Center for Embedded Computer Systems
University of California, Irvine

Towards A Unified Hardware Abstraction Layer Architecture for Embedded Systems

Hao Peng [1,2]
hao.peng@uci.edu

R. Dömer[1]
doemer@uci.edu

CECS Technical Report 12-14

Nov. 26, 2012

[1]Center for Embedded Computer Systems
University of California, Irvine

Irvine, CA 92697-3425, USA
(949) 824-8059
http://www.cecs.uci.edu/

[2]Engineering Research Center of Safety Critical Industry Measure and Control Technology of Ministry of Education

Hefei, Anhui 230009, China
(0551) 2903897
http://ialab.hfut.edu.cn

Towards A Unified Hardware Abstraction Layer Architecture for Embedded Systems

Hao Peng [1,2]
hao.peng@uci.edu

R. Dömer[1]
doemer@uci.edu

CECS Technical Report 12-14

Nov. 26, 2012

[1]Center for Embedded Computer Systems
University of California, Irvine

Irvine, CA 92697-3425, USA
(949) 824-8059
http://www.cecs.uci.edu/

[2]Engineering Research Center of Safety Critical Industry Measure and Control Technology of Ministry of Education

Hefei, Anhui 230009, China
(0551) 2903897
http://ialab.hfut.edu.cn

Abstract

The Hardware Abstraction Layer (HAL) is a software layer which resides between the hardware platform and the operating system (OS). The HAL hides the implementation details of the hardware platform from the upper layers of software. The purpose of using a HAL is to reduce the development period of new systems, shortening the pre-market time, and increasing software reusability. Although some OS's define an integrated HAL, these are typically OS-specific and thus not reusable.

In this report, we propose the idea of a Unified Hardware Abstraction Layer (UHAL) which contains the basic set of abstract features of the underlying hardware platform. With such a UHAL, programmers are able to easily compose a software foundation for different OS's.

We present a MP3 player case study to demonstrate the UHAL idea. Our case study experiment uses a BeagleBoard as the hardware platform and eCos as embedded operating system. The case study results show that the proposed UHAL clearly separates the hardware-dependent software development from the hardware-independent software development, while these two parts can be integrated quickly and with low effort afterwards.

Contents

List of Figures

List of Tables

List of Source Code Listings

Towards A Unified Hardware Abstraction Layer Architecture for Embedded Systems

Hao Peng [1,2]
hao.peng@uci.edu

R. Dömer[1]
doemer@uci.edu

[1]Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059
http://www.cecs.uci.edu/

[2]Engineering Research Center of Safety Critical
Industry Measure and Control Technology of
Ministry of Education
Hefei, Anhui 230009, China
(0551) 2903897
http://ialab.hfut.edu.cn

1. Introduction

In embedded systems, software is becoming more and more important than ever before due to its high complexity in the entire system. Because of the market requirements, decreasing design time is one of the key issues in the design process. Software portability and reuse are effective ways to address this problem. However, on account of a change of the hardware architecture and underlying platform, porting an operating system (OS) and/or an application requires a lot of efforts in adjustment, modification and debugging.

In order to improve the portability and reusability, software is generally separated into several layers, a layer of hardware-dependent software (HDS), typically called hardware abstraction layer (HAL), and several hardware-independent layers which include the operating system and application software (1), as shown in Figure 1.1.

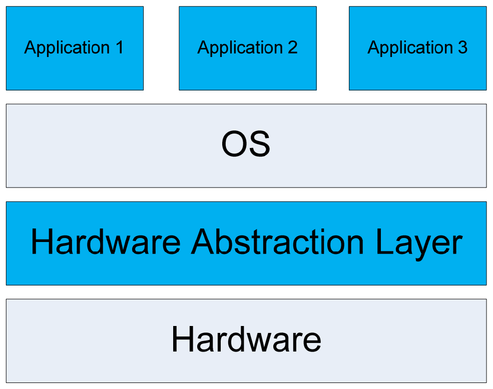


Figure 1.1 Hardware abstraction layer (HAL) in a layer-based software architecture.

Definition: Hardware Abstraction Layer (HAL)

In this paper, we define the hardware abstraction layer (HAL) as the part of the software that directly relies on the underlying hardware platform. The HAL provides a standard application procedural interface (API) to the OS and the actual applications with which they can control the hardware.

Conceptually, the presence of a HAL ensures software portability (1) (2). When porting the OS to different platforms, only the HAL needs to be modified. Based on a well-defined API between HAL and OS, hardware and software development can also be carried out simultaneously so that the overall development time is reduced.

In chapter 2, we first take a look at the underlying hardware platform of general embedded systems and show some examples of the architecture of microcontrollers and microprocessors. In chapter 3, we then discuss in detail the features and functions which are necessary and important in a HAL. We also list specific examples of how the HAL API may look like. In chapter 4, we then investigate the specific HAL in some existing operating systems and compare them to the features described in chapter 3. Finally, in chapter 5, we conclude our study.

2. The Architecture of Embedded Processors

In semiconductor market, there are hundreds of different kinds of embedded processors. They are widely used in various application areas, such as automotive, military, cell phone, office and home electronics, etc. In general, they have the similar architecture. We sort the components within the embedded processors into three categories: core, central processing unit (CPU) and peripherals, which is shown in figure 2.1. Core is the component which executes the instructions. CPU contains the core and the other components which support the core to execute programs. Peripherals are the components which communicate with other systems or physical world.

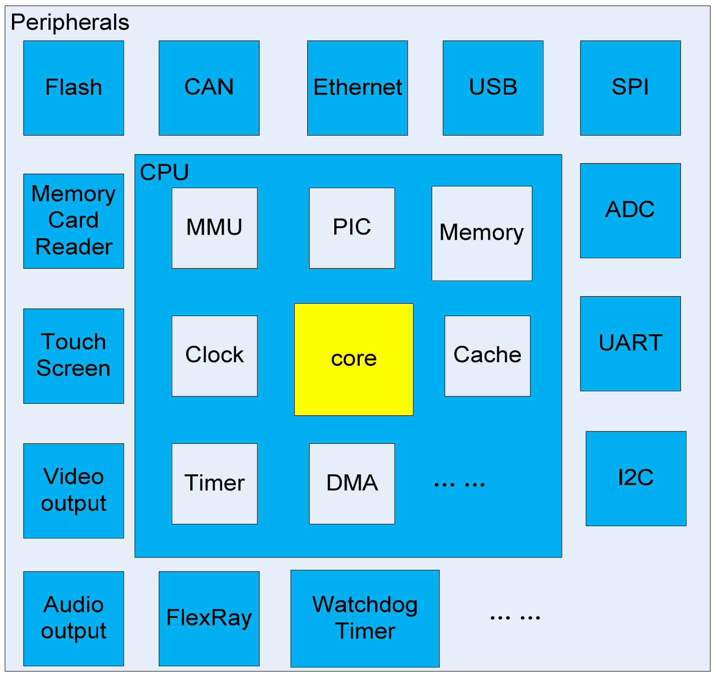


Figure 2.1 The general architecture of an embedded processor

Figure 2.2 illustrates the architecture of Freescale MPC8572E microprocessor (3). It is a dual-core microprocessor with two Power-architecture cores. The cores are separated from other components by the system bus. They are the most critical parts of this chip. The cache is also in the same scope with cores since they have matchable speed. Cache is not as important as cores because without cache the system is still able to work under lower performance. Of other components which connect to the system bus, some are sorted into CPU category while others are sorted into peripheral category based on their roles in the system. For instance, the SDRAM controller is definitely a necessary part for executing programs so that it is part of CPU. However, the Ethernet controller is an application specific component for communicating purposes which is then categorized as a peripheral.

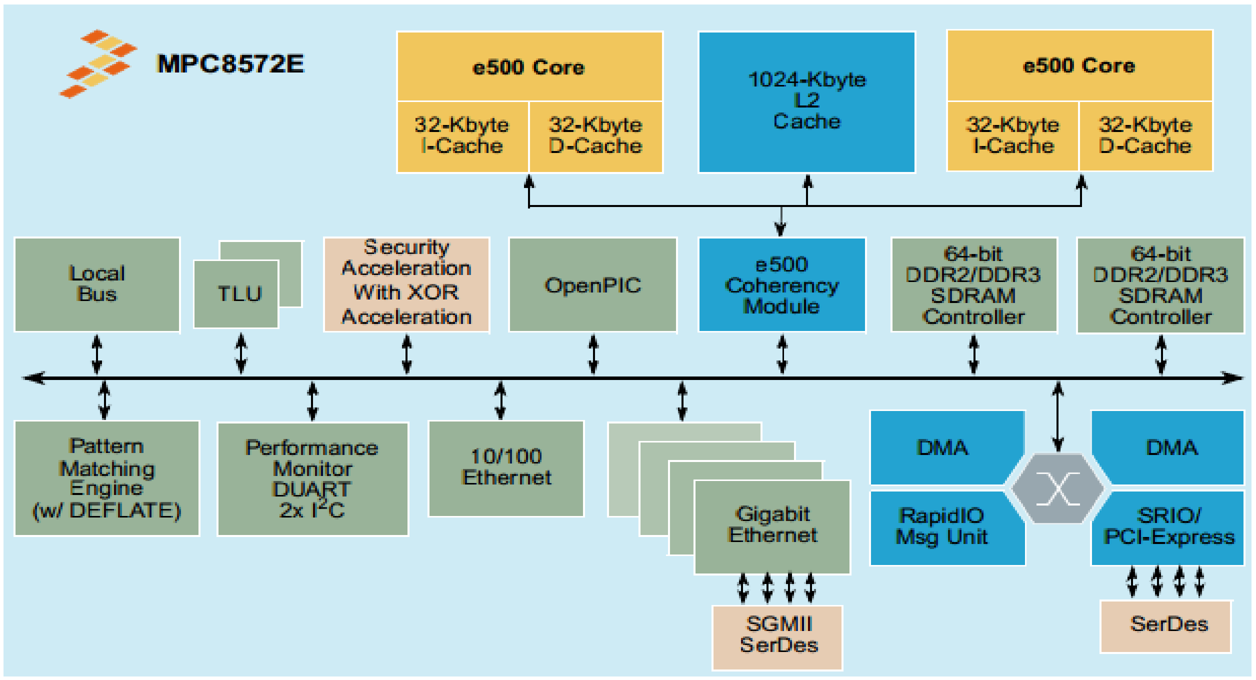


Figure 2.2 The architecture of Freescale MPC8572E PowerQUICC™ III processor (3)

The architecture of TI DM3730 ARM CORTEX-A8 based microprocessor is depicted in figure 2.3 (4). It has a single ARM core with other coprocessors. According to our definition, the ARM core and the coprocessors are classified into the core category as they all execute instructions. The rest components which are connected to “L3 interconnect network”, excluding camera and USB components, are sorted into CPU category. The camera and USB controllers are clearly peripherals. The reason that they are connected to “L3 interconnect” is for speed requirement. Then everything beyond the “L4 interconnect” is a peripheral.

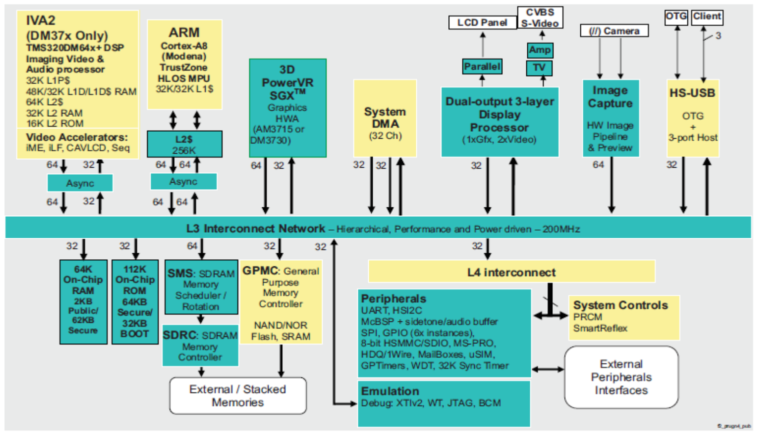

IVA2 (DM37x Only) TMS320DM64x+ DSP Imaging Video & Audio processor 32K L1P$ 48K/32K L1D/L1D$ RAM 64K L2$ 32K L2 RAM 16K L2 ROM
Video Accelerators: iME, iLF, CAVLCD, Seq
ARM Cortex-A8 (Modena) TrustZone HLOS MPU 32K/32K L1$
L2$ 256K
Async
Async
3D PowerVR SGX™ Graphics HWA (AM3715 or DM3730)
System DMA (32 Ch)
LCD Panel
CVBS S-Video
Amp
Parallel
TV
Dual-output 3-layer Display Processor (1xGfx, 2xVideo)
(//) Camera
Image Capture HW Image Pipeline & Preview
OTG
Client
HS-USB OTG + 3-port Host
L3 Interconnect Network – Hierarchical, Performance and Power driven – 200MHz
64K On-Chip RAM 2KB Public/ 62KB Secure
112K On-Chip ROM 64KB Secure/ 32KB BOOT
SMS: SDRAM Memory Scheduler / Rotation
SDRC: SDRAM Memory Controller
GPMC: General Purpose Memory Controller NAND/NOR Flash, SRAM
External / Stacked Memories
L4 interconnect
Peripherals UART, HSI2C McBSP + sidetone/audio buffer SPI, GPIO (6x instances), 8-bit HSMMC/SDIO, MS-PRO, HDQ/1Wire, MailBoxes, uSIM, GPTimers, WDT, 32K Sync Timer
System Controls PRCM SmartReflex
External Peripherals Interfaces
Emulation Debug: XTIv2, WT, JTAG, BCM

Figure 2.3 The architecture of TI DM3730 ARM CORTEX-A8 based processor (4)

Figure 2.4 depicts the architecture of Renesas (NEC) microcontroller (5). It shows a clearly hierarchical structure. The upmost part is the core which connects with supporting components via the CPU bus and the internal bus with a bus bridge. All the peripherals connect to a peripheral bus and communicate with upper hierarchies via a peripheral bus bridge. Then from the top of the picture, the first row of blocks is in the core category, the second and third row of blocks are in the supporting component category, and the last two rows are in peripheral category.

