

YCProductSDK 修订记录

版本	内容描述	修订日期	修订人
1.0.0	1.Swift初始版本	2022-09-16	刘
1.1.0	1.支持杰理平台	2023-05-06	刘
1.2.0	1.合并不同平台代码	2023-12-30	刘
2.0.0	1.通用版本	2024-01-31	刘
2.0.1	1. 删除不常用的API	2021-02-27	刘

1. 概述

本文档将对蓝牙设备中使用到的功能相关的API及使用场景进行解释说明。
此文档仅适用于玉成创新的手环或手表等可穿戴式设备。

1.1 适用读者

对于使用此文档进行APP开发的工程师，应该有以下几点基本的技能：

1. 有基本的iOS开发经验
2. 需要掌握Swift语言，有Objective-C与Swift相互调用的知识储备。
3. 了解iOS中的蓝牙开发基本流程

1.2 相关术语

App: 本文指的是手机端或平板电脑上运行的应用程序

设备: 本文指的是可穿戴式硬件设备:如手环、手表等

上传: 指的是设备向App发送数据

下发: 指的是App向设备发送数据

1.3 说明

1.文档中的所有API都在对应的Demo会有演示，对于手环的功能，可以参照我们发布在AppStore中的SmartHealth 应用程序进行使用。结合使用的经验再阅读此文档将会大大提高效率。

