in the GIOP message level protocol (General Inter-ORB Protocol). Containers must exchange messages somehow to convey requests and responses. CORBA ORBs may use any protocol, but are required to at least support GIOP over TCP/IP (together called IIOP).

Finally, containers provide a set of local services defined by the SCA Application Environment Profile. This is a set of local language interfaces (APIs) used by the component. For GPP components it is based on a POXIS subset. Component implementation code *provides* the specific interface and provider port interfaces. It *uses* user port interfaces and local AEP services. GPP component implementations are also allowed to use a set of CORBA-based services (file, name, log, event).

3.3.2 SHP Component Interfaces

3.3.2.1 The Component's Specific Interface

This SCA extension defines how SHP implementations can coexist with GPP implementations in the same SPD. This implies that SCDs also apply to SHP component implementations. The SHP generic proxy acts directly as the component, the specific interface of the SHP component is simply *CF::Resource*. Appendix B describes "custom proxies", which can implement any specific interface (which inherits CF::Resource), and any ports, and can act as an adapter for the underlying SHP implementation, using the generic proxy. Custom proxies are optional and only needed in certain cases.

3.3.2.2 Port Interfaces

Port interfaces of components with SHP implementations are restricted to a subset of IDL. This supports the replace-ability goal while also attending to the efficiency and performance goal: intercomponent communications do not have to be indirectly handled with any adapters or proxies. Component port interfaces that are not restricted to the SCD/IDL subset can only have GPP implementations.

The key refinements to support SHP component port interfaces are:

- Defining a subset of the IDL and component definition (SCD) languages that can be supported across all classes of containers (a subset of the existing SCA specification for GPP components). This subset is required for components with SHP implementations and can be used for equivalent GPP implementations.

- Defining the mapping from that subset definition into the specific languages of SHP components, in an appropriately simple, functional and usable way that captures the variability of interfaces without making such interfaces much more complex than typical inter-component interfaces in the SHP domain.

For RPL (e.g. FPGA/ASIC) interfaces, the Open Core Protocol (OCP) provides the basis for interface definition since it provides both bus/technology/language independence as well as a richness that is optional and is easily "compiled out" when not used by either side of a communication. It can be used in VHDL, Verilog, Systemic, etc. It is open and does not require implementation license fees. OCP by itself is not sufficient, but some modest messaging semantics overlaid on OCP result in an appropriate solution. See the section 5 below for how the IDL subset maps to a set of OCP-defined interfaces. Thus the IDL-defined interfaces map to OCP-defined interfaces and message formats. Then the standard well-defined VHDL/Verilog mapping from OCP is used.

The C-based RCC (e.g. DSP) interfaces are derived from this RPL-centered approach since this keeps them minimal and does not introduce unnecessary diversity in the profiles for these two classes of SHP implementation. OCP is not used for RCC interfaces, although RCC workers can communicate with RPL workers that do use OCP interfaces. Some DSP environments are rich enough to support the existing full GPP specification for CORBA+POSIX-based components, and are not applicable here. The RCC profile defined here does not try to "interpolate" in using parts of CORBA, but keeps the model as simple and similar to the RPL model as possible. On the other hand, the RCC-related specifications defined here, in the C language, are easily implemented in any GPP environment.

For RCC components, these interfaces are defined by the SHP IDL subset map to C-based "ports", as well as C-based message data structures.

Thus the OCP interface for RPL worker implementations and the C structure interfaces are both derived from the IDL-subset used for port interfaces.

3.3.2.3 Control Interfaces

For SHP component implementations we define a fixed subsidiary control/configuration/management interface that does not depend on the component's own IDL. It is a functional subset of the full functionality of CF::Resource, since the full functionality of CF::Resource is implemented partly by the generic proxy (and container) to simplify the worker. It is implemented by the worker itself, to enable the container to control it.

The *CF::Resource* interface supported by the generic proxy, is also a fixed interface not specialized for the component. It hides and abstracts any mechanism used by the LogicalDevice software to actually control and communicate with the physical device, container and the worker.

The *Worker* is the interface used by the container to control the worker, and provided by the worker implementation. It has a language mapping for each SHP class: one in OCP for RPL worker implementations, and one in C for RCC component implementations. The author of SHP functionality (RCC or RPL component implementation developers) are only concerned with this control interface, and do not need to deal with the *CF::Resource* interface, which is implemented by the LogicalDevice software in the form of the generic proxy.

The generic proxy represents the component to the CF and to the Assembly Controller component of the waveform. The generic proxy talks privately to the container, which talks to

the worker locally using the *Worker* interface. The generic proxy may have direct (e.g. memory mapped) access to the worker, but that is invisible to the ApplicationFactory, assembly controller and the worker. The functionality of the Worker interface is logically a simplified subset of that of the *CF::Resource* interface. Configuration and query properties are mapped to indexed data structures. Worker configuration is reduced to accessing a data structure.

The *Worker* interface is discussed in more detail in the Execution Model section below. The details of the two language mappings of the Worker interface are defined in sections 5 and 6.

3.3.2.4 Local services interfaces

In addition to implementing its *specific* interface, and utilizing its *port* interfaces to communicate with other components, a component can use a standard set of services provided by its container. SHP containers provide a limited set of local services defined below.

RCC containers provide a (much) simpler subset of the POSIX AEP, defined in section 5 below.

RPL containers provide a very simple set of services, such as clocking and memory access. These are defined in section 6.

3.3.2.5 SHP Interface summary

The CF::Resource is implemented by the generic proxy and is used directly by the ApplicationFactory and assembly controller.

The SCD/IDL-defined *port* interfaces, used by the *worker* and the *container*

The *Worker* interface is provided by the SHP implementation, and used by the container, and indirectly, by the *generic proxy*.

The local services interface is implemented by the container and used by the worker.

These interfaces are summarized in figures 2 and 3.

3.4 EXECUTION MODEL FOR SHP COMPONENTS

How a container executes a component in a GPP container is defined by the fact that inbound CORBA requests to *provider* ports of the component are dispatched in the context of a thread of execution provided by the ORB. A component can also create its own threads, since it has access to the pthread API in the POSIX AEP. Thus operations performed on the *uses* ports may be executed either in the context of an ORB-provided thread or a component-created thread.

Workers must have a simpler execution model that is appropriate to their leaner, resource-constrained and performance-sensitive environments. The execution model generally summarized below is based on a proven set of tradeoffs that keeps worker implementations simple, and allows containers great flexibility in providing for worker execution, inter-worker communication, and inter-container communication. This flexibility allows maximum use of existing or legacy communication and connectivity infrastructures. The model is similar for RCC and RPL workers, although there are differences in the details described in the specific requirements for each class.

The worker execution models have these attributes:

- Worker ports are unidirectional. They consume messages or produce messages, not both.
- Two-way “uses” or “provides” ports (SCD component ports with interfaces with two-way IDL operations) map into a pair of worker ports: the first for input (requests for “provides”

ports, responses for “uses” ports), the second for output (responses for “provides” ports, requests for “uses” ports).
Reason: simplifies how two-way operations are supported, lets workers simply produce one message (response) corresponding to a request, keeping container simple and unaware of request and response matching
Reason: enables maximum concurrency in RPL workers

- During execution, the worker accesses input or output messages by accessing the “current buffer” at a given port.
- During execution, the worker indicates which ports should "advance"
  Advancing an input port means a new message buffer should be made available/current in order for the port to become ready, and implies that the current buffer can be released and become unavailable to the worker.
  Advancing an output port means the current message buffer is ready to be used by any downstream connected input ports, with a supplied length, and that a new output buffer (to fill) is required to make the port ready.
  Reason: simplest way to do simple things: default execution of worker processes all inputs (read from current input buffers) to produce all outputs (write to current output buffers).
- Worker accesses the current buffers on all “ready” ports, including the command and length of the message in the buffer.
  “Command” is operation ordinal from IDL interface for requests.
  “Command” is error indicator (or zero) for responses (error is exception ordinal starting with 1)
- Multi-buffering, circular buffering, or FIFO buffering, is transparent to worker code and can be configured in the container, possibly based on QoS or rate matching.
  Reason: delegate buffer management approach to container, no mandates on buffering.
  Reason: allow zero copy buffer management, transport-specific memory management
- All worker types have *initialize*, *start*, *stop*, *test*, and *release* operations directly analogous to the similar operations in the CF::Resource interface. Configuration and query are similar but based on reading and writing a configuration property space whose layout is derived from the property file (PRF) of the SPD containing the worker implementation.

The execution model above requires that the container assess which workers have run conditions that are satisfied, and enable their execution. In RPL environments, workers can run concurrently. In a (single core) RCC/DSP environment, workers must be run sequentially. In RCC environments, the container may decide to run workers inside different threads (using priorities or time slicing) but may also run workers in a single thread. It is common for RCC environments to be configurable as single-stack/single-thread or multi-stack/multithread. This execution model supports both.

When workers indicate the consumption of buffers on input ports, the container can recycle the buffer. When workers indicate the filling of buffers on output ports, the container can start the transmission of that buffer/message on to its destination. When two workers are collocated, the buffer at an output port may be directly used as a buffer at the downstream worker's input port. Workers are assumed to not modify their input buffers unless they specify that they do.

The actual details and interfaces of this execution model are specified in sections 5 (for RCC) and 6 (for RPL) below.

3.5 CONNECTIVITY

Communication paths among components are indicated in the assembly (SAD). Communication between ports of components should normally be direct, rather than via any proxies or adapters. SHP components cannot afford the overhead of communicating with each other via GPP proxies.

3.5.1 Establishing connections

Connectivity of SHP components is accomplished normally by the CF ApplicationFactory as follows:

1. The CF ApplicationFactory performs the *getPort* operation on the component with the "uses" port, obtaining the CORBA object reference. For SHP components this is a reference returned by the generic proxy, possibly in collaboration with the container (for addressing information).
2. The CF ApplicationFactory performs the getPort operation on the component with the "provides" port, and obtains the "provides" port's CORBA object reference. For SHP components this is a reference returned by the generic proxy that contains transport addressing information for the actual worker's port. It is the logical device and container software that understands the details of the particular transport hardware, addressing and protocols.
3. The CF ApplicationFactory performs the connectPort operation on the "uses" port reference, providing the "provides" port reference. This action conveys detailed transport addressing information to the container holding the "uses" port. For SHP implementations, the connectPort is implemented in the LogicalDevice software (generic proxy), which conveys the transport addressing details to the container.
4. The container with the "uses" port performs the appropriate communication setup for the actual source worker to communicate with the "provides" port.

The architectural refinement here for SHP components is that the generic proxy has the responsibility to convey the connectivity information to the container. How this is accomplished is not in the scope of this document, and does not need to be standardized to meet the goals of this specification. The supplier of the processing device must supply the LogicalDevice, generic proxy and container for that device.

No standardization is required for transport addressing since the platform supplier must ensure interoperation among containers and ORB transports. There is no standardization for addressing in CORBA references either, unless TCP/IP is the protocol. This is consistent with the current SCA requirement that CORBA message communication, and IOR addressing, between different processors and ORBs is not standardized. The only requirement is that the processing environments assembled into the platform can intercommunicate.

This sequence is included in Figure 3.

3.5.2 Container interoperability

Containers are required to have a common understanding of transport addressing and protocol. For GPP containers, this is the CORBA object addressing information, which requires either an

IIOP path between containers (using GIOP and TCP/IP) or ORBs that have a common understanding of some non-standard transport protocols (e.g. GIOP over a PCI bus messaging system).

Similarly, for communication between SHP containers, there must also be a commonly supported transport protocol over some hardware path, but this does not have any particular requirements other than the reliable conveyance of properly formatted messages from a worker on one container to the worker on another container. Legacy and simple embedded hardware paths require great flexibility for SHP containers. This includes simple point-to-point links with no addressing and little protocol.

Communication between SHP containers and GPP ORBs must adapt the CORBA messaging in the GPP ORB to the particular messaging/transport system supported by the SHP container. Since the SHP containers are typically running in resource-constrained environments, this adaptation would commonly happen in the GPP environment. When a platform is assembled and integrated, the various containers and ORBs must be interoperable. Thus the platform integrator has the responsibility to ensure messaging adaptation between all pairs of containers and ORBs that will support connections. If the SCA Core Framework assumes that all executable devices can communicate with each other, then all the containers must indeed have interoperable communication. If the SCA Core Framework has specialized knowledge of which device pairs can interoperate, then the interoperability can be correspondingly less complete.

The SHP-GPP interoperation can be accomplished in many ways, including:

- *The SHP container can translate simple worker messages into fully compliant GIOP messages, and support an ORB transport protocol supported by the GPP ORBs.*
- *The SHP container can move worker messages directly onto a hardware connection, optionally inserting a message prefix to help a message level bridge in the GPP environment translate the message into a GIOP message for the receiving ORB's transport.*
- *The GPP ORB can have interceptors to translate the messages above the level of ORB transport.*
- *The SHP container can, during connection setup, provide a repository ID to help the GPP properly translate the messages.*

The main point here is that this interoperability is delegated to the platform integrator and from there to the container implementers. It is kept distinct and independent of the portability requirements presented to SHP component authors. Note also that the addressing information for all communication paths among ORBs and SHP container is not standardized, and does not need to be. Even ORBs have no such standardization other than when GIOP and TCP/IP are used. The addressing information in object references need only be understood by ORBs and/or containers that participate in that particular transport path.

4 SHP REQUIREMENTS

This section describes requirements common to both RCC and RPL classes of components and containers.

4.1 SHP LOGICAL DEVICE AND CONTAINER REQUIREMENTS

SHP Logical Device implementations act as a factory for SHP generic proxies (one per worker/component). This is analogous to how the CF DomainManager acts as a factory for ApplicationFactories, how CF ApplicationFactories act as factories for CF Applications and how GPP ExecutableDevices act as factories for application components (*Resources*). SHP Logical Device implementations are independent of SHP component implementations according to the goal of separation of concerns between platform and waveform developers.

4.1.1 SHP Logical Device Requirements

Requirement 1 A SHP Logical Device shall act as a factory for SHP generic proxies

SHP Logical Devices shall create objects which support the CF::Resource interface when the execute operation is performed. The terminate operation shall tear down the corresponding generic proxy object and all associated resources shall be released, including those for the associated worker in the associated container.

The interface to the object created by an SHP LogicalDevice in response to the *execute* operation, is used by the Assembly Controller to control and configure one worker.

4.1.2 SHP Container Requirements

SHP containers are controlled by the SHP LogicalDevice (in general) and generic proxies (per worker). Thus most of its actions are determined by SCACS requirements on that software. The remaining requirements are to support the specific interfaces visible to workers: The *Worker* interface, the port interfaces and the local services interfaces. All of these are specific to the class of container and thus are described in the sections 5 and 6 below for each class.

Container interoperability is in general a platform requirement: the platform integrator includes containers that interoperate. For GPP containers, CORBA compliance requires at least that interoperation via IIOP (GIOP over TCP/IP) be supported. In all other cases the mechanisms (messaging formats, protocol stacks, connection hardware (NIC) or wire level signaling (e.g. Ethernet)) have no requirements other than that the containers in a platform interoperate. This enables platform integrators to take maximum advantage of existing communication mechanisms and simply adapt the various containers to use them.

4.2 SHP COMPONENT REQUIREMENTS

4.2.1 Component interfaces

IDL has a rich set of features and data types that make its wholesale adoption for defining interfaces for SHP component ports and containers inappropriate. Thus this extension defines a proper subset that, while being appropriate and efficiently implementable for SHP components, is still fully compatible and usable for GPP components and containers using CORBA. The most significant areas of subsetting are with respect to data types for arguments and return values. The key subset is defined by removing features from IDL:

Requirement 2 IDL constraints on component port interfaces

SHP component ports shall only support one interface, which is the most derived interface, and not any intermediately inherited interfaces.

IDL interfaces for the ports of components with SHP implementations shall use only IDL language features supported in the CORBA/e Micro profile as defined by the OMG to not use these IDL language features:

- *Value types*
- *Contexts*
- *Component features (those IDL features for CCM)*
- *Fixed point, wide char or wide string types*
- *Long double types (greater than 64 bit floating point)*
- *Any, or TypeCode*
- *Abstract interfaces*
- *Imports*

This subsetting essentially leaves messages (requests, responses, exceptions) consisting of scalar, structure/union, fixed array, strings, variable array types, of 8, 16, 32 and 64 bit integers or 32 or 64 bit floating point. It does support two way operations as well as oneway operations.

4.2.2 Worker ports

SCA component ports, as defined in the SCD, act as "uses" ports or "provides" ports, with "uses" ports issuing requests and receiving responses, and "provides" ports receiving requests and issuing responses. Thus SCA component ports, whether "uses" or "provides", both send and receive messages. When the IDL interface associated with a port has only oneway operations, then a "uses" port only issues requests, and a "provides" port only receives requests.

