Advanced Card Systems Ltd.
Card & Reader Technologies

ACR890
オールインワンモバイル
スマートカード端末

リファレンスマニュアル　V2.00

info@acs.com.hk
www.acs.com.hk

カタログ

1.0. 紹介 6

2.0. 特性 7

3.0. ACR890 Phase 1 コマンドと Phase 2 コマンドの比較 8

3.1. ACR890 デバイスを認識する。 10

4.0. ファイルとディレクトリ構造 11

5.0. キーパッドの API 12

5.1. キーパッドのファイルディスクリプタを開く 12
5.2. キーパッドのファイルディスクリプタを閉じる 12
5.3. 現在のキーボード状態を取得します。 13
5.4. 電源ボタンの動作モードをセットする 13
5.5. 電源ボタンの動作モードを取得する 14

6.0. バックライト制御の API 16

6.1. 現在のバックライトレベルを取得する 16
6.2. バックライトのラベルを設定する 17

7.0. 電池と充電器の API 18

7.1. 電池と充電器の状態を取得する 18

8.0. LED 制御の API 20

8.1. LED の状態を設定する 20
8.2. LED の点滅状態を取得する 21

9.0. GPRS モジュールの電源管理の API 22

9.1. GPRS モジュールを有効にする 22
9.2. GPRS モジュールを無効にする 22
9.3. pppd の接続パラメータを設定する 23
9.4. pppd ダイヤラーのパラメータを設定する 23
9.5. pppd のプロセスを起動する 24
9.6. pppd のプロセスをオフにする 24
9.7. 送信タイムアウトを設定する 26
9.8. 1 つの AT コマンドを送信する 26
9.9. IMEI シリアル番号を取得する 27

10.0. オーディオ（ALSA）の API 29

10.1. システムオーディオの音量を取得する 29
10.2. システムオーディオの音量を設定する 30

10.3. サウンドファイルを再生する 30
10.4. スピーカーサウンドコントロール 31

11.0. ファームウェアの API 32

11.1. ファームウェアのバージョン番号を取得する 32

12.0. サーマルプリンタの API 33

12.1. プリンタポートを開く 33
12.2. プリンタポートを閉じる 33
12.3. プリンターの状態を取得する 34
12.4. プリンタに用紙を供給する 34
12.5. プリンターがデータをプリンタアウトように準備します 35
12.6. データをプリンタアウトし始めます。 35
12.7. プリンターを設定する 36
12.8. 標準モードで文字列を印刷する 37
12.9. ビットマップファイルを印刷する 37
12.10. FreeType ライブラリを初期化する 39
12.11. 文字サイズの設定 39
12.12. 字体画像を取得して、プリンターする 40
12.13. FreeType ライブラリを開放する 40

13.0. 無線の LAN モジュールの API 42

13.1. 無線の LAN モジュールを有効する 42
13.2. 無線の LAN モジュールをオフする 42

14.0. ブルートゥースモジュールの API 43

14.1. ブルートゥースモジュールをオンにする 43
14.2. ブルートゥースモジュールをオフにする 43

15.0. スマートカードリーダーの API 44

15.1. スマートカードリーダーモジュールをオンにする 44
15.2. スマートカードリーダーモジュールをオフにする 44
15.3. スマートカード操作の PC / SC API 45

16.0. 磁気ストライプインターフェースの API 47

16.1. 磁気ストライプカードからトラックデータを取得する 47

17.0. エラーコードを説明する API 49

17.1. エラーコードの説明を取得する 49

18.0. パワーマネジメントの API 50

18.1. システムスリープタイムアウトを設定する 50
18.2. 現在のシステムスリープモードを取得する 50

18.3. システムスリープ時間を取得する 51

19.0. バーコードスキャン器の API 52

19.1. バーコードスキャン器モジュールをオンにする 52
19.2. バーコードスキャン器モジュールをオフにする 52
19.3. スキャンを接続する 52
19.4. データスキャン 52
19.5. スキャナの読み取りセッションを取得する 53
19.6. スキャナのファームウェアバージョンを取得する 53
19.7. スキャナの接続を切る 53

附录 A.ACR890 フェーズ 1 コマンド 54

附录 A.1. 非接触インターフェースの疑似 APDU コマンド 54
附录 A1.1. データを取得する（Get Data） 54
附录 A.2. MIFARE® Classic 1K/4K メモリカードの PICC コマンド（T=CL エミュレーション） 56
附录 A.2.1. 認証キーのダンロード（Load authentication keys） 56
附录 A.2.2. MIFARE Classic (1K/4K)カードに対しての認証(Authenticate MIFARE Classic (1K/4K)) 57
附录 A.2.3. バイナリブロックを読み取る（Read Binary Blocks） 61
附录 A.2.4. バイナリブロックの更新（Update Binary Blocks） 62
附录 A.2.5. 数値ブロックの操作（Value block operation）（Increment, Decrement, Store） 63
附录 A.2.6. 数値ブロックを読み取る（Read Value Block） 64
附录 A.2.7. 数値ブロックをコピーする（Copy Value Block） 65
附录 A.3. サーマルプリンタの API 67
附录 A.3.1. プリンターを再起動する 67
附录 A.3.2. 標準モードでラインスペースを設定する 67
附录 A.3.3. 標準モードで文字列を印刷する 68
附录 A.3.4. ページモードで文字列を印刷する 68
附录 A.3.5. 標準モードでデータ配列を印刷する 70
附录 A.3.6. ページモードでデータ配列を印刷する 70
附录 A.3.7. 画像を印刷する 71
附录 A.4. 接触インターフェイス（ICC）の API 72
附录 A.4.1. 接触インターフェイスモジュールをオンにする 72
附录 A.4.2. 接触インターフェイスモジュールをオフにする 72
附录 A.4.3. 接触カードが存在するかどうかを確認する 73
附录 A.4.4. 接触式スマートカードを電気入れる 74
附录 A.4.5. 接触式スマートカードの電源をオフにする 74
附录 A.4.6. 接触スマートカードに PPS を送信する 75
附录 A.4.7. 接触スマートカードの APDU の転送 75
附录 A.5. 非接触インターフェイス（PICC）の API 78
附录 A.5.1. 非接触インターフェイスモジュールをオンにする 78

附录 A.5.2. 非接触インターフェイスモジュールをオフにする 78
附录 A.5.3. 非接触式カードを読み取る 79
附录 A.5.4. 非接触式カードを活性化する 80
附录 A.5.5. 非接触式カードの活性化を取り消す 80
附录 A.5.6. 非接触式カードのデータを送信する 81
附录 A.5.7. FeliCa カードのデータを送信する 81
附录 A.5.8. 非接触アンテナをオン/オフにする 82
附录 A.6. INI ファイル解析ツールの API 86
附录 A.6.1 INI キーワードの値を取得する 86
附录 A.6.2. INI キーワードの値を設定する 87
附录 A.6.3. INI キーワードを追加する 88
附录 A.6.4. 附录/etc/config.ini 内のすべてのキーワードに応じて、ハードウェアの値を設定する 89
附录 A.6.5. 指定したキーワードに応じて、ハードウェアの値を設定する 90

チャートカタログ

表 1：ACR890 Phase 1 コマンドと Phase 2 コマンドの比較 9
表 2：ACR890 Phase 1 コマンドと Phase 2 コマンドの比較 10
表 3：トラックデータの状態ビットテーブル 48
表 4：MIFARE 1K カードのメモリマップ 58
表 5：MIFARE 4K カードのメモリマップ 59
表 6：MIFARE Ultralight® カードのメモリマップ 60

1.0. 紹介

ACR890 は、スマートカード、磁気ストライプや非接触技術を組み合わせた次世代の高性能モバイルスマートカード端末です。その高解像度のタッチスクリーンと、市場で入手可能な最も対話型インタフェースと機能を体験したいお客様に適しています。この最先端の製品は、より高速な処理速度、大容量のメモリや携帯性を提供しています。

本書は ACR890 端末専用の API（アプリケーションプログラミングインターフェース）を説明します。アプリケーションソフトウェアの開発者は、スマートカード関連のアプリケーションを開発するために、これらの API を利用することができます。

2.0. 特性

- 32 ビットの A8 プロセッサ、実行の内蔵的な Linux® OS
- 4 GB フラッシュメモリ + 256 MB　DDR RAM
- 拡張可能な Micro SD カードサポート（1 GB – 16 GB）
- 接続方式サポート：
  - Wi-Fi
  - GPRS/GSM クワッドバンド（850 MHz、900 MHz、1800 MHz、1900 MHz）
  - 3G（900 MHz/2100 MHz 或いは 850 MHz/1900 MHz）
  - USB クライアント（高速、Micro-B タイプのコネクタ）
  - RS-232 シリアル（Mini-B タイプのコネクタ)
- 接触式画面：
  - 1 フルサイズの接触カードスロット（ランディングコネクタ）
- 非接触画面：
  - 非接触スマートカードインターフェース
- 磁気ストライプサポート
- SAM インターフェース：
  - 二つの SAM カードスロット（コンタクトコネクタ）
- SIM インターフェース：
  - GPRS 機能のための標準な SIM カードスロット一つ
- バーコードスキャン器（オプション）
- ファームウェアアップグレード
- 内蔵されている周辺機器：
  - 読みやすい高解像度のカラーLCD
  - 3.5"抵抗式のタッチスクリーン LCD
  - 高耐久性、耐薬品性の 20 ボタンのキーパッド
  - サーマルプリンタ
  - 独立のバックアップバッテリとリアルタイムクロック（RTC）
  - 4 つの LED ステータスインジケータ
  - スピーカー内蔵
- 以下の基準に一致している：
  - ISO 7816
  - ISO 14443
  - ISO 7811
  - USB Full Speed
  - RoHS 2

3.0. ACR890 Phase 1 コマンドと Phase 2 コマンドの比較

ACR890 Phase 1 コマンドと Phase2 にて、デバイスのいくつかのコマンドの比較については、以下の表を参照してください。

注：これは使用可能なすべてのコマンドのリストではありません。コマンドが表にリストされていない場合、このコマンドは両方とも *ACR890* フェーズ *1* およびフェーズ *2* デバイスに適用されます。

