SKYEYE

天目建模手册

版本　　1.3.5

日期　2013-12-30

目录

1 引言

1.1文档简介

此文档主要对天目外设建模的方法、接口以及示例进行相关说明，帮助天目用户对天目进行扩展和订制。

1.2适用对象

此文档适用于天目外设建模人员以及部分开发维护人员。

1.3外设建模

外设建模是指根据真实物理设备的功能而抽象出相应模型的过程，天目外设建模即将抽象出来的外设模型运行在天目仿真平台中。

2 综合描述

2.1 外设模型

在真实的硬件系统中存在各种各样的 io 设备，如串口、timer、pci 设备等。一般情况下，只有驱动程序负责跟硬件设备进行交互，上层应用只能看到一个抽象的接口，驱动程序负责完成与真实硬件的交互。

在仿真领域，我们需要模拟硬件设备的模型，也就是用软件来实现硬件的功能，我们将这些软件代码称之为设备模型。根据应用的需求和场景，模型的抽象层次也有所不同。按照抽象层次从低到高的顺序依次如下：门级电路仿真、计算机体系结构仿真、cycle 近似的仿真、快速的全系统仿真。其中抽象的层次越低，对硬件的功能和行为描述越细致，导致仿真的速度也就越慢，通常这种低层次的抽象用在芯片前期的方案验证方面，以便对模型的各个参数和表现有个更好的了解。

快速的全系统仿真则是抽象层次比较高的一个仿真技术，它只关注于软件能够看到的功能和行为，只需要模拟出软件能看到的功能即可，不需要关注真实硬件的实现。在这样的系统里，设备模型是基于传输级的，给软件呈现的是黑盒，对于软件来说重要的是传输的结果

而不是结果是怎么产生的过程。这样的模型因为省掉了大量的硬件操作细节，所以快速的系统仿真能够比较快地模拟出全系统。

2.2建模过程

虚拟化一个系统大致要经过以下几个流程:

l 通过阅读设计文档、程序员编程手册（PRM）以及其他相关文档列出组成系统的外设和处理器清单。

l 根据对系统的使用的预期，对各个设备的抽象层次做一个大致的评估。这个设备是否可以忽略或者是要全部实现吗？（举个例子，在功能仿真中 RTOS 是否需要使用 MMU？如果不需要，就不需要建模了）。

l 重用已经存在的设备模型以及成熟的处理器模型。一般的设备模型都是存在的，直接使用会节省大量时间。

l 使用设备建模语言对那些没有现成的设备进行建模，先尽量忽略细节建立一个能够运行的模型。然后再一边测试一边完善功能，这样会保证及时准确的找到错误。

l 把设备模型放在系统里面进行测试，增加软件所需要的功能。对于那些没有相应软件的设备模型，需要针对模型的各种场景去写测试用例，验证模型的正确性；对于有对应软件的模型，可以直接使用该软件进行测试。

