JES 软件开发指南

文档版本 1.0 – 2022.6.15

1 概述

JES（中维嵌入式系统/设备）软件按照分层分块的原则而设计，分层是从纵向上看对系统进行划分，分块是从横向上看对软件的的划分。

JES 引入“总线”的模式，即 JES 软件由多个进程组成，每个进程都要在 JES 总线上注册，称为应用挂载到总线上。进程通过总线进行通讯，并由总线统一管理。进程是独立的，即进程的运行不依赖于其他进程；进程提供的业务功能会依赖其他进程提供的服务，如果依赖的服务进程缺失，那么该进程不提供的相应的业务功能，可通知日志模块，在用户调用此业务功能时可返回对应的错误码。

JES 软件的很多业务功能是按需加载，把相应的应用挂载的总线上即为加载，不需要的业务功能则不挂载或把应用从系统中删除。

2 JES 软件架构介绍

2.1 JES 软件分层结构

JES 系统从下往上分为 4 层，而 JES 软件主要指上面 3 层：

- 业务应用层

  提供产品的各种业务功能，这些功能可按需加载，由各个进程组成，一个进程提供一个或几个业务功能或服务（以下称为模块）。能够定制开发，支持第三方开发。

- 业务开发与协议接口层

  为业务软件层提供支持，是开发业务功能的 API 和模块间服务协议的接口封装所组成的 SDK，模块间的接口规范在这一层定义。可对第三方提供。

- 基础框架与适配层

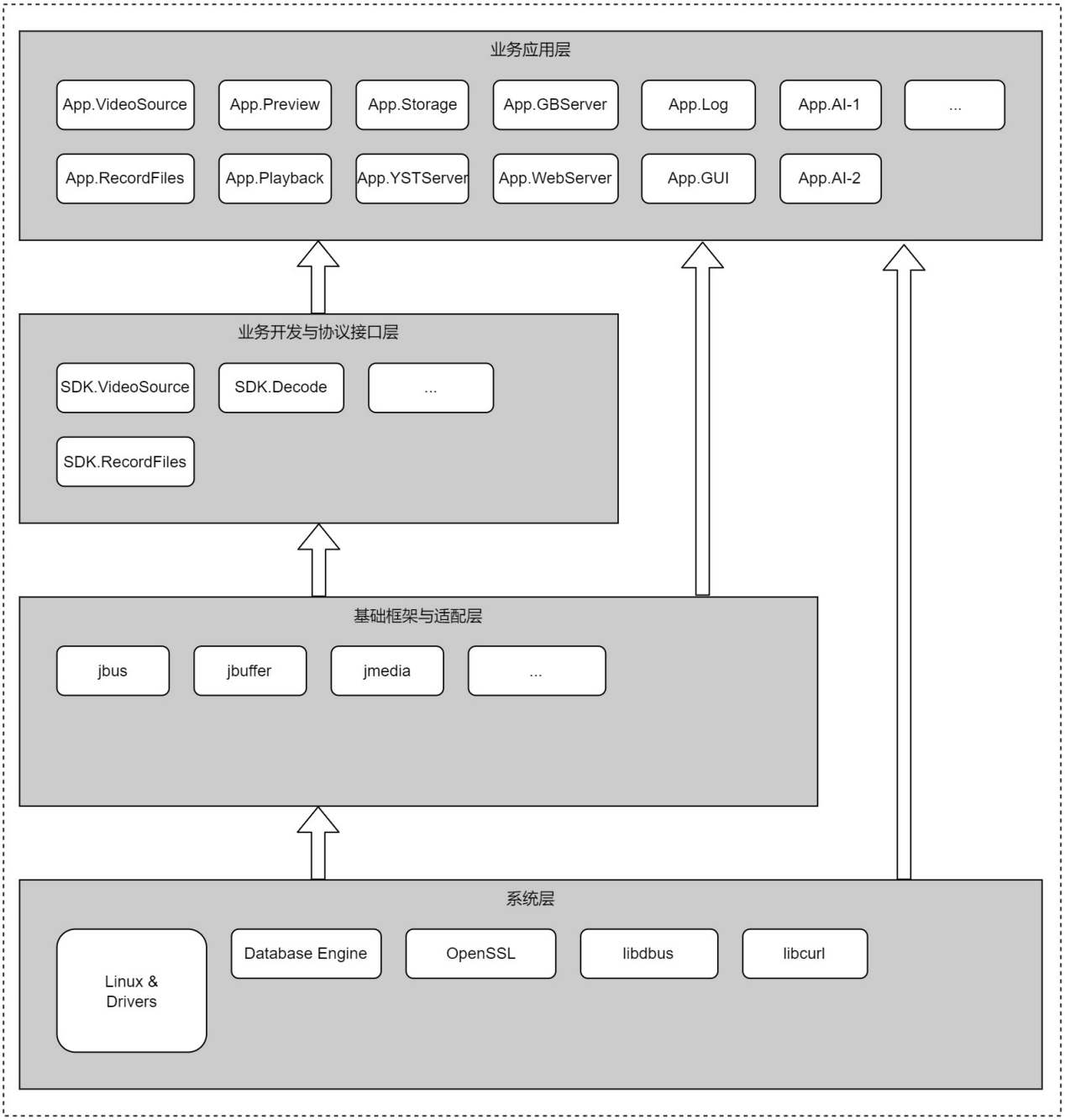
  为上层业务进程提供总线式管理、进程间通讯和共享缓存的机制、编解码管理等基础能力，包括了 jbus、jbuffer、jmedia 等基础库，这一层也负责将硬件平台的差异屏蔽，对上层提供统一的接口。

