SPRD Inter-Processor Communication

Version: 0.5

Date: 2013-12-02



[www.spreadtrum.com](http://www.spreadtrum.com)

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Author | Description |
| 0.1 | 2012-08-02 | Gaopeng | Initial version for SIPC design |
| 0.2 | 2012-11-03 | Gaopeng | Add SBLOCK module design |
| 0.3 | 2012-12-31 | Gaopeng | Update for Customer Version |
| 0.4 | 2013-02-16 | Gaopeng | Update Calibration part |
| 0.5 | 2013-04-03 | Gaopeng | Update modem functions for dual SIPC and DSDA |
| 0.6 | 2013-12-02 | Qiu Yi | Optimize SBLOCK Module and add SBUF/ SBLOCK recovery design |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[SPRD Inter-Processor Communication 1](#_Toc352785655)

[Revision History 2](#_Toc352785656)

[Table of Contents 3](#_Toc352785657)

[1. Introduction 4](#_Toc352785658)

[1.1. Purpose 4](#_Toc352785659)

[1.2. Abbreviations 4](#_Toc352785660)

[2. SIPC Overview 5](#_Toc352785661)

[3. SIPC Modules Design 6](#_Toc352785662)

[3.1. SMSG 7](#_Toc352785663)

[3.2. SMEM 10](#_Toc352785664)

[3.3. SBUF 12](#_Toc352785665)

[3.4. SBLOCK 16](#_Toc352785666)

[3.5. User Modules on SBUF/SBLOCK 19](#_Toc352785667)

[3.5.1. SPIPE - VBPIPE 19](#_Toc352785668)

[3.5.2. STTY - VCONSOLE 20](#_Toc352785669)

[3.5.3. SETH - VETH 20](#_Toc352785670)

[3.5.4. SAUDIO - VAUDIO 21](#_Toc352785671)

[3.6. SRPC 21](#_Toc352785672)

[4. Modem Functions on SIPC 23](#_Toc352785673)

[4.1. Overview 23](#_Toc352785674)

[4.2. Dual-Sim-Dual-Active 24](#_Toc352785675)

[4.3. Modem Recovery 26](#_Toc352785676)

[4.4. Modem Calibration 26](#_Toc352785677)

[4.4.1. Calibration on Bootloader 26](#_Toc352785678)

[4.4.2. Calibration on Linux Boot 27](#_Toc352785679)

# Introduction

## Purpose

This document is to propose an overall design for Spreadtrum Inter-Processor Communications between AP and CP system. It covers the function design, module diagram, user interface, and utility in modem system.

## Abbreviations

* AP Application Processor system
* CP Communication Processor system
* IPI Inter-Processor Interrupt
* SIPC SPRD Inter-Processor Communication
* SMSG SIPC message module
* SMEM SIPC memory management module
* SBUF SIPC FIFO buffer interface module
* SBLOCK SIPC block interface module
* SPIPE SIPC pipe driver module
* STTY SIPC simulated tty module
* SAUDIO SIPC simulated audio module
* SETH SIPC simulated eth module

# SIPC Overview

The fundamental concepts and ideas to build Inter-Processor Communication in Spreadtrum multi-processor system are,

1. A SMSG module is used to efficiently distribute the low-latency events between AP and CP, which is based on IPI and Shared Ringbuffer.
2. A SMEM module is used to manage the shared memory and data swap between AP and CP, which is basically managed in AP side and shared to CP.
3. The general communication interfaces (such as SBUF, SBLOCK, SRPC) are fully based on SMSG and SMEM. The simulated pipes between AP and CP can be implemented with SBUF interface. SBLOCK can be used to implement Ethernet and Audio channel.
4. AP the basic end of SIPC that can communicate with all other CP ends. No IPC between CPs.



图-1

Currently there are 4 end points (AP, TD, WCDMA, WNC) of SIPC in Shark system. There’ll be LTE modem in the future.

/\* sipc processor ID definition \*/

enum {

SIPC\_ID\_AP = 0, /\* Application Processor \*/

SIPC\_ID\_CPT, /\* TD processor \*/

SIPC\_ID\_CPW, /\* WCDMA processor \*/

SIPC\_ID\_WCN, /\* Wireless Connectivity \*/

SIPC\_ID\_NR,

};

# SIPC Modules Design

The basic modules and blocks of SIPC are showed in the below figure,



* SMSG

A fundamental low-level message upon IPI and shared ring-buffer

* SMEM

Share memory management module implemented in AP. CP can do remote Call.

* SBUF

A general bi-directional FIFO interface that can configure the size and number upon SMSG and SMEM module.