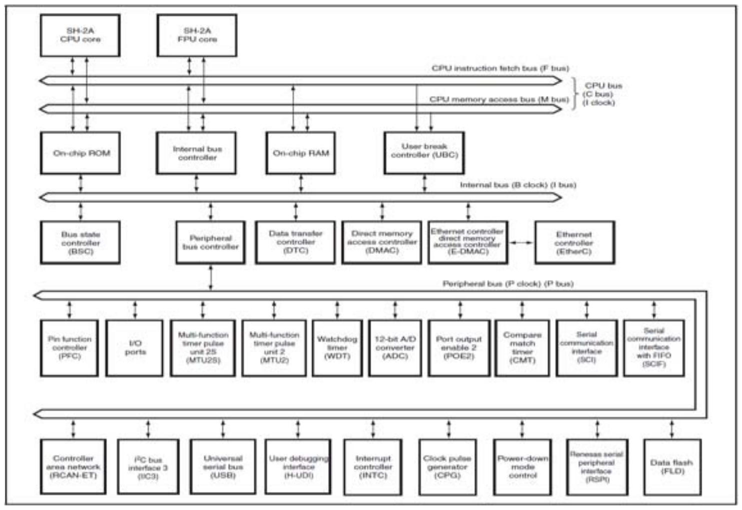

SH-2A CPU core
SH-2A FPU core
CPU instruction fetch bus (F bus)
CPU memory access bus (M bus)
CPU bus (C bus) (I clock)
On-chip ROM
Internal bus controller
On-chip RAM
User break controller (UBC)
Internal bus (B clock) (I bus)
Bus state controller (BSC)
Peripheral bus controller
Data transfer controller (DTC)
Direct memory access controller (DMAC)
Ethernet controller direct memory access controller (E-DMAC)
Ethernet controller (EtherC)
Peripheral bus (P clock) (P bus)
Pin function controller (PFC)
I/O ports
Multi-function timer pulse unit 2S (MTU2S)
Multi-function timer pulse unit 2 (MTU2)
Watchdog timer (WDT)
12-bit A/D converter (ADC)
Port output enable 2 (POE2)
Compare match timer (CMT)
Serial communication interface (SCI)
Serial communication interface with FIFO (SCIF)
Controller area network (RCAN-ET)
I²C bus interface 3 (IIC3)
Universal serial bus (USB)
User debugging interface (H-UDI)
Interrupt controller (INTC)
Clock pulse generator (CPG)
Power-down mode control
Renesas serial peripheral interface (RSPI)
Data flash (FLD)

Figure 2.4 The architecture of RENESAS (NEC) SH7214 Group, SH7216 Group MCU (5)

From the examples illustrated above, we have seen that a typical embedded processor matches our definition of general architecture. Then if we create an abstractive model for each component within one category, it is possible to use different combinations to abstract the specific underlying hardware platforms which are significantly different from each other.

3. A Unified Hardware Abstraction Layer (UHAL)

In the last section, we can see that the embedded processors have similar architectures. In general, they have one or multiple cores, execution supporting components, and peripherals. Even though they are implemented in different ways, from a software developer's point of view, the same hardware component from different vendors basically does the same thing. For instance, the core executes instructions regardless it is an ARM, POWER or MIPS core, the MMU unit always performs virtual-to-physical memory translation, and an Ethernet controller sends and receives Ethernet packages. Each implementation of a kind of hardware may have different registers, access method, or buffer size. However they have the same function which can be utilized by operating systems and applications. So the idea here is we can abstract a hardware component into basic functions and hide the details of the way how these functions are implemented. With these functions programmers are able to compose system calls and device drivers for various OS's.

According to the classification of hardware components, HAL is divided into three categories as well: core-related, CPU-related and peripherals-related, which is shown in figure 3.1.

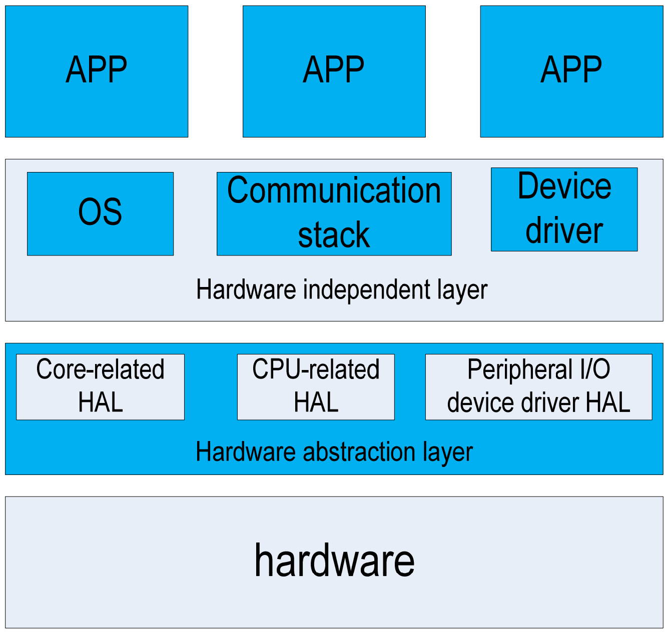

APP
APP
APP
OS
Communication stack
Device driver
Hardware independent layer
Core-related HAL
CPU-related HAL
Peripheral I/O device driver HAL
Hardware abstraction layer
hardware

Figure 3.1 Three categories of HAL

Each part of upper layer is able to invoke any part of the HAL if needed. For instance, the HAL of DMA controller, as well as the HAL of interrupt controller, can be utilized by either of OS, communication stack, or device driver.

4. Features and Functions of a UHAL

In all the sections below, all functions are defined with lowercase letters while the macros are defined with uppercase letters. In this section, the real code examples come from the eCos (6) distribution on the ARM architecture.

4.1 Core-related features

Core is the central part of the CPU. It carries out the instructions of programs. There are a lot of differences between different cores, such as registers, internal exception types, etc. Thus for running an OS on a specific architecture, the code related to the core should be modified, including the context creation and switching, internal exception handler, etc.

4.1.1 Hardware context creation and switch

Context switch is a critical feature to support multithread operating systems, which is directly related to hardware architecture. Regardless of the different software structure of a thread among different operating systems, the hardware context within a thread has some common features. The hardware context contains all the registers of a CPU, should be saved to memory when the thread stops running and loaded into registers when the thread is activated again. These features are hardware-dependent, consequently, should be implemented in HAL.

4.1.1.1 General model

First, HAL defines a structure which represents the specific hardware context shown in listing 4.1. This structure contains all general purpose registers, floating point registers and status registers corresponding to the core architecture.

Listing 4.1 Hardware context

```
struct
 {  general purpose registers;
    floating point registers;
    status registers;
    PC, SP, etc;
} hardware_context;
```

Then we need functions/macros which save the hardware context into memory and load hardware context into CPU registers to support multithreading, shown in table 4.1.

Table 4.1 Functions/macros support multithreading

Function/Macro	Comment
hardware_context * create_context (entry point, stack, entry data)	Create a hardware context for a thread.
SAVE_CONTEXT (hardware_context * ptr)	Save the hardware context of current executing thread to address “ptr”.
LOAD_CONTEXT (hardware_context * ptr)	Load the hardware context saved in address “ptr” to CPU registers.
void switch_context(hardware_context* from, hardware_context* to)	Save the current hardware context to address “from” and restore the hardware context from address “to”.

4.1.1.2 Hardware context creation and switch for eCos example

Listing 4.2 to 4.4 show the hardware context manipulation in eCos.

In listing 4.2, a structure which represents the hardware context architecture of ARM CORTEX-A8 CPU is defined. Because eCos is not an OS intended to do much floating-point arithmetic operations, the hardware context doesn’t include the floating point registers however they are easy to be added.

Listing 4.2 ARM CORTEX-A8 hardware context without floating point accelerator registers

```
typedef struct
```

```
{   cyg_uint32  d[11] ;         //r0-r10
    cyg_uint32  fp;                 // (r11) Frame pointer
    cyg_uint32  ip;                 // (r12)
    cyg_uint32  sp;                 // (r13) Stack pointer
    cyg_uint32  lr;                 // (r14) Link Reg
    cyg_uint32  pc;                 // (r15) PC
    cyg_uint32  cpsr;               // CPU status register
      // The data below are only saved for exceptions and interrupts
    cyg_uint32  vector;             // Vector number
    cyg_uint32  svc_lr;             // saved system mode lr
    cyg_uint32  svc_sp;             // saved system mode sp
} HAL_SavedRegisters;
```

Listing 4.3 shows the hardware context creation macro in eCos where "*_sparg_*" is the thread stack, " *_thread_*" is the data argument passed to the entry function "*_entry_*", "*_id_*" is an identification value assigned to this thread for debugging purpose only.

Listing 4.3 Hardware context creating for ARM core

```
#define HAL_THREAD_INIT_CONTEXT( _sparg_, _thread_, _entry_, _id_ )       \
     register uint32 _sp_ = ((uint32)_sparg_) &~15;                 \
     register HAL_SavedRegisters *_regs_;                           \
     int _i_;                                                       \
     _regs_ = (HAL_SavedRegisters *)((_sp_) - sizeof(HAL_SavedRegisters));     \
     for( _i_ = 0; _i_ <= 10;  _i_++ )                              \
           (_regs_)->d[_i_] = (_id_)|_i_;                           \
     (_regs_)->d[00] = (uint32)(_thread_);                          \
     (_regs_)->sp = (uint32)(_sp_);                                 \
     (_regs_)->lr = (uint32)(_entry_);                              \
     (_regs_)->pc = (uint32)(_entry_);                              \
     (_regs_)->cpsr = (CPSR_THREAD_INITIAL);                        \
     _sparg_ = (CYG_ADDRESS)_regs_;
```

In listing 4.4, there are two assembly functions, "*hal_thread_switch_context*" and "hal*_thread_load_context*". The former switches the program from one thread to another and the latter loads and executes a new thread. Before invoking these functions, the address from which the new hardware context is loaded is stored in register "r0" and the address to which the current hardware context is saved is stored in register "r1". From line 3 to line 9, it can be considered as a function which saves current thread as well.

Listing 4.4 Context switch code for ARM core

```
1.        .globl hal_thread_switch_context
2.   hal_thread_switch_context:
3.        mov      ip,sp
4.        sub      sp,sp,#(ARMREG_SIZE – 20)
5.        stmfd    sp!,{ip,lr}
6.        stmfd    sp!,{r0-r10,fp,ip}
7.        mrs      r2,cpsr
8.        str      r2,[sp,#armreg_cpsr]
```

```
        str     sp,[r1]
        .globl hal_thread_load_context
hal_thread_load_context:
        ldr     fp,[r0]
        mrs     r0,cpsr
        orr     r0,r0,#CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE
        msr     cpsr,r0
        ldr     r0,[fp,#armreg_cpsr]
        msr     spsr,r0
        ldmfd   fp,{r0-r10,fp,ip,sp,lr}
        mov     pc,lr
```

4.1.2 Internal exception handler

An exception is an event that disrupts normal execution of the program. It might be generated by an internal failure or an external signal. Based on the source, the exceptions are classified into internal exceptions and external exceptions which are also called interrupts. In this section, only internal exceptions are discussed. The external exceptions are left to the interrupt controller section.

The internal exceptions are brought about by execution of program, for instance, reading a non-existing memory address, executing an unknown instruction, executing a software exception instruction, etc. They are non-maskable and need to be handled effectively and in time.

4.1.2.1 General model

Different OS's have different methods to handle or utilize the internal exceptions. For example, Linux utilizes the "software interrupt" instruction to switch to kernel mode. If we ignore the exact meaning of an exception, the exception handler is a piece of code which resides on a specific address in memory and would be executed when the corresponding exception occurs. For achieving most portability, we propose a 3-stages general internal exception handler architecture, which is shown in Figure 4.1.

In the 1st stage process the current hardware context is saved, the exception type and address are recorded as well. Then the saved context is passed to the 2nd stage process which chooses the proper exception handler routine from a pre-defined internal exception handler routine table to handle the exception. This table is maintained by the operating system or applications by attaching exception handling function to or detaching it from the table. After finishing the 2nd stage process, the condition which causes the exception has been properly handled. The 3rd stage process restores the context saved in the 1st stage process and brings the program back to the normal execution flow.