SHP Worker ports are more primitive, and are unidirectional: input-only or output-only. Thus each *component* port implemented by the worker corresponds to two *worker* ports, one for input and one for output. A "uses" port of the component implies a worker output port for issuing request messages and an associated worker input port for receiving response messages. When the IDL interface of the component port has only oneway operations, only one underlying worker port is needed — for "uses" ports, an output port to send requests; for "provides" ports, an input port to receive requests.

Thus the SCD component port definition (including the associated IDL) implies the number of worker (input and output) ports, and there is a correspondence between each component port in the SCD, and the worker port used for requests and the worker port used for responses. The Worker APIs defined below for each SHP class will define how the correspondence works.

4.2.3 Worker port interfaces

Interfaces at worker ports (the APIs workers use to communicate over their ports) are defined in the sections 5 and 6 for each class of SHP devices below. The format of messages produced and consumed at those ports is derived from the IDL in the SCD of the SPD containing the SHP implementation. Programming language artifacts (e.g. header files for C) can be automatically

or manually generated from the port IDL to aid worker implementations in producing and consuming messages. The format of the messages is a simple mapping from the IDL subset, and is common to all classes of SHP components. The operation (in requests) and exception (in replies) identification are not included in the message formats since they are separately specified in the port APIs as members of the port structures and arguments to container functions.

Requirement 3 Message formats based on port IDL

Messages produced and consumed by workers ports (request, response, exception) shall be formatted based on the following rules.

- *The operation or exception identification shall not be included in the message.*
- *Alignments are relative to the start of the message as defined here.*
- *All scalar types shall be aligned on their [CDR]-defined sizes, with padding inserted to achieve this.*
- *Structures and unions shall be aligned based on the largest alignment requirement of their constituent members.*
- *All variable length types (string, sequence) shall be preceded by an unsigned long (32 bit) length value. The string length shall include a null terminating byte. This unsigned long length value shall be aligned on the maximum of its own size (32 bits) and the alignment of the elements of the sequence or string. For sequences, it is the number of elements.*

The latter two rules have stronger alignment requirements than are found in [CDR], to enable messages without unbounded elements to be directly defined with C-language data structures. The generated language artifacts expressing message formats, used by worker source code, are discussed in the sections for each class below. The messages are essentially [GIOP] request or reply bodies with the stronger alignment noted above. As seen at the worker port interfaces, request messages never have context information due to the CORBA/e Micro profile IDL restrictions, and reply messages never contain LOCATION_FORWARD or NEEDS_ADDRESSING_MODE information.

4.2.4 The *Worker* interface

Workers are controlled and configured by a stack of software elements from (at the top) the ApplicationFactory and Application, to the AssemblyController, to any (optional) custom proxy, to the generic proxy, and the container. The *Worker* interface is exposed directly by the worker to the container. In general, the interface is used to initialize, start, stop, test, configure, query, and shut down (release) the worker. The specifics are described in the sections 5 and 6 for each class, but the operations are described common to both models, in section 4.2.8 below.

4.2.5 Worker Property Address space

To provide for very efficient access to, and implementation of, a worker's configuration and test properties, they are mapped into a contiguous address space, with each property having an address (offset) in this address space. The property files (Properties Descriptors) referenced by the SPD that include the worker implementation are used to determine the overall size of this address space, and to determine the offset and type for each configuration property. These property files are, in order of consideration: 1) the one that is referenced by the SCD that is

referenced in the SPD, 2) the one that is a direct sub-element of the SPD (*softpkg*), and 3) the one that is a subelement of the *implementation* element in the SPD.

The generic proxy uses these offsets to implement the *CF::Resource configure* and *query* operations. The offsets are also used by the worker itself to locate its own properties. The property address space is defined using the same layout rules as specified above in Requirement 3 in section 4.2.3. The layout rules allow the property address space to be specified in a C data structure, with appropriate padding members inserted to ensure uniform padding. The layout rules are normative.

The generic proxy must have access to some form of the information in the relevant Properties Descriptor, to perform the required translation from the *configure* and *query* operations to underlying accesses to the Worker's property address space. A typical, efficient implementation of a generic proxy would obtain this information (property names, types, and offsets) during the load or execute operation (execparams or part of the actual binary file supplied during load), and would thus not have to process any XML files at run time.

Requirement 4 Worker Property Address space layout

The generic proxy and the worker shall access properties in a contiguous address space laid out according to the following rules:

- *Alignment and content rules are from requirement 3 in section 4.2.3.*
- *configure and test properties will be mapped*
- *execparam properties will not be mapped*
- *allocation properties will not be mapped*
- *factory properties will not be mapped*
- *objref property types are assumed to be stringified references thus treated as strings (unspecified in the SCA specification as of 2.2.2 – perhaps another CP).*
- *max_string_size attribute shall be specified for simple or simplesequence properties of type string or objref to indicate the maximum size of the simple string or objref values (max_string_size attribute is in a CP in Appendix A). Space is reserved for one byte more than this value to hold the (required) null byte to terminate the string or objref.*
  Note: until the dtd changes to allow this, this value will be specified as a separate line in the property's description element, of the form "max_string_size=NNN".
- *max_sequence_size attribute shall be specified for simplesequence and structsequence properties to indicate the maximum number of values in the sequence (max_sequence_size attribute is in a CP in Appendix A).*
  Note: until the dtd changes to allow this, this value will be specified as a separate line in the property's description element, of the form "max_sequence_size=NNN".
- *name attribute becomes the worker implementation language's structure member name*
- *a testId property will be added in the implementation SPD if any test properties are present.*
- *test input values are processed for new property names*
- *test output values are processed for new property names*

- *if test properties have the same name as other configuration properties, they must have the same (simple) data type.*

The max_string_size attribute has the semantics of the maximum size for bounded strings in IDL. The max_sequence_size attribute has the semantics of the maximum size for bounded sequences in IDL. In implementations without these attributes allowed (until the PRF DTD is updated per the change proposals in Appendix A), they are found on separate lines in the description field separated from their decimal values by '=' (e.g. "max_string_size=7").

4.2.6 The Local Services APIs

Each SHP component implementation restricts its dependencies and references, other than the Worker and port interfaces, to the local service API defined for its class. The specifics are described in the sections for each class.

4.2.7 Operation and exception ordinals from SCD IDL

Operation and exception ordinals shall be used at worker ports to identify operations (in requests) and exceptions (in responses), and are inferred from the SCD information as follows:

- Operations are ordered by depth-first traversal of interface inheritance, after considering the operations of CORBA::Object first.
- Exceptions are ordered by their appearance in operation signatures, starting with 2, with zero indicating no exception (normal response), and a 1 indicating a system exception.

4.2.8 Worker Interface Control Operations and States

The Worker Interface mentioned above provides access to the worker property address space for memory-mapped access for reading and writing the values of configuration properties. The Worker interface also provides a set of SHP "control operations".

SHP "control operations" are invoked on workers via the controlling software stack, with the generic proxy on top and the worker at the bottom. They are common to both RPL and RCC workers. These control operations are invoked in response to analogous (but not precisely the same) operations in the CF::Resource interface implemented by the generic proxy. Furthermore, these control operations are defined relative to a simple state machine model. The "worker control states" are defined below. The states are defined and then the normative control operations are defined.

4.2.8.1 Worker Control States

These states are not normative, but are used to clarify the use of the control operations. Also, access to configuration properties is only permitted (and thus expected) in certain states, and never during the execution of the defined control operations.

4.2.8.1.1 Worker State: ***Exists***

The worker exists as the result of instantiation via the *execute* operation on the SHP logical device. It is unconfigured (its properties are not in a known state), and has no defined functionality other than to accept the *initialize* control operation.

4.2.8.1.2 Worker State: ***Initialized***

The worker has successfully completed the *initialize* control operation, but is *not* in an operational state (performing its normal function). Its state is stable, known, and is ready to enter

its operational state. It can accept the *start* control operation to enter its operational state, or the *release* control operation to revert to the ***exists*** state. It can also accept configuration operations (to set or get values of configuration properties), and the configuration-related control operations: *beforeQuery* and *afterConfigure*.

4.2.8.1.3 Worker State: ***Operating***

The worker has successfully completed the *start* control operation and is in its operational state, performing its normal function based on configuration property values and behavior at its ports. It can accept the *stop* and *release* control operations. It can also accept configuration operations (to set or get values of configuration properties), and the configuration-related control operations: *beforeQuery* and *afterConfigure*.

4.2.8.1.4 Worker State: ***Suspended***

The worker has successfully completed the *stop* control operation, and has suspended operation. It is not initiating or accepting any activity at its ports. It can accept the *start* or *release* control operations. . It can also accept configuration operations (to set or get values of configuration properties), and the configuration-related control operations: *beforeQuery* and *afterConfigure*. The values of properties are as they were when the *stop* operation completed (unless subsequently changed by the generic proxy in response to a *configure* operation).

4.2.8.1.5 Worker State: ***Unusable***

The worker has unsuccessfully completed the *release* control operation, or is otherwise known to be unusable (i.e. not re-usable) after actions specific to an implementation model (RCC or RPL below). Any further control operations or configuration accesses have undefined behavior.

4.2.8.2 Worker Control Operations

These control operations apply to both RCC and RPL workers, and all have a simple result of success or failure. The mechanism and interface for invoking them and determining their result is specific to each model and is detailed below in sections 5 and 6.

A worker can assume that accesses to configuration properties will not occur during the execution of a control operation. The stack of control software/firmware must ensure that this is the case.

Control operations are mostly optional, meaning that the implementation may eliminate them altogether and communicate this to either the container or generic proxy. For each operation, it is defined what is assumed by the control stack if the operation is unimplemented. A portable implementation shall include the list of implemented control operations, which is used by platform-specific build/compile tools and scripts to inform the generic proxy.

4.2.8.2.1 initialize

4.2.8.2.1.1 Brief Rationale.

The *initialize* operation provides the worker with the opportunity to perform any one-time initialization to achieve a known state prior to normal execution. It is called as part of the generic proxy's CF::Resource *initialize* operation.

4.2.8.2.1.2 Behavior.

The initialize operation is implementation dependent. The configuration properties are not necessarily initialized with any initial configuration values, prior to invoking this operation. The behavior is similar to that specified for the CF::Resource *initialize* operation.

If the operation succeeds, the worker is considered to be in the ***initialized*** state. The operation is only defined to be invoked when the worker is considered to be in the ***exists*** state.

If unimplemented, successful initialization is assumed (i.e. no initialization is required).

4.2.8.2.2 start

4.2.8.2.2.1 Brief Rationale.

The *start* operation provides the worker with the opportunity to perform any one-time initialization that is dependent on initial configuration property values, since those property values are not set prior to the *initialize* operation. This operation also provides the opportunity to prepare to resume internal processing after the *stop* operation is used. It is called as part of the generic proxy's CF::Resource *start* operation.

4.2.8.2.2.2 Behavior.

The *start* operation shall put the worker in an operating condition. The configuration properties are initialized with any initial configuration values, prior to invoking this operation the first time. This operation also may be called after the worker is placed in a non-operating condition by the *stop* operation. The behavior is similar to that specified for the CF::Resource *start* operation.

If the operation succeeds, the worker is considered to be in the ***operational*** state. The operation is only defined to be invoked when the worker is considered to be in the ***initialized*** or ***suspended*** states.

4.2.8.2.3 stop

4.2.8.2.3.1 Brief Rationale.

The *stop* operation is provided to command a worker to stop internal processing in a way that can be later restarted (resumed) via the *start* operation. It is called as part of the generic proxy's CF::Resource *stop* operation. It can be useful in system analysis and debugging since the worker’s state can be examined via configuration properties and it is not performing further work.

4.2.8.2.3.2 Behavior.

The *stop* operation shall disable all processing, and put the worker in a non-operating condition. The behavior is similar to that specified for the CF::Resource *stop* operation.

If the operation succeeds, the worker is considered to be in the ***suspended*** state. The operation is only defined to be invoked when the worker is considered to be in the ***operational*** state.

If unimplemented, the worker is not stoppable, and thus the operation is assumed to fail.

4.2.8.2.4 release

4.2.8.2.4.1 Brief Rationale.

The release operation requests that the worker perform any final processing (except performing I/O via ports, which is not allowed). Any resources obtained from local (container-supplied)

services should be returned. It is invoked as part of the generic proxy's CF::Resource *releaseObject* operation.

4.2.8.2.4.2 Behavior.

The *release* operation shall release all resources obtained from local services. The behavior is similar to that specified for the CF::Resource *releaseObject* operation although it does not imply actual uninstantiation/destruction.

If the operation succeeds, the worker is considered to be in the ***exists*** state. If it fails, the worker is considered to be in the ***unusable*** state. The operation is defined to be invoked only when the worker is considered to be in the ***exists***, ***initialized***, ***operational***, or ***suspended***, states.

If unimplemented, the operation is assumed to fail, meaning that the resulting state is ***unusable***, rather than ***exists***.

4.2.8.2.5 afterConfigure

4.2.8.2.5.1 Brief Rationale.

The *afterConfigure* operation allows the worker to be notified when some of its configuration properties have changed. This operation notifies the worker that one or more configuration property values have changed as part of a single invocation of the generic proxy's CF::Resource::configure operation, and thus any action based on those changes should be taken.

The generic proxy knows, via private metadata, which writable properties require notification of changes via this operation. Thus this operation will only be invoked when such properties have been written. This enables the simplest and fastest model where the generic proxy may be able to perform highly efficient programmed I/O operations (load/store) directly into the worker's property structure with no further interaction. The worker's generic proxy causes the container to invoke this worker operation after modifying any properties during its performing a CF::Resource *configure* operation, before that operation returned.

4.2.8.2.5.2 Behavior.

The worker shall perform any special processing associated with the changed properties. An error will indicate that all property changes since the last afterConfigure operation was invoked were not accepted or acted upon.

The operation is only defined to be invoked when the worker is considered to be in the ***initialized, operating*** or ***suspended*** states. If unimplemented, the operation is assumed to succeed, meaning that the worker has no need to be informed after configuration properties have been changed.

A portable implementation shall include the list of properties requiring either afterConfigure or beforeQuery, which is used by platform-specific build/compile tools and scripts to inform the generic proxy.

4.2.8.2.6 beforeQuery

4.2.8.2.6.1 Brief Rationale.

The beforeQuery operation notifies the worker that the container is about to read one or more configuration property values. Similar to afterConfigure, the generic proxy will only invoke this operation when some of the properties that will be queried are known to require such notification, as specified in the implementation metadata available to the generic proxy.

The generic proxy, in performing the CF::Resource *query* operation, shall cause the container to call this operation, before reading values in the property structure.

4.2.8.2.6.2 Behavior.

The worker shall perform any required processing prior to configuration properties being accessed.

The operation is only defined to be invoked when the worker is considered to be in the ***initialized, operating*** or ***suspended*** states. If unimplemented, the operation is assumed to succeed, meaning that the worker has no need to be informed before configuration properties are read. A portable implementation shall include the list of properties requiring either afterConfigure or beforeQuery, which is used by platform-specific build/compile tools and scripts to inform the generic proxy.

4.2.8.2.7 test

4.2.8.2.7.1 Brief Rationale.

The *test* operation allows components to be "blackbox" tested. This allows Built-In Test (BIT) to be implemented and this provides a means to isolate faults (both software and hardware) within the system. It is called as part of the generic proxy's CF::Resource *runTest* operation.

4.2.8.2.7.2 Behavior.

The *test* operation shall use the *testId* property and the current value of properties to determine which of its predefined test implementations should be performed and to provide additional information to the implementation-specific test to be run. The *test* operation shall return the result(s) of the test in readable property values. The behavior is similar to that specified for the CF::Resource *runTest* operation, except that test properties are written prior to this operation, and test result properties are read after this operation.

No afterConfigure or beforeQuery operations will be called by the generic proxy or container for the properties written or read as a consequence of the execution of the runTest operation by the generic proxy.

The operation is only defined to be invoked when the worker is considered to be in the ***initialized*** state. If unimplemented, the operation is assumed to fail, meaning that the worker implements no tests and thus no testID can be valid.

