Official Release

EILM ROM Patch Application Note

Document Number AN028-10

Date Issued 2015-09-16

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, please contact Andes Technology Corporation
by email support@andestech.com
or online website https://es.andestech.com/eservice/
for support giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Chapter-Section	Revised Content
1.0	2015/09/16	All	Initial release

Table of Contents

List of Figures

Typographical Convention Index

Document Element	Font	Font Style	Size	Color
Normal text	Georgia	Normal	12	Black
`Command line, source code or file paths`	`Lucida Console`	`Normal`	`11`	`Indigo`
`VARIABLES OR PARAMETERS IN COMMAND LINE, SOURCE CODE OR FILE PATHS`	**`LUCIDA CONSOLE`**	**`BOLD + ALL-CAPS`**	**`11`**	**`INDIGO`**
Note or warning	Georgia	Normal	12	Red
Hyperlink	Georgia	Underlined	12	Blue

1. Introduction

A trade-off in the modern ASIC/SoC implementation is where to store the firmware. One may choose to put the firmware in on-chip or off-chip memory. Storing the firmware on-chip reduces the BOM cost while the capacity off-chip can be several orders of magnitude larger. For the on-chip option, one may also choose either ROM or embedded flashes. The embedded flash option allows programmability but it has higher manufacturing cost. The ROM option is a lot cheaper as well as denser in area, but the non-programmability is its major drawback. Once a firmware is commenced into ROM, it can no longer be changed. This is a problem if some bugs are discovered later or new features are required. An approach to solve the non-programmability problem is to implement a ROM patch unit to redirect fetches to buggy codes to a revised version (a patched version) in the programmable part of the system. This document provides a reference hardware design of an EILM ROM patch unit for this purpose. A programming sequence is also provided to illustrate how to apply patches with the ROM patch unit.

The reference hardware design connects to the AndesCore™ processor through the Andes EILM interface. You can revise the design to accommodate your ROM interface and bit-width.

2. EILM ROM Patch Unit

2.1. System Block Diagram

Figure 1 illustrates a system with an EILM ROM patch unit. The processor fetches ROM program through the ROM patch unit. The ROM patch registers record the function entry points to be redirected. When the processor fetches the first word of a buggy function, the ROM patch unit will return a jump instruction that jumps to a patched version in the RAM. Then, the processor executes the patched version instead of the function in the ROM.

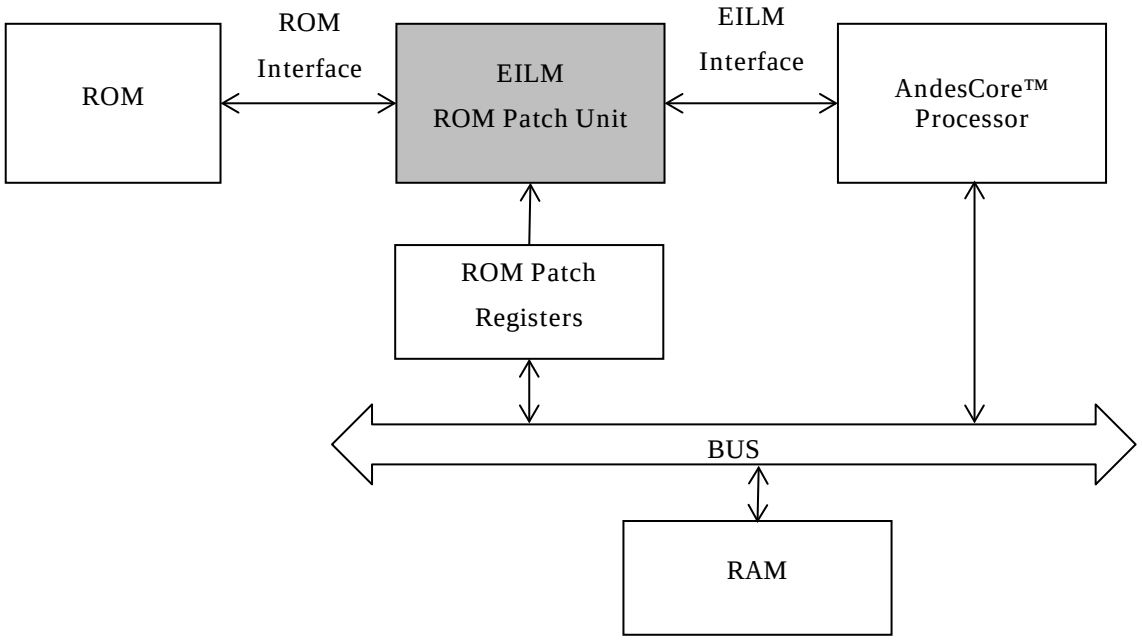

Figure 1. EILM ROM Patch Unit System Block Diagram

2.2. ROM Patch Unit Operations

The patch registers control the ROM patch unit operation. The number of supported patches depends on the number of patch registers. A patch register defines a patch point, which consists of three fields and the fields are connected to input ports of the ROM patch unit.


```
input            reg_patch0_en;
input       [ADDR_MSB:2] reg_patch0_addr;
input            [31:0]  reg_patch0_data;
```

The enable field determines whether to enable a patch point (0=disabled, 1=enabled). The address field points to the entry point of a function to patch, and the data field stores the instruction data for replacing the function entry point. The most significant bit of the address field is a parameter, which is dependent on the ROM size.