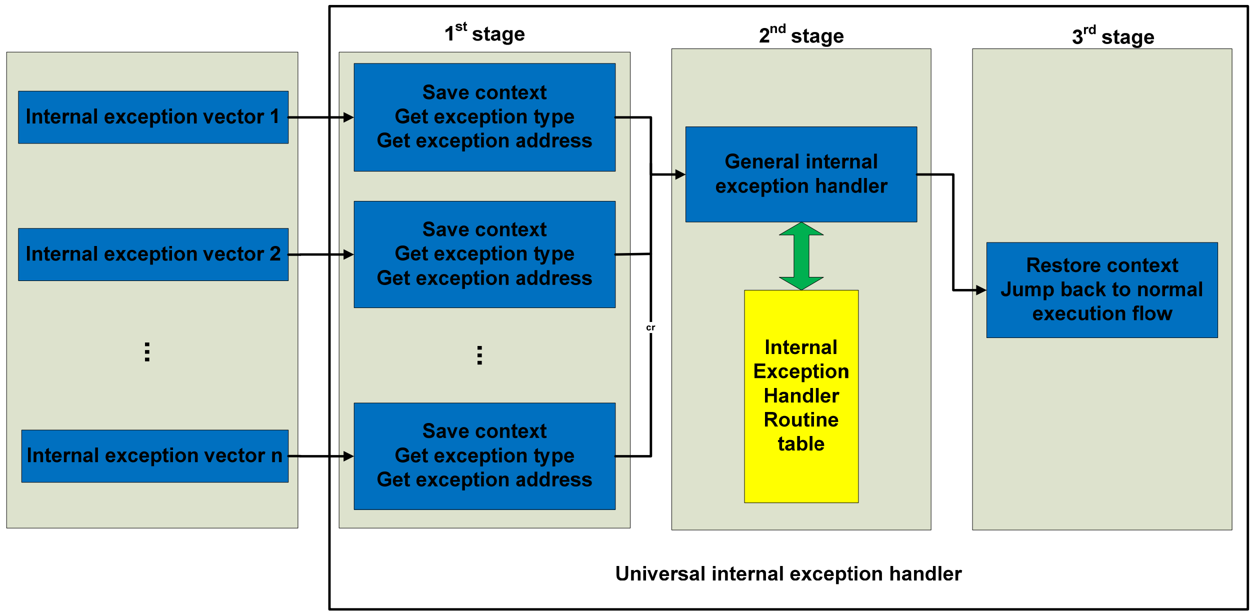

1st stage
2nd stage
3rd stage
Internal exception vector 1
Internal exception vector 2
Internal exception vector n
Save context
Get exception type
Get exception address
General internal exception handler
Internal Exception Handler Routine table
Restore context
Jump back to normal execution flow
Universal internal exception handler

Figure 4.1 3-stages general internal exception handler

To carry out this strategy, first, a macro which implements the 1st stage process should be attached to every exception vector respectively. Usually this macro is written by assembler. Then a function that invokes corresponding exception handler from the routine table is required for the 2nd stage process. Meanwhile the attaching and detaching functions for maintaining this table are also provided by HAL. Finally there is a macro that implements the 3rd stage process, e.g. restoring context and normal execution. The functions and macros that constitute the general internal exception handler are listed in table 4.2.

Table 4.2 Functions/macros of general internal exception handler

Function/Macro	Comment
PRECODE	This macro is the 1st stage process implementation. It saves hardware context and records exception type and address.
POSTCODE	This macro is the 3rd stage process implementation. It restores the hardware context saved in 1st stage then goes back to normal execution flow.
void call_handler (expt_type, expt_address)	Invoke the exception handler corresponding to the exception type “expt_type”. Pass the address “expt_address”, where the exception occurred, as a parameter.
void attach_handler (expt_type, handler_routine)	Attach an exception handler “handler_routine” to the corresponding position in routine table.
void detach_handler (expt_type)	Set corresponding position in routine table to NULL.

4.1.2.2 Internal exception handler for eCos example

In ARM architecture, there are 5 types of internal exceptions and 2 types of external exceptions. Each one has an exception vector respectively all of which reside in a consecutive space of eight 32-bit including a reserved vector.

Listing 4.5 and listing 4.6 is an example of "PRECODE" in an eCos distribution on BeagleBoard. Since each exception vector only has 32-bit space, obviously we have to make a jump in each vector to the "real PRECODE". Listing 4.5 shows the code making this jump. This part is totally dependent on the hardware architecture.

Listing 4.5 Code redirecting exception to handler address

```
.global __exception_handlers
__exception_handlers:
      b      reset_vector
      ldr    pc,.undefined_instruction
      ldr    pc,.software_interrupt
      ldr    pc,.abort_prefetch
      ldr    pc,.abort_data
      .word   0
      ldr    pc,.IRQ
      ldr    pc,.FIQ
.global vectors
vectors:
   .undefined_instruction: .word undefined_instruction
   .software_interrupt:    .word software_interrupt
   .abort_prefetch:        .word abort_prefetch
   .abort_data:            .word abort_data
                           .word 0
   .IRQ:                   .word IRQ
   .FIQ:                   .word FIQ
```

Listing 4.6 is the "real PRECODE" in the eCos distribution on BeagleBoard. The reset exception is different from others thus need to be written separately. The other three exception handlers have the same structure except saving different exception types in register "r2". The function "call_exception_handler" from line 27 to line 29 saves the context before the exception then call function "exception_handler", which is considered as the "general internal exception handler" in this example.

Listing 4.6 Internal exception handlers 1st stage process

```
1.        .global reset_vector
2.  reset_vector:
3.        ...        // initialize platform, then jump to main function
4.  undefined_instruction:
```

```
5.      ldr      sp,.__undef_exception_stack
6.      stmfd  sp!,{r0-r5}
7.      mrs     r1,spsr
8.      sub     r0,lr,#4
9.      mov     r2,#CYGNUM_HAL_EXCEPTION_ILLEGAL_INSTRUCTION
10.     mov    r3,sp
11.     b        call_exception_handler
12. software_interrupt:
13.     …       //save current context
14.     mov    r2,#CYGNUM_HAL_EXCEPTION_INTERRUPT
15.     …
16.     b      call_exception_handler
17. abort_prefetch:
18.     …       //save current context
19.     mov    r2,#CYGNUM_HAL_EXCEPTION_CODE_ACCESS
20.     …
21.     b      call_exception_handler
22. abort_data:
23.     …       //save current context
24.     mov    r2,#CYGNUM_HAL_EXCEPTION_DATA_ACCESS
25.     …
26.     b      call_exception_handler
27. call_exception_handler:
28.    …       //save the context before the exception
29.     bl      exception_handler
```

Listing 4.7 is an implementation of 2nd stage process, i.e. "general internal exception handler" in figure 4.1 . All the information is saved in structure "regs" and passed to the function "exception_handler( )" in line 1. Then it gets the exception handler from the array "exception_handler[ ]" in line 9 and invoke that handler to handle the exception.

Listing 4.7 Internal exception handlers 2nd stage process

```
1.  void exception_handler(HAL_SavedRegisters *regs){
2.     cyg_hal_deliver_exception( regs->vector, (int)regs );
3.     return;
4.  }
5.  void cyg_hal_deliver_exception( int code, int data ){
6.     Cyg_Thread::self()->deliver_exception( (cyg_code)code, data );
7.  }
8.  void Cyg_Exception_Control::deliver_exception(int  exception_number,int exception_info){
9.     handler = exception_handler[exception_number - CYGNUM_HAL_EXCEPTION_MIN];
10.    data = exception_data[exception_number - CYGNUM_HAL_EXCEPTION_MIN];
11.    handler( data, exception_number, exception_info );
12. }
```

Listing 4.8 shows an example of "POSTCODE". After restoring the context before exception in line 3, the program returns to the normal execution flow.

Listing 4.8 Internal exception handlers 3rd stage process

```
return_from_exception:
        msr     spsr,r0
        ldmeqfd sp,{r0-r14,pc}^
```

4.2 CPU-related features

In a CPU, there are supporting components around the core to perform instruction execution and respond to the external events, such as memory, interrupt controller, clock, etc. The CPU-related features contain the code for controlling these components.

4.2.1 Clock generation and control

The clock signal is used to synchronize the action of circuits. It is usually generated by a hardware component called clock generator. A typical clock generator receives one or a few input reference clock signals from an oscillator or an external chip, meanwhile outputs several different frequency clock signals to on-chip hardware components and/or off-chip peripherals.

In this section, stable input reference clock signals are assumed.

4.2.1.1 General model

The clock generator usually distributes different frequency signals. For instance, the clock signal for MPU always has a much higher frequency than that for peripherals. Furthermore each output usually can be configured as an optional frequency. So the function which configures the frequency of each output respectively is required as well as a function for selecting input reference clock. Functions for shutting down and turning on each clock output are also necessary for reducing power consumption which is of critical importance in mobile systems.

Figure 4.2 show the general architecture of the clock module. First it receives several different frequency reference signals and dispenses them to each channel, based on the system configuration. Then each channel generates a demanded clock signal of certain frequency and supplies a group of hardware components.

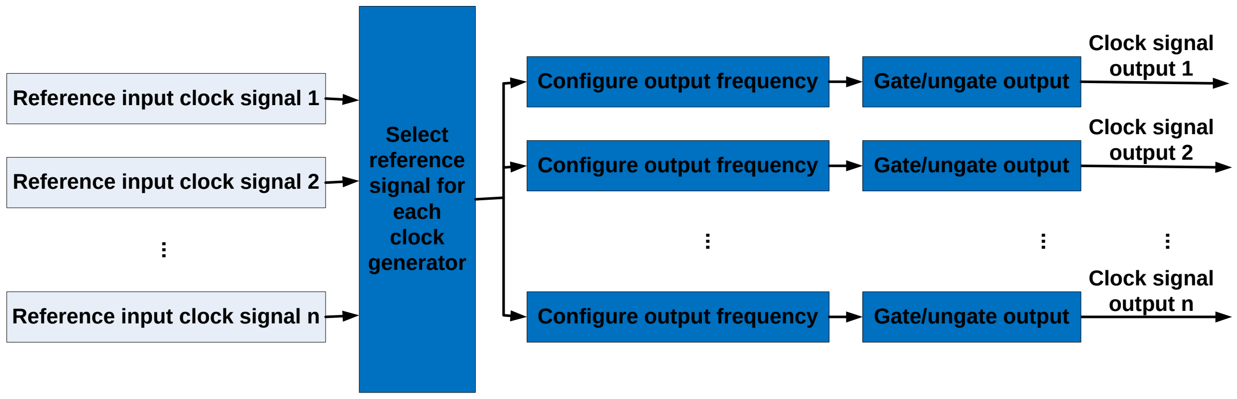

Reference input clock signal 1
Reference input clock signal 2
Reference input clock signal n
Select reference signal for each clock generator
Configure output frequency
Gate/ungate output
Clock signal output 1
Clock signal output 2
Clock signal output n

Figure 4.2 The clock module architecture

The functions/macros controlling clock module are listed in table 4.3. The function “clock_input_select( )” configures the clock module to receive a certain reference signal from a physical pin. The frequency of that signal is usually known by the developer. The function “output_config (channel, frequency)” sets the “channel” to output a clock of “frequency”. These two functions are commonly used in early system initialization process since the stable clock signals are the foundation of system execution. However the latter function is possibly used in run time when frequency adjustment is required for making trade-off between executing power and energy consuming. The function “clock_enable(channel)” and “clock_disble(channel)” start or stop the clock signal “channel” output. They can be implemented as gating/ungating the channel while keeping the channel active internally, or even shut down the entire channel and restart it, based on the system design.

Table 4.3 Functions/macros of clock control HAL

Function/Macro	Comments
void clock_input_select(void)	Choose the reference clock signal from multiple input signal sources.
void output_config (channel, frequency)	Set the clock output channel output the “frequency” signal.
void clock_enable (channel)	Enable the output clock signal “channel”.
void clock_disable (channel)	Gate the output clock signal “channel” for energy saving.

4.2.1.2 Clock generation and control for eCos example

Listing 4.9 shows an example from an eCos distribution on the Innovator (ARM) Board. It is included in platform initialization process. This small piece of code initializes and enables the DPLL1 clock signal output.

Listing 4.9 DPLL1 module initialization

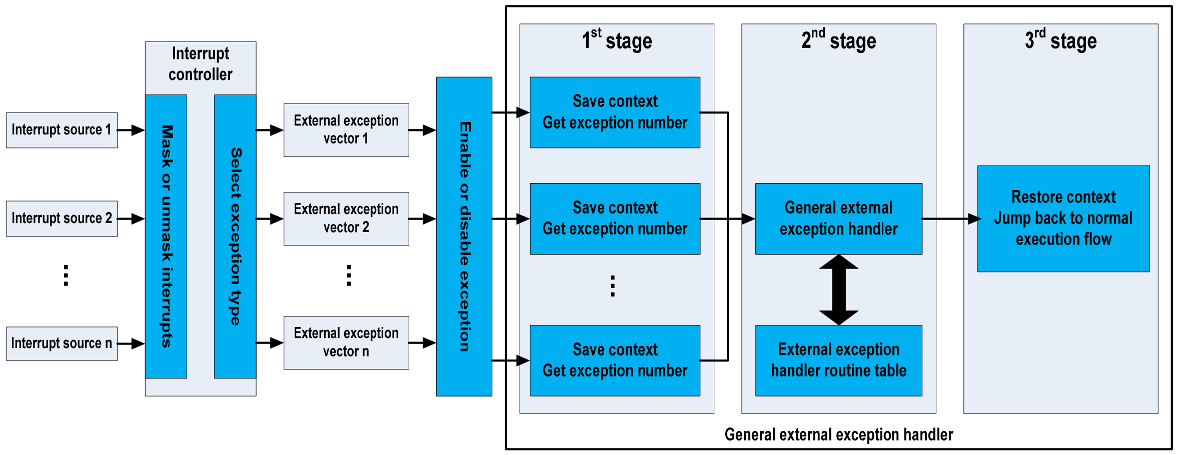
```
        ldr     r1,=DPLL1_BASE
        ldr     r2,=0x2290
        str     r2,[r1,#_DPLL_CTL_REG]
1:      ldr     r2,[r1,#_DPLL_CTL_REG]
        and     r2,r2,#1
        cmp     r2,#1
        bne     1b
```

4.2.2 External exception handler

An external exception, also known as an interrupt, is generated by a hardware component and delivered to CPU through the interrupt controller. Usually it is maskable and has lower priority than internal exceptions.

4.2.2.1 General model

The blue blocks in figure 4.3 illustrate the components of external exception handler. This HAL module involves operation on two hardware components, i.e. programmable interrupt controller and CPU. We assume that every interrupt source has a unique number.



Interrupt source 1
Interrupt source 2
Interrupt source n
Interrupt controller
Mask or unmask interrupts
Select exception type
External exception vector 1
External exception vector 2
External exception vector n
Enable or disable exception
1st stage
2nd stage
3rd stage
Save context Get exception number
Save context Get exception number
Save context Get exception number
General external exception handler
External exception handler routine table
Restore context Jump back to normal execution flow
General external exception handler

Figure 4.3 The architecture of general external exception handler

Apparently the external exception handler has a similar 3-stage general handler as internal exception handler. The differences are that the handler here gets and saves the exception number from interrupt controller at the 1st stage and checks a different "external exception handler routine table" at the 2nd stage to get the corresponding service function.

