

Common API Strategy Aids Heterogeneous Processing

Making software run smoothly on a heterogeneous multiprocessing system is no small challenge. By abstracting the hardware platform details and using a common API, system developers can smooth the way.

by Kevin Maier
Spectrum Signal Processing

A common method of managing system resources across disparate operating environments in a heterogeneous multiprocessing system is essential for success in today's competitive landscape. One way of adding value is to provide a software programming environment that is consistent across both evolutionary platforms and within platforms containing radically different types of devices. This programming environment will typically include a host and target operating system, a variety of programming tools and presumably a generic system control application programming interface (API). This level of API can be thought of as a hardware abstraction layer (HAL).

A hardware abstraction layer is intended by definition to abstract away hardware implementation details from higher levels of software and should enable algorithm partitioning across various signal processing devices. In a PC environment, this is relatively easy as the hardware and buses tend to be relatively well defined: PCI, memory, AGP graphics bus, and so on.

In an industrial-level environment such as a software defined radio (SDR) system, however, the variety and complexity of the hardware is more extreme, thus leading to a more difficult job of creating a proper, general, hardware abstraction layer. In addition, in most embedded systems, how the various pieces of hardware can

communicate with each other requires some knowledge of system configuration, which should be accounted for in the API.

Dimensions of Portability

An ideal system configuration, control and communication API (a C3API, if you will) consists of a number of function calls which, when coupled with some system management concepts, allow a user to write code which can be ported easily from one type of hardware to another. As typical software drivers include open, close, read, write, and ioctl routines, this API would enhance this standard driver methodology and include functionality for:

- opening and closing both resources and systems of resources
- resetting and loading devices
- reading and writing both memory and streams
- adding, deleting and controlling abstract resources contained in FPGA cores
- responding to and generating interrupts and signals
- handling communications channels between different embedded hardware objects (ASICs, DSPs, FPGAs, GPPs and so on).

Such an API should enable efficient acquisition and transfer of data through the signal processing system as the data moves through various data paths and processing blocks. The same function calls should be used by the programmer to access any resource in the system, regardless of the type of processor, data path and operating system. All these requirements significantly ease programming difficulty on a complex heterogeneous processing platform.

Beyond the form and functionality, there are numerous other dimensions of portability as shown in Figure 1, which are critical to protecting one's software investment and must be addressed in the development of a C3API. A designer must consider at minimum:

- Will the API be run on one bus architecture or multiple architectures?
- Which operating systems will the API be run on and ported to?
- Should the design allow for platform evolution or scalability?

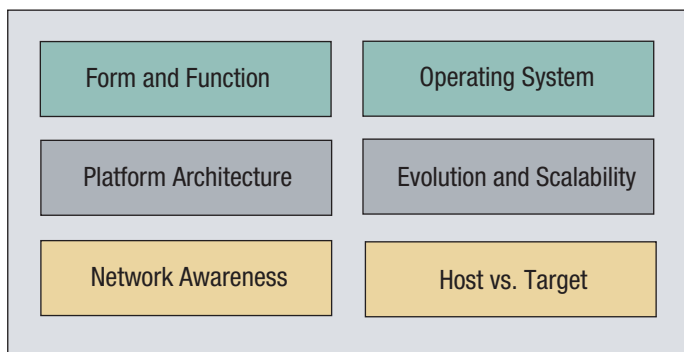


Figure 1 There's more to portability than meets the eye. Shown here are some of the many dimensions of portability that are critical to protecting software investment.

- Will remote configuration and control (network awareness) be built in?
- How will the differences between host and target be reconciled?

Platform Architecture Portability

A COTS hardware vendor will often support a similar architecture of devices across multiple backplanes. For instance, a PCI card with two TMSC6416 Texas Instruments DSPs and a Xilinx Virtex-II FPGA onboard might have the same basic architecture as a quad DSP/ dual FPGA VME card. The PCI card could be very useful for prototyping and early development due to close interaction with the host PC it was plugged into and thus have no need for a single board computer or chassis. However, the usefulness of the PCI card prototyping would be nullified if there were no method of abstracting away the backplane and would only be useful for prototyping a deployable VME system if the host and target programming interfaces were identical for both the PCI board and the VME board.