コマンド	ACR890 フェーズ 1	ACR890 フェーズ 2
GPRS モジュール		
GPRS モジュールを有効にする	`Int gprs_power_on(void)`	`int gprs_power_on(unsigned char timeout)`
プリンターモジュール		
プリンターを再起動する	適用	未適用
プリンタに用紙を供給する	`int printer_page_feed(unsigned char nr_len)`	`int printer_page_feed(char cControl, char cLine)`
プリンターがデータをプリンタアウトように準備します	未適用	適用
標準モードでラインスペースを設定する	適用	未適用
データをプリンタアウトし始めます。	未適用	適用
ｂｍｐファイルをプリンタアウトします	未適用	適用
ページモードで文字列を印刷する	適用	未適用
標準モードで文字列を印刷する	`int printer_printStrSM(const char *str)`	`int printer_printStrSM(const char *pString, const int nLen, unsigned int Mode)`
標準モードでデータ配列を印刷する	適用	未適用
ページモードでデータ配列を印刷する	適用	未適用
画像を印刷する	`int printer_print_img(const unsigned char *bitmap, unsigned short width, unsigned short height, unsigned char mode);`	未適用
スマートカード操作		
接触カードが存在するかどうかを確認する	適用	**スマートカード操作の PC / SC API に変更する**
接触式のスマートカードをオンにする	適用	**スマートカード操作の PC / SC API に変更する**

コマンド	ACR890 フェーズ 1	ACR890 フェーズ 2
接触式のスマートカードをオフにする	適用	**スマートカード操作の PC / SC API に変更する**
接触スマートカードに PPS を送信する	適用	**スマートカード操作の PC / SC API に変更する**
接触スマートカードに APDU を送信する	適用	**スマートカード操作の PC / SC API に変更する**
非接触カードリーダーモジュールを開く	適用	**スマートカード操作の PC / SC API に変更する**
非接触カードリーダーモジュールを閉じる	適用	**スマートカード操作の PC / SC API に変更する**
非接触式カードを読み取る	適用	**スマートカード操作の PC / SC API に変更する**
非接触式カードを活性化する	適用	**スマートカード操作の PC / SC API に変更する**
非接触式カードの活性化を取り消す	適用	**スマートカード操作の PC / SC API に変更する**
非接触式カードのデータを送信する	適用	**スマートカード操作の PC / SC API に変更する**
FeliCa カードのデータを送信する	適用	**スマートカード操作の PC / SC API に変更する**
非接触アンテナをオン/オフにする	適用	**スマートカード操作の PC / SC API に変更する**
管理モジュールの電源をオンにする		
現在のシステムスリープ時間を取得する	未適用	適用
システムの自動スリープを無効/有効にする	適用	未適用

表 1：ACR890 Phase 1 コマンドと Phase 2 コマンドの比較

ACR890 フェーズ 1 デバイスに適用できるコマンドについては、**ACR890 フェーズ 1 コマンド**を参照してください。

3.1. ACR890 デバイスを認識する。

ACR890 デバイスのフェーズリリースは、背面の製品ラベルで確認できます。下記の表を参考にして、あなたが持っている ACR890 のフェーズを確認してください。

ACR890	シリアルナンバー	ファームウェアのバージョン番号
フェーズ 1	RR312 で始まる	1XXX
フェーズ 2	RR400 で始まる	2XXX

表 2：ACR890 Phase 1 コマンドと Phase 2 コマンドの比較

4.0. ファイルとディレクトリ構造

ファイルの名称	ファイルのアドレス	説明
acs_api.h	ホスト	API ヘッダー
acs_errno.h	ホスト	API から返されたエラー番号の定義
libacs_api.so	ターゲット	API の共有ライブラリ

5.0. キーパッドの API

このセクションでは、デバイスのキーパッドを構成するための API 関数について説明します。

5.1. キーパッドのファイルディスクリプタを開く

この関数は、キーパッドのファイル記述子を開くために使用されます。

```
int kpd_open()
```

パラメータ

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリ libacs_api.so を使用する

5.2. キーパッドのファイルディスクリプタを閉じる

この関数は、キーパッドのファイル記述子を閉じるために使用されます。

```
int kpd_close()
```

パラメータ

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリ libacs_api.so を使用する

5.3. 現在のキーボード状態を取得します。

この機能は、キーが押されるたびに押し状態とキーコード値を返すために使用されます。

```
int kpd_state_get(struct kPoint *keycode, unsigned int timeout)
```

パラメーター

```
struct kPoint {
        unsigned short type;
        unsigned short code;
};
```

keycode [out]　　押されたキーのキーコード。

timeout [in]　　待ち時間は（ミリ秒）押されたキーの有効なキーコードを取得します。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗またタイムアウトした場合、戻り値は-2 です。

それ以外の場合、戻り値は-1 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリ　　libacs_api.so を使用する

5.4. 電源ボタンの動作モードをセットする

この機能は、電源ボタンの動作モードを設定するために使用されています。

```
int pwrbtn_set_mode(enum pwrbtnMode nMode)
```

パラメーター

```
enum pwrbtnMode {
        CMD_TESTMODE=0,
        CMD_ONOFFMODE,
        CMD_FAIL
};
```

nMode [in]　　入力されるモードの値；

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリ libacs_api.so を使用する

5.5. 電源ボタンの動作モードを取得する

この機能は、電源ボタンの動作モードを取得するために使用されています。

```
int pwrbtn_get_mode (enum pwrbtnMode *pMode)
```

パラメ―タ―

```
enum pwrbtnMode {
      CMD_TESTMODE=0,
      CMD_ONOFFMODE,
      CMD_FAIL
};
```

pMode [out] メモリモードの値へのポインタ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリ libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   struct kPoint key_Point;
   enum pwrbtnMode mode = CMD_TESTMODE;
   enum pwrbtnMode m;

   ret = kpd_open();
```

```
    pwrbtn_get_mode(&m);       //obtain current powerkey working mode
    printf("m1 = %d\n",(int)m);

    pwrbtn_set_mode(mode);  //set current powerkey working mode to Test
    Mode

    pwrbtn_get_mode(&m); //obtain current powerkey working mode
    printf("m2 = %d\n",(int)m);

    ret = kpd_state_get(&key_Point,5000);  //read key press within 5s

    printf("Type:%d, Code:%d\n", key_Point.type, key_Point.code);

    mode = CMD_ONOFFMODE;
    pwrbtn_set_mode(mode);  //set  current  powerkey  working  mode  to
    PowerKey Mode

    pwrbtn_get_mode(&m); //obtain current powerkey working mode
    printf("m3 = %d\n",(int)m);

    ret = kpd_close();
    printf("ret = %d\n", ret);

    return 0;
}
```

6.0. バックライト制御の API

このセクションでは、バックライトを配置するための API 関数について説明します。

6.1. 現在のバックライトレベルを取得する

この関数は現在のバックライトレベルを返すするために使用されます。

```
int backlight_get(struct bl_state *stat)
```

パラメ―タ―

```
struct bl_state {
      int brightness;   //current user requested brightness level(0 -
      max_brightness)
      int max_brightness;// maximal brightness level
      int fb_power;  //current fb power mode (0: full on, 1..3: power
      saving; 4: full off)
      int actual_brightness;// actual brightness level
} ;
```

stat [out]　返されたバックライト状態へのポインタ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1/-2 です。

必要条件

ヘッダ　acs_api.h で宣言される

ライブラリファイル　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   struct bl_state state;

   ret = backlight_get(&state); //call api to get backlight state
   if(0 == ret)
   {//show out the backlight state you get just now.
      printf("brightness=%d.max_brightness=%d,fb_power=%d,actual_brightn
   ess=%d",
   state.brightness,state.max_brightness,state.fb_power,state.actual_bri
   ghtness);
   }
```

```
    return ret;
}
```

6.2. バックライトのラベルを設定する

この関数は、バックライトのラベルを設定するのに使用されます。

```
int backlight_set(enum bl_level level)
```

パラメーター

```
enum bl_level {
      BACKLIGHT_LEVEL_0 = 0,/* Turn off */
      BACKLIGHT_LEVEL_1,
      BACKLIGHT_LEVEL_2,
      BACKLIGHT_LEVEL_3,
      BACKLIGHT_LEVEL_4,
      BACKLEGHT_LEVEL_5,
      BACKLEGHT_LEVEL_6,
      BACKLEGHT_LEVEL_7,
      BACKLEGHT_LEVEL_8,
      BACKLEGHT_LEVEL_9,
      BACKLIGHT_LEVEL_MAX
};
```

level [in]　　指定されたバックライトのレベル。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1/-2 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
    int ret;
    enum bl_level level = BACKLIGHT_LEVEL_4;

    ret = backlight_set(level); //call api to set the level of backlight
    brightness.

    return ret;
}
```

7.0. 電池と充電器の API

このセクションでは、電池と充電器を配置するための API 関数について説明します。

7.1. 電池と充電器の状態を取得する

この関数は現在電池と充電器の状態を返すするために使用されます。電力管理 IC は動作状態じゃない場合、バッテリが検出されません。

```
int battery_state_get(struct battery_state *stat)
```

パラメーター

```
struct battery_state {
   int ifdc;//if have dc power   [0/1 = dc power absent/present]
   int ifbattery;//if have battery power  [0/1 = battery absent/present]
   int chargerstate;//charger state
      [0/1/2/3=discharging/charging/full]
   unsigned int batt_voltage; //battery voltage[uV]
   unsigned int batt_voltage_max; //battery max voltage[uV]
   unsigned int batt_voltage_min; //battery min voltage[uV]
   unsigned int batt_volpercent; //battery capacity [%]
   };
```

stat [out]　返されたバッテリーの状態情報。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は 0 より小さいです。

必要条件

ヘッダ　acs_api.h で宣言される

ライブラリファイル　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   struct battery_state stat;
   ret = bat_get_charger_state(&state); //call api to get battery and
   charger state
   if(ret == 0)
   //print the battery state you get just now.
      printf("ifdc = %d, ifbattery = %d, chargerstate = %d, batt_voltage
   = %d,       batt_voltage_max    =    %d,    batt_voltage_min    =    %d,
```

```
            batt_volpercent           =           %d\n",           state.ifdc,
            state.ifbattery,state.chargerstate,           state.batt_voltage,
            state.batt_voltage_max,                 state.batt_voltage_min,
            state.batt_volpercent);
        }
        return ret;
    }
```

8.0. LED 制御の API

このセクションでは、LED インジケータを配置するための API 関数について説明します。

8.1. LED の状態を設定する

この機能は個々の LED の状態を**オン/オフ/点滅**に設定するために使用されます。。

```
int led_set_state(enum led_id led, struct led_state stat)
```

パラメ―タ―

```
enum led_id {
   LED_ID_BLUE = 0,
   LED_ID_YELLOW,
   LED_ID_GREEN,
   LED_ID_RED,
   LED_ID_MAX
};
enum led_blink_state {
   LED_STATE_SOLID_OFF = 0,
   LED_STATE_SOLID_ON,
   LED_STATE_BLINK,
   LED_STATE_MAX
};
struct led_state {
enum led_blink_state bs;//led blink state
unsigned int on_time;   //led blink state on period time in ms
unsigned int off_time;  //led blink state off period time in ms
};
```

led [in] 指定された LED の ID 番号。

stat [in] 指定された LED の状態

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1/-2 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

8.2. LED の点滅状態を取得する

この関数は、指定された LED の現在の状態を返します。

```
int led_get_state(enum led_id led, struct led_state *stat)
```