- 系统层

  由操作系统、驱动、数据库引擎等组成的基本嵌入式系统，包括芯片厂商的提供的 SDK 等，以及一些基本的开源库，如 OpenSSL/WolfSSL、libcurl、libdbus、libdrm 等。

如图 2-1 所示，上一层可以基于或使用下一层的功能，业务应用层可以直接调用系统层的库等，对很多开源库从调用关系上看是直接供业务应用层使用，而这些库可能依赖于系统层中其他开源库的支持，我们仍把这些库看作是系统层的一部分。

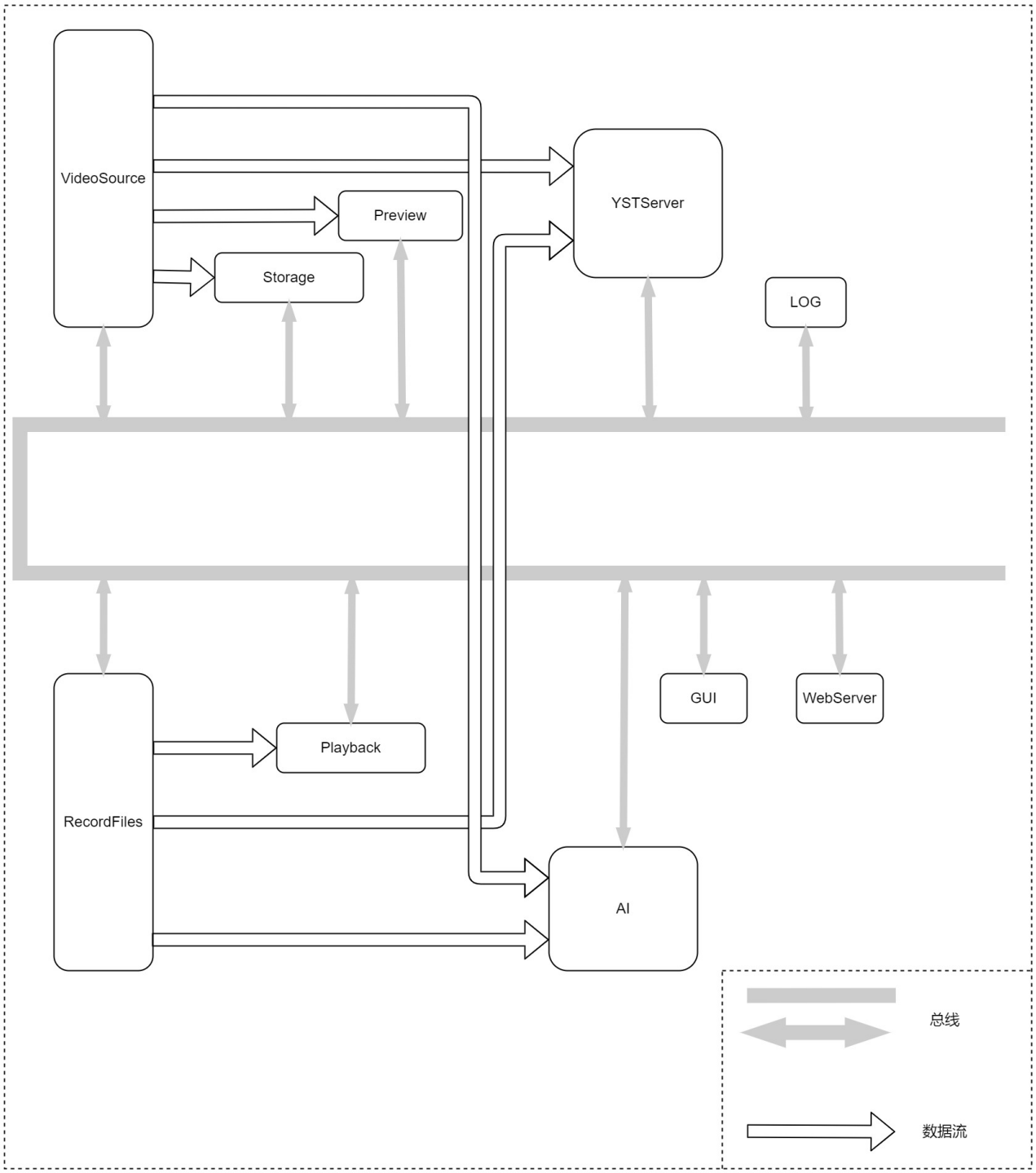
图 2-1 JES 软件分层示意图



2.2 JES 软件模块组成

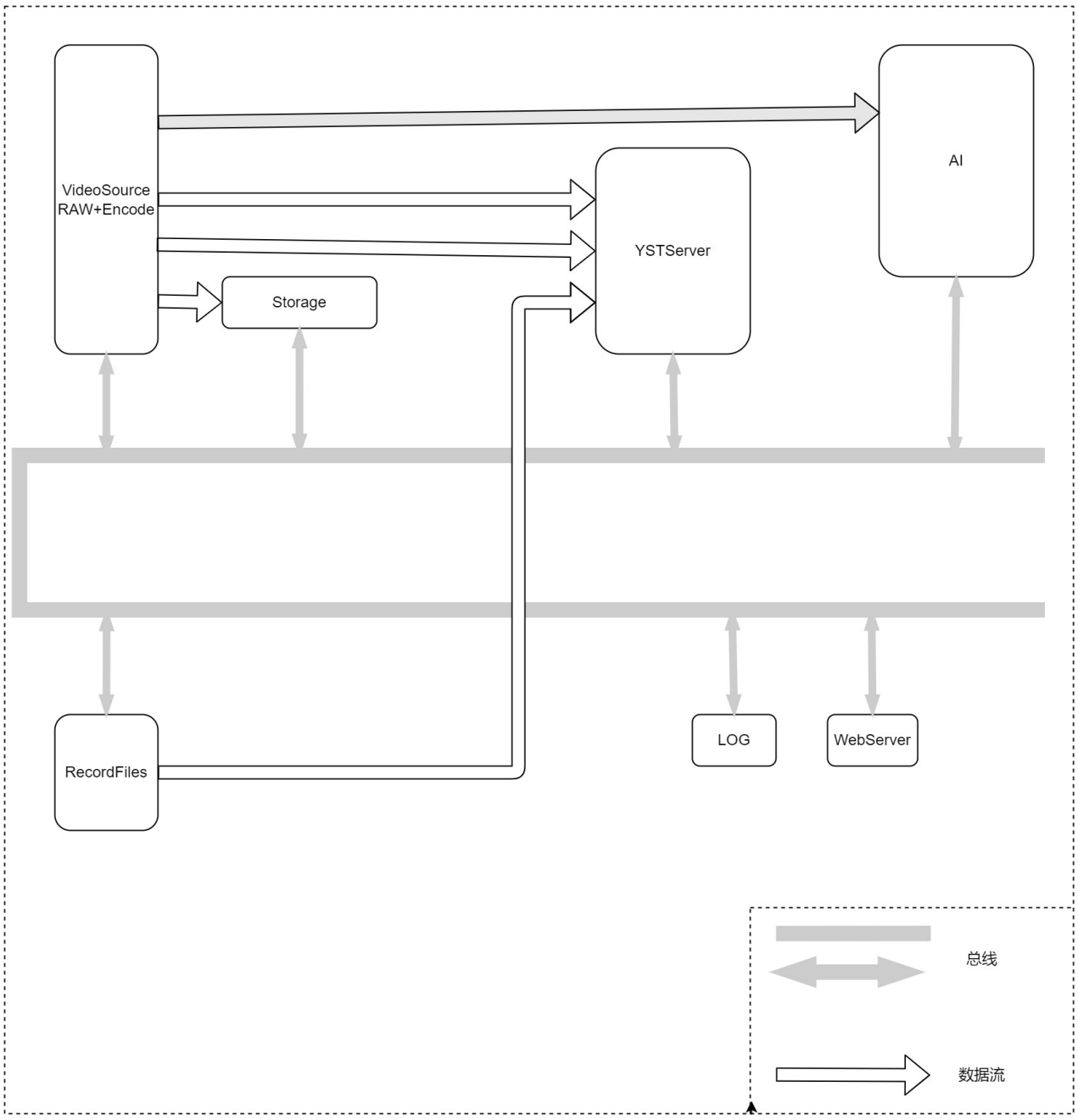
JES 软件的分块主要指在业务软件层和业务开发与协议接口层的模块化，在基础框架与适配层，尽管也是有多个模块组成，但通常我们把它看作一个整体。

图 2-2 NVR 软件组成示意图



