杰理蓝牙OTA开发说明（iOS端）

- 对应的芯片类型：AC692x，BD29
- APP开发环境：iOS平台，iOS 10.0以上，Xcode 11.0以上
- 基于「杰理蓝牙控制库SDK v1.4.0**(单设备版)**」 开发
- 对应于苹果商店上的APP: **【OTA Update】**
- 源码连接： https://github.com/Jieli-Tech/iOS-JL_OTA

声明

1. 本项目所参考、使用技术必须全部来源于公知技术信息，或自主创新设计。
2. 本项目不得使用任何未经授权的第三方知识产权的技术信息。
3. 如个人使用未经授权的第三方知识产权的技术信息，造成的经济损失和法律后果由个人承担。

版本

版本	日期	编辑	修改内容
v1.2	2020年12月09日	冯 洪鹏	更新文档
v1.1	2020年04月20日	冯 洪鹏	增加升级的错误回调
v1.0	2019年09月09日	冯 洪鹏	OTA升级功能

概述

本文档是为了后续开发者更加便捷移植杰理OTA升级功能而创建。

1、导入JL_BLEKit.framework

将*JL_BLEKit.framework*导入Xcode工程项目里，添加*Privacy - Bluetooth Peripheral Usage Description*和*Privacy - Bluetooth Always Usage Description*两个权限。

2、SDK具体使用的两种方式

第一种，使用SDK内的蓝牙连接API进行OTA：完全使用SDK。

第二种，使用自定义的蓝牙连接API进行OTA：所有BLE的操作都自行实现，SDK只负责对OTA数据包解析

从而实现OTA功能。

2.1、使用SDK内的蓝牙连接API进行OTA

1、支持的功能：

- BLE设备的扫描、连接、断开、收发数据、回连功能；
- BLE设备过滤；
- BLE设备握手连接；
- BLE连接服务和特征值设置；
- 获取设备信息；
- OTA升级能实现；

2、只能用到的类：

- **JL_BLEUsage**： 可设置BLE过滤、握手、参数；

  查看蓝牙状态；（详情看*2.1.1*，*2.1.2*，*2.1.3*，*2.1.4*，*2.1.5*）
- **JL_Entity**：BLE设备的模型类，记录设备的相关信息（如名字、UUID、UID、PID等）；
- **JL_Manager**：BLE扫描、连接、断开、回连、获取设备信息、OTA操作；

2.1.1、过滤BLE外设机制

```
    /*--- YES开启过滤，NO关闭过滤 ---*/
    JL_BLEUsage *usage = [JL_BLEUsage sharedMe];
    usage.bt_ble.BLE_FILTER_ENABLE = YES;

    /*--- 过滤码设置，赋值nil为默认值 ---*/
    usage.bt_ble.filterKey = nil;//一般情况赋值nil即可
```

2.1.2、与BLE外设握手机制

```
/*--- YES开启握手，NO关闭握手BLE直接连接 ---*/
JL_BLEUsage *usage = [JL_BLEUsage sharedMe];
usage.bt_ble.BLE_PAIR_ENABLE = YES;

/*--- 配对码设置，赋值nil为默认值 ---*/
usage.bt_ble.pairKey = nil;//一般情况赋值nil即可
```

2.1.3、回连BLE外设机制

```
/*--- 1、BLE外设自己断开后，APP的主动回连 ---*/
JL_BLEUsage *usage = [JL_BLEUsage sharedMeNO
usage.bt_ble.BLE_RELINK_ACTIVE = YES; //一般情况赋值NO即可

/*--- 2、iPhone关闭蓝牙后，开启蓝牙，APP的主动回连 ---*/
usage.bt_ble.BLE_RELINK = YES;//一般情况赋值NO即可
```

2.1.4、BLE连接服务和特征值

```
//一般情况以下设置不用更改，不设置就是默认这些值。
usage.bt_ble.JL_BLE_SERVICE = @"AE00"; //服务号
usage.bt_ble.JL_BLE_RCSP_W  = @"AE01"; //命令“写”通道
usage.bt_ble.JL_BLE_RCSP_R  = @"AE02"; //命令“读”通道
usage.bt_ble.JL_BLE_PAIR_W  = @"AE03"; //暂无使用
usage.bt_ble.JL_BLE_PAIR_R  = @"AE04"; //暂无使用
usage.bt_ble.JL_BLE_AUIDO_W = @"AE05"; //暂无使用
usage.bt_ble.JL_BLE_AUIDO_R = @"AE06"; //暂无使用
```