* SBLOCK

A general bi-directional block interface that can share memory blocks between AP and CP.

* SRPC

The implementation of SPRD Remote-Procedure-Call.

SPIPE/STTY/SETH/SAUDIO can be implemented based on SBUF and SBLOCK. They are corresponding to the VBPIPE/VCONS/VETH/VAUDIO in VLX system. Initially we’ll implement these interfaces to be compatible and re-use some functions in the VLX system. The SRPC requires the whole AP/CP system re-constructed, which may be moved forward step by step.

## SMSG

Physically there’s an IPI and shared memory for SPRD embedded AP/CP communication. SMSG is an event distribution module that can broad and queue low latency events between AP and CP. The message are queued and transferred in bi-directional ring-buffer to avoid lock or mutex protection. The IPI is used to trigger the notification for each SMSG event. The ring-buffer may be on DDR memory or RAM in chip.

The unit of SMSG is described by an 8-byte structure,

/\* share-mem ring buffer short message \*/

struct smsg {

uint8\_t channel; /\* channel index \*/

uint8\_t type; /\* msg type \*/

uint16\_t flag; /\* reserved \*/

uint32\_t value; /\* msg body \*/

};

The channel concept is virtualized in SMSG to separate the message owners or consumers. We can support 256 independent channels. To keep the channel mechanism simple and fast, the messages in ring-buffer are directly distributed in IRQ context. The SMSG read/write interfaces are not thread-safe. The owner of a channel should be a thread context, which owns the thread to re-distribute the channel message and guarantee the sequence. The message handler priority is dependent on the owner thread priority. The channels are pre-defined as,

/\* smsg channel definition \*/

enum {

SMSG\_CH\_CTRL = 0, /\* some emergency control \*/

SMSG\_CH\_COMM, /\* general communication channel \*/

SMSG\_CH\_RPC\_AP, /\* RPC server channel in AP side \*/

SMSG\_CH\_RPC\_CP, /\* RPC server channel in CP side \*/

SMSG\_CH\_PIPE, /\* general pipe channel \*/

SMSG\_CH\_PLOG, /\* pipe for debug log/dump \*/

SMSG\_CH\_TTY, /\* virtual serial for telephony \*/

SMSG\_CH\_DATA, /\* 2G/3G wireless data \*/

SMSG\_CH\_PCM, /\* voice data stream \*/

SMSG\_CH\_VBC, /\* audio control channel \*/

SMSG\_CH\_NR, /\* channel number \*/

};

When system boots up, AP and CP sides requires drivers or modules to own these channels. The CTRL channel is reserved for emergency message; COMM channel is reserved for general communication; RPC\_AP/RPC\_CP channels are reserved for SRPC.

The types of SMSG are defined，

/\* smsg type definition \*/

enum {

SMSG\_TYPE\_OPEN = 0, /\* first msg to open a channel \*/

SMSG\_TYPE\_CLOSE, /\* last msg to close a channel \*/

SMSG\_TYPE\_DATA, /\* data, value=addr, no ack \*/

SMSG\_TYPE\_EVENT, /\* event with value, no ack \*/

SMSG\_TYPE\_CMD, /\* command, value=cmd \*/

SMSG\_TYPE\_DONE, /\* return of command \*/

SMSG\_TYPE\_SMEM\_ALLOC, /\* allocate smem, flag=order \*/

SMSG\_TYPE\_SMEM\_FREE, /\* free smem, flag=order, value=addr \*/

SMSG\_TYPE\_SMEM\_DONE, /\* return of alloc/free smem \*/

SMSG\_TYPE\_FUNC\_CALL, /\* RPC func, value=addr \*/

SMSG\_TYPE\_FUNC\_RETURN, /\* return of RPC func \*/

SMSG\_TYPE\_NR, /\* type number \*/

};

The OPEN and CLOSE type message are used to open and close SMSG channel. The first message on a channel must be OPEN. AP and CP send an OPEN message to the other side when they want to open a channel. After they get the OPEN from the other side, it means the handshake to open a channel is done, or they may have to stopped and wait for the response message. When a CLOSE message is got from a channel, no more response in this channel. To guarantee the OPEN/CLOSE valid, a magic value should be sent.

#define SMSG\_OPEN\_MAGIC 0xBEEE

#define SMSG\_CLOSE\_MAGIC 0xEDDD

The other message types are general ones. DATA is used to send data address without ack. EVENT is used to send simple event value without ack. CMD and DONE are pairs for a command and response. SMEM\_\* and FUNC\_\* are the extention of CMD and DONE. We define them because they are special cases.

