Design and Implementation of an Efficient SCA Core Framework for a DSP Platform

WAEL A. MURTADA
Satellite Communications and Ground Stations Department, Space Sciences and Strategic Studies Division
National Authority for Remote Sensing and Space Sciences
Cairo
EGYPT.
Email: wael_murtada@narss.sci.eg
Website: http://www.narss.sci.eg
MOHAMED M. ZAHRA
Communications and Electronics Engineering Department, Faculty of Engineering
Al-Azhar University
Cairo
EGYPT.
Email: mzahra15455@gmail.com
MAGDI FIKRI
[3]Communications and Electronics Engineering Department, Faculty of Engineering, Cairo University
Cairo
EGYPT.
Email: magdi.fikri@gmail.com
MOHAMED I. YOUSEF
Communications and Electronics Engineering Department, Faculty of Engineering
Al-Azhar University
Cairo
EGYPT.
Email: drmiyoussef@yahoo.com

SALWA EL-RAMLY
Communications and Electronics Engineering Department, Faculty of Engineering, Ain Shams University
Cairo
EGYPT.
Email: sramlye@netscape.net

Abstract: The Software Communications Architecture (SCA) was developed to improve software reuse and interoperability in Software Defined Radios (SDR). However, there have been performance concerns since its conception. Arguably, the majority of the problems and inefficiencies associated with the SCA can be attributed to the assumption of modular distributed platforms relying on General Purpose Processors (GPPs) to perform all signal processing. Significant improvements in cost and power consumption can be obtained by utilizing specialized and more efficient platforms. Digital Signal Processors (DSPs) present such a platform and have been widely used in the communications industry. Improvements in development tools and middleware technology opened the possibility of fully integrating DSPs into the SCA. This approach takes advantage of the exceptional power, cost, and performance characteristics of DSPs, while still enjoying the flexibility and portability of the SCA.

This paper presents the design and implementation of an SCA Core Framework (CF) for a TI TMS320C6416 DSP. The framework is deployed on a C6416 Device Cycle Accurate Simulator and TI C6416 Development board. The SCA CF is implemented by leveraging OSSIE, an open-source implementation of the SCA, to support the DSP platform. OIS's ORBExpress DSP and DSP/BIOS are used as the middleware and

operating system, respectively. A sample waveform was developed to demonstrate the framework's functionality. Benchmark results for the framework and sample applications are provided. Benchmark results show that, using OIS ORBExpress DSP ORB middleware has an impact for decreasing Memory Footprint and increasing the Performance compared with PrismTech's e*ORB middleware.

Key-words: Software Communications Architecture (SCA), Software Defined Radio (SDR), Digital Signal Processors, Embedded Object Request Broker (ORB).

1 Introduction

The Software Communications Architecture (SCA) was developed by the Joint Tactical Radio System (JTRS) program of the US Department of Defense to standardize the development of Software Defined Radio (SDR) technology. The SCA was developed to enhance system flexibility and interoperability, while reducing development and deployment costs. Early implementations of SCA SDRs have struggled to meet performance, cost, size, and power requirements. Arguably, many of the these problems have their origin in the assumption of a modular and distributed platform based on General Purpose Processor (GPP) to perform all signal processing. In order to overcome these problems, it is necessary to make better use of specialized hardware optimized for signal processing. Digital Signal Processors (DSPs) are specialized microprocessors designed specifically for real-time digital signal processing. However, DSPs have been relegated as secondary elements in the SCA, requiring a Hardware Abstraction Layer (HAL) for connectivity. Ongoing improvements in development tools and middleware technology allow the implementation of SCA systems using only DSPs. By following this approach the flexibility and reusability brought by the SCA are complimented by the cost and power efficiency of DSPs. If taken to a logical extent, this approach could eliminate the need for a GPP on certain SDR implementations. In this paper we present the design and development of an SCA implementation for a homogeneous TI C6416 DSP platform [4].

2 System Architecture

The goal of this paper is to study the repercussions of implementing the SCA in an optimized DSP platform. Therefore, we aim to minimize, or eliminate, the use of GPPs for this implementation. We leveraged the existing implementation of MPRG's Open Source SCA Implementation::Embedded (OSSIE) [5], by porting it to the C64 platform. The system implements the SCA version 2.2 [6] in C++. Our development environment is TI Code Composer Studio running on a Windows PC. Most of the development is done using the Device Accurate simulator of the C6000. The final target platform is a C6416 board from Texas Instruments.