2.1.5、初始化SDK

```
//根据需求，按照文档的1、2、3、4点设置SDK
/*--- 初始化JL_SDK ---*/
[JL_Manager setManagerDelegate:self];
JL_BLEUsage *usage = [JL_BLEUsage sharedMe];
usage.bt_ble.BLE_PAIR_ENABLE = YES;
usage.bt_ble.BLE_FILTER_ENABLE = YES;
usage.bt_ble.BLE_RELINK_ACTIVE = NO;
usage.bt_ble.BLE_RELINK = NO;
usage.bt_ble.filterKey = nil;
usage.bt_ble.pairKey = nil;
```

2.1.6、扫描设备

```
//API通过【JL_Manager】使用
/**
开始扫描
*/
+(void)bleStartScan;
/**
停止扫描
 */
+(void)bleStopScan;

//监听通知【JL_BLEStatusFound】回调设备数组
JL_BLEUsage *JL_ug = [JL_BLEUsage sharedMe];
NSArray     *btEnityList = JL_ug.bt_EntityList;
```

2.1.7、连接和断开设备

```
//API通过【JL_Manager】使用
/**第一种连接方式:
连接蓝牙外设，如果外设不在已发现的外设列表中，则返回失败
@param peripheral 要连接的蓝牙外设
@return 返回是否成功发起连接
*/
+(BOOL)bleConnectToDevice:(CBPeripheral *)peripheral;

/**第二种连接方式:
 通过UUID的连接设备
 @param uuid 设备的UUID
 */
+(void)bleConnectDeviceWithUUID:(NSString*)uuid;

/**
 断开当前连接的蓝牙设备，不会影响下次的自动连接
 */
+(void)bleDisconnect;

//1、在发现的设备数组里以【JL_Entity】存储，详情可以看【JL_BLEUsage.h】头文件。
//2、连接成功，回调【JL_BLEStatusPaired】
//3、连接失败、BLE断开，回调【JL_BLEStatusDisconnected】
//4、手机蓝牙关闭，回调【JL_BLEStatusOff】
//5、手机蓝牙开启，回调【JL_BLEStatusOn】
//6、连接过程错误，回调【kJL_BLE_ERROR】
     /**
     *  错误代码:
     *  4001  BLE未开启
     *  4002  BLE不支持
     *  4003  BLE未授权
     *  4004  BLE重置中
     *  4005  未知错误
     *  4006  连接失败
     *  4007  连接超时
     *  4008  特征值超时
     *  4009  配对失败
     *  4010  设备UUID无效
     */
```

2.1.8、获取设备信息(必须)

```
//注意：API通过【JL_Manager】使用，连上设备必须先获取设备的信息!
//在连接成功后，即可调用获取设备信息(最好延时0.5秒执行)
    /*--- 获取设备信息 ---*/
    [JL_Manager cmdTargetFeatureResult:^(NSArray *array) {
        JL_CMDStatus st = [array[0] intValue];
        if (st == JL_CMDStatusSuccess) {
            NSLog(@"---> 正常获取设备信息.");

            JLDeviceModel *md = [JL_Manager outputDeviceModel];
            if (md.otaBleAllowConnect == JL_OtaBleAllowConnectNO) {
                //OTA 禁止连接后，断开连接清楚连接记录。
                [JL_Manager bleClean];
                [JL_Manager bleDisconnect];
                return;
            }

            /*--- 后续会用版本来决定是否要OTA升级 ---*/
            NSLog(@"---> 当前固件版本号：%@",md.versionFirmware);

            JL_OtaStatus upSt = md.otaStatus;
            if (upSt == JL_OtaStatusForce) {
                NSLog(@"---> 进入强制升级.");
                //此处必须将设备升级，否则无法使用
            }else{
                JL_OtaHeadset hdSt = md.otaHeadset;
                if (hdSt == JL_OtaHeadsetYES) {
                    //此处必须将设备升级，否则无法使用，针对单备份的耳机设备OTA操作。
                    //(一般情况不回来到这)
                }
            }
        }else{
            NSLog(@"---> 错误提示：%d",st);
        }
    }];
```