パラメーター

led [in]　　LED の ID 番号。

stat [out]　　返された LED 状態へのポインタ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリ　　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   enum led_id led = LED_ID_BLUE; //get 0-blue led state
   struct led_state stat;
   int ret;

   memset(&stat, 0x00, sizeof(struct led_state));

   ret = led_get_state(led, &state); //call API to get led state
   if(0 == ret)
   {
      printf("led-d%,state=d%,ontime=%d,offtime=%d.\n",
      stat.bs, stat.on_time, stat.off_time);
   }
   else
   {
      printf(“Fail to get current led state, ret=%d\n”, ret);
   }
   stat.bs = LED_STATE_BLINK;
   stat.on_time = 100;
   stat.off_time = 900;
   //call API to set blue led blink on for 100ms and blink off for 900ms
   periodically.
   ret = led_set_state(led, stat);
   if(0 != ret)
   {
      printf( " Set led blink state failed !, ret = %d\n", ret);
   }

   return ret;
}
```

9.0. GPRS モジュールの電源管理の API

このセクションでは、GPRS モジュールを配置するための API 関数について説明します。

9.1. GPRS モジュールを有効にする

この機能は、GPRS モジュールをオンにするために使用されます。

```
int gprs_power_on(unsigned char timeout)
```

注：*ACR890 フェーズ 1 デバイス*を使っている場合、コマンドのフォーマットは：

int gprs_power_on(void)

パラメーター

timeout [in] */dev/ttyUSB2* を検出するための待機時間。タイムアウトしたり、*/ dev / ttyUSB2* が存在しない場合は、*EGPRS_POWER_ON_TIMEOUT* を返します。 */ dev / ttyUSB2* が見つかった場合は、EGPRS_SUCCEEDED を返します。

timeout の値が 0 で、*/ dev / ttyUSB2* を検出できない場合は、すぐに戻ります。

timeout の値が 0 でなければ、タイムアウトを≧5 に設定し、≤9 に設定できます。API は*/ dev / ttyUSB2* を検出します。

返ってきた数値

成功した場合、戻り値は *EGPRS_SUCCEEDED* です。

失敗した場合、戻り値は *ENODEV* または *EGPRS_POWER_ON_FAILED* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

9.2. GPRS モジュールを無効にする

この機能は、GPRS モジュールをオフにするために使用されます。

```
int gprs_power_off(void)
```

パラメーター

なし。

返ってきた数値

成功した場合、戻り値は *EGPRS_SUCCEEDED* です。

失敗した場合、戻り値は *ENODEV* または *EGPRS_POWER_OFF_FAILED* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

9.3. pppd の接続パラメ—タ—を設定する

この機能は、電話、ローカル IP、リモート IP、およびネットマスクなどの *pppd* パラメータを設定することに使用されます。

```
int set_ppp_param(char *telephone, char *local_ip, char *remote_ip, char
*netmask).
```

パラメ—タ—

telephone [in] ダイヤルアップネットワークの電話番号（例えば*99***1#）。

local_ip *in]* 既知の場合は、ローカル IP のアドレス（がダイナミック=0.0.0.0）。

remote_ip *[in]* 必要に応じて、リモート IP のアドレス（通常は 0.0.0.0）。

netmask *[in]* 適切なネットマスク（必要であれば）。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

9.4. pppd ダイヤラーのパラメ—タ—を設定する

この機能は、プロトコルおよびログインポイントとしてダイヤラパラメータを設定するために使用されます。

```
int set_dialer_param(char *protocol , char *login_point).
```

パラメ—タ—

protocol[in] 通信プロトコル（例えば IP）。

login_point *[in]*　　モバイルネットワークオペレータのサポートの APN

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

9.5. pppd のプロセスを起動する

この機能は、pppd のダイヤルプロセスを起動するために使用されます。

```
void ppp_on(void ).
```

パラメ―タ―

なし。

返ってきた数値

なし。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

9.6. pppd のプロセスをオフにする

この機能は、pppd のダイヤルプロセスをオフにするために使用されます。

```
void ppp_off(void ).
```

パラメ―タ―

なし。

返ってきた数値

なし。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
/* Tips :After you finish using ppp_on() connected the Internet, please
execute  ppp_off()  to  disconnect  Internet,  and  finally  execute
gprs_power_off() to turnoff 3g modules;*/

int main(int argc, char *argv[])
{
   int ret=0;
   int count = 0;

   ret = gprs_power_on();
   if(ret != EGPRS_SUCCEEDED)
   {
      printf("gprs power on failed, ret = %d\n”,ret);
      return -1;
   }

   /* notice:After poweron 3g module, must wait for 9s, and then check
if ‘/dev/ttyUSB2’ exist */
   sleep(9);
   if(access(“/dev/ttyUSB2”,0) != 0)
   {
      printf(“no exist /dev/ttyUSB2\n”);
      return -1;
   }

   ret     =     set_ppp_param(“*99***1#”,     “0.0.0.0”,     “0.0.0.0”,
“255.255.255.0”);
   if(ret != 0)
   {
      printf("set ppp param Failed!\n);
      gprs_power_off();
      return -1;
   }

   ret = set_dialer_param(“IP”, “3gnet”);
   if(ret != 0)
   {
      printf(“set dialer param Failed!\n”);
      return -1;
   }
   ppp_on();

   while(1)
   {
      printf(“count = %d\n”,count);
      ret = system(“ ifconfig | grep ‘ppp0’ ”);
      if(ret == 0)
      {
         system(“cp /etc/ppp/resolv.conf /etc/resolv.conf”);
         break;
      }
```

```
            sleep(1);
            count++;
            if(count > 15)
            {
               printf("Timeout!!!\n");
               break;
            }
         }
         return 0;
      }
```

9.7. 送信タイムアウトを設定する

この関数は、TransmitATCmd API 関数のタイムアウトをミリ秒単位で設定します。

```
int SetTransmitTimeoutMs (int tMs)
```

パラメータ

tMs [in]　　transmit_timeout を TMS のミリ秒に設定します。

返ってきた数値

なし

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

9.8. 1 つの AT コマンドを送信する

この機能は、3G モジュールに 1 つの AT コマンドを送信し、応答文字列のコードを取得するために使用されます。

```
int TransmitATCmd(char *AtCmd, char *RecvBuffer, int RecvLength).
```

パラメータ

AtCmd [in]　　3G モジュールに送信する AT コマンド。

RecvBuffer [out]　　3G モジュールの応答の文字列コードをにメモリするバッファ。

RecvLength [in]　　RecvBuffer の長さ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

9.9. IMEI シリアル番号を取得する

この機能は、3G モジュールの IMEI（国際モバイルデバイスの身分番号）シリアル番号情報を返すするために使用されます。

```
int Get_IMEI_SN(char *IMEI_SN, int IMEILength).
```

パラメ―タ―

IMEI_SN [out] IMEI シリアル番号情報のためのバッファ

IMEILength [in] *MEI_SN* バッファの長さ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(int argc, char *argv[])
{
   int ret=0;
   char IMEI_Buf[64];

   ret = gprs_power_on();
   if(ret != EGPRS_SUCCEEDED)
   {  printf("gprs power on failed, ret = %d\n”,ret);
      return -1;
   }
   /* notice:After poweron 3g module, must wait for 9s, and then check
if ‘/dev/ttyUSB2’ exist */
   sleep(9);
   if(access(“/dev/ttyUSB2”,0) != 0)
   {
      Printf(“no exist /dev/ttyUSB2\n”);
      return -1;
   }
```

```
    ret = Get_IMEI_SN(IMEI_Buf, sizeof(IMEI_Buf));
    if(ret != 0)
    {  printf("Get IMEI_SN Failed, ret = %d\n”,ret);
       gprs_power_off();
       return -1;
    }
    printf(“IMEI Serial Number = %s\n”, IMEI_Buf);

    ret = gprs_power_off();
    if(ret != EGPRS_SUCCEEDED)
    {  printf(“gprs power off Failed!\n”);
       return -1;
    }
    return 0;
}
```

10.0.オーディオ（ALSA）の API

このセクションでは、デバイスのオーディオ設定を構成する際の API 関数について説明します。

10.1. システムオーディオの音量を取得する

この関数はシステムオーディオの音量を返すするために使用されます。

```
int audio_volume_get(struct volume_state *stat)
```

パラメ―タ―

```
struct volume_state {
   unsigned int min_vol; //the minimal level of volume
   unsigned int max_vol; //the maximal level of volume
   unsigned int current_vol; //the left current volume
};
```

stat [out]　返された音量の状態へのポインタ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　acs_api.h で宣言される

ライブラリファイル　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   struct volume_state stat;

   memset(&stat, 0x00, sizeof(struct volume_state));

   ret = volume_get((&state); //call API to Obtain volume level
   if(0 == ret)
   {//print the volume state you get just now.
      printf("max volume is %ld\nmin volume is %ld\n, left
      volume is %ld\nright volume is %ld\n", stat.max_vol,
      state.min_vol, stat.left_vol, stat.right_vol);
   }

   return ret;
}
```

10.2. システムオーディオの音量を設定する

この関数はボリュームの値を設定するために使用されます。

```
int audio_volume_set(unsigned int volume)
```

パラメーター

volume [in]　　音量レベル（レベルは 1～18 まで、１８以上は 18 として扱います）を設定します。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

10.3. サウンドファイルを再生する

この機能は、WAVE 形式のサウンドファイルを再生するために使用されます。

```
int sound_play(char *file_path)
```

パラメーター

file_path　*[in]*　　サウンドファイルの完全なパス。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;

   ret = sound_play ("./Niose.wav");//call api to play a specified audio
   file

   return ret;
}
```

10.4. スピーカーサウンドコントロール

この機能は、スピーカをオン/オフにするために使用されます。

```
int speaker_onoff(int onoff)
```

パラメ―タ―

onoff *[in]* 1 = オン ; 0 = オフ

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;

   ret = speaker_onoff(1);//call api to sound on speaker.

   return ret;
}
```

11.0.ファームウェアの API

このセクションでは、デバイスのファームウェアを配置するための API 関数について説明します。

11.1. ファームウェアのバージョン番号を取得する

この機能は、デバイスのファームウェアのバージョン（メジャーバージョン、マイナーバージョン、リビジョンバージョン）を返すするために使用されます。

```
int get_acr890_version(struct acr890_version *version)
```

パラメ―タ―

```
struct acr890_version{
   unsigned int major;
   unsigned int minor;
   unsigned int revision;

};
```

version [out]　ファームのバージョン（メジャーバージョン、マイナーバージョン、リビジョンバージョン）へのポインタ

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ　acs_api.h で宣言される

ライブラリファイル　libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   struct acr890_version version;
   ret = get_acr890_version(&version);
   if(ret == 0) {
      printf("Major = %d\n", version.major);
      printf("Minor = %d\n", version.minor);
      printf("Revision = %d\n", version.revision);