Table 4.4 lists the features of external exception handling HAL, which is quite similar to the internal exception handling HAL, plus the interrupt controller operations. The first 5 functions or macros in table 4.4 do the same thing as their counterparts in internal exception handler. The rest functions are related to interrupt controller. The difference among "enable_expt_signal ( )", "disable_expt_signal ( )", "mask_int ( )" and "unmask_int ( )" is the former two functions control the signal between interrupt controller and CPU while the latter two control the signal between other hardware components and interrupt controller.

Table 4.4 Functions/macros of external exception handling HAL

Functions/Macro	Comments
PRECODE	This macro saves context and records exception type and address.
POSTCODE	This macro restores the context saved before then goes back to normal flow.
void call_handler (expt_num)	Invoke the exception handler corresponding to the number.
void attach_handler (expt_num, handler)	Attach an exception handler to the corresponding position in routine table.
void detch_handler (expt_num)	Set corresponding position in routine table to NULL.
void enable_expt_signal (expt_signal_type)	Enable the exception “expt_signal_type” in CPU.
void disable_expt_signal (expt_signal_type)	Disable the exception “expt_signal_type” in CPU.
void set_type (int_num, expt_signal_type)	Set interrupt “int_num” to generate the “expt_signal_type” interrupt signal to CPU.
void mask_int (int_num)	Mask the interrupt input signal from an external component to the interrupt controller with number “int_num”.
void unmask_int (int_num)	Unmask the interrupt input signal with number “int_num”.
void clear_int (expt_num)	Clear corresponding interrupt status bit after being handled to enable new interrupt generation

4.2.2.2 The external exception handler for eCos example

In listing 4.10, an external exception handler routine table “hal_interrupt_handlers” is established. Since eCos defined an object and data associated with each interrupt handler, a data table and an object table are established as well. The handler gets the data and object from corresponding position in these two tables. “CYGNUM_HAL_ISR_COUNT” is the amount of interrupt sources which in ARM DM3730 processor is 96.

Listing 4.10 External exception handler routine table

```
.globl  hal_interrupt_handlers
hal_interrupt_handlers:
        .rept   CYGNUM_HAL_ISR_COUNT
        .long   0
        .endr
.globl  hal_interrupt_data
hal_interrupt_data:
        .rept   CYGNUM_HAL_ISR_COUNT
        .long  0
        .endr
.globl  hal_interrupt_objects
hal_interrupt_objects:
        .rept   CYGNUM_HAL_ISR_COUNT
        .long  0
        .endr
```

The implementation of "PRECODE" is shown in listing 4.11. The function "hal_IRQ_handler" in line 3 is responsible to check the status register in interrupt controller and pass the interrupt number to the 2nd stage process by "v1".

Listing 4.11 External exception handlers 1st stage process

```
1.  FIQ:  ...    //disable IRQ and FIQ, save the context
2.  IRQ: …    //save the context
3.          bl      hal_IRQ_handler
```

In listing 4.12, from line 1 to line 7, it is the "general exception handler" which invokes the handler routine from table "hal_interrupt_handlers" defined in listing 4.10. Then in line 8, the "POSTCODE" is called to finish the interrupt handling and go back to normal execution flow.

Listing 4.12 External exception handlers 2nd stage process

```
1.          ldr       r1,.hal_interrupt_data
2.          ldr       r1,[r1,v1,lsl #2]
3.          ldr       r2,.hal_interrupt_handlers
4.          ldr       v3,[r2,v1,lsl #2]
5.          mov     r2,v6
6.          mov     lr,pc
7.          mov     pc,v3
8.          b         return_from_exception
```

Listing 4.13 shows the functions and macros for controlling interrupt delivery in the eCos distribution on BeagleBoard. Function "hal_interrupt_mask( )" and function "hal_interrupt_unmask( )" are used for masking/unmasking a specific interrupt source in interrupt controller. The other two macros, i.e. "HAL_DISABLE_INTERRUPTS (_old_)" and "HAL_ENABLE_INTERRUPTS( )" are for disabling or enabling external exception signals on CPU side. All these functions and macros are invoked many times in the kernel and applications.

Listing 4.13 Interrupt controller operations

```
void hal_interrupt_mask(int vector){
     int i,j;
    i=vector/32;
    j=vector%32;
```

```
      HAL_WRITE_UINT32(INTCPS_MIR_SET(i),(1<<j));
}
void hal_interrupt_unmask(int vector){
      int i,j;
      i=vector/32;
      j=vector%32;
     HAL_WRITE_UINT32(INTCPS_MIR_CLEAR(i),(1<<j));
}
#define HAL_DISABLE_INTERRUPTS(_old_)          \
   asm volatile (                              \
      "mrs %0,cpsr;"                           \
      "mrs r4,cpsr;"                           \
      "orr r4,r4,#0xC0;"                       \
      "msr cpsr,r4"                            \
      : "=r"(_old_)                            \
      :                                        \
      : "r4"                                   \
      );
#define HAL_ENABLE_INTERRUPTS()        \
   asm volatile (                      \
      "mrs r3,cpsr;"                   \
      "bic r3,r3,#0xC0;"               \
      "msr cpsr,r3"                    \
      :                                \
      :                                \
      : "r3"                           \
      );
```

4.2.3 Timer control

Timer is usually used for the OS and applications to perform schedule and synchronization. It receives a clock signal and by counting that signal up or down to a specific value generates an interrupt.

4.2.3.1 General model

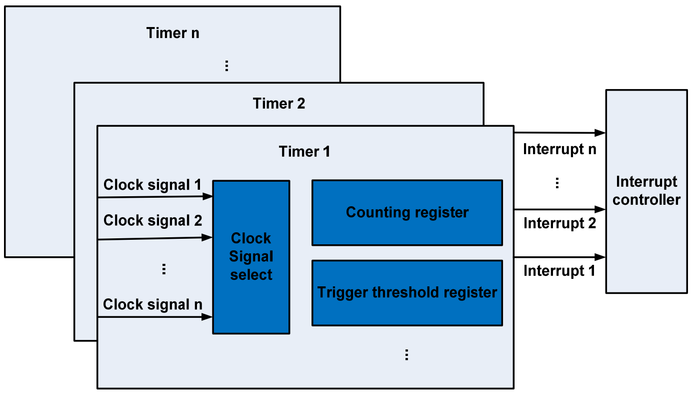


Figure 4.4 The general model of multiple timers within one processor

Figure 4.4 shows the common structure of the timer module. Usually in one processor there are multiple timers with the same architecture. The usage of each timer is determined by system designer. The differences among each timer configuration are primarily interrupt generation interval and repeatability.

Because of the identical function and architecture, we can use a set of functions to control multiple timers. Table 4.5 lists the operations within the timer HAL, with which programmers are able to get single or periodic interrupts. The first function in table 4.5 initializes a timer to work with desired behavior. The parameters for the function are determined by system demands. The next 3 functions are for run time. After being initialized, the timer doesn't have much control operation but just "start" and "stop", as well as the less important "get current value" operation. The last 4 functions are related to interrupt generation of the timer.

Table 4.5 Functions/macros of general purpose timer HAL

Functions/Macros	Comments
void init_timer (timer n, clock, period, iteration)	Initialize the "timer n", including selecting the reference signal, setting the interrupt period and setting the number of iteration.
void start_timer (timer n)	Start timer n.
void stop_timer (timer n)	Stop timer n.
current_val get_val_timer (timer n)	Read the current counting value from the corresponding register.
void enable_int_timer (timer n)	Enable the interrupt of timer n
void disable_int_timer (timer n)	Disable the interrupt of timer n.
void ISR_timer_m (void)*	The ISR of timer m. Using attaching function provided in interrupt controller HAL to attach it to the right place in the external exception handler routine table.
void clear_int_timer (timer n)	Clear the interrupt status bit to allow new interrupt generation.

* m indicates that there is an individual instance for each hardware component

4.2.3.2 Timer control for eCos example

Listing 4.14 shows the timer initialization routine for BeagleBoard. This function is simplified so that only applies to timer 10. However it implements all operations defined in function "init_timer( )" in table 4.5. After executing this function the timer 10 is in the proper status to generate demand interrupts but not start running yet.

Listing 4.14 Timer initialization routine

```
void hal_clock_initialize(void ){
stop_timer(timer);
int i;
HAL_READ_UINT32(0x48004A40,i);
```

```
i &= ~(1<<6);
HAL_WRITE_UINT32(0x48004A40,i);//select 32 khz as the input source
HAL_WRITE_UINT32(TIMER(10,TSICR),0);//non-posted mode
HAL_WRITE_UINT32(TIMER(10,TIER),2);//enable overflow int
HAL_WRITE_UINT32(TIMER(10,TCRR),0xFFFFFFE0);
HAL_WRITE_UINT32(TIMER(10,TLDR),0xFFFFFFE0);
HAL_WRITE_UINT32(TIMER(10,TTGR),0xFFFFFFFF);//load value into counter
HAL_WRITE_UINT32(TIMER(10,TMAR),0xFFFFFFFF);//march value to generate an int
HAL_WRITE_UINT32(TIMER(10,TPIR),232000);
HAL_WRITE_UINT32(TIMER(10,TNIR),-768000
HAL_WRITE_UINT32(TIMER(10,TOCR),0);//1ms timer int
HAL_WRITE_UINT32(TIMER(10,TOWR),0);
}
```

Listing 4.15 is the interrupt handler of a timer. It simply makes the counting register load a configured value from another register to resume counting and clear the interrupt flag to enable the next interrupt.

Listing 4.15 Interrupt handler of a timer

```
void hal_clock_reset(cyg_uint32 vector, cyg_uint32 period){
        HAL_WRITE_UINT32(TIMER(10,TISR),7);
        hal_interrupt_acknowledge_IRQ(CYGNUM_HAL_INTERRUPT_RTC);
        return;
}
```

Listing 4.16 is the function for reading current counting value. This function is feasible only if the device has the on-the-fly (while counting) read capability.

Listing 4.16 Reading the current counting value of the timer

```
void hal_clock_read(cyg_uint32 *pvalue){
        HAL_READ_UINT32(TIMER(10,TCRR),*pvalue);
}
```

4.2.4 Cache control

Cache is a small piece of memory within the processor memory system. It has very fast speed, almost as quick as CPU. The performance of cache usually has a big impact to that of overall system.

4.2.4.1 General model

Cache exists in almost all the embedded processors for improving system performance. It stores a computed value or duplicates of an area in main memory. When the CPU requests data, first of all it checks the cache. If the requested data is in cache (cache hit), the CPU will use the data in cache directly. If the data is not in cache (cache miss), the CPU will access the main memory which is much slower.

Figure 4.5 briefly illustrates the memory architecture of the Harvard architecture embedded processors. It has separate instruction cache and data cache in level one memory.

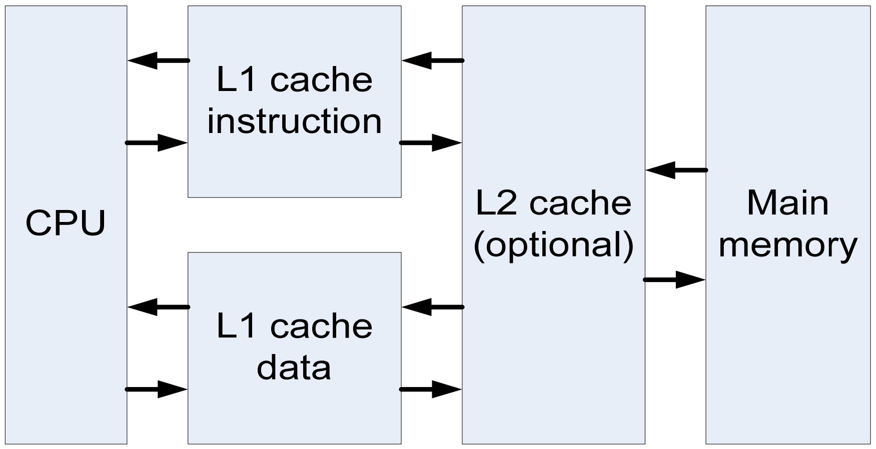


Figure 4.5 Memory hierarchy

For achieving best performance, cache is mostly operated by hardware. But under certain circumstance, software controlled cache maintenance is necessary, for instance, when another bus master instead of cores writes the memory. In general, when a coherency problem is potential, programmers need to invalidate or synchronize the cache lines. The common cache operations are listed in table 4.6.

Table 4.6 Functions/macros of cache control HAL

Function/Macro	Comments
void init_cache (void)	Initialize both data and instruction cache. Set configurable parameters.
ENABLE_ICACHE	Enable instruction cache.
ENABLE_DCACHE	Enable data cache.
DISABLE_ICACHE	Disable instruction cache.
DISABLE_DCACHE	Disable data cache.
FLUSH_ICACHE	Invalid instruction cache.
FLUSH_DCACHE	Invalid data cache.
SYN_ICACHE	Synchronize instruction cache and main memory.
SYN_DCACHE	Synchronize data cache and main memory.

4.2.4.2 Cache control for eCos example

Listing 4.17 is the code for enabling L1 instruction cache in ARM architecture. The L1 cache is controlled by CP15 coprocessor in ARM architecture.

Listing 4.17 Enabling instruction cache

```
#define HAL_ICACHE_ENABLE()          \
  asm volatile (                     \
     "mrc  p15,0,r1,c1,c0,0;"        \
     "orr  r1,r1,#0x1000;"           \
     "orr  r1,r1,#0x0003;"           \
     "mcr  p15,0,r1,c1,c0,0"         \
     :                               \
     :                               \
     : "r1"                          \
     );
```

Listing 4.18 is a macro for disabling instruction cache. The operation is also performed via writing several specific registers within CP15. Before disabling the instruction cache, a synchronization operation is necessary for avoiding coherency issues. The instructions on line 11 and 12, i.e. “nop”, are for the synchronization purpose. In newest ARM CORTEX A series processor, this method is not valid because of the out-of-order execution mechanism. The data and instruction barrier instructions can resolve this problem.

