SkyEye 编程手册

版本：draft-0.1

作者：Michael.kang

目录

一、API 接口

1.1 运行控制 *API* 接口

SIM_init()

声明：**void SIM_init()**

功能说明：调用 SIM_init 初始化 Simulator，SIM_init 负责做以下初始化工作：
初始化核心库中的所有数据结构
根据设置的动态库加载路径，加载各种模块
参数：无
返回值：无

SIM_start()

功能说明：根据配置文件，设置特定的数据结构，为运行做好准备。
参数：
返回值：

SIM_run()

功能说明：启动目标机器
参数：无
返回值：

SIM_stop(generic_core_t* core)

功能说明：停止目标机器
参数：指定的处理器核心
返回值：无

SIM_continue(generic_core_t* core)

功能说明：
参数：
返回值：

1.2 命令行接口 ***API***

add_command

声明：exception_t add_command(char* command_str, rl_icpfunc_t* func, const char* help_str);

功能说明：添加一个新的命令到命令行接口中。
参数：command_str 为命令的字符串，func 为执行命令时要调用的函数，help_str 是命令的帮助文档。
返回值：成功返回 No_exp，错误则返回相应异常的类型。

1.3 回调函数接口 *API*

register_callback

声明：exception_t register_callback(callback_func_t func, callback_kind_t kind);

功能说明：注册一个新的回调函数到系统中。
参数：**func** 为要执行的回调函数，kind 是回调函数的类型。
返回值：成功返回 No_exp，错误则返回相应异常的类型。

1.4 配置文件接口 *API*

get_current_config

声明：skyeye_config_t* get_current_config();

功能：获得当前的配置文件的数据结构

参数：无

返回值：当前配置文件的数据结构

skyeye_read_config

声明：exception_t skyeye_read_config (char* conf_filename);

功能：加载并解析相应的 skyeye 配置文件，保存在内存中。

参数：配置文件的文件名。

返回值：成功返回 No_exp，错误则返回相应的异常类型。

register_option

声明：exception_t register_option(char* option_name, do_option_t do_option_func, char* helper);

功能：注册新的配置文件选项到系统中。

参数：option_name 是配置选项的名称，do_option_func 是用来解析配置选项的函数，helper 是该配置选项的帮助文本。

返回值：成功则返回 No_exp，错误则返回相应的异常类型

1.5 uart 相关的 *API*

skyeye_uart_write

声明：

int skyeye_uart_write(int devIndex, void *buf, size_t count, int *wroteBytes[MAX_UART_DEVICE_NUM]);

功能说明：

提供给虚拟串口硬件调用，可以把数据写入底层的物理接口模块中。

参数：

返回值：

skyeye_uart_read

声明：

int skyeye_uart_read(int devIndex, void *buf, size_t count, struct timeval *timeout, int *retDevIndex);

功能说明：提供给虚拟串口硬件调用，可以把数据写入底层的物理链接层模块中。

1.6 模块相关的 *API*

skyeye_load_all_module

声明：void skyeye_load_all_module(const char* lib_dir, char* lib_suffix);

功能描述：

参数：

返回值：

skyeye_load_module

声明：exception_t SKY_load_module(const char* module_filename);

功能描述：

参数：

返回值：

1.7 内存访问相关的 *API*

bus_read

声明：int bus_read(short size, int addr, uint32_t * value);

功能描述：

参数

返回值：

bus_write

声明：int bus_write(short size, int addr, uint32_t value);

功能描述：从总线上写一个数据到某一个地址单元中。

参数：size 用来指定写数据的长度，addr 为写入的地址的值，value 为写入的数据。

返回值：

1.8 机器管理的 *API*

register_mach

声明：void register_mach(char* mach_name, mach_init_t mach_init);

功能描述：注册一个模拟的机器或者单板到系统中。

参数：

返回值：无

get_mach

声明：machine_config_t * get_mach(const char* mach_name);

功能描述：获得一个机器或者单板的数据结构

参数：机器的名称

返回值：模拟的机器的数据结构

二、提供的数据类型

2.1、异常类型

exception_t

声明：

```
typedef enum{
    /* No exception */
    No_exp = 0,
    /* Memory allocation exception */
    Malloc_exp,
    /* File open exception */
    File_open_exp,
    /* DLL open exception */
    Dll_open_exp,
    /* Invalid argument exception */
    Invarg_exp,
    /* Invalid module exception */
    Invmod_exp,
    /* wrong format exception for config file parsing */
    Conf_format_exp,
    /* some reference excess the predefiend range. Such as the index out of array range */
    Excess_range_exp,
    /* Unknown exception */
    Unknown_exp
}exception_t;
```