   }
   return ret;
}
```

12.0.サーマルプリンタの API

このセクションでは、デバイスのサーマルプリンタを配置するための API 関数について説明します。

12.1. プリンタポートを開く

この機能は、プリンタポートを開くために使用されます。

```
int printer_open(void)
```

パラメータ

なし

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_OPEN_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.2. プリンタポートを閉じる

この機能は、プリンタポートを閉じるために使用されます。

```
int printer_close(void)
```

パラメータ

なし

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_CLOSE_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.3. プリンターの状態を取得する

この関数はプリンターの状態を返します。

```
int printer_status_get(void)
```

パラメーター

```
enum printer_state {
      PRINT_STAT_UNKNOWN = 0, /* unknown state */
      PRINT_STAT_INT = 0x53, /* print suspend (no paper) */
      PRINT_STAT_IDLE = 0x90, /* Idle state */
      PRINT_STAT_BFULL = 0x65, /* full of buffer */
      PRINT_STAT_NOPAPER = 0x68, /* out of paper */
      PRINT_STAT_BFNP = 0x63, /* full of buffer and out of paper */
      PRINT_STAT_MAX
};
```

返ってきた数値

任意の *enum printer_state* 値。

必要条件

头文件 acs_api.h で宣言される。

ライブラリ libacs_api.so を使用する

12.4. プリンタに用紙を供給する

この機能は、プリンタに用紙を供給するために使用されます。

```
int printer_page_feed(char cControl, char cLine)
```

注：*ACR890 フェーズ 1 デバイス*を使っている場合、コマンドのフォーマットは：

```
int printer_page_feed(unsigned char nr_len)
```

パラメーター

cControl [in] プリンタのホイールを順方向または逆方向に制御するコード。

0 - 前にフィードする

1 - 後にフィードする

nr_len [in] 供給される用紙スペース（0～255）。スペースは nr_len x 0.125（ミリメートル単位）に等しい

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_FPAPER_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.5. プリンターがデータをプリンタアウトように準備します

この関数は、印刷データをプリンタに準備します。

```
int printer_prepare_data(char *pstr, const int nlen)
```

パラメ―タ―

pstr [in] 入力された str データをプリンタに渡します。

nlen [in] *str* データの長さ。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_DATA_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.6. データをプリンタアウトし始めます。

この関数は準備されたばかりの文字列データの印刷を開始します。

```
int printer_printing(void)
```

パラメ―タ―

なし

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_STRPRINT_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.7. プリンターを設定する

この関数は、印刷スペースとフォントピクセルサイズを設定します。

```
int printer_setPrinter(unsigned int uiFormat,unsigned int iFormatValue)
```

パラメーター

uiFormat [in] 行または列のスペースを設定します。

uiFormatValue [in] 设置字体像素大小，有效值如下：

EM_prn_ASCIIPRN1X1
EM_prn_HZPRN1X1
EM_prn_ASCIIPRN2X1
EM_prn_HZPRN2X1
EM_prn_ASCIIPRN1X2
EM_prn_HZPRN1X2
EM_prn_ASCIIPRN2X2
EM_prn_HZPRN2X2
EM_prn_ASCIIPRN1X3
EM_prn_HZPRN1X3
EM_prn_ASCIIPRN3X1
EM_prn_HZPRN3X1
EM_prn_ASCIIPRN3X2
EM_prn_HZPRN3X2
EM_prn_ASCIIPRN2X3
EM_prn_HZPRN2X3
EM_prn_ASCIIPRN3X3
EM_prn_HZPRN3X3

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_STRPRINT_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.8. 標準モードで文字列を印刷する

この機能は、標準モードで文字列を印刷するために使用されます。印刷データのサイズが 65535 バイト以下でなければなりません。制御文字「\ n」を使用することができます。

```
int printer_printStrSM(const char *pString, cons tint nLen, unsigned int
Mode)
```

パラメ―タ―

pString [in] 出力を待っているヌル終了文字列。

nLen[in] 入力 *pString* の長さ。

Mode [in] 印刷のアラインメントモード。

09h - 右揃え

0Ah - 中央揃え

08h また 0Bh - 左揃え

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_STRPRINT_SM* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

12.9. ビットマップファイルを印刷する

この関数は、ビットマップ（.bmp）ファイルを印刷します。

```
int printer_print_bmp(const unsigned char *pBmpName, unsigned int
uiStartPosition);
```

パラメ―タ―

pBmpName [入力] 入力ビットマップファイルのフルパス名。サポートされているビットマップファイルの色深度は、1 ビット、8 ビット、24 ビット、および 32 ビットです。

uiStartPosition [入力]　印刷の開始位置。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

成功した場合、戻り値は *ETHP_IMAGEPRINT* です。

必要条件

ヘッダ　acs_api.h で宣言される

ライブラリファイル　libacs_api.so を使用する

サンプルコード

```
int main(int argc, char *argv[])
{
   int nRet=0;
   char szcom[64]={"ABCDEF1234567890ABCDEFIJG12345QWERT"};
   nRet=printer_open();
   if(nRet<0)
   {
      printf("printer_open erro %d\n",nRet);
   }
    nRet = printer_status_get();
   if(nRet != PRINTER_READY)
   {
      printf("printf error %d\n", nRet);
      return -1;
   }
   //standard mode printing
   printer_printStrSM(szcom, strlen(szcom), 0x08);

   nRet = printer_status_get();
   if(nRet !=PRINTER_READY)
   {
      printf("printer_printStrPM nRet = [%x]\n",nRet);
      return -1;
   }

   //printing and change new line
    printer_page_feed(0, 3);


  //Reset and cache flush
   printer_close();
}
```

12.10. FreeType ライブラリを初期化する

この関数は FreeType ライブラリを初期化するために使われます。非英語のプリンターに専用です。

注： *v1.1.5* 以降のファームウェアがこの関数しかサポートできません。

```
int freetype_init(const char* fontname)
```

パラメーター

fontname [in]　　字体ファイルのパス

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_INITFREE_ERR* です。

必要条件

ヘッダファイル　lib_freetype.h に声明します

ライブラリ　libacs_printer.so を使用します

12.11. 文字サイズの設定

この関数は文字のサイズを設定するために使われます。非英語の字体のプリンターに専用です。

注： *v1.1.5* 以降のファームウェアがこの関数しかサポートできません。

```
int freetype_setsize(unsigned int width, unsigned int height)
```

パラメーター

width [in]　文字の幅（ピクセルを単位で）。有効範囲は 11 – 16。

height [in]　文字の高さ（ピクセルを単位で）。有効範囲は 11 – 16。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_SETSIZE_ERR* です。

必要条件

ヘッダファイル lib_freetype.h に声明します

ライブラリ libacs_printer.so を使用します

12.12. 字体画像を取得して、プリンターする

この関数は字体画像を取得して、プリンターするために使われます。非英語の字体のプリンターに専用です。

注： *v1.1.5* 以降のファームウェアがこの関数しかサポートできません。

```
int freetype_printString(char *string, int nlen)
```

パラメ―タ―

string[in] 印刷されていない文字の配列へのポインタ。

nlen [in] `印刷されていない文字の配列のサイズ（バイトを単位として）。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_IMAGEPRINT* です。

必要条件

ヘッダファイル lib_freetype.h に声明します

ライブラリ libacs_printer.so を使用します

12.13. FreeType ライブラリを開放する

この関数は FreeType ライブラリを開放するために使われます。非英語のプリンターに専用です。

注： *v1.1.5* 以降のファームウェアがこの関数しかサポートできません。

```
void freetype_release(void)
```

パラメ―タ―

なし。

返ってきた数値

なし。

必要条件

ヘッダファイル lib_freetype.h に声明します

ライブラリ libacs_printer.so を使用します

13.0.無線の LAN モジュールの API

このセクションでは、無線の LAN モジュールを配置するための API 関数について説明します。

13.1. 無線の LAN モジュールを有効する

この機能は、無線の LAN モジュールを電気入れるために使用されます。

```
int wifi_pwr_on(void)
```

パラメ―タ―

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 ではありません。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

13.2. 無線の LAN モジュールをオフする

この機能は、無線の LAN モジュールの電源をオフにするために使用されます。

```
int wifi_pwr_off(void)
```

パラメ―タ―

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

14.0.ブルートゥースモジュールの API

このセクションでは、ブルートゥースモジュールを配置するための API 関数について説明します。

14.1. ブルートゥースモジュールをオンにする

この機能は、Bluetooth モジュールをオンにするために使用されます。

```
int bluetooth_pwr_on(void)
```

パラメータ

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

14.2. ブルートゥースモジュールをオフにする

この機能は、Bluetooth モジュールをオフにするために使用されます。

```
int bluetooth_pwr_off(void)
```

パラメータ

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

15.0.スマートカードリーダーの API

このセクションでは、デバイスの接触式と非接触式のスマートカードリーダーモジュールの API 関数について説明します。

15.1. スマートカードリーダーモジュールをオンにする

スマートカードリーダーモジュールをオンにする時にこの関数を使用します。PC / SC コマンドを実行する前に約 2 秒の遅延が必要です。

```
int smartcard_open(void)
```

パラメ―タ―

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は != 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

15.2. スマートカードリーダーモジュールをオフにする

スマートカードリーダーモジュールをオフにする時にこの関数を使用します。

```
int smartcard_close(void)
```

パラメ―タ―

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は ≠ 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

15.3. スマートカード操作の PC / SC API

PC / SC API 関数のリストと説明については、次のリンクを参照してください。

https://pcsclite.alioth.debian.org/api/modules.html

サンプルコード

```
int main(int argc,char *argv[])
{
SCARDCONTEXT hcontext;
char readerName[8][32];

lone rv;
lone len;
int readerNum = 0;
char *pReader;
char pmszReaders[256];

int ret;

/***Open pcscd daemon process***/
ret = smartcard_open();
if(ret != 0)
{
   return -1;
}

/***List All Reader***/
rv = SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &hcontext);
if(rv != SCARD_S_SUCCESS)
{
   printf("rv = %s\n", pcsc_stringify_error(rv));
}
len = sizeof(pmszReaders);
rv = SCardListReaders(hcontext, NULL, pmszReaders, &len);
if(rv != SCARD_S_SUCCESS)
{
   printf("rv = %s\n", pcsc_stringify_error(rv));
   return rv;
}
pReader = pmszReaders;
while(*pReader != '\0')
{
   strcpy(readerName[readerNum], pReader);
   pReader = pReader + strlen(pReader) + 1;
   readerNum++;
}

/***Transmit Apdu Data***/
DWORD dwAP;
DWORD dwLen;
DWORD dwAtrLen;
unsigned char pAtr[64];
DWORD dwState;
DWORD dwProtocol;
int retval;
int I;
```