When ROM patch unit receives a request from the processor, it immediately forwards the request to the ROM and compares the requested address with all address fields of all patch points. When the address hits a patch point, the corresponding data field is returned to the processor. When no patch point matches the requested address, the ROM data is returned to the processors. The following RTL codes illustrate the comparison logic. The reference code supports four patches, and you can revise the codes to support more patches.

```
always @(posedge core_clk or negedge core_reset_n) begin
 if (!core_reset_n) begin
    patch0_hit <= 1'b0;
    patch1_hit <= 1'b0;
    patch2_hit <= 1'b0;
    patch3_hit <= 1'b0;
 end
 else if (eilm_addr_grant) begin
    patch0_hit <= patch0_hit_nx;
    patch1_hit <= patch1_hit_nx;
    patch2_hit <= patch2_hit_nx;
    patch3_hit <= patch3_hit_nx;
 end
end
```

```
assign patch0_hit_nx = reg_patch0_en & (reg_patch0_addr == eilm_addr);
assign patch1_hit_nx = reg_patch1_en & (reg_patch1_addr == eilm_addr);
assign patch2_hit_nx = reg_patch2_en & (reg_patch2_addr == eilm_addr);
assign patch3_hit_nx = reg_patch3_en & (reg_patch3_addr == eilm_addr);

assign patch_miss = ~(patch0_hit|patch1_hit|patch2_hit|patch3_hit);

assign eilm_rdata = ({32{patch_miss}} & rom_rdata)
                  | ({32{patch0_hit}} & reg_patch0_data)
                  | ({32{patch1_hit}} & reg_patch1_data)
                  | ({32{patch2_hit}} & reg_patch2_data)
                  | ({32{patch3_hit}} & reg_patch3_data);
```


The following RTL code illustrates the glue logic to handle the EILM interface protocol. For more information, please see the "External Local Memory" chapter in "*AndeStar™ Interface Architecture Manual (UM024)*".

```
always @(posedge core_clk) begin
 eilm_wait_d1 <= eilm_wait;
end

always @(posedge core_clk or negedge core_reset_n) begin
 if (!core_reset_n)
    wait_cnt <= 2'b0;
 else if (wait_cnt_en)
    wait_cnt <= wait_cnt_nx;
end

assign eilm_stall = (wait_cnt != 2'b0) | eilm_wait_d1;
assign eilm_addr_grant = eilm_req & ~eilm_stall;
assign wait_cnt_nx = (eilm_addr_grant) ? eilm_wait_cnt : wait_cnt - 2'b1;
assign wait_cnt_en = eilm_addr_grant | (wait_cnt != 2'b0);
```

3. Programming Sequence

This chapter provides reference C codes to illustrate how to apply a patch to a program. A SaG file is also provided for generating a link script.

3.1. Limitations

To cooperate with the ROM patch unit, the program has two limitations. First, the distance of a function and a patch should be within +-16MB. Essentially, the ROM patch unit replaces the first word of a function with a 32-bit jump instruction to patch. The jump offset is a 24-bit immediate, which means the distance of a function and a patch must be within +-16MB.

Second, each patch point of the patch registers defines a 4-byte aligned location. When a function is not aligned to 4-byte boundary, it requires two patch registers to replace its entry point with a 32-bit jump instruction. To efficiently utilize patch registers, the gcc option `-malign-functions` can guide the compiler to align functions to 4-byte boundary.

3.2. Memory Map

Figure 2 shows the memory map of the system. A 64 KB ROM starts at address 0x00000, and a RAM starts at address 0x10000. The patch registers are mapped to system memory, which starts at address 0x90082000. The register offset for each patch registers is also illustrated in the figure.

