# 启动

#include <hfp.h>

#include <a2dp.h>

#include <avrcp.h>

#include <connection.h>

#include <panic.h>

#include <pio.h>

#include <ps.h>

#include <string.h>

#include <boot.h>

#include <os.h>

#include <logging.h>

#include <app/message/system\_message.h>

#include "av\_headset.h"

#include "av\_headset\_auth.h"

#include "av\_headset\_ui.h"

#include "av\_headset\_pairing.h"

#include "av\_headset\_scan\_manager.h"

#include "av\_headset\_anc\_tuning.h"

#include "sw\_watchdog.h"

/\*! Application data structure \*/

appTaskData globalApp;

/\*! \brief Handle subsystem event report. \*/

static void appHandleSubsystemEventReport(MessageSubsystemEventReport \*evt)

{

UNUSED(evt);

// DEBUG\_LOGF("appHandleSubsystemEventReport, ss\_id=%d, level=%d, id=%d, cpu=%d, occurrences=%d, time=%d",

// evt->ss\_id, evt->level, evt->id, evt->cpu, evt->occurrences, evt->time);

}

/\*! \brief System Message Handler

This function is the message handler for system messages. They are

routed to existing handlers. If execution reaches the end of the

function then it is assumed that the message is unhandled.

\*/

static void appHandleSystemMessage(Task task, MessageId id, Message message)

{

bool handled = FALSE;

UNUSED(task);

UNUSED(message);

switch (id)

{

case MESSAGE\_SUBSYSTEM\_EVENT\_REPORT:

appHandleSubsystemEventReport((MessageSubsystemEventReport \*)message);

return;

case MESSAGE\_IMAGE\_UPGRADE\_ERASE\_STATUS:

case MESSAGE\_IMAGE\_UPGRADE\_COPY\_STATUS:

case MESSAGE\_IMAGE\_UPGRADE\_AUDIO\_STATUS:

case MESSAGE\_IMAGE\_UPGRADE\_HASH\_ALL\_SECTIONS\_UPDATE\_STATUS:

handled = appUpgradeHandleSystemMessages(id, message, FALSE);

break;

case MESSAGE\_USB\_ENUMERATED:

appAncTuningEnumerated();

return;

case MESSAGE\_USB\_SUSPENDED:

appAncTuningSuspended();

return;

default:

break;

}

if (!handled)

{

appHandleSysUnexpected(id);

}

}

/\* Handler for the INIT\_CFM message.

Used to register the handler that decides whether to allow entry

to low power mode, before passing the #INIT\_CFM message to

the state machine handler.

\param message The INIT\_CFM message received (if any).

\*/

static void appHandleInitCfm(Message message)

{

appSmHandleMessage(appGetSmTask(), INIT\_CFM, message);

#ifdef CONFIG\_STAROT

MessageSend(appGetUiTask(), INIT\_CFM, NULL);

#endif

}

/\*! \brief Message Handler

This function is the main message handler for the main application task, every

message is handled in it's own seperate handler function. The switch

statement is broken into seperate blocks to reduce code size, if execution

reaches the end of the function then it is assumed that the message is

unhandled.

\*/

static void appHandleMessage(Task task, MessageId id, Message message)

{

UNUSED(task);

if (id == appInitWaitingForMessageId())

{

appInitHandleMessage(id, message);

return;

}

/\* Handle Connection Library messages that are not sent directly to

the requestor \*/

if (CL\_MESSAGE\_BASE <= id && id < CL\_MESSAGE\_TOP)

{

bool handled = FALSE;

/\* Pass connection library messages in turn to the modules that

are interested in them.

\*/

handled |= appPairingHandleConnectionLibraryMessages(id, message, handled);

handled |= appConManagerHandleConnectionLibraryMessages(id, message, handled);

handled |= appLinkPolicyHandleConnectionLibraryMessages(id, message, handled);

handled |= appAuthHandleConnectionLibraryMessages(id, message, handled);

handled |= appAdvManagerHandleConnectionLibraryMessages(id, message, handled);

handled |= appGattHandleConnectionLibraryMessages(id, message, handled);

handled |= appTestHandleConnectionLibraryMessages(id, message, handled);

if (handled)

{

return;

}

}

switch (id)

{

/\* AV messages \*/

case AV\_CREATE\_IND:

case AV\_DESTROY\_IND:

return;

case INIT\_CFM:

appHandleInitCfm(message);

return;

case AV\_SYS\_RST\_WATCH\_DOG:

watchdogReset();

return;

}

appHandleUnexpected(id);

}

extern void \_init(void);

void \_init(void)

{

appAncTuningEarlyInit();

}

/\*! \brief Application entry point

This function is the entry point for the application, it performs basic

initialisation of the application task state machine and then sets

the state to 'initialising' which will start the initialisation procedure.

\returns Nothing. Only exits by powering down.

\*/

int main(void)

{

OsInit();

/\* Set up task handlers \*/

appGetApp()->task.handler = appHandleMessage;

appGetApp()->systask.handler = appHandleSystemMessage;

MessageSystemTask(appGetSysTask());

/\* Start the application module and library initialisation sequence \*/

appInit();

watchdogReset();

/\* Start the message scheduler loop \*/

MessageLoop();

/\* We should never get here, keep compiler happy \*/

return 0;

}

/\* Define pmalloc pools. This definition will be merged at run time with the

base definitions from Hydra OS - see 'pmalloc\_config\_P1.h'. \*/

#include <pmalloc.h>

\_Pragma ("unitsuppress Unused")

\_Pragma ("datasection apppool")

static const pmalloc\_pool\_config app\_pools[] =

{

{ 4, 14 },

{ 8, 25 },

{ 12, 17 },

{ 16, 7 },

{ 20, 22 },

{ 24, 15 },

{ 28, 55 },

{ 32, 21 },

{ 36, 19 },

{ 40, 10 },

{ 56, 9 },

{ 64, 7 },

{ 80, 9 },

{ 120, 16 },

{ 140, 4 },

{ 180, 6 },

{ 220, 3 },

{ 288, 1 },

{ 512, 1 },

{ 692, 3 }

};

# 手势设置、佩戴检测设置、唤醒助手设置

//

// Created by Administrator on 2019/11/11.

//

#ifndef EARBUDS1\_PEER\_H

#define EARBUDS1\_PEER\_H

#include "av\_headset\_av.h"

////////////////////////////////////////////////////////////////////////////////

/////////////// 左右耳机数据命令

////////////////////////////////////////////////////////////////////////////////

typedef struct {

Task client\_task; /\*!< Task to receive any response \*/

uint8 command;

uint8 data[7];

} PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T;

#define AVRCP\_PEER\_CMD\_TXDATA 0x69

#define AVRCP\_PEER\_CMD\_TXDATA\_SIZE (7) // command + data

enum {PEERTX\_CMD\_SYNCGAIA=0, /\* 主副耳机发送gaia状态 [payload[0], 0:disconnect 1:connect \*/

PEERTX\_CMD\_WAKEUP\_SYS, /\* 副耳机发送系统唤醒命令 [playload[0]: 0:keywakeup, 1:voice wakeup \*/

PEERTX\_CMD\_WAKEUP\_APP, /\* 副耳机发送唤醒APP命令 [playload[0]: 0:keywakeup, 1:voice wakeup \*/

};

void appPeerSigTxDataCommand(Task task, const bdaddr \*peer\_addr, uint8 command, uint16 size\_payload, const uint8 \*payload);

void appPeerSigTxDataCommandUi(uint8 command, uint8 payload); // task为UI 仅一个字节payhload

#define appPeerSigTxWakeupSys(apo) appPeerSigTxDataCommandUi(PEERTX\_CMD\_WAKEUP\_SYS, (apo))

#define appPeerSigTxWakeupApp(apo) appPeerSigTxDataCommandUi(PEERTX\_CMD\_WAKEUP\_APP, (apo))

void appPeerSigTxDataRequest(PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T \*req); // 发送方： 请求发送给对方

bool appPeerSigRxDataCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind); // 接收方： 接收数据处理

void appPeerSigTxDataConfirm(Task task, peerSigStatus status); // 发送方： 获取对方返回值

bool appUiRecvPeerCommand(PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T \*req); // 接收方： 返回给上层处理

////////////////////////////////////ble配对信息////////////////////////////////////////////

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Peer Signalling Message Definitions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define AVRCP\_PEER\_CMD\_BLE\_CONFIG 0x70

#define AVRCP\_PEER\_CMD\_BLE\_CONFIG\_SIZE 8 /// advCode + bondCode

typedef struct

{

Task client\_task; /\*!< Task to receive any response \*/

int advCode;

int bondCode;

} PEER\_SIG\_INTERNAL\_BLE\_CONFIG\_REQ\_T;

void appPeerSigTxBleConfigRequest(Task task, const bdaddr \*peer\_addr, int advCode, int bondCode);

void appPeerSigHandleInternalBleConfigRequest(PEER\_SIG\_INTERNAL\_BLE\_CONFIG\_REQ\_T \*req);

bool appPeerSigHandleBleConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind);

void appPeerSigMsgBleConfigConfirmation(Task task, peerSigStatus status);

////////////////////////////////////双击配置项////////////////////////////////////////////

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Peer Signalling Message Definitions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG 0x71

#define AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG\_SIZE 10 /// 左+右+软硬件版本号

typedef struct

{

Task client\_task; /\*!< Task to receive any response \*/

uint8 left;

uint8 right;

uint16 reserve;

uint8 peerver[DEV\_HWSWVER\_LEN];

} PEER\_SIG\_INTERNAL\_DOBULE\_CLICK\_CONFIG\_REQ\_T;

void appPeerSigTxDoubleClickConfigRequest(Task task, const bdaddr \*peer\_addr, uint8 left, uint8 right);

void appPeerSigHandleInternalDoubleClickConfigRequest(PEER\_SIG\_INTERNAL\_DOBULE\_CLICK\_CONFIG\_REQ\_T \*req);

bool appPeerSigHandleDoubleClickConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind);

void appPeerSigMsgDoubleClickConfigConfirmation(Task task, peerSigStatus status);

/////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////通用配置项////////////////////////////////////////////

#define AVRCP\_PEER\_CMD\_NORMAL\_CONFIG\_SIZE 12

typedef struct

{

Task client\_task; /\*!< Task to receive any response \*/

uint8 apollo\_config;

uint8 wear\_config;

uint8 assistant\_type;

uint8 dontused;

uint32 apollo\_timestamp;

uint32 wear\_timestamp;

} PEER\_SIG\_INTERNAL\_NORMAL\_CONFIG\_REQ\_T;

#define AVRCP\_PEER\_CMD\_NORMAL\_CONFIG 0x72

void appPeerSigTxNormalConfigRequest(Task task, const bdaddr \*peer\_addr);

void appPeerSigHandleInternalNormalConfigRequest(PEER\_SIG\_INTERNAL\_NORMAL\_CONFIG\_REQ\_T \*req);

bool appPeerSigHandleNormalConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind);

void appPeerSigMsgNormalConfigConfirmation(Task task, peerSigStatus status);

/////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////双击消息项////////////////////////////////////////////

#define AVRCP\_PEER\_DOUBLE\_CLICK\_WAKEUP 0x73

#define AVRCP\_PEER\_DOUBLE\_CLICK\_WAKEUP\_SYSTEM 0x74

//#define AVRCP\_PEER\_DOUBLE\_CLICK\_WAKEUP\_SIZE 10

typedef struct

{

Task client\_task; /\*!< Task to receive any response \*/

} PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T;

void appPeerSigTxDoubleClickWakeupRequest(Task task, const bdaddr \*peer\_addr, int num);

void appPeerSigHandleInternalDoubleClickWakeupRequest(PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T \*req);

void appPeerSigHandleInternalDoubleClickWakeupSystemRequest(PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T \*req);

bool appPeerSigHandleDoubleClickWakeupCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind);

bool appPeerSigHandleDoubleClickWakeupSystemCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind);

void appPeerSigMsgDoubleClickWakeupConfirmation(Task task, peerSigStatus status);

/////////////////////////////////////////////////////////////////////////////////////////

#endif //EARBUDS1\_PEER\_H

#include "av\_headset.h"

#include "peer.h"

#include "av\_headset\_gaia\_starot.h"

#define STAROT\_MAKE\_SIZE(TYPE) (((sizeof(TYPE)) / 8 \* 8) + ((sizeof(TYPE)) % 8 > 0 ? 8 : 0))

#define STAROT\_MAKE\_MESSAGE(TYPE) TYPE \*message = (TYPE \*) PanicUnlessMalloc(STAROT\_MAKE\_SIZE(TYPE))

/// 将消息发送至对方耳机

void appPeerSigTxDataCommand(Task task, const bdaddr \*peer\_addr, uint8 command, uint16 size\_payload, const uint8 \*payload) {

if (NULL == peer\_addr) {

return;

}

peerSigTaskData \*peer\_sig = appGetPeerSig();

STAROT\_MAKE\_MESSAGE(PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T);

message->client\_task = task;

message->command = command;

if(size\_payload > 0) {

if(size\_payload > (AVRCP\_PEER\_CMD\_TXDATA\_SIZE-1))

size\_payload = AVRCP\_PEER\_CMD\_TXDATA\_SIZE-1;

memcpy(message->data, payload, size\_payload);

}

MessageSendConditionally(&peer\_sig->task, PEER\_SIG\_INTERNAL\_TXDATA\_REQ, message, appPeerSigStartup(peer\_addr));

}

void appPeerSigTxDataCommandUi(uint8 command, uint8 payload) { // 仅一个字节payhload

bdaddr peer\_addr;

if(FALSE == appDeviceGetPeerBdAddr(&peer\_addr))

return;

appPeerSigTxDataCommand(appGetUiTask(), &peer\_addr, command, 1, &payload);

}

bool appUiRecvPeerCommand(PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T \*req) { // 接收方： 返回给上层处理

bool ret = TRUE;

ProgRIPtr progRun = appSubGetProgRun();

switch(req->command) {

case PEERTX\_CMD\_SYNCGAIA:

progRun->peerGaiaStat = (req->data[0] == 1) ? 1 : 0;

break;

case PEERTX\_CMD\_WAKEUP\_SYS: /\* 副耳机发送系统唤醒命令 [playload[0]: 0:keywakeup, 1:voice wakeup \*/

MessageSend(appGetUiTask(), (req->data[0] == 0) ? APP\_ASSISTANT\_TAP\_SYSTEM : APP\_ASSISTANT\_SYSTEM, 0);

break;

case PEERTX\_CMD\_WAKEUP\_APP:

MessageSend(appGetUiTask(), (req->data[0] == 0) ? APP\_ASSISTANT\_TAP\_AWAKEN : APP\_ASSISTANT\_AWAKEN, 0);

break;

default:

DEBUG\_LOG("Unknown command:%d", req->command);

ret = FALSE;

break;

}

return ret;

}

///

void appPeerSigTxDataRequest(PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigTxDataRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED:

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_CMD\_TXDATA,

AVRCP\_PEER\_CMD\_TXDATA\_SIZE, &req->command);

break;

default:

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

break;

}

}

/\*! \brief Receive connect handset command. \*/

bool appPeerSigRxDataCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

peerSigTaskData \*peer\_sig = appGetPeerSig();

DEBUG\_LOG("appPeerSigRxDataCommand");

/\* validate message \*/

if ((ind->size\_payload != AVRCP\_PEER\_CMD\_TXDATA\_SIZE)

|| !peer\_sig->rx\_handset\_commands\_task) {

return FALSE;

}

else {

PEER\_SIG\_INTERNAL\_TXDATA\_REQ\_T peerReq;

memcpy(&peerReq.command, ind->payload, AVRCP\_PEER\_CMD\_TXDATA\_SIZE);

return appUiRecvPeerCommand(&peerReq);

}

}

void appPeerSigTxDataConfirm(Task task, peerSigStatus status) {

UNUSED(task), UNUSED(status);

DEBUG\_LOG("appPeerSigTxDataConfirm:%d", status);

}

//////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////////

void appPeerSigTxBleConfigRequest(Task task, const bdaddr \*peer\_addr, int advCode, int bondCode) {

if (NULL == peer\_addr) {

return;

