SkyEye Internal

Draft 0.1

Blackfin.kang@gmail.com

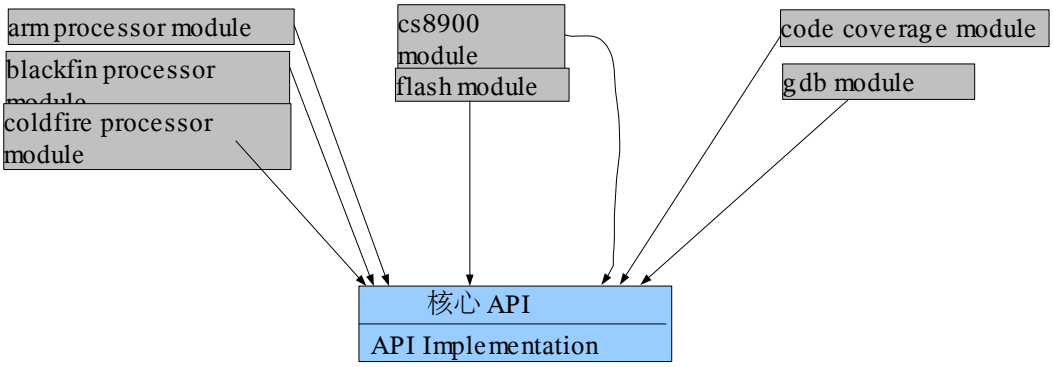
第一章 SkyEye 整体架构和代码结构

1.1 架构介绍

SkyEye 是一个仿真单板的开发平台，提供了丰富的 API 函数来基于已有的仿真模块进行二次开发。我们可以把整个 SkyEye 仿真平台分为两大部分，核心库和各种其他外围动态模块。其中外围动态模块大体分类如下：
处理器核的仿真模块：主要是仿真外设的指令集，中断等。目前可以仿真六个体系结构：arm, mips ,powerpc, blackfin, coldfire, sparc。
外设仿真模块：如网卡，LCD, Flash 等外设控制器的仿真模块
统计分析模块：有代码覆盖率分析模块，函数流跟踪模块等。

最新的 SkyEye 架构主要着眼于模块化和可扩展性，描述如下：
1、模块化
* 每一个模块可以以 so 或者 DLL 的形式存在，并且可以独立开发，独立编译。在 SkyEye 启动的时候会被 SkyEye 的核心库进行动态加载。
* 模块之间不存在任何依赖关系，相互独立，不存在相互调用关系。所有的模块都是调用 SkyEye 核心库提供的标准接口进行操作。
2、可扩展性
* 设计一组核心的 API 及其实现。把核心的 API 和实现放在 SkyEye 的核心库中，并提供给其他外围模块。
* 核心模块负责查找系统中可得到的其他外围模块，并进行动态加载。

其整体结构图如下：



1.2 目录结构

核心库位于 SkyEye 源代码的 common 目录，主要提供各种数据结构的注册管理，模块管理，命令行界面等等。其中功能在源码目录的分布如下：

breakpoint 目录：断点管理模块，实现了断点的插入，删除等。

Bus 目录：IO 读写接口的实现，包括 ram 的实现

callback: callback 函数的管理，实现了 callback 的插入，删除等。

cli: 调用了 readline 的库 实现了命令行接口。
conf_parser: 配置文件的解析函数
core: 对处理器核仿真模块的管理，包含模块的注册，查询等。
ctrl: 仿真平台的控制模块，提供了初始化，启动，停止等函数的实现。
device: 外设仿真模块的管理，包含外设仿真模块的注册，查询等。
loader: 实现把各种文件加载到指定地址空间的函数。
log: 实现了日志功能，来记录仿真平台运行过程中的调试信息，错误信息等。
mach: 实现了仿真单板模块的管理。
module : 实现了对 SkyEye 动态模块的管理。
preference: 实现了对预置选项的管理。

1.3 核心库的介绍

第二章　总体流程和关键数据结构

2.1 skyeye 的配置文件模块

2.1.1 介绍

当前 skyeye 使用了一个文本配置文件来描述仿真的目标平台和一些其他的特性。关于当前 SkyEye 支持的各种配置选项，可以参考 SkyEye 的用户手册。我们也可以对配置文件进行扩展来添加自己的选项。

2.1.2 数据结构

我们用一个数据结构来代表配置文件的中的一个选项：

```
 6 typedef struct skyeye_option_s
 7 {
 8        char *option_name;
 9        int (*do_option) (struct skyeye_option_s * this_opion,
10                          int num_params, const char *params[]);
11        char* helper;
17        struct skyeye_option_t *next;
18 } skyeye_option_t;
```

其中， option_name 是一个用来标志这个选项的字符串，do_option 成员用来解析这个选项的函数，helper 成员也是字符串变量用来对选项进行描述。最后 next 成员是一个指向下一个 option 元素的指针。

我们用一个链表来把所有的 option 管理起来。链表的头为 skyeye_option_list 变量，定义在文件 common/conf_parser/skyeye_options.c 下，代码如下：

```
71 static skyeye_option_t* skyeye_option_list;
```

在 common/conf_parser/skyeye_options.c 文件中，我们还实现了对配置选项的链表进行其他的一些操作。register_option 函数用来在链表里面添加一个配置选项，实现代码如下。

```
93 exception_t register_option(char* option_name, do_option_t do_option_func, char*
helper){
94        if(option_name == NULL || !do_option_func)
95                return Invarg_exp;
96        skyeye_option_t* node = malloc(sizeof(skyeye_option_t));
97        if(node == NULL)
98                return Malloc_exp;
```

```
 99        node->option_name = skyeye_strdup(option_name);
100        if(node->option_name == NULL){
101                skyeye_free(node);
102                return Malloc_exp;
103        }
104        node->do_option = do_option_func;
105        /* maybe we should use skyeye_mm to replace all the strdup */
106        node->helper = skyeye_strdup(helper);
107        if(node->option_name == NULL){
108                skyeye_free(node->option_name);
109                skyeye_free(node);
110                return Malloc_exp;
111        }
112        node->next = skyeye_option_list;
113        skyeye_option_list = node;
114        //skyeye_log(Info_log, __FUNCTION__, "register option %s successfully.",
option_name);
115        return No_exp;
116 }
```