4.2.8.3 Summary Diagram for Control States and Operations

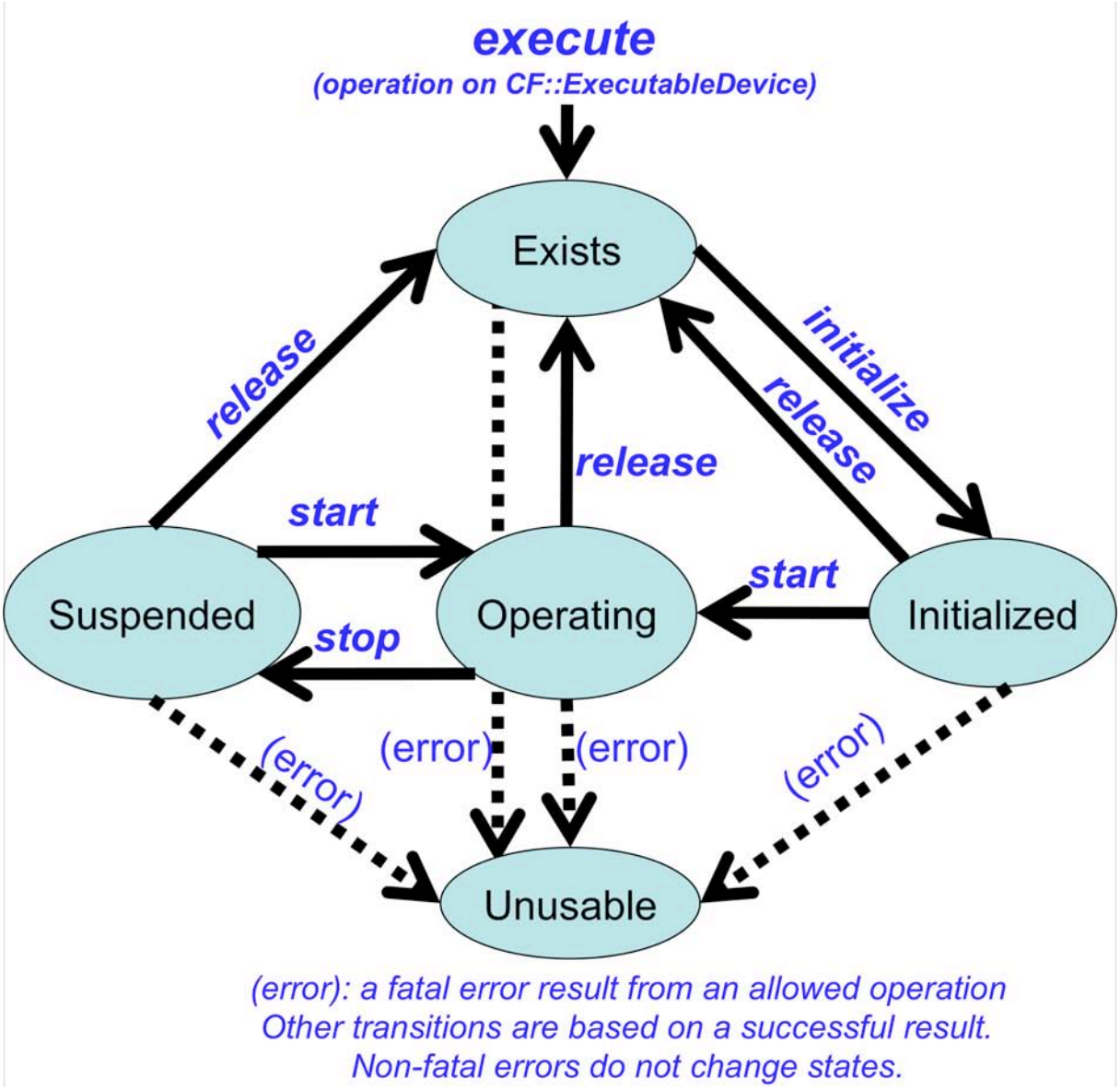

Figure 5 Control States and Control Operations for transition

4.2.9 Required Implementation MetaData

There are three items of metadata that all portable implementations must have, in addition to the source code: the implementation name, a list of implemented control operations, and a list of properties that require beforeQuery or afterConfigure control operations. These items are used by platform-specific code generation/build/compile tools and scripts. The exact format for these simple items is not defined, but will be needed for such tools and scripts to properly generate code artifacts and private metadata for generic proxies.

Since these items are implementation-specific, and used during the application build process, they are not considered appropriate for the SCA XML files (SPD, PRF, SCD).

5 RCC (E.G. DSP) REQUIREMENTS

The requirements listed here do not preclude RCC containers from implementing a complete CORBA and POSIX AEP environment, suitable for executing components written to the current SCA GPP software component specification. In that case such an RCC container can be used to also directly support existing SCA-compliant GPP-style components. Such a container is acting like a GPP container to support components written to the GPP requirements and specifications. In contrast, this specification addresses the more traditional resource constrained, performance sensitive RCC deployments. There are two profiles for RCC environments to service two design tradeoffs. Most RCC requirements are common to both profiles. There is a base profile, and an optional multithreaded profile that is a superset of the base profile. When a requirement is specific to the multithreaded profile, it will be identified as such.

- The *base* profile requires a minimal run-time infrastructure (and no threads) and places certain restrictions on the worker implementations. It enforces a Finite State Machine (FSM) or fully event-driven coding style that does not always allow easy porting or reengineering from the more conventional sequential, blocking style code.

  This profile is appropriate for simple worker implementations that can be easily coded this way, and for highly resource-constrained processing devices with little operating system and middleware support. It generally provides minimum footprint and maximum performance.

- A *multithreaded* profile, with a richer run-time infrastructure with fewer restrictions placed on the worker. A more common multithreaded and blocking I/O coding style is supported. Workers may create threads and make blocking calls to wait for I/O buffers.

  This profile is appropriate when porting code written in a multi-threaded/sequential style, or when the worker's behavior requires more capable operating system and middleware support (but for which full CORBA implementations are not available or appropriate).

An RCC container implementation may choose to support only the base profile, or the larger multithreaded profile. An RCC ExecutableDevice shall export a property that lists the supported profiles so that the ApplicationFactory can know whether a given device is an appropriate candidate for a given RCC component implementation. Any RCC container implementation that supports the *multithreaded* profile shall also support the simpler *base* profile.

A worker may be written to either profile. A component may provide separate implementations for both. An implementation makes use of the above-mentioned property to select a device and container that supports the profile to which it is written.

An implementation that supports the multithreaded profile must inform the platform-specific RCC compile/build tools that it does so, in order for the appropriate property values to be set that describe the implementation, enabling the CF::ApplicationFactory to match the implementation to a device with the appropriate runtime support. This is in addition to the class-independent metadata mentioned in section 4.2.9.

Worker examples for both profiles, illustrating both conventional and FSM coding styles, are presented in section 5.5. All RCC worker code, irrespective of profile, must be re-entrant since the same code may be run in multiple threads for different worker instances.

5.1 RCC EXECUTION MODEL

The specific requirements of the RCC execution model in the base profile include:

- All execution threads are supplied by the container so there is no need for workers to create threads.
  Reason: reduces complexity of component code, eliminates requirement to support a thread capability
  Reason: allows container to decide whether multithreading is even needed - frequently it is not.
  Reason: worker code is reusable between threaded and non-threaded containers.
  Reason: container can be interrupt driven, single stack multi-tasking, or multithreaded

- Execution of a worker is enabled when the worker is in the "operating" state, as defined above, and some combination of ports are "ready", or time has passed.
  A "run" method is called when ports are ready or time has passed.
  Worker execution is a series of "runs" initiated by container logic.

  Reason: simplify component code, eliminate any "wait for next thing" logic or loops
  Reason: make run decisions centralized in the container, fast, simple and small
  Reason: make worker multiplexing within a container able to be more efficient than multithreading

- Workers declare a "run condition" as a simple logical "or" of a small number of masks.
  Each mask indicates a logical "and" of "port readiness bits" (1 << port ordinal).
  An input port is ready when there is an input message for the worker to look at.
  An output port is ready when there is an output buffer that can be filled by the worker.
  The default run condition is one mask of all ones, meaning "run me if all my ports are ready".
  If a port being ready is not required for the run condition, the worker can still test its readiness. Otherwise it can assume the readiness of ports indicated in its run condition. A timeout is also provided in the run condition.

  Reason: simplest model for specifying when to run components with multiple ports; common cases are trivial, minimal code, container can optimize synchronization, and minimize worker code.

- Workers indicate which ports should "advance" at the end of each run, before returning.
  Advancing a port means releasing (input) or sending (output) the current buffer and implicitly requesting another one.

  Reason: most efficient way to indicate this since it piggy-backs on the "return".

5.2 RCC LOCAL SERVICES

Local service APIs support workers running in a container. Containers are required to supply them, and workers are constrained to use only them. GPP containers and components use the POSIX AEP defines in SCACS Appendix B.

5.2.1 Container Multithreading Support

Threading (in RCC environments) can generally be supplied by the container, and not require the component writer to create or manage threads. The RCC *base* profile does *not* allow workers

themselves to create or manipulate threads, and the worker control interface is guaranteed not to have more than one of its operations entered at a time. The RCC *multithreaded* profile *does* allow the use of the POSIX pthread API to create and manipulate threads, although control operations will not be called simultaneously in different threads. Only one control operation will be called at a time in any worker: the container must enforce this.

RCC environments are frequently configured without support for threads. The lack of thread support in the *base* profile is consistent with such environments. RCC containers supporting only the *base* profile may still run multiple workers in separate threads.

RCC containers supporting the *multithreaded* profile must support multithreading and support workers to wait/block for input or output buffers.

5.2.2 RCC AEP as a small subset of GPP AEP Appendix B of SCACS.

Requirement 5 DSP AEP POSIX Subset

SHP RCC Containers shall support the following POSIX AEP defined in SCACS Appendix B with the following deletions:

- *Table B-1 unchanged*
- *Table B-2: Set entirely to NRQ in the baseline profile.*
  Set all entries to NRQ except POSIX_THREAD options in multithreaded profile*
- *Table B-3, B-4, B-5 unchanged (all NRQ anyway)*
- *Table B-6 set all to NRQ except abort() – remove signals. Abort only affects calling worker.*
- *Table B-7, B-8, unchanged (all NRQ anyway)*
- *Table B-9:set all to NRQ, remove file system*
- *Table B-10 unchanged (all NRQ anyway)*
- *Table B-11 set all to NRQ, remove FD management*
- *Table B-12 set all to NRQ, remove device I/O, except:*
  stdout, fflush(stdout),putchar(),perror(),printf()
- *Table B-13, B-14, B15, B16 unchanged (all NRQ anyway)*
- *C-Language table B-17: unchanged [perhaps remove all the localtime/local?]*
- *Table B-18. B-19: unchanged*
- *Table B-20, semaphores, in base profile set entirely to NRQ*
- *Table B-21, timers, set all to NRQ except gettimeofday, without timezone support*
- *Table B-22, threads, in base file set entirely to NRQ*
- *Table B-23, thread safe, set readdir_r to NRQ (it is I/O)*

The philosophy of the *base* profile AEP subset is to allow functions that are simply libraries (rather than OS services), but remove services that could conflict with the lean container execution model in that profile. This basically leaves the typical ANSI-C (or ISO C89) runtime library (without I/O) – most DSPs have this available anyway, even those environments without any multithreading.

5.3 RCC INTERFACES BETWEEN WORKER AND CONTAINER

The RCC *Worker* interface is called by the container, and implemented by the worker. The worker provides its methods (private functions) to the container . A "worker context object" is provided to the worker by the container as the first argument to each worker method, analogous to the implied "this" argument to object methods in C++. This object is a data structure with some advertised structure members (rather than an explicit structure definition: this is similar to the way POSIX defines public structure members in C APIs, also analogous to public member data in C++). This structure also provides the worker with access to port-specific state, any worker-requested memory resources, and the data structure containing its configuration properties.

The profile (base vs. multithreaded) affects the Worker interface in two major ways (in addition to the expanded AEP in the multithreaded profile).

5.3.1 Execution based on events

- In the *base* profile, the container calls the worker's "run" method when the worker's "run condition" is true, based on the availability of input buffers (with data) or output buffers (with space) or the passage of time. The worker's run method executes, processes any available inputs and outputs, indicates when messages are completely consumed as input or produced as output, makes any changes to the "run condition", and returns. Workers never block. The container conveys the messages in buffers between collocated workers as well as into and out of the container as required by the application assembly's connections.
- In the *multithreaded* profile, in addition to calling the "run" method if the "run condition" is true, the worker may also execute in threads it creates, and use a blocking API to wait for input or output buffers.

5.3.2 Sending or receiving messages via ports

- In the *base* profile, the worker can only indicate that an input buffer has been processed and/or an output buffer has been filled in one execution of the run method. It must return from the run method in order to wait for more buffers to process. The worker never blocks.
- In the *multithreaded* profile, the worker may explicitly wait for buffers to become available on individual ports, via calls to the container that block if no buffers are available.

5.3.3 Buffer management

The Worker Interface is designed to have the container provide and manage all buffers. Thus input ports operate by the container providing buffers to the worker filled with incoming messages, and the output buffers operate by the container providing buffers for the worker to fill with messages before being sent. Output buffers are either:

- obtained for a specific output port (since they may be in a special memory or pool specific to an particular output hardware path), or
- originally obtained from an input port and passed to output ports, possibly with no copying.

Logically, there are three operations performed with buffers, which is the basis for the specific APIs defined later. They are all non-blocking functions:

- Request that a new buffer be made available. For an input port, it is filled by the container with a new input message. For an output port, it is to be filled by the worker with a new output message. In both cases the ownership of the buffer passes from container to worker when it becomes available. The new buffer may or may not be immediately available.
- Release a buffer to be reused, with its contents discarded. The ownership passes from worker to container. Input buffers must be released (or sent) in the order received, i.e. ownership of input buffers must be passed from worker to container in the order that ownership was given from container to worker.
- Send (enqueue) a buffer on an output port, to be automatically released (later) after the data is sent. The ownership passes from worker to container. If the buffer was originally obtained from an input port, it must be sent or released in the order received.

The concept of "current buffer" of a port exists to support a convenient model for workers that need no special buffer management. A port is ready if it has a current buffer. A current buffer on an input port is available to read data from. A current buffer on an output port is available to write data into. The concept of "advancing" a port, is simply a combination of releasing (input) or sending (output) the current buffer of the port, and requesting a new buffer to be made available on that port, to become the current buffer when it becomes available in the future. So simple workers wait for ports to be ready (to have buffers), process input buffers into output buffers, advance input and output ports, and return. The interface is designed to make this common model as simple as possible. Workers wait for some ports to be ready, and advance ports after processing buffers.

Several more advanced buffer management requirements are supported by adding extra capabilities for certain situations as needed:

- To support sliding window algorithms, allow the worker to own previous buffers (not releasing them), while new ones are requested; i.e. allow selective (in order) explicit input buffer release, not implicitly just the most recent buffer obtained. The worker must still release the buffers in the order received.
- To support zero copy from input ports to output ports, allow the worker to send a buffer obtained from an input port to an output port, and not require an empty current buffer to fill on the output port. Such buffers must be sent (or released) in the order received.
- To support port-specific callbacks without any worker dispatching overhead, allow the worker to associate callback functions with each port that convey new buffers from container to worker.
- To support blocking semantics in the *multithreaded* profile, allow for blocking calls to wait for an input buffer with new data on an input port, or wait for a new output buffer to fill on an output port.

The above features are only needed in certain cases, and can be ignored entirely for most simple workers. Supporting them results in the following interface features:

- Provide separate non-blocking interfaces for releasing, sending, requesting buffers.
- Provide an interface to wait for a buffer on a port.
- Provide for per-port worker-supplied callback functions.

5.3.4 Worker Interface Description

The operations in the interface described below are as if it was a C++ or IDL interface, but it is a C interface, with an initial explicit argument to each method that provides a worker context structure, analogous to the implicit "this" argument in C++.

Requirement 6 RCC Worker interface between container and worker

Containers shall use, and RCC Workers shall implement the following Worker C-language interface, defined in a header file named "RCC_Worker.h", and reserve the prefix "RCC" for compile-time constants and types defined in this file:

The Worker interface consists of control operations whose behavior is defined in section 4.2.8.2 (*initialize*, *start*, *stop*, *test (like runTest)*, and *release (like releaseObject), beforeQuery and afterConfigure)*. An additional *run* operation supports the event-driven execution model defined above. Only *run* is required in the base profile. Only *start* is required in the multithreaded profile. All the rest are optional. All processing of the worker occurs in the context of these operations unless, in the *multithreaded* profile, the worker creates its own threads. All operations take a pointer to a RCCWorker structure to provide a reentrant context to the worker (a C++ "this"), and to enable multiple instances of the same worker to coexist in a container.

All defined data types use the common prefix "RCC". There are no defined external symbols.

The interface uses several basic integer types consistent with their CORBA C/C++ language mapping, to provide some compiler independence. The integer types are defined using the ISO C 99 <stdint.h> types. These basic types are: uint16_t and uint32_t. The type RCCBoolean is aliased to uint8_t, to be consistent with the CDR-defined size in property space and message content layouts. The RCCOrdinal type is an alias for uint16_t, and is used when ordinals are required (ports, operations, exceptions, properties).