}

#ifdef CONFIG\_STAROT\_PEERPAIR

if(ParamUsingSingle()) { // 检测是否为独立使用

appBleSetSync(TRUE);

return;

}

#endif

peerSigTaskData \*peer\_sig = appGetPeerSig();

STAROT\_MAKE\_MESSAGE(PEER\_SIG\_INTERNAL\_BLE\_CONFIG\_REQ\_T);

message->client\_task = task;

message->advCode = advCode;

message->bondCode = bondCode;

MessageSendConditionally(&peer\_sig->task, PEER\_SIG\_INTERNAL\_BLE\_CONFIG\_REQ, message, appPeerSigStartup(peer\_addr));

}

/// 将消息发送至对方耳机

void appPeerSigHandleInternalBleConfigRequest(PEER\_SIG\_INTERNAL\_BLE\_CONFIG\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigHandleInternalBleConfigRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED: {

uint8 message[AVRCP\_PEER\_CMD\_BLE\_CONFIG\_SIZE];

\*((int \*) (message + 0)) = req->advCode;

\*((int \*) (message + 4)) = req->bondCode;

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_CMD\_BLE\_CONFIG,

AVRCP\_PEER\_CMD\_BLE\_CONFIG\_SIZE, message);

}

break;

default: {

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

}

break;

}

}

/\*! \brief Receive connect handset command. \*/

bool appPeerSigHandleBleConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

peerSigTaskData \*peer\_sig = appGetPeerSig();

DEBUG\_LOG("call appPeerSigHandleBleConfigCommand");

/\* validate message \*/

if ((ind->size\_payload != AVRCP\_PEER\_CMD\_BLE\_CONFIG\_SIZE)

|| !peer\_sig->rx\_handset\_commands\_task) {

return FALSE;

} else {

int \*data = (int \*) (ind->payload);

int advCode = data[0];

int bondCode = data[1];

/// 是直接存储，还是调用ui

appBleSetBond(advCode, bondCode);

appBleSetSync(TRUE);

return TRUE;

}

}

void appPeerSigMsgBleConfigConfirmation(Task task, peerSigStatus status) {

/// todo : 发送消息至指定task，告知同步情况

UNUSED(task), UNUSED(status);

DEBUG\_LOG("call appPeerSigMsgBleConfigConfirmation:%d", status);

if (peerSigStatusSuccess == status) {

appBleSetSync(TRUE);

}

}

void appPeerSigTxDoubleClickConfigRequest(Task task, const bdaddr \*peer\_addr, uint8 left, uint8 right) {

uint8 ver[8];

peerSigTaskData \*peer\_sig = appGetPeerSig();

STAROT\_MAKE\_MESSAGE(PEER\_SIG\_INTERNAL\_DOBULE\_CLICK\_CONFIG\_REQ\_T);

message->client\_task = task;

if(0xFF != left)

message->left = left;

if(0xFF != right)

message->right = right;

SystemGetCurrentVersion((uint8\*)ver);

memcpy(message->peerver, ver, DEV\_HWSWVER\_LEN);

MessageSendConditionally(&peer\_sig->task, PEER\_SIG\_INTERNAL\_DOUBLE\_CLICK\_SETTING\_REQ, message, appPeerSigStartup(peer\_addr));

}

void appPeerSigHandleInternalDoubleClickConfigRequest(PEER\_SIG\_INTERNAL\_DOBULE\_CLICK\_CONFIG\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigHandleInternalBleConfigRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED: {

uint8 message[AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG\_SIZE];

/\* Build data for message \*/

message[0] = req->left;

message[1] = req->right;

memcpy(&message[2], req->peerver, DEV\_HWSWVER\_LEN);

/\* Send the handset address over AVRCP \*/

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG,

AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG\_SIZE, message);

}

break;

default: {

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

}

break;

}

}

bool appPeerSigHandleDoubleClickConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

peerSigTaskData \*peer\_sig = appGetPeerSig();

/\* validate message \*/

if ((ind->size\_payload != AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG\_SIZE) || !peer\_sig->rx\_handset\_commands\_task) {

return FALSE;

} else {

// MAKE\_MESSAGE(PEER\_SIG\_CONNECT\_HANDSET\_IND);

// message->play\_media = !!(ind->payload[0] & AVRCP\_PEER\_CMD\_CONNECT\_HANDSET\_FLAG\_PLAY\_MEDIA);

// /\* tell client to connect to handset \*/

// MessageSend(peer\_sig->rx\_handset\_commands\_task, PEER\_SIG\_CONNECT\_HANDSET\_IND, message);

// DEBUG\_LOG("appPeerSigHandleConnectHandsetCommand %d", message->play\_media);

uint8 left = ind->payload[0];

uint8 right = ind->payload[1];

/// 是直接存储，还是调用ui

memcpy(gBtAddrParam.peerVer, &ind->payload[2], DEV\_HWSWVER\_LEN);

ParamSaveBtAddrPrm(&gBtAddrParam);

UserSetKeyFunc(left, right);

return TRUE;

}

}

void appPeerSigMsgDoubleClickConfigConfirmation(Task task, peerSigStatus status) {

/// todo : 发送消息至指定task，告知同步情况

UNUSED(task), UNUSED(status);

// MAKE\_MESSAGE(PEER\_SIG\_CONNECT\_HANDSET\_CFM);

// message->status = status;

// MessageSend(task, PEER\_SIG\_CONNECT\_HANDSET\_CFM, message);

}

extern UserParam gUserParam;

void appPeerSigTxNormalConfigRequest(Task task, const bdaddr \*peer\_addr) {

if (NULL == peer\_addr) {

return;

}

#ifdef CONFIG\_STAROT\_PEERPAIR

if(ParamUsingSingle()) { // 检测是否为独立使用

/// todo 设置已经同步

return;

}

#endif

peerSigTaskData \*peer\_sig = appGetPeerSig();

STAROT\_MAKE\_MESSAGE(PEER\_SIG\_INTERNAL\_NORMAL\_CONFIG\_REQ\_T);

message->client\_task = task;

/// 获取对应的值，并填写

message->apollo\_config = (gUserParam.apolloEnable > 0 ? 1 : 0);

message->assistant\_type = gUserParam.assistantType;

message->apollo\_timestamp = gUserParam.assistantModifyTime;

message->wear\_config = (gUserParam.sensorEnable > 0 ? 1 : 0);

message->wear\_timestamp = gUserParam.sensorModifyTime;

MessageSendConditionally(&peer\_sig->task, PEER\_SIG\_INTERNAL\_NORMAL\_SETTING\_REQ, message, appPeerSigStartup(peer\_addr));

}

void appPeerSigHandleInternalNormalConfigRequest(PEER\_SIG\_INTERNAL\_NORMAL\_CONFIG\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigHandleInternalNormalConfigRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED: {

uint8 message[AVRCP\_PEER\_CMD\_NORMAL\_CONFIG\_SIZE];

uint8 pos = 0;

\*((uint8\*) (message + pos)) = req->apollo\_config;

pos += sizeof(uint8);

\*((uint8\*) (message + pos)) = req->assistant\_type;

pos += sizeof(uint8);

\*((uint8\*) (message + pos)) = req->wear\_config;

pos += sizeof(uint8);

/// unused

\*((uint8\*) (message + pos)) = 0;

pos += sizeof(uint8);

memcpy(message + pos, &req->apollo\_timestamp, sizeof(uint32));

pos += sizeof(uint32);

memcpy(message + pos, &req->wear\_timestamp, sizeof(uint32));

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_CMD\_NORMAL\_CONFIG,

AVRCP\_PEER\_CMD\_NORMAL\_CONFIG\_SIZE, message);

}

break;

default: {

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

}

break;

}

}

bool appPeerSigHandleNormalConfigCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

peerSigTaskData \*peer\_sig = appGetPeerSig();

DEBUG\_LOG("call appPeerSigHandleNormalConfigCommand");

/\* validate message \*/

if ((ind->size\_payload != AVRCP\_PEER\_CMD\_NORMAL\_CONFIG\_SIZE)

|| !peer\_sig->rx\_handset\_commands\_task) {

return FALSE;

} else {

uint8 \*data = (uint8\*) (ind->payload);

uint8 pos = 0;

{ // apollo

pos = 0;

uint8 apollo\_enable = \*(uint8\*)(data + pos);

pos = sizeof(uint8);

uint8 assistant\_type = \*(uint8\*)(data + pos);

pos = sizeof(uint8) + sizeof(uint8) + sizeof(uint8) + sizeof(uint8);

uint32 apollo\_timestamp = 0;

memcpy(&apollo\_timestamp, (data + pos), sizeof(uint32));

/// 比较当前的时间戳，如果大于保存的时间，则更新当前数据

MAKE\_GAIA\_MESSAGE\_WITH\_LEN(APP\_STAROT\_WAKEUP\_CONFIG\_IND, 0);

message->command = STAROT\_BASE\_INFO\_SET\_APOLLO\_WAKEUP\_ENB;

message->messageFrom = MESSAGE\_FROM\_PEER;

message->apollo\_enable = (apollo\_enable & 0XFF);

message->assistant\_type = (assistant\_type & 0XFF);

message->timestamp = apollo\_timestamp;

MessageSend(appGetUiTask(), GAIA\_STAROT\_COMMAND\_IND, message);

}

{ // wear

pos = sizeof(uint8) + sizeof(uint8);

uint8 wear\_enable = \*(uint8\*)(data + pos);

pos = sizeof(uint8) + sizeof(uint8) + sizeof(uint8) + sizeof(uint8) + sizeof(uint32);

uint32 wear\_timestamp = 0;

memcpy(&wear\_timestamp, (data + pos), sizeof(uint32));

/// 比较当前的时间戳，如果大于保存的时间，则更新当前数据

MAKE\_GAIA\_MESSAGE\_WITH\_LEN(APP\_STAROT\_WEAR\_CONFIG\_IND, 0);

message->messageFrom = MESSAGE\_FROM\_PEER;

message->wear\_enable = wear\_enable;

message->timestamp = wear\_timestamp;

message->command = STAROT\_BASE\_INFO\_SET\_ADORN\_CHEAK\_ENB;

MessageSend(appGetUiTask(), GAIA\_STAROT\_COMMAND\_IND, message);

}

return TRUE;

}

}

void appPeerSigMsgNormalConfigConfirmation(Task task, peerSigStatus status) {

UNUSED(task);

UNUSED(status);

}

////-------------------------------------------华丽的分割线-------------------------------------------------

void appPeerSigTxDoubleClickWakeupRequest(Task task, const bdaddr \*peer\_addr, int num) {

peerSigTaskData \*peer\_sig = appGetPeerSig();

STAROT\_MAKE\_MESSAGE(PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T);

message->client\_task = task;

if(num == 0)

MessageSendConditionally(&peer\_sig->task,PEER\_SIG\_INTERNAL\_DOUBLE\_CLICK\_WAKEUP\_REQ , message, appPeerSigStartup(peer\_addr));

if(num == 1)

MessageSendConditionally(&peer\_sig->task, PEER\_SIG\_INTERNAL\_DOUBLE\_CLICK\_WAKEUP\_SYSTEM\_REQ, message, appPeerSigStartup(peer\_addr));

}

void appPeerSigHandleInternalDoubleClickWakeupRequest(PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigHandleInternalBleConfigRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED: {

// uint8 message[AVRCP\_PEER\_CMD\_DOUBLE\_CLICK\_CONFIG\_SIZE];

// /\* Build data for message \*/

// message[0] = req->left;

// message[1] = req->right;

// memcpy(&message[2], req->peerver, DEV\_HWSWVER\_LEN);

/\* Send the handset address over AVRCP \*/

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_DOUBLE\_CLICK\_WAKEUP,

0, NULL);

}

break;

default: {

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

}

break;

}

}

void appPeerSigHandleInternalDoubleClickWakeupSystemRequest(PEER\_SIG\_DOUBLE\_CLICK\_WAKEUP\_REQ\_T \*req) {

DEBUG\_LOG("appPeerSigHandleInternalBleConfigRequest, state %u", appPeerSigGetState());

switch (appPeerSigGetState()) {

case PEER\_SIG\_STATE\_CONNECTED: {

appPeerSigVendorPassthroughRequest(req->client\_task, AVRCP\_PEER\_DOUBLE\_CLICK\_WAKEUP\_SYSTEM,

0, NULL);

}

break;

default: {

appPeerSigMsgConnectHandsetConfirmation(req->client\_task, peerSigStatusLinkKeyTxFail);

}

break;

}

}

bool appPeerSigHandleDoubleClickWakeupCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

UNUSED(ind);

MessageSend(&appGetUi()->task, APP\_ASSISTANT\_TAP\_AWAKEN, 0);

return TRUE;

}

bool appPeerSigHandleDoubleClickWakeupSystemCommand(AV\_AVRCP\_VENDOR\_PASSTHROUGH\_IND\_T \*ind) {

UNUSED(ind);

MessageSend(&appGetUi()->task, APP\_ASSISTANT\_TAP\_SYSTEM, 0);

return TRUE;

}

void appPeerSigMsgDoubleClickWakeupConfirmation(Task task, peerSigStatus status) {

/// todo : 发送消息至指定task，告知同步情况

UNUSED(task), UNUSED(status);

}

# 通话录音

#ifndef AV\_HEADSET\_KYMERA\_H

#define AV\_HEADSET\_KYMERA\_H

#include <chain.h>

#include <transform.h>

#include <hfp.h>

#include <anc.h>

#include <audio\_plugin\_common.h>

#include "av\_headset\_chain\_roles.h"

/\*! \brief Time delay require in ANC peer synchronization. \*/

#define ANC\_PEER\_SYNC\_TIMEOUT\_DELAY (1000U)

#define ANC\_PEER\_REMOTE\_DEVICE\_TIME\_DELAY (100U)

#define ANC\_PEER\_LOCAL\_DEVICE\_TIME\_DELAY (10U)

/\*! \brief The kymera module states. \*/

typedef enum app\_kymera\_states

{

/\*! Kymera is idle. \*/

KYMERA\_STATE\_IDLE,

/\*! Starting master A2DP kymera in three steps. \*/

KYMERA\_STATE\_A2DP\_STARTING\_A,

KYMERA\_STATE\_A2DP\_STARTING\_B,

KYMERA\_STATE\_A2DP\_STARTING\_C,

/\*! Kymera is streaming A2DP locally. \*/

KYMERA\_STATE\_A2DP\_STREAMING,

/\*! Kymera is streaming A2DP locally and forwarding to the slave. \*/

KYMERA\_STATE\_A2DP\_STREAMING\_WITH\_FORWARDING,

/\*! Kymera is streaming SCO locally. \*/

KYMERA\_STATE\_SCO\_ACTIVE,

/\*! Kymera is streaming SCO locally, and may be forwarding \*/

KYMERA\_STATE\_SCO\_ACTIVE\_WITH\_FORWARDING,

/\*! Kymera is receiving forwarded SCO over a link \*/

KYMERA\_STATE\_SCO\_SLAVE\_ACTIVE,

/\*! Kymera is playing a tone or a prompt. \*/

KYMERA\_STATE\_TONE\_PLAYING,

/\*! Kymera is performing ANC tuning. \*/

KYMERA\_STATE\_ANC\_TUNING,

} appKymeraState;

/\*! \brief ANC state \*/

typedef enum app\_kymera\_anc\_states

{

/\*! ANC not initialised, or failed to initialise. \*/

KYMERA\_ANC\_UNINITIALISED,

/\*! ANC turned off. \*/

KYMERA\_ANC\_OFF,

/\*! ANC turned on. \*/

KYMERA\_ANC\_ON

} appKymeraAncState;

/\*! \brief Enumeration of microphone selection options.

The enumeration numbering is important, they match

the input numbering in the Switched Passthrough

Consumer operator.

\*/

typedef enum

{

/\*! Local earbud microphone selected. \*/

MIC\_SELECTION\_LOCAL = 1,

/\*! Remote earbud microphone selected. \*/

MIC\_SELECTION\_REMOTE = 2,

} micSelection;

typedef enum

{

NO\_SCO,

SCO\_NB,

SCO\_WB,

SCO\_SWB,

SCO\_UWB

} appKymeraScoMode;

typedef struct

{

appKymeraScoMode mode;

bool sco\_fwd;

bool mic\_fwd;

bool cvc\_2\_mic;

const chain\_config\_t \*chain;

uint32\_t rate;

} appKymeraScoChainInfo;

/\*! \brief Kymera instance structure.

