Introduction to Hardware Abstraction Layers for SoC

Sungjoo Yoo Ahmed A. Jerraya
System-Level Synthesis Group
TIMA Laboratory
Grenoble, France

Abstract

In this paper, hardware abstraction layer is explained in the context of SoC design. First, HAL definition is given and the difference between HAL and other similar concepts are given. Existing HALs are examined. The role of HAL is explained for SoC design. Finally, a proposal of standard HAL is presented.

1. INTRODUCTION

Standard on-chip buses have been developed to enable hardware (HW) component reuse and integration [1][2]. Recently, VSIA is investigating the same analogy in software (SW) component reuse and integration with its hardware dependent software (HdS) API. It is one that is conventionally considered as hardware abstraction layer (HAL) or board support package (BSP).

In this paper, we investigate the following questions related to HAL, especially for system-on-chip (SoC).
(1) What is HAL?
(2) What are existing HALs?
(3) What is the role of HAL for SoC?
(4) What does the HAL for SoC need to look like?

2. WHAT IS HAL?

In this section, we explain HAL definition, the conventional role of HAL, and the difference of HAL, nano-kernel, and device driver.

We define HAL as **all the software that is directly dependent on the underlying HW**. The examples of HAL include boot code, context switch code, codes for configuration and access to HW resources, e.g. MMU, on-chip bus, bus bridge, timer, etc. In real HAL usage, a practical definition of HAL can be done by the designer (for his/her HW architecture), by OS vendors, or by a standardization organization like VSIA.

HAL APIs give, to the operating system and application SW, an abstraction of underlying architecture, especially for data types (e.g. The integer type has different bit sizes depending on processors), task context (e.g. context switch), interrupt management (e.g. interrupt vector), I/O, memory management, etc. Figure 1 shows an example of HAL API for context switch, **__cxt_switch(cxt_type oldcxt, cxt_type newcxt)** and the real code of the HAL API for ARM7 processor.

When the OS or application SW is designed using HAL APIs, the code is portable as far as the HAL APIs can be implemented on the underlying HW architecture. Thus, conventionally, the HAL has been used to ease OS porting on a new HW architecture.

```
// HAL API for context switch
typedef int cxt_type[15];
void __cxt_switch(cxt_type oldcxt, cxt_type newcxt);

// ARM7-specific implementation of __cxt_switch
__cxt_switch          ;r0, old stack pointer, r1, new stack pointer
STMIA   r0!,{r0-r14}     ; save the registers of current task
LDMIA   r1!,{r0-r14}     ; restore the registers of new task
SUB     pc,lr,#0         ; return
END
```

Figure 1 An example of HAL API.

There are similar concepts to HAL: nano-kernel and device driver. Nano-kernel is usually defined to be an ensemble of interrupt service routines and task stacks [7]. It serves as a foundation where a micro-kernel can be built. In this definition, nano-kernel can be considered to be a part of HAL. However, nano-kernel is often used exactly to represent HAL. In the case of μChoice OS, nano-kernel is equivalent to HAL [8].

A device driver gives an abstraction of I/O device. Compared to HAL, it is limited to I/O, not covers context switch, interrupt management, etc. To be exact, the entire device driver does not belong to HAL. In the case of device driver, to identify the portion that depends on the underlying HW architecture, we need to separate the device driver into two parts: HW independent and HW dependent parts. Then, the HW dependent part can belong to HAL.

3. EXISTING HALS

Though HAL is an abstraction of HW architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of HALs are OS dependent. In the case of OS dependent HAL, it is often called board support package (BSP).

Window CE provides for BSPs for many standard development boards (SDBs) [3]. The BSP consists of boot loader, OEM abstraction layer (OAL), device drivers, and configuration files. To meet the designer's HW architecture, OAL can be configured. In eCos, a set of

1530-1591/03 $17.00 © 2003 IEEE

well-defined HAL APIs is presented [4]. However, there is no clear difference between HAL and device driver. In Real-time Linux, a HAL called real-time HAL (RTHAL) is defined to give an abstraction of interrupt mechanism to Linux [5]. It consists of three APIs for disabling and enabling interrupts and return from the interrupt.