In bootloader, the shared memory for SMSG should be cleared to make the shared ring-buffer clean. Then AP and CP boot up independently. The channel open is to send an OPEN and receive an OPEN, then get done. Because the action of receiving OPEN may be hold due to async of AP and CP boot, the channel open functions should be called in a thread context to avoid stopping boot sequence.

The basic interface of SMSG is smsg\_send()/smsg\_recv() upon a bi-directional ring-buffer. The unit in the ring-buffer is an 8-byte smsg structure. In the receiver side, there’s a cache buffer to queue the channel message for each channel in IRQ context. Afterwards, the channel thread is waken up to consume the smsg.



To guarantee the channel open synchronization, the interface smsg\_ch\_open() is declared. Please note that smsg\_ch\_open() may be stopped if the other side is not responding. So that smsg\_ch\_open() should be call in an individual thread context that we usually call channel thread.



According to the channel close sequential diagram, please note that smsg\_ch\_close() sends a close message by triggering a ipi irq to notify the opposite side, but it never guarantees that this message can be received. And it is also necessary to wake up the uplayer users for this channel, on which they are blocking, is gonna be closed.



The detailed declaration of SMSG interfaces are,

/\*\*

\* smsg\_ch\_open -- open a channel for smsg

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int smsg\_ch\_open(uint8\_t dst, uint16\_t channel, int timeout);

/\*\*

\* smem\_ch\_close -- close a channel for smsg

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int smsg\_ch\_close(uint8\_t dst, uint16\_t channel, int timeout);

/\*\*

\* smsg\_send -- send smsg

\*

\* @msg: smsg body to be sent

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int smsg\_send(struct smsg \*msg, int timeout);

/\*\*

\* smsg\_recv -- poll and recv smsg

\*

\* @msg: smsg body to be received

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int smsg\_recv(struct smsg \*msg, int timeout);

## SMEM

SMEM is a module to manage the shared memory, which is basically a memory allocator in AP side. The CP side can get the memory via SMSG communication. SMEM directly manages the physical address space.

AP manages the physical memory blocks with a first-fit allocator. The minimal unit is 4KB page in allocation. The interface in AP can call the allocator directly. The CP interface is a wrapper of SMSG communication and AP allocator. CP side can call SMEM msgs in any channel that should guarantee the thread-safe by itself. The sequence flowchart is as below. (Currently we didn’t implement it because SBUF/SBLOCK has private memory share protocol. It’s reserved will be required for SRPC in the future).



The interfaces in AP side is,

/\*\*

\* smem\_alloc -- allocate shared memory block

\*

\* @order: 2^order to be allocated

\* @return: phys addr or 0 if failed

\*/

uint32\_t smem\_alloc(uint32\_t order);

/\*\*

\* smem\_free -- free shared memory block

\*

\* @addr: smem addr to be freed

\* @order: 2^order to be freed

\*/

void smem\_free(uint32\_t addr, uint32\_t order);

## SBUF

In simple words, SBUF is an interface that can implement multiple bi-directional FIFOs for AP and CP communication. A SBUF instance consumes a SMSG channel and builds multiple ring-buffers upon SMEM.



Initially AP side caller allocates a SMEM block and constructs it as the following format. CP side should reverse the tx/rx.



In the head of the memory blocks, the ring number, rx/rx ring address, size, R/W pointer are recorded. The tx ring size and tx ring size can be different. When one of them is 0, it becomes a one directional FIFO. The data structure of the ring header is,

/\* ring pipe header \*/

struct sbuf\_ring\_header {

/\* send-buffer info \*/

uint32\_t txbuf\_addr;

uint32\_t txbuf\_size;

uint32\_t txbuf\_rdptr;

uint32\_t txbuf\_wrptr;

/\* recv-buffer info \*/

uint32\_t rxbuf\_addr;

uint32\_t rxbuf\_size;

uint32\_t rxbuf\_rdptr;

uint32\_t rxbuf\_wrptr;

};

/\* sbuf\_mem is the structure of smem for rings \*/

struct sbuf\_mem\_header {

uint32\_t bufnum;

struct sbuf\_ring\_header rings[0];

};

The SBUF module has create, destroy, read, write interfaces. SMSG channel is not thread safe, so that SBUF creates a channel thread to receive, distribute the message and wake up the stopped read/write actions. A read/write action is guaranteed to be completed and non-splitted.