如图 2-2 所示，NVR 软件的多个应用全部挂载到总线上，总线由 jbus 创建和管理；NVR 的应用中有两条数据流，一条数据流的源是 VideoSource（视频源，通过网络拉取的 IPC 的数据），另一条数据流的源是 RecordFiles（录像文件）。为了对数据源和录像文件统一管理，其它应用要读取数据都要通过这两条数据流获取。数据流由 jbuffer 实现，经由 jbuffer 将数据分发到不同的应用（进程）。

图 2-3 IPC 软件组成示意图



如图 2-3 所示，在 IPC 的软件中，VideoSource 是 IPC 的基本模块，业务应用通过总线通讯。VideoSource 提供 RAW 数据流、主码流和次码流，RAW 数据流可供 AI 应用使用，编码流提供给各业务应用。同样，总线由 jbus 实现，数据流通过 jbuffer 传输。

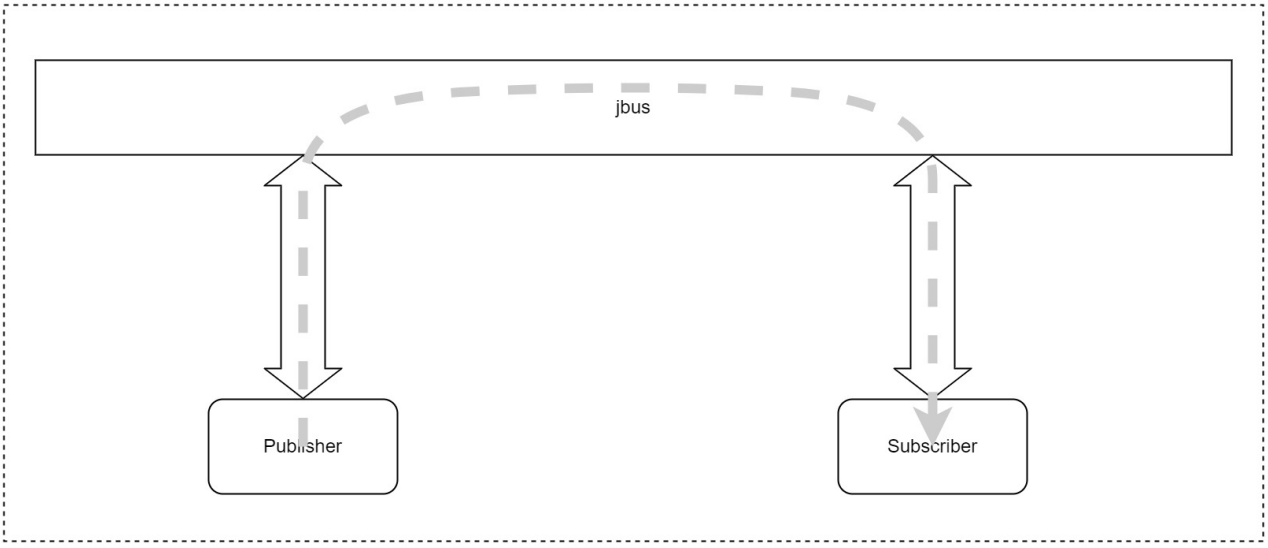
3 基础框架介绍

基础框架实现了 JES 软件总线、JES 共享缓存机制和 JES 编解码统一接口等。基础框架的代码风格更接近于 Linux 内核编码风格。本章节的阅读对象是业务开发与协议接口开发者和业务应用开发者。