5.3.5 Types

5.3.5.1 *RCCResult*

The RCCResult type is an enumeration type used as the return value for all worker operations. It indicates to the container what to do when the worker operation completes, as follows:

- *RCC_OK: worker operation succeeded without error*
- *RCC_ERROR: operation did not succeed, but error is not fatal to worker or container, thus the operation may be retried.*
- *RCC_FATAL: worker should never run again; it is non-functional; the container or other workers may be damaged; the worker is in an **Unusable** control state as defined above. Note that the container may know that it, or other workers are protected from damage, but the portable worker indicates this condition in case there is no such protection.*
- *RCC_DONE: worker needs no more execution; a normal completion. See usage below.*
- *RCC_ADVANCE: worker is requesting that all ports be advanced (run operation only).*

These result values apply to each operation as defined in their specific behavior. Some values are not valid results for all operations. When the result is RCC_ERROR or RCC_FATAL, the worker may set the *errorString* member of the RCCWorker structure to non-NULL to add a string description to the error. See the RCCWorker type below.

5.3.5.2 *RCCPortMask*

The RCCPortMask type is a bit mask used to indicate a boolean value for each port. It is an integer type where a given bit being == 1 (1L << port_ordinal) means the value for that port is TRUE.

5.3.5.3 *RCCRunCondition*

The RCCRunCondition structure type holds the information used by the container to determine when it is appropriate to invoke the *run* operation of a worker. The defined members are always written by the worker, and never by the container.

RCCRunCondition Member Name	**Member Data Type**	**Member Description**
portMasks	RCCPortMask *	A pointer to a zero-terminated sequence of port masks, each of which indicates a bit-mask of port readiness. The run condition is considered "true" when any of the masks is "true" (logical OR of masks). A mask is "true" when all indicated ports are ready (logical AND of port readiness). A port is indicated by its bit being set (1 << port_ordinal). If the pointer is NULL, it indicates no masks.
timeout	RCCBoolean	Indicates that the *usecs* member is used to determine when enough time has passed to make the run condition true. Thus this value can used to enable or disable the timeout, without changing *usecs*.
usecs	uint32_t	If this amount of time has passed (in microseconds) since the run operation was last entered, the run condition is true.

Thus the overall run condition is the logical OR of the portMasks and the timeout. If the worker offers no *run* method in its *RCCDispatch* structure (see below), run conditions are ignored. The structure is completely defined with ordered members defined to enable static initialization:

```
typedef struct {
    RCCPortMask *portMasks;
    RCCBoolean  timeout;
    uint32_t    usecs;
} RCCRunCondition;
```

5.3.5.4 *RCCMethod*

The RCCMethod type is simply a convenience type declaration to declare worker methods (in C) that do not have any arguments other than the RCCWorker structure. It declares a function type that takes a worker's context structure as the single input argument, and returns a RCCResult value. It is defined as follows:

```
typedef RCCResult RCCMethod(RCCWorker *this);
```

5.3.5.5 *RCCRunMethod*

The *RCCRunMethod* type specifies the signature of the worker's *run* operation. It is used in the definition of the *RCCDispatch* type, and can be used as a convenient way to forward-declare the method function for initializing the *RCCDispatch* structure. The operation is described below. The type is defined as follows:

```
typedef RCCResult RCCRunMethod(RCCWorker *this,
                               RCCBoolean timedout,
                               RCCBoolean *newRunCondition);
```

5.3.5.6 *RCCPortMethod*

The *RCCPortMethod* type specifies the signature of functions that the worker can supply for any port, to request that the container call that individual function when a buffer is available for a particular port, or other port conditions occur. This is an alternative to having the container call the *run* method based on run conditions involving *all* ports. It would normally be used to process the current buffer on a port (to process data in an input message, or to produce data for an output message).

The arguments indicate which port, and why the callback was called. The *reason* argument is set by the container as follows

- RCC_OK: buffers are now available
- RCC_ERROR: data or messages were lost
- RCC_FATAL: connectivity was lost.

[What is the responsibility of the container to report data or connectivity errors?]

The *port* argument allows one callback function to be used for multiple ports. The return value indicates RCC_OK, RCC_ERROR, or RCC_FATAL.

The type is defined as follows:

```
typedef RCCResult RCCPortMethod(RCCWorker *this,
                                RCCPort *port,
                                RCCResult reason);
```

5.3.5.7 *RCCPortInfo*

The RCCPortInfo structure type holds the static port-related information needed by the container. It is supplied statically by the worker if the port has non-default port attributes. An array of these structures, terminated by the "port" member being RCC_NO_ORDINAL, is referenced by the "portInfo" member of the RCCDispatch structure defined below. . If all the attributes have default values, there is no need for a structure to be included for a given port.

RCCPortInfo Member Name	**Member Data Type**	**Member Description**
port	RCCOrdinal	The port that this structure describes, using the port enumeration described in 5.4.1.
maxLength	uint32_t	The maximum size in bytes of messages for this port. Default is 0, meaning there is no maximum size. Supplying messages larger than this at an input port is not allowed. For output ports, the worker will not need to produce any messages exceeding this size.
minBuffers	uint32_t	Indicates the minimum number of buffers that the worker requires to be available at this port. The default is 1. Zero means 1.

5.3.5.8 *RCCDispatch*

This type is the "dispatch table" for the operations of the worker interface. It represents functionality a worker provides to a container when it is loaded. The container must gain access to this structure when the worker is loaded and executed. All members are initialized by the worker source code. This structure also contains other descriptive information required by the container to use the worker.

The members of this structure are defined as follows:

RCCDispatch Member Name	Member Data Type	Member Description
Members below	**enable the container**	**to check the metadata vs. the actual compiled description.**
version	uint32_t	Initialized from the RCC_VERSION macro to inform the container about version of RCC API (header files) used to check mismatches.
numInputs	uint16_t	The number of input ports.
numOutputs	uint16_t	The number of output ports.
propertySize	uint32_t	The size in bytes (sizeof) of the property structure.
memSizes	uint32_t*	The memory requirements that should also be in the allocation properties. This zero-terminated array of memory sizes indicates a set of memory allocations required by the worker. Multiple allocations are allowed so that the worker need not aggregate all its requirements in a single contiguous allocation when it doesn't need to. This pointer may be NULL, indicating no such allocations are required.
threadProfile	RCCBoolean	Initialized to allow the container to error check
Members below	**supply the container**	**with function pointers to methods.**
initialize	RCCMethod*	A pointer to the initialize method function. May be NULL if there is no functionality.
stop	RCCMethod*	A pointer to the stop method function. May be NULL if there is no functionality.
start	RCCMethod*	A pointer to the start method function. May be NULL if there is no functionality in the base profile. Required in the multi-threaded profile.
release	RCCMethod*	A pointer to the release method function. May be NULL if there is no functionality.
afterConfigure	RCCMethod*	A pointer to the test method function. May be NULL if there are no notify properties.
beforeQuery	RCCMethod*	A pointer to the test method function. May be NULL if there are no notify properties.
test	RCCMethod*	A pointer to the test method function. May be NULL if there are no tests.
run	RCCRunMethod*	A pointer to the run method function. May be NULL.
Members below	**supply the container**	**with behavioral attributes**
runCondition	RCCRunCondition*	The initial run condition used if the *run* member is not NULL. If this is NULL it implies a run condition of all ports ready and no timeout.
portInfo	RCCPortInfo *	A pointer to an array of RCCPortInfo structures, which describe static non-default port information to the container. Is NULL if no such non-default information is needed.
optionalPorts	RCCPortMask	A mask indicating ports that may be unconnected. The default, 0, means that all ports must be connected before the worker is started. The container enforces this.
portInfo	RCCPortInfo*	Points to an array of port descriptions for non-default port attributes. The pointer may be NULL, and if not, is terminated by a RCCPortInfo structure with the *port* member == RCC_NO_ORDINAL..

The portable worker source code shall provide an external symbol for its statically initialized *RCCDispatch* structure, and the development flow for the device will arrange for the container to find it.

There are several implementation-dependent approaches for the container to obtain the location of the *RCCDispatch* structure (which approach is used is determined by the development flow specified by the container for its workers), these include:

- Static linking of the worker with the container, using external symbols for worker implementation dispatch tables.
- Using the default (unnamed) entry point in the loadable worker code as the address of a function that returns the *RCCDispatch* structure or the address of the structure itself.
- Specifying the name of a symbol identifying the dispatch table in the *entrypoint* element for the implementation in the softpkg file for the component that is passed to the ExecutableDevice in the CF::execute() call.
- Specifying the name of a symbol identifying an entry point (when using DLL-style loading-dll.so) in the *entrypoint* element for the implementation in the softpkg file for the component that is passed to the ExecutableDevice in the CF::execute() call. The entry point function could return a pointer to the dispatch table.
- Providing a standard C "main" entry point which the container calls and which returns a pointer to the dispatch table.

Since the technique is determined by the platform, the developer of a portable implementation must only supply the dispatch table external symbol to the platform-specific build/compile tools/scripts.

In order to statically initialize this structure, the tagged member initialization syntax from the C99 standard should be used, e.g.:

```
RCCDispatch xyz = {
   .version = RCC_VERSION,
   .numInputs = XYZ_N_INPUT_PORTS,
   .numOutputs = XYZ_N_OUTPUT_PORTS,
   .threadedProfile = TRUE,
   .start = startXyz,
} RCCDispatch;
```

5.3.5.9 *RCCBuffer*

The RCCBuffer type is a structure that holds information about a buffer. It has the members described in the table below, all of which are written by container functions and not by the worker. There may be other undocumented members.

RCCBuffer Member Name	Member Data Type	Member Description
data	void * const	A const pointer to the data in the buffer. When this member of the structure is NULL, there is no buffer.
maxLength	const uint32_t	The maximum number of bytes that may be placed in the buffer (maximum message length). This is *not* the actual length of the valid data in the buffer.

5.3.5.10 *RCCPort*

The *RCCPort* type is a structure that contains the current state of a worker's port. The container is free to define this structure with any content and any ordering as long as the documented members are supported. (This style of structure standardization is from POSIX). The members are written by either the container or the worker, but never both. Members written by the container are declared "const" to enhance error checking when compiling worker implementations. For code readability and reliability, there is a union in the structure that aliases two substructures with members used differently for input ports and output ports. Members used for input ports only are prefixed with "input.". Members used for output ports only are prefixed with "output.".

RCCPort Member Name	Member Data Type	Written by	Member Description
current	const RCCBuffer	container	The current buffer to use for the port. The actual format of the message in the buffer depends on which request, response, or exception message is represented. The value of the *data* member is NULL if there is no current buffer. Port readiness is equivalent to the current.data != NULL.
input.length	const uint32_t	container	For input ports, the actual number of bytes in the current buffer (message length).
input.u.operation	const RCCOperation	container	For input ports associated with provider component ports, the operation ordinal of the request message in the current buffer.
input.u.exception	const RCCException	container	For input ports associated with "uses" component ports, the exception ordinal of the response message in the current buffer. Zero indicates no exception.
output.length	uint32_t	worker	For output ports, the number of bytes in the message the worker has placed in the current output buffer.
output.u.operation	RCCOperation	worker	For output ports associated with "uses" ports, the operation ordinal of the request message in the current buffer.
output.u.exception	RCCException	worker	For output ports associated with "provides" ports, the exception ordinal of the response message in the current buffer. Zero indicates no exception.
callback	RCCPortMethod*	worker	Per port callback function optionally set by worker, initialized to NULL by container. If not NULL, called by container when port becomes ready or when an error on the port occurred.

The worker writes the non-const "output" members prior to indicating to the container that the output port should be advanced. The worker writes the "callback" member (to non-NULL) in the *initialize* or *start* operation. The callback member is examined by the container after the start *operation* returns, and after subsequent callback functions on the port return.

5.3.5.11 *RCCContainer*

This structure provides the worker with a set of container functions. It is a dispatch table in the RCC*Worker* structure rather than a set of external symbol definitions, for the following reasons:

- Preserves the entire RCC worker interface with no external symbols other than the standard AEP. This enables certain optimal incremental linking and loading scenarios.
- Enables container functions to be varied per worker by the container to enable certain optimizations based on worker properties or behavior or port configurations.
- Preserves an object-oriented coding style by treating container functions as a "base class".
- Simplifies symbol namespace used by the RCC profiles: no non-AEP link-time symbols.

The entire structure is const, and initialized by the container. The use of these container methods is optional, and entirely unneeded for simple workers that only use run conditions and the *run* method. They are all used to provide additional flexibility and functionality in buffer handling.

RCCContainer Member Name	Member Data Type	Member (function) Description
release	`void (*) (RCCBuffer *buffer)`	Release a buffer for reuse. If the buffer is a current buffer for a port, it will no longer be the current buffer. Buffer ownership passes back to the container. Non-blocking. Must be done in the order obtained, per port.
send	`void (*) (RCCPort *port, RCCBuffer *buffer, RCCOrdinal op, uint32_t length)`	Send a buffer on an *output* port. If the buffer is a current buffer for a port, this buffer will no longer be the port's current buffer. The "op" argument is an operation or exception ordinal. Buffer ownership passes back to the container. Non-blocking.
request	`RCCBoolean (*) (RCCPort *port, uint32_t max)`	Request a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. If the port already has a current buffer, the request is considered satisfied. The return value indicates whether a new current buffer is available. Non blocking.
advance	`RCCBoolean (*) (RCCPort *port, uint32_t max)`	Release the current buffer *and* request that a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. This is a convenience/efficiency combination of release-current-buffer+request. The return value indicates whether a new current buffer is available. Non blocking.
wait	`RCCBoolean (*) (RCCPort *port, uint32_t max, uint32_t usecs)`	Block the caller until there is a current buffer for a port, implying a request for a new buffer. It will timeout after *usecs* microseconds, returning TRUE on timeout, FALSE otherwise. Only available in the *multithreaded* profile.
take	`void(*) (RCCPort *port, RCCBuffer *releaseBuffer, RCCBuffer *takenBuffer)`	Take the current buffer from a port, placing it at *takenBuffer. If releaseBuffer != NULL, first release that buffer. Non-blocking. Ownership is retained by the worker. The current buffer taken is no longer the current buffer. This is used when the worker needs access to more than one buffer at a time from an input port.

The worker source code would call these container functions based on the "this" argument, like:

```
this->container.wait(port, usecs);
this->container.release(id);
//or
RCCContainer *c = &this->container;
c->wait(port, usecs);
c->release(id);
```

This approach was taken both for brevity and to avoid any external symbolic linkage for this interface.

5.3.5.12 *RCCWorker*

This structure type (typedef) represents the visible state of a worker. It is passed by reference to every operation, analogous to the implicit "this" argument in C++. The container is free to define this structure (in RCC_Worker.h) with any content and any member ordering as long as the documented members are supported. (This style of structure standardization – defined members but no explicit standardized structure definition or member ordering, is from POSIX).

The members are written by either the container or the worker, but not both. Members written by the container are declared "const" to enhance error checking when compiling worker implementations.

RCCWorker Member Name	Member Data Type	Written by	Member Description
properties	void * const	container	A const pointer to the properties structure for the worker, whose layout is implied by the PRFs for the implementation. The value may be NULL if there are no such properties.
memories	void * const []	container	An array of const pointers to the memory resources requested by the worker in allocation properties in its implementation PRF. Any memories that are not read-only are initialized to zero before a worker executes any method.
container	const RCCContainer	container	A dispatch table of container functions.
runCondition	RCCRunCondition*	worker	Initialized from the RCCDispatch runCondition member. Checked by container after calling the start method.
errorString	char *	worker/ container	Initialized to NULL by the container. Optionally set by worker before returning RCC_ERROR or RCC_FATAL. Reset to NULL by container when worker returns.
connectedPorts	RCCPortMask	container	A mask indicating which ports are connected.
ports	RCCPort []	varies by member	An array of RCCPort structures defined above, indexed by port ordinals derived from the SCD. This array is the last member, and thus makes this structure variable length.

5.3.6 Operations

5.3.6.1 *initialize*

See section 4.2.8.2.1 for rationale and behavior.

5.3.6.1.1 Synopsis.

```
RCCResult initialize(RCCWorker *this);
```

5.3.6.1.2 Returns.

This operation shall return a RCCResult value.

5.3.6.1.3 Exceptions/Errors.

If the initialization cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK if normal worker execution should proceed.

5.3.6.2 *start*

See section 4.2.8.2.2 for rationale and behavior.

5.3.6.2.1 Synopsis.

```
RCCResult start(RCCWorker *this);
```

5.3.6.2.2 Returns.

This operation shall return a RCCResult value.