2.1 Software Architecture Elements

The general software structure can be seen in Fig. 1, showing the three different components of the SCA Operational Environment (OE): the Core Framework (CF), ORB middleware, and operating system. In this paper, we used OSSIE as the CF, ORBExpress DSP from Objective Interface Systems (OIS) as middleware, and DSP/BIOS as Real-Time OS. All of them are available commercially or as open source. Services (e.g. Log, Event, and Naming Services) are not considered in our current implementation. The DSP/BIOS is a scalable real-time multitasking operating system designed specifically for the TMS320 family of DSPs [4]. It is developed and maintained by Texas Instruments. The DSP/BIOS is built in modules, which allows developers to reduce the footprint to a minimum by only integrating the features that are strictly necessary for operation. It supports preemptive multithreaded operations thanks to a real time scheduler and provides

memory management modules for low overhead dynamic memory allocation. The DSP/BIOS is not a Portable Operating System Interface (POSIX) compliant, as required by the SCA, forcing a slight deviation from the specifications. The C6000 family of processors does not include a memory management unit [4]. The ORB used in this project is OIS's ORBExpress DSP C++ version for DSP. It is a very optimized and modular implementation of minimum-CORBA as standardized by the Object Management Group (OMG). However, ORBExpress DSP supports the Extensible Transport Framework (ETF), which allows custom transport plug-ins [3].

2.2 Platform

The target platform for this project is the TI C6416 development board from Texas Instrument [4]. This high performance board contains TI TMS320C6416T DSP. The system runs at 720 MHz and has 16 Mbytes of SDRAM memory and 1 Mbytes of internal memory. Only DSP are used for signal processing and framework functionality. Single TI C6416 DSP is a server and client node.

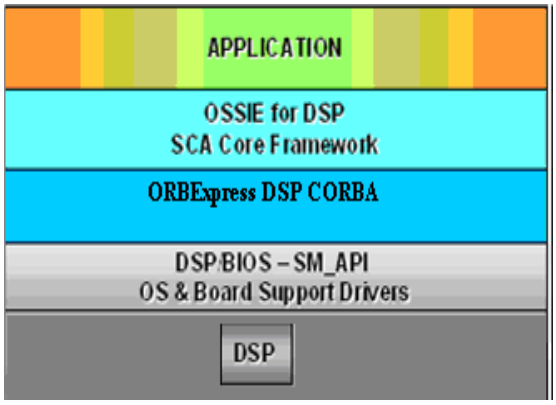

Fig.1: Software Structure.

3 Real-Time Implementation

The bulk of this project consists of porting the existing version of OSSIE to the C64 platform. The original OSSIE runs on an x86 platform running Linux as OS with *omniORB* as middleware. As with any other software project, development tools play a very important role. We use Code Composer Studio (CCS), an integrated development environment for TI DSPs, with version 6.0.8 of its Code Generation Tools. This particular version lacks the Standard Template Library (STL) and has limited support for C++ exceptions. The STL provides template classes such as Vector, widely used in the original OSSIE. In the absence of exception support, we use CORBA Environment variables coupled with a set of macros, distributed as part of ORBExpress DSP, for error handling purpose. These characteristics imposed significant changes in the original OSSIE source code.

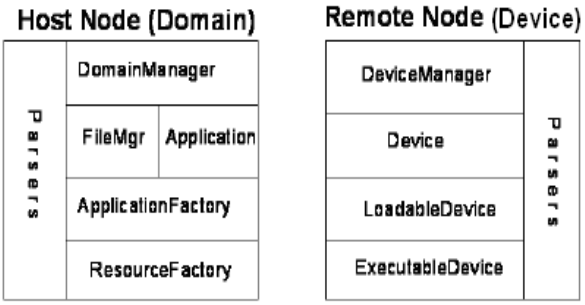

Fig. 2: Processing Node Deployment Scheme.