l 通过迭代的方式不断完善设备功能直到设备能够跑起来软件。

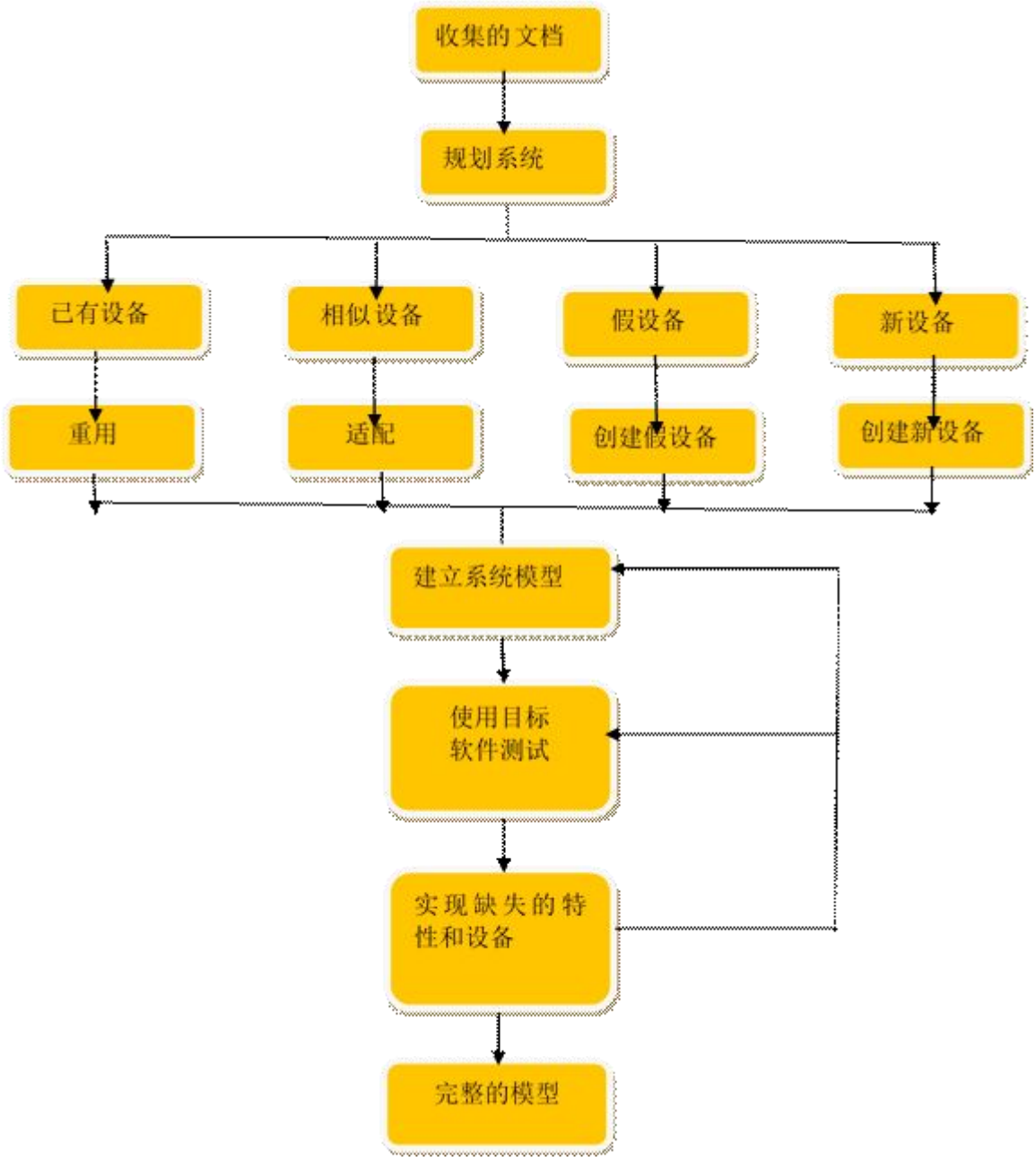

图 2-1 系统建模流程

2.3 设备建模概念

2.3.1 传输级设备建模

传输级设备建模是指与设备的交互是一次性完成的：设备接受请求，计算返回值然后返回。这些都在一个函数调用内完成。典型地，一个处理器要读写某个设备的寄存器，它会发起一个请求，然后通过系统总线根据设备所在总线和地址确定需要访问的设备，然后设备对

于这个请求进行相应的处理，然后将处理的结果返回给处理器。所有这些操作都是通过一个函数调用完成的。传输级的设备建模速度要远远高于模型设备的具体细节如字节是怎么传输通过互联总线的等等。

2.3.2 内存系统建模和地址映射

内存映射是外设模型需要的一个基本服务，也是加快仿真的关键组件。处理器的地址空间是通过内存映射的方式来建模实现的，这样的处理器对设备的操作就会通过内存映射的读写来进行，从而不需要显式地涉及到模型和连接他们的总线等物理连接关系。这样一层抽象把具体的设备访问的屏蔽了，对于处理器来说只需要调用内存映射的访问接口即可，这样就不需要关系这些具体的设备是怎么连接、数据怎么传输等问题，这极大地简化了设备访问所需要的工作量，同时也提供了一致的接口。