2.1.3 运行流程

在每一个模块初始化函数中，可以上面的 register_option 函数来注册自己的配置流程选项。在所有的配置选项注册完毕之后，我们通过输入"start”命令或者函数调用的方式去运行 SIM_start 函数，这时 skyeye_read_config 函数被调用来解析指定的配置文件，如下：

```
106        sky_pref_t *pref;
107        /* get the current preference for simulator */
108        pref = get_skyeye_pref();
109        skyeye_config_t* config = get_current_config();
110        if(pref->conf_filename)
111                skyeye_read_config(pref->conf_filename);
```

skyeye_read_config 会调用我们预先注册好的钩子函数对相应的选项进行解析每一个配置选项。

```
34 static skyeye_config_t skyeye_config;
35
```

get_current_config 用来获得当前配置数据结构体的指针。我们在编写其他模块的时候，有时候需要获得配置数据结构体的指针。

```
36 skyeye_config_t* get_current_config(){
37         return &skyeye_config;
38 }
```

2.2 SkyEye 的命令行接口模块

2.2.1 介绍

skyeye 的命令行接口是调用 readline 的库来实现的命令行解析和管理工作。Readline 是一个功能强大的命令行接口库。

2.2.2 数据结构

SkyEye 命令行接口中的每一条命令我们都用了一个 COMMAND 的数据结构进行描述，其代码如下：

```
41 /* A structure which contains information on the commands this program
42    can understand. */
43 struct command_s{
44    char *name;                 /* User printable name of the function. */
45    rl_icpfunc_t *func;          /* Function to call to do the job. */
46    char *doc;                 /* Documentation for this function.  */
47    struct command_s *next;
48 };
49 typedef struct command_s  COMMAND;
```

name 变量是我们敲入命令的字符串，func 是执行命令的函数，doc 是这条命令的帮助信息。next 是指向下一条命令数据结构的指针。我们把所有命令的数据结构用一个链表管理起来。

```
73 static COMMAND *command_list;
```

2.2.3 运行流程

2.3 模块动态加载部分

SkyEye 启动的时候会调用 SkyEye 模块加载函数，从系统安装 SkyEye 默认的目录去查找符合 SkyEye 规范的动态链接库。

实现代码位于 common/module/skyeye_module.c

```
/* on *nix platform, the suffix of shared library is so. */
const char* Default_libsuffix = ".so";
/* we will not load the prefix with the following string */
const char* Reserved_libprefix = "libcommon";

const char Dir_splitter = '/';

typedef struct skyeye_modules_s{
	skyeye_module_t* list;
	int total;
}skyeye_modules_t;

static skyeye_modules_t* skyeye_modules;

static void set_module_list(skyeye_module_t *node){
	skyeye_modules->list = node;
}
exception_t init_module_list(){
	skyeye_modules = skyeye_mm(sizeof(skyeye_modules_t));
	if(skyeye_modules == NULL)
		return Malloc_exp;
	return No_exp;
}
skyeye_module_t* get_module_list(){
	return skyeye_modules->list;
}

static exception_t register_skyeye_module(char* module_name, char* filename, void*
handler){
	exception_t ret;
	skyeye_module_t* node;
```

```
    skyeye_module_t* list;
    list = get_module_list();
    if(module_name == NULL|| filename == NULL)
        return Invarg_exp;

    node = malloc(sizeof(skyeye_module_t));
    if(node == NULL){
        skyeye_log(Error_log, __FUNCTION__, get_exp_str(Malloc_exp));
        return Malloc_exp;
    }

    node->module_name = strdup(module_name);
    if(node->module_name == NULL){
        free(node);
        return Malloc_exp;
    }

    node->filename = strdup(filename);
    if(node->filename == NULL){
        free(node->module_name);
        free(node);
        return Malloc_exp;
    }

    node->handler = handler;

    node->next = list;;
    set_module_list(node);
    return No_exp;
}

exception_t SKY_load_module(const char* module_filename){
    exception_t ret;
```

```
	char **module_name;
	void * handler;
	char* err_str;
	//skyeye_log(Debug_log, __FUNCTION__, "module_filename = %s\n",
module_filename);
	handler = dlopen(module_filename, RTLD_LAZY);
	if (handler == NULL)
	{
		err_str = dlerror();
		skyeye_log(Warnning_log, __FUNCTION__, "%s\n", err_str);
		return Dll_open_exp;
	}

	module_name = dlsym(handler, "skyeye_module");
	if((err_str = dlerror()) != NULL){
		skyeye_log(Warnning_log, __FUNCTION__, "dll error %s\n", err_str);
		skyeye_log(Warnning_log, __FUNCTION__, "Invalid module in file %s\n",
module_filename);
		dlclose(handler);
		return Invmod_exp;
	}
	//skyeye_log(Debug_log, __FUNCTION__, "Load module %s\n", *module_name);

	ret = register_skyeye_module(*module_name, module_filename, handler);
	if(ret != No_exp){
		dlclose(handler);
		return ret;
	}
	return No_exp;
}

void SKY_load_all_modules(char* lib_dir, char* suffix){
	/* we assume the length of dirname + filename does not over 1024 */
```

```
char* full_filename[1024];
char* lib_suffix;
/* Find all the module under lib_dir */
DIR *module_dir = opendir(lib_dir);
exception_t exp;
/*FIXME we should throw some exception. */
if(module_dir == NULL)
	return;
if(suffix == NULL)
	lib_suffix = Default_libsuffix;
else
	lib_suffix = suffix;
struct dirent* dir_ent;
while((dir_ent = readdir(module_dir)) != NULL){
	char* mod_name = dir_ent->d_name;
	/* exclude the library not end with lib_suffix */
	char* suffix = strrchr(mod_name, '.');
	if(suffix == NULL)
		continue;
	else{
		//skyeye_log(Debug_log, __FUNCTION__, "file suffix=%s\n", suffix);
		if(strcmp(suffix, lib_suffix))
			continue;
	}
		/* exclude the reserved library */
	if(!strncmp(mod_name, Reserved_libprefix, strlen(Reserved_libprefix)))
		continue;

	/* contruct the full filename for module */
	int lib_dir_len = strlen(lib_dir);
	memset(&full_filename, '\0', 1024);
	strncpy(&full_filename[0], lib_dir, lib_dir_len);
	full_filename[lib_dir_len] = Dir_splitter;
```

```
            full_filename[lib_dir_len + 1] = '\0';
            //skyeye_log(Debug_log, __FUNCTION__, "1 full_filename=%s\n",
full_filename);
            strncat(full_filename, mod_name, strlen(mod_name) + 1);
            //skyeye_log(Debug_log, __FUNCTION__, "full_filename=%s\n",
full_filename);
            /* Try to load a module */
            exp = SKY_load_module(full_filename);
            if(exp != No_exp)
                skyeye_log(Info_log, __FUNCTION__, "Can not load module from file
%s.\n", dir_ent->d_name);
            //}
        }
        closedir(module_dir);
}
skyeye_module_t * get_module_by_name(const char* module_name){
        skyeye_module_t* list = get_module_list();
        while(list != NULL){
            if(!strncmp(list->module_name, module_name, strlen(module_name)))
                return list;
            list = list->next;
        }
        return NULL;
}
```