3.1 JES 软件总线

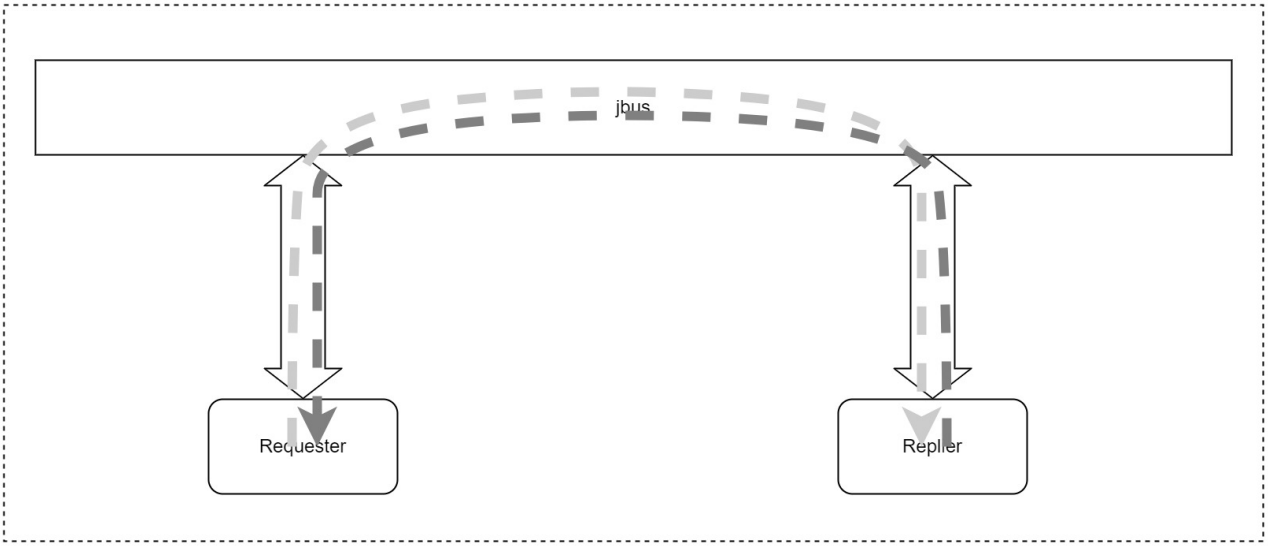
由 jbus 实现，jbus 定义了总线中的 4 种角色：发布者与订阅者、请求者与答复者。jbus 总线解决的是同机不同进程间的通讯，在设计良好的系统中，一般不会丢失数据，可以认为通讯是可靠的。

图 3-1 发布者与订阅者示意图



发布者只负责将信息发布到总线上，并不负责消息是否成功到达订阅者；订阅者将只订阅自己感兴趣的信息。在发布者和订阅者中，都有一个特殊的发布者和订阅者：BROADCAST（interface 为 JBUS_INTERFACE_BROADCAST），一个应用要广播信息，则创建 BROADCAST 发布者，而一个应用要接收广播信息，则创建 BROADCAST 订阅者。BROADCAST 是由 jbus 约定的，它的实现与普通发布者和订阅者无异。

图 3-2 请求者与答复者示意图



请求者发出请求，然后等待答复者的回复，在实现上，这两个步骤是分开的，便于应用决定是以同步（Blocked）的方式接收回复还是以异步（Unblocked）的方式接收回复。在大多数场景中，应用需要以同步的方式完成一次请求与答复，这个过程类似于一次函数调用。应答者创建服务执行请求并对请求者做出回复。

3.1.1 jbus 接口参考

jbus 全局操作：

- jbus_init：jbus 的初始化
- jbus_cleanup：jbus 的清理所占资源

发布者操作：

- jbus_create_publisher：创建发布者
- jbus_destroy_publisher：销毁发布者
- jbus_publish：发布

订阅者操作

- jbus_create_subscriber：创建订阅者
- jbus_destroy_subscriber：销毁订阅者
- jbus_sub_set_callback：设置订阅者回调函数

请求者操作：

- jbus_create_requester：创建请求者
- jbus_destroy_requester：销毁请求者
- jbus_req_set_timeout：设置等待答复的超时
- jbus_request：请求
- jbus_req_get_reply：获取答复

应答者操作：

- jbus_create_replier：创建答复者
- jbus_destroy_replier：销毁答复者
- jbus_rep_set_callback：设置答复者回调函数
- jbus_reply：答复

jbus_init

描述：

jbus 的初始化并创建句柄，后续的一些函数需要使用该句柄。每个使用 jbus 的应用需要且仅调用一次初始化。

声明：

jbus_hdl_t jbus_init(const char *name);