Listing 4.18 Disabling instruction cache

```
1.  #define HAL_ICACHE_DISABLE()              \
2.  CYG_MACRO_START                           \
3.    asm volatile (                          \
4.       "mrc    p15,0,r1,c1,c0,0;"           \
5.       "bic    r1,r1,#0x1000;"              \
6.       "mcr    p15,0,r1,c1,c0,0;"           \
7.       "mov   r1,#0;"                       \
8.       "mcr    p15,0,r1,c7,c5,0;"           \
9.       "mcr    p15,0,r1,c7,c5,6;"           \
10.      "mcr    p15,0,r1,c8,c5,0;"           \
11.      "nop;"                               \
12.      "nop;"                               \
13.      :                                    \
14.      :                                    \
15.      : "r1"                               \
16.      );
```

The data cache synchronization code is shown in listing 4.19. Since the new ARM architecture only supports synchronization on an individual line each time, we have to go through all the lines to synchronize the entire data cache.

Listing 4.19 Data cache synchronization

```
#define HAL_DCACHE_SYNC()                           \
for(level=1;level>0;level--)                        \
  for(way=3;way>=0;way--)                           \
          for(set=127;set>=0;set--)                 \
5.          {setway=(level << 1) | (set << 6) | (way << 30);         \
6.           asm volatile ("   mcr p15, 0, %0, c7, c14, 2;": : "r" (setway)); }
```

4.2.6 Memory management unit

Since the eCos doesn't support virtual memory, we are not able to verify the HAL for memory management feature recently. So this part of work is left to the future when we implement our HAL under a "rich" operating system which supports virtual memory.

4.2.7 Direct memory access (DMA)

Direct memory access is a feature that certain hardware can access memory independently of core. It is usually implemented by a dedicated hardware component called "DMA controller". With DMA, the core is released for other important work during the slow data transfer. Otherwise the core would be fully occupied by read and write instructions. DMA controller is widely found in modern embedded processors for efficient data transfer.

4.2.7.1 General model

DMA controller performs data transfer between memories or memory and peripheral. A DMA controller usually has several channels. Each channel can be configured to perform a specific data transfer task. The architecture of DMA controller is illustrated in figure 4.6.

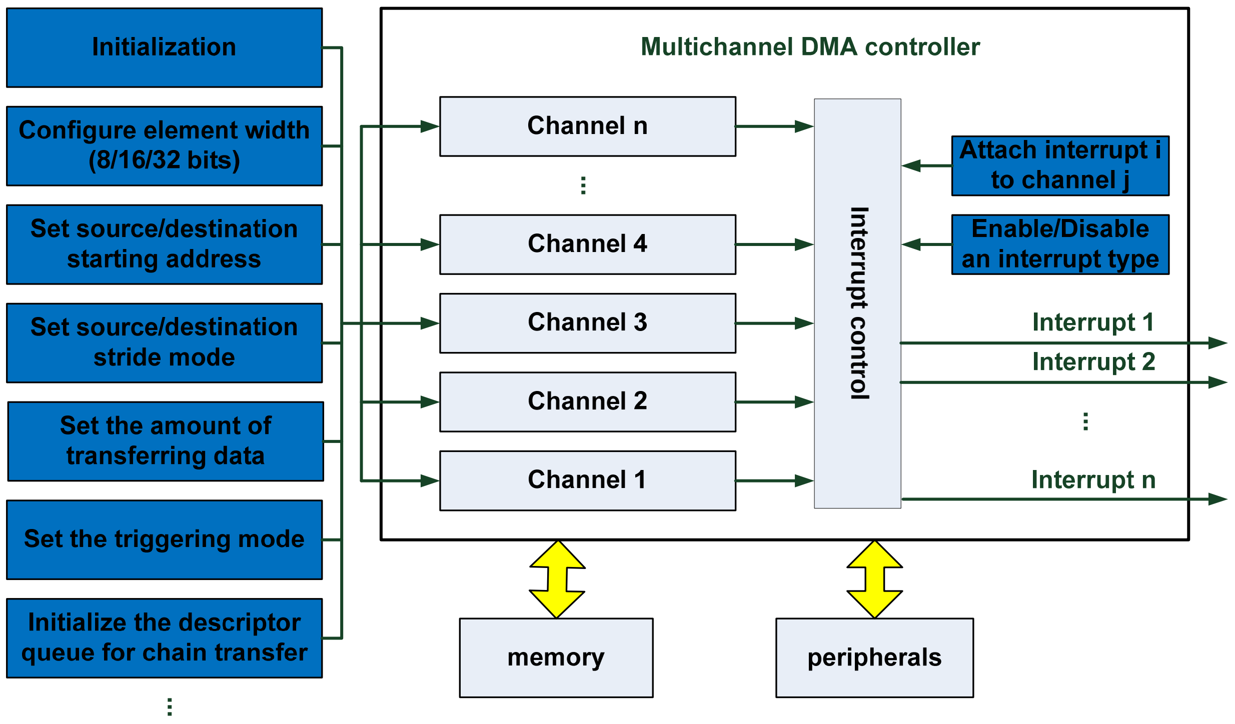

Initialization
Configure element width (8/16/32 bits)
Set source/destination starting address
Set source/destination stride mode
Set the amount of transferring data
Set the triggering mode
Initialize the descriptor queue for chain transfer
Multichannel DMA controller
Channel n
Channel 4
Channel 3
Channel 2
Channel 1
Interrupt control
Attach interrupt i to channel j
Enable/Disable an interrupt type
Interrupt 1
Interrupt 2
Interrupt n
memory
peripherals

Figure 4.6 The architecture of DMA controller

Before using any channel for data transfer, an overall initialization process should be implemented, in which a bunch of global parameters are set, for instance, the biggest burst transfer size. These parameters are common for all channels and usually not change during execution. After that, each channel can be configured for specific use, such as transferring data from Ethernet receiving buffer to memory which is classified as peripheral to memory transfer. Other transfer types are memory to memory and memory to peripheral. For implementing a transfer, several control parameters, such as element width, source/destination address, triggering source, etc, should be configured, as well as the interrupt service routines (ISRs). The functions for implementing DMA data transfer are shown in table 4.7.

Table 4.7 Functions/macros of DMA controller HAL

Function/Macro	Comments
void init_global_parameters (void)	Set common configurations for all channels, such as power saving strategy.
void set_type (channel ,type)	Set the source and destination type of the "channel", e.g. memory to peripheral, etc.
void set_element_width (channel , width)	Set the width of transferring element of "channel" to 8, 16, 32 or 64 bits.
void set_source_addr (channel, addr)	Set the source starting address of the "channel".
void set_destination_addr (channel, addr)	Set the destination starting address of "channel".
void ISR_m (void)*	Interrupt service routine for channel "m".
void enable_int (channel,event)	Enable the "channel" to generate an interrupt when "event" occurs.
void disable_int (channel,event)	Prevent the "channel" from generating an interrupt when "event" occurs.

*m indicates that there is an individual instance for each channel

4.2.7.2 Direct memory access control for eCos example

Listing 4.20 is a function which set the element width for DMA channel on TI DM3730 processor. The DMA within this chip supports 8, 16 or 32 bits element transfer. Furthermore each channel can have different element width.

Listing 4.20 Transferring element width configuration

```
void SDMA_set_element_width(unsigned int channel,unsigned int width){
        switch(width){
        case 8:
                clear_bit(DMA4_CSDP(channel),0);
                clear_bit(DMA4_CSDP(channel),1);
                break;
        case 16:
                set_bit(DMA4_CSDP(channel),0);
                clear_bit(DMA4_CSDP(channel),1);
                break;
        case 32:
                clear_bit(DMA4_CSDP(channel),0);
                set_bit(DMA4_CSDP(channel),1);
                break;
        }
}
```

Listing 4.21 shows two functions which set the source and destination address of a channel. Any one of them can be a memory address or a peripheral buffer register address.

Listing 4.21 Source and destination starting address configuration

```
void SDMA_set_src_addr(unsigned int channel, void* src_addr){
        write_reg(DMA4_CSSA(channel),(unsigned int)src_addr);
}
void SDMA_set_dst_addr(unsigned int channel, void *dst_addr){
        write_reg(DMA4_CDSA(channel),(unsigned int)dst_addr);
}
```

4.3 Peripheral-related features

In modern embedded systems most peripheral I/O devices are integrated into the microcontroller and mapped into the physical address space. The operation of reading data from or writing data to these devices is as same as reading and writing memory. In general, there are four kinds of registers of I/O devices: control register, status register, input register and output register. CPU communicates with the I/O devices by reading or writing these registers, as shown in Figure 4.7.

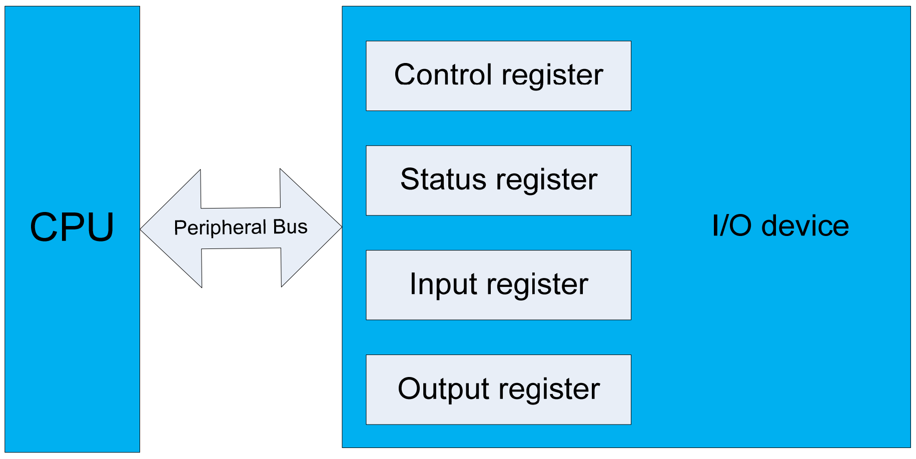


Figure 4.7 Registers of I/O devices

The components within a microcontroller differ in different application areas. For example, CAN module is mostly found in the automotive/industrial microcontroller, while the Ethernet module is common in consumer electronics. But generally the drivers of these devices could be separated into two parts: hardware-independent layer and hardware-dependent layer. The hardware-dependent layer, which is hardware abstraction layer, controls the device directly by reading and writing the registers and provides APIs to upper part, as shown in Figure 4.8.

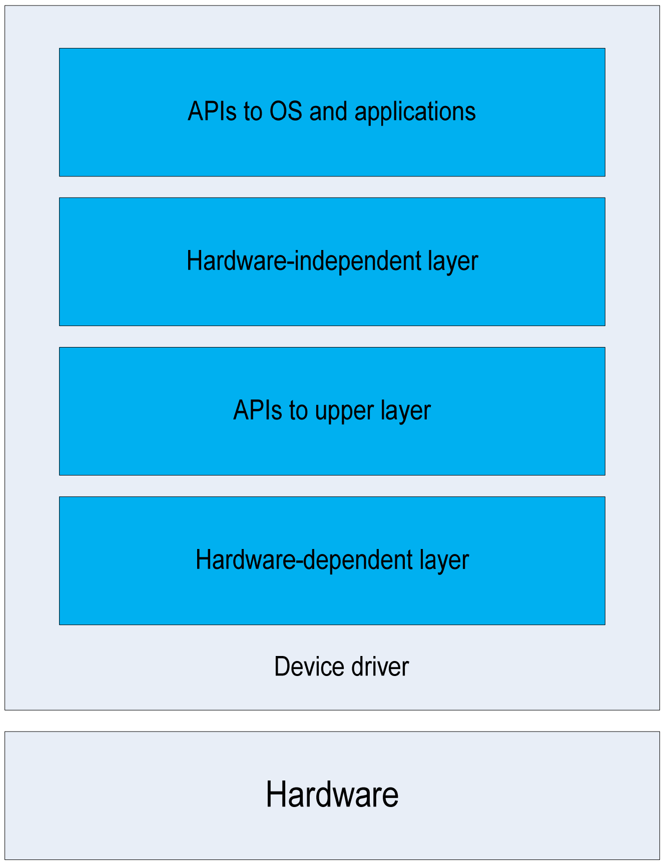


Figure 4.8 Architecture of device drivers

The HAL of device driver can be split into two parts. One takes care of communication with CPU, defined as internal interface in this section, while the other takes care of communication with other systems, defined as external interface. The external interface is an implementation of a protocol, for instance, UART module implements UART protocol, memory card module implements MMC or SD protocol. Before using the device, the protocol defined parameters should be configured, such as data bits in UART protocol and MAC address in Ethernet protocol.

Since the parameters of different protocols are of significant difference, we consider an UART module which implements standard UART protocol as an example. For communicating with another system which has UART as well, the baud rate, data bits, parity, and stop bits should be as same as that system. The table 4.8 shows the configuration functions for communicating between systems with UART protocol.

Table 4.8 An example of protocol-defined parameters configuration HAL: UART protocol parameter configuration functions

Protocol parameter	Configuration function	Comment
baud rate	void set_baudrate (baudrate)	Set the baud rate of UART module transmitting/receiving.
data bits	void set_databits (length)	Set the length of data in one frame, typical number is 7 or 8.
parity	void set_parity (type)	Choose from odd, even or none parity.
stop bits	void set_stopbits (length)	Set the length of stop session, typical number is 1, 1.5 or 2.
flow control	void set_flowcontrol (type)	Choose from hardware, software or none flow control.