2.1.9、开始OTA升级

```
_otaData = [NSData dataWithContentsOfFile:@"升级文件的路径"];
[JL_Manager cmdOTAData:self.otaData Result:^(JL_OTAResult result, float progress) {
    if (result == JL_OTAResultUpgrading ||
        result == JL_OTAResultPreparing)
    {
        [self isUpdatingUI:YES];
        //NSLog(@"%.1f%%",progress*100.0f);
        NSString *txt = [NSString stringWithFormat:@"%.1f%%",progress*100.0f];
        self.updateSeek.text = txt;
        self.updateProgress.progress = progress;

        if (result == JL_OTAResultPreparing) self.updateTxt.text = kJL_TXT("校验文件中");
        if (result == JL_OTAResultUpgrading) self.updateTxt.text = kJL_TXT("正在升级");

        [self otaTimeCheck];//增加超时检测
    }else if(result == JL_OTAResultPrepared){
        NSLog(@"OTA is ResultPrepared...");
        [self otaTimeCheck];//增加超时检测
    }else{
        [self otaTimeClose];//关闭超时检测
```

```
        }

        if (result == JL_OTAResultSuccess) {
            NSLog(@"OTA 升级完成.");
            self.updateTxt.text = kJL_TXT("升级完成");
            self.updateProgress.progress = 1.0;
        }

        if (result == JL_OTAResultReboot) {
            NSLog(@"OTA 设备准备重启.");
            //self.updateTxt.text = kJL_TXT("设备准备重启");
            self.updateTxt.text = kJL_TXT("升级完成");
            [DFUITools showText:kJL_TXT("升级完成") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
                //[JL_Tools post:@"UI_CHANEG_VC" Object:@(1)];
                [JL_Manager bleConnectLastDevice];
            }];
        }

        if (result == JL_OTAResultFailCompletely) {
            self.updateTxt.text = kJL_TXT("升级失败");
            [DFUITools showText:kJL_TXT("升级失败") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }

        if (result == JL_OTAResultFailKey) {
            self.updateTxt.text = kJL_TXT("升级文件KEY错误");
            [DFUITools showText:kJL_TXT("升级文件KEY错误") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }

        if (result == JL_OTAResultFailErrorFile) {
            self.updateTxt.text = kJL_TXT("升级失败");
            [DFUITools showText:kJL_TXT("升级失败") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }
    }];
```

2.2、使用自定义的蓝牙连接API进行OTA

1、支持的功能：

- BLE设备握手连接；
- 获取设备信息；
- OTA升级能实现；

  注意：相对于2.1中描述的所有BLE操作都需自行实现。

2、只能用到的类：

- **JL_BLEAction**：实现BLE设备握手连接；(可选)
- **JL_Manager**：只能用**获取设备信息、OTA升级**的APIs；

2.2.1、初始化SDK (跟2.1.5有点区别)

```
    //安装JLSDK即可，无其他设置，因为蓝牙控制权都不在SDK内部。
    //这种情况SDK相当于OTA数据的解析器的角色。
    [JL_Manager installManager];
```

2.2.2、为SDK部署蓝牙传输通路

```
    //1、外部蓝牙连成功的回调处，Post以下通知给SDK；
    [JL_Tools post:kUI_JL_BLE_PAIRED Object:nil];//详情看2.2.3

    //2、外部蓝牙数据接收回调处，将数据一起Post以下通知给SDK；
    [JL_Tools post:kJL_RCSP_RECEIVE Object:data];

    //3、SDK请求外部蓝牙帮忙发送数据，请监听通知【kJL_RCSP_SEND】
    if ([name isEqual:kJL_RCSP_SEND]) {
        NSData *bleData = [note object];
        [bt_ble writeRcspData:bleData];//此处用外部蓝牙发数API，将数据发给设备即可。
    }

    //4、外部蓝牙断开的回调处，Post以下通知给SDK；
    [JL_Tools post:kUI_JL_BLE_DISCONNECTED Object:nil];
```