参数：

参数名	描述	输入/输出
name	jbus 的名字，其它进程以名字区分此总线创建的角色。建议以进程名来命名此，即 jbus 的名字为该进程名。	输入

返回：

返回值	描述
句柄	初始化成功，返回 jbus_hdl_t 句柄。
NULL	jbus 初始化失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

一个应用使用 jbus 时，需首先调用此函数。

举例：

无。

jbus_cleanup

描述：

jbus 的清除所占用的资源包括销毁句柄，基于句柄所创建的角色也一并销毁，即调用此函数后，不能再使用 jbus 资源。通常在进程销毁前调用一次此函数。

声明：

void jbus_cleanup(jbus_hdl_t hdl);

参数：

参数名	描述	输入/输出
hdl	句柄（由 jbus_init 所创建）。	输入

返回：

返回值	描述
无	无

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_create_publisher

描述：

基于 jbus（句柄）创建的发布者，一个应用可以创建多个发布者。

声明：

jbus_pub_t *jbus_create_publisher(jbus_hdl_t hdl, const char *iface);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
iface	发布者的 interface（一个字符串），订阅者根据 iface 来订阅。	输入

返回：

返回值	描述
指向发布者的指针	创建成功，返回 jbus_pub_t 指针。
NULL	创建失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_destroy_publisher

描述：

销毁发布者。

声明：

void jbus_destroy_publisher(jbus_hdl_t hdl, jbus_pub_t *pub);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
pub	指向一个发布者的指针。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_publish

描述：

发布者发布信息。

声明：

int jbus_publish(jbus_pub_t *pub, const char *name, const void *content, int length);

参数：

参数名	描述	输入/输出
pub	指向发布者的指针（由 jbus_create_publisher 创建）。	输入
name	发布信息的名称，订阅者据此识别发布信息的内容。	输入
content	指向发布内容的指针，发布内容的数据类型由发布者与订阅者约定，即可以是数值、字符串或者结构体，根据 name 确定。	输入

length	发布内容的长度。	输入

返回：

返回值	描述
0	发布成功（表明已发布到总线上，不关心订阅者是否接收到）。
非 0	发布失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_create_subscriber

描述：

基于 jbus（句柄）创建的订阅者，一个应用可以创建多个订阅者。

声明：

```
jbus_sub_t *jbus_create_subscriber(jbus_hdl_t hdl, const char *iface);
```

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
iface	订阅者所订阅的 interface，根据 iface 确定订阅的哪个发布者。	输入

返回：

返回值	描述
指向订阅者的指针	创建成功，返回 jbus_sub_t 指针。
NULL	创建失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_destroy_subscriber

描述：

销毁订阅者。

声明：

void jbus_destroy_subscriber(jbus_hdl_t hdl, jbus_sub_t *sub);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
sub	指向一个订阅者的指针。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_sub_set_callback

描述：

设置订阅者回调函数。

声明：

int jbus_sub_set_callback(jbus_sub_t *sub, jbus_subscriber_cb cb, void *arg);

参数：

参数名	描述	输入/输出
sub	指向订阅者的指针（由 jbus_create_subscriber 创建）。	输入
cb	指向回调函数的指针。	输入
arg	指向一个参数的指针，此参数地址会原样作为回调函数的参数传递到回调函数。	输入

返回：

返回值	描述
0	成功。
非 0	失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

一个订阅者必须设置回调函数才能接收到订阅信息，回调函数可以多次设置，最后设置会覆盖以前的设置，即最后设置的才是有效的。

举例：

```
static void subsciber_cb(void *ctx, const char *name, const void *content, int length)
{
    jbus_sub_t *sub = ctx;
```

```
    ......
}

int main()
{
    jbus_hdl_t hdl = jbus_init("test");
    jbus_sub_t *sub = jbus_create_subscriber(hdl, "test.hello");
    jbus_sub_set_callback(sub, subsciber_cb, sub);
    while (1)
    {
        sleep(1);
    }

    return 0;
}
```

jbus_create_requester

描述：

基于 jbus（句柄）创建的请求者，一个应用可以创建多个请求者。

声明：

```
jbus_req_t *jbus_create_requester(jbus_hdl_t hdl, const char *dest, const char *iface);
```

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
dest	目的地（一个字符串），即向谁（jbus 的名字）请求。	输入
iface	请求的 interface（一个字符串），请求者与答复者根据 iface 来形成请求与答复。	输入

返回：

返回值	描述
指向请求者的指针	创建成功，返回 jbus_req_t 指针。

NULL	创建失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_destroy_requester

描述：

销毁请求者。

声明：

void jbus_destroy_requester(jbus_hdl_t hdl, jbus_req_t *req);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
req	指向一个请求者的指针。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_req_set_timeout

描述：

设置等待答复的超时。一个请求者创建后，有默认等待答复的超时时间，用户根据需求修改超时值，避免请求者陷入等待错误的答复者的情况，如答复者出现异常或无答复者。

声明：

int jbus_req_set_timeout(jbus_req_t *req, int timeout);

参数：

参数名	描述	输入/输出
req	指向请求者的指针。	输入
timeout	超时值，单位毫秒。	输入

返回：

返回值	描述
0	成功。
非 0	失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_request

描述：

请求者发出请求，该函数立即返回。

声明：

int jbus_request(jbus_req_t *req, const char *name, const void *arg, int length);

参数：

参数名	描述	输入/输出
req	指向请求者的指针。	输入
name	请求的操作名称，请求者与答复者根据 name 来约定作何操作。	输入
arg	请求的操作所带的参数的指针。	输入
length	参数的长度。	输入

返回：

返回值	描述
0	成功。
非 0	失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_req_get_reply

描述：

等待并获取答复者的答复，此函数阻塞直到收到答复或超时。

声明：

int jbus_req_get_reply(jbus_req_t *req, void *buffer, int length);