This structure contains all the information for Kymera audio chains.

\*/

typedef struct

{

/\*! The kymera module's task. \*/

TaskData task;

/\*! The current state. \*/

appKymeraState state;

/\*! The input chain is used in TWS master and slave roles for A2DP streaming

and is typified by containing a decoder. \*/

kymera\_chain\_handle\_t chain\_input\_handle;

/\*! The tone chain is used when a tone is played. \*/

kymera\_chain\_handle\_t chain\_tone\_handle;

/\*! The output\_vol\_handle/sco\_handle chain are used mutually exclusively.

The output\_vol\_handle/sco\_handle both contain OPR\_SOURCE\_SYNC/OPR\_VOLUME\_CONTROL.

These chains are unioned to simplify volume control code: output\_vol\_handle

can always be used with OPR\_VOLUME\_CONTROL regardless of whether SCO/A2DP is active. \*/

union

{

/\*! Used in TWS master and slave roles for A2DP streaming \*/

kymera\_chain\_handle\_t output\_vol\_handle;

/\*! Used for SCO audio. \*/

kymera\_chain\_handle\_t sco\_handle;

} chainu;

kymera\_chain\_handle\_t chain\_record\_handle; // 保存当前的 record CHAIN

/\*! The TWS master packetiser transform packs compressed audio frames

(SBC, AAC, aptX) from the audio subsystem into TWS packets for transmission

over the air to the TWS slave.

The TWS slave packetiser transform receives TWS packets over the air from

the TWS master. It unpacks compressed audio frames and writes them to the

audio subsystem. \*/

Transform packetiser;

/\*! The current output sample rate. \*/

uint32 output\_rate;

/\*! A lock bitfield. Internal messages are typically sent conditionally on

this lock meaning events are queued until the lock is cleared. \*/

uint16 lock;

uint16 busy\_lock;

/\*! The current A2DP stream endpoint identifier. \*/

uint8 a2dp\_seid;

/\*! The current playing tone client's lock. \*/

uint16 \*tone\_client\_lock;

/\*! The current playing tone client lock mask - bits to clear in the lock

when the tone is stopped. \*/

uint16 tone\_client\_lock\_mask;

/\*! Number of tones/prompts playing and queued up to be played \*/

uint8 tone\_count;

/\*! Which microphone to use during mic forwarding \*/

micSelection mic;

const appKymeraScoChainInfo \*sco\_info;

/\*! The prompt file source whilst prompt is playing \*/

Source prompt\_source;

/\*! ANC state \*/

appKymeraAncState anc\_state;

anc\_mic\_params\_t anc\_mic\_params;

audio\_mic\_params mic\_params[4];

uint8 dac\_amp\_usage;

/\*! ANC tuning state \*/

uint16 usb\_rate;

BundleID anc\_tuning\_bundle\_id;

#ifdef DOWNLOAD\_USB\_AUDIO

BundleID usb\_audio\_bundle\_id;

#endif

Operator usb\_rx, anc\_tuning, usb\_tx;

} kymeraTaskData;

/\*! \brief Start streaming A2DP audio.

\param client\_lock If not NULL, bits set in client\_lock\_mask will be cleared

in client\_lock when A2DP is started, or if an A2DP stop is requested,

before A2DP has started.

\param client\_lock\_mask A mask of bits to clear in the client\_lock.

\param codec\_settings The A2DP codec settings.

\param volume The start volume.

\param master\_pre\_start\_delay This function always sends an internal message

to request the module start kymera. The internal message is sent conditionally

on the completion of other activities, e.g. a tone. The caller may request

that the internal message is sent master\_pre\_start\_delay additional times before the

start of kymera commences. The intention of this is to allow the caller to

delay the starting of kymera (with its long, blocking functions) to match

the message pipeline of some concurrent message sequence the caller doesn't

want to be blocked by the starting of kymera. This delay is only applied

when starting the 'master' (a non-TWS sink SEID).

\*/

void appKymeraA2dpStart(uint16 \*client\_lock, uint16 client\_lock\_mask,

const a2dp\_codec\_settings \*codec\_settings,

uint8 volume, uint8 master\_pre\_start\_delay);

/\*! \brief Stop streaming A2DP audio.

\param seid The stream endpoint ID to stop.

\param source The source associatied with the seid.

\*/

void appKymeraA2dpStop(uint8 seid, Source source);

/\*! \brief Set the A2DP streaming volume.

\param volume The desired volume in the range 0 (mute) to 127 (max).

\*/

void appKymeraA2dpSetVolume(uint16 volume);

/\*! \brief Start SCO audio.

\param audio\_sink The SCO audio sink.

\param codec WB-Speech codec bit masks.

\param wesco The link Wesco.

\param volume The starting volume.

\param pre\_start\_delay This function always sends an internal message

to request the module start SCO. The internal message is sent conditionally

on the completion of other activities, e.g. a tone. The caller may request

that the internal message is sent pre\_start\_delay additional times before

starting kymera. The intention of this is to allow the caller to

delay the start of kymera (with its long, blocking functions) to match

the message pipeline of some concurrent message sequence the caller doesn't

want to be blocked by the starting of kymera.

\param allow\_scofwd Allow the use of SCO forwarding if it is supported by

this device.

\*/

bool appKymeraScoStart(Sink audio\_sink, appKymeraScoMode mode, bool \*allow\_scofwd, bool \*allow\_micfwd,

uint8 wesco, uint16 volume, uint8 pre\_start\_delay);

/\*! \brief Stop SCO audio.

\*/

void appKymeraScoStop(void);

/\*! \brief Start a chain for receiving forwarded SCO audio.

\param link\_source The source used for data received from the SCO forwarding link

\param volume volume level to use

\param enable\_mic\_fwd Should MIC forwarding be enabled or not.

\param pre\_start\_delay Maximum number of times message is resent before being actioned

\*/

void appKymeraScoSlaveStart(Source link\_source, uint8 volume, bool enable\_mic\_fwd,

uint16 pre\_start\_delay);

/\*! \brief Stop chain receiving forwarded SCO audio.

\*/

void appKymeraScoSlaveStop(void);

/\*! \brief Start sending forwarded audio.

\param forwarding\_sink The BT sink to the air.

\param enable\_mic\_fwd Should MIC forwarding be enabled or not.

\note If the SCO is to be forwarded then the full chain,

including local playback, is started by this call.

\*/

void appKymeraScoStartForwarding(Sink forwarding\_sink, bool enable\_mic\_fwd);

/\*! \brief Stop sending received SCO audio to the peer

Local playback of SCO will continue, although in most cases

will be stopped by a separate command almost immediately.

\*/

void appKymeraScoStopForwarding(void);

/\*! \brief Enable/Disable generation of forwarded SCO audio.

\param[in] pause If TRUE will stop generating SCO audio data, if

FALSE will resume.

This function leaves the SCO/SFWD chains up and running and flips

the switched passthrough consumer between 'consume' and 'passthrough'

modes to control if forwarded SCO data generated on the stream

source for the SCOFWD packetiser to send.

\*/

void appKymeraScoForwardingPause(bool pause);

/\*! \brief Set SCO volume.

\param[in] volume HFP volume in the range 0 (mute) to 15 (max).

\*/

void appKymeraScoSetVolume(uint8 volume);

/\*! \brief Enable or disable MIC muting.

\param[in] mute TRUE to mute MIC, FALSE to unmute MIC.

\*/

void appKymeraScoMicMute(bool mute);

/\*! \brief Get the SCO CVC voice quality.

\return The voice quality.

\*/

uint8 appKymeraScoVoiceQuality(void);

/\*! \brief Select the local Microphone for use

Switches audio from remote (forwarded) Microphone to local Mic src.

\*/

void appKymeraScoUseLocalMic(void);

/\*! \brief Select the remote Microphone for use

Switches audio from local Microphone to the remote (forwarded) Mic src.

\*/

void appKymeraScoUseRemoteMic(void);

/\*! \brief Play a tone.

\param tone The address of the tone to play.

\param interruptible If TRUE, the tone may be interrupted by another event

before it is completed. If FALSE, the tone may not be interrupted by

another event and will play to completion.

\param client\_lock If not NULL, bits set in client\_lock\_mask will be cleared

in client\_lock when the tone finishes - either on completion, or when

interrupted.

\param client\_lock\_mask A mask of bits to clear in the client\_lock.

\*/

void appKymeraTonePlay(const ringtone\_note \*tone, bool interruptible,

uint16 \*client\_lock, uint16 client\_lock\_mask);

/\*! \brief The prompt file format \*/

typedef enum prompt\_format

{

PROMPT\_FORMAT\_PCM,

PROMPT\_FORMAT\_SBC,

} promptFormat;

/\*! \brief Play a prompt.

\param prompt The file index of the prompt to play.

\param format The prompt file format.

\param rate The prompt sample rate.

\param interruptible If TRUE, the prompt may be interrupted by another event

before it is completed. If FALSE, the prompt may not be interrupted by

another event and will play to completion.

\param client\_lock If not NULL, bits set in client\_lock\_mask will be cleared

in client\_lock when the prompt finishes - either on completion, or when

interrupted.

\param client\_lock\_mask A mask of bits to clear in the client\_lock.

\*/

void appKymeraPromptPlay(FILE\_INDEX prompt, promptFormat format,

uint32 rate, bool interruptible,

uint16 \*client\_lock, uint16 client\_lock\_mask);

/\*! \brief Initialise the kymera module. \*/

void appKymeraInit(void);

void appKymeraMicInit(void);

void appKymeraMicSetup(uint8 mic1a, Source \*p\_mic\_src\_1a, uint8 mic\_1b, Source \*p\_mic\_src\_1b, uint16 rate);

void appKymeraMicCleanup(uint8 mic1a, uint8 mic1b);

void appKymeraAncInit(void);

void appKymeraAncEnable(void);

void appKymeraAncDisable(void);

bool appKymeraAncIsEnabled(void);

void appKymeraAncTuningStart(uint16 usb\_rate);

void appKymeraAncTuningStop(void);

kymera\_chain\_handle\_t appKymeraScoCreateChain(const appKymeraScoChainInfo \*info);

#define appKymeraAncMicParams() (&(appGetKymera()->anc\_mic\_params))

#define appKymeraIsTonePlaying() (appGetKymera()->tone\_count > 0)

/\*! \brief To enable the ANC in peer synchronization part

using Kymera task to send the ANC ON message

\*/

void appKymeraAncEnableSynchronizeWithPeer(void);

/\*! \brief To disable the ANC in peer synchronization part

using Kymera task to send the ANC OFF message

\*/

void appKymeraAncDisableSynchronizeWithPeer(void);

/\*! \brief To enable the ANC after some specific time delay. Queued messages will be cancelled if any.

using Kymera task to send the ANC ON message

\*/

void appKymeraAncDelayedEnable(uint16 anc\_delay);

/\*! \brief To disable the ANC after some specific time delay. Queued messages will be cancelled if any.

using Kymera task to send the ANC OFF message

\*/

void appKymeraAncDelayedDisable(uint16 anc\_delay);

#endif // AV\_HEADSET\_KYMERA\_H

#include "av\_headset\_kymera\_private.h"

#include "chains/chain\_sco\_nb.h"

#include "chains/chain\_sco\_wb.h"

#include "chains/chain\_sco\_swb.h"

#include "chains/chain\_micfwd\_wb.h"

#include "chains/chain\_micfwd\_nb.h"

#include "chains/chain\_scofwd\_wb.h"

#include "chains/chain\_scofwd\_nb.h"

#include "chains/chain\_scofwd\_wb1.h"

#include "chains/chain\_scofwd\_wb2.h"

#include "chains/chain\_micfwd1\_wb.h"

#include "chains/chain\_micfwd2\_wb.h"

#include "chains/chain\_sco\_nb\_2mic.h"

#include "chains/chain\_sco\_wb\_2mic.h"

#include "chains/chain\_micfwd\_wb\_2mic.h"

#include "chains/chain\_micfwd\_nb\_2mic.h"

#include "chains/chain\_scofwd\_wb\_2mic.h"

#include "chains/chain\_scofwd\_nb\_2mic.h"

#include "chains/chain\_micfwd\_send.h"

#include "chains/chain\_scofwd\_recv.h"

#include "chains/chain\_micfwd\_send\_2mic.h"

#include "chains/chain\_scofwd\_recv\_2mic.h"

/\*! Macro for creating messages \*/

#define MAKE\_KYMERA\_MESSAGE(TYPE) \

TYPE##\_T \*message = PanicUnlessNew(TYPE##\_T);

const appKymeraScoChainInfo appKymeraScoChainTable[] =

{

/\*sco\_mode sco\_fwd mic\_fwd cvc\_2\_mic chain rate \*/

{ SCO\_NB, FALSE, FALSE, FALSE, &chain\_sco\_nb\_config, 8000 },

{ SCO\_WB, FALSE, FALSE, FALSE, &chain\_sco\_wb\_config, 16000 },

{ SCO\_SWB, FALSE, FALSE, FALSE, &chain\_sco\_swb\_config, 32000 },

{ SCO\_NB, TRUE, FALSE, FALSE, &chain\_scofwd\_nb\_config, 8000 },

#if (FORWARD\_AUDIO\_TYPE == (FORWARD\_AUDIO\_SCO | FORWARD\_AUDIO\_MIC))

{ SCO\_WB, TRUE, FALSE, FALSE, &chain\_scofwd\_wb\_config, 16000 },

{ SCO\_WB, TRUE, TRUE, FALSE, &chain\_micfwd\_wb\_config, 16000 },

#elif (FORWARD\_AUDIO\_TYPE == FORWARD\_AUDIO\_SCO)

{ SCO\_WB, TRUE, FALSE, FALSE, &chain\_scofwd\_wb1\_config, 16000 },

{ SCO\_WB, TRUE, TRUE, FALSE, &chain\_micfwd1\_wb\_config, 16000 },

#elif (FORWARD\_AUDIO\_TYPE == FORWARD\_AUDIO\_MIC)

{ SCO\_WB, TRUE, FALSE, FALSE, &chain\_scofwd\_wb2\_config, 16000 },

{ SCO\_WB, TRUE, TRUE, FALSE, &chain\_micfwd2\_wb\_config, 16000 },

#endif

{ SCO\_NB, TRUE, TRUE, FALSE, &chain\_micfwd\_nb\_config, 8000 },

{ SCO\_NB, FALSE, FALSE, TRUE, &chain\_sco\_nb\_2mic\_config, 8000 },

{ SCO\_WB, FALSE, FALSE, TRUE, &chain\_sco\_wb\_2mic\_config, 16000 },

/\*{ SCO\_SWB, FALSE, FALSE, TRUE, &chain\_sco\_swb\_2mic\_config, 32000 },\*/

{ SCO\_NB, TRUE, FALSE, TRUE, &chain\_scofwd\_nb\_2mic\_config, 8000 },

{ SCO\_WB, TRUE, FALSE, TRUE, &chain\_scofwd\_wb\_2mic\_config, 16000 },

{ SCO\_NB, TRUE, TRUE, TRUE, &chain\_micfwd\_nb\_2mic\_config, 8000 },

{ SCO\_WB, TRUE, TRUE, TRUE, &chain\_micfwd\_wb\_2mic\_config, 16000 },

{ NO\_SCO }

};

const appKymeraScoChainInfo appKymeraScoSlaveChainTable[] =

{

/\* sco\_mode sco\_fwd mic\_fwd cvc\_2\_mic chain rate \*/

{ SCO\_WB, FALSE, FALSE, FALSE, &chain\_scofwd\_recv\_config, 16000 },

{ SCO\_WB, FALSE, TRUE, FALSE, &chain\_micfwd\_send\_config, 16000 },

{ SCO\_WB, FALSE, FALSE, TRUE, &chain\_scofwd\_recv\_2mic\_config, 16000 },

{ SCO\_WB, FALSE, TRUE, TRUE, &chain\_micfwd\_send\_2mic\_config, 16000 },

{ NO\_SCO }

};

static const capability\_bundle\_t capability\_bundle[] =

{

#ifdef DOWNLOAD\_SWITCHED\_PASSTHROUGH

{

"download\_switched\_passthrough\_consumer.edkcs",

capability\_bundle\_available\_p0

},

#endif

#ifdef DOWNLOAD\_APTX\_CLASSIC\_DEMUX

{

"download\_aptx\_demux.edkcs",

capability\_bundle\_available\_p0

},

#endif

#ifdef DOWNLOAD\_AEC\_REF

{

"download\_aec\_reference.edkcs",

capability\_bundle\_available\_p0

},

#endif

#ifdef DOWNLOAD\_APTX\_ADAPTIVE\_DECODE

{

"download\_aptx\_adaptive\_decode.edkcs",

capability\_bundle\_available\_p0

},

#endif

#ifdef DOWNLOAD\_AAC\_DECODER

{

"download\_aac\_decode.edkcs",

capability\_bundle\_available\_p0

},

#endif

#if defined(DOWNLOAD\_ASYNC\_WBS\_DEC) || defined(DOWNLOAD\_ASYNC\_WBS\_ENC)

/\* Chains for SCO forwarding.