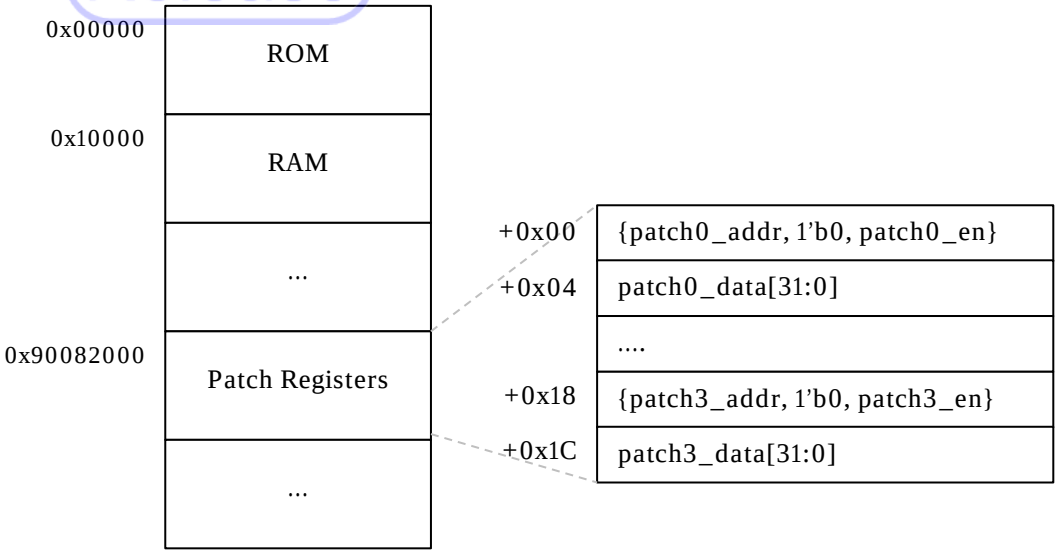

Figure 2. Memory Map

3.3. Reference C Codes without Patches

The following C code illustrates a program before it is patched. The program has two source files, main.c and patch.c. A dummy patch_init() stub is inserted for patch provisioning.


```
//main.c
#include <stdio.h>

extern int patch_init();

__attribute__ ((noinline))
void rom_func() {
	printf("rom_func");
}

int main(int argc, char* argv[]) {
	if (patch_init() != 0) {
		return 1;
	}
	rom_func();
	return 0;
}

// patch.c
__attribute__ ((section (".patch_init")))
__attribute__ ((noinline))
int patch_init() {
	 return 0;
}
```

Both of `rom_func` and `main` functions are allocated in the ROM, and `rom_func` is the target function that needs a patch. The `__attribute__ ((noinline))` is added to the function to disable compiler optimization that inlines the function.

The `patch_init` stub function is provisioned for applying patches. The function attributes help linker allocate `patch_init` to RAM with a fixed address. The function is initially a dummy function. If patching is needed, the function should download the revised code into the RAM space and initial the patch registers to redirect the function entry being patched to the revised versions.

The following code illustrates a SaG file that allocates the `patch_init` to the RAM space. You can execute the command "`nds_ldsag patch.sag`" to generate a link script (nds32.ld). For more information, please see "*Andes SaG Application Examples (WP-020)*" and the "Linker Script Generation" chapter in "*Andes Programming Guide (PG-009)*".


```
// patch.sag
USER_SECTIONS .patch_init
ROM 0x00000
{
       EILM_ROM 0x00000 {
              * EXCLUDE_FILE (patch.o) (+RO)
       }
}

RAM 0x10000 {
       PATCH 0x10000 {
              * (.patch_init)
              patch.o (+RO)
       }
       DATA 0x10100 {
              * (+RW, +ZI)
       }
}
```

3.4. Reference C Codes with Patches

The following C code illustrates how to replace `rom_func` with a revised version `rom_func_v2`.


```
//patch.c
#include <stdio.h>
#define PATCH_REG_BASE  ((unsigned int volatile*)0x90082000)

extern void rom_func();

void rom_func_v2() {
       printf("rom_func_v2\n");
}

__attribute__ ((section (".patch_init")))
__attribute__ ((noinline))
int patch_init() {
       unsigned int jump_instr = 0x48000000 | ((rom_func_v2 - rom_func)/2));
       PATCH_REG_BASE[1] = __builtin_bswap32(jump_instr);
       PATCH_REG_BASE[0] = (unsigned int)&rom_func | 0x1;

       return 0;
}
```

The `patch_init` stub function initializes the patch registers. The `PATCH_REG_BASE[0]` is the address field and enable field of the patchpoint0 register. The address field should points to the entry address of `rom_func`, and the enable field should be set. The `PATCH_REG_BASE[1]` is data field of the patchpoint0 register, which should be a jump instruction to the revised `rom_func_v2`. The immediate field of the jump instruction is a relative amount between the instruction PC and the target of the jump, so the offset value is (`rom_func_v2 - rom_func`). The example codes assumes the system data endian is little endian, and `__builtin_bswap32` converts the machine code for the instruction to little endian.