参数：

参数名	描述	输入/输出
req	指向请求者的指针。	输入
buffer	存放答复结果的缓冲区。	输出
length	缓冲区的大小。	输入

返回：

返回值	描述
大于等于 0	成功获得答复，返回答复结果的大小。
小于 0	失败或超时。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

```
int main()
{
    char buffer[128] = {};
    jbus_hdl_t hdl = jbus_init("test");
    jbus_req_t *req = jbus_create_requester(hdl, "rep.test", "hello");
    jbus_request(req, "get_hello", NULL, 0);
    jbus_req_get_reply(req, buffer, sizeof(buffer));
    ......

    return 0;
}
```

jbus_create_replier

描述：

基于 jbus（句柄）创建的答复者，一个应用可以创建多个答复者。

声明：

jbus_rep_t *jbus_create_replier(jbus_hdl_t hdl, const char *iface);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
iface	答复者的 interface（一个字符串），请求者与答复者根据 iface 来形成请求与答复。	输入

返回：

返回值	描述
指向答复者的指针	创建成功，返回 jbus_rep_t 指针。
NULL	创建失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_destroy_replier

描述：

销毁答复者。

声明：

void jbus_destroy_replier(jbus_hdl_t hdl, jbus_rep_t *rep);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
rep	指向一个答复者的指针。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

无。

jbus_rep_set_callback

描述：

设置答复者回调函数。

声明：

```
int jbus_rep_set_callback(jbus_rep_t *rep, jbus_replier_cb cb, void *arg);
```

参数：

参数名	描述	输入/输出
rep	指向答复者的指针。	输入
cb	指向回调函数的指针。	输入
arg	指向一个参数的指针，此参数地址会原样作为回调函数的参数传递到回调函数。	输入

返回：

返回值	描述

0	成功。
非 0	失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

回调函数可以多次设置，最后设置会覆盖以前的设置，即最后设置的才是有效的。

举例：

无。

jbus_reply

描述：

答复请求者具体内容，此函数通常在答复回调函数中调用。如果答复者没有调用此函数，那么请求者收到的就是空答复（类似与函数调用返回值为 void）。

声明：

```
int jbus_reply(jbus_rep_t *rep, const void *arg, int length);
```

参数：

参数名	描述	输入/输出
rep	指向答复者的指针。	输入
arg	答复操作所带的参数指针，参数即答复的具体内容。	输入
length	参数的长度。	输入

返回：

返回值	描述
0	成功。
非 0	失败。

需求：

头文件：jbus.h

库文件：libjbus.so

注意：

无。

举例：

```
static void replier_cb(void *ctx, const char *name, const void *content, int length)
{
    jbus_rep_t *rep = ctx;
    ......
    jbus_reply(rep, &value, sizeof(value));
    ......
}

int main()
{
    jbus_hdl_t hdl = jbus_init("test");
    jbus_rep_t *rep = jbus_create_replier(hdl, "test.hello");
    jbus_rep_set_callback(rep, replier_cb, rep);
    while (1)
    {
        sleep(1);
    }

    return 0;
}
```

3.1.2 jbus 数据类型

- jbus_ctx_t：定义 jbus 的上下文
- jbus_hdl_t：定义 jbus 句柄
- jbus_pub_t：定义发布者

- jbus_sub_t：定义订阅者
- jbus_req_t：定义请求者
- jbus_rep_t：定义答复者
- jbus_subscriber_cb：定义订阅者的回调函数
- jbus_replier_cb：定义答复者的回调函数

jbus_ctx_t

说明：

定义 jbus 的上下文，成员对用户不可见。

定义：

typedef struct _jbus_ctx jbus_ctx_t;

jbus_hdl_t

说明：

定义 jbus 的句柄，是 jbus 的上下文的指针。

定义：

typedef struct _jbus_ctx* jbus_hdl_t;

jbus_pub_t

说明：

定义发布者，成员对用户不可见。

定义：

typedef struct _jbus_publisher_ctx jbus_pub_t;

jbus_sub_t

说明：

定义订阅者，成员对用户不可见。

定义：

typedef struct _jbus_subscriber_ctx jbus_sub_t;

jbus_req_t

说明：

定义请求者，成员对用户不可见。

定义：

typedef struct _jbus_requester_ctx jbus_req_t;

jbus_rep_t

说明：

定义答复者，成员对用户不可见。

定义：

typedef struct _jbus_replier_ctx jbus_rep_t;

jbus_subscriber_cb

说明：

定义订阅者的回调函数。回调函数的参数：

void* ： 在设置回调函数时给定的参数地址，可用作传递上下文。

const char* ： 接收到的发布信息的名称。

const void* ： 接收到的发布信息内容的地址。

int ： 接收到的发布信息内容的长度。

定义：

typedef void (*jbus_subscriber_cb)(void*, const char*, const void*, int);

jbus_replier_cb

说明：

定义答复者的回调函数。回调函数的参数：

void* ： 在设置回调函数时给定的参数地址，可用作传递上下文。

const char* ： 接收到的请求名称。

const void* ： 接收到的请求所带参数的地址。

int ： 接收到的请求所带参数的长度。

定义：

typedef void (*jbus_replier_cb)(void*, const char*, const void*, int);

3.1.3 场景举例说明

有一个传感器应用，采集了温度（temperature）、湿度（humidity）、大气压（atmospheric_pressure）、电压（voltage）、电流（ampere）等数据。这些数据被分为两类：环境（environment）、电气（electric），发布信息就按这两类来发：

interface 有： sensor.environment、sensor.electric

name 有： temperature、humidity、atmospheric_pressure、volage、ampere

创建发布者：

```
jbus_pub_t *sub_env = jbus_create_publish(hdl, "sensor.environment");
jbus_pub_t *sub_ele = jbus_create_publish(hdl, "sensor.electric");
```