```
SCARD_IO_REQUEST ioSendPci;
SCARD_IO_REQUEST ioRecvPci;

unsigned char pTx[] = {0x80, 0x84, 0x00, 0x00, 0x08};
unsigned int txLen;
unsigned char pRx[64];
unsigned char rxLen = sizeof(pRx);

rv    =    SCardConnect(hcontext,    readerName[0],    SCARD_SHARE_SHARED,
SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1, &hcard, &dwAP);
if(rv != SCARD_S_SUCCESS)
{
   printf("rv = %s\n", pcsc_stringify_error(rv));
   return rv;
}

ioSendPci.dwProtocol = dwAP;
ioSendPci.cbPciLength = sizeof(SCARD_IO_REQUEST);

rv  =  SCardTransmit(hcard,  &ioSendPci,  pTx,  (DWORD)txLen,  &ioRecvPci,
pRx, (DWROD*)&rxLen);
if(rv != SCARD_S_SUCCESS)
{
   printf("rv = %s\n", pcsc_stringify_error(rv));
   return rv;
}

/***Close Smart Card Reader***/
SCardDisconnect(hcard, SCARD_LEAVE_CARD);
SCARDReleaseContext(hcontext);

smartcard_close();
}
```

16.0.磁気ストライプインターフェースの API

このセクションでは、磁気ストライプカードモジュールを配置するための API 関数について説明します。

16.1. 磁気ストライプカードからトラックデータを取得する

この関数は、与えられた時間内に、磁気ストライプカードからトラックデータを読み取るために使用されます。

```
int msr_trackdata_get(struct msr_data *data, unsigned int time)
```

パラメーター

```
struct msr_data {/* Each track data end with character '\0' */
   char track_data1[80]; /* track #1 data */
   char track_data2[41]; /* track #2 data */
   char track_data3[108]; /* track #3 data */
   unsigned int track_state;
      };
```

data [out]　　トラックデータと状態が含まれています。

time [in]　　スワイプイベントの待機時間（秒単位）。通常、スワイプイベントは 5～30 秒の範囲で実行する必要があります。

ビット	説明
Bit 31:27	保留
Bit 26	トラック＃3 データ存在
Bit 25	トラック＃2 データ存在
Bit 24	トラック＃1 データ存在
Bit 23:20	保留
Bit 19	トラック＃3 データ LRC エラー
Bit 18	トラック＃3 データエンドバイト誤り
Bit 17	トラック＃3 データパリティエラー
Bit 16	トラック＃3 データスタートバイト誤り
Bit 15:12	保留
Bit 11	トラック＃2 データ LRC エラー
Bit 10	トラック＃2 データエンドバイト誤り
Bit 9	トラック＃2 データパリティエラー
Bit 8	トラック＃2 データスタートバイト誤り
Bit 7:4	保留

ビット	説明
Bit 3	トラック＃1 データ LRC エラー
Bit 2	トラック＃1 データエンドバイト誤り
Bit 1	トラック＃1 データパリティエラー
Bit 0	トラック＃1 データスタートバイト誤り

表 3：トラックデータの状態ビットテーブル

返ってきた数値

すべてのトラックが OK であれば、戻り値は 0 です。

それ以外の場合、戻り値は 0 に等しくありません。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(int argc, char *argv[])
{
      struct msr_data tMSRData;
      int iTimeout =30;
      int ret;
      int i;

      if(2 == argc)
      {
         iTimeout = atoi(argv[1]);/* Get how much time need to wait */
      }
      memset(&tMSRData, 0x00, sizeof(struct msr_data));
      ret = msr_track_data_get(&tMSRData, iTimeout);
      if (tMSRData.track_state & BIT(24))/* Got track #1 data */
      {
         printf("track #1 data :\n");
         for(i = 0; i < sizeof(tMSRData. track_data1); i++)
         {
      printf(“0x%02X ”, tMSRData. track_data1[i]);
         }
         printf(“\n”);
      }
      else
      {
         printf("track 1 error :");
         PRINT_MSR_ERROR(tMSRData.track_state);
         printf("\n");
      }

      return ret;
}
```

17.0. エラーコードを説明する API

このセクションでは、エラーコード説明の API 関数について紹介します。

17.1. エラーコードの説明を取得する

デバッグ目的のために、指定されたエラーコードで詳細を取得するために、この API を使用します。

```
char *acs_err(const int errno_code)
```

パラメーター

errno_code [in]　　解析されていないエラーの番号。

返ってきた数値

エラー番号ための解析された文字列。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

18.0.パワーマネジメントの API

このセクションでは、デバイスのシステムパワーマネジメントの API 関数について説明します。

18.1. システムスリープタイムアウトを設定する

この機能は、デバイスのアイドル時間を設定するために使用されます。アイドルタイマーが満了したときにシステムがスリープモードに切り替わります。しかし、任意の入力イベントは、アイドルタイマーがリセットさせます。

```
int pm_sleep_timeout_set(unsigned long seconds)
```

パラメータ

seconds[in]　　スリープモードに入る前の最大時間。有効範囲は 30-1800。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダー　　acs_api.h で宣言される。

ライブラリ　　libacs_api.so を使用する。

18.2. 現在のシステムスリープモードを取得する

この関数は、デバイスの現在のアイドルモードを取得します。

```
unsigned int pm_get_sleep_mode(enum sleep_mode *mode);
```

パラメータ

mode [out]　　現在のシステムスリープモードの値を格納します。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダー acs_api.h で宣言される。

ライブラリ libacs_api.so を使用する。

18.3. システムスリープ時間を取得する

この機能は、デバイスのアイドル時間を取得するために使用されます。

```
unsigned int pm_sleep_timeout_get(void)
```

返ってきた数値

デバイスのアイドル時間の値。

必要条件

ヘッダー acs_api.h で宣言される。

ライブラリ libacs_api.so を使用する。

19.0.バーコードスキャン器の API

このセクションでは、バーコードスキャン器を配置するための API 関数について説明します。

必要条件

ヘッダー acs_api.h で宣言される。

ライブラリ libacs_api.so を使用する。

19.1. バーコードスキャン器モジュールをオンにする

```
int scanner_open(void)
```

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_OPEN_ERR (1056)。

19.2. バーコードスキャン器モジュールをオフにする

```
int scanner_close(void)
```

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_CLOSE_ERR (1057)。

19.3. スキャンを接続する

```
int scanner_Connect(void)
```

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_CONNECT_ERR (1058)。

19.4. データスキャン

```
int scanner_Scan(unsigned char pScanBuf, unsigned int pScanLen,unsigned
int TimeOutSec);
```

パラメ―タ― :

[out] - pScanBuf =スキャンしたデータのバッファ

[out] - pScanLen =スキャンしたデータの長さ

[in] - timeoutSec =スキャンタイムアウト時間（秒単位）

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_SCAN_ERR (1059)。

19.5. スキャナの読み取りセッションを取得する

```
int scanner_ReadingSession(int status)
```

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_READSESSION_PM (1062)。

19.6. スキャナのファームウェアバージョンを取得する

```
int scanner_GetFirmwareVersion(unsigned char pBuf, unsigned int pLen); --
- get Scanner's firmware version
```

パラメ—タ—：

[out] - pBuf =ファームウェアバージョンデータのバッファ

[out] pLen =返されるバッファの長さ

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_GETVERSION_SM (1061)。

19.7. スキャナの接続を切る

```
`int scanner_Disconnect(void)
```

返ってきた数値

成功した場合、戻り値は SUCCESS (0)。

失敗した場合、戻り値は ESCAN_DISCONNECT (1063)。

附录 A. ACR890 フェーズ 1 コマンド

このセクションでは、ACR890 Phase 1 デバイスにのみ適用されるコマンドの一部を示します。

附录 A.1. 非接触インターフェースの疑似 APDU コマンド

このセクションでは、ACR890 Phase 1 デバイスにのみ適用可能な非接触インターフェイス関連のコマンドについて説明します。

附录 A1.1. データを取得する（Get Data）

GET DATA コマンドは“接続された PICC”のシリアルナンバーもしくは ATS を返します。

GET UIDAPDU フォーマット（5 バイト）

コマンド	CLA	INS	P1	P2	Le
Get Data	FFh	CAh	00h 01h	00h	00h （全長）

例え P1 = 00h、応答フォーマットは UID 取得（UID + 2 バイト）

応答	データ出力					
結果	UID (LSB)			UID (MSB)	SW1	SW2

例え P1 = 01h、ISO14443 A タイプのカードの ATS を取得します（ATS + 2 バイト）

応答	データ出力		
結果	ATS	SW1	SW2

Get Data の応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
警告	62 82h	UID/ATS の終わりが Le バイトの前に達しました（Le は UID の長さより大きいです）
エラー	6C XXh	間違った長さ（間違ったナンバーLe：‘XX’は精確な数字を表す）、Le は、利用可能な UID の長さ未満である場合

結果	SW1 SW2	意味
エラー	63 00h	操作が失敗しました。
エラー	6A 81h	この機能をサポートできません。

例 1：

“接続された PICC”のシリアルナンバーを取得します

UINT8 GET_UID[5]={FFh, CAh, 00h, 00h, 00h};

例 2：“接続された ISO 14443-A PICC”の ATS を取得します

UINT8 GET_ATS[5]={FFh, CAh, 01h, 00h, 00h};

附录 A.2. MIFARE® Classic 1K/4K メモリカードの PICC コマンド（T=CL エミュレーション）

附录 A.2.1. 認証キーのダンロード（Load authentication keys）

このコマンドはリーダーにキーをロードする時に使われる。このキーは MIFARE Classic 1K/4K メモリカードの特定なセクターを認証するために使用される。リーダーは二種の認証キーのアドレスが提供られている：失いやすいキーのアドレスと失いやすくないキーのアドレス。

Load Authentication Keys APDU フォーマット（11 バイト）

コマンド	CLA	INS	P1	P2	Lc	データイン
Load Authentication Keys	FFh	82h	キー構造	キーナンバー	06h	キー（6 バイト）

その中：

キー構造 1 バイト。

00h = キーが失いやすいキーのメモリにロードされる。

その他＝保留

キーの番号 1 バイト。

00h ー 01h = キーの場所。リーダーが PC から切断された時、キーが消削除されます。

キー 6 バイト。リーダーにロードされるきーの値。

例：{FF FF FF FF FF FFh}

Load Authentication Keys 応答フォーマット（2 バイト）

応答	データ出力	
結果	SW1	SW2

Load Authentication Keys の応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

例：

失いやすいキーの場所に 00h キーをロードする {FF FF FF FF FF FFh}。

APDU = {FF 82 00 00 06 FF FF FF FF FF FFh}

附录 A.2.2. MIFARE Classic (1K/4K) カードに対しての認証 (Authenticate MIFARE Classic (1K/4K))