2.文档中的API并不适用于所有的设备，即设备可能只支持API中的部分功能，可以通过API的返回值或属性来判是否可用。

3.在开发调试阶强烈建议您打开日志开关，方便出错信息帮助查找定位问题。

4.此文档只对Swift版本进行说明，如果您使用Objective-C语言开发，可以结合文档再借助Xcode的智能提示，调用对应的方法。

5.对API的介绍，原则是依据指令的分类进行说明，如果有涉及到相关的功能，可能会放在一个章节或者会引入参考API的标记。

2. SDK集成说明

2.1 SDK说明

2.1.1 资源概述

SDK中有部分功能如OTA长级等会依赖一些第三方的库文件，在SDK的资源包中都有提供，在需要用的时候会在文档中提到。

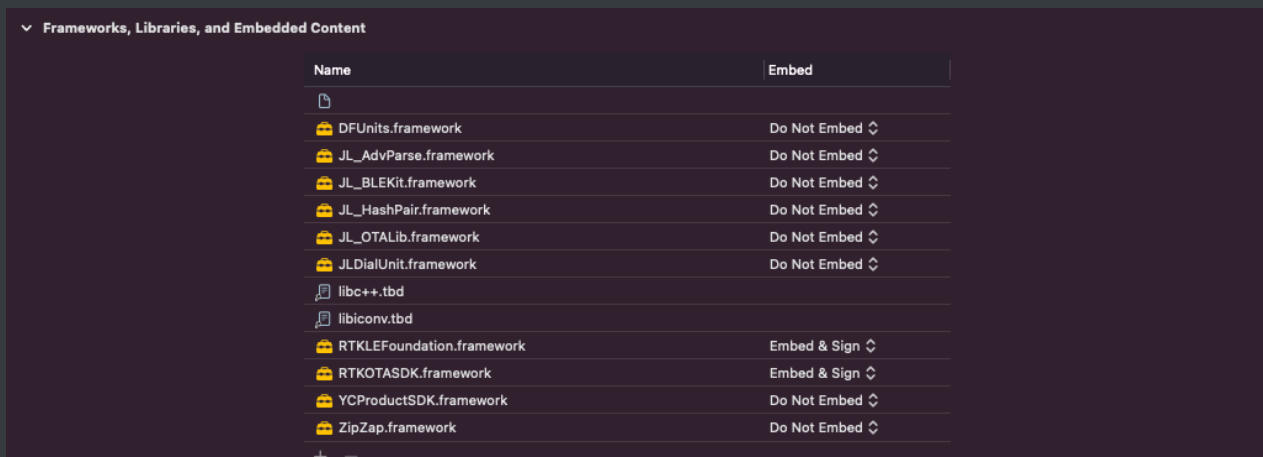
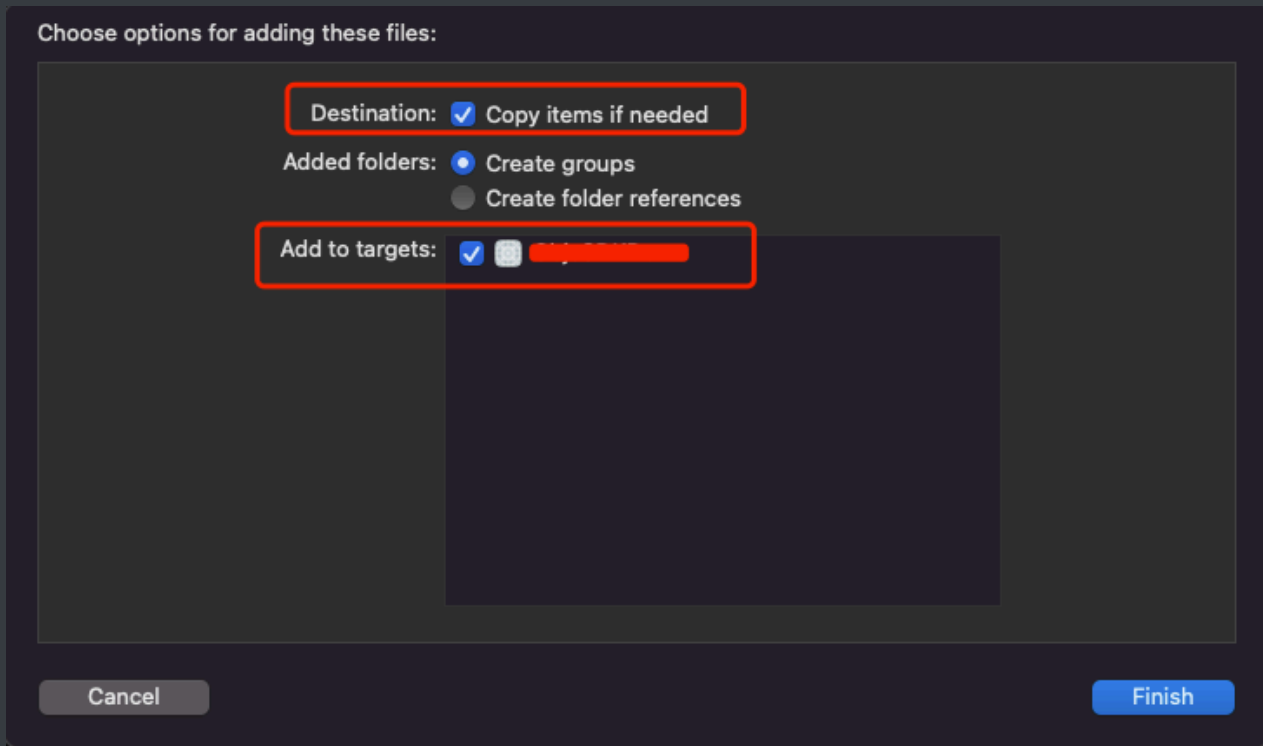
此外特别注意，SDK不提供模拟器版本，因为在模拟器上无法调试BLE且我们依赖的第三方库部分也不支持模拟器环境下运行。

2.1.2 使用说明

1. API出现了大量的类型声明，只有在使用到时，才会进行说明。
2. 如果在阅读文档时，发现某些类型没有给出说明，可以使用全局搜索的方式，可能在其它地方已经给出。
3. 如果整篇文档都没有给出定义，可以直接在Xcode写出此类型，再使用Xcode 跳转功能，跳转到框架内部定义查看。
4. 使用SDK的过程中，如果遇到问题，可以先尝试使用我们的应用程序或Demo以确定是否设备本身存在故障，或者SDK内部存在bug等，及时向我们反馈。