The sequence of SBUF functions are below. To create a SBUF instance, a shared memory block is allocated from SMEM in AP side, and it’s initialized for header and ring-buffer structure. Afterwards, a channel thread is created. This thread opens a SMSG channel and waits for a CMD message in AP side. After it gets a CMD from CP side, it send a DONE msg with the address of smem block. CP can construct the FIFO structures after getting it. So that AP and CP is not symmetric for SBUF creation. AP does parameter configuration, smem allocation and initialization, and CP requests the information from AP to build its own structure. This design can make the CP SBUF recovery easier when CP is rebooted.

The SBUF tx-ring and rx-ring are implemented as ring-buffer. SMSG events are used as notification. The read/write interface can accept non-limited size other than ring-buffer size and return the exact size that sends or receives. The sender will send an event to notify ring-buffer write pointer update to wake up the receiver read thread. The receiver will send an event to notify ring-buffer read pointer update to wake up the sender write thread. The sender may be stopped if the tx buffer is full, and wait for R pointer updated. The receiver may be stopped if the rx buffer is empty, and wait for W pointer updated. To raise the throughput of SBUF, the ring buffer size may be bigger since each read/write requires a SMSG event.



As a host side, AP plays a maintain role to manage the resource of SBUF. When CP happends to reset, in AP side, it is necessary to recover SBUF related information to avoid the influence caused by dirty data left in the ring buffer. During the CP reset, AP keeps working as usual until it receives a OPEN massage from CP. AP should accomplish sbuf recovery before the link construction.



The detailed declaration of SBUF interfaces are,

/\*\*

\* sbuf\_create -- create pipe ring buffers on a channel

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @txbufsize: tx buffer size

\* @rxbufsize: rx buffer size

\* @bufnum: how many buffers to be created

\* @return: 0 on success, <0 on failue

\*/

int sbuf\_create(uint8\_t dst, uint16\_t channel, uint32\_t bufnum,

uint32\_t txbufsize, uint32\_t rxbufsize);

/\*\*

\* sbuf\_destroy -- destroy the pipe ring buffers on a channel

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @return: 0 on success, <0 on failue

\*/

void sbuf\_destroy(uint8\_t dst, uint16\_t channel);

/\*\*

\* sbuf\_write -- write data to a sbuf

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @bufid: buffer ID

\* @buf: data to be written

\* @len: data length

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: tx bytes on success, <0 on failue

\*/

int sbuf\_write(uint8\_t dst, uint16\_t channel, uint32\_t bufid,

void \*buf, uint32\_t len, int timeout);

/\*\*

\* sbuf\_read -- write data to a sbuf

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @bufid: buffer ID

\* @buf: data to be written

\* @len: data length

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: rx bytes on success, <0 on failue

\*/

int sbuf\_read(uint8\_t dst, uint16\_t channel, uint32\_t bufid,

void \*buf, uint32\_t len, int timeout);

## SBLOCK

SBLOCK is a module that provides another data communication interface other than SBUF. SBUF transfers data as stream directly in ring-buffer FIFO. SBLOCK shares the data blocks and transfer block pointer and size in FIFO. In some cases, SBLOCK may avoid memory copy.



The TX blocks are stored in a TX POOL. The sender should get a free block from the pool, then it sends to the opposite side. The receiver should release the block back to the pool after it uses the block. Usually sblock\_send/sblock\_release are non-blocked since the total block numbers are fixed and equal to pool or fifo size. And sblock\_get/sblock\_recv may be blocked when the block pool or fifo is empty.

Initially AP allocates physical memory from SMEM and construct it as the following layout. Then it creates a thread to open SMSG channel communication, and waits for CP request. When a CMD from CP is got, AP sends the smem address and lets CP construct the sblock structure. Similar to SBUF, AP is the host server to do parameter configuration, smem allocation and initialization.

As a host side, AP plays a maintain role to manage the resource of SBLOCK. When CP happends to reset, in AP side, it is necessary to recover SBLOCK related information to avoid the influence caused by dirty data left in the ring buffer. During the CP reset, AP keeps working as usual until it receives a OPEN massage from CP. AP should accomplish sblock recovery before the link construction.