提供的 API 定义接口位于 common/include/skyeye_module.h,部分代码如下：

```
/*
 * the contructor for module. All the modules should implement it.
 */
void module_init ()  __attribute__((constructor));

/*
```

```
 * the decontructor for module. All the modules should implement it.
 */
void module_fini () __attribute__((destructor));


typedef struct skyeye_module_s{
	/*
	 * the name for module, should defined in module as an varaible.
	 */
	char* module_name;
	/*
	 * the library name that contains module
	 */
	char* filename;
	/*
	 * the handler for module operation.
	 */
	void* handler;
	/*
	 * next node of module linklist.
	 */
	struct skyeye_module_s *next;
}skyeye_module_t;

/*
 * load all the modules in the specific directory with specific suffix.
 */
void SKY_load_all_module(const char* lib_dir, char* lib_suffix);

/*
 * load one module by its file name.
 */
exception_t SKY_load_module(const char* module_filename);
```

模块加载的命令行演示

ksh@server:/opt/skyeye> bin/skyeye

SkyEye is an Open Source project under GPL. All rights of different parts or modules are reserved by their author. Any modification or redistributions of SkyEye should note remove or modify the annoucement of SkyEye copyright.

Get more information about it, please visit the homepage http://www.skyeye.org.

Type "help" to get command list.

(skyeye)list-modules

Module Name	File Name
sparc	/opt/skyeye/lib/skyeye/libsparc.so
mips	/opt/skyeye/lib/skyeye/libmips.so
flash	/opt/skyeye/lib/skyeye/libflash.so
code_cov	/opt/skyeye/lib/skyeye/libcodecov.so
uart	/opt/skyeye/lib/skyeye/libuart.so
gdbserver	/opt/skyeye/lib/skyeye/libgdbserver.so
coldfire	/opt/skyeye/lib/skyeye/libcoldfire.so
arm	/opt/skyeye/lib/skyeye/libarm.so
touchscreen	/opt/skyeye/lib/skyeye/libts.so
net	/opt/skyeye/lib/skyeye/libnet.so
nandflash	/opt/skyeye/lib/skyeye/libnandflash.so
ppc	/opt/skyeye/lib/skyeye/libppc.so
bfin	/opt/skyeye/lib/skyeye/libbfin.so

(skyeye)

SkyEye 模块规范及编写示例：

每个模块需要实现两个函数 module_init 和 module_fini，其中 module_init 函数会在动态模块加载的时候被自动执行，而 module_fini 函数会在动态模块卸载的时候被自动调用。

为了区分 SkyEye 的动态模块和其他动态链接库，每个 SkyEye 的动态模块需要定义一个全局的字符串变量, skyeye_module, skyeye 会通过判断当前的动态链接库中是否存在变量 skyeye_module 来决定这个动态链接库是否是 SkyEye 的合法模块。

第三章、仿真平台的初始化代码分析

当前 SkyEye 提供了核心库和插件的实现方式，任何功能都可以以插件的形式实现。SkyEye 的核心库只是提供了一组函数，并不包含 SkyEye 的主函数。

3.1 main 函数

SkyEye 的主函数 main 位于 utils/main/skyeye.c。

```
477 /**
478  *  The main function of skyeye
479  */
480
481 int
482 main (int argc, char **argv)
483 {
484         int ret;
485
486         sky_pref_t* pref = get_skyeye_pref();
487         assert(pref != NULL);
488         /* initialization of options from command line */
489         ret = init_option(argc, argv, pref);
490         /* set the current preference for skyeye */
491         //update_skyeye_pref(pref);
492
493         SIM_init();
544         return ret;
545 }
```

在代码的 486 到 489 行，我们设置了 SkyEye 需要的运行环境变量，其中可以是 elf 镜像文件的名称，要运行机器的大小端等信息。在这里，我们的这些运行环境变量主要是通过解析 skyeye 的命令行参数和镜像文件获得。

在 493 行中，通过调用 SIM_init 函数，SkyEye 的命令行接口会在 SIM_init 中启动，这时我们就可以输入各种命令来控制 SkyEye 的运行。而在执行完 SIM_init，main 函数的任务就基本完成了，剩下的执行都交给 SkyEye 的 CLI 的界面了。

3.2 SIM_init 函数

SIM_init 的功能主要是动态模块的加载以及各种初始化的工作，相关实现代码位于 common/ctrl/sim_ctrl.c 。

```
27 void SIM_init(){
28      sky_pref_t* pref;
29      char* welcome_str = get_front_message();
30      /*
31       * get the corrent_config_file and do some initialization
32       */
33      skyeye_config_t* config = get_current_config();
34      skyeye_option_init(config);
35      /*
36       * initialization of callback data structure, it needs to
37       * be initialized at very beginning.
38       */
39      init_callback();
40
41      /*
42       * initilize the data structure for command
43       * register some default built-in command
44       */
45      init_command_list();
46
47      init_stepi();
48
49      /*
50       * initialization of module manangement
51       */
52      init_module_list();
53
54
55      /*
56       * initialization of architecture and cores
```

```
57          */
58         init_arch();
59
60         /*
61          * initialization of bus and memory module
62          */
63         init_bus();
64
65
66         /*
67          * initialization of machine module
68          */
69         init_mach();
70
71
72         /*
73          * initialization of breakpoint, that depends on callback module.
74          */
75         init_bp();
75         init_bp();
76
77         /*
78          * get the current preference for simulator
79          */
80         pref = get_skyeye_pref();
81
82         /*
83          * loading all the modules in search directory
84          */
85         if(!pref->module_search_dir)
86                 pref->module_search_dir = skyeye_strdup(default_lib_dir);
87         SKY_load_all_modules(pref->module_search_dir, NULL);
88         //skyeye_config_t *config;
```

```
89        //config = malloc(sizeof(skyeye_config_t));
90        if(try_init() == No_exp){
91                if(pref->autoboot == True){
92                        SIM_run();
93                }
94        }
95        /*
96         * if we run simulator in GUI or external IDE, we do not need to
97         * launch our CLI.
98         */
99        if(pref->interactive_mode == True){
100               SIM_cli();
101        }
102 }
```

在上面代码的 100 行处，SkyEye 调用了 SIM_cli 函数，从而进入了命令行界面，我们可以输入各种命令对仿真平台进行操作。

如代码 99 到 100 行所示，我们也可以通过在 SkyEye 的 main 函数中通过设置 pref->interactive_mode 的参数来对 SkyEye 进行设置，选择是否要启动 SkyEye 的命令行界面。