Likely to update to use the downloadable AEC regardless

as offers better TTP support (synchronisation) and other

extensions \*/

{

"download\_async\_wbs.edkcs",

capability\_bundle\_available\_p0

},

#endif

#if defined(DOWNLOAD\_PASSSTHROUGH)

{

"download\_passthrough.dkcs",

capability\_bundle\_available\_p0

},

#endif

#if defined(DOWNLOAD\_G722\_ENCODER)

{

"download\_G722Codec.dkcs",

capability\_bundle\_available\_p0

},

#endif

{

0, 0

}

};

static const capability\_bundle\_config\_t bundle\_config = {capability\_bundle, ARRAY\_DIM(capability\_bundle) - 1};

void appKymeraPromptPlay(FILE\_INDEX prompt, promptFormat format, uint32 rate,

bool interruptible, uint16 \*client\_lock, uint16 client\_lock\_mask)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraPromptPlay, queue prompt %d, int %u", prompt, interruptible);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_TONE\_PROMPT\_PLAY);

message->tone = NULL;

message->prompt = prompt;

message->prompt\_format = format;

message->rate = rate;

message->interruptible = interruptible;

message->client\_lock = client\_lock;

message->client\_lock\_mask = client\_lock\_mask;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_TONE\_PROMPT\_PLAY, message, &theKymera->lock);

theKymera->tone\_count++;

}

#ifdef CONFIG\_REC\_ASSISTANT

// 其它模块调用这个函数来开启模块

void appKymeraRecordStart(void)

{

kymeraTaskData \*theKymera = appGetKymera();

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_RECORD);

DEBUG\_LOGF("appKymeraStartRecord");

message->rate = 16000;

MessageSend(&theKymera->task, KYMERA\_INTERNAL\_RECORD\_START, message);

}

#endif

void appKymeraTonePlay(const ringtone\_note \*tone, bool interruptible,

uint16 \*client\_lock, uint16 client\_lock\_mask)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraTonePlay, queue tone %p, int %u", tone, interruptible);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_TONE\_PROMPT\_PLAY);

message->tone = tone;

message->prompt = FILE\_NONE;

message->rate = KYMERA\_TONE\_GEN\_RATE;

message->interruptible = interruptible;

message->client\_lock = client\_lock;

message->client\_lock\_mask = client\_lock\_mask;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_TONE\_PROMPT\_PLAY, message, &theKymera->lock);

theKymera->tone\_count++;

}

void appKymeraA2dpStart(uint16 \*client\_lock, uint16 client\_lock\_mask,

const a2dp\_codec\_settings \*codec\_settings,

uint8 volume, uint8 master\_pre\_start\_delay)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraA2dpStart, seid %u, lock %u, busy\_lock %u", codec\_settings->seid, theKymera->lock, theKymera->busy\_lock);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_START);

message->lock = client\_lock;

message->lock\_mask = client\_lock\_mask;

message->codec\_settings = \*codec\_settings;

message->volume = volume;

message->master\_pre\_start\_delay = master\_pre\_start\_delay;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_A2DP\_START,

message,

&theKymera->lock);

}

void appKymeraA2dpStop(uint8 seid, Source source)

{

kymeraTaskData \*theKymera = appGetKymera();

MessageId mid = appA2dpIsSeidSource(seid) ? KYMERA\_INTERNAL\_A2DP\_STOP\_FORWARDING :

KYMERA\_INTERNAL\_A2DP\_STOP;

DEBUG\_LOGF("appKymeraA2dpStop, seid %u", seid);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_STOP);

message->seid = seid;

message->source = source;

MessageSendConditionally(&theKymera->task, mid, message, &theKymera->lock);

}

void appKymeraA2dpSetVolume(uint16 volume)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraA2dpSetVolume, volume %u", volume);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_SET\_VOL);

message->volume = volume;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_A2DP\_SET\_VOL, message, &theKymera->lock);

}

void appKymeraScoStartForwarding(Sink forwarding\_sink, bool enable\_mic\_fwd)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraScoStartForwarding, queue sink %p, state %u", forwarding\_sink, appKymeraGetState());

PanicNull(forwarding\_sink);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_START\_FORWARDING\_TX);

message->forwarding\_sink = forwarding\_sink;

message->enable\_mic\_fwd = enable\_mic\_fwd;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_START\_FORWARDING\_TX, message, &theKymera->lock);

}

void appKymeraScoStopForwarding(void)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraScoStopForwarding, state %u", appKymeraGetState());

if (!appKymeraHandleInternalScoForwardingStopTx())

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_STOP\_FORWARDING\_TX, NULL, &theKymera->lock);

}

kymera\_chain\_handle\_t appKymeraScoCreateChain(const appKymeraScoChainInfo \*info)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraCreateScoChain, mode %u, sco\_fwd %u, mic\_fwd %u, cvc\_2\_mic %u, rate %u",

info->mode, info->sco\_fwd, info->mic\_fwd, info->cvc\_2\_mic, info->rate);

theKymera->sco\_info = info;

/\* Ensure audio is turned on \*/

OperatorFrameworkEnable(1);

/\* Configure DSP power mode appropriately for SCO chain \*/

appKymeraConfigureDspPowerMode(FALSE);

/\* Create chain and return handle \*/

DEBUG\_LOG("appKymeraScoCreateChain chain=%p\n", info->chain);

theKymera->chainu.sco\_handle = ChainCreate(info->chain);

/\* Now chain is created, we can decrement reference count \*/

OperatorFrameworkEnable(0);

return theKymera->chainu.sco\_handle;

}

static void appKymeraScoStartHelper(Sink audio\_sink, const appKymeraScoChainInfo \*info, uint8 wesco,

uint16 volume, uint8 pre\_start\_delay, bool conditionally)

{

kymeraTaskData \*theKymera = appGetKymera();

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_START);

PanicNull(audio\_sink);

message->audio\_sink = audio\_sink;

message->wesco = wesco;

message->volume = volume;

message->pre\_start\_delay = pre\_start\_delay;

message->sco\_info = info;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_START, message, conditionally ? &theKymera->busy\_lock : NULL);

}

bool appKymeraScoStart(Sink audio\_sink, appKymeraScoMode mode, bool \*allow\_scofwd, bool \*allow\_micfwd,

uint8 wesco, uint16 volume, uint8 pre\_start\_delay)

{

#ifdef CONFIG\_MIC\_SELECT

const bool cvc\_2\_mic = (appConfigScoMic1() != NO\_MIC) && (appConfigScoMic2() != NO\_MIC);

#else

const bool cvc\_2\_mic = appConfigScoMic2() != NO\_MIC;

#endif

const appKymeraScoChainInfo \*info = appKymeraScoFindChain(appKymeraScoChainTable,

mode, \*allow\_scofwd, \*allow\_micfwd,

cvc\_2\_mic);

if (!info)

info = appKymeraScoFindChain(appKymeraScoChainTable,

mode, FALSE, FALSE, cvc\_2\_mic);

if (info)

{

DEBUG\_LOGF("appKymeraScoStart: selected chain: %d", info - appKymeraScoChainTable);

DEBUG\_LOGF("appKymeraScoStart, queue sink 0x%x", audio\_sink);

\*allow\_scofwd = info->sco\_fwd;

\*allow\_micfwd = info->mic\_fwd;

appKymeraScoStartHelper(audio\_sink, info, wesco, volume, pre\_start\_delay, TRUE);

return TRUE;

}

else

{

DEBUG\_LOGF("appKymeraScoStart, failed to find suitable SCO chain");

return FALSE;

}

}

void appKymeraScoStop(void)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOG("appKymeraScoStop");

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_STOP, NULL, &theKymera->lock);

}

void appKymeraScoSlaveStartHelper(Source link\_source, uint8 volume, const appKymeraScoChainInfo \*info, uint16 delay)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraScoSlaveStartHelper, delay %u", delay);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_SLAVE\_START);

message->link\_source = link\_source;

message->volume = volume;

message->sco\_info = info;

message->pre\_start\_delay = delay;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_SLAVE\_START, message, &theKymera->lock);

}

void appKymeraScoSlaveStart(Source link\_source, uint8 volume, bool allow\_micfwd, uint16 pre\_start\_delay)

{

DEBUG\_LOGF("appKymeraScoSlaveStart, source 0x%x", link\_source);

#ifdef CONFIG\_MIC\_SELECT

const bool cvc\_2\_mic = (appConfigScoMic1() != NO\_MIC) && (appConfigScoMic2() != NO\_MIC);

#else

const bool cvc\_2\_mic = appConfigScoMic2() != NO\_MIC;

#endif

const appKymeraScoChainInfo \*info = appKymeraScoFindChain(appKymeraScoSlaveChainTable,

SCO\_WB, FALSE, allow\_micfwd,

cvc\_2\_mic);

PanicNull(link\_source);

appKymeraScoSlaveStartHelper(link\_source, volume, info, pre\_start\_delay);

}

void appKymeraScoSlaveStop(void)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOG("appKymeraScoSlaveStop");

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCOFWD\_RX\_STOP, NULL, &theKymera->lock);

}

void appKymeraScoSetVolume(uint8 volume)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraScoSetVolume msg, vol %u", volume);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_SET\_VOL);

message->volume = volume;

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_SCO\_SET\_VOL, message, &theKymera->lock);

}

void appKymeraScoMicMute(bool mute)

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOGF("appKymeraScoMicMute msg, mute %u", mute);

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_MIC\_MUTE);

message->mute = mute;

MessageSend(&theKymera->task, KYMERA\_INTERNAL\_SCO\_MIC\_MUTE, message);

}

void appKymeraScoUseLocalMic(void)

{

/\* Only do something if both EBs support MIC forwarding \*/

if (appDeviceIsPeerMicForwardSupported())

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOG("appKymeraScoUseLocalMic");

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_MICFWD\_LOCAL\_MIC, NULL, &theKymera->lock);

}

}

void appKymeraScoUseRemoteMic(void)

{

/\* Only do something if both EBs support MIC forwarding \*/

if (appDeviceIsPeerMicForwardSupported())

{

kymeraTaskData \*theKymera = appGetKymera();

DEBUG\_LOG("appKymeraScoUseRemoteMic");

MessageSendConditionally(&theKymera->task, KYMERA\_INTERNAL\_MICFWD\_REMOTE\_MIC, NULL, &theKymera->lock);

}

}

static void kymera\_dsp\_msg\_handler(MessageFromOperator \*op\_msg)

{

PanicFalse(op\_msg->len == KYMERA\_OP\_MSG\_LEN);

switch (op\_msg->message[KYMERA\_OP\_MSG\_WORD\_MSG\_ID])

{

case KYMERA\_OP\_MSG\_ID\_TONE\_END:

DEBUG\_LOG("KYMERA\_OP\_MSG\_ID\_TONE\_END");

appKymeraTonePromptStop();

break;

default:

break;

}

}

static void kymera\_msg\_handler(Task task, MessageId id, Message msg)

{

kymeraTaskData \*theKymera = appGetKymera();

UNUSED(task);

switch (id)

{

case MESSAGE\_FROM\_OPERATOR:

kymera\_dsp\_msg\_handler((MessageFromOperator \*)msg);

break;

case MESSAGE\_SOURCE\_EMPTY:

break;

case MESSAGE\_STREAM\_DISCONNECT:

DEBUG\_LOG("appKymera MESSAGE\_STREAM\_DISCONNECT");

appKymeraTonePromptStop();

break;

case KYMERA\_INTERNAL\_A2DP\_START:

{

const KYMERA\_INTERNAL\_A2DP\_START\_T \*m = (const KYMERA\_INTERNAL\_A2DP\_START\_T \*)msg;

uint8 seid = m->codec\_settings.seid;

/\* Check if we are busy (due to other chain in use) \*/

if (!appA2dpIsSeidSource(seid) && theKymera->busy\_lock)

{

/\* Re-send message blocked on busy\_lock \*/

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_START);

\*message = \*m;

MessageSendConditionally(&theKymera->task, id, message, &theKymera->busy\_lock);

break;

}

/\* If there is no pre-start delay, or during the pre-start delay, the

start can be cancelled if there is a stop on the message queue \*/

MessageId mid = appA2dpIsSeidSource(seid) ? KYMERA\_INTERNAL\_A2DP\_STOP\_FORWARDING :

KYMERA\_INTERNAL\_A2DP\_STOP;

if (MessageCancelFirst(&theKymera->task, mid))

{

/\* A stop on the queue was cancelled, clear the starter's lock

and stop starting \*/

DEBUG\_LOGF("appKymera not starting due to queued stop, seid=%u", seid);

if (m->lock)

{

\*m->lock &= ~m->lock\_mask;

}

/\* Also clear kymera's lock, since no longer starting \*/

appKymeraClearStartingLock(theKymera);

break;

}

if (m->master\_pre\_start\_delay)

{

/\* Send another message before starting kymera. \*/

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_START);

\*message = \*m;

--message->master\_pre\_start\_delay;

MessageSend(&theKymera->task, id, message);

appKymeraSetStartingLock(theKymera);

break;

}

}

// fallthrough (no message cancelled, zero master\_pre\_start\_delay)

case KYMERA\_INTERNAL\_A2DP\_STARTING:

{

const KYMERA\_INTERNAL\_A2DP\_START\_T \*m = (const KYMERA\_INTERNAL\_A2DP\_START\_T \*)msg;

if (appKymeraHandleInternalA2dpStart(m))

{

/\* Start complete, clear locks. \*/

appKymeraClearStartingLock(theKymera);

if (m->lock)

{

\*m->lock &= ~m->lock\_mask;

}

}

else

{

/\* Start incomplete, send another message. \*/

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_A2DP\_START);

\*message = \*m;

MessageSend(&theKymera->task, KYMERA\_INTERNAL\_A2DP\_STARTING, message);

}

}

break;

case KYMERA\_INTERNAL\_A2DP\_STOP:

case KYMERA\_INTERNAL\_A2DP\_STOP\_FORWARDING:

appKymeraHandleInternalA2dpStop(msg);

break;

case KYMERA\_INTERNAL\_A2DP\_SET\_VOL:

{

KYMERA\_INTERNAL\_A2DP\_SET\_VOL\_T \*m = (KYMERA\_INTERNAL\_A2DP\_SET\_VOL\_T \*)msg;

appKymeraHandleInternalA2dpSetVolume(m->volume);

}

break;

case KYMERA\_INTERNAL\_SCO\_START:

{

const KYMERA\_INTERNAL\_SCO\_START\_T \*m = (const KYMERA\_INTERNAL\_SCO\_START\_T \*)msg;

if (m->pre\_start\_delay)

{

/\* Resends are sent unconditonally, but the lock is set blocking

other new messages \*/

appKymeraSetStartingLock(appGetKymera());

appKymeraScoStartHelper(m->audio\_sink, m->sco\_info, m->wesco, m->volume,

m->pre\_start\_delay - 1, FALSE);

}

else

{

appKymeraHandleInternalScoStart(m->audio\_sink, m->sco\_info, m->wesco, m->volume);

appKymeraClearStartingLock(appGetKymera());