2.2 集成SDK

1. 直接将 Lib中的库文件拖拽到工程中去。

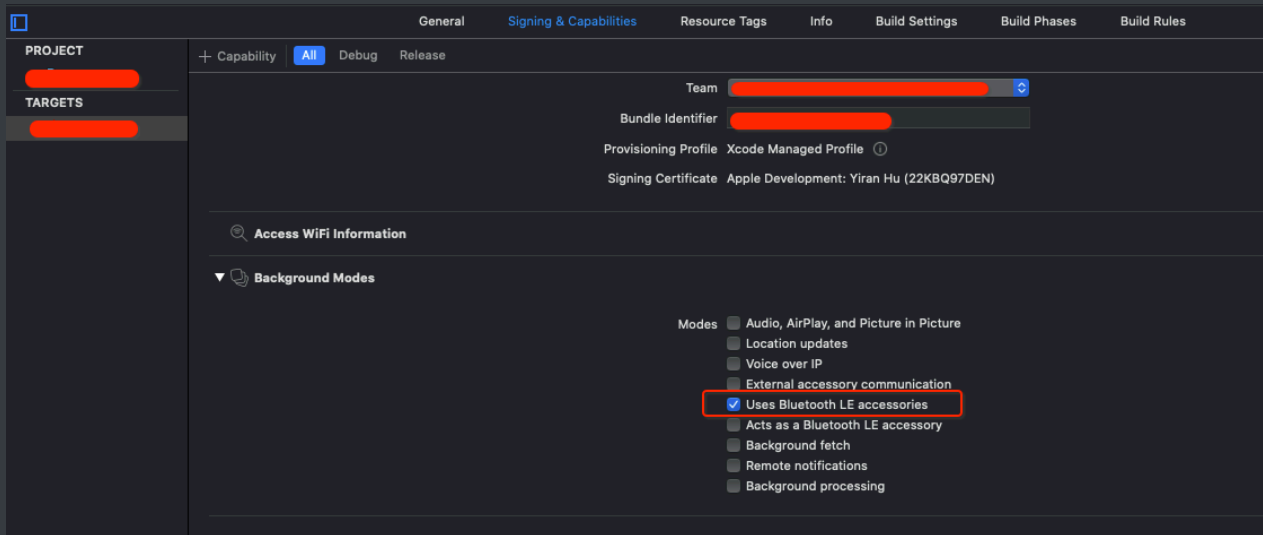


2. 配置 info.plist文件

需要在info.plist中对使用蓝牙进行说明，iOS13前后的版本使用的key有所不同

```
NSBluetoothAlwaysUsageDescription (iOS 13+)  
NSBluetoothPeripheralUsageDescription
```

3. 配置蓝牙访问



4. 项目导入

```
// Swift项目的导入
import CoreBluetooth
import YCProductSDK
```

```
// OC项目的导入
#import CoreBluetooth;
#import YCProductSDK;
```

其它的第三方依赖库，在需要的时候导入。

2.3 SDK的日志打印

日志打印打印和日志保存可以独立设置不同的等级，SDK默认都是关闭。

```
/// 日志等级
@objc public enum YCProductLogLevel : Int {
    case off           // 关闭
    case normal        // 普通
    case info          // 重要
    case error         // 错误
}

/// 设置SDK的日志等级开关(默认都是关闭)
/// - Parameters:
///   - printLevel: 控制台打印日志等级
```

```

/// - saveLevel: 保存日志等级
public static func setLogLevel(
    _ printLevel: YCProductLogLevel = .off,
    saveLevel: YCProductLogLevel = .off
)

/// 日志管理器
@objcMembers public class YCProductLogManager: NSObject {

    /// 读取日志文件中的数据
    /// - Returns: 日志内容
    public static func readLogFileData() -> String?

    /// 清除日志内容
    public static func clear()
}

```

2.4 SDK中的状态码

- 每个API的回调都会包含两个部分，一个是状态码，一个是具体的信息，具体信息是Any类型，主要是基本数据类型和class为主。
- 返回的具体信息将在具体的API上使用时进行说明，所有的状态统一在此进行说明。

```

@objc public enum YCProductState : Int {

    case unknow                // 蓝牙状态未知
    case resetting             // 蓝牙复位
    case unsupported           // 不支持蓝牙
    case unauthorized          // 蓝牙未授权
    case poweredOff            // 蓝牙关闭
    case poweredOn             // 蓝牙开启
    case disconnected           // 蓝牙断开连接
    case connected             // 蓝牙已连接
    case connectedFailed       // 蓝牙连接失败

    case succeed               // 成功
    case failed                // 失败

    case unavailable           // API不可用，设备不支持
    case timeout               // 超时
    case dataError             // 数据错误
    case crcError              // crc错误
    case dataTypeError         // 数据类型错误
}

```

```

    case noRecord // 没有记录
    case parameterError // 参数错误

    case alarmNotExist // 闹钟不存在
    case alarmAlreadyExist // 闹钟已存在
    case alarmCountLimit // 闹钟数量达到上限
    case alarmTypeNotSupport // 闹钟类型不支持
}