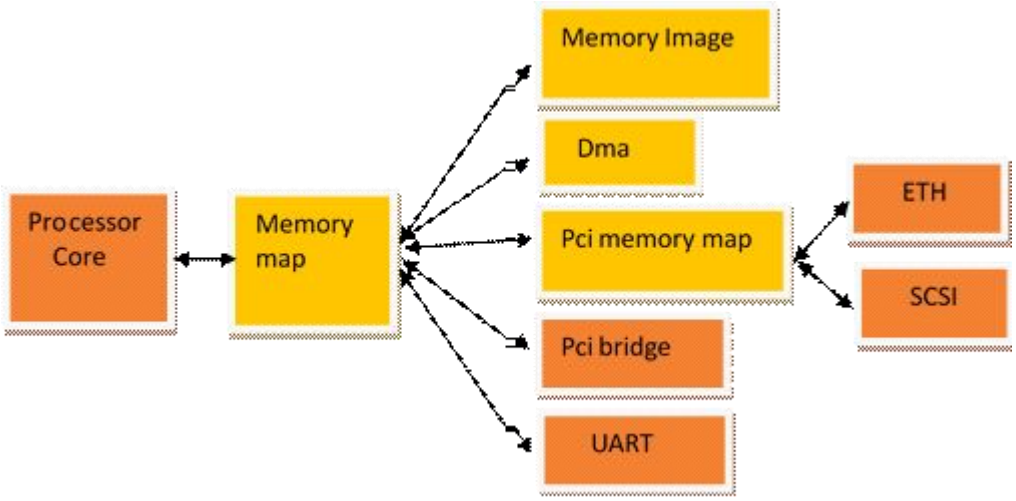

图 2-2 内存系统建模和地址映射

2.3.3 模型属性

设备的模型是基于模块化的，通常一个单板是由很多设备实例组成的。每个实例都有自己的属性。典型的设备属性有寄存器、定时的时长等信息，当然用户也可以定义自己想要保存的设备状态作为属性。属性是设备的有名字的可读可写的特性。属性的值可以是常见的基础数据类型。属性最重要的用途是配置和 ckeckpoint。设备的内部状态必须通过可以属性

获取，这样当前状态的快照就可以通过读取设备的属性然后存储在磁盘里。通过重新加载快照的配置和设置属性，设备的状态可以恢复到之前快照的状态，然后继续仿真。默认情况下，设备的寄存器都是属性，此外还可以是接口，以及使用设备建模语言定义的属性。

2.3.4 模型互联与交互

设备除了属性之外还有接口。接口是一组函数指针，完成某个特定 功能。设备还可以使用端口接口。每个端口接口都有一个名字和一个或多个接口。这样设备可以实现多个相同的接口，然后使用名字将接口区分开。通常一个系统都是由很多设备实现的，设备间的交互必须通过接口来实现。具体实例：当 cpu 想要进行一个内存访问，cpu 将根据访问的物理地址找到该设备，然后调用该设备的 memory_space 接口。接口这个概念包含两个层面的概念，一个是接口使用者，另外一个是接口提供者（实现者）。一般在配置脚本或者组件里，使用者有一个 connect 属性会连接到接口提供者，而这个属性里面包含一个指向接口提供者的指针。设备建模语言里面提供了接口和连接的关键字，用户可以很方面地使用这些概念来实现设备间的交互。

2.3.5 处理器建模

DML 主要是用来开发高性能和抽象系统设备的，并不能使用这个来开发处理器核。在仿真系统里，主机的处理资源绝大多数都用来模拟处理器核，因此处理器需要经过高度优化过的高速仿真核，一般可以使用厂商提供的处理器核心或者开源软件 Qemu 等。

2.3.6 抽象建模方法论

Dml 建模是从函数的角度来建模的，整个传输在一个函数内完成。

- 模型应该关注是什么而不是怎么样。
  - 在功能仿真中，建模的细节是根据运行在该模块的软件的需要来决定的，软件执行中不需要的细节不需要建模。
  - 显式的预先定义的状态可以很容易提供给软件。
- 模型应该根据产品生命周期的不同阶段的需求增量的创建。
  - 最初的模型只需要满足能够让固件开发人员开始工作的需求即可。

l 功能模型能够快速地创建以及跟其他模型连接起来。

Ø 系统配置应该跟模型分开。

2.4 准备系统建模

在开始系统建模之前，需要搜集一些信息。一些有用的文档和信息如下：

l 系统文档如硬件设计文档，包括模块互联图。

l 主芯片、设备、功能模块的 PRM（程序员参考手册）

l BSP（板级支持包）、设备驱动、固件、操作系统等控制硬件的软件源码。

一旦对系统有了大致的了解，需要对要建模的设备进行优先级排序。原则如下:

l 识别出能够运行起软件的最小系统组件。

l 识别出技术风险最大的设备。这个风险既是开发这个设备模型的风险又是控制该设备的软件的风险。这可能是整个项目的瓶颈。

l 识别对系统没有最初影响的设备。这些设备包括没有被操作到的或者没有使用的设备。

l 找出系统已经提供的设备模型。例如 ram、rom、flash 等。

3 建模技术

3.1 外设建模的设计原则如下

l 遵从硬件功能

l 遵从软件需要

l 不要模拟不需要的细节

l 重用和适配现有模型

l 使用设备建模语言建模

3.1.1 遵从硬件功能

外设建模的目的是确保仿真系统上开发的软件在真实物理机器上也能够正常工作，反之

亦然。这就意味着外设模型需要在尽可能抽象硬件功能的时候，同时需要遵从的真实硬件的结构。这个包括硬件配置的变化等。因此，对于所有软件可以看得见的功能必须在仿真中体现出来，最简单的建模方法就是使用跟真实硬件一样的逻辑组建和通信结构。通常对于一个单板来说会有硬件的功能设计图，建模需要参考这些设计图纸了解单板的组建和连接关系。通常 soc 的 datasheet 也包含了逻辑功能图，列出了逻辑功能的划分和连接关系等，设备建模的时候可以参考这些资料资料来划分模型的组成等。

3.1.2 遵从软件需要

通常建模不会实现系统所有的功能，而是只需要实现软件使用到的那些功能。对于一些集成芯片和 soc 来说它们往往具有非常多的功能，但是我们在某一个特定场景下只需要一小部分的功能，因此只需要实现这一小部分功能即可，没有必要一下子把所有的功能都实现了。如果后续需要哪些，后面在实现。在芯片内部，只有那些软件使用到的功能需要实现。这意味着要聚焦在少数几个操作模式或者功能上。例如对于 pci express 桥设备 PEX PLX 8254 可以工作在透明和非透明模式下。如果系统没有使用到非透明模式，它完全可以不实现。

3.1.3 不要模拟不需要的细节

不要模拟软件看不到的硬件细节，模拟细节的代价是将极大地降低仿真速度。好的模型实现了硬件具备的功能而不是硬件怎么实现这些功能的。这个就要求建模要跟设备的功能规格匹配，而不是硬件精确的实现细节。一个例子就是网络设备，在真实世界中网卡需要5/4 编码来传输，但是在设备模型中是将整个报抽象成一个单元发送到链路层的，这简化了实现。

3.1.4 重用和适配现有模型

重用已经存在的模型代码，这将大大缩短建模时间。一般系统会提供大量的模板和实例代码，很多设备模型可以参考和引用这样库来实现。

3.1.5 使用设备建模语言建模

使用设备建模语言进行建模将简化建模的过程，缩短开发时间以及提高建模的速度和质

量。设备建模语言使用了跟硬件相似的概念抽象，可以快速地帮助开发者完成设备模型的功能。

3.2 测试设备模型

设备模型建好之后需要进行测试，一般需要进行单元测试和集成测试。单元测试是指只加载一个设备，针对设备的接口和针对特定的寄存器的功能进行测试，主要是测试设备模型自身的逻辑是否正确。集成测试是指把设备模型放在系统中，然后再加载系统软件进行测试，测试其是否满足系统软件的需求。使用各种 OS/driver 组合来测试设备。使用操作该设备的用户态程序来测试模型的各个特性等。使用仿真系统能够发现模型的问题迹象，这些迹象包括：

1. 模型报告没有实现的特性。这些特性需要实现。
2. 程序，OS，或者虚拟机出现行为异常，这意味着模型的功能有缺陷。
3. 仿真性能很差，模型需要优化。

4 天目外设建模

4.1 外设建模要素

每一个外设都在 skyeye/device 目录下，例如 uart_leon2，进入该目录，里面有三个文件分别是

uart_leon2.c 、uart_leon2.h 、 uart_leon2_module.c

1. uart_leon2.h：声明该外设所有属性、寄存器、接口以及所需的中间变量等等，全部定义到一个设备的结构体中。
2. uart_leon2.c：对该外设的具体实现。
3. uart_leon2_module.c：将该设备定义成一个可动态加载的模块。