}

}

break;

case KYMERA\_INTERNAL\_SCO\_START\_FORWARDING\_TX:

{

const KYMERA\_INTERNAL\_SCO\_START\_FORWARDING\_TX\_T \*m =

(const KYMERA\_INTERNAL\_SCO\_START\_FORWARDING\_TX\_T\*)msg;

appKymeraHandleInternalScoForwardingStartTx(m->forwarding\_sink);

}

break;

case KYMERA\_INTERNAL\_SCO\_STOP\_FORWARDING\_TX:

{

appKymeraHandleInternalScoForwardingStopTx();

}

break;

case KYMERA\_INTERNAL\_SCO\_SET\_VOL:

{

KYMERA\_INTERNAL\_SCO\_SET\_VOL\_T \*m = (KYMERA\_INTERNAL\_SCO\_SET\_VOL\_T \*)msg;

appKymeraHandleInternalScoSetVolume(m->volume);

}

break;

case KYMERA\_INTERNAL\_SCO\_MIC\_MUTE:

{

KYMERA\_INTERNAL\_SCO\_MIC\_MUTE\_T \*m = (KYMERA\_INTERNAL\_SCO\_MIC\_MUTE\_T \*)msg;

appKymeraHandleInternalScoMicMute(m->mute);

}

break;

case KYMERA\_INTERNAL\_SCO\_STOP:

{

appKymeraHandleInternalScoStop();

}

break;

case KYMERA\_INTERNAL\_SCO\_SLAVE\_START:

{

const KYMERA\_INTERNAL\_SCO\_SLAVE\_START\_T \*m = (const KYMERA\_INTERNAL\_SCO\_SLAVE\_START\_T \*)msg;

if (theKymera->busy\_lock)

{

/\* Re-send message blocked on busy\_lock \*/

MAKE\_KYMERA\_MESSAGE(KYMERA\_INTERNAL\_SCO\_SLAVE\_START);

\*message = \*m;

MessageSendConditionally(&theKymera->task, id, message, &theKymera->busy\_lock);

}

#if 0

/\* If we are not idle (a pre-requisite) and this message can be delayed,

then re-send it. The normal situation is message delays when stopping

A2DP/AV. That is calls were issued in the right order to stop A2DP then

start SCO receive but the number of messages required for each were

different, leading the 2nd action to complete 1st. \*/

if ( start\_req->pre\_start\_delay

&& appKymeraGetState() != KYMERA\_STATE\_IDLE)

{

DEBUG\_LOG("appKymeraHandleInternalScoForwardingStartRx, re-queueing.");

appKymeraScoFwdStartReceiveHelper(start\_req->link\_source, start\_req->volume,

start\_req->sco\_info,

start\_req->pre\_start\_delay - 1);

return;

}

#endif

else

appKymeraHandleInternalScoSlaveStart(m->link\_source, m->sco\_info, m->volume);

}

break;

case KYMERA\_INTERNAL\_SCOFWD\_RX\_STOP:

{

appKymeraHandleInternalScoSlaveStop();

}

break;

case KYMERA\_INTERNAL\_TONE\_PROMPT\_PLAY:

appKymeraHandleInternalTonePromptPlay(msg);

break;

#ifdef CONFIG\_REC\_ASSISTANT

case KYMERA\_INTERNAL\_RECORD\_START:

appKymeraHandleInternalRecordStart(msg);

break;

#endif

case KYMERA\_INTERNAL\_MICFWD\_LOCAL\_MIC:

appKymeraSwitchSelectMic(MIC\_SELECTION\_LOCAL);

break;

case KYMERA\_INTERNAL\_MICFWD\_REMOTE\_MIC:

appKymeraSwitchSelectMic(MIC\_SELECTION\_REMOTE);

break;

case KYMERA\_INTERNAL\_ANC\_TUNING\_START:

{

const KYMERA\_INTERNAL\_ANC\_TUNING\_START\_T \*m = (const KYMERA\_INTERNAL\_ANC\_TUNING\_START\_T \*)msg;

appKymeraAncTuningCreateChain(m->usb\_rate);

}

break;

case KYMERA\_INTERNAL\_ANC\_TUNING\_STOP:

appKymeraAncTuningDestroyChain();

break;

case KYMERA\_INTERNAL\_AUDIO\_SS\_DISABLE:

DEBUG\_LOG("appKymera KYMERA\_INTERNAL\_AUDIO\_SS\_DISABLE");

OperatorFrameworkEnable(MAIN\_PROCESSOR\_OFF);

break;

case KYMERA\_INTERNAL\_ANC\_ON:

appKymeraAncEnable();

break;

case KYMERA\_INTERNAL\_ANC\_OFF:

appKymeraAncDisable();

break;

default:

break;

}

}

void appKymeraInit(void)

{

kymeraTaskData \*theKymera = appGetKymera();

memset(theKymera, 0, sizeof(\*theKymera));

theKymera->task.handler = kymera\_msg\_handler;

theKymera->state = KYMERA\_STATE\_IDLE;

theKymera->output\_rate = 0;

theKymera->lock = theKymera->busy\_lock = 0;

theKymera->a2dp\_seid = AV\_SEID\_INVALID;

theKymera->tone\_count = 0;

appKymeraExternalAmpSetup();

if (bundle\_config.number\_of\_capability\_bundles > 0)

ChainSetDownloadableCapabilityBundleConfig(&bundle\_config);

theKymera->mic = MIC\_SELECTION\_LOCAL;

appKymeraMicInit();

appKymeraAncInit();

}

# 数据传输

#ifndef AV\_HEADSET\_GAIA\_STAROT\_H

#define AV\_HEADSET\_GAIA\_STAROT\_H

#include <gaia.h>

#include <../av\_headset.h>

enum GAIA\_AUDIO\_TYPE {

GAIA\_AUDIO\_SPEAKER = 1,

GAIA\_AUDIO\_MIC = 2

};

/// 消息的来源

enum MessageFromSource {

MESSAGE\_FROM\_APP, /// 从APP发送来的消息

MESSAGE\_FROM\_PEER /// 从Peer耳机转发的消息

};

typedef struct

{

uint16 command;

uint16 payloadLen;

uint16 messageFrom;

uint8 payload[4];

} GAIA\_STAROT\_CONFIG\_IND\_T;

typedef GAIA\_STAROT\_CONFIG\_IND\_T GAIA\_STAROT\_CONFIG\_IND;

enum {

ASSISTANT\_TYPE\_APP = 1,

ASSISTANT\_TYPE\_SYSTEM = 2

};

typedef struct

{

uint16 command;

uint8 messageFrom;

uint8 apollo\_enable;

uint32 timestamp;

uint8 assistant\_type;

uint8 unused;

} APP\_STAROT\_WAKEUP\_CONFIG\_IND\_T;

typedef APP\_STAROT\_WAKEUP\_CONFIG\_IND\_T APP\_STAROT\_WAKEUP\_CONFIG\_IND;

typedef struct

{

uint16 command;

uint8 messageFrom;

uint8 wear\_enable;

uint32 timestamp;

} APP\_STAROT\_WEAR\_CONFIG\_IND\_T;

typedef APP\_STAROT\_WEAR\_CONFIG\_IND\_T APP\_STAROT\_WEAR\_CONFIG\_IND;

////////////////////////////EVENT//////////////////////////////

enum {

/// 临时，不能这样定义，会冲突

GAIA\_STAROT\_AUDIO\_INTERVAL = GAIA\_MESSAGE\_TOP + 1,

GAIA\_STAROT\_RESEND\_AUDIO,

};

///////////////////////////COMMAND/////////////////////////////

#define GAIA\_COMMAND\_STAROT\_FIRST\_COMMAND (0x5001)

#define GAIA\_COMMAND\_STAROT\_START\_SEND\_TIMER (0x5002)

#define GAIA\_COMMAND\_STAROT\_STOP\_SEND\_TIMER (0x5003)

#define GAIA\_COMMAND\_STAROT\_TRANS\_AUDIO (0x5004)

///////////////////////////dialog//////////////////////////////////

/// 电话开始

enum {

GAIA\_COMMAND\_STAROT\_CALL\_BEGIN = 0X5005, // 通话接入 还没有接通

GAIA\_COMMAND\_STAROT\_CALL\_END =0x5006 , // 电话结束

GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_IND = 0x5007, //

GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_CFM = 0x5008, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_END = 0x5009, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_SOURCE\_IND = 0x500a, // xxx

GAIA\_COMMAND\_STAROT\_START\_TRANS\_AUDIO\_IND = 0x500b, // xxx

GAIA\_COMMAND\_STAROT\_END\_TRANS\_AUDIO\_IND = 0x500c, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_DEVICE = 0X500D, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_DEVICE\_TRANS\_AUDIO\_REQUEST = 0X500E, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_ATTR = 0X500F, // xxx

GAIA\_COMMAND\_STAROT\_CALL\_ACTIVE = 0X5010, // 电话接通

GAIA\_COMMAND\_STAROT\_CALL\_INACTIVE = 0X5011, // 电话结束

};

///////////////////////////bond code///////////////////////////////

#define GAIA\_COMMAND\_STAROT\_SET\_BOND\_CODE\_IND (0x5100)

#define GAIA\_COMMAND\_STAROT\_CHECK\_BOND\_CODE\_IND (0x5101)

#define GAIA\_COMMAND\_STAROT\_SET\_DOUBLE\_CLICK\_SETTING 0X5104

#define GAIA\_COMMAND\_STAROT\_NOTIFY\_CASE\_STATUS 0X5105

///////////////////////////控制类命令///////////////////////////////

enum {

GAIA\_COMMAND\_STAROT\_CONTROL\_CALL\_DIALOG = 0X5400, // 拨打电话

GAIA\_COMMAND\_STAROT\_CONTROL\_ACCEPT\_DIALOG = 0X5401, // 接听电话

GAIA\_COMMAND\_STAROT\_CONTROL\_REJECT\_DIALOG = 0X5402, // 拒绝电话

GAIA\_COMMAND\_STAROT\_CONTROL\_PREVIOUS\_MUSIC= 0X5403, // 上一首音乐

GAIA\_COMMAND\_STAROT\_CONTROL\_NEXT\_MUSIC= 0X5404, // 下一首音乐

GAIA\_COMMAND\_STAROT\_CONTROL\_VOLUME\_SET= 0X5405, // 音量设置

};

///////////////////////////base info///////////////////////////////

enum {

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_VERSION = 0X5500, // 获取版本

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_DOUBLE\_CLIENT\_CONFIG= 0X5501, // 获取双击设置

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_DOUBLE\_CLIENT\_CONFIG= 0X5502, // 设置双击设置

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_NOTIFY\_CASE\_STATUS = 0X5503, // 通知充电盒状态

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_NOTIFY\_POWER\_POSITION\_CONNECTION = 0X5504, // 通知电量、位置、连接信息

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_APPGET\_POWER\_POSITION\_CONNECTION = 0X5505, // app获取电量、位置、连接信息

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_APOLLO\_WAKEUP\_ENB = 0X5506, // App设置语言唤醒是否使能

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_APOLLO\_WAKEUP\_ENB = 0X5507, // App获取语言唤醒是否使能

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_ADORN\_CHEAK\_ENB = 0X5508, // App设置佩戴检测是否使能

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_ADORN\_CHEAK\_ENB = 0X5509, // App获取佩戴检测是否使能

GAIA\_COMMAND\_STAROT\_BASE\_INFO\_ACTIVE\_DISCONNECT = 0X550A, // 设备主动断开连接

};

/////////////////////////////助手控制////////////////////////////////

enum {

GAIA\_COMMAND\_STAROT\_AI\_DEVICE\_REQUEST\_START = 0X5200, // 请求开始助手

GAIA\_COMMAND\_STAROT\_AI\_CONTROL = 0X5201, // 助手控制录音

GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_APP = 0X5202, // 助手音频Device->App

GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_DEVICE = 0X5203, // 助手音频App->Device

GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_DEVICE\_ACK = 0X5204, // 音频确认包

GAIA\_COMMAND\_STAROT\_AI\_BEGIN\_RECORD = 0X5205, // 开始录音

GAIA\_COMMAND\_STAROT\_AI\_END\_RECORD = 0X5206, // 停止录音

GAIA\_CONNECT\_STAROT\_RECORD\_STOP\_REPORT = 0X5207, // 强制停止录音

};

/////////////////////////////测试与生产///////////////////////////////

enum {

GAIA\_COMMAND\_STAROT\_TEST\_PRODUCT\_REST = 0X5600, // 恢复出厂设置

};

#define W16(x) (((\*(x)) << 8) | (\*((x) + 1)))

#define GAIA\_OFFS\_VENDOR\_ID (4)

#define GAIA\_OFFS\_COMMAND\_ID (6)

#define GAIA\_OFFS\_PAYLOAD\_LENGTH (3)

#define GAIA\_OFFS\_VERSION (1)

#define GAIA\_OFFS\_PAYLOAD (8)

enum GAIA\_TRANSFORM\_AUDIO\_STATUS {

GAIA\_TRANSFORM\_AUDIO\_IDLE,

GAIA\_TRANSFORM\_AUDIO\_ING,

GAIA\_TRANSFORM\_AUDIO\_WAIT\_MORE\_SPACE,

};

enum GAIA\_AUDIO\_TRANSFORM\_FLAG {

TRANSFORM\_NONE = 0,

TRANSFORM\_COMING = 1,

TRANSFORM\_CANT = 2,

DIALOG\_CAN\_TRANSFORM = GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_IND,

RECORD\_CAN\_TRANSFORM = GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_APP,

};

typedef struct {

uint16 command;

Source source;

uint8 \*data;

uint16 len;

uint16 audioType;

} GAIA\_STAROT\_AUDIO\_T;

typedef GAIA\_STAROT\_AUDIO\_T GAIA\_STAROT\_AUDIO\_IND\_T;

typedef GAIA\_STAROT\_AUDIO\_T GAIA\_STAROT\_AUDIO\_CFM\_T;

typedef struct {

uint16 command;

Source speark;

Source mic;

} GAIA\_STAROT\_DIALOG\_SOURCE;

typedef GAIA\_STAROT\_DIALOG\_SOURCE GAIA\_STAROT\_DIALOG\_SOURCE\_T;

#define CALL\_AUDIO\_IND(cmd) ((cmd) == GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_IND || (cmd) == GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_APP)

/////////////////////////////////////////////初始化与销毁/////////////////////////////////////////

void starotGaiaInit(void);

void starotGaiaReset(void);

////////////////////////////////////////////////////////////////////////////////////////////////

bool starotGaiaHandleCommand(GAIA\_STAROT\_IND\_T \*message);

void starotGaiaParseMessageMoreSpace(void);

bool starotGaiaSendAudio(GAIA\_STAROT\_AUDIO\_IND\_T \*message);

void starotNotifyAudioForward(bool st, uint8 flag);

void notifyGaiaDialogSource(Source speaker, Source mic);

uint8 starotGaiaTransGetAudioType(void);

void starotGaiaDefaultParse( MessageId id, Message message);

void starotGaiaParseCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m);

void starotGaiaSetTransportType(gaia\_transport\_type gaiaTransportType);