```

2.5 SDK初始化

在集成SDK后，需要调用一下初始化方法(必须)，SDK初始化方法会做一些简单的设置工作。

```

_ = YCProduct.shared

```

3. 设备搜索与连接

3.1 设备状态

▪ 方法

```

/// 设备状态变化
public static let deviceStateNotification: Notification.Name

/// 状态的key
public static let connecteStateKey: String

```

▪ 说明

- SDK会监听设备的连接状态，同时会以Notification的形式进行发送，App端可以全局监听此消息。
- Notification中的消息保存在 connecteStateKey 的key中，通过key可以获到对应的状态。（Objc调用时，key名称是 connecteStateKeyObjc）

▪ 使用举例

```

NotificationCenter.default.addObserver(
    self,
    selector: #selector(deviceStateChange(_:)),

```

```

        name: YCProduct.deviceStateNotification,
        object: nil
    )

    @objc private func deviceStateChange(_ ntf: Notification) {
        guard let info = ntf.userInfo as? [String: Any],
              let state = info[YCProduct.connecteStateKey] as? YCProductState else {
            return
        }
        print("=== stateChange \(state.rawValue)")
    }
}

```

3.2 搜索设备

▪ 方法

```

/// 开始扫描设备
/// - Parameters:
///   - delayTime: 延时停止搜索，默认3秒
///   - completion: 返回结果
public static func scanningDevice(
    delayTime: TimeInterval = 3.0,
    completion: (([CBPeripheral], NSError?) -> ())?
)

/// 设备厂商ID
public static var filterProductID:[Int]

```

▪ 说明

- 调用方法搜索设备结束后，会自动停止，搜索时间由delayTime来决定，默认时间是3秒，也可以设置任意时间，建议使用默认值。
- 搜索到的设备将会在结束时在回调中返回。
- filterProductID如果不设置默认支持玉成所有的型号，如果指定了ID，则只能搜索指定型号的设备。

▪ 使用举例

```

YCProduct.scanningDevice(delayTime: 3.0) { devices, error in
    for device in devices {
        print(device.name ?? "", device.identifier.uuidString)
    }
}

```

3.3 连接设备

- 方法

```
/// 连接设备
/// - Parameter peripheral: 用户选择需要连接的设备
public static func connectDevice(
    _ peripheral: CBPeripheral,
    completion: ((YCProductSDK.YCProductState, NSError?) -> ())?
)
```

- 说明
 - 可以从搜索到的设备中，选择任意一个设备进行连接。
 - 选择 已连接设备后，需要等待一段时间，连接结果会在回调中返回。
- 使用举例

```
YCProduct.connectDevice(device) { state, error in
    if state == .connected {
        print("connected")
    }
}
```

3.4 断开设备连接

- 方法

```
/// 断开连接
/// - Parameter peripheral: 当前连接的设备
public static func disconnectDevice(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ error: NSError?) -> ())?
)

/// 回连标记
open var isReconnectEnable: Bool = true
```

- 说明
 - 将已连接的设备，调用 此方法，将会与SDK断开，注意不会断开与iOS系统的连接。
 - 断开连接后，SDK会清除回连标记，不会回连此设备。
- 使用举例


```
YCProduct.disconnectDevice(devcie) { state, _ in  
  
}
```

3.5 获取当前连接的设备

```
/// 当前的连接的外设  
public var currentPeripheral: CBPeripheral? { get }  
  
/// 当前连接的所有的外设（保留参数，扩展使用）  
public var connectedPeripherals: [CBPeripheral] { get }
```

- 说明
 - SDK会将连接的设备进行保存到数组中，同时会将最后一个连接的设备保存到当前设备中，方便使用。
- 使用举例

```
let devcie = YCProduct.shared.currentPeripheral  
let lastDevice = YCProduct.shared.connectedPeripherals.last
```

3.6 设备的回连与长连接

1. SDK在连接成功后，会将最后一个连接成功的设备参数进行保存，一旦设备断开连接，SDK将会主动重新连接此设备。
2. 如果调用SDK中的断开 已连接设备的方法(3.3), SDK将不会回连此设备。
3. 如果App想实现一直保持连接的功能，由App本身实现，SDK不实现后台保持长连接的功能。