4.2 外设建模要素

4.2.1 外设接口

打开 uart_leon2.c 可以看到如下接口：

1. new_leon2_uart：pre_conf_obj 通过调用此接口实例化该设备。

2. free_leon2_uart：当仿真平台不需要该设备，调用该接口释放该设备。

3. leon2_uart_write：对该设备寄存器写接口。

4. leon2_uart_read：对该设备寄存器读接口。

5. leon2_uart_get_regname_by_id:通过寄存器 ID 获得外设寄存器的名字。

6. leon2_get_regvalue_by_id：通过寄存器 ID 获得外设寄存器的有效值。

7. leon2_uart_set_attr：设置外设的属性。

8. leon2_uart_get_attr：获得外设的属性。

9. init_leon2_uart()：对该设备的外设类进行初始化。

4.2.2 模块接口

打开 uart_leon2_module.c，实现如下主要接口，实现模块的动态加载。

1. const char* skyeye_module：skyeye_module 必须定义，来表示模块的名字。

2. module_init()：在天目加载此模块的时候，调用该函数。

3. module_fini()：在天目卸载此模块的时候，调用该函数。

4.3 建模数据结构

4.3.1 基类

说明：

天目中最广泛使用的一个数据结构，一般天目中其它对象都是通过基类进行引用。

定义：

```
typedef struct conf_object_s{
    char* objname;
    void* obj;
    char* class_name;
}conf_object_t
```

void* obj; 存储任意数据结构的指针。

char* objname; 对所存储数据结构进行标记的字符串（方便索引，必须唯一）

char* class_name; 如果基类存储的数据结构是个外设，要初始化此变量。

4.3.2 对象类

说明：

天目建模的每个外设模块都是一个类，例如：s3c6410的触摸屏、LCD、串口，分别都定义为一个对象类。

定义：

```
typedef struct skyeye_class{
      char* class_name;
      char* class_desc;
      conf_object_t* (*new_instance)(char* obj_name);
      exception_t (*free_instance)(conf_object_t* obj);
      exception_t (*reset_instance)(conf_object_t* obj, const
                                    char* parameter);
      attr_value_t* (*get_attr)(const char* attr_name,
                                    conf_object_t* obj);
      exception_t (*set_attr)(conf_object_t* obj, const char*
                           attr_name, attr_value_t* value);
      char** interface_list;
}skyeye_class_t;
```

1. class_name：对象类的名字，如touchscreen_s3c6410、lcd_s3c6410等

2. class_desc：对类的简介

3. conf_object_t* (*new_instance)(char* obj_name);：实例化本对象类的接口，传入一个你喜欢的名字，返回本对象示例化的基类。

4. exception_t (*free_instance)(char* obj_name);：释放一个已经存在的类，传入要释放的外设类名，释放成功返回No_exp;

5. attr_value_t* (*get_attr)(const char* attr_name, conf_object_t* obj);

6. exception_t (*set_attr)(const char* attr_name, conf_object_t* obj, attr_value_t);

以上两个接口分别是，通过一个属性名和外设实体的基类取得或设置指定对象的属性。

7. exception_t (*reset_instance)(conf_object_t* obj, const char* parameter)：重置对象

4.3.3 通用属性

说明：

所有对象的属性，对外界都是以通用属性的数据结构来体现。

定义：

```
typedef struct attr_value {
	value_type_t type;
	union {
		const char *string;		/* Sim_Val_String */
		integer_t integer;		/* Sim_Val_Integer */
		integer_t uinteger;		/* Sim_Val_Integer */
		bool_t boolean;			/* Sim_Val_Boolean */
		double floating;		/* Sim_Val_Floating */
		/* Sim_Val_List */
		struct {
			size_t size;
			struct attr_value *vector;  /* [size] */
		} list;
		/* data array */
		struct {
			size_t size;
			uint8 *data;		/* [size] */
		} data;
		conf_object_t* object;		/* Sim_Val_Object */
		void* ptr;
	} u;
} attr_value_t;
```