An important aspect in the development of this project is the lack of a Memory Management Unit (MMU) in the C64 [4]. The MMU is responsible for handling memory access requests. It takes care of virtual memory management, paging, memory protection, and bus arbitration. Its job is to take pieces of dispersed physical memory and present them to the requesting process as a contiguous block. In porting OSSIE to the MMUless C64 platform, all memory management is the responsibility of the developer. Certain OS functions, such as spawning a copy of the running process i.e. child process from certain running parent process, are not supported. Another important area in the development is the porting of all schedulable tasks to the preemptive, multithreaded DSP/BIOS. The main difference from a traditional fair-share OS is that the active task with the highest priority will be scheduled for execution; no matter how many other tasks are waiting, or for how long. This characteristic allows deterministic execution, crucial in real-time systems, but makes the developer completely responsible for task

scheduling and priority assignment. The functionality of the Core Framework is split between Host and Remote nodes. The Host node includes an instance of ***DomainManager***, while a remote node includes an instance of ***DeviceManager*** and other Devices. Fig. 2 shows the CF interfaces allocated to each node. There are other possible strategies, for example having a node hosting both ***DomainManager*** and ***DeviceManager***, while the rest of the nodes in the platforms only host Devices. We propose this approach to stress our implementation and evaluate the degree of flexibility delivered by it.

3.1 Proposed XML Domain Profile Parsing Strategy

The SCA specification requires parsing of the XML Domain Profile at runtime to obtain deployment and configuration information [6]. For example, the ***ApplicationFactory*** interface must read a Software Assembly Descriptor (SAD) file in order to know what components are included in a given waveform application and their connections. Parsing an XML file is a complicated task for a DSP and there are not many tools available to perform this. In order to facilitate development, reduce memory requirements, and speed execution, we developed a two-step parsing scheme designed to facilitate Domain Profile parsing by the DSP. In this scheme, an offline translation of the XML files into a simplified proposed format is performed. The proposed simplified format only keeps the most important information from the profile files and stores it in a simple text file. The information kept includes all the data required for successful deployment and configuration of waveforms and components: UUIDs, descriptors locations, connections, etc.

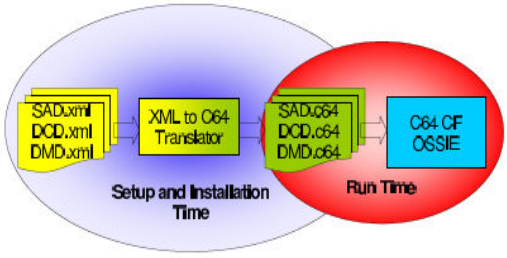

Fig.3: Proposed XML Domain Profile Parsing Strategy.

The information discarded represents information not indispensable for waveform deployment and operation: descriptions, headers, authors, etc. Even though the discarded information is important, and therefore must be provided when developing an SCA component, the main framework functionality does not require it for proper operation. A graphical representation of this approach is shown in Fig. 3. It can be argued that this approach is not SCA compliant. However, having this two-step parsing strategy does not affect the design cycle of traditional SCA waveforms and only adds one extra step at installation time. The savings in time and complexity, along with the uncompromised portability of the resulting waveforms justify this decision. We implemented the XML translator in VB6 under windows XP; it uses Microsoft MSXML library to parse XML domain profile. The translator parses an SCA compliant XML file, gathers the required information, and writes the translated file with a .c64 extension, preserving file names and directory structure. These simplified .c64 files are then parsed at real time by the framework running on the C64 platform.

3.2 File System

Our hardware platform does not have long-term storage capability [4]. Therefore, only a partial file system is implemented in this project. The host computer's hard drive and file system are used by the framework. This is accomplished by CCS I/O utilities. To implement the file system interfaces we relied on IO functions from the TI run-time support library. However, the access allowed by this library is limited primarily in terms of directory manipulation. Therefore, functionality such as ***mkdir***, ***rmdir***, ***mount***, and ***unmount*** is not implemented.

3.3 Software Component Deployment

The SCA specifies two equivalent mechanisms to launch software components [6]. One is using ***ResourceFactory*** and the other using ***ExecutableDevice***. The ***ExecutableDevice*** interface typically represents processors with a multithreaded operating system capable of launching software

components. ***ExecutableDevice*** has access to the OS directives to schedule the component. ***ResourceFactory*** performs the exact same functionality and is used as a local tool to deploy components without a ***DeviceManager***. In this project we use the ***ResourceFactory*** interface to deploy components in the host node and an ***ExecutableDevice*** for remote nodes. The implementation of these interfaces uses DSP/BIOS static task scheduler. Every time a new component instance is required, a new task is created and scheduled. The ***ResourceFactory*** and ***ExecutableDevice*** implementations are in charge of managing the new task's priority. Because of the lack of an MMU and long-term storage capability, it is necessary to have all the tasks loaded in program memory before they can be scheduled. This mechanism is proposed due to real time nature of the system.