4. 获取健康数据

- 获取设备的健康数据，指的是查询设备通过主动检测或自动检测的心率，血压，睡眠，血氧等数据。
- 设备中的健康数据类型都不相同，请依据设备的返回值与功能支持开关来进行调用。
- 注意：通过combinedData获取到的 心率，步数，睡眠，血压，不要使用，也就是说，通过此类型获取到的数据，只能使用血氧、呼吸率、体温、体脂等。而心率、步数、睡眠、血压这几个只能通过单独的类型去获取。

- 设备中的数据 不会主动删除，所以App获取以后，应该主动删除，否则下次获取时，会获得相同的数据，且一旦设备中的数据超过存储大小将会被自动删除。

4.1 获取数据

- 方法

```

/// 查询健康数据的类型定义
@objc public enum YCQueryHealthDataType: UInt8 {

    case step                // 步数数据
    case sleep               // 睡眠数据
    case heartRate           // 心率数据
    case bloodPressure       // 血压数据
    case combinedData        // 组合数据（血氧，呼吸率，温度，体脂，
    hrv, cvrr)

    case sportModeHistoryData // 运动历史数据
    case invasiveComprehensiveData // 有创测量综合数据
}

/// 查询健康数据
/// - Parameters:
///   - peripheral: 当前设备
///   - dataType: 数据类型 YCQueryHealthDataType
///   - completion: 调用结果
public static func queryHealthData(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCQueryHealthDataType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 查询到应的数据后，结果会在 response中保存，每个类型返回的结果也不相同，需要转换成对应的对象。具体的类型已经逐个列出，参考4.3的内容。
- 使用举例

```

// 查询睡眠
YCProduct.queryHealthData(dataType: YCQueryHealthDataType.sleep) { state,
response in

    if state == .succeed, let datas = response as? [YCHealthDataSleep] {
        for info in datas {

```

```

        print(info.startTimeStamp,
              info.endTimeStamp,
              info.sleepDetailDatas
        )
    }
} else {
    print("No data")
}
}

// 查询组合数据
YCProduct.queryHealthData(datatType: YCQueryHealthDataType.combinedData) {
    state, response in

        if state == .succeed, let datas = response as?
[YCHealthDataCombinedData] {
            for info in datas {
                print(info.startTimeStamp,
                      info.bloodOxygen,
                      info.respirationRate,
                      info.temperature,
                      info.fat
                )
            }
        } else {
            print("No data")
        }
    }
}

// 其它类型请参照Demo演示

```

4.2 删除数据

- 方法

```

/// 删除健康数据的类型定义
@objc public enum YCDeleteHealthDataType: UInt8 {

    case step
    case sleep
    case heartRate
    case bloodPressure
    case combinedData
}

```

```

        case sportModeHistoryData
        case invasiveComprehensiveData
    }

    /// 删除健康数据
    /// - Parameters:
    ///   - peripheral: 当前连接的设备
    ///   - dataType: 删除的数据类型 YCDeleteHealthDataType
    ///   - completion: 删除是否成功
    public static func deleteHealthData(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        dataType: YCDeleteHealthDataType,
        completion: ((_ state: YCProductState, _ response: Any?) -> (()))?
    )

```

- 说明：
 - 调用方法时，传入具体的类型到 dataType 中即可，如果删除成功则 state 会返回 succeed，否则表示操作失败。
- 使用举例

```

/// 删除步数
YCProduct.deleteHealthData(dataType: YCDeleteHealthDataType.step) { state,
response in

    if state == .succeed {
        print("Delete succeed")
    } else {
        print("Delete failed")
    }
}

/// 删除组合数据
YCProduct.deleteHealthData(dataType: YCDeleteHealthDataType.combinedData)
{ state, response in

    if state == .succeed {
        print("Delete succeed")
    } else {
        print("Delete failed")
    }
}

```

```
// 其它类型，请参考Demo演示
```

4.3 健康数据类型与对应的返回值类型

4.3.1 步数

```
// YCQueryHealthDataType.step
/// 步数信息
@objcMembers public class YCHealthDataStep : NSObject {