//-----------------------------------------------------

// 定义GAIA与UI、DSP关于电话时的消息

//-----------------------------------------------------

enum {

STAROT\_DIALOG\_STATUS = AV\_GAIA\_MESSAGE\_BASE + 0X20, // ui -> gaia 通话状态

STAROT\_DIALOG\_TYPE, // ui -> gaia 报告当前通话是电话还是WX其它通话

STAROT\_DIALOG\_USER\_ACCEPT\_RECORD, // gaia -> (ui & dsp) 用户请求录音

STAROT\_DIALOG\_AUDIO\_DATA, // dsp -> gaia 请求传输数据

STAROT\_DIALOG\_USER\_REJECT\_RECORD, // gaia -> (ui & dsp) 用户请求录音

STAROT\_DIALOG\_DEVICE\_REQUEST\_RECORD, // ui -> gaia 设备请求用户开始录音

STAROT\_DIALOG\_DEVICE\_REQUEST\_REJECT\_RECORD, // ui -> gaia 设备请求用户停止录音

STAROT\_DIALOG\_AUDIO\_DEVICE\_PAUSE, // ui -> gaia 音频设备发生切换 暂停

STAROT\_DIALOG\_AUDIO\_DEVICE\_CONTINUE, // ui -> gaia 音频设备发生切换 继续

STAROT\_DIALOG\_CALL\_ATTR\_TIMEOUT, // gaia -> gaia 属性发送超时，如果还在通话中，继续发送

STAROT\_DIALOG\_CALL\_BEGIN\_TIMEOUT, // gaia -> gaia 电话开始发送命令超时

STAROT\_DIALOG\_CALL\_END\_TIMEOUT, // gaia -> gaia 电话结束发送超时

STAROT\_DIALOG\_CASE\_STAT, // ui -> gaia 盒子当前信息

STAROT\_DIALOG\_CASE\_VER, // ui -> gaia 盒子当前版本

STAROT\_AI\_USER\_START\_RECORD, // ui -> (ui & dsp) AI请求录音

STAROT\_AI\_USER\_STOP\_RECORD, // ui -> (ui & dsp) AI停止录音

STAROT\_RECORD\_STOP\_STATUS\_REPORT, // dsp -> ui 上报停止录音状态

STAROT\_RECORD\_CALLIN\_STOP\_STATUS\_REPORT, // ui -> ui 拨打上报停止录音状态

STAROT\_RECORD\_CALLOUT\_STOP\_STATUS\_REPORT, // ui -> ui 接听上报停止录音状态

STAROT\_RECORD\_RETURN\_THREE\_POWER, // gaia -> ui App主动获取电量状态

STAROT\_NOTIFY\_STATUS, // ui -> gaia 状态发生变化

STAROT\_BASE\_INFO\_SET\_APOLLO\_WAKEUP\_ENB, // gaia -> ui App设置语言唤醒是否使能

STAROT\_BASE\_INFO\_SET\_ADORN\_CHEAK\_ENB, // gaia -> ui App设置是否佩戴使能

STAROT\_APP\_CONTROL\_PREVIOUS\_TRACK, // gaia -> ui App控制上一首

STAROT\_APP\_CONTROL\_NEXT\_TRACK, // gaia -> ui App控制下一首

};

#define STAROT\_COMMAND\_TIMEOUT 1000 // 命令超时时间

/////////////////////////////////////////命令，超时，重发//////////////////////////////////////////

struct StarotResendCommand\_T {

uint16 command;

uint16 len; /// payload的长度

uint8 payload[4];

};

typedef struct StarotResendCommand\_T StarotResendCommand;

StarotResendCommand\* starotResendCommandInit(uint16 command, uint16 len, uint8\* payload);

StarotResendCommand\* starotResendCommandDo(StarotResendCommand\* resendCommand, bool stillNeedResend);

/////////////////////////////////////////////////////////////////////////////////////////////////

void gaiaNotifyAudioAcceptStatus(Task task, int command);

bool appGaiaIsConnectBySpp(void);

#endif // AV\_HEADSET\_GAIA\_STAROT\_H

#include <pmalloc.h>

#include "av\_headset\_log.h"

#include "av\_headset.h"

#include "av\_headset\_gaia\_starot.h"

#include "audio\_forward.h"

#include "tws/attr.h"

#include "tws/audio\_forward.h"

#include "tws/peer.h"

uint16 bufferSendUnit = 80;

#ifdef GAIA\_TEST

extern void gaiaClearDropAudioSize(void);

extern int gaiaGetDropAudioSize(void);

StarotAttr \*attrDecode(uint8 \*data, int len);

extern void appGaiaSendResponse(uint16 vendor\_id, uint16 command\_id, uint16 status,

uint16 payload\_length, uint8 \*payload);

extern bool appGaiaSendPacket(uint16 vendor\_id, uint16 command\_id, uint16 status,

uint16 payload\_length, uint8 \*payload);

static void gaiaParseDialogStatus(GAIA\_STAROT\_IND\_T \*message);

static void starotGaiaDialogStopTransport(GAIA\_STAROT\_IND\_T \*message);

static void starotGaiaDialogStartTransport(GAIA\_STAROT\_IND\_T \*message);

static void gaiaParseCaseStatVer(const GAIA\_STAROT\_IND\_T \*message);

static void gaiaGetHeadsetVer(GAIA\_STAROT\_IND\_T \*message);//APP主动获取耳机版本

static void gaiaGetDoubleClickSet(GAIA\_STAROT\_IND\_T \*message);//App获取设备的耳机的双击配置信息

static void gaiaSetDoubleClickSet(GAIA\_STAROT\_IND\_T \*message);//App设置设备的耳机的双击配置信息

static void gaiaAppSetWakeupParameter(GAIA\_STAROT\_IND\_T \*mess);

static void gaiaAppGetWakeupParameter(GAIA\_STAROT\_IND\_T \*mess);

static void gaiaAppSetWearParameter(GAIA\_STAROT\_IND\_T \*mess);

static void gaiaAppGetWearParameter(GAIA\_STAROT\_IND\_T \*message);

static void gaiaGetNotifyPowPositionConn(GAIA\_STAROT\_IND\_T \*message);//上报电量-位置-连接状态信息

static void gaiaAppGetNotifyPowPositionConncet(GAIA\_STAROT\_IND\_T \*message);//App主动获取电量-位置-连接状态信息

static void gaiaSetRequestRecord(GAIA\_STAROT\_IND\_T \*message, bool isBegin);//App请求录音

static void gaiaAssistantAwake(GAIA\_STAROT\_IND\_T \*message, uint8 type);//ui上报gaia助手唤醒消息

static void gaiaAssistantAudioAppDev(GAIA\_STAROT\_IND\_T \*message);//App播放录音

static void gaiaDevRecordStopInfo(GAIA\_STAROT\_IND\_T \*message);//接受设备传过来的停止信息

static void gaiaControlCallDialog(GAIA\_STAROT\_IND\_T \*mess);

static void gaiaControlAcceptDialog(GAIA\_STAROT\_IND\_T \*message);

static void gaiaControlRejectDialog(GAIA\_STAROT\_IND\_T \*message);

static void gaiaControlPreviousMusic(GAIA\_STAROT\_IND\_T \*message);

static void gaiaControlNextMusic(GAIA\_STAROT\_IND\_T \*message);

static void gaiaControlVolumeSet(GAIA\_STAROT\_IND\_T \*message);

static void gaiaSetBondCode(GAIA\_STAROT\_IND\_T \*message);

static void gaiaCheckBondCode(GAIA\_STAROT\_IND\_T \*message);

static void starotGaiaParseTestCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m);

static void starotGaiaParseAudioCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m);

static void starotSpeedSendIntervalParse(void);

static void gaiaTestProductRest(GAIA\_STAROT\_IND\_T \*message);

static void gaiaSendDialogActiveStatus(int command, uint8\* phone, int len);

struct GaiaStarotPrivateData\_T {

Source dialogSpeaker;

Source dialogMic;

int audioTransType;

int speakerDropNum;

int micDropNum;

gaia\_transport\_type gaiaTransportType; // 默认为0 == gaia\_transport\_none;

uint8 testSpeedIndex;

uint16 speedTestSendUnit;

};

static struct GaiaStarotPrivateData\_T gaiaStarotPrivateData;

void starotGaiaInit(void) {

memset(&gaiaStarotPrivateData, 0x00, sizeof(struct GaiaStarotPrivateData\_T));

gaiaStarotPrivateData.speedTestSendUnit = 81;

}

void starotGaiaReset(void) {

memset(&gaiaStarotPrivateData, 0x00, sizeof(struct GaiaStarotPrivateData\_T));

appGetGaia()->transformAudioFlag = TRANSFORM\_NONE;

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

appGetGaia()->needCycleSendAudio = 0;

}

static uint32 gaia\_dbg\_cnt = 0;

bool starotGaiaHandleCommand(GAIA\_STAROT\_IND\_T \*message) {

/// 内部消息处理

switch (message->command) {

case STAROT\_NOTIFY\_STATUS:

gaiaGetNotifyPowPositionConn(message);

break;

case STAROT\_DIALOG\_TYPE:

gaiaParseDialogStatus(message);

break;

case STAROT\_DIALOG\_STATUS:

gaiaParseDialogStatus(message);

break;

case STAROT\_DIALOG\_AUDIO\_DATA:

if (gaia\_dbg\_cnt++ % 100 == 0) {

DEBUG\_LOG("call STAROT\_DIALOG\_AUDIO\_DATA EVENT ");

}

starotGaiaSendAudio(NULL);

break;

}

/// 处理来自APP的消息

switch (message->command) {

case GAIA\_COMMAND\_STAROT\_FIRST\_COMMAND: {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

DEBUG\_LOG("call GAIA\_COMMAND\_STAROT\_FIRST\_COMMAND");

}

break;

case GAIA\_COMMAND\_STAROT\_START\_SEND\_TIMER: {

gaiaStarotPrivateData.testSpeedIndex = 0;

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

if (message->payloadLen > 0) {

gaiaStarotPrivateData.speedTestSendUnit = message->payload[0] + 1;

} else {

gaiaStarotPrivateData.speedTestSendUnit = 80 + 1;

}

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

appGetGaia()->needCycleSendAudio = 1;

DEBUG\_LOG("call GAIA\_COMMAND\_STAROT\_START\_SEND\_TIMER, unit is : %d", gaiaStarotPrivateData.speedTestSendUnit);

MessageSendLater(&appGetGaia()->gaia\_task, GAIA\_STAROT\_AUDIO\_INTERVAL, NULL, 10);

}

break;

case GAIA\_COMMAND\_STAROT\_STOP\_SEND\_TIMER: {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

DEBUG\_LOG("call GAIA\_COMMAND\_STAROT\_STOP\_SEND\_TIMER");

appGetGaia()->needCycleSendAudio = 0;

}

break;

case GAIA\_COMMAND\_STAROT\_CALL\_BEGIN | GAIA\_ACK\_MASK:

DEBUG\_LOG("GAIA\_COMMAND\_STAROT\_CALL\_BEGIN replay");

MessageCancelAll(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_BEGIN\_TIMEOUT);

break;

case GAIA\_COMMAND\_STAROT\_CALL\_END | GAIA\_ACK\_MASK:

DEBUG\_LOG("GAIA\_COMMAND\_STAROT\_CALL\_END replay");

MessageCancelAll(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_END\_TIMEOUT);

break;

case GAIA\_COMMAND\_STAROT\_CALL\_ATTR | GAIA\_ACK\_MASK:

DEBUG\_LOG("GAIA\_COMMAND\_STAROT\_CALL\_ATTR replay");

MessageCancelAll(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_ATTR\_TIMEOUT);

break;

/// APP希望接受耳机的音频

case GAIA\_COMMAND\_STAROT\_START\_TRANS\_AUDIO\_IND:

starotGaiaDialogStartTransport(message);

break;

/// APP希望停止接受音频或压根从头不想要音频数据

case GAIA\_COMMAND\_STAROT\_END\_TRANS\_AUDIO\_IND:

starotGaiaDialogStopTransport(message);

break;

case GAIA\_COMMAND\_STAROT\_SET\_BOND\_CODE\_IND:

gaiaSetBondCode(message);

break;

case GAIA\_COMMAND\_STAROT\_CHECK\_BOND\_CODE\_IND:

gaiaCheckBondCode(message);

break;

}

/// 版本信息，状态，双击配置

switch (message->command) {

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_DOUBLE\_CLIENT\_CONFIG:

gaiaGetDoubleClickSet(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_DOUBLE\_CLIENT\_CONFIG:

gaiaSetDoubleClickSet(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_VERSION:

gaiaGetHeadsetVer(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_APPGET\_POWER\_POSITION\_CONNECTION:

gaiaAppGetNotifyPowPositionConncet(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_APOLLO\_WAKEUP\_ENB:

gaiaAppSetWakeupParameter(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_APOLLO\_WAKEUP\_ENB:

gaiaAppGetWakeupParameter(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_SET\_ADORN\_CHEAK\_ENB:

gaiaAppSetWearParameter(message);

break;

case GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_ADORN\_CHEAK\_ENB:

gaiaAppGetWearParameter(message);

break;

}

/// app控制耳机，发送相应蓝牙指令

switch (message->command) {

case GAIA\_COMMAND\_STAROT\_CONTROL\_CALL\_DIALOG:

gaiaControlCallDialog(message);

break;

case GAIA\_COMMAND\_STAROT\_CONTROL\_ACCEPT\_DIALOG:

gaiaControlAcceptDialog(message);

break;

case GAIA\_COMMAND\_STAROT\_CONTROL\_REJECT\_DIALOG:

gaiaControlRejectDialog(message);

break;

case GAIA\_COMMAND\_STAROT\_CONTROL\_PREVIOUS\_MUSIC:

gaiaControlPreviousMusic(message);

break;

case GAIA\_COMMAND\_STAROT\_CONTROL\_NEXT\_MUSIC:

gaiaControlNextMusic(message);

break;

case GAIA\_COMMAND\_STAROT\_CONTROL\_VOLUME\_SET:

gaiaControlVolumeSet(message);

break;

}

/// 助手52NN

switch (message->command) {

case GAIA\_COMMAND\_STAROT\_AI\_DEVICE\_REQUEST\_START:

gaiaAssistantAwake(message, message->payload[0]);

break;

case GAIA\_COMMAND\_STAROT\_AI\_BEGIN\_RECORD:

gaiaSetRequestRecord(message, TRUE);

break;

case GAIA\_COMMAND\_STAROT\_AI\_END\_RECORD:

gaiaSetRequestRecord(message, FALSE);

break;

case GAIA\_COMMAND\_STAROT\_AI\_AUDIO\_TO\_DEVICE:

gaiaAssistantAudioAppDev(message);

break;

case GAIA\_CONNECT\_STAROT\_RECORD\_STOP\_REPORT:

gaiaDevRecordStopInfo(message);

break;

}

/// 测试与生产

switch (message->command) {

case GAIA\_COMMAND\_STAROT\_TEST\_PRODUCT\_REST:

gaiaTestProductRest(message);

break;

}

return TRUE;

}

/// 主要处理设备内部的消息

void starotGaiaDefaultParse(MessageId id, Message message) {

switch (id) {

case GAIA\_STAROT\_COMMAND\_IND:

starotGaiaHandleCommand((GAIA\_STAROT\_IND\_T \*) message);

break;

case GAIA\_STAROT\_MORE\_SPACE: {

DEBUG\_LOG("Call GAIA\_STAROT\_MORE\_SPACE");

starotGaiaParseMessageMoreSpace();

}

break;

case GAIA\_STAROT\_AUDIO\_INTERVAL:

starotSpeedSendIntervalParse();

break;

case STAROT\_DIALOG\_CALL\_BEGIN\_TIMEOUT:

/// 确定现在是否再通话中，如果在，需要继续发送

if (TRANSFORM\_NONE != appGetGaia()->transformAudioFlag) {

DEBUG\_LOG("recv STAROT\_DIALOG\_CALL\_BEGIN\_TIMEOUT, need retry send to app");

StarotResendCommand \*cmd = starotResendCommandDo((StarotResendCommand \*) message, TRUE);

MessageSendLater(appGetGaiaTask(), id, cmd, STAROT\_COMMAND\_TIMEOUT);

}

break;

case STAROT\_DIALOG\_CALL\_END\_TIMEOUT:

if (TRANSFORM\_NONE == appGetGaia()->transformAudioFlag) {

DEBUG\_LOG("recv STAROT\_DIALOG\_CALL\_END\_TIMEOUT, need retry send to app");

if (NULL != appGetGaia()->transport) {

StarotResendCommand \*cmd = starotResendCommandDo((StarotResendCommand \*) message, TRUE);

MessageSendLater(appGetGaiaTask(), id, cmd, STAROT\_COMMAND\_TIMEOUT);

}

}

break;

case STAROT\_DIALOG\_CALL\_ATTR\_TIMEOUT:

if (TRANSFORM\_NONE != appGetGaia()->transformAudioFlag) {

DEBUG\_LOG("recv STAROT\_DIALOG\_CALL\_ATTR\_TIMEOUT, need retry send to app");

