Component Definition Document (CDD) for the SignalProcessing Example Component Assembly

Rev. A

February 27, 2010

Prepared By:

Northrop Grumman Corporation

Electronic Systems

Baltimore, MD

Table of Contents

1 Introduction

1.1 Scope

This document captures the specification and design for the Signal Processing Example software component assembly. This example assembly is targeted for deployment on the Scalable Node Architecture (SNA) real-time component framework. As such, it must be compliant with SNA Component Based Architecture (CBA) design guidelines.

This specification defines the component assembly's functional, interface and performance requirements, the context in which it must operate, and any design constraints it must adhere to. It provides criteria for verifying compliance, but it does not state methods for achieving results.

This is intended to be a relatively informal living document, to be included in same CM repository and package as the component source code. This CDD will initially be populated by a system engineer or software architect/lead to define component design constraints & guidelines. Over time, it will transition to enhance the "to be built" specification sections with "as built" design information documenting the final component product.

2 Applicable Documents

2.1 Applicable Government Documents

Document No.	Title

2.2 Other Applicable Documents

Document No.	Title
	SNA Signal Processing Users Guide

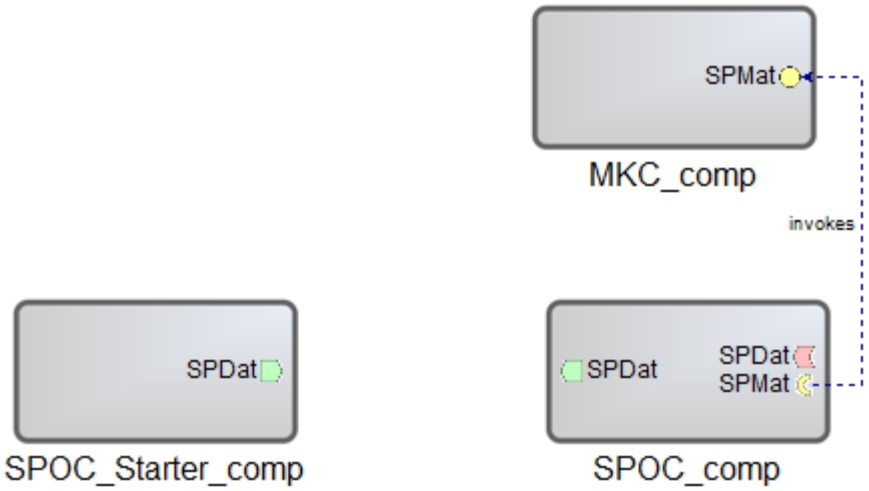
3 Component Description

3.1 Overview

The Signal Processing Example component assembly is one of the component source examples included in the SNA SDK for reference, testing and experimentation. It illustrates a design pattern where one component handles the data movement and orchestration via publication and subscription mechanisms and another component performs the actual signal processing math, which is invoked via a request/reply interface . These two components are generally referred to as the Signal Processing Orchestration Component (SPOC) and Math Kernel Component (MKC), respectively. Separating the data movement from the math operations is desirable because it leads to a design that is more modular with a higher potential for component reuse. In addition to the SPOC and MKC, this example contains a third component called the

SPOC_Starter, which simply creates some test data and publishes it to kick off the example. The assembly containing these three components is shown in below.

Figure 3-1 – Signal Processing Example Component Assembly



3.2 Operational Context

This simple assembly is completely self contained and has no external connections. It is designed to operate solely within the constraints of the SNA SDK development environment to allow a new user/developer to step through the SNA component based development (CBD) process of loading a component assembly into the SNA IDE, building/compiling it, and then executing it.

The example is provided with an appropriate set of SNA configuration files and a deployment plan to support its execution within a single-host SNA SDK "localhost" Virtual Machine (VM). Alternative variations on the default supplied design and deployment are possible via experimentation by a software developer.

4 Component Interfaces

The Signal Processing Example assembly defines a single internal connection between a "required" client "receptacle" port on the SPOC monolithic component and a "provided" service "facet" port on the MKC monolithic component, as shown in . The component assembly has no external interfaces.

4.1 Service Ports

4.1.1 SPMathFacet Service (internal)

This service interface provided by the MKC monolithic component defines three methods: `init`(), `compute`(...), and `cleanup`(). `init`() and `cleanup`() take no arguments and serve merely as exemplar methods that a mode designer might want to incorporate into their design.