    /// 开始时间戳(秒)
    public var startTimeStamp: Int { get }

    /// 结束时间戳(秒)
    public var endTimeStamp: Int { get }

    /// 步数 (步)
    public var step: Int { get }

    /// 距离 (米)
    public var distance: Int { get }

    /// 卡路里(千卡)
    public var calories: Int { get }
}
```

4.3.2 睡眠

```
// YCQueryHealthDataType.sleep

/// 睡眠数据
@objcMembers public class YCHealthDataSleep : NSObject {

    /// 开始时间戳
    public var startTimeStamp: Int { get }

    /// 结束时间戳
    public var endTimeStamp: Int { get }

    /// 深睡次数 (如取值0xFFFF为新格式)
    public var deepSleepCount: Int { get }

    /// 深睡总时长 (单位:分)
```

```

    public var deepSleepMinutes: Int { get }

    /// 快速眼动时长（单位:分）
    public var remSleepMinutes: Int { get }

    /// 浅睡次数
    public var lightSleepCount: Int { get }

    /// 浅睡总时长（单位:分）
    public var lightSleepMinutes: Int { get }

    /// 深睡总时长（单位:秒） deepSleepCount == 0xFFFF 有效
    public var deepSleepSeconds: Int { get }

    /// 快速眼动时长（单位:秒） deepSleepCount == 0xFFFF 有效
    public var remSleepSeconds: Int { get }

    /// 浅睡总时长（单位:秒） deepSleepCount == 0xFFFF 有效
    public var lightSleepSeconds: Int { get }

    /// 睡眠详细数据
    public var sleepDetailDatas: [YCProductSDK.YCHealthDataSleepDetail] {
get }
}

/// 深睡和浅睡类型
@objc public enum YCHealthDataSleepType : Int {
    case unknow                // 未知
    case deepSleep              // 深睡
    case lightSleep              // 浅睡
    case rem                    // 快速眼动
    case awake                  // 清醒
}

/// 睡眠详细数据
@objcMembers public class YCHealthDataSleepDetail : NSObject {

    /// 深睡或浅睡标记
    public var sleepType: YCProductSDK.YCHealthDataSleepType { get }

    /// 睡眠开始时间戳（秒）
    public var startTimeStamp: Int { get }

    /// 睡眠时长 单位：秒

```

```
    public var duration: Int { get }  
}
```

- 注意：如果deepSleepCount取值为0xFFFF则表示为新格式，使用deepSleepSeconds、remSleepSeconds、lightSleepSeconds，否则使用 deepSleepMinutes、remSleepMinutes、lightSleepMinutes。

4.3.3 心率

```
// YCQueryHealthDataType.heartRate  
/// 心率健康数据  
@objcMembers public class YCHealthDataHeartRate : NSObject {  
  
    /// 开始时间戳(秒)  
    public var startTimestamp: Int { get }  
  
    /// 模式  
    public var mode: YCProductSDK.YCHealthDataMeasureMode { get }  
  
    /// 心率值  
    public var heartRate: Int { get }  
}
```

4.3.4 血压

```
// YCQueryHealthDataType.bloodPressure  
/// 血压健康数据  
@objcMembers public class YCHealthDataBloodPressure : NSObject {  
  
    /// 开始时间戳  
    public var startTimestamp: Int { get }  
  
    /// 测量方式  
    public var mode: YCProductSDK.YCHealthDataBloodPressureMode { get }  
  
    /// 收缩压  
    public var systolicBloodPressure: Int { get }  
  
    /// 舒张压  
    public var diastolicBloodPressure: Int { get }  
}
```

```

/// 测量模式
@objc public enum YCHealthDataBloodPressureMode : UInt8 {
    case single                // 单次测量
    case monitor               // 自动监测
    case inflated              // 精准测量
}

```

4.3.5 组合数据

```

// YCQueryHealthDataType.combinedData
/// 组合数据
@objcMembers public class YCHealthDataCombinedData : NSObject {