3.3 SIM_start

接下来，如果我们要启动我们要运行的目标板，我们可以输入 start 命令来初始化仿真目标板，并加载要运行的镜像文件。

下面的代码为 start 命令对应的执行函数。

```
54 /*
55  * start running of  SkyEye
56  */
57
58 com_start (arg)
59      char *arg;
60 {
61        int flag = 0;
62        SIM_start();
63        return flag;
```

```
64 }
```

如 62 行，SIM_start 会被调用。SIM_start 的实现如下：

```
105 void SIM_start(void){
106         sky_pref_t *pref;
107         /* get the current preference for simulator */
108         pref = get_skyeye_pref();
109         skyeye_config_t* config = get_current_config();
110         if(pref->conf_filename)
111                 skyeye_read_config(pref->conf_filename);
112
113         if(config->arch == NULL){
114                 skyeye_log(Error_log, __FUNCTION__, "Should provide valid arch
option in your co    nfig file.\n");
115                 return;
116         }
117         generic_arch_t *arch_instance = get_arch_instance(config->arch->arch_name);
118
119         if(config->mach == NULL){
120                 skyeye_log(Error_log, __FUNCTION__, "Should provide valid mach
option in your co    nfig file.\n");
121                 return;
122         }
123
124         arch_instance->init();
125
126         /* reset all the memory */
127         mem_reset();
128
129         config->mach->mach_init(arch_instance, config->mach);
130         /* reset current arch_instance */
131         arch_instance->reset();
132         /* reset all the values of mach */
133         config->mach->mach_io_reset(arch_instance);
```

```
134
135         if(pref->exec_file){
136                 exception_t ret = load_elf(pref->exec_file);
137         }
138
139         skyeye_log(Info_log, __FUNCTION__, "Set PC to the address 0x%x\n",
config->start_address    );
140         /* set pc from config */
141         arch_instance->set_pc(config->start_address);
142
143         pthread_t id;
144         create_thread(skyeye_loop, arch_instance, &id);
166 }
```

SIM_start 函数主要完成如下功能，在代码 110 和 111 行，读入 skyeye.conf 文件并解析文件中的所有配置选项。

代码 117 到 133 行，根据配置选项，对所选择的 arch，mach, memory 等数据结构进行初始化，为运行做准备。

代码 135 行到 137 行，是根据 pref 文件中的设置来加载一个 elf 镜像文件。

代码 141 行来设置仿真目标板的 PC 地址。

144 行用来创建一个线程并执行 skyeye_loop 函数。

3.4 skyeye_loop 函数

skyeye_loop 是仿真平台用来执行每一条指令的主循环，其代码如下：

```
137 /*
138  * mainloop of simulatior
139  */
140 void skyeye_loop(generic_arch_t *arch_instance){
141         for (;;) {
142                 /* chech if we need to run some callback functions at this time */
143                 exec_callback(Step_callback, arch_instance);
144                 while (!running) {
145                         /*
146                          * spin until it's time to go.  this is useful when
147                          * we're not auto-starting.
```

```
148                     */
149                     sleep(1);
150             }
157             /* run step once */
158             arch_instance->step_once ();
159         }
160 }
```

上面的函数主体是一个无限循环，143 行调用 exec_callback 函数，来检测是否在这个指令执行周期中有需要执行的 callback 函数，如果有，则执行。

144 到 150 行是通过检测 running 变量来判断当前仿真目标板是否处于运行状态，如果 running 为 0, 仿真目标板处于停止状态，则调用 sleep 进行睡眠来释放处理器资源。然后继续进行 while 循环。

如果 running 变量为 1，则整个仿真目标板处于运行态，则在 158 行处调用 arch_instance->step_once 来执行目标单板的一条指令。

第四章、PowerPC 处理器仿真模块分析

4.1 PowerPC 仿真模块的背景介绍

SkyEye 中 PowerPC 仿真模块的部分代码来自于 PearPC 项目的代码。SkyEye 目前主要是仿真 e500 系列的处理器，mpc8560 和 mpc8572。其中我们仿真的 mpc8572 处理器是一款双核的处理器，在 SkyEye 的仿真代码中也实现了对双核启动和通信的一些仿真。关于 mpc8560 和 mpc8572 的相关文档可以在 FreeScale 的官方网站上进行下载。

我们可以运行 linux-2.6.22 和 linux-2.6.23 的内核在 SkyEye 的 PowerPC 仿真模块上。

4.2 和 skyEye 核心模块的接口部分

如我们前面介绍，对于需要仿真的每个体系结构都需要实现 arch_config_t 的接口数据结构。在 PowerPC 仿真模块中，实现 arch_config_t 接口的数据结构的代码位于[arch/ppc/common/ppc_arch_interface.c : init_ppc_arch]中，代码如下：

```
350     static arch_config_t ppc_arch;
351
352     ppc_arch.arch_name = "ppc";
353     ppc_arch.init = ppc_init_state;
354     ppc_arch.reset = ppc_reset_state;
355     ppc_arch.set_pc = ppc_set_pc;
356     ppc_arch.get_pc = ppc_get_pc;
357     ppc_arch.get_step = ppc_get_step;
358     ppc_arch.step_once = ppc_step_once;
359     ppc_arch.ICE_write_byte = ppc_ICE_write_byte;
360     ppc_arch.ICE_read_byte = ppc_ICE_read_byte;
361     ppc_arch.parse_cpu = ppc_parse_cpu;
362     ppc_arch.get_regval_by_id = ppc_get_regval_by_id;
363     ppc_arch.get_regname_by_id = ppc_get_regname_by _id;
```

```
364        //ppc_arch.parse_mach = ppc_parse_mach;
365
366        register_arch (&ppc_arch);
```

init_ppc_arch 函数会在模块加载的时候被调用，在上面代码的 366 行 register_arch 完成把 ppc_arch 数据结构注册到 SkyEye 中的过程。后面的过程基本上是 SkyEye 通过操作注册进来的 ppc_arch 的函数指针来获得和设置 PowerPC 仿真模块的各种信息。

描述 PowerPC 处理器状态的数据结构为 PPC_CPU_State，其定义位于 arch/ppc/common/ppc_cpu.h 文件，

```
41 typedef struct PPC_CPU_State_s {
42        e500_core_t * core;
43        uint32_t bptr;
44        uint32_t eebpcr;
45        uint32_t ccsr;
46        uint32_t core_num;
47 }PPC_CPU_State;
```