描述：

No_exp, 当函数按照正常的流程结束，没有发生任何异常。
Malloc_exp, 当函数在分配内存出错的时候，返回的异常类型。
File_open_exp，当文件打开错误返回的异常类型。
Dll_open_exp，当 DLL 文件打开错误返回的异常类型。
Invarg_exp，当传入参数不合法的时候返回的异常类型。
Invmod_exp，当加载的模块不符合规范的时候返回的异常类型
Conf_format_exp，当配置文件选项有错误的时候，返回的异常类型
Excess_range_exp，当给定的索引超出预定义的范围的时候返回的异常类型
Unknown_exp，未知的异常类型。

2.2 回调函数相关的数据类型

callback_kind_t

定义：

```
typedef enum{
    Step_callback = 0,
    Mem_read_callback,
    Mem_write_callback,
    Exception_callback,
    Max_callback
}callback_kind_t;
```

描述：回调函数的类型, 分别描述如下

Step_callback，是用来在虚拟机单步执行的时候可以被调用的函数。目前在断点模块，单步执行模块等中应用。

Mem_read_callback，是在虚拟机读内存的时候可以被调用的函数。

Mem_write_callback，是在虚拟机写内存的时候可以被调用的函数。

Exception_callback，是在虚拟机发生异常的时候可以被调用的函数。

Max_callback，在这里只是用来表示异常类型的数目。

callback_func_t

定义：

```
typedef void(*callback_func_t)(generic_arch_t* arch_instance);
```

描述：

回调函数的定义。

2.3 配置文件相关的数据类型

skyeye_option_t

声明

```
typedef struct skyeye_option_s
{
	char *option_name;
	int (*do_option) (struct skyeye_option_s * this_opion,
			int num_params, const char *params[]);
	char* helper;

	struct skyeye_option_t *next;
} skyeye_option_t;
```

描述：
用来实现每个配置选项的数据结构。其中 do_option 是用来解析模块的函数。helper 是此配置选项的描述。

do_option_t

声明

```
typedef int(*do_option_t)(struct skyeye_option_t *option, int num_params, const char *params[]) ;
```

描述：定义了解析配置文件选项的函数。

2.4 模块相关的数据类型

skyeye_module_t

声明：

```
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;
```

描述：用来记录每个模块的数据结构。

三、编程示例

3.1 记录运行时的 *PC* 的模块编程示例

介绍了 skyeye 模块编程的一个示例程序 log-pc 模块。本模块演示了如何编写独立的 skyeye 模块，并加载编译和运行的过程。模块功能主要是用来记录 skyeye 执行过的所有 PC 指令。

3.1.1 编写相关代码

模块由以下三个文件组成。

log.c：实现了记录 PC 的功能，

```
view plaincopy to clipboardprint?
#include <stdlib.h>
#include <stdio.h>
#include "skyeye_arch.h"
#include "skyeye_callback.h"
/* flag to enable log function. */
static int enable_log_flag;
/* the log file for record. */
static const char* log_filename = "./pc.log";
/* fd of log_filename */
static FILE* log_fd;
/* callback function for step exeuction. Will record pc here. */
static void log_pc_callback(generic_arch_t* arch_instance){
	if(enable_log_flag){
		fprintf(log_fd, "pc=0x%x\n", arch_instance->get_pc());
	}
}
/* enable log functionality */
static void com_log_pc(char* arg){
	enable_log_flag = 1;
}
/* some initialization for log functionality */
int log_init(){
	exception_t exp;
	/* open file for record pc */
```

```
	log_fd = fopen(log_filename, "w");
	if(log_fd == NULL){
		fprintf(stderr, "Can not open the file %s for log-pc module.\n", log_filename);
		return;
	}
	/* register callback function */
	register_callback(log_pc_callback, Step_callback);
	/* add correspinding command */
	add_command("log-pc", com_log_pc, "record the every pc to log file.\n");
}
/* destruction function for log functionality */
int log_fini(){
	if(log_fd != NULL){
		fclose(log_fd);
	}
}
```

log_module.c：用来实现模块的加载和卸载函数

```
view plaincopy to clipboardprint?
#include <stdio.h>
#include <errno.h>
#include "skyeye_types.h"
#include "skyeye_arch.h"
#include "skyeye_module.h"
/* module name */
const char* skyeye_module = "log-pc";
/* module initialization and will be executed automatically when loading. */
void module_init(){
	log_init();
}
/* module destruction and will be executed automatically when unloading */
void module_fini(){
	log_fini();
```

```
}
```