5.3.6.2.3 Exceptions/Errors.

If the *start* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK if normal worker execution should proceed. It shall return RCC_DONE, if no worker execution should proceed. Returning RCC_DONE indicates to the container that the worker will never require execution, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker being terminated (when the generic proxy receives the *terminate* operation).

5.3.6.3 *stop*

See section 4.2.8.2.3 for rationale and behavior.

5.3.6.3.1 Synopsis.

```
RCCResult stop(RCCWorker *this);
```

5.3.6.3.2 Returns.

This operation shall return a RCCResult value.

5.3.6.3.3 Exceptions/Errors.

If the *stop* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

5.3.6.4 *release*

See section 4.2.8.2.4 for rationale and behavior.

5.3.6.4.1 Synopsis.

```
RCCResult release(RCCWorker *this);
```

5.3.6.4.2 Returns.

This operation shall return a RCCResult value.

5.3.6.4.3 Exceptions/Errors.

If the *release* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, or it is unable to return resources it allocated, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

5.3.6.5 *afterConfigure*

See section 4.2.8.2.5 for rationale and behavior.

5.3.6.5.1 Synopsis.

```
RCCResult afterConfigure(RCCWorker *this);
```

5.3.6.5.2 Returns.

This operation shall return a *RCCResult* value.

5.3.6.5.3 Exceptions/Errors.

If the *afterConfigure* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

5.3.6.6 *beforeQuery*

See section 4.2.8.2.6 for rationale and behavior.

5.3.6.6.1 Synopsis.

```
RCCResult beforeQuery(RCCWorker *this);
```

5.3.6.6.2 Returns.

This operation shall return a RCCResult value.

5.3.6.6.3 Exceptions/Errors.

If the *beforeQuery* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

5.3.6.7 *run*

5.3.6.7.1 Brief Rationale.

The *run* operation requests that the worker perform its normal computation. The container only calls this operation when the worker's *run condition* is true. This allows the container to fully manage all synchronization, minimizing the resources in use when the worker is not able to run, and minimizes the time to switch between running one worker and running another when the container uses a simple single threaded execution model.

5.3.6.7.2 Synopsis.

```
RCCResult run(RCCWorker *this,
              RCCBoolean timedout,
              RCCBoolean *newRunCondition);
```

5.3.6.7.3 Behavior.

The *run* operation shall perform the worker's computational function and return a result. The *run* operation may use information in its property structure, state of its ports, and its private memory to decide what to do. Normally this involves using messages in buffers at input ports to produce messages in buffers at output ports.

The *timedout* input parameter indicates whether the run operation is being invoked due to time passing (the *usecs* value of the run condition). This indication is independent of port readiness.

Each port's *current.data* member may be tested to indicate port readiness.

The run operation can indicate that all ports should be advanced by a special return value. It can also indicate disposition of buffers and ports by using the release, send, request, or advance container functions. Each function may only be called once per port per execution of the run operation.

The *run* operation may change the run condition by writing a TRUE value to the location indicated by the newRunCondition output argument, and setting a new runCondition in the runCondition member of RCCWorker. The worker maintains storage ownership of all run conditions.

5.3.6.7.4 Returns.

This operation shall return a RCCResult value.

The value of RCC_ADVANCE shall indicate that all ports should be advanced that were ready on entry to the *run* function and were not subject to container functions called since then (e.g. advance, request etc.).

5.3.6.7.5 Exceptions/Errors.

If the *run* operation cannot succeed, it shall return RCC_ERROR, indicating that it should not be called again (run condition always false). If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK or RCC_ADVANCE if normal worker execution should proceed (and the run operation called again when the run condition is true). It shall return RCC_DONE, if no worker execution should proceed. Returning RCC_DONE indicates to the container that the worker will never require execution, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker being terminated (when the generic proxy receives the *terminate* operation).

5.3.6.8 *test*

See section 4.2.8.2.7 for rationale and behavior.

5.3.6.8.1 Synopsis.

```
RCCResult test(RCCWorker *this);
```

5.3.6.8.2 Returns.

This operation shall return a RCCResult value.

5.3.6.8.3 Exceptions/Errors.

If the *test* operation cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK. Note that test "results" are provided in readable properties, so returning RCC_ERROR implies that the test could not be run at all, usually due to invalid test properties. It does not indicate that the result of running the test was not success.

5.4 CODE GENERATION FROM METADATA

5.4.1 Ports inferred from SCD IDL

The port interfaces for a worker shall be consistent with the associated port IDL definitions in the SCD for which the worker is an RPL implementation. RPL worker ports are inferred from the SCD information as follows:

- Ports are ordered according to their appearance in the SCD XML as port elements.
- Ports with only oneway operations imply one worker port
- Ports with some two way operations imply two worker ports; first read-only input then write-only output.
- Provides ports read request messages and write response messages
- Uses ports write request messages and read response messages
- All worker port interfaces are read or write, but not both.

Thus there is a single ordering of all worker ports, combining the required input and output ports implied by the SCD and IDL. An enumeration of named worker port ordinals will be generated in the file Xyz_Worker.h, , using the typedef name XyzPort (Where Xyz is the worker implementation name). The enumerated constants will be all upper case, of the form WWW_PPP_RRR_DIR. WWW is the worker implementation name and PPP is the port name. RRR and DIR are only included with bidirectional component ports (when some IDL operations are *not* oneway). RRR will be "REQUEST" or "REPLY" as appropriate. DIR will be "IN" or "OUT". These rules fully describe port roles for authors and readers.

For aid in initializing the numInputs and numOutputs members of the RCCDispatch structure, the macros WWW_N_INPUT_PORTS and WWW_N_OUTPUT_PORTS shall also be generated with the appropriate values.

5.4.2 Property structure from SPD Property Files

The layout of the property structure is derived from the property files (PRF) of the implementation as defined for the property address space layout rules defined in Requirement 4. This layout is expressed as a normal C struct, with member names being the properties' name attributes from the PRFs. This struct type can used for accessing property values using the "properties" pointer in the *RCCWorker* structure. This struct can be automatically generated. The rules for generating this structure are:

- *The standard name of the struct type (typedef name) is XyzProperties where "Xyz" is the mixed case (usually with initial upper case) worker implementation name (chosen by the implementer, not necessarily any name from the SPD, since there is no "name" attribute in the implementation element of the SPD).*
- *The standard name for the generated header file containing this definition (and others) is "Xyz_Worker.h". This precludes using "RCC" for the worker implementation name.*
- *Simplesequence or structsequence properties are preceded by an unsigned long member whose name is the property name with "_length" appended. Padding may be added before and after the length to achieve the required alignment of this length field as well as the sequence data following it.*
- *Sequence properties are represented by structure members that are C arrays whose length is the max_sequence_size attribute from the PRF.*
- *Struct or structsequence properties have structure tags the same name as the property name, preceded by the worker implementation name.*
- *The names of the PRF integer types for simple properties: short, ushort, long, ulong, octet, char are mapped to the respective CDR-sized <stdint.h> types: int16_t, uint16_t, int32_t, uint32_t, uint8_t.*
- *The names of the non-integer types for simple properties are the SCA-defined names capitalized and prefixed with "RCC": e.g. RCCBoolean, RCCChar, RCCFloat, RCCDouble.*
- *Properties that are simple string or objref properties are "RCCChar" arrays whose size is one more than the max_string_size attribute of that simple property in the PRF, and the values are null terminated strings.*

5.4.3 Operation and exception ordinals from SCA IDL files

The Xyz_Worker.h file will supply an enumeration for each port, defining operation or exception ordinals. The ordinals will have the typedef name XyzPqrOperation or XyzPqrException, and the constant name XYZ_PQR_ORD, where ORD is the name of the operation or exception. For any reply port, RCC_NO_EXCEPTION and RCC_SYSTEM_EXCEPTION are also valid. PQR is the upper-cased port name.

5.4.4 Message structures from IDL Files

Since the IDL allowed to be used for RCC component ports is according to the CORBA/e micro profile, arguments to operations can be variable in length. Thus in the general case, when multiple arguments are unbounded, no C data structure can capture the entire message format. This specification defines a normative structure for message contents up to and including the first variable length argument, and, for structures or unions, the first variable length member. This applies to request, reply and exception messages.

This partial solution to mapping the message contents into a normative structure is intended to facilitate many, but not all message content access. Since fixed size messages, or messages whose only variable element is the last argument, are very common, this solution adds portability value in many cases. More complex cases, after the first variable argument, must be accessed according to the message layouts as described in Requirement 3.

The message structures generated for each port, will have the typedef name XyzPqrOpr. Xyz is worker name, Pqr is port name, and Opr is operation or exception name. The members of the structure are named the same as the property structure above, but with argument names from IDL rather then property names from the PRF file. Within structures, unions, or exceptions, the names are from the IDL data type definition members. If the first variable length argument is a sequence of variable length types (sequence or string), the type of the member for that variable length type will be uint32_t, which will be the last member in the structure.

The message structure member types for the IDL integer types: *short, unsigned short, long, unsigned long, long long, unsigned long long, octet, and enum* are mapped to the respective CDR-sized <stdint.h> types: int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t, uint8_t, uint32_t.

The member types for the non-integer IDL basic types are IDL names capitalized and prefixed with “RCC”: e.g. RCCBoolean, RCCChar, RCCFloat, RCCDouble. String types are “RCCChar” arrays.

[This partial coverage of structures covers all the SCA 2.2.2 APIs with two exceptions: the fixed size Ethernet MAC address in the EthernetDevice pushpacket operation is defined as a sequence rather than an array, and the CF:InvalidProperties exception used (incorrectly?) in several places does not fit in the CORBA/E micro file at all due to use of type Any.]

5.5 CODE EXAMPLES

5.5.1 Worker code example for base profile

Here is a simple example of an "Xyz" worker using the base profile whose:

- initial run condition was the default (condition == NULL, usecs == 0, run when all ports are ready, no timeout),
- initialize, release and test methods are empty
- one input port (0) with interface XyxIn (only oneways) and one output port (1) XyzOut (only oneways)
- one oneway IDL interface operation Op1 on input (i.e. can ignore "operation"), which is an array of 100 "shorts".
- one oneway IDL interface operation Op2 on output
- one simple property, called center_frequency, of type float.

The worker could have automatic or manually generated structures like this (based on SCD, IDL, and property files), and put in a file called "Xyz_Worker.h":

```
#include "RCC_Worker.h"
/* Generated from Properties Descriptor */
typedef struct {  /* structure for defined properties /
   RCCFloat center_frequency;
} XyzProperties;
/* Generated from SCD/IDL from here down */
typedef struct {  // structure for message for operation
   int16_t ishorts[100];
} XyzInOp1;

typedef struct {  // structure for message for operation
   int16_t oshorts[100];
} XyzOutOp2;

typedef enum { // port ordinals
   XYZ_IN,
   XYZ_OUT
} XyzPort;

typedef enum { // operation ordinals
   XYZ_OUT_OP2
} XyzOutOperation;

typedef enum { // operation ordinals
   XYZ_IN_OP1
} XyzInOperation;
```

The actual code for the worker would look like this:

```
#include "Xyz_Worker.h"

/* Define the initialize method, setting output operation to be a */
/* constant, since it is the only one. Make it static because it */
/* doesn't need to be global */
 static RCCResult
initialize(RCCWorker *w) {
   w->ports[XYZ_OUT].output.u.operation = XYZ_OUT_OP2;
   return RCC_OK;
}

/* Define run method to call the "compute" function, reading from */
/* input buffer, writing to output, applying current value of the */
/* "center frequency" property.  Make it static because there is no */
/* reason to make it global. */
 static RCCResult
run(RCCWorker *w, RCCBoolean timedout, RCCBoolean *newRunCondition)
{
   XyzProperties *p = w->properties;
   XyzInOp1 *in = w->ports[XYZ_IN].current.data;
   XyzOutOp2 *out = w->ports[XYZ_OUT].current.data;

   /* Do computation based in ishorts, and frequency put results */
   /* in oshorts.  Extern is here simply for readability. */
   extern void compute(int16_t *, int16_t *, float);

   compute(in->ishorts, out->oshorts, p->center_frequency);
   /* Ask container to get new input and output buffers */
   return RCC_ADVANCE;
}

/* Initialize dispatch table for container, in a global symbol /
 RCCDispatch
Xyz = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XyzProperties), RCC_NULL, RCC_FALSE,
   /* Methods */
   initialize, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL,
   RCC_NULL, run,
   /* Default run condition */
   RCC_NULL
};
```

Other than the compute function, the above example compiles to use less than 120 bytes on a Pentium processor.

A simple non-preemptive single-threaded container implementation would simply have a loop, testing run conditions, and calling run methods. A more complex environment might run workers in different threads for purposes of time preemption, prioritization, etc. This model allows a variety of container execution models while keeping the worker model simple.

So, on each execution, the worker sees the status of all I/O ports, and can read from current input buffers, and write to current output buffers. It must return to get new buffers, after specifying whether buffers are consumed or filled during the execution.

This simple execution environment can be easily implemented in GPP environments, providing a test environment and a migration path to GPPs.

5.5.2 Worker code example for multithreaded profile

Consider a similar but less trivial worker than that presented in section 5.3.5 above. This xyt worker which:

- does not require or support multithreading.
- has no processing to perform on initialize, release, test, start or stop.
- exports one SCA provides port with name IN that supports a single oneway IDL interface operation Op1 (i.e. can ignore "operation"), which is an array of 100 "shorts". This maps to worker port(0).
- has two SCA uses ports:
    - one that has the name OUT that contains a single oneway interface operation Op2. This maps to worker port(1)
    - one that has the name Compute that contains a single two way interface operation Process. This maps to.
    - output worker port (2) for output commands
    - input worker port (3) for input responses
- has one property, called center_frequency
- responds to the oneway operation call on its provides port by:
    - manipulating the input data in some local Compute function based on the current center frequency value.
    - passing the intermediate result to another component via the two way Process operation on its one uses port for further processing.
    - forwarding the final result to its destination via Op2 oneway operation on its other uses port.
- Takes advantage of the fact that the data structure for the provides and both uses operations is the same to avoid copying data.

This worker would have code-generated structures like this (based on SCD, IDL, and property files):

```
#include "RCC_Worker.h"
/* Generated from Properties Descriptor */
typedef struct { /* structure for defined properties */
   RCCFloat center_frequency;
} XytProperties;

// Generated from SCD/IDL from here down
typedef struct { /* structure for message for operation */
   int16_t ishorts[100];
} XytInOp1;

typedef struct { /* structure for message for operation */
   in16_t oshorts[100];
} XytOutOp2;

typedef struct { /* structure for message for operation */
   int16_t ishorts[100];
} XytProcessInOp3;

typedef struct { /* structure for message for response */
   int16_t oshorts[100];
} XytProcessOutOp3;

typedef enum {
   XYT_IN,
   XYT_OUT,
   XYT_COMPUTE_REQUEST_OUT,
   XYT_COMPUTE_REPLY_IN
} XytPort;

typedef enum {
   XYT_OUT_OP2
} XytOutOperations;

typedef enum {
   XYT_IN_OP1
} XytInOperations;

typedef enum {
   XYT_COMPUTE_PROCESS
} XytComputeOperations;
```

The actual code for the worker is presented below for each of the two profiles (most error handling omitted for sake of clarity):

5.5.2.1 Worker implementation using the multithreaded profile, with input callback

This illustrates an input-based callback function that has a sequential coding style in which the worker blocks on the two way remote call until the response is received and processing can continue. No extra threads are used.

The code is:

```
#include "Xyt_Worker.h"

/* Define the input port callback method to call the local "compute"
 * function, reading from input buffer, applying current value of the
 * "center frequency" property, followed by remote two way  "process"
 * operation, finally writing the result to output port XYT_OUT */
 static RCCResult
computeInput(RCCWorker *this, RCCPort *inPort, RCCResult reason)
{
   XytProperties *p = this->properties;
   RCCContainer *c = &this->container;
   XytInOp1 *data = inPort->current.data;
   RCCPort *computeOut = &this->ports[XYT_COMPUTE_REQUEST_OUT],
           *computeIn = &this->ports[XYT_COMPUTE_REPLY_IN],
           *otherOut = &this->ports[XYT_OUT];
   extern void compute(int16_t *, int16_t *, float);

   if (reason != RCC_OK)
      return RCC_FATAL;

   /* Do some computation based in ishorts, and frequency; put results
    * back in ishorts, in place */
   compute(data->ishorts, data->ishorts, p->center_frequency);

   /* Call Process operation on "uses" port: buffer ownership passes
    * back to container. */
   c->send(computeOut, &inPort->current, XYT_COMPUTE_PROCESS,
           inPort->input.length);
   /* Wait until response received (or 100 msec error timeout). */
   c->wait(computeIn, 100, 100000);
   if (computeIn->input.u.exception == RCC_NO_EXCEPTION)
      c->send(otherOut, &computeIn->current, XYT_OUT_OP2,
              computeIn->input.length);
   else
      c->release(&computeIn->current);
   return RCC_OK;
}

/* Define the initialize method to perform worker initialization. */
 static RCCResult
initialize(RCCWorker *this)
{
  this->ports[XYT_IN].callBack = computeInput;
  return RCC_OK;
}

/* Initialize dispatch table provided to container. We only need the
 * initialize method to register the callback.  No run method needed */
 RCCDispatch
Xyt = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XytProperties), RCC_NULL, RCC_TRUE,
   /* Methods */
   initialize
   /* all remaining members zero/NULL */
};
```

Other than the compute function, the above example compiles to use less than 300 bytes on a Pentium processor.