There are a number of popular operating systems on the market today for embedded systems programming. Although a “C” language API is very portable, efforts must be made to ensure that the underlying software is also portable, and that the behavior of the API is the same across multiple operating systems. Using standard POSIX calls as a basis for your operating system interaction aids in this regard, as this design decision allows fairly easy porting of your API library between both compliant and pseudo-compliant POSIX operating systems. However, the mere use of POSIX does not encompass all of the programming that is required for a heterogeneous system configuration, control and communication API.

The depth of operating system portability can be a driving factor in your API development. It can be as thin as requiring a POSIX operating system, using a thin layer of operating system abstraction (such as declaring explicitly sized types like UINT32 or INT8 for different processor types) or something as heavyweight as the ACE environment. The ACE environment provides a framework that implements many of the core patterns for concurrent communication such as event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization.

By layering software appropriately and building on common lower layers, an application developer can more easily port their application from one operating system to a different operating system. If the hardware is the same and the lower level API has been developed appropriately, it may only be required to recompile the application level code in order to complete the port.

Platform Evolution and Scalability

Platform evolution can include moving from an early development system, such as PCI, to a deployed system on a VME backplane, or it could mean evolving from a system of TI C6203 chips and Xilinx Virtex FPGAs to a next-generation platform of TI

A Possible Layering for Software Portability

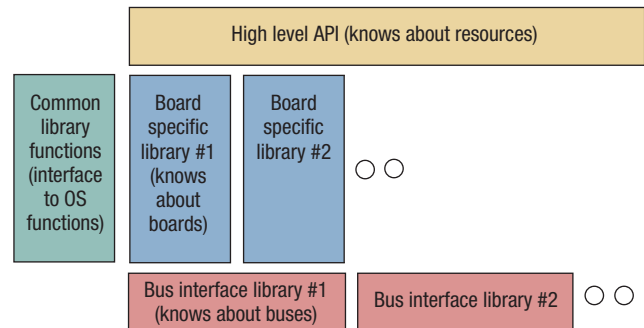


Figure 2 A multi-layered implementation as shown can be crafted using a high-level common API. This API manages system and board resources, splitting out board-specific, bus-specific and common OS interfacing.

C6416 DSPs and Xilinx Virtex-II PRO FPGAs. In either case, having an API that is consistent across different generations is necessary for preserving application-level software investment. One can consider system scalability in the same context: the issue of moving from a single board system in a single chassis to a multiple board system in a rack of chassis can constitute a very similar problem as generational change. Your API must be scalable and expandable.

Many modern operating systems have network awareness, as simple as a basic TCP/IP stack, built into them. In the design of your API, you need to consider if you wish to exploit this in your API or not. Middleware such as CORBA, DCOM, RPC or .Net can be used in the implementation of your API to allow “remote control” of your system through TCP/IP or any other network interface. A designer may also choose to integrate network awareness to their API after the fact, which allows greater independence and does not tie your design to a third-party middleware implementation. The most extreme case of remote control would be, however, to create a protocol stack from scratch in order to allow configuration and control of your embedded system from a remote host.

In software defined radio systems, there is often a clear delineation between the host processors and the target processors. Most often the two run different operating systems, have access to different hardware and have different characteristics and responsibilities. Quite often, the host CPU will be used to process baseband, low bandwidth data and the embedded processor(s) will be used to process the high bandwidth RF or IF data. It is important to reconcile a system control API across these varied requirements. This point is in fact the combination of virtually all of the previous discussion and in fact defines the crucial framework for the next part of the discussion.