このコマンドは、MIFARE Classic 1K/4K カード（PICC）との認証を行うためにリーダーに格納された鍵を使用しています。認証キーの二種類が用いられています：TYPE_A と TYPE_B。

Load Authentication Keys APDU フォーマット（6 バイト）

コマンド	CLA	INS	P1	P2	P3	データイン
Authentication	FFh	88h	00h	ブロック番号	キーのタイプ	キーナンバー

Load Authentication Keys APDU フォーマット（10 バイト）

コマンド	CLA	INS	P1	P2	Lc	データイン
Authentication	FFh	86h	00h	00h	05h	データバイト認証

データバイト認証（5 バイト）

バイト 1	バイト 2	バイト 3	バイト 4	バイト 5
バージョン番号 01h	00h	ブロック番号	キーのタイプ	キーナンバー

その中：

ブロック番号 1 バイト。認証されていないメモリブロック。

キーのタイプ 1 バイト。

60h = TYPE A キーとして、認証用に使われる。

61h = TYPE B キーとして、認証用に使われます。

キーの番号 1 バイト。

00h ~ 01h = キーの場所。

注： 一枚の *MIFARE Classic 1K* カードが *16* 個セクターと分けて、各セクターには *4* 個の連続的なブロックが含めています。例：セクター*00* が含めているブロック*{00h、01h、02h* および *03h}*；セクター*01h* が含めているブロック*{04h、05h、06h* および *07h}*；ラストセクター*0Fh* が含めているブロック*{3Ch、3Dh、3Eh* および *3Fh}*。

当ブロックが成功に認証されると、同じセクターの全てのブロックをアクセスできる。詳しい情報は *MIFARE Classic 1K/4* k 基準を参照してください。

Load Authentication Keys 応答フォーマット（2 バイト）

応答	データ出力	
結果	SW1	SW2

Load Authentication Keys の応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

セクター (16 個のセクター，各セクターには 4 個の連続的なブロックが含めている)	データブロック (3 個のブロック，各には 16 バイト)	トレーラーブロック (1 つのブロック，16 バイト)
セクター0	00h – 02h	03h
セクター1	04h – 06h	07h
..	..	..
..	..	..
セクター14	38h – 0Ah	3Bh
セクター15	3Ch – 3Eh	3Fh

1 KB

表 4：MIFARE 1K カードのメモリマップ

セクター (32 個のセクター，各セクターには 4 個の連続的なブロックが含めている)	データブロック (3 個のブロック，各には 16 バイト)	トレーラーブロック (1 つのブロック，16 バイト)	
セクター0	00h – 02h	03h	
セクター1	04h – 06h	07h	
…	…	…	
…	…	…	2 KB
セクター30	78h – 7Ah	7Bh	
セクター31	7Ch – 7Eh	7Fh	

セクター (8 個のセクター，各セクターには 16 個の連続的なブロックが含めている)	データブロック (15 個のブロック，各には 16 バイト)	トレーラーブロック (1 つのブロック，16 バイト)	
セクター32	80h – 8Eh	8Fh	
セクター33	90h – 9Eh	9Fh	
…	…	…	
…	…	…	2 KB
セクター38	E0h – EEh	EFh	
セクター39	F0h – FEh	FFh	

表 5：MIFARE 4K カードのメモリマップ

例 1：

次の特徴でブロック 04h を認証します：A タイプ,失いやすくない,キーの番号 00h, PC/SC V2.01 から(廃棄)。

APDU = {FF 88 00 04 60 00h};

例 2：

前と同じで、次の特徴でブロック 04h を認証します：A タイプ,失いやすくない,キーの番号 00h, PC/SC V2.07 から。

APDU = {FF 86 00 00 05 01 00 04 60 00h}

注：これはユーザーデータ領域への無料アクセスを提供するので、認証が必要ではありません。

バイトナンバー	0	1	2	3	ページ
シリアルナンバー	SN0	SN1	SN2	BCC0	0
シリアルナンバー	SN3	SN4	SN5	SN6	1
内部/ロック	BCC1	Internal	Lock0	Lock1	2
OTP	OPT0	OPT1	OTP2	OTP3	3
データリーダー/ライター	Data0	Data1	Data2	Data3	4
データリーダー/ライター	Data4	Data5	Data6	Data7	5
データリーダー/ライター	Data8	Data9	Data10	Data11	6
データリーダー/ライター	Data12	Data13	Data14	Data15	7
データリーダー/ライター	Data16	Data17	Data18	Data19	8
データリーダー/ライター	Data20	Data21	Data22	Data23	9
データリーダー/ライター	Data24	Data25	Data26	Data27	10
データリーダー/ライター	Data28	Data29	Data30	Data31	11
データリーダー/ライター	Data32	Data33	Data34	Data35	12
データリーダー/ライター	Data36	Data37	Data38	Data39	13
データリーダー/ライター	Data40	Data41	Data42	Data43	14
データリーダー/ライター	Data44	Data45	Data46	Data47	15

512 ビット
または
64 バイト

表 6：MIFARE Ultralight® カードのメモリマップ

附录 A.2.3. バイナリブロックを読み取る（Read Binary Blocks）

複数のデータブロックを PICC カードから取り出すことに使われます。このコマンドを実行する前に、データブロック/トレーラーブロックを認証しなければなりません。

Read Binary の APDU フォーマット（5 バイト）

コマンド	CLA	INS	P1	P2	Le
Read Binary Blocks	FFh	B0h	00h	ブロック番号	更新していないバイト

その中：

ブロック番号 1 バイト。アクセスされていない数値のブロック。

更新していない 1 バイト。最大の数値は 16 バイトです。

Read Value Block の応答フォーマット（N + 2 バイト）

応答	データ出力		
結果	0 <= N <= 16	SW1	SW2

Read Binary Block 応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

例 1：バイナリブロック 04h から 16 バイトを読み取る（MIFARE 1K または 4K）

APDU = {FF B0 00 04 10}

例 2：バイナリブロック 04h から 4 バイトを読み取る（MIFARE Ultralight））

APDU = {FF B0 00 04 04}

例 3：バイナリページ 04h から 16 個バイトを読み取る（MIFARE Ultralight）（4, 5, 6 および 7 ページを読み取ります）

APDU = {FF B0 00 04 10}

附录 A.2.4. バイナリブロックの更新（Update Binary Blocks）

Update Binary Blocks コマンドは複数のデータブロックを PICC カードに書き入れるのに使われます。このコマンドを実行する前に、データブロック/トレーラーブロックを認証しなければなりません。

Update Binary の APDU フォーマット（4 または 16 の倍数 + 5 バイト）

コマンド	CLA	INS	P1	P2	Lc	データイン
Update Binary Blocks	FFh	D6h	00h	ブロック番号	**更新していないバイト**	データブロック MIFARE Ultralight：4 バイト または MIFARE 1K/4K：16 バイト

その中：

ブロック番号 1 バイト。更新していない開始ブロック

更新していない 1 バイト。

MIFARE 1K/4K 更新されていないバイトは 16 です

MIFARE Ultralight 更新されていないバイトは 4 です

データブロック 4 または 16 バイト。

バイナリブロックに書き入れていないデータ。

Update Binary Block の応答コード（2 バイト）

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

例 1：MIFARE 1K/4K カード中のバイナリブロック 04h を{00 01 ..0Fh}に更新します

APDU = {FF D6 00 04 10 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0Fh}

例 2：MIFARE Ultralight 中のバイナリブロック 04h を{00 01 02 03h}に更新する

APDU = {FF D6 00 04 04 00 01 02 03h}

附录 A.2.5. 数値ブロックの操作（Value block operation）（Increment, Decrement, Store）

このコマンドは数値を基づいてのトランザクションを実行する時に使われます。例：数値ブロックの数値を増える。

Value Block Operation の APDU フォーマット（10 バイト）

コマンド	CLA	INS	P1	P2	Lc	データイン	
Value Block Operation	FFh	D7h	00h	ブロック番号	05h	VB_OP	VB_Value（4 バイト）{MSB ..LSB}

その中：

ブロック番号 1 バイト。操作されていない数値のブロック

VB_OP 1 バイト。

00h = VB_Value をブロックにストアーして、このブロックは数値ブロックになります。

01h = VB_Value で値ブロックの値をインクリメントします。このコマンドは、値ブロックに対してのみ有効です。

02h = VB_Value で値ブロックの値を減らします。このコマンドは、値ブロックに対してのみ有効です。

VB_Value 4 バイト。このデータはシンボル付いてる、数値操作ための長い整数です（4 バイト）。

例 1：Decimal - 4 = {FFh, FFh, FFh, FCh}

VB_Value			
MSB			LSB
FFh	FFh	FF	FCh

例 2：Decimal 1 = {00h, 00h, 00h, 01h}

VB_Value			
MSB			LSB
00h	00h	00h	01h

Value Block Operation の応答フォーマット（2 バイト）

応答	データ出力	
結果	SW1	SW2

Value Block Operation 応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

附录 A.2.6. 数値ブロックを読み取る（Read Value Block）

このコマンドは数値ブロックの数値を返すために使われます。数値ブロック対しての操作のみに適用しています。

Copy Value Block の APDU フォーマット（5 バイト）

コマンド	CLA	INS	P1	P2	Le
Read Value Block	FFh	B1h	00h	ブロック番号	04h

その中：

ブロック番号 1 バイト。読み取られていない数値ブロック

Read Value Block の応答フォーマット（4 + 2 バイト）

応答	データ出力		
結果	数値 {MSB ..LSB}	SW1	SW2

その中：

値 4 バイト。カードから返された数値。この値は符号付き長い整数です（４バイト）。

例 1：Decimal - 4 = {FFh, FFh, FFh, FCh}

数値			
MSB			LSB
FFh	FFh	FFh	FC

例 2：Decimal 1 = {00h, 00h, 00h, 01h}

数値			
MSB			LSB
00h	00h	00h	01h

Read Value Block コマンドの応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。
エラー	63 00h	操作が失敗しました。

附录 A.2.7. 数値ブロックをコピーする（Copy Value Block）

このコマンドは一つの数値ブロック中の数値を別の数値ブロックにコピーする時に使われます。

Copy Value Block の APDU フォーマット（7 バイト）

コマンド	CLA	INS	P1	P2	Lc	データイン	
Copy Value Block Operation	FFh	D7h	00h	**ソース**ブロック番号	02h	03h	ターゲットブロック番号