发布信息：

```
int ret;
float value = 25.8; // 25.8°C
ret = jbus_publish(sub_env, “temperature”, &value, sizeof(value));
value = 52.6; // 52.6%
```

```
ret = jbus_publish(sub_env, “humidity”, &value, sizeof(value));
value = 3.3; // 3.3V
ret = jbus_publish(sub_ele, “voltage”, &value, sizeof(value));
value = 0.562; // 562mA
ret = jbus_publish(sub_ele, “ampere”, &value, sizeof(value));
```

上面按照类别创建了两个发布者，订阅者则可以按照类别分别订阅，如果不关心某一个类别可以不订阅。我们不对每一个数据各创建发布者，是为了避免发布和订阅时过于繁琐，现在，按照类别来发布和订阅对软件设计和代码编写来说更简洁。

3.2 JES 共享缓存机制

由 jbuffer 实现，jbuffer 定义了 2 种角色：数据生产者与数据消费者。

数据生产者创建一个环形缓存区，并对外公开这个缓存区的 name，jbuffer 对缓存区按队列进行管理。数据生产者直接将数据送进队尾，不负责判断数据是否被全部取走，如果缓冲区满，最后的数据将覆盖最早的数据，并更新队列状态。

数据消费者根据缓存区的 name 打开一个缓存区，然后从队首获取数据。数据消费者打开的缓存区是相互独立的，从一个打开的缓存区获取数据并不会对另外打开这个缓存区的数据产生影响，从逻辑上看，这个队列对每一个数据消费者来说都是私有的。数据消费者总是从队首取走数据，如果数据生产者覆盖了队首数据并更新了队列，那么数据消费者就错过一次数据并从更新后的队首获取数据。

在一个系统中，根据缓存区的大小，允许数据消费者以不均匀的速度取走数据（如果缓存区够大，就允许等待数据消费者更多的时间）。一段给定的时间内，数据消费者取走数据的速度要大于或等于数据生产者的送入数据的速度，如果不满足这一要求，则是系统设计错误，例如，磁盘存储器写入带宽总是小于数据产生的量，无论如何设计缓存都无法满足数据存储的要求。

使用 jbuffer 应首先启动 jbased（jbase 的 deamon 进程）。

3.2.1 jbuffer 接口参考

数据生产者操作：

- jbuffer_create：创建共享缓存
- jbuffer_destroy：销毁共享缓存
- jbuffer_alloc：从共享缓存申请一块区域
- jbuffer_release：释放从共享缓存申请的区域

数据消费者操作：

- jbuffer_get_name：检索已创建的共享缓存的名字
- jbuffer_open：打开共享缓存
- jbuffer_close：关闭共享缓存
- jbuffer_get：从共享缓存获取数据
- jbuffer_get_last：从共享缓存获取最后一帧数据

jbuffer_create

描述：

创建共享缓存区并创建句柄。

声明：

jbuffer_hdl_t jbuffer_create(const char *name, int size);

参数：

参数名	描述	输入/输出
name	创建的共享缓存区的名字，数据消费者根据 name 打开该缓存区。	输入
size	创建的共享缓存区的大小。	输入

返回：

返回值	描述
句柄	创建成功，返回 jbuffer_hdl_t 句柄。
NULL	创建失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_destroy

描述：

销毁共享缓存区。

声明：

void jbuffer_destroy(jbuffer_hdl_t hdl);

参数：

参数名	描述	输入/输出
hdl	句柄（由 jbuffer_create 所创建）。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_alloc

描述：

从已创建的共享缓存中申请一块区域。

声明：

void *jbuffer_alloc(jbuffer_hdl_t hdl, int size);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
size	申请的区域的大小。	输入

返回：

返回值	描述
指向缓存区域的指针	申请成功。
NULL	申请失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_release

描述：

释放由 jbuffer_alloc 申请的共享缓存区域，此函数调用后这块区域进入队列。

声明：

int jbuffer_release(jbuffer_hdl_t hdl, int size);

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
size	对申请的缓存区域实际使用的大小（也就是加入队列的实际大小），这个 size 不能大于申请时的 size。	输入

返回：

返回值	描述
0	成功。
非 0	失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_get_name

描述：

按给定的顺序号检索出已创建的共享缓存的名字。

声明：

```
const char *jbuffer_get_name(int id, char *name);
```

参数：

参数名	描述	输入/输出
id	共享缓存创建顺序值，从 0 开始。	输入
name	存放检索到的共享缓存名字的缓冲区。	输出

返回：

返回值	描述
字符串指针	成功，指针指向输入参数 name。
NULL	失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_open

描述：

打开共享缓存区，并创建句柄。

声明：

jbuffer_hdl_t jbuffer_open(const char *name);

参数：

参数名	描述	输入/输出
name	要打开的共享缓存区的名字（一个字符串）。	输入

返回：

返回值	描述
句柄	打开成功，返回 jbuffer_hdl_t 句柄。
NULL	打开失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_close

描述：

关闭共享缓存区，并销毁句柄。

声明：

void jbuffer_close(jbuffer_hdl_t hdl);

参数：

参数名	描述	输入/输出
hdl	句柄（由 jbuffer_open 创建）。	输入

返回：

返回值	描述
无	无。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_get

描述：

从共享缓存获取数据（得到存放数据的缓存区指针）。

声明：

const void *jbuffer_get(jbuffer_hdl_t hdl, int *size);