The compute method contains the real meat of the MKC and is invoked by the SPOC when it's time to perform the signal processing math. Logically speaking, this method takes input and output data (in the form of VSIPL++ data objects) by reference and uses those data objects when performing the SP calculations. There is also an argument for 'SP Controls', which would typically be some custom data structure that contains dynamic information required by the MKC to perform its task. In this example, the SP controls structure is simply a single integer and is not actually utilized by the MKC.

In reality, one cannot simply declare a VSIPL++ data object as an argument in an interface definition because it's not a valid IDL type. To get around this, VSIPL++ data objects are passed as two separate pieces: a structure containing metadata describing the object (number of dimensions, complex format, data type, etc) and the underlying data itself. Further details about this topic are available in the Signal Processing Users Guide.

4.2 Client Ports

4.2.1 SPMathRecept Client (internal)

This client interface on the SPOC monolithic component is used to call the `init`(), `compute`(...), and `cleanup`() methods provided by the SPMathFacet service.

4.3 Publisher Ports

4.3.1 SPDataPub (internal)

The SPDataPub port on the SPOC_Starter monolithic component publishes test data in the form of a VSIPL++ data object.

4.4 Subscriber Ports

4.4.1 SPDataSub (internal)

The SPDataSub port on the SPOC monolithic component subscribes to the test data that is published by the SPOC_Starter component.

5 Component Functionality

At startup, the SPOC_Starter component sets an SNA timer (per the SNA Time Management API) to generate a one-shot timer event that fires 10 seconds after startup. Upon timer expiration, the SPOC_Starter creates a sample data set, outputs the data to the log and terminal consoles, and publishes it.

The SPOC receives this sample data and outputs it to the console. It then creates a VSIPL++ data object to hold the output. The `init`() method in the MKC is invoked by the SPOC. Next, the `compute`(…) method is invoked. Within this method inside the MKC, the input and output data objects are again output to the console to ensure correctness. The MKC then performs some simple math, adding std::complex<float> (10, -10) to each element in the input data set to form the output data. After completing `compute`(…), the MKC's `cleanup`() function is then invoked. Finally, the SPOC logs the output data to the console. In a real application, this is the point where the SPOC would generally publish this output data. In this simplistic example, the data is not published and the application enters an idle state.

Per the default deployment plan provided with this example, the SPOC and MKC components will execute in the same process/container, while the SPOC_Starter will execute in a separate processes/container – and all three components will run on the same host. Alternative deployments wherein the two processes run on different hosts, or wherein all three components are locally collocated within the same process/container are also possible. No code changes are required in either case.

6 Configurable Parameters

Since this component assembly is intended to support run-time experimentation, it is also packaged with a full set of SNA compliant run-time execution configuration and deployment files. Changes can be made that affect logging, thread priority, processor affinity, and the names of messages that the components publish and subscribe to – all of which can be modified via configuration (.cfg) files. This example does not use any Attributes, which are set using the deployment plan itself.

7 Design Constraints

1. An assembly of two components will be used to perform basic signal processing functionality whereby one component handles data publication and subscription and the other performs the math, which is provided via a request/reply style interface.
2. An MKC component will provide an implementation of the `compute`(…) method on an SPMathFacet service port and output a log message to indicate each call made to it.
3. The Signal Processing Example component assembly will follow established naming conventions and code organization guidance defined in the SNA SDK documentation, such that new SNA software developers can use it as a design reference.
4. A full set of SNA compliant configuration and deployment files will be provided with this example in order to support run-time execution in a default single-host target deployment.
5. This example will utilize the standard SNA APIs to perform all functions.

8 Component Test

This 3-component assembly must be executable on a single "localhost" development computer, to include the SNA SDK x86-64 VM at a minimum. Users can experiment with the deployment plan to redeploy to an alternative 2-host target environment if desired.

9 Component Dependencies

The Signal Processing Example assembly is self-contained and has no dependencies on any other application components. Its only dependency is on the SNA SDK's run time execution environment.

10 Notes

The SPOC_Starter creates a sample data set of type std::complex<float> and stores the data in complex-split format. The current version of VSIPL++ (Sourcery VSIPL++ 2.4) provided with the SNA SDK has a native complex format for each type of computer architecture, which happens to be interleaved for x86 and split for PowerPC/Cell. If user-defined data is admitted to VSIPL++ in a non-native format, the library will perform an internal copy to put the data into the native format. Upon release from the library, the data will be restored to its original state. As the Signal Processing Example uses complex split format, running on an x86 architecture will result in internal copying of user data. The application will function correctly, but with diminished performance. A user will only notice this if they are using timers trying to analyze the performance of the application.