An example of HAL that does not depend on an OS is a386. [6]. It is a HAL that depends on the i386 processor architecture. Later, it is ported on the ARM processor.

4. HAL AS A STANDARD FOR HW/SW INTEGRATION

In the context of SoC design, HAL keeps still the original role of enabling the portability of upper layer SW. However, in SoC design, the portability impacts on the design productivity in two ways: design reuse and concurrent HW and SW design.

Portability enables to port the SW on different HW architectures. In terms of design reuse, the portability enables to reuse the SW from one SoC design to another. It can reduce the design efforts otherwise necessary to adapt the SW on the new HW architecture. The portability eases also the exchange of SW code and architecture exploration, e.g. trying different target processors to find an optimal target processor.

In many SoC designs, complete SW reuse is infeasible (e.g. due to a new functionality or performance optimisation). Thus, in such cases, both SW and HW need to be designed. The conventional design flow is that the HW architecture is designed first, then the SW design is performed based on the fixed HW architecture. In terms of design cycle, this practice takes a long design cycle since the SW and HW design steps are sequential. HAL serves to enable the SW design early before finishing the HW architecture design. The SW is designed using HAL APIs without considering the details of HAL API implementation. Since the SW design can start as soon as the HAL APIs are defined, SW and HW design can be performed concurrently thereby reducing the design cycle.

To enable the above two benefits, standard HAL APIs are favoured. To establish a standard HAL, first we need to ask a question: Can there exist one standard HAL? We think that the answer will be NO. We consider that one of distinguished aspects of SoC design compared to conventional board design is application-specific HW architecture design. Thus, the SoC design can invent any HW architectures that are suited to the given SoC specification. In such a case, .a fixed set of HAL APIs will not be able to support new HW architectures and HW components. Instead, the standard HAL needs to be a generic set of APIs that consist of common HAL APIs and a design guideline for extensible HAL APIs. It will be also possible to prepare a set of common HAL APIs suited to several application domains (e.g. HAL APIs for multimedia application).

As common HAL APIs, we consider the following five categories of common APIs.

(1) Kernel HAL APIs: They perform context management (e.g. context creation/deletion/switch) and atomic operations (e.g. read-modify-write).

(2) Interrupt management HAL APIs: They are conventionally called nano-kernel. They consist of fast interrupt service routine (user ISR will be in OS interrupt service) and interrupt/interrupt vector management APIs (e.g. interrupt enable/disable, etc.)

(3) I/O HAL APIs: They perform I/O device configuration/access. Especially, bus abstraction APIs (e.g. to allow accesses to the secondary bus as well as the primary one) and memory management APIs (e.g. to access a shared memory) are important since SoC bus architectures and memory hierarchy are becoming more complex.

(4) Resource management HAL APIs: They are architecture (re)configuration APIs, e.g. tracking system resource usage (check_battery()), power management (set_cpu_speed()), real-time APIs (timer_set/reset(), wait_cpu_cycle(), etc.).

(5) Design time HAL APIs: They are used only to facilitate the design process, especially, to facilitate simulation. One of examples is consume_cpu_cycle() to simulate the advance of SW execution time.

As a guideline to develop extensible HAL APIs, our proposal is a component-based construction of HAL as in [8]. HAL consists of components. Components communicate via clear interface. Their internal implementations depend on HW architectures.

REFERENCES

[1] Virtual Socket Interface Alliance, http://www.vsi.org/

[2] Open Core Protocol, http:// http://www.ocpip.org/home

[3] Windows CE, http://www.microsoft.com/windows/embedded/

[4] eCos, http://sources.redhat.com/ecos/

[5] RTLinux, http://fsmlabs.com/community/

[6] a386, http://a386.nocrew.org/

[7] D. Probert, *et.al*., "SPACE: A New Approach to Operating System Abstraction", Proc. International Workshop on Object Orientation in Operating Systems, pp. 133—137, Oct. 1991.

[8] S. M. Tan, D. K. Raila and R. H. Campbell, An Object-Oriented Nano-Kernel for Operating System Hardware Support. In Fourth International Workshop on Object-Orientation in Operating Systems, Lund, Sweden, August 1995.