The data structure of sblock header is,

struct sblock\_mng\_header {

sblock\_ring\_header ring;

sblock\_ring\_header pool;

}

struct sblock\_ring\_header {

/\* send-block ring info \*/

uint32\_t txblk\_addr;

uint32\_t txblk\_count;

uint32\_t txblk\_size;

uint32\_t txblk\_blks;

uint32\_t txblk\_rdptr;

uint32\_t txblk\_wrptr;

/\* recv-block ring info \*/

uint32\_t rxblk\_addr;

uint32\_t rxblk\_count;

uint32\_t rxblk\_size;

uint32\_t rxblk\_blks;

uint32\_t rxblk\_rdptr;

uint32\_t rxblk\_wrptr;

};

To make the interfaces more convenient, a callback interface is added to make sblock\_get() and sblock\_receive() called asynchronously. This call back is called in the channel thread context. The detailed declaration of SBLOCK interfaces are,

/\* sblock structure: addr is the uncached virtual address \*/

struct sblock {

void \*addr;

uint32\_t length;

};

/\*\*

\* sblock\_create -- create sblock manager on a channel

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @txblocknum: tx block number

\* @txblocksize: tx block size

\* @rxblocknum: rx block number

\* @rxblocksize: rx block size

\* @return: 0 on success, <0 on failue

\*/

int sblock\_create(uint8\_t dst, uint8\_t channel,

uint32\_t txblocknum, uint32\_t txblocksize,

uint32\_t rxblocknum, uint32\_t rxblocksize);

/\*\*

\* sblock\_destroy -- destroy sblock manager on a channel

\*

\* @dst: dest processor ID

\* @channel: channel ID

\*/

void sblock\_destroy(uint8\_t dst, uint8\_t channel);

#define SBLOCK\_NOTIFY\_GET 0x01

#define SBLOCK\_NOTIFY\_RECV 0x02

#define SBLOCK\_NOTIFY\_STATUS 0x03

/\*\*

\* sblock\_register\_notifier -- register a callback that's called

\* when a tx sblock is available or a rx block is received.

\* non-blocked sblock\_get or sblock\_receive can be called.

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @handler: a callback handler

\* @event: SBLOCK\_NOTIFY\_GET, SBLOCK\_NOTIFY\_RECV, or both

\* @data: opaque data passed to the receiver

\* @return: 0 on success, <0 on failue

\*/

int sblock\_register\_notifier(uint8\_t dst, uint8\_t channel,

void (\*handler)(int event, void \*data), void \*data);

/\*\*

\* sblock\_get -- get a free sblock for sender

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @blk: return a gotten sblock pointer

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int sblock\_get(uint8\_t dst, uint8\_t channel, struct sblock \*blk, int timeout);

/\*\*

\* sblock\_send -- send a sblock, it should be from sblock\_get

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @blk: the sblock to be sent

\* @return: 0 on success, <0 on failue

\*/

int sblock\_send(uint8\_t dst, uint8\_t channel, struct sblock \*blk);

/\*\*

\* sblock\_receive -- receive a sblock, it should be released after it's handled

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @blk: return a received sblock pointer

\* @timeout: milliseconds, 0 means no wait, -1 means unlimited

\* @return: 0 on success, <0 on failue

\*/

int sblock\_receive(uint8\_t dst, uint8\_t channel, struct sblock \*blk, int timeout);

/\*\*

\* sblock\_release -- release a sblock from reveiver

\*

\* @dst: dest processor ID

\* @channel: channel ID

\* @return: 0 on success, <0 on failue

\*/

int sblock\_release(uint8\_t dst, uint8\_t channel, struct sblock \*blk);

## User Modules on SBUF/SBLOCK

In VLX system, we have the VLX modules for AP/CP communications, such as VBPIPE, VCONSOLE, VETH, VAUDIO, etc. To be compatible and re-use the old system modules, we’ll simulate the interfaces in native AP/CP system. SBUF/SBLOCK export general interfaces to provides basic data communication between AP and CP. We’ll implement the SPIPE, STTY, SETH, SAUDIO modules based on these interfaces. So that we can nearly keep the android HAL system consistent.

### SPIPE - VBPIPE



VBPIPE is a general pipe interfaces between Guest OSes in VLX system. In native AP/CP system, we implement a SPIPE to export pipe interfaces between AP and CP. The description and mapping of these interfaces for TD modem is as below,

|  |  |  |
| --- | --- | --- |
| SIPC | VLX | Comments |
| /dev/slog\_td | /dev/vbpipe0 | Log for DIAG |
| /dev/spipe\_td0 | N/A | spipe\_td0 has loopback function for unit test |
| /dev/spipe\_td1 | /dev/vbpipe1 | NVitem data communications |
| /dev/spipe\_td2 | /dev/vbpipe2 | Modem assert notification |
| /dev/spipe\_td3 | /dev/vbpipe3 | Charger parameter communications |
| /dev/spipe\_td4 | /dev/vbpipe4 | Not used |
| /dev/spipe\_td5 | /dev/vbpipe5 | Not used |
| /dev/spipe\_td6 | /dev/vbpipe6 | Audio VBC switch notification |
| /dev/spipe\_td7 | /dev/vbpipe7 | Not used |

In AP linux kernel, spipe is a general char device driver that exports read/write/poll/ioctl interfaces. It can easily have multiple instances to create sbuf in different SMSG channels.