value_type_t type：记录通用属性所包含的数据类型。

属性的数据类型如下：

```
typedef enum {
	Val_Invalid = -1,
	Val_String,
	Val_Integer,
	Val_UInteger,
	Val_Floating,
	Val_List,
	Val_Data,
	Val_Nil,
	Val_Object,
	Val_Dict,
	Val_Boolean,
	Val_ptr,
	Sim_Val_Unresolved_Object
} value_type_t;
```

union u：存储属性数据的共用体。

4.4 外设建模接口

4.4.1 中断接口

说明：

为需要触发中断的外设提供发送中断信号的接口。

定义：

```
typedef struct signal_interface{
    conf_object_t* conf_obj;
    int (*raise_signal)(conf_object_t* target, int line);
    int (*lower_signal)(conf_object_t* target, int line);
}general_signal_intf;
#define  GENERAL_SIGNAL_INTF_NAME "general_signal_intf”
```

成员：

conf_object_t* conf_obj：对应触发中断的中断控制器的基类。

int (*raise_signal)(conf_object_t* target, int line);

int (*lower_signal)(conf_object_t* target, int line);

分别为拉高拉低中断地址线的函数，触发中断的方式具体情况具体分析。

GENERAL_SIGNAL_INTF_NAME：调用中断接口的宏（接口名）。

4.4.2 地址接口

说明：

地址接口是映射在地址空间的接口，当 CPU 访问地址空间时，会访问映射在对应地址空间的对象（ram, uart, pci...），从而调用对象的地址空间接口来访问该对象的寄存器或存储空间。

定义：

```
typedef struct memory_space{
    conf_object_t* conf_obj;
    read_byte_t read;
    write_byte_t write;
}memory_space_intf;
#define MEMORY_SPACE_INTF_NAME "memory_space
```

成员：

conf_object_t* conf_obj：保存该设备的基类
read_byte_t read; ： 该设备 IO 空间的读接口
write_byte_t write; ：该设备 IO 空间的写接口
MEMORY_SPACE_INTF_NAME：索引地址接口的宏。

4.4.3 Term 接口

说明：

当串口要显示信息时，需要通过 term 将串口的数据显示出来，相当于超级终端的窗口。

定义：

```
typedef struct skyeye_uart_intf{
        conf_object_t* conf_obj;
        exception_t (*write) (conf_object_t *opaquet, uint32_t*
buf, size_t count);
        exception_t (*read) (conf_object_t *opaque, void* buf,
size_t count);
}skyeye_uart_intf;
#define SKYEYE_UART_INTF       "skyeye_uart_intf"
```

成员：

conf_object_t* conf_obj：保存 term 对象的基类

```
exception_t (*write) (conf_object_t *opaquet, uint32_t* buf,
size_t count);
exception_t (*read) (conf_object_t *opaque, void* buf, size_t
count);
```

读写 term 对象的方法。

SKYEYE_UART_INTF：引用 term 接口的宏。

4.4.4 can 总线接口

说明：

定义：

```
typedef struct can_ops_intf{
        conf_object_t* obj;
        exception_t (*start)(conf_object_t* obj);
        exception_t (*stop)(conf_object_t* obj);
        exception_t (*transmit)(conf_object_t* obj, void* buf, int
                                nbytes);
        exception_t (*receive)(conf_object_t* obj, void* buf, int
                                nbytes);
}can_ops_intf;
#define C conf_object_t* conf_obj;          保存 CAN 对象的基类
```

```
exception_t (*start)(conf_object_t* obj);    启动 CAN 设备。
exception_t (*stop)(conf_object_t* obj);     停止 CAN 设备。
exception_t (*transmit)(conf_object_t* obj, void* buf,
                    int nbytes);             发送 CAN 数据包。
exception_t (*receive)(conf_object_t* obj, void* buf,
                    int nbytes);             接收 CAN 数据包
```

4.4.5 寄存器接口

说明：

每个外设都应该定义自己的寄存器接口，以便外界访问自己的寄存器。

定义：

```
typedef struct skyeye_reg_intf{
       conf_object_t* conf_obj;
       uint32_t (*get_regvalue_by_id) (conf_object_t* conf_obj,
                        uint32_t id);
       char* (*get_regname_by_id) (conf_object_t* conf_obj,
                        uint32_t id);
       exception_t (*set_regvalue_by_id) (conf_object_t*
                        conf_obj, uint32_t value, uint32_t id);
}skyeye_reg_intf;
#define SKYEYE_REG_INTF       "skyeye_register_intf"
```