    /// 开始时间戳
    public var startTimeStamp: Int { get }

    /// 步数（步）（无效值）
    public var step: Int { get }

    /// 心率值（无效值）
    public var heartRate: Int { get }

    /// 收缩压（无效值）
    public var systolicBloodPressure: Int { get }

    /// 舒张压（无效值）
    public var diastolicBloodPressure: Int { get }

    /// 血氧值
    public var bloodOxygen: Int { get }

    /// 呼吸率值
    public var respirationRate: Int { get }

    /// HRV
    public var hrv: Int { get }

    /// CVRR
    public var cvrr: Int { get }

    /// 温度
    public var temperature: Double { get }
}

```



```

    /// 温度是否有效
    public var temperatureValid: Bool { get }

    /// 体脂
    public var fat: Double { get }

    /// 血糖
    public var bloodGlucose: Double {get}
}

```

4.3.6 运动历史数据

```

// YCQueryHealthDataType.sportModeHistoryData

/// 运动历史数据
@objcMembers public class YCHealthDataSportModeHistory : NSObject {

    /// 开始时间戳(秒)
    public var startTimeStamp: Int { get }

    /// 结束时间戳(秒)
    public var endTimeStamp: Int { get }

    /// 步数 (步)
    public var step: Int { get }

    /// 距离 (米)
    public var distance: Int { get }

    /// 卡路里(千卡)
    public var calories: Int { get }

    /// 运动方式
    public var sport: YCProductSDK.YCDeviceSportType { get }

    /// 启动方式
    public var flag: YCProductSDK.YCHealthDataSportModeStartMethod { get }

    /// 心率
    public var heartRate: Int { get }

    /// 运动时间(有效运动时间)

```

```

    public var sportTime: Int { get }

    /// 最小心率
    public var minimumHeartRate: Int { get }

    /// 最大心率
    public var maximumHeartRate: Int { get }
}

/// 运动启动方式
@objc public enum YCHealthDataSportModeStartMethod : UInt8 {
    case app
    case device
}

```

4.3.7 有创测量综合数据

```

/// 有创测量组合数据
@objcMembers public class YCHealthDataInvasiveComprehensiveData: NSObject {

    /// 开始时间戳
    public var startTimeStamp: Int = 0

    /// 血糖
    public var bloodGlucose: Double = 0

    /// 尿酸
    public var uricAcid : UInt16 = 0

    /// 血酮
    public var bloodKetone: Double = 0
}

```

5. 获取设备的信息

5.1 支持功能

- 方法

```
extension CBPeripheral {  
    /// 支持功能列表  
    public var supportItems: YCProductSDK.YCProductFunctionSupportItems  
}
```

- 说明
 - 对于定制设备来说，功能是已经固定的，不是很有必要使用此参数。此参数只有设备连接成功后，才有效。
 - 此参数包含的属性过多，不在此文档列举，读者可以通过Xcode跳转到定义仔细查看。
- 使用举例

```
guard let device = YCProduct.shared.currentPeripheral else {  
    return  
}  
  
if device.supportItems.isSupportStep {  
    print("step")  
}  
  
if device.supportItems.isSupportBloodPressure {  
    print("blood pressure")  
}
```

5.2 设备基本信息

- 方法

```
public static func queryDeviceInfo(  
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,  
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?  
)
```



```

}

/// 电池电量状态
@objc public enum YCDeviceBatteryState : UInt8 {
    case normal          // 正常
    case low             // 低电量
    case charging        // 充电中
    case full            // 已充满
}

```

- 说明
 - 查询基本信息，会获得一个 YCDeviceBasicInfo 类型的对象，
- 使用举例

```

YCProduct.queryDeviceBasicInfo { state, response in
    if state == YCProductState.succeed,
        let info = response as? YCDeviceBasicInfo {
            print(info.batteryPower)
        }
}

```

5.3 mac地址

- 方法

```

/// 获取mac地址
public static func queryDeviceMacAddress(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// mac地址属性
extension CBPeripheral {
    public var macAddress: String
}

```

- 说明
 - 获取mac地址，可以直接通过访问它的属性来获得，如果属性中的值为空，再通过方法调用来获取。
- 使用举例

```
YCProduct.queryDeviceMacAddress { state, response in
    if state == YCProductState.succeed,
        let macaddress = response as? String {
            print(macaddress)
        }
}
```

5.4 设备型号

- 方法

```
/// 获取型号
public static func queryDeviceModel(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- 说明
 - 此方法主要是用于获取设备的类型编号，用于区分同一系列衍生的不同产品的型号。
- 使用举例

```
YCProduct.queryDeviceModel { state, response in
    if state == YCProductState.succeed,
        let name = response as? String {
            print(name)
        }
}
```

5.5 主题信息

- 方法

```
/// 获取主题
public static func queryDeviceTheme(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 主题信息
@objcMembers public class YCDeviceTheme : NSObject {
```

```

    /// 总数
    public var themeCount: Int { get }

    /// 当前主题索引
    public var themeIndex: Int { get }
}