### STTY - VCONSOLE

VONSOLE is serial-style interface to build cmux 0710 protocol for modem communication. Since AP/CP system uses the shared memory as data communication that guarantees no data error or losing, we can simply use spipe to simulate the mux interfaces. AP phone RIL and CP ATC modules can send/receive AT commands on STTY interfaces that are multiple bi-directional FIFOs.

### SETH - VETH

In VLX system, VETH is a virtual Ethernet driver based on VLINK. In native AP/CP system, a virtual SETH driver is used for AP/CP 3G data communication.



### SAUDIO - VAUDIO

In VLX system, we use VAUDIO device to do recording in call. In native AP/CP system, we construct SAUDIO device as a new Sound Card to do playback/record in call mode. Because SPRD platform has only one HW Codec path that can accept one route at a time. The codec should be switched to modem DSP in call mode. In this mode, if AP side requires playback or record, DSP should help on it. To make up-layer applications handle this transparently, a virtual Sound Card is created. In normal mode, playback/record request is sent to the real sound device. In call mode, playback/record request is sent to saudio device. Just as we have two HW codec device, the applications switch to the codec that requires. When AP dials out or receives a call, it will listen and wait for CP request. If it’s confirmed, AP will switch codec owner to CP and use saudio for further audio operations. We reserved 2 channels for SAUDIO module.



### SPTY – UART

In native AP/CP system, we construct SPTY device as a UART device. we construct SPTY device as bt AP/CP Command and data transmit channel



## SRPC

Spreadtrum RPC interfaces are designed to make it available to do Remote-Procedure-Call between AP and CP.

The basic activity flowchart in AP side is as below. There’s a srpc\_thread in kernel to maintain SRPC in/out. Each user should register a thread stub to get callback.



***TODO...***

# Modem Functions on SIPC

## Overview

In Shark system, there’re 2 individual modems. SPRD will implement the DSDA system. The basic module diagram is as below.

Each modem has an individual SIPC instance and communicates with AP system. In AP side, kernel creates CPROC(Communication Processor driver) and SIPC instances for each modem. CPROC is a device driver corresponding to a modem, which export interfaces to modem control, such as stop, start, etc. The communication services are based on SIPC channels.



## Dual-Sim-Dual-Active

To support DSDA, TD and WCDMA modems both keep active at the same time. Two modems have individual CPROC and SIPC interfaces in AP. This makes it possible to create individual services for modem communications.

For CPROC driver, the device node lists are,

|  |  |  |  |
| --- | --- | --- | --- |
| **NODE** | **TD** | **WCDMA** | **Comments** |
| DEVICE | /dev/cpt | /dev/cpw | Modem device node for ioctl and mmap |
| SEG-CP | /proc/cpt/modem | /proc/cpw/modem | Download modem image |
| SEG-DSP | /proc/cpt/dsp | /proc/cpw/dsp | Download dsp image |
| MEM | /proc/cpt/mem | /proc/cpw/mem | CP memory dump |
| STATE | /proc/cpt/status | /proc/cpw/status | CP status: normal, stopped, wdtirq |
| STOP | /proc/cpt/stop | /proc/cpw/stop | Make CP stopped and ready to start |
| START | /proc/cpt/start | /proc/cpw/start | Trigger CP restart |

The SIPC interfaces in user-land are,

|  |  |  |  |
| --- | --- | --- | --- |
| **MODULE** | **TD** | **WCDMA** | **Comments** |
| RIL | /dev/stty\_td[0~31] | /dev/stty\_w[0~31] | RIL communication fifo |
| DATA | seth\_td[0~2] | seth\_w[0~2] | 2G/3G data ethernet |
| SOUND | /dev/snd/..[Cx] | /dev/snd/..[Cx] | Sound card of play/record in call mode |
| DIAG | /dev/slog\_td | /dev/slog\_w | Diag and Log communication |
| LOOP | /dev/spipe\_td0 | /dev/spipe\_w0 | Loopback pipe |
| NV | /dev/spipe\_td1 | /dev/spipe\_w1 | NV item data communications |
| ASSERT | /dev/spipe\_td2 | /dev/spipe\_w2 | Modem assert notification and feedback |
| N/A | /dev/spipe\_td3 | /dev/spipe\_w3 | Not used |
| N/A | /dev/spipe\_td4 | /dev/spipe\_w4 | Not used |
| N/A | /dev/spipe\_td5 | /dev/spipe\_w5 | Not used |
| AUDIOC | /dev/spipe\_td6 | /dev/spipe\_w6 | Audio VBC switch notification |
| N/A | /dev/spipe\_td7 | /dev/spipe\_w7 | Not used |