Makefile： 负责编译整个模块，并安装至 skyeye 的模块目录下。

```
SKYEYE_PREFIX=/opt/skyeye/
CC=gcc
liblog.so:
	$(CC) -I${SKYEYE_PREFIX}/include/include -L$(SKYEYE_PREFIX)/lib/skyeye/ -lcommon -shared -oliblog.so log.c log_module.c
clean:
	rm liblog.so *.o -r -f
install:
	cp liblog.so ${SKYEYE_PREFIX}/lib/skyeye/
```

3.1.2 代码编译与安装

编译模块

创建一个新的目录，并把上面的 log.c log_module.c Makefile 都放入此目录。在编译模块之前，先确认 skyeye 已经安装到系统中，可以查看/opt/skyeye/include/include/中是否有相关头文件存在，如 skyeye_types.h, skyeye_arch.h 等等。

如果 skyeye 安装正常，我们可以在你的模块目录下运行"make"来编译模块生成 log.so 文件。

模块安装

运行"make install"命令，可以把你编译的模块安装到 skyeye 的相应模块目录，skyeye 会在启动的时候加载模块。

3.1.3 运行测试

进入到/opt/skyeye/testsuite/arm_hello 目录下，测试我们的 log-pc 模块的运行。运行 skyeye 如下：

```
ksh@linux-gvai:/opt/skyeye/testsuite/arm_hello> ../../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)
```

然后再运行 list-modules 可以发现，我们的 log-pc 模块已经被加载，输出如下：

```
(skyeye)list-modules
Module Name            File Name
nandflash              /opt/skyeye/lib/skyeye/libnandflash.so
arm                    /opt/skyeye/lib/skyeye/libarm.so
log-pc                 /opt/skyeye/lib/skyeye/log.so
bfin                   /opt/skyeye/lib/skyeye/libbfin.so
log-pc                 /opt/skyeye/lib/skyeye/liblog.so
uart                   /opt/skyeye/lib/skyeye/libuart.so
mips                   /opt/skyeye/lib/skyeye/libmips.so
net                    /opt/skyeye/lib/skyeye/libnet.so
code_cov               /opt/skyeye/lib/skyeye/libcodecov.so
sparc                  /opt/skyeye/lib/skyeye/libsparc.so
ppc                    /opt/skyeye/lib/skyeye/libppc.so
touchscreen            /opt/skyeye/lib/skyeye/libts.so
coldfire               /opt/skyeye/lib/skyeye/libcoldfire.so
flash                  /opt/skyeye/lib/skyeye/libflash.so
lcd                    /opt/skyeye/lib/skyeye/liblcd.so
gdbserver              /opt/skyeye/lib/skyeye/libgdbserver.so
(skyeye)
```

然后还可以运行我们在前面 log.c 文件中注册的命令 log-pc，来使能日志功能：

```
ksh@linux-gvai:/opt/skyeye/testsuite/arm_hello> ../../bin/skyeye -e arm_hello
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)start
arch: arm
cpu info: armv3, arm7tdmi, 41007700, fff8ff00, 0
In do_mach_option, mach info: name at91, mach_init addr 0xb72a0f70
```

```
uart_mod:3, desc_in:, desc_out:, converter:
In create_uart_console
cpu info: armv3, arm7tdmi
SKYEYE: use arm7100 mmu ops
In SIM_start, Set PC to the address 0x0


(skyeye)log-pc
(skyeye)
```

如果一切顺利的话，你在退出 SkyEye 的时候，发现当前目录中会多了一个 pc.log 文件，里面记录了你刚刚在 SkyEye 上运行的软件的指令流。

3.2 simple_uart 的外设模拟

3.2.1 simple_uart 模块的介绍

3.2.2 simple_uart 模块的代码实现

3.2.3 simple_uart 模块的代码分析

3.2.4 simple_uart 的编译，加载和使用