其中的 core 变量是一个类型为 e500_core_t 的数组，数组中的每一个成员代表了一个 e500 核。数组的大小由 core_num 来确定。在 PowerPC 仿真模块启动的时候，根据 core_num 的数值，对 core 数组进行初始化。

对 core 的初始化在 ppc_cpu_init [arch/ppc/common/ppc_arch_interface.c]函数中，代码如下：

```
73        if(!gCPU.core_num){
74                fprintf(stderr, "ERROR:you need to set numbers of core in mach_init.\n");
75                skyeye_exit(-1);
76        }
77        else
78                gCPU.core = malloc(sizeof(e500_core_t) * gCPU.core_num);
```

4.3 运行流程分析

在 SkyEye 执行 SIM_start 函数之后，skyeye 的配置文件 skyeye.conf 会被解析。SkyEye 会根据配置文件的选项，对各个模块进行了相应的设置，加载我们要运行的 elf 文件或者其他镜像文件，最后设置了我们仿真处理器的 PC 地址，为

运行目标机上的第一条指令做好准备。在 SIM_start 函数的最后会调用 skyeye_loop 函数。

4.4 中断和异常仿真的代码分析

e500 平台的中断和异常的架构是一种向量化的表，一共有 32 个中断向量。其描述位于 e500 的手册

当出现异常的时候，SkyEye 会调用 ppc_exception[位于文件 arch/ppc/common/ppc_e500_exc.c] 函数对各种异常进行模拟，我们以 PowerPC 架构中的时钟模拟为例来介绍异常和中断的触发和模拟的过程。

4.4.1 时钟中断仿真代码分析

PowerPC 的时钟中断涉及到的寄存器为

```
223 exec_npc:
224         if(!ppc_divisor){
225                 dec_io_do_cycle(core);
```

```
226             ppc_divisor = 0;
227         }
228         else
229             ppc_divisor--;
```

在每一个条指令执行的周期中，dec_io_do_cycle 函数被调用，函数的实现如下：

```
28 #define TCR_DIE (1 << 26)
 29 #define TSR_DIS (1 << 27)
 30 void dec_io_do_cycle(e500_core_t * core){
 31         core->tbl++;
 32         /**
 33          *  test DIE bit of TCR if timer is enabled
 34          */
 35         if(!(core->tsr & 0x8000000)){
 36             if((core->tcr & 0x4000000) && (core->msr & 0x8000)) {
 37
 38                 if(core->dec > 0)
 39                     core->dec--;
 40                 /* if decrementer eqauls zero */
 41                 if(core->dec == 0){
49                     ppc_exception(core, DEC, 0, core->pc);
 50                 }
 51             }
 52         }
 53         return;
 54 }
```

第 31 行，对 e500 中的 tbl 寄存器进行累加。然后分别判断 tsr 寄存器的

ppc_exception 中的时钟中断的实现代码如下：

```
78             case DEC:
 79                 core->srr[0] = core->npc;
 80                 core->srr[1] = core->msr;
 81
 82                 /* CE,ME and DE bit unchanged, other bit should be clear*/
 83                 core->msr &= 0x21200;
```

```
84
85                  /* DIS bit is set */
86                  core->tsr |= 0x8000000;
87                  //printf("In %s, timer interrupt happened.\n", __FUNCTION__);
88                  break;
```

4.4.2 数据异常的仿真代码分析

相比于上面的时钟中断异常的仿真，数据 TLB 异常的仿真复杂很多。一般来说，操作系统会在初始化的时候对 TLB 和 MMU 进行初始化，添加一些 TLB 表项，来建立虚实地址的映射。然后在后续的数据和指令访问过程中，TLB 不断的进行虚实地址翻译。一旦一些虚拟地址无法在 TLB 表项找到对应的表项，这时就会产生一个 TLB 异常。会跳转到操作系统的 TLB 异常处理函数进行中异常处理。

SkyEye 中对 e500 的 MMU 的地址翻译过程的仿真代码位于文件 arch/ppc/common/ppc_mmu.c 文件中。

```
 68 int ppc_effective_to_physical(e500_core_t * core, uint32 addr, int flags, uint32 *result){
 69      int i,j;
 70      uint32 mask;
 71      ppc_tlb_entry_t *entry;
 72      int tlb1_index;
 73      int pid_match = 0;
 74
 75      if((gCPU.bptr & 0x80000000) && (addr >> 12 == 0xFFFFF)){ /* if bootpage translation enabled? */
 76          //printf("do bootpage translation\n");
 77          *result = (addr & 0xFFF) | (gCPU.bptr << 12); /* please refer to P259 of MPC8572UM */
 78          return PPC_MMU_OK;
 79      }
 80      i = 0;
 81      /* walk over tlb0 and tlb1 to find the entry */
 82      while(i++ < (L2_TLB0_SIZE + L2_TLB1_SIZE)){
 83          if(i > (L2_TLB0_SIZE - 1)){
 84              tlb1_index = i - L2_TLB0_SIZE;
 85              entry = &current_core->mmu.l2_tlb1_vsp[tlb1_index];
```

```
 86              }
 87              else
 88                      entry = &current_core->mmu.l2_tlb0_4k[i];
 89              if(!entry->v)
 90                      continue;
 91              //if(addr == 0xfdff9080)
 92              //      printf("In %s,entry=0x%x, i = 0x%x, current_core->pir=0x%x\n",
__FUNCTION__, entry, i, current_c     ore->pir);
 93              /* FIXME, not check ts bit now */
 94              if(entry->ts & 0x0)
 95                      continue;
 96              if(entry->tid != 0){
97                      /*
 98                      for(j = 0; j < 3; j++){
 99                              if(current_core->mmu.pid[j] == entry->tid)
100                                      break;
101                      }*/
102                      //printf("entry->tid=0x%x\n", entry->tid);
103                      /* FIXME, we should check all the pid register */
104                      if(current_core->mmu.pid[0] != entry->tid)
105                              continue;
106
107              }
108              if(i > (L2_TLB0_SIZE - 1)){
109                      int k,s = 1;
110                      for(k = 0; k < entry->size; k++)
111                              s = s * 4;
112                      mask = ~((1024 * (s - 1) - 0x1) + 1024);
113              }
114              else
115                      mask = ~(1024 * 4 - 0x1);
116              if(entry->size != 0xb){
117                      if((addr & mask) != ((entry->epn << 12) & mask))
```

```
118                         continue;
119                 /* check rwx bit */
120                 if(flags == PPC_MMU_WRITE){
121                         if(current_core->msr & 0x4000){ /* Pr =1 , we are in user mode
*/
122                                 if(!(entry->usxrw & 0x8)){
123                                         //printf("In %s,usermode,offset=0x%x, entry-
>usxrw=0x%x,pc=0x%x\n", __FU     NCTION__, i, entry->usxrw, current_core->pc);
124                                         ppc_exception(core, DATA_ST, flags, addr);
125                                         return PPC_MMU_EXC;
126                                 }
127                         }
128                         else{/* Or PR is 0,we are in Supervisor mode */
129                                 if(!(entry->usxrw & 0x4)){/* we judge SW bit */
130                                         //printf("In %s,Super mode,entry->usxrw=0x
%x,pc=0x%x\n", __FUNCTION__, e     ntry->usxrw, current_core->pc);
131                                         ppc_exception(core, DATA_ST, flags, addr);
132                                         return PPC_MMU_EXC;
133                                 }
134                         }
135                 }
136
137                 *result = (entry->rpn << 12) | (addr & ~mask); // get real address
138             }
139             else {/*if 4G size is mapped, we will not do address check */
140                 //fprintf(stderr,"warning:4G address is used.\n");
141                 if(addr < (entry->epn << 12))
142                         continue;
143                  *result = (entry->rpn << 12) | (addr - (entry->epn << 12)); // get real
address
144
145             }
146             return PPC_MMU_OK;
147         }
```