StarotResendCommand \*cmd = starotResendCommandDo((StarotResendCommand \*) message, TRUE);

MessageSendLater(appGetGaiaTask(), id, cmd, STAROT\_COMMAND\_TIMEOUT);

}

break;

case STAROT\_DIALOG\_CASE\_VER:

case STAROT\_DIALOG\_CASE\_STAT:

gaiaParseCaseStatVer(message);

break;

default:

DEBUG\_LOG("appGaiaMessageHandler Unknown GAIA message 0x%x (%d)", id, id);

break;

}

}

/\*

\* 使用事件去驱动让audio\_forward去发送数据

\*/

void starotGaiaParseMessageMoreSpace(void) {

if (appGetGaia()->nowSendAudioPhase != GAIA\_TRANSFORM\_AUDIO\_WAIT\_MORE\_SPACE) {

return;

}

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

if (appGetGaia()->needCycleSendAudio > 0) {

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

MessageSend(&appGetGaia()->gaia\_task, GAIA\_STAROT\_AUDIO\_INTERVAL, NULL);

} else {

/// 尝试使用messagemorespace这唯一的命令去让他发送消息

starotNotifyAudioForward(FALSE, 0);

}

}

void starotNotifyAudioForward(bool st, uint8 flag) {

UNUSED(st);

if (appGetGaia()->transformAudioFlag <= TRANSFORM\_NONE) {

return;

}

if (TRUE == st && flag > 0) {

if ((flag & GAIA\_AUDIO\_SPEAKER) > 0 && NULL != gaiaStarotPrivateData.dialogSpeaker) {

gaiaStarotPrivateData.speakerDropNum += bufferSendUnit;

SourceDrop(gaiaStarotPrivateData.dialogSpeaker, bufferSendUnit);

}

if ((flag & GAIA\_AUDIO\_MIC) > 0 && NULL != gaiaStarotPrivateData.dialogMic) {

gaiaStarotPrivateData.micDropNum += bufferSendUnit;

SourceDrop(gaiaStarotPrivateData.dialogMic, bufferSendUnit);

}

}

{

GAIA\_STAROT\_AUDIO\_IND\_T \*starot = PanicUnlessMalloc(sizeof(GAIA\_STAROT\_AUDIO\_IND\_T));

starot->command = STAROT\_DIALOG\_AUDIO\_DATA;

starot->data = NULL;

MessageSendLater(appGetGaiaTask(), GAIA\_STAROT\_COMMAND\_IND, starot, (TRUE == st ? 0 : 1));

}

}

bool starotGaiaSendAudio(GAIA\_STAROT\_AUDIO\_IND\_T \*message) {

UNUSED(message);

/// 不能传输，在source中缓存，如果缓存过多，会丢弃

//DEBUG\_LOG("transformAudioFlag: %d nowSendAudioPhase is %d", appGetGaia()->transformAudioFlag, appGetGaia()->nowSendAudioPhase);

if (appGetGaia()->transformAudioFlag < TRANSFORM\_CANT) {

return FALSE;

}

if (appGetGaia()->nowSendAudioPhase != GAIA\_TRANSFORM\_AUDIO\_IDLE) {

return FALSE;

}

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_ING;

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

static uint8 payload[256];

uint16 flag = 0, pos = 1;

if (NULL != gaiaStarotPrivateData.dialogSpeaker) {

int size = SourceSize(gaiaStarotPrivateData.dialogSpeaker);

if (size >= bufferSendUnit) {

flag |= GAIA\_AUDIO\_SPEAKER;

const uint8 \*ptr = (const uint8 \*) SourceMap(gaiaStarotPrivateData.dialogSpeaker);

memcpy(payload + pos, ptr, bufferSendUnit);

pos += bufferSendUnit;

}

}

if (NULL != gaiaStarotPrivateData.dialogMic) {

int size = SourceSize(gaiaStarotPrivateData.dialogMic);

if (size >= bufferSendUnit) {

flag |= GAIA\_AUDIO\_MIC;

const uint8 \*ptr = SourceMap(gaiaStarotPrivateData.dialogMic);

memcpy(payload + pos, ptr, bufferSendUnit);

pos += bufferSendUnit;

}

}

payload[0] = ((uint8) flag) | ((gaiaStarotPrivateData.testSpeedIndex & 0x0F) << 4);

gaiaStarotPrivateData.audioTransType = flag;

if (flag <= 0) {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

return FALSE;

}

bool st = appGaiaSendPacket(GAIA\_VENDOR\_STAROT, appGetGaia()->transformAudioFlag, 0xfe, pos, payload);

if (TRUE == st) {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_ING;

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

} else {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_WAIT\_MORE\_SPACE;

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

//// 分配内存失败，需要使用定时器/数据驱动

//DEBUG\_LOG("send data failed, wait more memory");

}

return TRUE;

}

void notifyGaiaDialogSource(Source speaker, Source mic) {

gaiaStarotPrivateData.dialogSpeaker = speaker;

gaiaStarotPrivateData.dialogMic = mic;

}

uint8 starotGaiaTransGetAudioType(void) {

return gaiaStarotPrivateData.audioTransType;

}

void gaiaParseDialogStatus(GAIA\_STAROT\_IND\_T \*message) {

uint8 status = message->payload[0];

uint8 dialogIn = 0X01, dialogOut = 0X02, dialogActive = 0X04, dialogInActive = 0X08;

DEBUG\_LOG("call STAROT\_DIALOG\_STATUS STATUS IS %02x", status);

StarotAttr \*head = NULL;

char temp[128] = {0};

int k = 0;

for (k = 0; k < message->payloadLen; ++k) {

sprintf(temp + k \* 2, "%02X", message->payload[k]);

}

DEBUG\_LOG("call gaiaParseDialogStatus");

/// 电话接入,只想通知一次APP电话来的消息

if ((((status & dialogIn) > 0) && ((appGetGaia()->dialogStatus & dialogIn) < 1))

|| (((status & dialogOut) > 0) && ((appGetGaia()->dialogStatus & dialogOut) < 1))) {

DEBUG\_LOG("Send GAIA\_COMMAND\_STAROT\_CALL\_BEGIN");

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_CALL\_BEGIN, 0xfe, 0, NULL);

// StarotResendCommand \*resend = starotResendCommandInit(GAIA\_COMMAND\_STAROT\_CALL\_BEGIN, 0, 0);

// MessageSendLater(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_BEGIN\_TIMEOUT, resend, STAROT\_COMMAND\_TIMEOUT);

StarotAttr \*attr = attrMalloc(&head, 1);

attr->attr = 0X02;

attr->payload[0] = ((status & dialogOut) > 0) ? 0X02 : 0X01;

appGetGaia()->transformAudioFlag = TRANSFORM\_COMING;

appGetGaia()->dialogStatus = appGetGaia()->dialogStatus | dialogIn | dialogOut;

}

/// 通话类型

if (message->command == STAROT\_DIALOG\_TYPE) {

StarotAttr \*attr = attrMalloc(&head, 1);

attr->attr = 0X04;

attr->payload[0] = (status > 0) ? 0X01 : 0X02;

}

/// 电话挂断

bool needSendEnd = FALSE;

if ((0 == status) || (status & dialogInActive) > 0) {

DEBUG\_LOG("Send GAIA\_COMMAND\_STAROT\_CALL\_END");

{

StarotAttr \*attr = attrMalloc(&head, 4);

attr->attr = 0X06;

int num = gaiaGetDropAudioSize();

attr->payload[0] = (uint8)((num >> 0) & 0X00FF);

attr->payload[1] = (uint8)((num >> 8) & 0X00FF);

attr->payload[2] = (uint8)((num >> 16) & 0X00FF);

attr->payload[3] = (uint8)((num >> 24) & 0X00FF);

}

// {

// StarotAttr \*attr = attrMalloc(&head, 1);

// attr->attr = 0X03;

// }

needSendEnd = TRUE;

if (appGetGaia()->transformAudioFlag > TRANSFORM\_CANT) {

/// todo 建议放到知道电话彻底结束的地方调用，需要考虑多方会话的情况

disable\_audio\_forward(TRUE);

DEBUG\_LOG("call disable\_audio\_forward(TRUE);");

}

appGetGaia()->dialogStatus = 0;

appGetGaia()->transformAudioFlag = TRANSFORM\_NONE;

}

/// 电话号码

uint8\* phoneInfo = NULL;

int phoneLen = 0;

if (message->payloadLen > 2) {

phoneInfo = message->payload + 2;

phoneLen = message->payloadLen - 2;

StarotAttr \*attr = attrMalloc(&head, phoneLen);

attr->attr = 0X01;

memcpy(attr->payload, phoneInfo, phoneLen);

}

/// 发送通话属性

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_CALL\_ATTR, 0xfe, len, data);

// StarotResendCommand \*resend = starotResendCommandInit(GAIA\_COMMAND\_STAROT\_CALL\_ATTR, len, data);

// MessageSendLater(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_ATTR\_TIMEOUT, resend, STAROT\_COMMAND\_TIMEOUT);

attrFree(head, data);

}

/// 电话接通

if ((status & dialogActive) > 0) {

gaiaSendDialogActiveStatus(GAIA\_COMMAND\_STAROT\_CALL\_ACTIVE, phoneInfo, phoneLen);

} else if ((status & dialogInActive) > 0) {

gaiaSendDialogActiveStatus(GAIA\_COMMAND\_STAROT\_CALL\_INACTIVE, phoneInfo, phoneLen);

}

if (TRUE == needSendEnd) {

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_CALL\_END, 0xfe, 0, NULL);

// StarotResendCommand \*resend = starotResendCommandInit(GAIA\_COMMAND\_STAROT\_CALL\_END, 0, 0);

// MessageSendLater(appGetGaiaTask(), STAROT\_DIALOG\_CALL\_END\_TIMEOUT, resend, STAROT\_COMMAND\_TIMEOUT);

}

}

void gaiaNotifyAudioAcceptStatus(Task task, int command) {

GAIA\_STAROT\_IND\_T \*starot = PanicUnlessNew(GAIA\_STAROT\_IND\_T);

starot->command = command;

starot->payloadLen = 0;

MessageSend(task, GAIA\_STAROT\_COMMAND\_IND, starot);

}

void starotSpeedSendIntervalParse(void) {

if (0 == appGetGaia()->needCycleSendAudio) {

return;

}

// DEBUG\_LOG("call starotSpeedSendIntervalParse, nowSendAudioPhase: %d", appGetGaia()->nowSendAudioPhase);

if (appGetGaia()->nowSendAudioPhase != GAIA\_TRANSFORM\_AUDIO\_IDLE) {

return;

}

static uint8 data[201];

for (int i = 1; i <= 1; ++i) {

data[0] = i;

data[1] = gaiaStarotPrivateData.testSpeedIndex;

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_CALL\_AUDIO\_IND, 0xfe, gaiaStarotPrivateData.speedTestSendUnit, data);

// DEBUG\_LOG("now send speed index is %02x", testSpeedIndex);

}

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_ING;

//DEBUG\_LOG("now send audio is : %d", appGetGaia()->nowSendAudioPhase);

// MessageSendLater(&appGetGaia()->gaia\_task, GAIA\_STAROT\_AUDIO\_INTERVAL, NULL, 10);

}

// { 命令重新发送

#define MAKE\_RESEND\_COMMAND\_WITH\_LEN(TYPE, LEN) TYPE \*message = (TYPE \*) PanicUnlessMalloc(((sizeof(TYPE) + LEN) / 4 \* 4) + ((sizeof(TYPE) + LEN) % 4 > 0 ? 4 : 0));

StarotResendCommand \*starotResendCommandInit(uint16 command, uint16 len, uint8 \*payload) {

MAKE\_RESEND\_COMMAND\_WITH\_LEN(StarotResendCommand, len);

message->command = command;

message->len = len;

if (len > 0) {

memcpy(message->payload, payload, len);

}

return message;

}

StarotResendCommand \*starotResendCommandDo(StarotResendCommand \*resendCommand, bool stillNeedResend) {

if (NULL == resendCommand) {

return NULL;

}

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, resendCommand->command, 0xfe,

resendCommand->len, resendCommand->payload);

if (TRUE == stillNeedResend) {

return starotResendCommandInit(resendCommand->command, resendCommand->len, resendCommand->payload);

}

return NULL;

}

// }

//向app发送盒子状态版本

void gaiaParseCaseStatVer(const GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*head = NULL;

StarotAttr \*attr = NULL;

if (message->command == STAROT\_DIALOG\_CASE\_STAT) {

DEBUG\_LOG("call STAROT\_DIALOG\_CASE\_STAT");

attr = attrMalloc(&head, 4);

attr->attr = 0X01;

attr->payload[0] = message->payload[0];

attr->payload[1] = message->payload[1];

attr->payload[2] = message->payload[2];

attr->payload[3] = message->payload[3];

}

if (message->command == STAROT\_DIALOG\_CASE\_VER) {

DEBUG\_LOG("call STAROT\_DIALOG\_CASE\_VER");

attr = attrMalloc(&head, 4);

attr->attr = 0X02;

memcpy(&head->payload[0], &message->payload[0], 2);

memcpy(&head->payload[2], &message->payload[2], 2);

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

// DEBUG\_LOG("len is :%d %p", len, data);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_NOTIFY\_CASE\_STATUS, 0xfe, len, data);

attrFree(head, data);

DEBUG\_LOG("call STAROT\_DIALOG\_CASE\_STAT");

}

}

// APP主动获取盒子耳机版本

void gaiaGetHeadsetVer(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*body = attrDecode(message->payload, message->payloadLen);

if (NULL == body) {

return;

}

StarotAttr \*head = NULL;

StarotAttr \*attr = NULL;

if ((body->payload[0] & 0X01) > 0) { // LEFT

attr = attrMalloc(&head, 8);

attr->attr = 0X01;

uint8 buffer[8] = {0};

SystemGetVersion(DEV\_LEFT, buffer);

memcpy(attr->payload, buffer, 8);

}

if ((body->payload[0] & 0X02) > 0) {

attr = attrMalloc(&head, 8);

attr->attr = 0X02;

uint8 buffer[8] = {0};

SystemGetVersion(DEV\_RIGHT, buffer);

memcpy(attr->payload, buffer, 8);

}

if ((body->payload[0] & 0X04) > 0) {

attr = attrMalloc(&head, 8);

attr->attr = 0X04;

uint8 buffer[8] = {0};

SystemGetVersion(DEV\_CASE, buffer);

memcpy(attr->payload, buffer, 8);

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_VERSION, GAIA\_STATUS\_SUCCESS, len, data);

attrFree(head, data);

} else {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_BASE\_INFO\_GET\_VERSION, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

attrFree(body, NULL);

}

// ui主动上报电量-位置-连接状态信息

void gaiaGetNotifyPowPositionConn(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*head = NULL;

StarotAttr \*attr = NULL;

if (message->payload[0] >= 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X01;

attr->payload[0] = message->payload[0];

}

if (message->payload[1] >= 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X02;

attr->payload[0] = message->payload[1];

}

if (message->payload[2] >= 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X03;

attr->payload[0] = message->payload[2];

}

if (message->payload[3] >= 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X04;

attr->payload[0] = message->payload[3];

}

if (message->payload[4] >= 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X05;

attr->payload[0] = message->payload[4];

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_BASE\_INFO\_NOTIFY\_POWER\_POSITION\_CONNECTION, 0xfe, len, data);

attrFree(head, data);

}

}

// App主动获取电量-位置-连接状态信息

void gaiaAppGetNotifyPowPositionConncet(GAIA\_STAROT\_IND\_T \*message) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_RECORD\_RETURN\_THREE\_POWER);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// App获取设备的耳机的双击配置信息

void gaiaGetDoubleClickSet(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*body = attrDecode(message->payload, message->payloadLen);

if (NULL == body) {

return;

}

StarotAttr \*head = NULL;

StarotAttr \*attr = NULL;

if ((body->payload[0] & 0X01) > 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X01;

UserGetKeyFunc(&attr->payload[0], 0);

}

if ((body->payload[0] & 0X02) > 0) {

attr = attrMalloc(&head, 1);

attr->attr = 0X02;

UserGetKeyFunc(0, &attr->payload[0]);