Besides these protocol parameter configuration functions, a few other functions are necessary to make the device work. Some of the I/O devices are able to support several protocols, for instance, universal asynchronous receiver/transmitter (UART), infrared data association (IrDA) and consumer infrared (CIR) can be integrated into one hardware module, multimedia card (MMC) and secure digital (SD) card interfaces are always integrated into one module as well. Thus, for a multifunction device, a mode selection function is needed. Then, the interrupt enable/disable functions which control the interrupts raised by protocol specified events, such as parity error interrupt, are provided, as well as the interrupt service routine (ISR) for these events. The configuration functions are listed in table 4.9.

Table 4.9 An example of hardware module configuration

Functions/Macros	Comments
void mode_selection (mode)	Set the multifunction device to the right mode.
void enable_int (interrupt event)	Enable the xxx event interrupt. xxx event is defined in protocol.
void disable_int (interrupt event)	Disable the “event interrupt”. xxx event is defined in protocol.
void ISR_events (void)	Interrupt service routine for protocol specified events

The internal interface is similar among all devices. The width of input/output register differs between chips and components. We define the data that can be written into output register or read from input register at one time as a transfer unit. If an IO device transfers one unit at a time, it is defined as a word-transfer device. If multiple units are transferred at a time, it is a block-transfer device. Some devices are able to work in either of the two modes. Then we consider the device has the type as its current working mode, e.g. if the device is configured to transfer one unit after another, it is a word-transfer device, otherwise it is a block-transfer device. Programmers have the choice to provide the code supporting either or both modes.

Usually, there are three different methods for reading/writing the input/output register, e.g. polling, interrupt and DMA. So, theoretically, there are 6 HAL models, e.g. polling word transfer, interrupt word transfer, DMA word transfer, polling block transfer, interrupt block transfer and DMA block transfer. However, interrupt word transfer, DMA word transfer and polling block transfer model are not feasible. Thus we only talk about the other three feasible models. In each section below, we assume that the transferring model is supported by hardware.

4.3.1 Polling word transfer

In polling word transfer model, software checks the status of read/write buffer before accessing the device. If the buffer is available, software reads one word from or writes one word to the corresponding register. Then it checks the status again if this is a multiple word transfer. Table 4.10 lists a set of functions for polling transfer. First of all, the hardware module has to be configured as working in this mode, which is done by function "set_model_PW". When using the I/O device, the program checks the status of device buffer by function "check_receive_reg" and "check_transmit_reg" respectively then reads or writes the device by the rest two functions when it is available.

Table 4.10 Functions/macros for polling word transfer

Functions/Mcros	Comments
void set_model_PW (void)	Set the device working on polling word transfer model.
status check_receive_reg (void)	Check the status of receiving register to see if there is data in buffer waiting for being read.
status check_transmit_reg (void)	Check the status of transmitting register to see if the buffer is available.
value read_word (void)	Read one word from receiving buffer.
void write_word (value)	Write one word to transmitting buffer.

Listing 4.22 shows two functions for writing or reading data to or from the UART module buffer. In line 3 and line8, program continues checking the status of buffer until it becomes available then performs the reading or writing operation.

Listing 4.22 Data transmitting/receiving functions for UART module on BeagleBoard

```
1.  void output_char(char c)
2.  {
3.    while ((read_serial(LSR) & 0x20) == 0) ;
4.    write_serial(THR, c);
5.  }
6.  Int receive_char(void)
7.  {
8.    while ((read_serial(LSR) & 0x01) == 0) ;
9.    return(read_serial(RHR));
}
```

4.3.2 Interrupt block transfer

In interrupt block transfer model, the read/write operations are triggered by interrupt and handled in ISR. For improving efficiency, an output queue and an input queue are established in memory. Applications only access output queue and input queue. ISR is responsible for transferring data between transmitting/receiving register and output/input queue. The protocol specified events interrupt service routines are included if available. The functions for supporting interrupt block transfer are shown in table 4.11.

Table 4.11 Functions/macros for interrupt block transfer

Functions/Macros	Comments
void set_model_IB (void)	Set the device working on interrupt block transfer model.
void enable_int (void)	Enable device to generate an interrupt when receiving or transmitting buffer is available.
void disable_int (void)	Disable the device from generating any interrupt.
void ISR (void)	In ISR, program checks the interrupt source. Then It writes all elements in output queue to transmitting buffer or reads the data from receiving buffer and writes to input queue according to the interrupt source.
value read_from_queue (void)	Get a word from the head of input queue. It is called in ISR.
void write_to_queue (value)	Write a word to the rear of output queue. It is called in ISR.
void transmit (data)	Write data into output queue then enable the transmit interrupts to accomplish transmitting.
data receive (void)	Read data from the input queue.
status check_receive_reg (void)	Check the status of receiving register to see if there is data in buffer waiting for being read.
status check_transmit_reg (void)	Check the status of transmitting register to see if the buffer is available.

Listing 4.23 is an interrupt service routine for UART interrupt mode transfer. It reads data from receiving buffer and writes them into a memory buffer.

Listing 4.23 Interrupt service routine for UART module on BeagleBoard

```
Void cyg_hal_plf_serial_isr (void){
        while((read_serial(LSR)&1)!=0){
                (*Rx_buf_head)=(unsigned char)(read_serial(RHR)&0xFF);
                Rx_buf_head++;
    if(Rx_buf_head==&(Rx_buffer[200]))
```

```
        Rx_buf_head=&(Rx_buffer[0]);}
    hal_interrupt_acknowledge_IRQ(CYGNUM_HAL_INTERRUPT_UART3);
        return;
}
```

4.3.3 DMA block transfer

In DMA block transfer, program initiates a DMA channel before transfer. Then the DMA controller will take charge of scheduling transfers. The CPU is released to execute other programs so that an amount of CPU cycles are saved from transferring data. Table 4.12 lists the functions for supporting DMA block transfer, which invoke DMA HAL to set up a DMA transfer.

Table 4.12 Functions/macros for DMA block transfer

Functions/Macros	Comments
void set_model_DMAB (void)	Set the device working in DMA block transfer mode.
void set_transmit_channel (channel, type, sync, source, destination, length, width)	Initialize a DMA channel for transmitting. This function invokes configuration functions in DMA HAL. “sync” means the hardware synchronizing source.
void ISR (void)	The ISR here deals with some unexpected situation such as buffer overflow but not regular read/write operations.
void enable_int (channel, event)	Enable the interrupt of channel “channel_num” and attach “ISR”.
void disable_int (channel, event)	Disable the interrupt of channel “channel_num”.
void start_trans (channel)	Start the transmitting by starting initialized “channel” and the peripheral.

Listing 4.24 is an example of initializing a DMA channel on TI DM3730 processor. It eliminates some special feature provided by this device but implements a common DMA transfer.

Listing 4.24 Initializing a DMA channel

```
void SDMA_init_channel(unsigned int channel, DMA4_trans_type type, unsigned int sync,
                    void *src_addr,void *dst_addr,unsigned int length,unsigned int width){
        int i;
        write_reg(DMA4_CLNK_CTRL(channel),0);//disable linking
        SDMA_disable_int(channel,type,DMA4_ALL);
        write_reg(DMA4_CSDP(channel),0x14000);
        write_reg(DMA4_CEN(channel),64);//64 elements in a frame
        write_reg(DMA4_CFN(channel),length/64);//number of frames in a block
        switch(type){
        case MtoM:
                write_reg(DMA4_CCR(channel),0x1045000);
        case MtoP:
                write_reg(DMA4_CCR(channel),0x801020);//frame sync
        case PtoM:
                write_reg(DMA4_CCR(channel),0x1045000);
        }
        if(sync!=0){
                for(i=0;i<=6;i++){
                        if((sync&(1<<i))==0)
                                if(i<=4)
                                        clear_bit(DMA4_CCR(channel),i);
                                else
                                        clear_bit(DMA4_CCR(channel),i+14);
                        else
                                if(i<=4)
                                        set_bit(DMA4_CCR(channel),i);
                                else
                                        set_bit(DMA4_CCR(channel),i+14);
                }
        }
        SDMA_set_element_width(channel,width);
        SDMA_set_src_addr(channel,src_addr);
        SDMA_set_dst_addr(channel,dst_addr);
        SDMA_enable_int(channel,type,DMA4_EOB);
}
```

5. Case Study of a MP3 Player on BeagleBoard

This section presents the experiment to verifying the architecture proposed above. We implement a mp3 player case which is based on BeagleBoard (7) and eCos (6) embedded operating system. The system configuration is shown in figure 5.1.

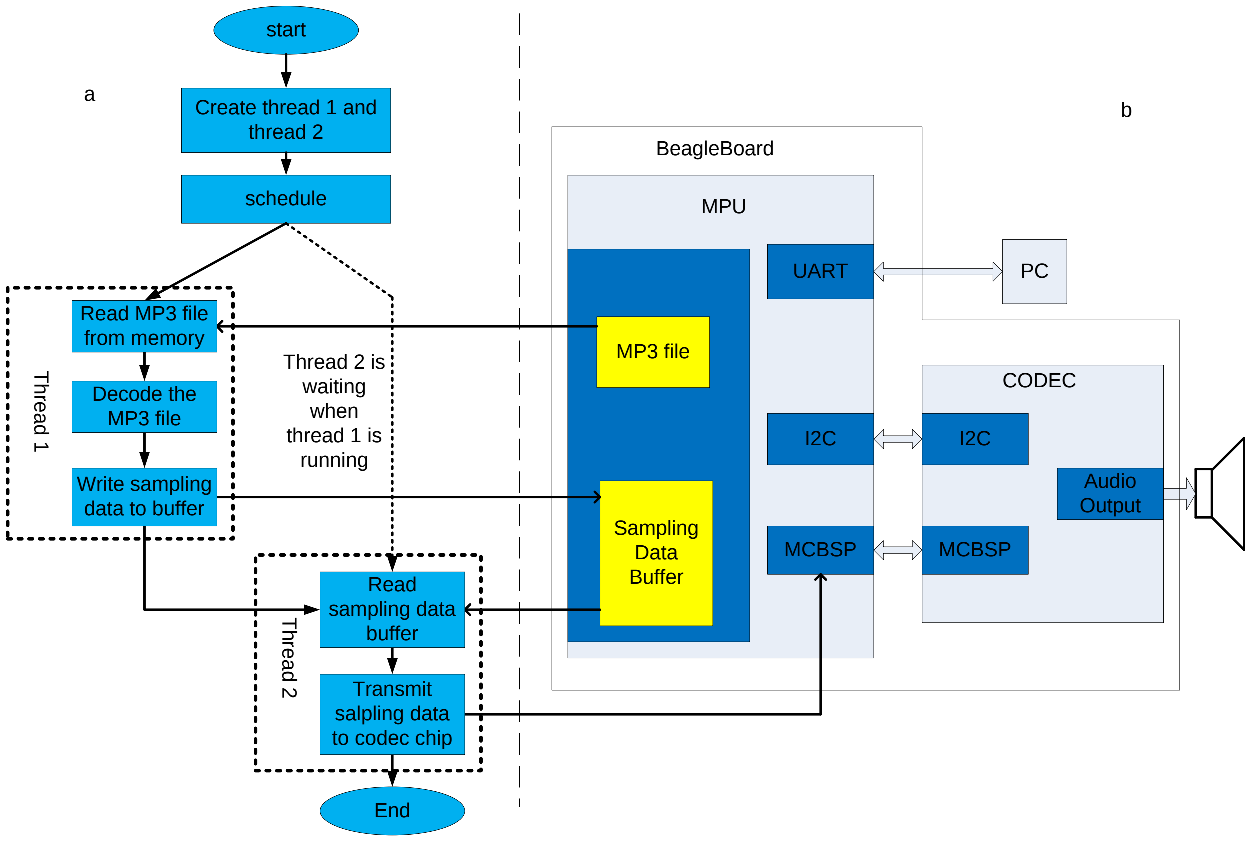

start
a
Create thread 1 and thread 2
schedule
Read MP3 file from memory
Decode the MP3 file
Write sampling data to buffer
Thread 1
Thread 2 is waiting when thread 1 is running
Read sampling data buffer
Transmit salpling data to codec chip
Thread 2
End
b
BeagleBoard
MPU
UART
PC
MP3 file
I2C
MCBSP
Sampling Data Buffer
CODEC
I2C
MCBSP
Audio Output

Figure 5.1 The system configuration of MP3 player experiment

5.1 Hardware configuration

Figure 5.1b depicts the hardware configuration. This experiment involves three serial communication ports and components within CPU. UART is used for console channel which receives command from and sends debug information to PC. I2C is the control channel for codec through which we can configure the audio chip. MCBSP is the audio data channel which is responsible for transmitting the sampling data to codec chip.

5.2 Software configuration

Figure 5.1a depicts the application software which contains two threads. Thread 1 decodes the MP3 file while thread 2 transmits the sampling data to audio chip through MCBSP channel. Both threads are created at the beginning of the execution when thread 1 has the higher priority. Since we didn't implement the storage device driver as well as the file system, the MP3 file is hard coded into the program. So the thread 1 just reads MP3 file from memory then writes the decoded sampling data back to memory which keeps the experiment straightforward. After thread 1 finished its job it suspends itself so that eCos scheduler is triggered which activates thread 2. Thread 2 invokes the interface defined in I2C and MCBSP HAL to control the audio chip.

5.3 Implementation

We carry out our experiment in the way that first write the HAL for the BeagleBoard and the application on eCos independently, and then make necessary changes in eCos kernel to make it work with our HAL. Thanks to eCos' well-designed architecture, it is relatively straightforward to adapt its interface to our HAL. The entire work contains about 2500 lines of code (LOC), most of which are hardware dependent.

For example, in eCos, the scheduler call the function "hal_thread_switch_context ( )" for switching the hardware context. In our HAL, we defined a function "switch_context( )" which is shown in section 3. So by making the former function call the latter one, context switch is implemented without any modification.