5.5.2.2 Worker implementation using the base profile using state-machine style

This illustrates a finite state machine coding style in a worker that maintains an internal state to simulate blocking on the two way remote call until the response is received and processing can continue.

```
#include "Xyt_Worker.h"
/* Define two different run conditions to represent two states */
static uint32_t
  state1Ports[] = {1 << XYT_IN, 0},
  state2Ports[] = {1 << XYT_COMPUTE_REPLY_IN, 0};
static RCCRunCondition
  awaitingInput = {state1Ports},
  awaitingResponse = {state2Ports};

/* Define start method, setting run condition, which is also state. */
 static RCCResult
initialize(RCCWorker *this)
{
   this->runCondition = &awaitingInput;
   return RCC_OK;
}
/* Define run method to call the local "compute" function
 * reading from input buffer, applying current value of the
 * "center frequency" property, followed by remote two way
 * "process" operation, finally writing result to output. */
 static RCCResult
run(RCCWorker *this, RCCBoolean timedout, RCCBoolean *newRunCondition)
{
   RCCContainer *c = &this->container;

   /* Use run condition as state indicator */
   if (this->runCondition == &awaitingInput) {
      RCCPort
         *inPort       = &this->ports[XYT_IN],
         *computeOut   = &this->ports[XYT_COMPUTE_REQUEST_OUT];
      XytInOp1 *in     = inPort->current.data;
      XytProperties *p = this->properties;

      /* do some computation based in ishorts, and frequency;
       * put results back in same buffer (in-place) */
      extern void compute(int16_t *, int16_t *, float);
      compute(in->ishorts, in->ishorts, p->center_frequency);

      /* Call Process op on uses port - buffer ownership passes back to
       * container.  Input port is advanced by taking buffer away from
       * it. */
      c->send(computeOut, &inPort->current, XYT_COMPUTE_PROCESS,
              inPort->input.length);      /* length of message */
      this->runCondition = &awaitingResponse; /* update state */
   } else {
      RCCPort *computeIn = &this->ports[XYT_COMPUTE_REPLY_IN];
      RCCPort *otherOut = &this->ports[XYT_OUT];

      if (computeIn->input.u.exception == 0)
         c->send(otherOut, &computeIn->current, XYT_OUT_OP2,
                 computeIn->input.length);
      else
         c->advance(computeIn, 0);
      this->runCondition = &awaitingInput; /* update state */
   }
   *newRunCondition = RCC_TRUE; /* to container: runcondition is new */
   return RCC_OK;
}
```

```
/* Initialize dispatch table provided to container. We only need the
 * initialize method to register the callback.  No run method needed */
 RCCDispatch
Xyt = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XytProperties), RCC_NULL, RCC_FALSE,
   /* Methods */
   initialize, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL,
   RCC_NULL, run,
   /* all remaining members zero/NULL */
};
```

Other than the compute function, the above example compiles to use less than 330 bytes on a Pentium processor.

5.6 REQUIREMENTS SUMMARY FOR RCC WORKERS

- RCC workers are written to implement the Worker interface, called by the container, and optionally use the Container interface, called by the worker. The interface is conveyed to the container via the RCCDispatch type.
- RCC workers may call functions defined in the RCC AEP.
- RCC workers use data structures and ordinals as defined in the metadata code generation section (which will normally be automatically generated, but don't have to be).
- RCC worker authors must make the following implementation-specific metadata available to the build process for compiling/linking/loading worker implementation binaries on a given platform:
    - Implementation name
    - Which control operations are implemented
    - Which properties require beforeQuery or afterConfigure notification
    - Memory allocation requirements of the implementation code
    - Profile used in the implementation (base or multithreaded).
    - Minimum number of buffers required at each port (default is one).
- The interface metadata from SCA XML and CORBA IDL is used, along with the implementation metadata above, to drive the code generation process.

6 RPL (E.G. FPGA/ASIC) REQUIREMENTS

For RPL interfaces, the Open Core Protocol (OCP) provides the basis for interface definition since it provides both bus/technology/language independence as well as a richness that is optional and is easily "compiled out" when not used by either side of a communication. It can be used in VHDL, Verilog, SystemC, etc. It is proven, open, fully defined, documented and does not require implementation license fees. Implementations of master or slave can be very simple, and ignore all the advanced features, while still being compatible with rich implementations. OCP by itself is not sufficient, but the modest messaging semantics defined here overlaid on OCP results in the appropriate solution.

The RPL Worker control interface (with Worker as OCP slave), the interface for each port, with Worker as master (or optionally as slave in some cases), and the local memory interfaces, with Worker as master, are all defined as OCP interfaces.

The RPL interface requirements are designed to support partial reconfiguration of FPGAs, based on currently available technology and information about future capabilities.

6.1 INTRODUCTION TO OCP

From "Design and Reuse", http://www.us.design-reuse.com/articles/article5816.html:

OCP is an internationally supported IP core interconnection standard advanced by the OCP International Partnership, or OCP-IP (www.opcip.org). OCP provides cores with significant modularity that increases genuine design reusability without core rework.

OCP provides a combined communication protocol and hardware connection scheme for SOC logic blocks. As a protocol, OCP provides a minimal command set for memory-mapped, split-transaction activities such as read, write, and read exclusive. (Read exclusive is a command variant of the read command specifically designed for use within semaphore processing.) These atomic activities occur over threads – communication channels that guarantee the response sequence order matches that of the original associated requests. Thread tags accompanying memory-mapped requests therefore providing an efficient way to present requests from an OCP master interface to an OCP slave interface.

Slaves can independently process requests arriving on different threads using any priority algorithm or use the thread tags to vector the individual requests to different independent logic such as different DRAM banks. However, on any single thread, a slave core must preserve the response order, though there is no consideration for order preservation across collective threads. This consideration is essential when the slave core is a DRAM memory controller. Here, in-order responses guarantee a stringent memory consistency model generally required for coherent multiprocessor system operation.

Finally, OCP has a number of extensions in addition to threads. One key extension is burst support, which provides slave cores hints that might enable them to optimize operations. For more OCP information and how it supports control and test signals see the Open Core Protocol Specification available from www.ocpip.org.

In summary, OCP is a standard socket architecture that provides a configurable socket interface and simple protocol for IP cores. Simplicity is essential because "All freshly designed protocols, no matter how disciplined their designers have been, must be treated with suspicion. Every protocol should be considered to be incorrect until the opposite is proven."

From “OCP Core Specification”

The OCP is equivalent to VSIA’s Virtual Component Interface (VCI). While the VCI addresses only data flow aspects of core communications, the OCP is a superset of VCI that also supports configurable sideband control signaling and test harness signals. Only the OCP defines protocols to unify all of the inter-core communication.

All the interfaces between workers and containers defined below use OCP, but the worker author may or may not use the full range of OCP features. The summary section below shows the allowable range of features that may be used for each OCP interface (worker, port, memory).

6.2 THE RPL WORKER CONTROL INTERFACE

The worker control interface is an OCP interface with the container as master and the worker as slave. Each worker has this single worker control interface, as well as interfaces for ports and local memory.

Similar to the RCC workers, the RPL worker receives an operation indication (initialize, run, release, test), performs the operation, and "returns". This standard control interface to the worker enables execution and receives indication of “DONE”, "ABORTED" or “FATAL”. The interface also provides configuration operations to read or write the worker's configuration address space. The interface also provides a way for the container to interrupt the control operations.

Requirement 7 RPL worker interface between container and worker

RPL containers shall use, and RPL Workers shall implement the following Worker OCP interface:

6.2.1 OCP Constraints

- Worker control port is named in the OCP configuration file by the interface_name parameter as worker_control.
- Container is master, worker is slave
  (OCP configuration parameters in parenthesis, unmentioned parameters have OCP-defined default values)
- Only read and write operations are supported
  (read_enable=1, write_enable=1, sdata=1, mdata=1)
- Three threads are used to provide for concurrency between control operations, configuration operations, and interrupt operations.
  (threads=3, sthreadbusy=1)
- Control operations are encoded on OCP address signals, in thread 0.
  - *initialize* is operation 0, as a read operation
  - *run* is operation 1, as a read operation
  - *release* is operation 2, as a read operation
  - *test* is operation 3, as a read operation
- Configuration operations are simply reads and writes in thread 1, using the address signals to access the configuration address space implemented by the worker.

- A write command on thread 2 is used by the container to interrupt any outstanding control or configuration operations. The worker, if it is functional, is expected to return a result from any outstanding control or configuration operation. Thread 2 can never be busy.
- No write data is required if there are no configurable properties (mdata=0, write_enable=0)
- Data width is 32, byte enables are optional (data_wdth=32, byteen=1)
- Address width is from 0 to 16, to accommodate the property space. (addr_wdth=0 to 16)
- No burst capabilities are allowed (*burst*=0, atomiclength=0)
- No request or data pipelining, or intermediate handshaking (datahandshake=0, writeresp_enable=0,respaccept=0,dataaccept=0)
- Reset signal is hard reset of worker, using the Mreset_n signal (mreset=1)
- Container will support workers written to allowable variations, implying generated or library-based "couplings".

6.2.2 Operation descriptions

All operations are OCP commands. All control operations (initialize, run, release, test) shall use the OCP read command on thread 0. All configuration operations shall use read and write commands on thread 1. The interrupt operation shall use a write command on thread 2. Control operations are encoded in the address signals (MAddr).

The worker shall use the SERR OCP response on any command, to indicate a fatal error.

6.2.2.1 initialize

The *initialize* operation shall put the worker in an operating condition. The property space is not necessarily initialized by the container before this operation, with any initial configuration values . This operation allows the worker to perform one-time initializations (e.g. initializing memories).

The data value returned by the read command for this operation is ignored. The DVA OCP response indicates success, ERR indicates error.

6.2.2.2 start

The *start* operation shall put the worker in an operating condition. The property space is initialized by the container before this operation, with any initial configuration values.

The data value returned by the read command for this operation is ignored. The DVA OCP response indicates success, ERR indicates error.

6.2.2.3 stop

The *stop* operation shall take the worker out of an operating condition.

The data value returned by the read command for this operation is ignored. The DVA OCP response indicates success, ERR indicates error.

6.2.2.4 release

The *release* operation shall perform any final processing (except performing I/O).

The value returned by the read command for this operation is ignored. The DVA OCP response indicates success, ERR indicates error.

6.2.2.5 run

The worker shall perform its function and return a result. The worker uses information in its property address space, state of its ports, to decide what to do. Normally this involves using data in buffers at input ports to produce data in buffers at output ports.

The value returned by the read command for this operation is ignored. The DVA OCP response indicates success, ERR indicates error.

6.2.2.6 configure

The *configure* operation is used to read or write new configuration property values on thread 1. The address width is dependent on the size of the property address space. The worker shall support the address width it needs, and shall define the OCP *addr_wdth* parameter accordingly. If the worker can accept configuration values concurrent with execution, it shall accept operations on thread 1 while thread 0 is being used for a control operation, so that configuration commands (to read and write configuration values) can be accepted during execution (i.e. during a run operation on thread 0).

The worker thus shall support reading and writing its configuration address space, whose layout is defined based in its Properties Descriptor of the SPD (see requirement 4).

6.2.2.7 test

The *test* operation shall use the current value of properties to determine which of its predefined test implementations should be performed and to provide additional information to the implementation-specific test to be run. The *test* operation shall return the result(s) of the test in readable property values.

A DVA OCP response indicates normal test completion. An ERR response indicates the test was not able to run and could indicate any results in properties. The data value returned is ignored.

6.3 THE RPL PORT INTERFACES

In addition to the worker control interface, there are interfaces for each of the worker's ports.

Port interfaces are generally based on OCP masters (for the worker), to allow workers to read and write buffered data in any addressing pattern. Operation/exception codes and length information (of the current buffer) are supplied by OCP *control* signals (for reads of requests or replies) or OCP *status* signals (for writes of requests or replies). Port readiness is indicated by the OCP SThreadBusy signal on thread 0. Advancing the port (making a new buffer of data available to the worker to read or write) is accomplished by an OCP write command on thread 1.

Several simplified configurations are made available:

- When only sequential access is required, no addressing is required.
- When no buffer boundaries are meaningful, no indications are required to move from one buffer to the next.
- When only a single IDL operation is used, no operation needs to be indicated.

- Input ports without addressing or buffer boundaries may be OCP slaves.

OCP configuration parameters allow for workers to request FIFO-style interfaces to ports to further simplify workers that have no need to address data within frames at some of its ports. Thus the worker author can state which of several styles of OCP-defined interfaces it requires for each port. For a given worker port, the container must provide adaptation ("shims", "gaskets") between the worker's ports and the containers communication infrastructure. The container supplier can decide whether these gaskets all normalize to some single container-provided port interface, or to provide a more optimized mapping from the different OCP port styles to the container's own private communication infrastructure.

Requirement 8 RPL worker port interface between container and worker

RPL containers shall implement, and RPL Workers shall use the following Port OCP interface:

6.3.1 Ports inferred from SCD IDL

The worker port interfaces shall be consistent with the associated component port IDL definitions in the SCD for which the worker is an RPL implementation. RPL worker ports are inferred from the SCD component port information as follows:

- Worker ports are named in the OCP configuration file by the interface_name parameter as <SCD-port-name>_request or <SCD-port-name>_response.
- Ports with only oneway operations imply one worker port
- Ports with some two way operations imply two worker ports; one for requests and one for responses.
- Provides component ports receive request messages on input ports and send response messages on output ports
- Uses component ports send request messages on output ports and receive response messages on input ports.
- All worker port interfaces are read or write, but not both.

6.3.2 OCP Constraints for Worker as OCP master for the port.

- Worker is master, container is slave
  (OCP configuration parameters in parenthesis)
- Input ports only support read operations.
  (read_enable=1, write_enable=0, sdata=1, mdata=0)
- Output ports only support write operations.
  (read_enable=0, write_enable=1, sdata=0, mdata=1)
- Port readiness is indicated using the SThreadBusy signal on thread 0
  (sthreadbusy=1, threads=1)
- FIFO usage by worker is indicated by zero width address, SRespInfo, 2 bits, indicates end of input data (bit 0 is data valid, bit 1 is end of data). End of output data may be indicated by MReqInfo, with the same encoding, and it also implies output port advancement.
  (addr_wdth=0, srespinfo=1, respinfo_wdth=2, reqinfo=0 or 1, reqinfo_wdth=0 or 2)

- Addressable buffer usage by worker is indicated by non-zero address width, which constrains maximum buffer size. The current buffer is always addressed at address 0.
  (addr_wdth=16 or 32)
- On output ports, operation or exception ordinals are indicated on Control signals
  (control=1, control_wdth=8)
- On input ports, operation or exception ordinals are indicated on Status signals
  (status=1, status_wdth=8)
- When the IDL indicates a single operation, request inputs or outputs can eliminate the corresponding control or status signals.
  (status =0, or control=0)
- When the IDL indicates no responses (all oneways), response inputs or outputs can eliminate the corresponding control or status signals.
- Data width is 16 or 32, byte enables are optional.
  (data_wdth=16 or 32, byteen=0 or 1)
- Workers may use aligned, precise, single request, incrementing bursts on input or output ports
  (burstprecise=1, burstsinglereq=1,burstlength=1,reqdata_together=1)
- No request or data pipelining, or intermediate handshaking
  (datahandshake=0 except for burst w/reqdata_together=1, writeresp_enable=0,respaccept=0, cmdaccept=0, dataaccept=0)
- Ports may be advanced by a write operation on thread 1, unless it is implied by using the FIFO mode above (zero address width, and threads=1)
  (threads=2)
- Container will support workers written to allowable variations, implying generated or library-based "couplings"

6.3.3 OCP Constraints for Worker as OCP slave for FIFO input ports

Input ports may be implemented as OCP slaves when there is no requirement for the worker to randomly address the input data. In this case it is as described in the FIFO case above with the following exceptions:

- No explicit port advancement
- No precise bursts
- No operation ordinals, single-operation IDL assumed
- Write operations only

6.3.4 Port operations

Worker ports shall use standard OCP read and write operations, creating messages formatted as defined in requirement 4 above.