参数：

参数名	描述	输入/输出

hdl	句柄。	输入
size	指向变量的指针，变量指出了保存数据的缓冲区的大小（数据长度）。	输出

返回：

返回值	描述
指向存放数据的缓存区指针	成功。
NULL	失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

jbuffer_get_last

描述：

从共享缓存获取最后一帧数据（得到存放数据的缓存区指针）。

声明：

```
const void *jbuffer_get_last(jbuffer_hdl_t hdl, int *size);
```

参数：

参数名	描述	输入/输出
hdl	句柄。	输入
size	指向变量的指针，变量指出了保存数据的缓冲区的大小（数据长度）。	输出

返回：

返回值	描述
指向存放数据的缓存区指针	成功。
NULL	失败。

需求：

头文件：jbuffer.h

库文件：libjbuffer.so

注意：

无。

举例：

无。

3.2.2 jbuffer 数据类型

- jbuffer_ctx_t：定义 jbuffer 的上下文
- jbuffer_hdl_t：定义 jbuffer 的句柄

jbuffer_ctx_t

说明：

定义 jbuffer 的上下文，成员对用户不可见。

定义：

typedef struct _jbuffer_ctx jbuffer_ctx_t;

jbuffer_hdl_t

说明：

定义 jbuffer 的句柄，是 jbuffer 的上下文的指针。

定义：

typedef struct _jbuffer_ctx* jbuffer_hdl_t;

3.3 JES 编解码统一接口

待补充。

3.4 JES Board 板级硬件适配接口

另附文档。

4 业务开发与协议接口介绍

JES 的协议接口以 SDK 的方式提供，SDK 所包含的 API 全部以 JES_为前缀，以模块或功能簇为次前缀，例如 JES_MSS_StreamOpen()。

4.1 NVR

待补充。

4.2 IPC

待补充。

5 业务应用介绍

5.1 NVR

待补充。

5.2 IPC

待补充。

6 系统层部分规定

6.1 JES 启动流程

JES 系统的内核启动流程按 SoC 芯片提供商提供的方式，内核启动完成后，init 进程是 BusyBox init。不同于 SysV init，BusyBox init 不支持 runlevels，因此 runlevels 字段被忽略。init 启动后执行以下动作：

- 读取/etc/inittab 并按 inittab 所定义的行为执行，如果没有/etc/inittab 文件，BusyBox init 将按默认行为执行：

  ::sysinit:/etc/init.d/rcS

  ::askfirst:/bin/sh

  ::ctrlaltdel:/sbin/reboot

  ::shutdown:/sbin/swapoff -a

  ::shutdown:/bin/umount -a -r

  ::restart:/sbin/init

- 执行/etc/init.d/rcS 脚本，一个 rcS 脚本如下：

```
#!/bin/sh


# Start all init scripts in /etc/init.d
# executing them in numerical order.
#
for i in /etc/init.d/S??* ;do

    # Ignore dangling symlinks (if any).
    [ ! -f "$i" ] && continue

    case "$i" in
      *.sh)
          # Source shell script for speed.
          (
           trap - INT QUIT TSTP
           set start
           . $i
          )
          ;;
      *)
          # No sh extension, so fork subprocess.
          $i start
          ;;
    esac
done
```

- 按照顺序号执行/etc/init.d/S[0-9][0-9]*脚本：

  JESBase 启动顺序号为 S40jesbase，JES 的应用启动顺序号在 S50 ~ S80 范围内。

  脚本要至少实现 start 和 stop，例：

```
#!/bin/sh

NAME=xxx
DAEMON=/usr/bin/$NAME
PID_FILE="/var/run/$NAME.pid"
CONF_FILE="/etc/$NAME/$NAME.conf"

[ -r /etc/default/$NAME ] && . /etc/default/$NAME

start() {
      echo -e "Starting $NAME: "
      start-stop-daemon -S -q -p $PID_FILE --exec $DAEMON -- -f $CONF_FILE
      echo "OK"
}
stop() {
      echo -e "Stopping $NAME: "
      start-stop-daemon -K -q -p $PID_FILE
      echo "OK"
}
restart() {
      stop
      start
}

case "$1" in
  start)
      start
      ;;
  stop)
      stop
      ;;
  restart|reload)
      restart
      ;;
  *)
      echo "Usage: $0 {start|stop|restart}"
      exit 1
esac

exit $?
```

6.2 文件系统

JES 使用 Linux 的 OverlayFS。JES 的基本文件系统为 squash 文件系统（只读），扩展文件系统根据存储介质可为 jffs2、ubi、ext4 等文件系统（可读写），两种文件系统堆叠在一起，在逻辑上是可读写的一套文件系统。

6.3 系统分区

至少包括如下分区：

- boot
- boot_env (64KB)
- factory (64KB)
- kernel
- rootfs
- rootfs_data

6.4 软件包管理

JES 使用 OPKG 进行软件包管理，在 JES 系统上发行的应用以 ipk 提供。

JES 的软件包至少要包括以下文件：

- CONTROL/conffiles

  记录配置文件名称，在升级时这些被记录的文件不会被新版本覆盖。

- CONTROL/control

  例：

  Package: xxx

  Version: 1.0.0

  Depends: libc, libjbus, libjbuffer

License:

Maintainer: Jovision

Architecture: arm

Installed-Size: nnn

Description: xxx xxx xxx

xxx xxx

- CONTROL/postinst

  安装后执行的脚本，如果安装后要启动应用，则

  /etc/init.d/S??xxx start

- CONTROL/prerm

  卸载前执行的脚本

- etc/init.d/S??xxx

  开机自动启动文件

- usr/bin/xxx

  应用程序