Another example is the console communication in eCos. The eCos uses a mechanism called virtual vectors to control and utilize the communication channels. Essentially the virtual vectors are pointers to service functions and data. The eCos uses an array of 64 pointers to store these vectors, which is called virtual vector table (VVT). The default configuration of VVT is shown in figure 5.2.

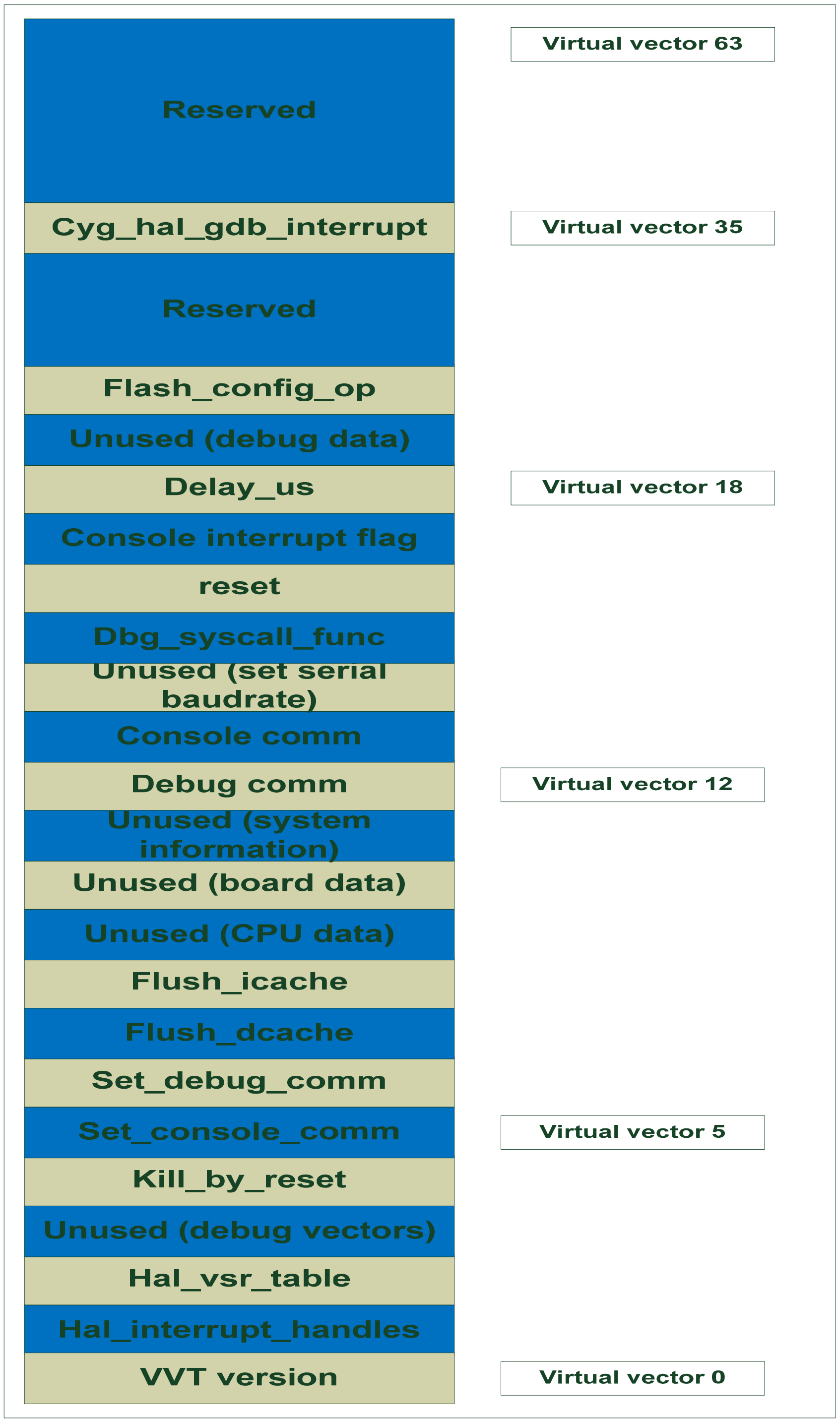


Figure 5.2 The default configuration of virtual vector table (VVT)

From figure 5.2 we can see that there are two virtual vectors are associated with the console communication, e.g. virtual vector 5 and virtual vector 13. Virtual vector 5 points to a function which associates a group of services supporting console communication to virtual vector 13. By default console channel uses the address stored in virtual vector 13 to get the service routines for communicating with PC. Furthermore, the eCos uses a structure called communication interface table (CIT) to model each communication channel. The CIT is essentially an array which contains 7 pointers pointing to the service functions or data associated with a specific communication channel. Figure 5.3 illustrates the structure of the CIT associated with the UART which is used for console channel in our system.

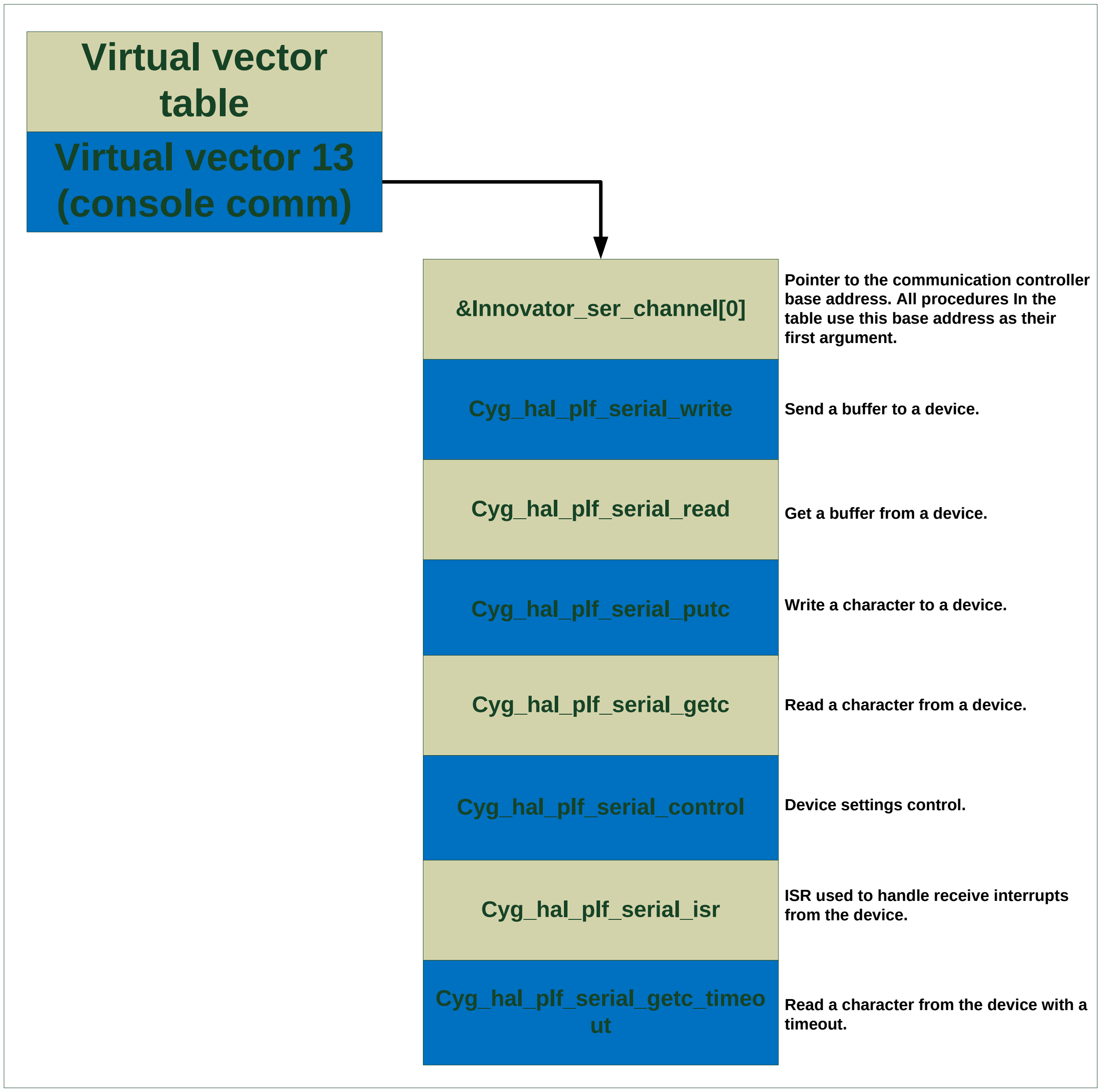

Virtual vector table
Virtual vector 13 (console comm)
&Innovator_ser_channel[0]
Pointer to the communication controller base address. All procedures In the table use this base address as their first argument.
Cyg_hal_plf_serial_write
Send a buffer to a device.
Cyg_hal_plf_serial_read
Get a buffer from a device.
Cyg_hal_plf_serial_putc
Write a character to a device.
Cyg_hal_plf_serial_getc
Read a character from a device.
Cyg_hal_plf_serial_control
Device settings control.
Cyg_hal_plf_serial_isr
ISR used to handle receive interrupts from the device.
Cyg_hal_plf_serial_getc_timeout
Read a character from the device with a timeout.

Figure 5.3 Communication interface table (CIT) of UART

For the UART channel on our BeagleBoard, first we implement the functions defined in table 4.8 which are protocol-specific configurations and operations. Then the functions listed in table 4.9 and 4.12 are implemented as well. So far the control functions we have are listed in table 5.1.

Table 5.1 Functions of UART HAL for BeagleBoard

Number	Function
1	void set_baudrate (baudrate)
2	void set_databits (length)
3	void set_parity (type)
4	void set_stopbits (length)
5	void set_flowcontrol (type)
6	void mode_selection (mode)
7	void enable_int (interrupt event)
8	void disable_int (interrupt event)
9	void ISR_events (void)
10	void set_model_IB (void)
11	void enable_int (void)
12	void disable_int (void)
13	void ISR (void)
14	value read_from_queue (void)
15	void write_to_queue (value)
16	void transmit (void)
17	void receive (void)
18	status check_receive_reg (void)
19	status check_transmit_reg (void)

Now comparing figure 5.3 and table 5.1, all the system calls listed in figure 5.3 can be composed by the functions listed in table 5.1. The pointer "&Innovator_ser_channel[0]" can be ignored without any impact to the system. With function 1-8 and 10-12 in table 5.1 we can compose the system call "cyg_hal_plf_serial_control( )" which configures the UART to the desired mode. Similarly we use function 9 and 13 to compose "cyg_hal_plf_serial_isr", use function 14 to compose "cyg_hal_plf_serial_getc", etc.

5.4 Issues encountered and solutions

During the experiment, we met some new techniques (to us) and some small tricks which cost us a little time to cope with. These are summarized in below subsections.

5.4.1 Remote debugging with GDB remote serial protocol

As this MP3 player is built on bare metal platform, e.g. no OS is available, it is a good choice to use remote debugging method. A GDB stub is already included in the Redboot which is a boot loader associated with eCos. What we need to do is modifying the hardware dependent code of the GDB stub to make it work on the BeagleBoard. Then we are able to debug the OS or application which is cross compiled on host PC for target hardware.

Figure 5.4 shows the concept of GDB serial protocol based remote debugging. In this scenario, most of the debugging functions are still provided by GDB debugger running on the host machine which is also called GDB host. The GDB stub runs on the target and communicates with the GDB host via the serial port.

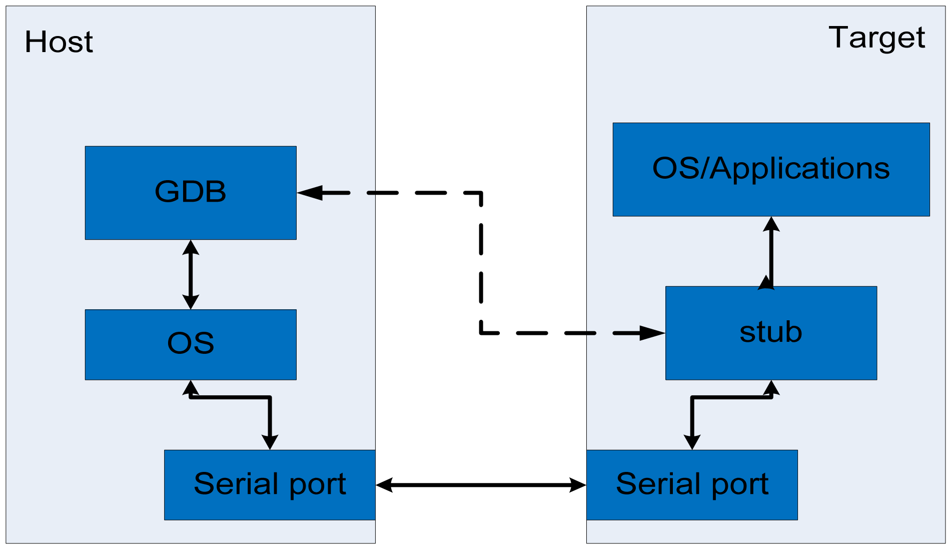


Figure 5.4 Remote debugging via GDB

The GDB stub basically consists of exception handlers and support for these handlers. In our system, there are two exception handlers: the handler for break point instruction and the handler for “Ctrl+c” message. The former communicates to GDB host whenever meeting a breakpoint instruction. The latter is essentially an interrupt handler for serial communication which directs the program to GDB stub when detecting a “Ctrl+c” message.

The communication between GDB host and GDB stub is subject to GDB remote serial protocol (RSP), which is an ASCii message based protocol (8). The basic format of a RSP package is shown in figure 5.5.