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, len, data);

attrFree(head, data);

} else {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

attrFree(body, NULL);

}

extern void appPeerSigTxDoubleClickConfigRequest(Task task, const bdaddr \*peer\_addr, uint8 left, uint8 right);

// App设置设备的耳机的双击配置信息

void gaiaSetDoubleClickSet(GAIA\_STAROT\_IND\_T \*message) {

bdaddr peer\_addr;

StarotAttr \*body = attrDecode(message->payload, message->payloadLen);

if (NULL == body) {

return;

}

uint8 leftKey = 0XFF, rightKey = 0XFF;

StarotAttr \*head = body;

while (NULL != head) {

if (0X01 == head->attr) {

leftKey = head->payload[0];

} else if (0X02 == head->attr) {

rightKey = head->payload[0];

}

head = head->next;

}

if ((0XFF != leftKey) || (0XFF != rightKey)) {

UserSetKeyFunc(leftKey, rightKey);

}

if (appDeviceGetPeerBdAddr(&peer\_addr)) {

appPeerSigTxDoubleClickConfigRequest(appGetGaiaTask(), &peer\_addr, gUserParam.lKeyFunc, gUserParam.rKeyFunc);

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

attrFree(body, NULL);

}

void gaiaAppSetWakeupParameter(GAIA\_STAROT\_IND\_T \*mess) {

StarotAttr \*pAttr = attrDecode(mess->payload, mess->payloadLen);

if (NULL == pAttr) {

return;

}

StarotAttr \* head = pAttr;

DEBUG\_LOG("gaiaAppSetWakeupParameter");

// 1 2 3

// enable(1) + type(1) + time(4)

MAKE\_GAIA\_MESSAGE\_WITH\_LEN(APP\_STAROT\_WAKEUP\_CONFIG\_IND, 6);

message->command = STAROT\_BASE\_INFO\_SET\_APOLLO\_WAKEUP\_ENB;

message->messageFrom = MESSAGE\_FROM\_APP;

while (NULL != pAttr) {

if (0X01 == pAttr->attr) {

message->apollo\_enable = pAttr->payload[0];

} else if (0X02 == pAttr->attr) {

message->assistant\_type = pAttr->payload[0];

} else if (0X03 == pAttr->attr) {

memcpy(&(message->timestamp), pAttr->payload, pAttr->len - 1);

}

pAttr = pAttr->next;

}

MessageSend(appGetUiTask(), GAIA\_STAROT\_COMMAND\_IND, message);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, mess->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

attrFree(head, NULL);

}

void gaiaAppGetWakeupParameter(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*head = NULL;

DEBUG\_LOG("gaiaAppGetWakeupParameter");

{

StarotAttr \*attr = attrMalloc(&head, 1);

attr->attr = 0X01;

attr->payload[0] = gUserParam.apolloEnable;

}

{

StarotAttr \*attr = attrMalloc(&head, 1);

attr->attr = 0X02;

attr->payload[0] = gUserParam.assistantType;

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, len, data);

attrFree(head, data);

}

}

void gaiaAppSetWearParameter(GAIA\_STAROT\_IND\_T \*mess){

StarotAttr \*pAttr = attrDecode(mess->payload, mess->payloadLen);

if (NULL == pAttr) {

return;

}

StarotAttr \*head = pAttr;

DEBUG\_LOG("gaiaAppSetWearParameter");

MAKE\_GAIA\_MESSAGE\_WITH\_LEN(APP\_STAROT\_WEAR\_CONFIG\_IND, 0);

message->command = STAROT\_BASE\_INFO\_SET\_ADORN\_CHEAK\_ENB;

while (NULL != pAttr) {

if (0X01 == pAttr->attr) {

message->wear\_enable = pAttr->payload[0];

} else if (0X02 == pAttr->attr) {

memcpy((uint8\*)(&(message->timestamp)), pAttr->payload, pAttr->len - 1);

}

pAttr = pAttr->next;

}

MessageSend(appGetUiTask(), GAIA\_STAROT\_COMMAND\_IND, message);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, mess->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

attrFree(head, NULL);

}

void gaiaAppGetWearParameter(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*head = NULL;

DEBUG\_LOG("gaiaAppGetWearParameter");

{

StarotAttr \*attr = attrMalloc(&head, 1);

attr->attr = 0X01;

attr->payload[0] = gUserParam.sensorEnable;

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, len, data);

attrFree(head, data);

}

}

// App请求录音

void gaiaSetRequestRecord(GAIA\_STAROT\_IND\_T \*message, bool isBegin) {

if (TRUE == isBegin) {

if (TRANSFORM\_NONE == appGetGaia()->transformAudioFlag) {

appGetGaia()->transformAudioFlag = RECORD\_CAN\_TRANSFORM;

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_AI\_USER\_START\_RECORD);

}

} else if (FALSE == isBegin) {

if (RECORD\_CAN\_TRANSFORM == appGetGaia()->transformAudioFlag) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_AI\_USER\_STOP\_RECORD);

appGetGaia()->transformAudioFlag = TRANSFORM\_NONE;

}

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

forwardSetDataClient(DATA\_CLIENT\_GAIA);

}

void gaiaAssistantAwake(GAIA\_STAROT\_IND\_T \*message, uint8 type)

{

(void)message;

StarotAttr \*head = NULL;

DEBUG\_LOG("gaiaAssistantAwake");

{

StarotAttr \*attr = NULL;

attr = attrMalloc(&head, 1);

attr->attr = 0X01;

attr->payload[0] = 0X01;

}

{

StarotAttr \*attr = NULL;

attr = attrMalloc(&head, 1);

attr->attr = 0X02;

attr->payload[0] = type;

}

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_AI\_DEVICE\_REQUEST\_START, 0xfe, len, data);

attrFree(head, data);

DEBUG\_LOG("call GAIA\_COMMAND\_STAROT\_AI\_DEVICE\_REQUEST\_START");

}

}

void gaiaAssistantAudioAppDev(GAIA\_STAROT\_IND\_T \*message) {

//App播放音频数据

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// 接受设备传过来的停止信息

void gaiaDevRecordStopInfo(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*head = NULL;

StarotAttr \*attr = NULL;

DEBUG\_LOG("gaiaDevRecordStopInfo");

attr = attrMalloc(&head, 1);

attr->attr = 0X01;

attr->payload[0] = message->payload[0];

if (NULL != head) {

uint16 len = 0;

uint8 \*data = attrEncode(head, &len);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_CONNECT\_STAROT\_RECORD\_STOP\_REPORT, 0xfe, len, data);

attrFree(head, data);

DEBUG\_LOG("call GAIA\_CONNECT\_STAROT\_RECORD\_STOP\_REPORT");

}

}

// APP中拨打电话

void gaiaControlCallDialog(GAIA\_STAROT\_IND\_T \*mess) {

StarotAttr \*body = attrDecode(mess->payload, mess->payloadLen);

if (NULL == body) {

return;

}

if (0X01 == body->attr) {

uint16 length = 0;

/// 电话号码长度不会大于16

const int MaxPhoneNuberLen = 16;

if (body->len <= MaxPhoneNuberLen) {

length = body->len;

} else {

length = MaxPhoneNuberLen;

}

MAKE\_GAIA\_MESSAGE\_WITH\_LEN(GAIA\_STAROT\_IND, length);

message->command = GAIA\_COMMAND\_STAROT\_CONTROL\_CALL\_DIALOG;

message->payloadLen = length;

memcpy(message->payload, body->payload, length);

MessageSend(appGetUiTask(), GAIA\_STAROT\_COMMAND\_IND, message);

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, mess->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

attrFree(body, NULL);

}

// APP中接听电话

void gaiaControlAcceptDialog(GAIA\_STAROT\_IND\_T \*message) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), GAIA\_COMMAND\_STAROT\_CONTROL\_ACCEPT\_DIALOG);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// APP中拒接电话

void gaiaControlRejectDialog(GAIA\_STAROT\_IND\_T \*message) {

// appHfpCallReject();

gaiaNotifyAudioAcceptStatus(appGetUiTask(), GAIA\_COMMAND\_STAROT\_CONTROL\_REJECT\_DIALOG);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// APP控制上一首

void gaiaControlPreviousMusic(GAIA\_STAROT\_IND\_T \*message) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_APP\_CONTROL\_PREVIOUS\_TRACK);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// APP控制下一首

void gaiaControlNextMusic(GAIA\_STAROT\_IND\_T \*message) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_APP\_CONTROL\_NEXT\_TRACK);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

void gaiaControlVolumeSet(GAIA\_STAROT\_IND\_T \*message) {

StarotAttr \*body = attrDecode(message->payload, message->payloadLen);

if (NULL == body) {

return;

}

if (0X01 == body->attr) {

//

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

static void gaiaSetBondCode(GAIA\_STAROT\_IND\_T \*message) {

if (!appBleIsBond()) {

uint8 \*data = message->payload;

uint16 advCode = (((uint16) data[0]) << 8) | data[1];

uint32 bindCode = (((uint32) data[2]) << 24) | (((uint32) data[3]) << 16) | (((uint32) data[4]) << 8) | data[5];

appBleSetBond(advCode, bindCode);

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_RECORD\_RETURN\_THREE\_POWER);

GattManagerCancelWaitForRemoteClient();

/// 查找对方地址

bdaddr peer\_addr;

if (appDeviceGetPeerBdAddr(&peer\_addr)) {

appPeerSigTxBleConfigRequest(appGetGaiaTask(), &peer\_addr, advCode, (int) bindCode);

}

/// 取消超时如果没有发送bondCode断开连接的定时器

} else {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_NOT\_SUPPORTED, 0, NULL);

}

}

static void gaiaCheckBondCode(GAIA\_STAROT\_IND\_T \*message) {

do {

if (appBleIsBond()) {

uint8 \*data = message->payload;

uint32 bindCode = (((uint32) data[0]) << 24) | (((uint32) data[1]) << 16) | (((uint32) data[2]) << 8) | data[3];

if (bindCode == appBleGetBondCode()) {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_RECORD\_RETURN\_THREE\_POWER);

/// 取消超时如果没有发送bondCode断开连接的定时器

break;

}

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_NOT\_SUPPORTED, 0, NULL);

} while (0);

}

void starotGaiaParseCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m) {

if (appGetGaia()->needCycleSendAudio > 0) {

starotGaiaParseTestCfm(m);

} else {

starotGaiaParseAudioCfm(m);

}

}

void starotGaiaParseTestCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m) {

uint8 \*packet = m->packet;

uint16 vendor\_id = W16(packet + GAIA\_OFFS\_VENDOR\_ID);

uint16 command\_id = W16(packet + GAIA\_OFFS\_COMMAND\_ID);

if (!CALL\_AUDIO\_IND(command\_id) || vendor\_id != GAIA\_VENDOR\_STAROT) {

return;

}

if (FALSE == m->success) {

if (gaiaStarotPrivateData.gaiaTransportType == gaia\_transport\_rfcomm) {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_WAIT\_MORE\_SPACE;

//DEBUG\_LOG("now send audio is : %d: %d", appGetGaia()->nowSendAudioPhase);

} else {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

//DEBUG\_LOG("now send audio is : %d: %d", appGetGaia()->nowSendAudioPhase);

MessageSendLater(&appGetGaia()->gaia\_task, GAIA\_STAROT\_AUDIO\_INTERVAL, NULL, 1);

}

} else {

MessageSend(&appGetGaia()->gaia\_task, GAIA\_STAROT\_AUDIO\_INTERVAL, NULL);

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

//DEBUG\_LOG("now send audio is : %d: %d", appGetGaia()->nowSendAudioPhase);

gaiaStarotPrivateData.testSpeedIndex += 1;

}

}

void starotGaiaParseAudioCfm(const GAIA\_SEND\_PACKET\_CFM\_T \*m) {

uint8 \*packet = m->packet;

uint16 vendor\_id = W16(packet + GAIA\_OFFS\_VENDOR\_ID);

uint16 command\_id = W16(packet + GAIA\_OFFS\_COMMAND\_ID);

if (GAIA\_COMMAND\_STAROT\_CALL\_END == command\_id && vendor\_id == GAIA\_VENDOR\_STAROT && FALSE == m->success) {

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, GAIA\_COMMAND\_STAROT\_CALL\_END, 0xfe, 0, NULL);

return;

}

if (!CALL\_AUDIO\_IND(command\_id) || vendor\_id != GAIA\_VENDOR\_STAROT) {

return;

}

// DEBUG\_LOG("now send audio is : %d: command:%04x vendor:%04x status:%d", appGetGaia()->nowSendAudioPhase, command\_id, vendor\_id, m->success);

if (FALSE == m->success) {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

if (gaiaStarotPrivateData.gaiaTransportType == gaia\_transport\_rfcomm) {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_WAIT\_MORE\_SPACE;

//DEBUG\_LOG("now send audio is : %d: %d", appGetGaia()->nowSendAudioPhase);

} else {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

//DEBUG\_LOG("now send audio is : %d: %d", appGetGaia()->nowSendAudioPhase);

starotNotifyAudioForward(FALSE, 0);

}

} else {

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

gaiaStarotPrivateData.testSpeedIndex += 1;

starotNotifyAudioForward(TRUE, starotGaiaTransGetAudioType());

}

}

void starotGaiaSetTransportType(gaia\_transport\_type gaiaTransportType) {

gaiaStarotPrivateData.gaiaTransportType = gaiaTransportType;

}

void starotGaiaDialogStartTransport(GAIA\_STAROT\_IND\_T \*message) {

if (appGetGaia()->transformAudioFlag == TRANSFORM\_COMING) {

gaiaStarotPrivateData.testSpeedIndex = 0;

appGetGaia()->transformAudioFlag = DIALOG\_CAN\_TRANSFORM;

appGetGaia()->nowSendAudioPhase = GAIA\_TRANSFORM\_AUDIO\_IDLE;

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_DIALOG\_USER\_ACCEPT\_RECORD);

DEBUG\_LOG("call disable\_audio\_forward(FALSE);");

gaiaStarotPrivateData.audioTransType = 0;

gaiaClearDropAudioSize();

}

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command,

((appGetGaia()->transformAudioFlag & DIALOG\_CAN\_TRANSFORM) > 0 ? GAIA\_STATUS\_SUCCESS : GAIA\_STATUS\_INCORRECT\_STATE),

0, NULL);

}

void starotGaiaDialogStopTransport(GAIA\_STAROT\_IND\_T \*message) {

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command,

((appGetGaia()->transformAudioFlag == DIALOG\_CAN\_TRANSFORM) ? GAIA\_STATUS\_SUCCESS : GAIA\_STATUS\_INCORRECT\_STATE), 0, NULL);

if (appGetGaia()->transformAudioFlag == DIALOG\_CAN\_TRANSFORM) {

gaiaNotifyAudioAcceptStatus(appGetUiTask(), STAROT\_DIALOG\_USER\_REJECT\_RECORD);

appGetGaia()->transformAudioFlag = TRANSFORM\_COMING;

}

}

// { 测试与生产

void gaiaTestProductRest(GAIA\_STAROT\_IND\_T \*message) {

appSmFactoryReset();

// appPowerReboot();

/// todo 清空和设置

appGaiaSendResponse(GAIA\_VENDOR\_STAROT, message->command, GAIA\_STATUS\_SUCCESS, 0, NULL);

}

// }

void gaiaSendDialogActiveStatus(int command, uint8\* phone, int len) {

const uint8 CallerAttr = 0X01;

StarotAttr \*starotAttr = NULL;

if ((NULL != phone) && (len > 0)) {

StarotAttr \*attr = attrMalloc(&starotAttr, len);

attr->attr = CallerAttr;

memcpy(attr->payload, phone, len);

}

uint16 attrLen = 0;

uint8 \*attrData = attrEncode(starotAttr, &attrLen);

appGaiaSendPacket(GAIA\_VENDOR\_STAROT, command, 0xfe, attrLen, attrData);

attrFree(starotAttr, attrData);

}

bool appGaiaIsConnectBySpp(void)

{

return ((gaiaStarotPrivateData.gaiaTransportType == gaia\_transport\_rfcomm) ||

((gaiaStarotPrivateData.gaiaTransportType == gaia\_transport\_spp)));

}

#endif