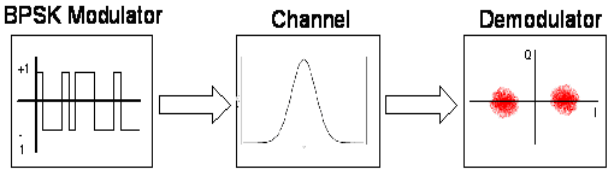

Fig.4: Sample BPSK Application Waveform.

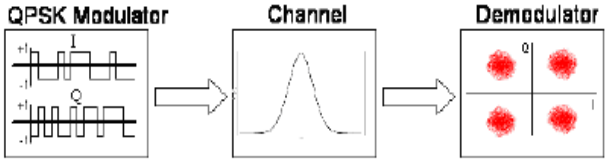

Fig.5: Sample QPSK Application Waveform.

4 Sample Application

In order to demonstrate the framework functionality, two sample applications are developed. These applications are intended for demonstration purposes and nothing else. No extensive signal processing is performed. The main goal for these applications is to verify the operation of the framework and to corroborate the feasibility of deploying SCA compliant waveforms onto the C64 platform. The first application includes three simple components: BPSK Modulator, AWGN Channel, and Demodulator as in Fig. 4. The BPSK modulator generates a random stream of 1's and -1's. The stream is passed to the Channel component which adds Gaussian noise to the In-Phase and Quadrature components of the stream. The Demodulator only displays the constellation diagram of the signal. The second waveform includes a QPSK modulator and demodulator instead of BPSK modulator and demodulator. Fig. 5 shows a graphical representation of the second waveform. Both waveforms were successfully deployed on a single chip configuration using the ***ResourceFactory*** interface to launch the components.

5 Results

In this section we present general profiling results for our implementation. The framework capabilities are demonstrated by switching back and forth between two waveforms. Code Composer Studio (CCS) is used to control the execution, display information and error messages, and enter selection values. Keep in mind that from the framework perspective there is no difference between deploying these simple waveforms and deploying more sophisticated ones.

5.1 Profiling

Profiling was performed on the framework and application using two different metrics: ***memory footprint*** and ***cycle count*** as in [2]. The former represents the extra memory space necessary to support the SCA framework. The latter represents the amount of overhead imposed by the framework in terms of processing power. All results were obtained from a single-chip configuration. That is, all framework and waveform components were collocated within the same processor; they do not include a transport layer. No optimizations were performed in either the framework or the waveform components. All performance tests were carried out using the C6416 Device Cycle Accurate Simulator and the Code Composer Studio profiler. It is very important to emphasize that these results represent initial measurements and are subject to further investigation, validation, and optimization.

5.2 Memory Footprint

Memory allocation results are obtained from the .MAP file generated by CCS Code Generation Tools. This file contains a mapping of all sections allocated in memory. It includes program memory and data memory. All dynamic memory allocation requests are served from a memory pool or heap, which is also included in the .MAP file. All profiling results are presented in 8-bit memory word. Note that the C6416 DSP has 16M bytes of external memory (SDRAM) besides the 1M Bytes of internal memory (IRAM). The total memory used by the system is shown in Table 1. It represents a little less than 1% of the available memory in the platform. These results correspond to the single-chip implementation of both, BPSK and QPSK, sample waveforms. The footprint is directly related to the application's functionality and the number of components. Table 2 shows the memory breakdown by major components. The ***.SDRAM$heap*** field represents the total heap available to serve dynamic memory allocation requests from the application. The footprint contribution from support libraries (e.g. Generic Runtime Library, Math Library, etc) is considered under the "Other" category. Fig. 6 shows a pie chart representation of the main components' contribution to the total memory allocation.

Table 1: Total Software Memory Allocation.

Memory Type	Size in Bytes
Used Internal Memory (IRAM)	57,384
Used External Memory (SDRAM)	1,185,110
Total Used Memory	1,242,494
Total Available Memory	135,266,303

Due to space limitations, we do not break down the memory footprint for each major component. Instead, we comment on some important aspects and state some qualifiers for these results. In the break down of the memory requirements for the Core Framework (CF) we find that almost 70% of the total memory allocated for the CF comes from the C++ mapping of the SCA CF IDL interfaces.