Appendix

Appendix I. ROM Patch Unit Reference Design


```
module rom_patch_unit(
        core_clk,
        core_reset_n,
        eilm_addr,
        eilm_req,
        eilm_wdata,
        eilm_web,
        eilm_ifetch_n,
        eilm_wait_cnt,
        eilm_wait,
        eilm_rdata,
        eilm_status,
        eilm_size,
        rom_addr,
        rom_req,
        rom_rdata,
        rom_wait_cnt,
        rom_wait,
        rom_status,
        reg_patch0_en,
        reg_patch0_addr,
        reg_patch0_data,
        reg_patch1_en,
        reg_patch1_addr,
        reg_patch1_data,
        reg_patch2_en,
        reg_patch2_addr,
        reg_patch2_data,
```

```
            reg_patch3_en,
            reg_patch3_addr,
            reg_patch3_data
);
parameter               ADDR_MSB = 15;
parameter                EILM_SIZE = 4;
input                    core_clk;
input                    core_reset_n;
input       [ADDR_MSB:2] eilm_addr;
input                    eilm_req;
input             [31:0] eilm_wdata;
input              [3:0] eilm_web;
input                    eilm_ifetch_n;
output             [1:0] eilm_wait_cnt;
output                   eilm_wait;
output            [31:0] eilm_rdata;
output                   eilm_status;
output             [3:0] eilm_size;

input                    reg_patch0_en;
input       [ADDR_MSB:2] reg_patch0_addr;
input             [31:0] reg_patch0_data;
input                    reg_patch1_en;
input       [ADDR_MSB:2] reg_patch1_addr;
input             [31:0] reg_patch1_data;
input                    reg_patch2_en;
input       [ADDR_MSB:2] reg_patch2_addr;
input             [31:0] reg_patch2_data;
input                    reg_patch3_en;
input       [ADDR_MSB:2] reg_patch3_addr;
input             [31:0] reg_patch3_data;
```

```
output          [ADDR_MSB:2] rom_addr;
output                       rom_req;
input                 [31:0] rom_rdata;
input                  [1:0] rom_wait_cnt;
input                        rom_wait;
input                        rom_status;

reg                    [1:0] wait_cnt;
wire                   [1:0] wait_cnt_nx;
wire                         wait_cnt_en;
wire                         eilm_addr_grant;
wire                         eilm_stall;
reg                          eilm_wait_d1;

reg                          patch0_hit;
reg                          patch1_hit;
reg                          patch2_hit;
reg                          patch3_hit;
wire                         patch0_hit_nx;
wire                         patch1_hit_nx;
wire                         patch2_hit_nx;
wire                         patch3_hit_nx;

wire                         patch_miss;

// Glue Logic
assign rom_addr      = eilm_addr;
assign rom_req       = eilm_req;
assign eilm_wait_cnt = rom_wait_cnt;
assign eilm_wait     = rom_wait;
assign eilm_status   = rom_status;
assign eilm_size     = EILM_SIZE;
```

```
always @(posedge core_clk) begin
    eilm_wait_d1 <= eilm_wait;
end

always @(posedge core_clk or negedge core_reset_n) begin
    if (!core_reset_n)
        wait_cnt <= 2'b0;
    else if (wait_cnt_en)
        wait_cnt <= wait_cnt_nx;
end

always @(posedge core_clk or negedge core_reset_n) begin
    if (!core_reset_n) begin
        patch0_hit <= 1'b0;
        patch1_hit <= 1'b0;
        patch2_hit <= 1'b0;
        patch3_hit <= 1'b0;
    end
    else if (eilm_addr_grant) begin
        patch0_hit <= patch0_hit_nx;
        patch1_hit <= patch1_hit_nx;
        patch2_hit <= patch2_hit_nx;
        patch3_hit <= patch3_hit_nx;
    end
end

assign eilm_stall = (wait_cnt != 2'b0) | eilm_wait_d1;
assign eilm_addr_grant = eilm_req & ~eilm_stall;
assign wait_cnt_nx = (eilm_addr_grant) ? eilm_wait_cnt : wait_cnt - 2'b1;
assign wait_cnt_en = eilm_addr_grant | (wait_cnt != 2'b0);
```

```
// Patch Logic
assign patch0_hit_nx = reg_patch0_en & (reg_patch0_addr == eilm_addr);
assign patch1_hit_nx = reg_patch1_en & (reg_patch1_addr == eilm_addr);
assign patch2_hit_nx = reg_patch2_en & (reg_patch2_addr == eilm_addr);
assign patch3_hit_nx = reg_patch3_en & (reg_patch3_addr == eilm_addr);

assign patch_miss = ~(patch0_hit|patch1_hit|patch2_hit|patch3_hit);

assign eilm_rdata = ({32{patch_miss}} & rom_rdata)
                  | ({32{patch0_hit}} & reg_patch0_data)
                  | ({32{patch1_hit}} & reg_patch1_data)
                  | ({32{patch2_hit}} & reg_patch2_data)
                  | ({32{patch3_hit}} & reg_patch3_data)
          ;

endmodule
```