6.4 RPL LOCAL SERVICES

For RPL environments, the list of “local system services” is short:

- *Reset services*
- *Clock services*
- *Local memory resources (e.g. low latency/SRAM and large/DRAM types)*

The reset service is implemented by the OCP *Mreset_n* in the *Worker* interface using the "mreset" configuration parameter. This signal is not configured in any of the port interfaces.

Memory services are supplied as OCP slave ports, with broad configurability due to the wide range of memory usage scenarios (especially due to high performance memory requirements).

Requirement 9 RPL worker clocks

RPL containers shall support the following clock usage cases, and workers may choose from the following clock configurations:

- *Single container-supplied clock, Clk signals for all ports are one clock, from the Worker interface.*
- *Independent container-supplied clocks, Clk signals on all ports may be different. Workers must assume they are different.*
- *Independent worker-supplied clocks, Clk on Worker interface is from container, all others are independent. Containers must assume they are different.*

[TBD: clock scheme selection indication by worker]

Requirement 10 RPL memory ports

RPL containers shall support the following memory interface, as a slave, for workers that require memory resources:

6.4.1 Multiple ports for highest memory performance

The worker may implement two independent OCP interfaces for memory access, to fully exploit memory technologies that provide for simultaneous read and write access with different addresses (like on-chip BRAM or some modern SRAMs). When a worker chooses this option, one port will be read-only and one will be write-only. This is termed the "double port" option, vs. the "single port" option. [In general, should worker have more than one memory interface to support multiple memory resources, like TI XDAIS?]

6.4.2 OCP Constraints

- Worker memory port is named in the OCP configuration file by the interface_name parameter as memory_N, where N is the number associated with the requested memory port.[to support more than one, as in XDAIS].
- Worker is master, container is slave
  (OCP configuration parameters in parenthesis)
- Input port of double port option only supports read operations.
  (read_enable=1, write_enable=0, sdata=1, mdata=0)
- Output port of double port option only supports write operations.
  (read_enable=0, write_enable=1, sdata=0, mdata=1)

- Single port option supports read and write operations
  (read_enable=1, write_enable=1, sdata=1, mdata=1)
- Data width is 16, 32, or 64, byte enables are optional
  (data_wdth=16, 32, or 64, byteen=0 or 1)
- Address width is 32
  (addr_wdth=32)
- Workers may use aligned, precise, single request, incrementing bursts on input or output
  (burstprecise=1, burstsinglereq=1,burstlength=1,reqdata_together=1)
- Workers may use request pipelining (cmdaccept=1, OCP default) and/or data pipelining
  (datahandshake=1, OCP default is 0) to hide latencies and optimize throughput
- No threads
- Container will support workers written to allowable variations, implying generated or library-based "couplings"

7 SHP PLATFORM REQUIREMENTS

7.1 CONTAINER INTEROPERATION REQUIREMENTS

Platforms (a.k.a. JTR sets) containing multiple SHP and GPP devices and containers must ensure IDL-defined inter-component communication works in cases where the communicating components are:

- Collocated in any class of container
- Located in different containers of the same class
- Located in different containers of different classes.
- Of course a platform only needs to worry about the actual container in that platform.

7.2 HOW PLATFORMS ENSURE COMPONENT INTEROPERABILITY

7.2.1 GPP Container interoperability

The SCA defines the interactions between components to be IDL-based, but in the case of non-GPP containers, the interactions should not necessarily be CORBA-based. In the GPP environment, a CORBA IDL compiler is used to read IDL and generate CORBA interface artifacts (called "stubs" for the client/user and "skeletons" for the server/provider). These artifacts are what the component code actually calls (on the client/user side) and what calls the component (on the server/provider side). These artifacts collaborate with the local CORBA library (called an ORB, in the GPP container) to encode, send, receive, and decode messages between GPP components. They are specific to the implementation of CORBA (the ORB), and not considered portable code in any sense.

Most such ORBs have a special optimization to notice when the client and server are collocated and thus remove the "encode/marshal, send, receive, decode/unmarshal, dispatch" overhead and use a local functional call instead. CORBA IDL compilers that generate these stubs do not have access to the configuration of components that are being executed together in a container. Thus, the generated artifacts must be prepared for non-collocated as well as collocated cases, even when the configuration may never need it.

The platform must allow IDL-defined messages (a.k.a. requests/replies, invocations, etc.) to flow between components in different containers. While this requires that the container implementations in the platform know how to talk to each other, it does ***not*** imply that these mechanisms (middleware or hardware machinery) need to be standardized. The SCA requires that CORBA be used within and between GPP containers, but does not specify any communication machinery or hardware or ORB or transport API.

7.2.2 Non-GPP Container interoperability

This extension only requires that the various containers in a platform know how to convey messages between portable components that are written to use portable APIs. No further mandatory mechanisms are needed to achieve portability. This is shown in the diagram below, where the black arrows represent the logical communication between components, the orange arrows shows where the components actually communicate with their containers via specific interfaces, and the blue arrows show how containers in the platform communicate with each other to effect the inter-component communication. The blue arrows are ***NOT*** relevant to component portability, but represent the required communication inside the platform. The HAL-

C APIs address interfaces that might be used in the blue arrows, but still does not address actual protocols or busses.

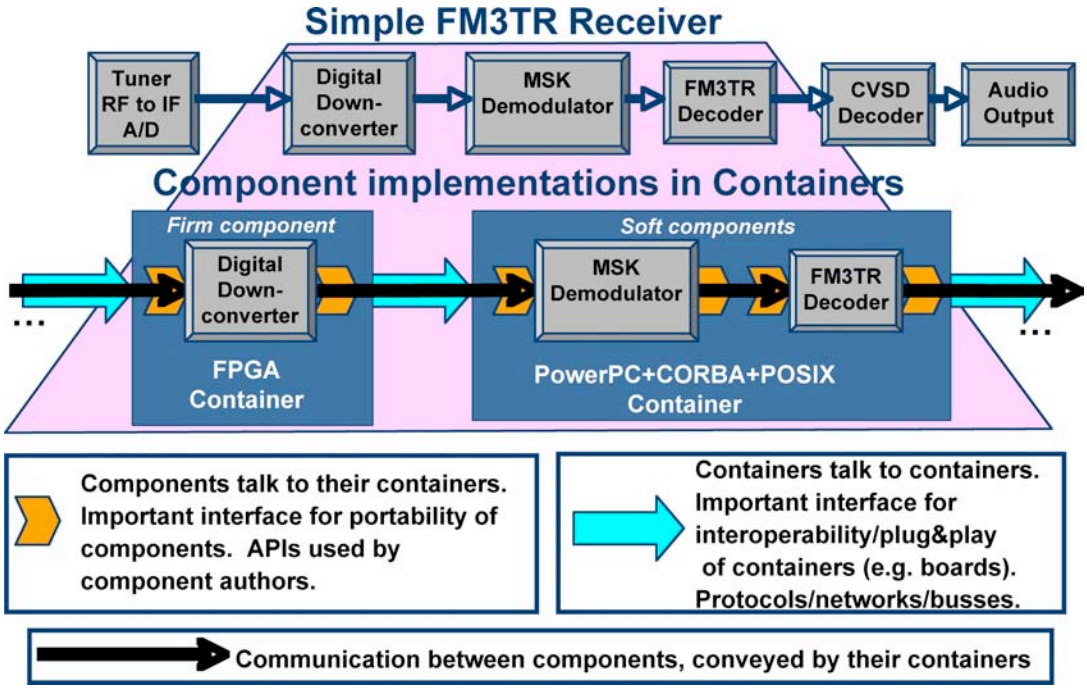

Figure 6 Container Interoperability

There are four cases of inter-container interoperability that must be supported;

- Port is for SHP component port, "other side" is GPP/CORBA component port.
- Port is for GPP/CORBA component port, "other side" is SHP component port.
- Port is for SHP component port, "other side" is same type of device component port.
- Port is for SHP component port, "other side" is different type of SHP comp port.

Within a container (collocated case), the container is simply responsible for providing local inter-worker interoperability. Similarly, between two containers of the same type, connected by some hardware connection path, the messages/frames flowing between the ports of workers in different containers is entirely managed by the implementer of that type of container.

When the inter-container communication flows between different types of SHP containers, the message/frame formats should be made compatible between those container implementations for maximum performance and minimum overhead. The platform implementer has control over all container implementations in the platform, so this uniformity is easily achieved.

When messages are flowing between GPP components and firmware components, then they are flowing between a CORBA environment (via the stubs and skeletons provided by the ORB), and the firmware container, where they are sent/received at the worker's port.

While the message format in a buffer visible to the workers is defined in this section (the mapping of the subset-IDL interface operation to a simple data structure), the format at the CORBA side is not defined anywhere in the SCA (and does not have to be). The standard

CORBA-defined message formats (in the GIOP and CDR specifications) only applies to interoperation between CORBA implementations, and thus does not apply in this case.

The requirement is that the message formats defined for worker ports interoperate with the generated stubs and skeletons of the GPP components. There are a variety of techniques to accomplish this interoperability, but it is out of the scope for this specification.

8 REALIZATION

8.1 FEASIBILITY

8.1.1 RCC Workers and Containers

Previous prototype systems.

8.1.2 RPL Workers and Containers

OCP adoption and maturity, FPGA hardware abstraction and infrastructure products.

8.1.3 Intercontainer Message Interoperability

Associating IDL with connections, object references

8.1.4 Generic Proxies, Worker Properties

8.2 IMPACT

Very minor, backward compatible, SCA Core Framework enhancements are required.

Existing algorithm implementations can be wrapped to implement standard worker interfaces.

RCC container suppliers must support RCC workers, but since the execution model can be supported in a wide range of DSP environments, including simple single stack configurations, the implementation effort should be modest. Legacy interconnect structures and drivers are efficiently adapted.

RPL container suppliers must support a range of OCP configurations for the various worker interfaces (Worker, ports, memory), and clock options (single container-supplied, multiple container-supplied, multiple worker supplied). Many of these can come from libraries of "couplings" from various OCP configurations to the native memory and/or communication interfaces. New combinations, if not already implemented, are very well defined and easily added.

Platform suppliers need to provide message adaptation for communication paths between containers to provide IDL-based message interoperability.

8.3 INFOSEC CONSIDERATIONS

TBS

9 APPENDIX A: CHANGE PROPOSALS FOR THE SCA.

9.1 CHANGE PROPOSAL FOR CLARIFYING IMPLEMENTATION DEPENDENCY

9.1.1 Title

Clarify the semantics of the handling of softpkgref dependency elements.

9.1.2 Reference

SCA Appendix D, section D.2.6.9, and D.2.6.9.1

SCA Section 3.1.3.2.2.5.1.3

9.1.3 Description

In SCA Appendix D, section D.2.6.9, and D.2.6.9.1 specifies that a softpkgref subelement of a dependency element indicates a "file load" dependency, which "must be resident" for the component with the dependency to "load without errors".

In SCA Section 3.1.3.2.2.5.1.3, which describes how the ApplicationFactory's create operation says that 1) it "shall use the SAD SPD implementation element to locate candidate devices capable of loading and executing Application components", 2) it must "load the Application components (including all of the Application-dependent components)" and 3) it must "execute the Application components (including all of the Application-dependent components)".

The text should clarify that locating devices on which to load and execute components and dependent components is a recursive process that requires each component to be loaded on an appropriate device before the dependent component is loaded, and that the dependency should be executed before the execution of the dependent component. The fact that a dependency does not have any collocation requirement with the component depending on it should also be made clear.

Appendix D does not specify "execution" of dependent components, but section 3.1.3.2.2.5.1.3 does. Appendix D should be consistent with section 3.1.3.2.2.5.1.3 in this regard, and state that there is both a loading dependency as well as an execution dependency (as section 3.1.3.2.2.5.1.3 already does).

9.1.4 Recommendation

In section 3.1.3.2.2.5.1.3 change the phrase

"shall use the SAD SPD implementation element to locate candidate devices capable of loading and executing Application components"

to

"shall use the implementation elements in each Application component's SPD to locate one or more candidate devices capable of loading and executing one or more of the implementations, including, recursively, any SPDs referred to via softpkgref elements of the dependency elements of the implementations"

Change the phrase (twice):

"(including all Application-dependent components)"

to

“(including the components each Application component depends on, recursively)”

In section D.2.6.9, change:

“indicates a file-load dependency on that file”

to

“indicates that the dependency SPD should be loaded before this implementation, and, if the dependency requires execution, that it be executed before this SPD is executed.”

9.1.5 Rationale

The handling of dependencies is inconsistent and unclear. This clarifies it, and in particular clarifies the important use case of software proxies and adapters for specialized implementations acting as alternative implementations in the same SPD.

9.2 CHANGE PROPOSAL FOR DEPENDENCY HANDLES

9.2.1 Title

Add an option for a softpkgref dependency to provide a handle to the dependent component implementation.

9.2.2 Reference

SCA Appendix D, section D.2.6.9.

9.2.3 Description

There is currently no way for a component to be given a handle to the components it depends on. An optional attribute to the dependency *softpkgref* element should provide the name of an execution parameter to use to communicate the reference of the executed dependency to the dependent component.

When the dependency SPD implementations have *code* types of shared library (with entrypoint) or executable, it implies the use of the execute operation of an executable device, but there is no way to communicate any identity or handle of the component being created back to the component that depended on it to be created.

9.2.4 Recommendation

Add the paragraph:

“If the *dependency softpkgref* element of an implementation contains a *dependencyid* attribute with string value, then an execution parameter with the name equal to the value of that attribute, shall be added when the dependent component is executed. The value of that added execution parameter shall be the reference produced by executing the dependency component (based on a *softpkgref*). The ApplicationFactory obtains the dependency reference using the same NAMING_CONTEXT_IOR and NAME_BINDING execute parameters as it uses for application components referenced directly from the SAD. The NAME_BINDING value is implementation defined. The presence of the *dependencyid* attribute implies that dependency component requires execution and that its execution will produce a reference and support the name binding execution parameters.

9.2.5 Rationale

This is a simple feature that allows dependencies to have identities provided to the dependent component. When a component depends on another component, it is told just which thing it got in response to that need. Providing the simple “connectivity” in this way allows alternative implementations in an SPD to have different such dependencies without affecting the SAD.

9.3 CHANGE PROPOSAL FOR MAXIMUM PROPERTY SIZES

9.3.1 Title

Add an optional "max_string_size" attribute for simple and simplesequence property elements, and a “max_sequence_size” attribute for simplesequence and structsequence property elements, consistent with maximum string length and maximum sequence length features that are in IDL.

9.3.2 Reference

SCA Appendix D, section D.4.1.1, D.4.1.2

9.3.3 Description

An optional max_string_size attribute is added to the ATTLIST of the "simple" and “simplesequence” elements and an optional max_sequence_size attribute is added to the "simplesequence" and “structsequence” elements to provide the same type description capability as in IDL.

9.3.4 Recommendation

Add the line:

max_string_size CDATA #IMPLIED

after the line:

name CDATA #IMPLIED

in sections D.4.1.1 and D.4.1.2 (simple and simplesequence)

Add the following sentence to end of the first paragraph of D.4.1.1 and D.4.1.2:

The optional *max_string_size* attribute is a positive integer indicating the maximum allowable size when the type attribute is *string* or *objref* (as in IDL)

Add the line:

max_sequence_size CDATA #IMPLIED

after the line:

name CDATA #IMPLIED

in sections D.4.1.2 and D.4.1.5 (simplesequence and structsequence)

Add the following sentence to the end of the first paragraph of D.4.1.2 and D.4.1.5

The optional *max_sequence_size* attribute is a positive integer indicating the maximum allowable size of the sequence (as in IDL).

9.3.5 Rationale

There is currently no way to specify in simple properties or simplesequence properties, the maximum size of the string or sequence, which prevents specifying maximum sizes as you can in IDL. This addition allows software that is handling properties to compute the maximum memory required to hold property values, and to create linear data structures to hold properties.

9.4 CHANGE PROPOSAL FOR PORTABLE COMPONENTS FOR SPECIALIZED PROCESSORS (HIGH LEVEL THAT INCLUDES THE PREVIOUS ONES).

9.4.1 Title

The referenced document contains a specification that augments the core SCA and defines how waveform components written for resource constrained C-programmable processors (e.g. DSPs) and RTL-programmable logic processors (e.g. FPGAs and ASICs). These specialized processors are collectively referred to as SHP (Specialized hardware Processors).

9.4.2 Reference

[document reference]

9.4.3 Description

(This description is from the introduction section of the referenced document)

This document defines a set of requirements that can be placed on certain JTRS waveform components to maximize their portability. Acquisitions may or may not require this level of portability. It addresses components that cannot use the portability requirements defined in the section 3.2 of the SCA core specification (SCACS), since that section addresses components written for feature-rich general purpose software environments that assume 1) the availability of CORBA, 2) significant amounts of the POSIX OS standard, and 3) general purpose object oriented programming languages mapped to CORBA. The software components addressed in section 3.2 of the SCACS are hereafter termed GPP components, for General Purpose Processing.