It is important to note that the CF IDL descriptions, cf.idl file, contain all the interfaces defined in the SCA CF, including some that are not used in single-processor operation (e.g. ***Device***, ***DeviceManager***). It is possible to optimize the C++ bindings of IDL interfaces by adding more control to the IDL compiler, enabling more selective code generation (e.g. for specific interfaces generate client stub only, or server skeletons only, or nothing). This approach opens the door for potentially large improvements depending on how much of the IDL interfaces are being used [3]. This is a well understood approach, although it is not implemented in this project. Another important qualifier for these results is the absence of Device-related interfaces. No ***DeviceManager*** or ***Device*** interfaces were implemented. The methods in ***DomainManager*** relative to ***Device*** and service registration and un-registration are not implemented in this project version as well. The memory requirement results for the application include both BPSK and QPSK components, along with Channel, Demodulator, Resource Factory, Assembly Controller, and the user interface. The main waveform components have a very similar footprint as expected. However, the functionality of these components is extremely simple. More complex waveforms will require more memory. The results correspond to the ORB are from an ORBExpress DSP libraries' memory footprint.

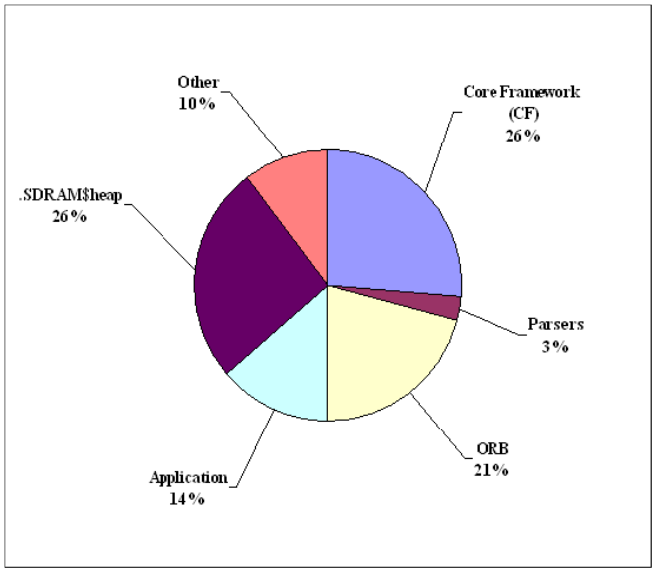

Fig. 6: Memory Footprint Summary.

Table 2: Memory Breakdown and Component Contribution.

Allocated Memory Component	Size in Bytes
Core Framework (CF)	327,748
Parsers	33,793
ORB	260,571
Application	169,028
Sub-Total	791,140
.SDRAM$heap	323,584
Other	127,770
TOTAL Memory	1,242,494

5.3 Performance Profile

CPU Cycle requirements are collected for the most significant sections of the implementation. The sections profiled were domain initialization and waveform creation. The results are shown in Table 3. Domain initialization is not application dependant and includes the instantiation of Domain Manager, ApplicationFactory, and ResourceFactory. Waveform creation represents the execution of ***ApplicationFactory***'s ***create*** (). It includes descriptor parsing, task scheduling and initialization, and component connection. Keep in mind that waveform creation is waveform specific and these results only apply to our test waveforms.

Table 3: Core Framework Tasks Performance Profile.

Task	Cycles	Time(ms) at 720 MHz
Domain Initialization	1,174,726	1.632
Create Application	5,474,664	7.604

6 Conclusion

One of the main concerns of applying the SCA is the heavy infrastructure required to support it. In order to ease requirements in terms of performance, cost, and power consumption, we propose an implementation of the SCA Core Framework for a TI C6416 DSP platform. This approach minimizes the total memory footprint of our complete implementation to about 1.2 MB, which represents 7.5% of the 16 MB available memory in our DSP platform. Benchmarks show that, using OIS ORBExpress DSP ORB middleware decreases Memory Footprint and increases Processing power compared with PrismTech's e*ORB middleware [2].

7 References

[1] Aguayo Gonzalez, Carlos R., Francisco M. Portelinha, and Jeffrey H. Reed, "Part 1: Design and implementation of an SCA core framework for a DSP platform," Military Embedded Systems, May 2007.

[2] Aguayo Gonzalez, Carlos R., Francisco M. Portelinha, and Jeffrey H. Reed, "Part 2: Design and implementation of an SCA core framework for a DSP platform," Military Embedded Systems, May 2007.

[3] Objective Interface Systems website Available at: http://www.ois.com

[4] Texas instruments inc. website Available at: http://www.ti.com

[5] Open-source SCA implementation::embedded Available at: http://www.mprg.org/people/caguayog/OSSIE_TI/index.htm

[6] Software Communications Architecture Specification V2.2. http://jtrs.army.mil.