To distinguish DSDA, DSDS and single SIM configuration, we set different properties in different products. So that the services(such as rild, phoneserver, modemd) can get exact modem information. Each modem has a separate rild and phoneserver to build independent telephony service for android up-level applications. Some general services(such as modemd, nvitemd, engpcclient) has 2 threads to manage the 2 modems.

A typical T+W 2-SIMs may have the following properties in product,

# total sim count

ro.msms.phone\_count=2

persist.msms.phone\_default=0

# modem interfaces for TD modem

ro.modem.t.enable=1

ro.modem.t.dev=/dev/cproc\_td

ro.modem.t.tty=/dev/stty\_td

ro.modem.t.eth=seth\_td

ro.modem.t.snd=1

ro.modem.t.diag=/dev/slog\_td

ro.modem.t.loop=/dev/spipe\_td0

ro.modem.t.nv=/dev/spipe\_td1

ro.modem.t.assert=/dev/spipe\_td2

ro.modem.t.vbc=/dev/spipe\_td6

ro.modem.t.id=0

ro.modem.t.count=1

# modem interfaces for W modem

ro.modem.w.enable=1

ro.modem.w.dev=/dev/cproc\_w

ro.modem.w.tty=/dev/stty\_w

ro.modem.w.eth=seth\_w

ro.modem.w.snd=2

ro.modem.w.diag=/dev/slog\_w

ro.modem.w.loop=/dev/spipe\_w0

ro.modem.w.nv=/dev/spipe\_w1

ro.modem.w.assert=/dev/spipe\_w2

ro.modem.w.vbc=/dev/spipe\_w6

ro.modem.w.id=1

ro.modem.w.count=1

A TD 2-SIM system may have the following properties,

# total sim count

ro.msms.phone\_count=2

persist.msms.phone\_default=0

# modem interfaces for VLX TD modem

ro.modem.t.enable=1

ro.modem.t.dev=/dev/cproc\_td

ro.modem.t.tty=/dev/stty\_td

ro.modem.t.eth=seth\_td

ro.modem.t.snd=1

ro.modem.t.diag=/dev/slog\_td

ro.modem.t.loop=/dev/spipe\_td0

ro.modem.t.nv=/dev/spipe\_td1

ro.modem.t.assert=/dev/spipe\_td2

ro.modem.t.vbc=/dev/spipe\_td6

ro.modem.t.id=0

ro.modem.t.count=2

A W-only 1-SIM system may have the following properties,

# total sim count

ro.msms.phone\_count=1

persist.msms.phone\_default=0

# modem interfaces for W modem

ro.modem.w.enable=1

ro.modem.w.dev=/dev/cproc\_w

ro.modem.w.tty=/dev/stty\_w

ro.modem.w.eth=seth\_w

ro.modem.w.snd=2

ro.modem.w.diag=/dev/slog\_w

ro.modem.w.loop=/dev/spipe\_w0

ro.modem.w.nv=/dev/spipe\_w1

ro.modem.w.assert=/dev/spipe\_w2

ro.modem.w.vbc=/dev/spipe\_w6

ro.modem.t.id=0

ro.modem.t.count=1

The VLX 2-SIM system can be compatible to have the following proprieties,

# total sim count

ro.msms.phone\_count=2

persist.msms.phone\_default=0

# modem interfaces for VLX TD modem

ro.modem.t.enable=1

ro.modem.t.dev=

ro.modem.t.tty=/dev/ts0710mux

ro.modem.t.eth=veth

ro.modem.t.snd=1

ro.modem.t.diag=/dev/vbpipe0

ro.modem.t.loop=

ro.modem.t.nv=/dev/vbpipe1

ro.modem.t.assert=/dev/vbpipe2

ro.modem.t.vbc=/dev/vbpipe6

ro.modem.t.id=0

ro.modem.t.count=2

## Modem Recovery

The AP/CP system regards AP as major host server and CP as slave client in SIPC communication. Once AP crashed, the whole system should be rebooted. If CP crashed, it can be easily recovered.