2.2.3、BLE握手连接

```
/**
 蓝牙设备配对
 @param pKey 配对码（默认传nil）
 @param bk   配对回调YES：成功 NO：失败
 */
-(void)bluetoothPairingKey:(NSData*)pKey Result:(ATC_Block)bk

//外部蓝牙更新通知特征的状态的回调处实现，以下：
#pragma mark - 更新通知特征的状态
- (void)peripheral:(CBPeripheral *)peripheral didUpdateNotificationStateForCharacteristic:
(nonnull CBCharacteristic *)characteristic
             error:(nullable NSError *)error
{
    if (error) { NSLog(@"Err: Update NotificationState For Characteristic fail.");}
    //NSLog(@"-----> %@ %d",characteristic.UUID.UUIDString,characteristic.isNotifying);

    if (characteristic.isNotifying) {
        if ([characteristic.UUID.UUIDString containsString:JL_BLE_RCSP_R])
        {
            [[JL_BLEAction sharedMe] bluetoothPairingKey:nil Result:^(BOOL ret) {
                if (ret == YES) {
                     NSLog(@"---->握手成功，发出蓝牙连接成功的通知。");
                     [JL_Tools post:kUI_JL_BLE_PAIRED Object:nil];
                }else{
                     NSLog(@"---->握手失败，断开蓝牙。");
                }
```

```
            }];

            //若设备不需要握手流程，则此处直接[JL_Tools post:kUI_JL_BLE_PAIRED Object:nil]即可。
        }
    }
}
```

2.2.4、获取设备信息 (同2.1.8)

2.2.5、开始OTA升级 (与2.1.9有点区别)

```
    //注意：由于使用的是外部的蓝牙连接流程，OTA过程可能需要断开重连BLE外设，
    //必须在以下回调【JL_OTAResultReconnect】中实现，否则流程会走不下！！！

    _otaData = [NSData dataWithContentsOfFile:@"升级文件的路径"];

    [JL_Manager cmdOTAData:self.otaData Result:^(JL_OTAResult result, float progress) {
        if (result == JL_OTAResultUpgrading ||
            result == JL_OTAResultPreparing)
        {
            [self isUpdatingUI:YES];
            //NSLog(@"%.1f%%",progress*100.0f);
            NSString *txt = [NSString stringWithFormat:@"%.1f%%",progress*100.0f];
            self.updateSeek.text = txt;
            self.updateProgress.progress = progress;

            if (result == JL_OTAResultPreparing) self.updateTxt.text = kJL_TXT("校验文件中");
            if (result == JL_OTAResultUpgrading) self.updateTxt.text = kJL_TXT("正在升级");

            [self otaTimeCheck];//增加超时检测
        }else if(result == JL_OTAResultPrepared){
            NSLog(@"OTA is ResultPrepared...");
            [self otaTimeCheck];//增加超时检测

        }else if(result == JL_OTAResultReconnect){
            [self otaTimeCheck];//增加超时检测
            //1、前提：若没有使用SDK内的蓝牙连接流程。
            //    则需用外部蓝牙API连接设备，再走获取设备信息，然后判断到强制升级的标志
            //    继续调用此API进行OTA升级。(此处必须重连设备，否则升级无法成功!!!)

            //2、前提：若使用了SDK内部的蓝牙连接流程，则此处无需做任何连接操作。
        }else{
            [self otaTimeClose];//关闭超时检测
        }

        if (result == JL_OTAResultSuccess) {
            NSLog(@"OTA 升级完成.");
            self.updateTxt.text = kJL_TXT("升级完成");
            self.updateProgress.progress = 1.0;
        }

        if (result == JL_OTAResultReboot) {
            NSLog(@"OTA 设备准备重启.");
            //self.updateTxt.text = kJL_TXT("设备准备重启");
            self.updateTxt.text = kJL_TXT("升级完成");
            [DFUITools showText:kJL_TXT("升级完成") onView:self.view delay:1.0];
```

```
            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
                //[JL_Tools post:@"UI_CHANEG_VC" Object:@(1)];
                [JL_Manager bleConnectLastDevice];
            }];
        }

        if (result == JL_OTAResultFailCompletely) {
            self.updateTxt.text = kJL_TXT("升级失败");
            [DFUITools showText:kJL_TXT("升级失败") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }

        if (result == JL_OTAResultFailKey) {
            self.updateTxt.text = kJL_TXT("升级文件KEY错误");
            [DFUITools showText:kJL_TXT("升级文件KEY错误") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }

        if (result == JL_OTAResultFailErrorFile) {
            self.updateTxt.text = kJL_TXT("升级失败");
            [DFUITools showText:kJL_TXT("升级失败") onView:self.view delay:1.0];

            [DFAction delay:1.5 Task:^{
                [self isUpdatingUI:NO];
            }];
        }
    }];
```