成员：

conf_object_t* conf_obj：保存外设的基类

get_regvalue_by_id：通过寄存器 ID 获得寄存器值

get_regname_by_id：通过寄存器 ID 获得寄存器名字

set_regvalue_by_id：通过寄存器 ID 设置寄存器的值

4.4.6 IIC 总线接口

说明：这套接口是 IIC 总线实现， IIC 外设需要使用这套接口与 IIC 总线通讯。

定义：

```
typedef struct skyeye_i2c_bus_interface{
   conf_object_t *conf_obj;
   int (*start)(conf_object_t* i2c_bus, uint8 address);
```

```
    int (*stop)(conf_object_t* i2c_bus);
    uint8 (*read_data)(conf_object_t* i2c_bus);
    void (*write_data)(conf_object_t* i2c_bus, uint8 value);
    int (*register_device)(conf_object_t* i2c_bus,
              conf_object_t* device, uint8 address, uint8 mask,
              i2c_device_flag_t flags);
    void (*unregister_device)(conf_object_t* i2c_bus,
           conf_object_t* device, uint8 address, uint8 mask);
}i2c_bus_intf;
#define I2C_BUS_INTF_NAME "i2c_bus"
```

成员：

start：主设备向从设备发起开始通讯信号

stop：主设备向从设备发起停止通讯信号

read_data：主设备向从设备发起读取数据请求， 返回值为读取到的数据

write_data:主设备向从设备发起写数据请求， 参数 value 为要写入的数据。

register_device：IIC 总线使用此接口来向总线注册一个外部设备，一个总线可以注册多个外部设备， 总线使用主设备提供的地址在这些注册好的外部设备中自动寻找与之相匹配的外部设备，然后开始通讯。

unregister_device：IIC 总线使用此接口来卸载一个已经注册到总线的外部设备。

4.4.7 IIC 从设备接口

说明：这套接口是 IIC 总线实现， 注册外设时会自动将这套接口注册给总线， 供通讯时总线调用。

定义：

```
typedef struct i2c_device_interface{
    conf_object_t *obj;
    int (*set_state)(conf_object_t* device, i2c_device_state_t
state, uint8_t address);
    uint8_t (*read_data)(conf_object_t*device);
    void (*write_data)(conf_object_t*device, uint8_t value);
    uint8_t (*get_address)(conf_object_t*device);
}i2c_device_intf;
#define I2C_DEVICE_INTF_NAME "i2c_device"
```

成员：

set_state：总线调用 start 函数是会间接调用 set_state 来设置当前总线的状态： 读， 写， 空闲。

read_data：主设备调用总线的 read_data,总线会间接调用此接口来读从设备

数据。

write_data：主设备调用总线的 write_data,总线会间接调用此接口来写从设备数据。

get_address：总线注册外设时， 通过此接口获知外设地址。

4.4.8 SPI 总线接口

说明：

定义：

```
typedef struct skyeye_spi_bus_interface{
    conf_object_t *conf_obj;
    int (*spi_slave_request)(conf_object_t *spi_bus, int first,
                    int last, char *buf, int len, char *feedback);
    int (*connect_slave)(conf_object_t *spi_bus, char
                    *slave_name);
    int (*disconnect_slave)(conf_object_t *spi_bus, char
                    *slave_name);
    int  (*register_device)(conf_object_t* spi_bus,
                    conf_object_t* device);
    void (*unregister_device)(conf_object_t* spi_bus,
                conf_object_t* device);
}spi_bus_intf;
#define SPI_BUS_INTF_NAME "spi_bus"
```

成员：

spi_slave_request：主设备向从设备发起读写请求

connect_slave： 主设备连接从设备

disconnect_slave：主设备断开从设备连接

register_device：SPI 总线使用此接口来向总线注册一个外部设备，一个总线可以注册多个外部设备， 总线使用主设备提供的从设备名称，在这些注册好的外部设备中自动寻找与之相匹配的外部设备， 然后