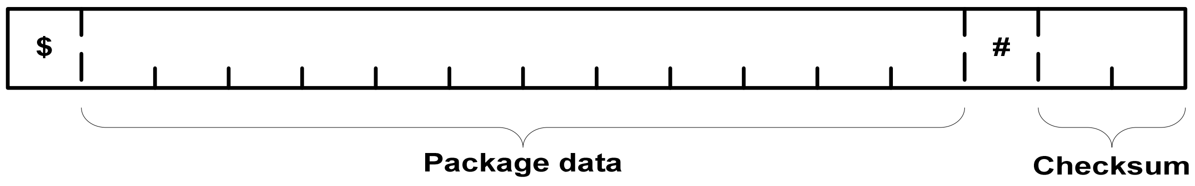


Figure 5.5 RSP package format (8)

Each RSP packet starts with a “$” following by the data area. The binary data is represented as two hexadecimal digits per byte of data. The “#” split the data and the checksum which is the unsigned sum of all the characters in the packet data modulo 256. For any package an acknowledgement is required when a single character "+" represents successful receipt while a "-" stands for a failure and retransmitting.

In our case, the Redboot is loaded into memory and executed at the first place. When it gets a "$" which means a GDB RSP package is coming, the Redboot creates a new thread of GDB stub and pass the control to it by context switch. The GDB stub takes the full control of the system until it gets a "quit" command. Then the control is passed back to Redboot.

Using the GDB remote debugging is quite similar as using common GDB debugger. A graphic interface frontier is also available for remote debugging, such as insight and eclipse, which can significantly improve debugging efficiency.

5.4.2 Codec chip configuration and debugging

The audio output of the BeagleBoard is driven by an independent chip which is called "TPS65950". The "TPS65950" is a multifunction device. But we only use its power supple and audio components in our experiment. The chip connects with the processor by several serial communication interfaces. Then the configuration and debugging of that chip is a little tricky. It uses an I2C interface as the control interface, which means all the read/write operations towards its registers are based on I2C protocol. The HAL of I2C component of the processor is implemented beforehand and won't be discussed in this section. The music sampling data is transmitted through the MCBSP interface.

The first thing we must notice is that the "TPS65950" doesn't have buffer on audio receiving channel. This means we should configure the transmission speed as same as the sampling frequency, which in our system is 44.1k sampling data per second. The unmatched speed would damage the data.

The MCBSP module of the processor supports several data format transmitting. In our system we choose I2S protocol format, which is shown in figure 5.5 (9). The data is transmitted with MSB first and the length can be 16, 24, or 32 bits with or without padding. One thing should be noticed is that if transmitting less than 32 bits length data, the data have to be put in the least significant bits within a 32-bits word.

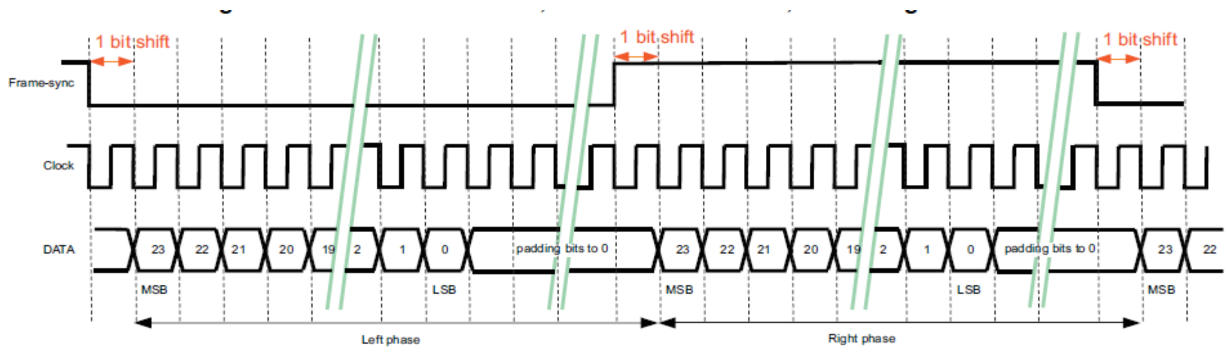


Figure 5.5 I2S protocol data format (9)

Another thing that should be paid attention to is the endianness, since we don't know the system configuration when the MP3 was encoded.

Debugging audio chip is a little tricky because it is hard to know the status inside the external chip. Within the audio component of "TPS65950" there are two signal loops which can be utilized for debugging. Figure 5.6 shows the architecture of audio module (9). The green, red, and blue lines represent three signal paths respectively, while the black line represents the shared path. The green path is the analog loop which sends the analog signal coming from audio-in jack directly to headset jack. The red path is the digital loop which sends the sampled data back to output channel. The blue path sends the received data to output channel.

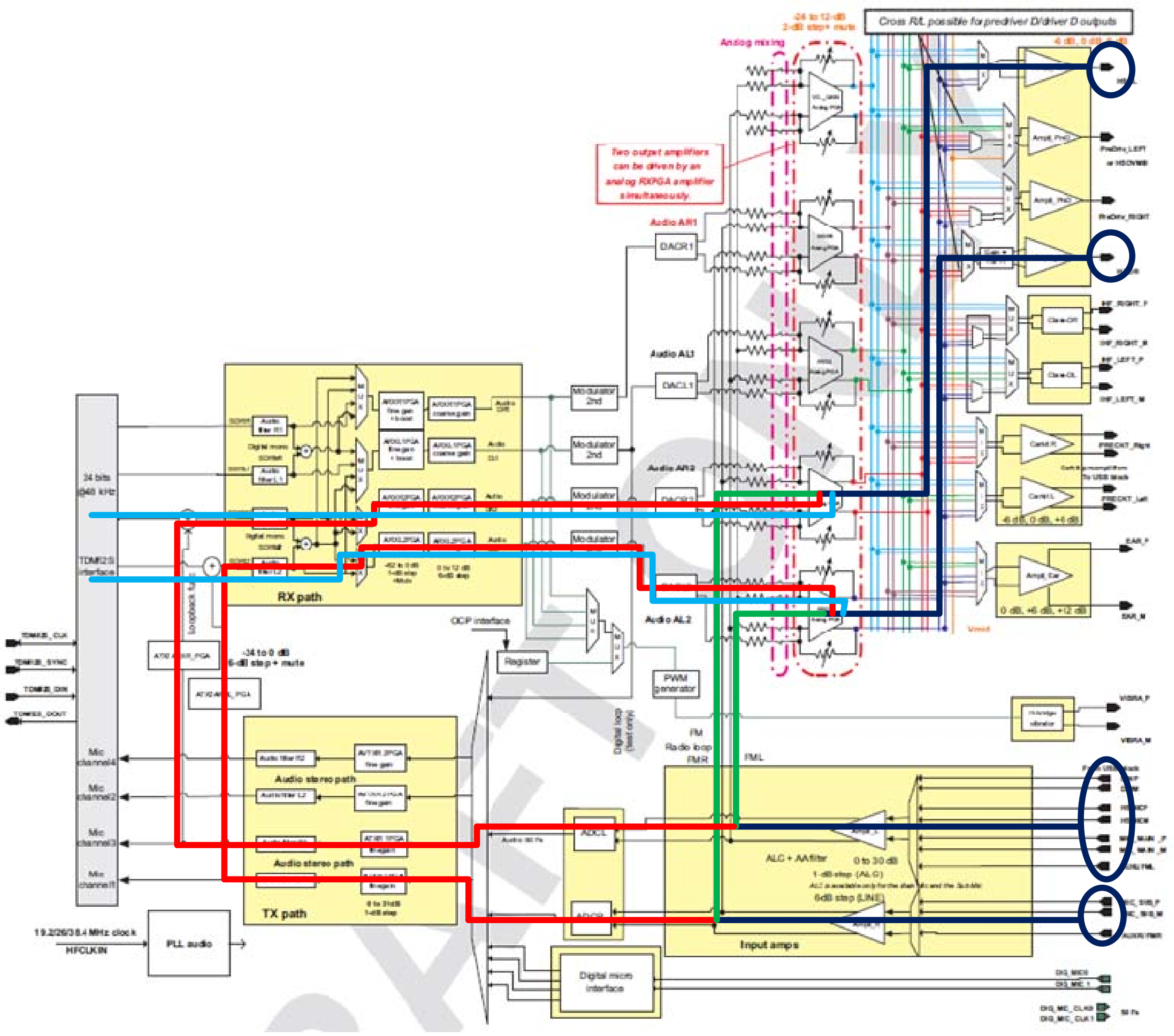


Figure 5.6 The architecture of audio device (9)

The major work of configuring the audio output is setting up the right data path and the proper gain. For narrowing the potential error in as small scope as possible, we implement the analog loop first, then the digital loop, and the "real" data path at last. A music signal is connected to audio-in channel. In each step, a part of the register set need to be configured correctly to get the music output.

After getting the digital loop work, we just simply turn off the loop back channel so that a working music channel is established.

5.5 Results

We use 9 mp3 files to test the "mp3 player". Each one has a different parameter set which is shown in table 5.2 and table 5.3.

First, we turned off the compiler optimization feature by adding "-O2" option to command line. The results are shown in table 5.2. For most situations, the decoding time is much longer than playing time, which means we can't execute decoding and playing concurrently.

Table 5.2 The performance without compiler optimization

file name	4432	4464	4496	2232	2264	2296	1132	1164	1196
channel	stereo	stereo	stereo	stereo	stereo	stereo	stereo	stereo	stereo
size (KB)	50.3	92.7	135	45	87.2	129.5	44.7	86.6	86.6
sampling frequency (kHz)	44.1	44.1	44.1	22.05	22.05	22.05	11.025	11.025	11.025
bit rate	32000	64000	96000	32000	64000	96000	32000	64000	96000
decoded file size (KB)	4428	4080	3972	1984	1924	1904	984	948	948
decoding time (s)	33.365	35.344	38.578	17.485	21.099	21.697	10.738	11.361	11.361
playing time (s)	12.863	11.843	11.530	11.540	11.173	11.069	11.462	11.043	11.043
ratio	259%	298%	335%	152%	189%	196%	94%	103%	103%

For the second experiment, we turn on the level 2 compiler optimization by adding “-O2” option to command line. The result is shown in table 5.3.

Table 5.3 The performance with compiler optimization level 2 (O2)

file name	4432	4464	4496	2232	2264	2296	1132	1164	1196
channel	stereo	stereo	stereo	stereo	stereo	stereo	stereo	stereo	stereo
size (KB)	50.3	92.7	135	45	87.2	129.5	44.7	86.6	86.6
sampling frequency (kHz)	44.1	44.1	44.1	22.05	22.05	22.05	11.025	11.025	11.025
bit rate	32000	64000	96000	32000	64000	96000	32000	64000	96000
decoded file size (KB)	4428	4080	3972	1984	1924	1904	984	948	948
decoding time (s)	22.106	22.284	22.436	11.514	11.624	11.839	5.852	6.05	6.05
playing time (s)	12.863	11.843	11.530	11.540	11.173	11.069	11.462	11.043	11.043
ratio	172%	188%	195%	99%	104%	107%	51%	55%	55%

Comparing table 5.2 and table 5.3, we can see that the decoding performance has been improved significantly by turning on compiler optimization feature. For each individual mp3 file, the decoding time decreases 30%-50% respectively. But in table 5.3, we still can find part of the files have longer decoding time than playing time, especially the files with 44.1k sampling rate. This is because our system is lack of hardware accelerator support, i.e. floating point accelerator or NEON. Further experiment will be carried out after hardware support is added into our system.

6 Conclusions

A well-defined HAL can effectively split the embedded software development into hardware dependent code design and hardware independent code design. Then the two developing phases can be done in parallel by which the pre-market time is reduced. The reliability of the system is also improved because of the possibility of formal verification and reuse of the application code and the operating system kernel. The experiment shows that the interface defined in our HAL architecture is capable to support a real time operating system with slight modification of OS kernel. We can conclude that our HAL architecture is capable to supporting a real time operating system. But considering the complexity of a rich feature operating system such as Linux, the modification of OS kernel for adapting to our HAL might be still “too much”, which may be solved by adding more features to the HAL architecture.

7 Future Work

So far, we have implemented our unified HAL in the eCos. However the eCos does not support virtual memory which limits its functionality. The next step of this project is building the HAL for MMU. Then we would port a full-featured OS such as Linux on the BeagleBoard with the UHAL.

8 Acknowledgement

The authors would like to acknowledge the Texas Instruments University Program for the donation of the BeagleBoard hardware for this project. The authors thank the TI University Program for the valuable support.

Bibliography

1. **Wolfgang Ecker, Wolfgang Mueller, Rainer Doemer.** *Hardware-dependent Software: Principles and Practice.* s.l. : Springer, 2009. ISBN 978-1-4020-9435-4.

2. *Introduction to Hardware Abstraction Layers for SoC.* **Sungjoo Yoo, Ahmed A. Jerraya.** Munich, Germany : In Proceedings of DATE 2003, 2003. pp. pp 336-337.

3. MPC8572E PowerQUICC™ III Integrated Host Processor Family Reference Manual. [Online] http://cache.freescale.com/files/32bit/doc/ref_manual/MPC8572ERM.pdf?fpsp=1.

4. AM/DM37x Multimedia Device Technical Reference Manual. [Online] http://www.ti.com/lit/ug/sprugn4q/sprugn4q.pdf.

5. SH7214 Group, SH7216 Group User's Manual: Hardware. [Online] http://am.renesas.com/products/mpumcu/superh/sh7216/sh7216/Documentation.jsp.

6. eCos Home. [Online] http://ecos.sourceware.org/.

7. BeagleBoard.org. [Online] http://beagleboard.org/.

8. *Embedding with GNU: the gdb Remote Serial Protocol.* **Gatliff, Bill.** 12, s.l. : Embedded Systems Programming, 1999, Embedded Systems Programming, Vol. 12, pp. 108-113.

9. TPS65950 OMAP Power Management and System Companion Device Technical Reference Manual. *www.TI.com.* [Online] http://www.ti.com/lit/ug/swcu050g/swcu050g.pdf.