最后，如果没有找到合适的 TLB 选项，则会触发 TLB 异常。

```
149     if(flags == PPC_MMU_CODE){
150         ppc_exception(core, INSN_TLB, flags, addr);
151         return PPC_MMU_EXC;
152     }
153     else{
154         if(ppc_exception(core, DATA_TLB, flags, addr))
155             return PPC_MMU_EXC;
156     }
157     return PPC_MMU_FATAL;
```

数据 TLB 异常的代码位于 ppc_exception 函数中，主要做了如下动作

第一步、设置处理器核相应的寄存器，如代码 94 行到 101 行：

```
94          case DATA_TLB:
95              //printf(" In %s, DATA_TLB exp happened, pc=0x%x,addr=0x%x, pir=0x%x\n", __FUNCTION__, core->pc,  a, core->pir);
96              core->srr[0] = core->pc;
97              core->srr[1] = core->msr;
98              //core->esr |= ST;
99              core->dear = a; /* save the data address accessed by exception instruction */
100
101             core->msr &= 0x21200;
```

其中 96 行设置 srr[0]为发生异常的 pc，97 行用来保存发生异常的 MSR 到 srr[1]寄存器中。99 行保存异常访问的数据的地址到处理器核的 dear 寄存器中。最后 101 行根据数据 TLB 异常的定义，设置当前的 MSR 的值。

第二步、更新 MMU 中的相关寄存器，如下代码

```
102             /* Update TLB */
103         /**
104          * if TLBSELD = 00, MAS0[ESEL] is updated with the next victim information for TLB0.Finially,      * the MAS[0] field is updated with the incremented value of TLB0[NV].Thus, ESEL points to
105          * the current victim
106          * (the entry to be replaced), while MAS0[NV] points to the next victim to be used if a TLB0      * entry is replaced
```

```
107             */
108
109             /**
110              * update TLBSEL with TLBSELD
111              */
112                     core->mmu.mas[0] = (core->mmu.mas[4] & 0x10000000) | (core->mmu.mas[0] & (~0x10000000));
113                     /* if TLBSELD == 0, update ESEL and NV bit in MAS Register*/
114                     if(!TLBSELD(core->mmu.mas[4])){
115                             /* if TLBSELD == 0, ESEL = TLB[0].NV */
core->mmu.mas[0] = (core->mmu.tlb0_nv << 18) | (core->mmu.mas[0] & 0xFFF0FFFF);
126                             /* update NV of MAS0 , NV = ~TLB[0].NV */
127                             core->mmu.mas[0] = (~core->mmu.tlb0_nv & 0x3) | (core->mmu.mas[0] & 0xFFFFFFFC);
128                             //printf("In %s,core->mmu.mas[0]=0x%x\n", __FUNCTION__, core->mmu.mas[0]);
129                     }
130             /**
131              *  set zeros of permis and U0 - U3
132              */
133                     core->mmu.mas[3] &= 0xFFFFFC00;
134             /**
135              *  set zeros of RPN
136              */
137                     core->mmu.mas[3] &= 0xFFF;
138
139             /**
140              * Set EPN to EPN of access
141              */
142                     core->mmu.mas[2] = (a & 0xFFFFF000) | (core->mmu.mas[2] &0xFFF);
143             /**
144              * Set TSIZE[0 - 3] to TSIZED
```

```
145             */
146                 core->mmu.mas[1] = (core->mmu.mas[4] & 0xF00)|(core->mmu.mas[1] & 0xFFFFF0FF);
147             /**
148              * Set TID
149              */
150                 core->mmu.mas[1] = (core->mmu.mas[1] & 0xFF00FFFF)|((core->mmu.pid[0] & 0xFF) << 16);
151
152             /**
153              * set Valid bit
154              */
155                 core->mmu.mas[1] = current_core->mmu.mas[1] | 0x80000000;
156             /* update SPID with PID */
157                 core->mmu.mas[6] = (core->mmu.mas[6] & 0xFF00FFFF) | ((core->mmu.pid[0] & 0xFF) << 16);
158                 if(flags == PPC_MMU_WRITE)
159                     core->esr = 0x00800000;
160                 else
161                     core->esr = 0x0;
162                 break;
```

4.5 多核仿真的代码分析

SkyEye 当前实现了 mpc8572 双核处理器的模拟，可以运行支持 SMP 的 Linux 内核。

4.5.1 多核启动分析

多核仿真是通过在一个循环中每个处理器轮流运行一条执行实现的，虽然在 SkyEye 的内部实现中，两个处理器核运行指令是顺序执行，但是对于运行其上的系统软件，它“意识”不到这种顺序执行，它会认为 SkyEye 仿真的两个核是并行的。实现代码如下：

```
246             /* if CPU1_EN is set? */
247             if(!i || gCPU.eebpcr & 0x2000000)
248                 per_cpu_step(current_core);
```

其中 247 行通过判断 eebpcr 寄存器的相应位，判断是否第二个处理器核已经启动。在这个 for 循环中，SkyEye 仿真了两个核的指令执行。

PowerPC 的 e500 系列为了支持第二个核的启动，还添加了 bootpage 的特性。当 bptr 的在开始执行的时候，判断 bptr 寄存器的

```
 75        if((gCPU.bptr & 0x80000000) && (addr >> 12 == 0xFFFFF)){ /* if bootpage translation enabled? */
 76                //printf("do bootpage translation\n");
 77                *result = (addr & 0xFFF) | (gCPU.bptr << 12); /* please refer to P259 of MPC8572UM */
 78                return PPC_MMU_OK;
 79        }
```