开始通讯。

unregister_device：IIC 总线使用此接口来卸载一个已经注册到总线的外部设备。

4.4.9 SPI 从设备接口

说明：这套接口是 IIC 总线实现， 注册外设时会自动将这套接口注册给总线， 供通讯时总线调用。

定义：

```
typedef struct spi_device_interface{
    conf_object_t *obj;
    void (*spi_request)(conf_object_t *slave, int first, int last,
                  char *buf, int len, char *feedback);
    int (*connect_master)(conf_object_t *slave);
    int (*disconnect_master)(conf_object_t *slave);
}spi_device_intf;

#define SPI_DEVICE_INTF_NAME "spi_device"
```

成员：

spi_request： 主设备调用总线的 spi_request,总线会间接调用此接口来读写从设备数据。

connect_master：主设备调用总线的 connect_slave,总线会间接调用此接口。

disconnect_master：主设备调用总线的 disconnect_slave,总线会间接调用此接口。

4.5外设通用接口

4.5.1 new_addr_space

函数原型：

addr_space_t* new_addr_space(char* obj_name)

功能：

实例化一块地址空间

参数：

char* obj_name：要实例化地址空间的名字

返回值：

实例化的地址空间

4.5.2 pre_conf_obj

函数原形：

```
conf_object_t* pre_conf_obj(const char* objname, const char*
                            class_name)
```

功能：

实例化一个已有对象类，并命名为 objname(如实例一个 uart_term，名字为 uart_term0)

参数：

objname:自定义的一个外设名

class_name:已经被注册到 skyeye 中的外设模块的名字 （如：uart_term，uart_leon2）

返回值：

与 class_name 对应的外设基类（包含该外设结构体）

4.5.3 SKY_register_interface

函数原形：

```
exception_t SKY_register_interface(void* intf_obj, const char*
                    objname, const char* iface_name)
```

功能

将设备名为 objname 的接口 intf_obj 注册给 skyeye。

参数：

intf_obj：接口的结构体

objname：此接口所属的外设模块名

iface_name：接口名（每个接口对应一个宏）

返回值：

如果注册成功，返回 No_exp

4.5.4 SKY_get_interface

函数原形：

```
void* SKY_get_interface(conf_object_t* obj, const char*
                        iface_name)
```

功能：

获取 obj 基类对应设备的接口（前提，在建模此设备时，这个接口要通过4.5.3函数注册给 skyeye）

参数：

obj：一个被实例化的外设基类

iface_name：一个接口的接口名，如中断信号接口 GENERAL_SIGNAL_INTF_NAME

返回值：

接口的指针（与 iface_name 对应），在使用之前，需要强制类型转换为对应接口的类型。

4.5.5 add_map

函数原形：

```
exception_t add_map(addr_space_t* space, generic_address_t
                base_addr, generic_address_t length,
                generic_address_t start, memory_space_intf*
                memory_space, int priority, int swap_endian)
```

功能：

映射一个设备的内存接口映射到 memory_space 中，指定的地址和长度，一旦指令访问这块 memory_space 的某个地址，就会调用对应地址被映射的内存接口

参数：

space：一块地址空间的对象

base_addr：映射基地址

length：映射长度

start：基于基地址的起始地址

memory_space：一个建模设备的内存接口

priority：优先级

swap_endian：大小端

返回值

如果映射无错误，返回 No_exp。

4.5.6 make_new_attr

函数原型：

```
attr_value_t* make_new_attr(value_type_t type, void* value)
```

功能：

创建一个通用属性对象。

参数：

Value_type_t type：属性的类型

Void* value： 属性的值

返回值：

返回一个通用属性。

4.5.7 set_conf_attr

函数原型：

```
exception_t set_conf_attr(conf_object_t* obj, char* attr_name,
                    attr_value_t* value)
```

功能：

为一个目标设置属性。

参数：

conf_object_t obj：设置属性的目标对象

char* attr_name：要设置属性的名字

attr_value* value：要设置属性的值

返回值：

如果设置成功：返回 No_exp。

4.5.8 reset_conf_obj

函数原型：

```
exception_t reset_conf_obj(conf_object_t* obj)
```

功能：

复位指定的对象

参数：

conf_object_t* obj：要复位的对象基类

返回值：

如果复位成功，返回 No_exp。

4.5.9 free_conf_obj

函数原型：

```
void free_conf_obj(conf_object_t* obj)
```

功能：

释放以实例化的对象

参数：

要实例化对象的基类

返回值：

无