その中：

元ブロックの番号　　1 バイト。ソース値のブロックの値が目標値ブロックにコピーされる。

ターゲットブロック番号　　1 バイト。復元する値ブロック。ソースとターゲット値のブロックは、必ず同じセクター内にある。

Copy Value Block の応答フォーマット（2 バイト）

応答	データ出力	
結果	SW1	SW2

Copy Value Block の応答コード

結果	SW1 SW2	意味
成功	90 00h	操作が成功に完了しました。

結果	SW1 SW2	意味
エラー	63 00h	操作が失敗しました。

例 1：“1”をブロック 05h にストアーします。

APDU = {FF D7 00 05 05 00 00 00 00 01h}

例 2：数値ブロック 05h を読み取ります

APDU = {FF B1 00 05 00h}

例 3：数値ブロック 05h 中の数値をブロック 06h にコピーします。

APDU = {FF D7 00 05 02 03 06h}

例 4：数値ブロック 05h の値を“5”インクリメントします

APDU = {FF D7 00 05 05 01 00 00 00 05h}

応答：90 00h [$9000]

附录 A.3. サーマルプリンタの API

このセクションでは、ACR890 フェーズ 1 デバイスにのみ適用可能なサーマルプリンタのコマンドの一部について説明します。

附录 A.3.1. プリンターを再起動する

この機能は、受信バッファおよびプリントバッファに格納されたすべてのデータを消去するために使用されます。プリンタをリセットし、デフォルト値をすべてのユーザー設定に復元します。

```
void printer_reset(void)
```

パラメータ

なし。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_RESET_ERR* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.3.2. 標準モードでラインスペースを設定する

この機能は、標準モードでの行間を設定するために使用されます。

```
int printer_setLineSpaceSM(unsigned char nr_step)
```

パラメータ

nr_len [in] 紙供給のペーパー空間（0 から 255 までの範囲、単位は mm で、スペースは nr_len*0.125 に等しいです）

返ってきた数値

成功した場合、戻り値は SUCCESS です。

失敗した場合、戻り値は ETHP_SETLINE_SPACE です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.3.3. 標準モードで文字列を印刷する

この機能は、標準モードで文字列を印刷するために使用されます。印刷データのサイズが 65535 バイト以下でなければなりません。制御文字「\ n」を使用することができます。

```
int printer_printStrSM(const char *str)
```

パラメータ

str [in] 出力・を待っているヌル終了文字列。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_STRPRINT_SM* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.3.4. ページモードで文字列を印刷する

この機能は、ページモードで文字列を印刷するために使用されます。印刷データのサイズが 490 バイト以下でなければなりません。データサイズが 490 バイトよりも大きい場合には、超過したデータは破棄されます。制御文字「\ n」を使用することができます。

```
int printer_printStrPM(const struct print_page_mode *param,
const char *data, unsigned short size)
```

パラメータ

```
typedef struct print_page_mode {
   unsigned short HorizontalOrigin_X;
   unsigned short VerticalOrigin_Y;
   unsigned short PrintWidth_X;
   unsigned short PrintHeight_Y;
   unsigned short ucLineSpace;
}PRT_PAGE_MODE_PARAM;
```

「ページモード」で印刷範囲を設定するために使用します。

データメンバー	値（含み）	説明
HorizontalOrigin_X	0 ～383	X 軸の出発点
VerticalOrigin_Y	0 ～882	Y 軸の出発点
PrintWidth_X	1～384	印刷領域の幅
PrintHeight_Y	1～883	印刷領域の高さ
ucLineSpace	24～255	行間

注釈：

- *HorizontalOrigin_X + PrintMidth_X* 未満または *384* と同じである必要があります
- *VerticalOrigin_Y + PrintHeight_Y* 未満または *883* と同じである必要があります
- 水水平方向の出発点は絶対原点から *HorizontalOrigin_X*0.125*mmに等しいです。
- 垂直方向の出発点は絶対原点から *VerticalOrigin_Y*0.125*mmに等しいです。
- 実際の印刷の幅= *PrintWidth_X*0.125*mm。
- 実際の印刷の高さ = *PrintHeight_Y*0.125*mm。
- 実際の印刷の行間 = *ucLineSpace*0.125*mm。
- 絶対原点は印刷領域の左上を示します。印刷の幅と高さを *0* に設定することができません。
- 行間はフォントの高さが含まれています。

param [in]　　印刷されていない領域。

data [in]*[in] data*　　印刷されていない文字の配列へのポインタ

size [in]　　印刷されていない文字の配列のサイズ（1～490 バイト）。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_STRPRINT_PM* です。

必要条件

ヘッダ　　acs_api.h で宣言される

ライブラリファイル　　libacs_api.so を使用する

附录 A.3.5. 標準モードでデータ配列を印刷する

この関数は、"標準モード"で文字の配列を出力します。制御文字 '\ n'を使用することができます。

```
int printer_printDataSM(const unsigned char *data, unsigned short size)
```

パラメ—タ—

data [in]*[in] data* 印刷されていない文字の配列へのポインタ

size [in] 印刷されていない文字の配列のサイズ（バイトを単位として）。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_DATAPRINT_PM* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.3.6. ページモードでデータ配列を印刷する

この関数は、データの配列を "ページモード"で表示します。 印刷データサイズは 490 バイト以下でなければなりません。制御文字「\ n」を使用することができます。

```
int printer_printDataPM(const struct print_page_mode *param,
const unsigned char *data, unsigned short size);
```

パラメ—タ—

param [in] 印刷の領域。

data [in] 印刷されていない文字の配列へのポインタ

size [in] 印刷されていない文字の配列のサイズ（1～490 バイト）。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_DATAPRINT_PM* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.3.7. 画像を印刷する

この機能は、画像を印刷するために使用されます。各バイトは、水平方向に印刷された 8 点を表します。画像データが左から右へ、上から下への方向で紙にバイトでプリントされています。

```
int printer_print_img(const unsigned char *bitmap, unsigned short width,
unsigned short height, unsigned char mode);
```

パラメ―タ―

bitmap*[in]* 印刷されていない画像へのデータポインタ。

width [in] 画像の幅。

height [in] 画像の高さ。

mode [in] シングルモードを選択して、幅のパラメーターは 1 ～192 の範囲（両端を含む）を超えていない場合、"FALSE"を入力します。二重モードを選択して、幅のパラメーターは 1 ～384 の範囲（両端を含む）を超えていない場合、“TRUE”を入力します。

返ってきた数値

成功した場合、戻り値は *SUCCESS* です。

失敗した場合、戻り値は *ETHP_IMAGEPRINT* です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4. 接触インターフェイス（ICC）の API

このセクションでは、ACR890 Phase 1 デバイスにのみ適用可能な接触インターフェイス関連のコマンドについて説明します。

附录 A.4.1. 接触インターフェイスモジュールをオンにする

この機能は、ICC モジュールをオンにするために使用されます。

```
int icc_open(void)
```

パラメーター

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-ENODEV です。

必要条件

ヘッダ	acs_api.h で宣言される
ライブラリファイル	libacs_api.so を使用する

附录 A.4.2. 接触インターフェイスモジュールをオフにする

この機能は、ICC モジュールをオフにするために使用されます。

```
int icc_close(void)
```

パラメーター

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-ENODEV です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4.3. 接触カードが存在するかどうかを確認する

この関数は、指定された接触カードの状態を確認するために使用されます。

```
int icc_slot_check(enum icc_slot idx)
```

パラメ―タ―

```
enum icc_slot {
ICC_SLOT_ID_0,
SAM_SLOT_ID_1,
SAM_SLOT_ID_2,
ICC_SLOT_MAX
};
```

idx [in] 指定されたスロット（IFD）のインデックス。

返ってきた数値

成功した場合、戻り値は 0 です（カードがある）。

失敗した場合、戻り値は≠0 です（カードが存在しない）。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4.4. 接触式スマートカードを電気入れる

この機能は、接触スマートカードをオンにするために使用されます。

```
int icc_power_on(enum icc_slot ifd, unsigned char *atr, unsigned int
*atrLen)
```

パラメ—タ—

idx [in] 指定されたスロット（IFD）のインデックス。

atr [out] 返された ATR データのバッファ

patrLen [out] 返された ATR の長さ

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は ≠ 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4.5. 接触式スマートカードの電源をオフにする

この機能は、接触スマートカードをオフにするために使用されます。

```
int icc_power_off(enum icc_slot idx)
```

パラメ—タ—

idx [in] 指定されたスロット（IFD）のインデックス。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は ≠ 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4.6. 接触スマートカードに PPS を送信する

このコマンドは接触式カードに PPS 請求を送信するためです。

```
int icc_pps_set(enum icc_slot idx, unsigned char fidi)
```

パラメータ

idx [in] 指定されたスロット（IFD）のインデックス。

fidi [in] *fi* と *di* 数値。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は ≠ 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.4.7. 接触スマートカードの APDU の転送

この機能は、接触スマートカードに APDU コマンドを送信するために使用されます。

```
int icc_apdu_transmit(enum icc_slot idx, unsigned char *cmd,
unsigned long cmdLen, unsigned char *resp, unsigned long *respLen)
```

パラメータ

idx [in] 指定されたスロット（IFD）のインデックス。

cmd [in] 送信されていない APDU コマンドのバッファ。

cmdLen [in] 送信されていない APDU コマンドの長さ。

resp [out] 応答データへのおインタ。

respLen [out] 応答データの長さ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は ≠ 0 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(int argc,char *argv[])
{
   int ret = -EINVAL;
   enum icc_slot idx = ICC_SLOT_ID_0;
   unsigned int i = 0;
   unsigned char atr[33];
   unsigned long mlen = 0;
   unsigned char mfidi = 0x95;
   unsigned char txcmd[5] = {0x80,0x84,0x00,0x00,0x08};
   unsigned char rxcmd[256];
   unsigned long rxlen = 0;

   ret = icc_open();
   if(0 == ret)
   {
      ret = icc_slot_check(idx);
      if(0 == ret)
      {
         printf("Found card in slot %d!\n", idx);
         ret = icc_power_on(idx, atr, &mlen);
         if(0 == ret)
         {
            printf("ATR ");
            for(i = 0; i < mlen; i++)
            {
               printf("%02X ",atr[i]);
            }
            printf("\n");

         ret =icc_pps_set(idx, mfidi);
         if(0 == ret)
         {
            printf("Set PPS succeed!\n");
         }

         ret  =  icc_apdu_transmit(idx,  txcmd,  sizeof(txcmd),  rxcmd,
      &rxlen);
         if(0 == ret)
         {
            printf("RES ");
            for(I = 0; I < rxlen; i++)
            {
               printf("%02X ",rxcmd[i]);
            }
            printf("\n");
         }
         ret = icc_power_off(idx);
      }
      else
      {
         printf("Power on card %d failed!\n", idx);
```

```
            }
        }
        else
        {
            printf("No card in slot %d!\n", idx);
        }
    }
    icc_close();

    return ret;
}
```

附录 A.5. 非接触インターフェイス（PICC）の API

このセクションでは、ACR890 Phase 1 デバイスにのみ適用可能な非接触カード関連のコマンドについて説明します。

附录 A.5.1. 非接触インターフェイスモジュールをオンにする

この機能は、PICC モジュールをオンにするために使用されます。