下面的代码是对处理器核的初始化，每一个处理器核都有一个单独的寄存器 pir 用来标志自己的 ID。在这个函数中，如代码 77 行，处理器核会根据参数 core_id 来对自己的 pir 寄存器进行初始化。

```
66 /*
67  * Initialization for e500 core
68  */
69 void ppc_core_init(e500_core_t * core, int core_id){
70        // initialize srs (mostly for prom)
71        int j;
72        for (j = 0; j < 16; j++) {
73                core->sr[j] = 0x2aa*j;
74        }
75        //core->pvr = 0x8020000; /* PVR for mpc8560 */
76        core->pvr = 0x80210030; /* PVR for mpc8572 */
77        core->pir = core_id;
78
79        e500_mmu_init(&core->mmu);
80 }
```

4.5.2 多核同步

PowerPC 通过发送 IPI 中断实现了多核之间的同步。实现代码如下：

```
965                 case 0x60040:
 966                         io->mpic.ipidr[0] = data;
 967                         int core_id = -1;
 968                         if (data & 0x1) /* dispatch the interrupt to core 0 */
 969                                 core_id = 0;
 970                         if (data & 0x2) /* dispatch the interrupt to core 1 */
 971                                 core_id = 1;
 972                         if(data & 0x3){
 973                                 /* trigger an interrupt to dedicated core */
 974                                 e500_core_t* core = &cpu->core[core_id];
 975                                 core->ipr |= IPI0;
 976                                 io->mpic.ipivpr[0] |= 0x40000000; /* set activity bit in vpr
*/
 978                                 io->pic_percpu.iack[core_id] = (io-
>pic_percpu.iack[core_id] & 0xFFFF0000) | (io->mpic.ipivpr[0] & 0xFFFF);
 980                                 ppc_exception(core, EXT_INT, 0, 0);
 981                                 core->ipi_flag = 1; /* we need to inform the core that
npc is changed to exception vector */
 983                         }
 984                         return;
```

我们当前只判断了当写入数据等于 3 的时候，我们需要设置处理器核的 ipr 寄存器为相应的中断位。

4.6 在 *SkyEye* 上运行和调试 *PowerPC* 平台的 *Linux*

第五章、ARM 处理器仿真模块分析

5.1 介绍

SkyEye 的 ARM 仿真代码最初的版本来自于 gdb 的 ARMulator 模块。

5.2 与 SkyEye 核心模块的接口

每个处理器架构需要实现

```
void
init_arm_arch ()
{
	static arch_config_t arm_arch;

	arm_arch.arch_name = "arm";
	arm_arch.init = arm_init_state;
	arm_arch.reset = arm_reset_state;
	arm_arch.set_pc = arm_set_pc;
	arm_arch.get_pc = arm_get_pc;
	arm_arch.get_step = arm_get_step;
	arm_arch.step_once = arm_step_once;
	arm_arch.ICE_write_byte = arm_ICE_write_byte;
	arm_arch.ICE_read_byte = arm_ICE_read_byte;
	arm_arch.parse_cpu = arm_parse_cpu;
	//arm_arch.parse_mach = arm_parse_mach;
	//arm_arch.parse_mem = arm_parse_mem;
	arm_arch.parse_regfile = arm_parse_regfile;
	arm_arch.get_regval_by_id = arm_get_regval_by_id;
	arm_arch.get_regname_by_id = arm_get_regname_by_id;

	register_arch (&arm_arch);
}
```

5.3 内部运行流程

在这里我们介绍对于 arm 处理器仿真的一些细节。每一个体系结构的仿真需要实现单步执行的操作。arm 体系结构单步执行的函数实现在文件 arch/arm/common/arm_arch_interface.c 中。实现如下：

```
static void

arm_step_once ()

{

	//ARMul_DoInstr(state);
```

```
        step++;

        cycle++;

        state->EndCondition = 0;

        stop_simulator = 0;

        state->NextInstr = RESUME;      /* treat as PC change */

        state->Reg[15] = ARMul_DoProg(state);

        //state->Reg[15] = ARMul_DoInstr(state);

        FLUSHPIPE;

}
```

5.4 MMU 相关接口和实现

ARM 体系结构的不同处理器核在协处理器 15，也就是内存管理单元的差别也比较大，所以我们设计了一个 MMU 的抽象接口，来屏蔽不同的 MMU 硬件细节。实现代码如下：

```
typedef struct mmu_state_t
{
        ARMword control;
        ARMword translation_table_base;
        ARMword domain_access_control;
        ARMword fault_status;
        ARMword fault_address;
        ARMword last_domain;
        ARMword process_id;
        ARMword cache_locked_down;
        ARMword tlb_locked_down;
//chy 2003-08-24 for xscale
        ARMword cache_type;     // 0
        ARMword aux_control;    // 1
        ARMword copro_access;   // 15

        mmu_ops_t ops;
        union
        {
                sa_mmu_t sa_mmu;
                arm7100_mmu_t arm7100_mmu;
                arm920t_mmu_t arm920t_mmu;
                arm926ejs_mmu_t arm926ejs_mmu;
        } u;
} mmu_state_t;
```

第六章、MIPS处理器仿真模块的代码分析

6.1 背景介绍

MIPS 仿真实现了对 MIPS32 平台的仿真，支持 au1100, godson 处理器的仿真。

6.2 与核心模块的接口部分

```
603     static arch_config_t mips_arch;
604     mips_arch.arch_name = arch_name;
605     mips_arch.init = mips_init_state;
606     mips_arch.reset = mips_reset_state;
607     mips_arch.step_once = mips_step_once;
608     mips_arch.set_pc = mips_set_pc;
609     mips_arch.get_pc = mips_get_pc;
610     mips_arch.ICE_read_byte = mips_ICE_read_byte;
611     mips_arch.ICE_write_byte = mips_ICE_write_byte;
612     mips_arch.parse_cpu = mips_parse_cpu;
613     mips_arch.get_step = mips_get_step;
614     //mips_arch.parse_mach = mips_parse_mach;
615     //mips_arch.parse_mem = mips_parse_mem;
616     mips_arch.get_regval_by_id = mips_get_regval_by_id;
617     mips_arch.get_regname_by_id = mips_get_regname_by_id;
618     register_arch (&mips_arch);
```