An Example Programming Environment

One possible implementation fulfilling the listed portability issues could be created by using a multi-layered implementation as follows. In Figure 2, one can see that by creating a high-level

Possible Functions of the Common High-Level API

Category	Function	Description
Management	Open	Open a resource in the system, create a handle to the resource
	Close	Close the resource, free the resource handle
	Reset	Put the resource in a know state
	Execute	Load a resource or cause a process to start
	Claim	Lock access to a shared resource
	Release	Unlock access to a shared resource
Memory Access	MemRead	Read the memory resource (this may be local or remote via a bus)
	MemWrite	Write the memory resource
Link Access	LinkRead	Read the stream resource into a buffer
	LinkWrite	Write to the stream resource from a buffer
Signals	SendSignal	Send a signal to the resource
	WaitSignal	Wait for a signal from the resource
	CallbackConnect	Attach a callback to a resource's signal
	CallbackDisconnect	Detach a callback from a resource's signal

Table 1 Listed here are some of the possible functions of the high-level common layer, which would use something like COM to implement the remappable interfaces to the next layer down, the board services layer. Every device in a system would be given a resource name/handle, allowing it to be addressed and interacted with.

common API for managing system and board resources, and splitting out board-specific, bus-specific and common OS interfacing, many of the portability requirements are realized.

The common API addresses issues at a system level, the board libraries at the board level, and the bus interface layer at the bus level. An extra library of operating-specific implementations of common utilities even allows for porting to processors without operating systems. By structuring your software offering in this way, it is much easier to evolve from board-to-board and bus-to-bus, because the application-level code investment is largely preserved. The board vendor would of course be responsible for building and maintaining each of these libraries, which is not an inconsiderable task. But, if the name of the game is portability, this structure would fulfill your mandate.

Table 1 describes some of the possible functions of the high-level common layer, which would use something like COM to implement the remappable interfaces to the next layer down, the board services layer. Every device in a system would be given a resource name/handle, allowing it to be addressed and interacted with. Resources could be, for example, a PCI configuration space, local memory on a processor, an FPGA or even a channel between serial RapidIO parts. Each resource handle would have a predefined subset of functions that could be performed on it.

“C” vs. Java

Implementation in our example should be in “C” due to the lack of performance by Java currently and the requirement of in-depth interaction with the hardware at the lowest level. A Java Native Interface (JNI) could be added on top of the “C” layer if Java was truly desired. Additionally, “C” is still the most common programming language in the embedded industry and has compilers supported on virtually every widely used operating system.

In our example we would use external files to describe the system configuration. This requires processors to have access to

a file system, or to have configuration information written to them by a host that does have file access. By having the ability to read system configuration files to create an internal representation of a system, including boards and devices on boards, we can have access to all resources in the system from any other part of the system, even to the point of using a CORBA/COM interface to allow remote control of devices only available on local buses.

Using system definition files to create the representation of the system can be expanded to allow creation and removal of resources “on-the-fly” when boards are hot-swapped into and out of

the system or FPGA images are loaded and unloaded. Additional highly portable, high-level libraries can be created by using the resource handles managed by the common API. An example of such a high-level library would be for a particular piece of I/O, whose functions do not map well to a truly generic interface as described.

Common API Solution

For its part, Spectrum Signal Processing has been developing for a number of years a standard application programming interface that can be common across all of our various hardware platforms. The result of this effort is the quicComm software framework, which consists of an API addressing both system and individual resource management.

All of the common functions and more are available for the framework. The software stack is structured to allow a large degree of portability between buses, boards and devices. The quicComm software has been ported to a number of operating systems, both on the host and target. A version of the API even runs on Texas Instruments DSPs with no OS. Further, quicComm is extremely scalable and has been integrated with the Harris SCA Core Framework by Spectrum to provide a fully SCA-compliant software defined radio system. ■

Spectrum Signal Processing
Burnaby, BC, Canada.
(604) 421-5422.
[www.spectrumsignal.com].