In contrast to the discussion of GPP component requirements in section 3.2 of the SCACS, this document defines portability requirements for waveform components targeting typical DSP, FPGA and ASIC environments. Although some DSP environments are rich enough to be treated as general-purpose environments as addressed in SCACS section 3.2, this document addresses more limited and more typical DSP environments. The components addressed in this section are heretofore termed SHP components, for Specialized Hardware Processing. The scope of SHP could naturally include resource-constrained processors such as microcontrollers, and programmable logic processors not fully captured by "FPGA" or "ASIC" terminology.

Thus the purpose of this document is to enable portability requirements to be placed on DSP, FPGA, and ASIC source code in a way consistent with, and analogous to, the portability requirements that SCACS section 3.2 defines. This increases the portability of waveforms in general since it increases the functionality and source code that can be subjected to portability mandates in procurements. A related purpose of this document is to facilitate the replacement or migration of components in waveforms among the different types of processing environments without affecting the waveform as a whole, or other components in the waveform. This reduces the effort of technology insertion and migration when deployment requirements change or new

technology emerges: it enables multiple compatible, replaceable implementations of the same component functionality, targeting either GPPs or SHPs.

While addressing SHP component portability in general, this document addresses DSP components separately from FPGA/ASIC components due to differences in their common source languages (C, VHDL/Verilog etc.), operating environments and development processes. For brevity the term FPGA is hereafter used to refer to both FPGA and ASIC components. There is significant commonality in concept and architecture between the treatment of FPGA and DSP components, and many requirements termed SHP requirements apply equally to both.

This document purposely does not define any interoperability facilities like CORBA's GIOP. Providing working connections between processing devices is considered the integrator's responsibility. The reason for this is to enable preexisting hardware and software interprocessor messaging mechanisms to be leveraged while complying with this specification.

The referenced document enumerates specific goals for full portability and interoperability of these SHP components with existing SCA components written using CORBA.

9.4.4 Recommendation

The recommendation is to add the referenced specification as an extension to the SCA that can be referenced by acquisitions to require portable components.

9.4.5 Rationale

Significant functionality of waveforms is currently outside the scope of the portability mandates contained in the existing SCA specification (2.2.1). This specifications enables SHP components to have a level of portability similar to the portability of the CORBA-enabled components running on general purpose processors (or DSPs when they are fully CORBA-enabled).

This can significantly enhance the overall portability and reusability of waveforms, which will further accomplish the core goals of the JTRS program.

10 APPENDIX B: CUSTOM PROXY POSSIBILITIES

10.1 SHP CUSTOM PROXY

SHP custom proxies are optional implementation elements. An SHP component implementer creates a SHP custom proxy only for components whose specific interface and/or behavior is specialized beyond the generic *CF::Resource* behavior. SHP custom proxies act like normal GPP software components. The custom proxies embody the knowledge of how to translate specialized and generic configuration and control operations to the fixed and more primitive control and configuration model represented by *CF::Resource*. The configurable properties of a custom proxy may be different from those of the underlying SHP component.

The custom proxy code is represented by one of the implementation elements in the SPD for a given SCD. It implements the specific interface for the component (which inherits the *CF::Resource* interface). It is a GPP component that uses CORBA normally. Its implementation element in the SPD has a dependency on the underlying SHP worker implementation (e.g. FPGA code). This causes the CF::ApplicationFactory to load and execute the worker using the SHP LogicalDevice, prior to loading and executing the custom proxy, using a GPP LogicalDevice.

The custom proxy receives the object reference to the generic proxy by setting the *dependencyid* attribute in the *softpkgref* element of its *dependency* element in its SPD. The value of this attribute is the name of an execparam that the ApplicationFactory will use to pass the generic proxy's reference to the custom proxy. This is based on the small CP in section 9.2 above.

The custom proxy looks and acts like any other GPP component to the SCA core framework and assembly controllers, and thus enables the replace-ability goal mentioned above. It is similar to the adapter concept described in the SCACS, but its purpose is only to adapt the *specific* interface, not necessarily any of the *port* interfaces. Port communication is typically delegated to the worker for performance reasons. Such intercomponent communication via port interfaces is *not* necessarily touched by the custom proxy.

A simple example of a custom proxy would be to translate a property setting convenience operation (e.g. "set_frequency") into a string-based property setting to the underlying generic proxy using the *configure* operation of the CF::Resource interface.

A custom proxy is written knowing the particular property set of the underlying SHP implementation, but not knowing anything specific to the SHP logical device or container that the worker is executing in. Thus an SHP custom proxy can be reusable across multiple worker implementations that implement the same property (configuration) interface. When the specific interface and properties of a component are simple, the custom proxy is simple. If collocated with the SHP LogicalDevice it can be very efficient.

When a custom proxy exists, the worker's SPD is a *dependency* of the SHP custom proxy (a GPP component). Thus there is no confusion on the part of the ApplicationFactory for the generic proxy object reference. When there is a custom proxy, the ApplicationFactory and assembly controller use that as the component's CF::Resource. When there is no custom proxy, they simply use the generic proxy as the component's CF::Resource.

Thus GPP implementations, SHP implementations with custom proxies, and SHP implementations without custom proxies, can all coexist in the same SPD, transparent to the SAD and assembly controller.

Collocating the custom proxies with the logical devices and the generic proxy can easily eliminate the "extra level of CORBA indirection" when custom proxies are needed.

The three parts of SHP Component Implementations (custom proxy, generic proxy and worker) are shown in the following 2 diagrams, which show the case that includes the optional custom proxy:

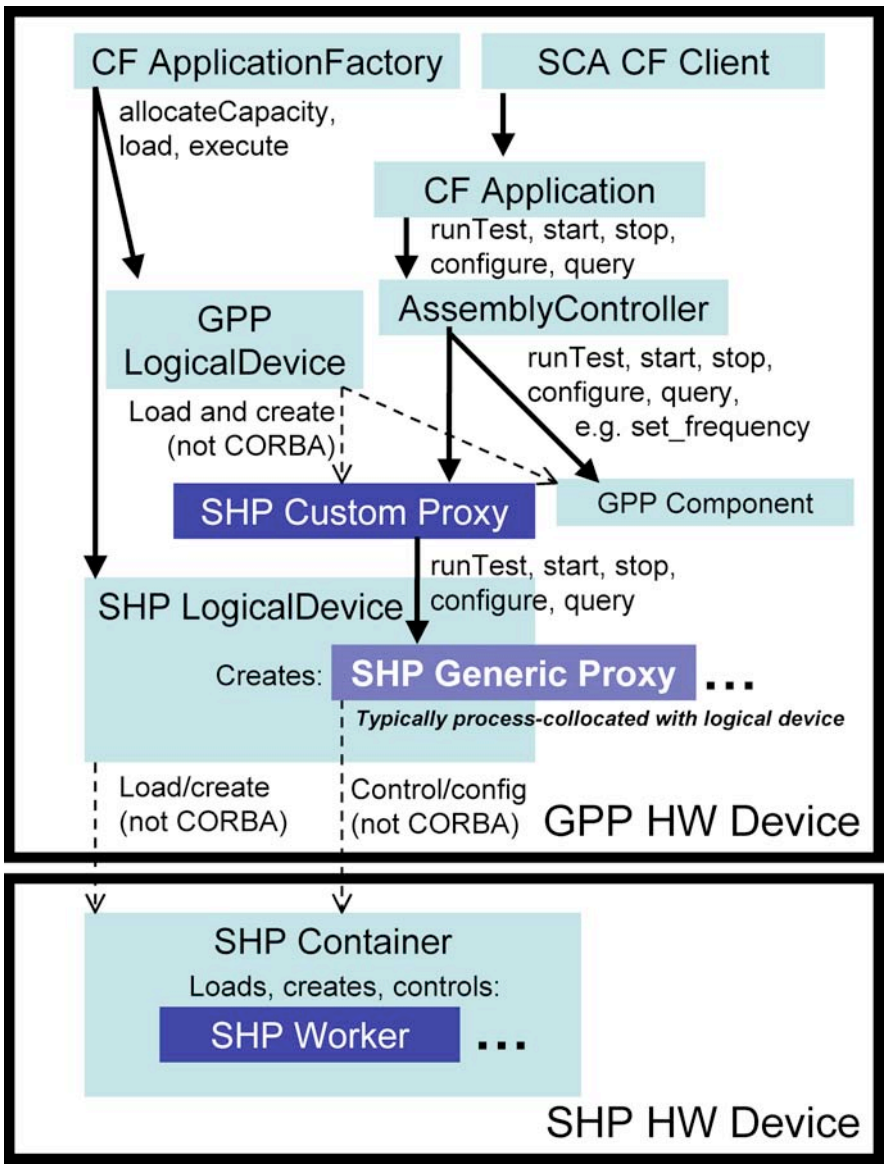

Figure 7 SHP Component Implementation Relationships with Custom Proxy

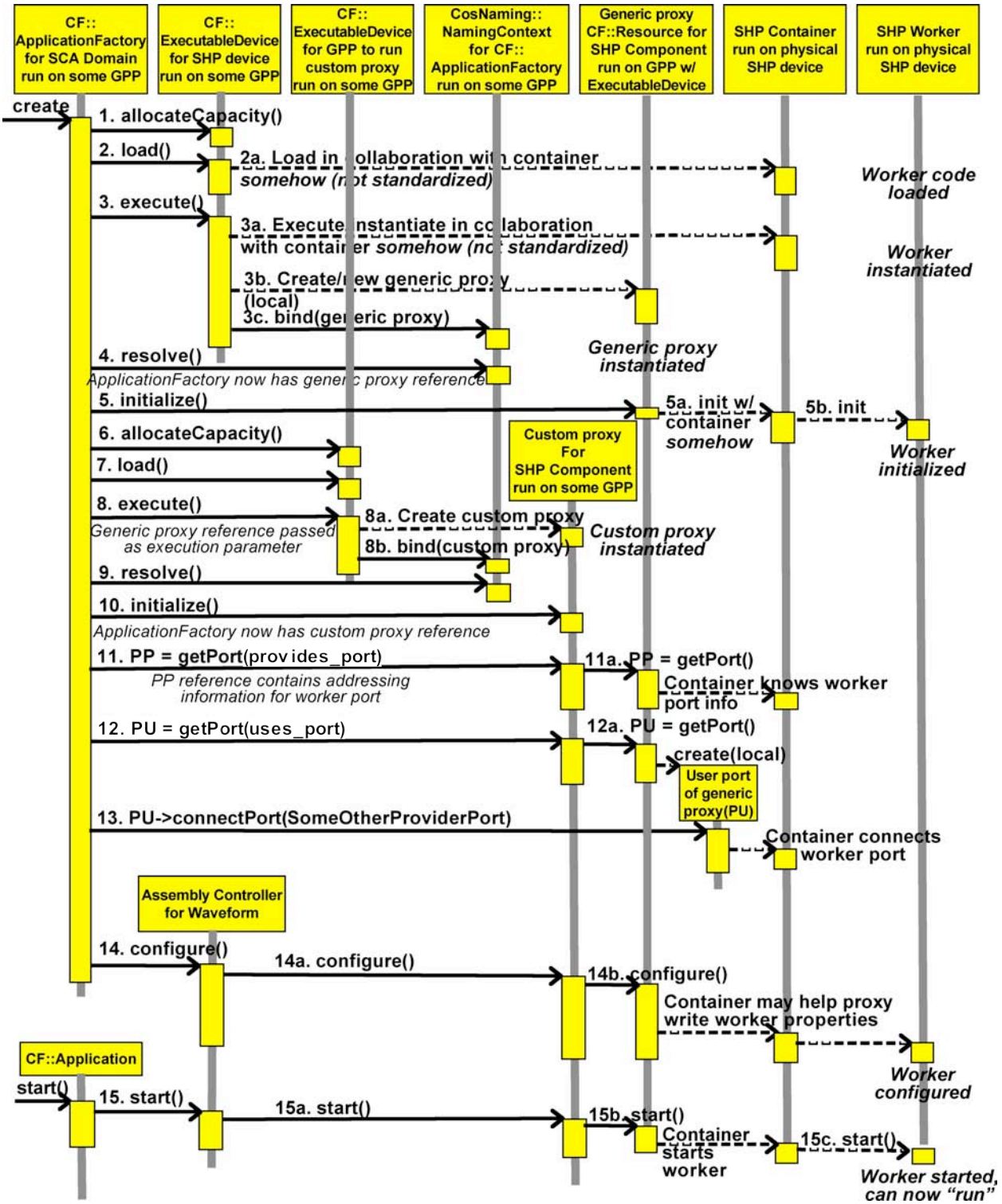

Figure 8 SHP Component Implementation Sequence Diagram with Custom Proxy

For a component implementation with a custom proxy, that implementation's executable code file implements the SHP custom proxy for that component. This SHP custom proxy

implementation element contains an execution dependency on an SPD that specifies the worker implementation. This relationship is depicted below:

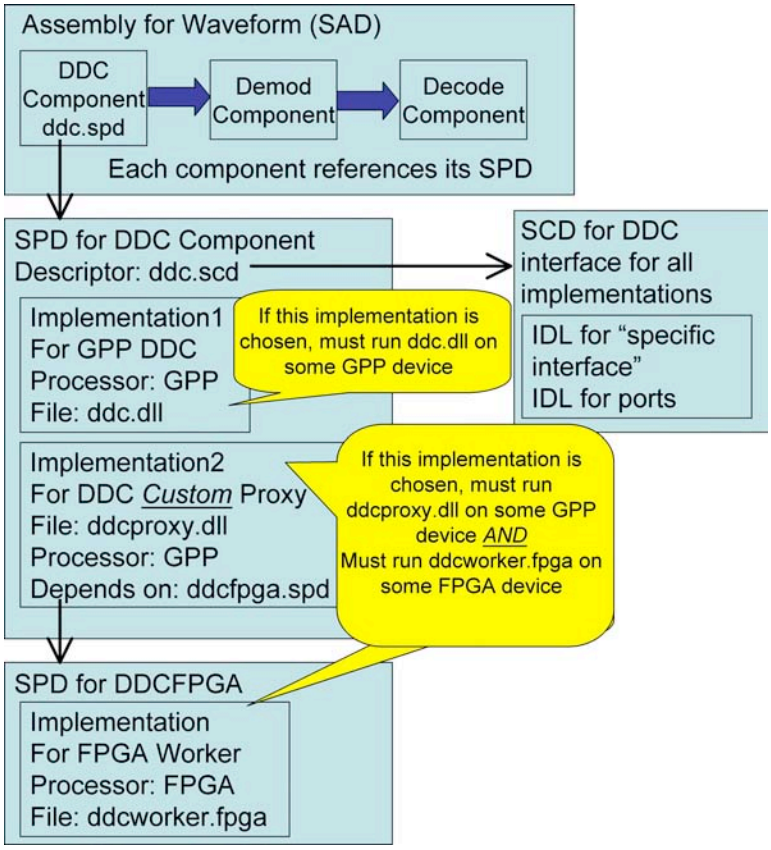

Figure 9 Assembly with DDC Component with SHP implementation with custom proxy

As described above, the SHP custom proxy supports the specific interface described in the SCD of the SPD referenced by the SAD. The worker, with help from its container, supports and uses the port interfaces described in the worker's SPD/SCD. Normally, these port interfaces are identical to those in the custom proxy's SCD: all port communication is fully delegated to the worker. There are other special cases where the custom proxy may implement some ports directly, and thus only delegate some ports to the worker, in which case the worker's SCD would have a subset of the ports in the custom proxy's SCD.

There can also be private communication established between custom proxy and worker, unknown to the application factory and SAD. A custom proxy can use the generic proxy's PortSupplier interface any way it wants. Normally it would simply delegate all port operations to the generic proxy. [Show an example, e.g. where custom proxy has two inputs, A, and B, and one output C, where it delegates port A and B to the worker, but creates an internal connection with the worker to intercept the worker's C, and provide some transformation, and implement its own C output]

The specific interface of the custom proxy is no different from normal GPP-implemented components, except that, as mentioned earlier, the SHP custom proxy must implement the

specific interface using only the underlying SHP generic proxy's *CF::Resource* interface to control the worker.

Thus, the optional custom proxy implements the *specific* interface, which inherits the SCACS's standard *CF::Resource* interface.

The custom proxy uses a *CF::Resource* interface to talk to the LogicalDevice software (using the generic proxy) about a specific worker executing in the container. It supports the custom proxy in translating interface specific operations to a more primitive, and efficiently implemented interface.