6.3 运行流程的代码分析

mips 单步执行一条指令的代码在函数 mips_step_once 函数，实现如下：

```
281 static void
282 mips_step_once()
```

```
283 {
284         mstate->gpr[0] = 0;
285
286         /* Check for interrupts. In real hardware, these have a priority lower
287          * than all exceptions, but simulating this effect is too hard to be
288          * worth the effort (interrupts and resets are not meant to be
289          * delivered accurately anyway.)
290          */
291         if(mstate->irq_pending)
292         {
293                 mips_trigger_irq(mstate);
294         }
295
296         /* Look up the ITLB. It's not clear from the manuals whether the ITLB
297          * stores the ASIDs or not. I assume it does. ITLB has the same size
298          * as in the real hardware, mapping two 4KB pages.  Because decoding a
299          * MIPS64 virtual address is far from trivial, ITLB and DTLB actually
300          * improve the simulator's performance: something I cannot say about
301          * caches and JTLB.
302          */
303
304         PA pa; //Shi yang 2006-08-18
305         VA va;
306         Instr instr;
307         int next_state;
```

```
308     va = mstate->pc;
309     mstate->cycle++;
310     if(translate_vaddr(mstate, va, instr_fetch, &pa) == TLB_SUCC){
311         mips_mem_read(pa, &instr, 4);
312         next_state = decode(mstate, instr);
313         //skyeye_exit(-1);
314     }
315     else{
316         //fprintf(stderr, "Exception when get instruction!\n");
317     }
318
319     /* NOTE: mstate->pipeline is also possibely set in decode function */
332
333     switch (mstate->pipeline) {
334         case nothing_special:
335             mstate->pc += 4;
336             break;
337         case branch_delay:
338             mstate->pc = mstate->branch_target;
339             break;
340         case instr_addr_error:
341             process_address_error(mstate, instr_fetch, mstate->branch_target);
342         case branch_nodelay: /* For syscall and TLB exp, we donot like to add pc
*/
343             mstate->pipeline = nothing_special;
344             return; /* do nothing */
```

```
345         }
346         mstate->pipeline = next_state;
```

310 到 312 行实现了地址翻译，取指，执行指令的过程。

6.4 异常和中断的仿真的代码分析

```
57 void
58 process_exception(MIPS_State* mstate, UInt32 cause, int vec)
59 {
61         UInt32 exc_code = cause & 0x7f;
62         mstate->now += 5;
63         /* we need to modify pipeline according to different exception */
64
65         VA epc;
66         if (!branch_delay_slot(mstate))
67                 epc = mstate->pc;
68         else {
69                 epc = mstate->pc - 4;
70                 cause = set_bit(cause, Cause_BD);
71         }
72
73         if((exc_code == EXC_Sys) || (exc_code == EXC_TLBL)
74                 ||(exc_code) == EXC_CpU || (exc_code == EXC_TLBS) || (exc_code ==
EXC_Mod)){
75                 mstate->pipeline = branch_nodelay;
76                //fprintf(stderr, "KSDBG:1 in %s, vec=0x%x, cause=0x%x, ,v0=0x%x,
pc=0x%x\n", __FUNCTION__, vec, exc_code    , mstate->gpr[2], mstate->pc);
77         }
```

```
78        else{
79                mstate->pipeline = nothing_special;
80        }
81
82        /* Set ExcCode to zero in Cause register */
83        mstate->cp0[Cause] &= 0xFFFFFF83;
84        mstate->cp0[Cause] |= cause;
85                mstate->cp0[EPC] = epc;
86        mstate->pc = vec + (bit(mstate->cp0[SR], SR_BEV) ? general_vector_base :
boot_vector_base);
87        /* set EXL to one */
88        mstate->cp0[SR] |= 0x2;
89        /* Set Exl bit to zero, disable interrupt */
90        mstate->cp0[SR] &= 0xFFFFFFFD;
91        enter_kernel_mode(mstate);
94 }
```

6.5 多核仿真的代码分析

无

第七章、外设仿真－cs8900网卡仿真

5.1 关键数据结构介绍

每一个外设需实现数据结构 device_desc_t，其描述如下：

```
84 typedef struct device_desc
85 {
86        /* device type name.
87         * if inexistance, can be gotten from "mach name"
88         * e.g. ep7312, at91.
89         * */
90        char type[MAX_STR_NAME];
91
92        /* device instance name.
93         * The same type of device may have two or more instances, but they
94         * have different name. "name" can identify different instances.
95         * e.g. the "s3c4510b" uart has two instances: uart1 and uart2.
96         * */
97        char name[MAX_STR_NAME];
98
99        /*I/O or memory base address and size */
100        uint32 base;
101        uint32 size;
102
103        /* interrupt of device.
104         * */
105        struct device_interrupt intr;
106
107        /* mem operation
108         * */
109        struct device_mem_op mem_op;
110
111        void (*fini) (struct device_desc * dev);        /*finish routine */
112        void (*reset) (struct device_desc * dev);       /*reset device. */
113        void (*update) (struct device_desc * dev);      /*called by io_do_cycle */
114
115        int (*filter_read) (struct device_desc *dev, uint32 addr, uint32 *data, size_t
count);
116        int (*filter_write) (struct device_desc *dev, uint32 addr, uint32 data, size_t
count);
117
118        int (*read_byte) (struct device_desc * dev, uint32 addr, uint8 * result);
119        int (*read_halfword) (struct device_desc * dev, uint32 addr,
120                              uint16 * result);
121        int (*read_word) (struct device_desc * dev, uint32 addr, uint32 * result);
122
```

```
123         int (*write_byte) (struct device_desc * dev, uint32 addr, uint8 data);
124         int (*write_halfword) (struct device_desc * dev, uint32 addr, uint16 data);
125         int (*write_word) (struct device_desc * dev, uint32 addr, uint32 data);
126
127         /* refer the "mach" that the device belongs to.
128          * */
129         void *mach;
130
131         /* specific common data for a type of device
132          * */
133         void *dev;
134
135         /* device specific data
136          * usually be an "io" struct.
137          * */
138         void *data;
139 } device_desc_t;
```

device_desc_t 描述了一个外设需要实现的接口和数据变量的实例。每一个这样的数据结构都代表了系统中的一个外设实例或者表示一个实际的物理外设设备。

然后我们在 device/net/dev_net_cs8900a.c 文件中定义了大小为 MAX_DEVICE_NUM 的 cs8900a_devs 这样一个数组，用来代表们最大可以有 MAX_DEVICE_NUM 个 cs8900a 的网卡同时存在在系统中。

```
41 #define MAX_DEVICE_NUM 10
42 static struct device_desc *cs8900a_devs[MAX_DEVICE_NUM];
```

第八章、代码覆盖率模块的实现代码和分析(暂无)

第九章、gdb 远程调试代理模块的实现代码和分析（暂无）