```
int picc_open(void)
```

パラメーター

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.2. 非接触インターフェイスモジュールをオフにする

この機能は、PICC モジュールをオフにするために使用されます。

```
int picc_close(void)
```

パラメーター

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.3. 非接触式カードを読み取る

この機能は、ポーリングし、カードのステータスを返すために使用されます。

```
int picc_poll_card(struct picc_card *card)
```

パラメーター

```
enum picc_card_type {
PICC_TYPE_UNKNOWN = 0,
PICC_TYPE_A = 0x01,
PICC_TYPE_B = 0x02,
PICC_TYPE_FELICA212 = 0x04,
PICC_TYPE_FELICA424 = 0x08,
PICC_TYPE_END
   };
struct picc_card {
enum picc_card_type type;/* Card's type */
unsigned char uid[16];/* Card's UID */
unsigned char uidlength;/* Length of the card UID */
};
```

card [out] 返されるデータへのポインタは、返されたカードタイプおよび UID を示しています。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

注：この関数は *FeliCa* カードに適用していません。*FeliCa* ポーリングコマンドを使用してカード情報を取得できます（例 *2* を参照**非接触アンテナをオン/オフにする**）。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.4. 非接触式カードを活性化する

この関数は、非接触カードを活性化するために使用され、それから ATR を取得しています。

```
int picc_power_on(unsigned char *atr, unsigned char *atr_len)
```

パラメ—タ—

atr [out] 非接触カードの ATR の文字列を返します。

atrLen [out] 返された ATR の長さ。

注釈：

- ATR の最大のサイズは 32 バイトです。したがって、ATR 文字列の容器のメモリサイズは 32 バイトに等しくなければなりません。
- この関数は FeliCa カードに適用していません。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダー acs_api.h で宣言される。

ライブラリ libacs_api.so を使用する。

附录 A.5.5. 非接触式カードの活性化を取り消す

この関数は、非接触カードの活性化を取り消すために使用されます。

```
int picc_power_off(void)
```

パラメ—タ—

なし。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.6. 非接触式カードのデータを送信する

この機能は、APDU コマンドを送信するために使用されます。

```
int picc_transmit(unsigned char *cmd, unsigned long cmdLen,
unsigned char *resp, unsigned long *respLen)
```

パラメ—タ—

cmd[in] 送信されていない APDU コマンド。

cmdLen [in] 送信されていない APDU コマンドの長さ。

resp [out] 応答データへのおインタ。

respLen [out] 応答データの長さ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

注：応答バッファの最大の長さは 272 バイトです。ですから*resp 文字列コンテナの容量は必ず 272 バイトより大きいです。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.7. FeliCa カードのデータを送信する

この関数は Felica コマンドのみを伝送するためにつかわれます。

注：*v1.1.0* 以降のファームウェアがこの関数しかサポートできません。

```
int picc_felica_transmit(unsigned char *cmd, unsigned long cmdLen,
unsigned char *resp, unsigned long *respLen)
```

パラメーター

cmd[in] 送信されていない FeliCa コマンド。

cmdLen [in] 送信されていない FeliCa コマンドの長さ。

resp [out] 応答データへのおインタ。

respLen [out] 応答データの長さ。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

注：この関数は *FeliCa* カードのみに適用しています。応答バッファのサイズは *272* バイトです。したがって、* *resp* 文字列コンテナの記憶容量は *272* バイト以上でなければならない（非接触アンテナをオン/オフにする例 *2* を参照）。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

附录 A.5.8. 非接触アンテナをオン/オフにする

この機能は、磁場（13.56 MHz）をオン/オフにするために使用されます。

```
int picc_field_ctrl(enum field_ctrl mode)
```

パラメーター

mode [in] オン/オフモード。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

1. TypeA と TypeB に使う

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include "acs_api.h"

int main(int argc, char* argv[])
{
   int rv = 0, i = 0;
   unsigned char txbuf[5] = {0x00,0x84,0x00,0x00,0x08};
   unsigned int txlen = 5;
   unsigned char rxbuf[272];
   unsigned int rxlen = 0;
   unsigned char buf[36];
   unsigned int len = 0;
   struct picc_card info;

   picc_open();
   picc_field_ctrl(1);

   rv = picc_poll_card(&info);
   if(!rv)
   {
      printf("Card uid[%d]::",info.uidlength);
      for(i=0;i<info.uidlength;i++)
         printf("%02x ",info.uid[i]);
      printf("\n");
   }
   else
   {
      printf("poll card fail rv = %d\n",rv);
      picc_field_ctrl(0);
      picc_close();
      return 0;
   }

   len = 36;
   rv = picc_power_on(buf,&len);
   if(!rv)
   {
      printf("Card ATR[%d]::",len);
      for(i=0;i<len;i++)
         printf("%02x ",buf[i]);
      printf("\n");
   }
   else
   {
      printf("power on fail rv = %d\n",rv);
      picc_field_ctrl(0);
      picc_close();
      return 0;
   }
```

```
   rxlen = 272;
   rv = picc_transmit(txbuf,txlen,rxbuf,&rxlen);
   if(!rv)
   {
      printf("Command:");
      for(i=0;i<5;i++)
         printf("%02x ",tx1buf[i]);
      printf("\n");

      printf("Response:");
      for(i=0;i<rxlen;i++)
         printf("%02x ",rxbuf[i]);
      printf("\n\n");
   }
   else
   {
      printf("picc_transmit fail, return code = %d\n",rv);
   }

   picc_power_off();
   picc_field_ctrl(0);
   picc_close();

   return 0;
}
```

2. FeliCa に使う

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include "acs_api.h"

int main(int argc, char* argv[])
{
   int rv = 0, i = 0;

   unsigned char polling[6] = {0x06,0x00,0xff,0xff,0x00,0x00};
   unsigned char reqservice[17] =
{0x11,0x02,0x01,0x01,0x06,0x01,0x5F,0x03,0x34,0x03,0x03,0x48,0x39,0x8
8,0x39,0xC9,0x39};
   unsigned char rxbuf[272];
   unsigned long rxlen = 0;
   unsigned long rxlen = 0;
   unsigned char uid[8];
   unsigned char uid_len = 0;

   picc_open();
   picc_field_ctrl(1);


   printf( "Send Polling\n");
   txlen = 6;
   rxlen = 272;
```

```
            rv = picc_felica_transmit(polling,txlen,rxbuf,&rxlen);
            if(!rv)
            {
               printf("Command:");
               for(i=0;i<txlen;i++)
                  printf("%02x ",polling[i]);
               printf("\n");

               printf("Response:");
               for(i=0;i<rxlen;i++)
                  printf("%02x ",rxbuf[i]);
               printf("\n\n");

               if(rxlen > 2){
                  memcpy(uid,&rxbuf[2],8);
                  uid_len = 8;
               }
            }
            else
            {
               printf("picc_felica_transmit fail, return code = %d\n",rv);
            }

            if(uid_len == 8)
            {
               printf( "Send Request Service\n");
               memcpy( &reqservice[2], uid, 8);
               txlen = 17;
               rxlen = 272;
               rv = picc_felica_transmit(reqservice,txlen,rxbuf,&rxlen);
               if(!rv)
               {
                  printf("Command:");
                  for(i=0;i<txlen;i++)
                     printf("%02x ",reqservice[i]);
                  printf("\n");

                  printf("Response:");
                  for(i=0;i<rxlen;i++)
                     printf("%02x ",rxbuf[i]);
                  printf("\n\n");
               }
               else
               {
                  printf("picc_felica_transmit fail, return code = %d\n",rv);
               }
            }

            picc_field_ctrl(0);
            picc_close();

            return 0;
}
```

附录 A.6. INI ファイル解析ツールの API

このセクションでは、初期化ファイルを解析する API 関数について説明します。

附录 A.6.1 INI キーワードの値を取得する

この関数は、ini ファイル（/etc/config.ini）からキーワード値を得るために使用されます。

```
Int get_a_ini_key_value(const char * module_name, const char * key_name,
char *key_value)
```

パラメーター

module_name [in] .ini ファイルのセクション名。

key_name [in] .ini ファイル内のキーワード名。

key_value [out] バッファへのポインタは、返されたキーワード文字列値を保存します。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;
   char key_value[255];

   ret = get_a_ini_key_value (“lcd”, “brightness”, key_value); //call
api to get brightness value
   if(0 == ret)
   {//show out the brighness you get from ini file just now.
      printf("[lcd]\nbrightness=%s\n”, key_value);
   }

   return ret;
}
```

附录 A.6.2. INI キーワードの値を設定する

この関数は、ini ファイル（/etc/config.ini）のキーワード値を設定するために使用されます。

```
int set_a_ini_key_value(const char * module_name, const char * key_name,
char *key_value)
```

パラメーター

module_name [in] .ini ファイルのセクション名。

key_name [in] .ini ファイル内のキーワード名。

key_value [in] ini ファイルに設定されるキーワード文字列値。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;

   ret = set_a_ini_key_value (“lcd”, “brightness”, 3); //call api to set
brightness value to 3 in the ini file.

   return ret;
}
```

附录 A.6.3. INI キーワードを追加する

この関数は、ini ファイル（/etc/config.ini）中でキーワードを追加するために使用されます。

```
int add_a_ini_key_value(const char * module_name, const char * key_name,
char *key_value)
```

パラメーター

module_name [in] ini ファイルに追加するセクション名。

key_name [in] ini ファイルに追加するキーワード名。

key_value [in] ini ファイルに追加するキーワード文字列値。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
   int ret;

   ret = add_a_ini_key_value (“lcd”, “brightness”, 3);

   return ret;
}
```

附录 A.6.4. 附录/etc/config.ini 内のすべてのキーワードに応じて、ハードウェアの値を設定する

この関数は、ini ファイル（/etc/config.ini）内のすべてのキーワードの値に応じてすべてのハードウェアを設定するために使用されます。

```
void ini_init_hw_all(void);
```

パラメーター

なし。

返ってきた数値

なし。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
    ini_init_hw_all();

    return 0;
}
```

附录 A.6.5. 指定したキーワードに応じて、ハードウェアの値を設定する

この関数は、ini ファイル（/etc/config.ini）内の指定したキーワードに応じて、ハードウェアを設定するために使用されます。

```
int record_set_to_hw (const char * module_name, const char * key_name,
const char *key_value)
```

パラメーター

module_name [in] .ini ファイルのセクション名。

key_name [in] .ini ファイル内のセクションのキーワード名。

key_value [in] ハードウェア中で設定する値。

返ってきた数値

成功した場合、戻り値は 0 です。

失敗した場合、戻り値は-1 です。

必要条件

ヘッダ acs_api.h で宣言される

ライブラリファイル libacs_api.so を使用する

サンプルコード

```
int main(void)
{
    int ret;

    ret = record_set_to_hw (“lcd”, “brightness”, 3); //call api to set
brightness value to 3

    return ret;
}
```