When CP watchdog barks or assert message is issued, the AP system will stop phone-related service to stop SIPC messages. No more request for SIPC in AP side. Even if AP applications continue to send data to SIPC, they’ll be stored in SIPC buffer before it’s blocked because CP can’t consume them. AP can start to download the CP images and reset CP after some diagnose. The CP boot with SIPC initialization will not be stopped since AP lives as the host server. Note that AP should response all OPEN message on open channel in channel thread. This is why we have asymmetric design in SBUF/SBLOCK creation. To eliminate the garbage content, SBUF/SBLOCK should reset the R/W pointers when it gets OPEN smsg.



In modemd, here’s the major procedure to handle modem recovery,

POLL on /dev/spipe\_td2 AND /proc/cpt/wdtirq

IF read(/dev/spipe\_td2) == “Assert” THEN

send Assert to socket (debug version)

wait on Assert Handler (debug version)

stop telephony services

write(/dev/spipe\_td2, “^Z”)

read(/dev/spipe\_td2) == “OK”

CALL reset\_modem

start telephony services

END

IF read(/proc/cpt/wdtirq) == “wdtirq” THEN

dump MEM from /proc/cpt/mem (debug version)

stop telephony services

CALL reset\_modem

start telephony services

END

PROCEDURE reset\_modem

write(/proc/cpt/stop, “1”)

write(/proc/cpt/modem, MODEM\_IMG)

write(/proc/cpt/dsp, DSP\_IMG)

write(/proc/cpt/start, “1”)

END

## WCN Interface

The AP/CP system regards AP as major host server and CP as slave client in SIPC communication. Once AP crashed, the whole system should be rebooted. If CP crashed, it can be easily recovered.

|  |  |  |
| --- | --- | --- |
| **NODE** | **WCN** | **Comments** |
| DEVICE | /dev/cpwcn | Modem device node for ioctl and mmap |
| SEG-CP | /proc/cpwcn/modem | Download modem image |
| MEM | /proc/ cpwcn /mem | CP memory dump |
| STATE | /proc/ cpwcn /status | CP status: normal, stopped, wdtirq |
| STOP | /proc/ cpwcn /stop | Make CP stopped and ready to start |
| START | /proc/ cpwcn /start | Trigger CP restart |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **MODULE** | **WCN** | **Comments** |
| DATA | seth\_wcn[0~2] | Wifi data ethernet |
| DIAG | /dev/slog\_wcn | Diag and Log communication |
| LOOP | /dev/spipe\_wcn0 | Loopback pipe |
| N/A | /dev/spipe\_wcn1 | Not used |
| ASSERT | /dev/spipe\_wcn2 | Modem assert notification and feedback |
| N/A | /dev/spipe\_wcn3 | Not used |
| N/A | /dev/spipe\_wcn4 | Not used |
| N/A | /dev/spipe\_wcn5 | Not used |
| N/A | /dev/spipe\_wcn6 | Not used |
| WIFI | /dev/spipe\_wcncn7 | Wifi Command data transmit channel |
| WIFI | /dev/spipe\_wcncn8 | Wifi Command data transmit channel |
| WIFI | /dev/spipe\_wcncn9 | Wifi Command data transmit channel |
| BT | /dev/spipe\_wcncn10 | BT Command data transmit channel |
| BT | /dev/spipe\_wcncn11 | BT Command data transmit channel |
| N/A | /dev/spipe\_wcncn12 | Not used |

The SIPC interfaces in user-land are,

## Modem Calibration

### Calibration on Bootloader

**Not Supported**.

### Calibration on Linux Boot

In VLX system, Modem Calibration is done after Linux kernel is booted. CP takes over USB and communicates with PC directly. VBPIPE is used for AP/CP data communication.



To make the system extendible in future system, the communication with PC is done by AP now. Each CP modem has independent spipe to talk with AP and then to PC host.



Theoretically, software system can support multiple modems to do calibration at the same time if the hardware environment is available. And the Calibration mode can be implemented in recovery image. For the sake of PC tool and factory environment, we suppose to do single mode calibration on time for the dual-mode phone.



The calibration flow is as above,

1) PC tool initially sends calibration request and mode to the phone in the early boot stage. The U-Boot of phone detects whether system can enter calibration and the modem mode.

2) If calibration request is detected, U-Boot loads and boots the corresponding modem system into calibration. U-boot sends calibration parameters to kernel boot cmdline and boots to recovery image.

3) In Android recovery image, a bridge between usb/uart and spipe according to the calibration parameters is built to connect PC Tool and the selected Modem. NV data is sent to AP side to store to the storage.

4) In the calibration mode, PC should enumerate USB/UART twice. It sends calibration request in the first time to make phone enter calibration mode. And then it builds connection to modem by the bridge.