```

- 说明
 - 获取到设备当前显示的主题索引和主题的总数
- 使用举例

```

YCProduct.queryDeviceTheme { state, response in
    if state == YCProductState.succeed,
        let info = response as? YCDeviceTheme {
            print(info.themeCount, info.themeIndex)
        }
}

```

5.6 获取芯片信息

- 方法

```

/// 获取芯片型号
public static func queryDeviceMCU(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// MCU
@objc public enum YCDeviceMCUType : UInt8 {
    case nrf52832
    case rtk8762c
    case rtk8762d
    case jl701n
}

```

- 说明
 - 获取芯片信息指的获取设备的MCU的厂家品牌和型号
- 使用举例

```

YCProduct.queryDeviceMCU { state, response in
    if state == .succeed,
    let mcu = response as? YCDeviceMCUType{
        print(mcu)
    } else if state == .unavailable {
        print("nrf52832")
    }
}

```

5.7 获取用户配置信息

- 方法

```

/// 获取用户配置信息
public static func queryDeviceUserConfiguration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 用户配置
@objcMembers public class YCProductUserConfiguration : NSObject {

    /// 步数目标
    public var stepGoal: Int { get }

    /// 卡路里目标
    public var calorieGoal: Int { get }

    /// 距离
    public var distanceGoal: Int { get }

    /// 睡眠目标(小时部分)
    public var sleepGoalHour: Int { get }

    /// 睡眠目标(分钟部分)
    public var sleepGoalMinute: Int { get }

    /// 身高(cm)
    public var height: Int { get }

    /// 体重(kg)
    public var weight: Int { get }
}

```



```
/// 性别
public var gender: YCProductSDK.YCSettingGender { get }

/// 年龄
public var age: Int { get }

/// 距离单位
public var distanceUnit: YCProductSDK.YCDeviceDistanceType { get }

/// 体重单位
public var weightUnit: YCProductSDK.YCDeviceWeightType { get }

/// 温度单位
public var temperatureUnit: YCProductSDK.YCDeviceTemperatureType { get }

/// 显示时间方式
public var showTimeMode: YCProductSDK.YCDeviceTimeType { get }

/// 久坐提醒
public var startHour1: UInt8 { get }
public var startMinute1: UInt8 { get }
public var endHour1: UInt8 { get }
public var endMinute1: UInt8 { get }
public var startHour2: UInt8 { get }
public var startMinute2: UInt8 { get }
public var endHour2: UInt8 { get }
public var endMinute2: UInt8 { get }
public var sedentaryReminderInterval: UInt8 { get }
public var sedentaryReminderRepeat: Set<YCProductSDK.YCDeviceWeekRepeat>
{ get }

/// 防丢
public var antiLostType: YCProductSDK.YCDeviceAntiLostType { get }
public var rssi: Int8 { get }
public var antiLostDelay: UInt8 { get }
public var antiLostDisconnectDelay: Bool { get }
public var antiLostRepeat: Bool { get }
public var infomationPushEnable: Bool { get }
public var infomationPushItems: Set<YCProductSDK.YCDeviceInfoPushType> {
get }

/// 左右手
public var wearingPosition: YCProductSDK.YCDeviceWearingPositionType {
get }
```



```

public var sleepReminderEnable: Bool { get }
public var sleepReminderStartHour: Int { get }
public var sleepReminderStartMinute: Int { get }

/// 日程开关
public var scheduleEnable: Bool { get }

/// 事件提醒开关
public var eventReminderEable: Bool { get }

/// 意外监测开关
public var accidentMonitorinEnable: Bool { get }

/// 体温报警开关
public var bodyTemperatureAlarm: Bool { get }
}

```

- 说明
 - API返回的是全部的字段，但具体的设备支持的配置信息可能不同。
 - 返回值中部分属性没有具体的类型解析，可以结合设置部分来阅读。
- 使用举例

```

YCProduct.queryDeviceUserConfiguration { state, response in
    if state == .succeed,
        let info = response as? YCProductUserConfiguration {
            print(info.age)
        }
}

```

6. 设置设备

6.1 时间

- 方法

```

/// 时间设置
/// - Parameters:
///   - peripheral: 外设
///   - year: 年 2000+

```

```

/// - month: 月 1 ~ 12
/// - day: 日 1 ~ 31
/// - hour: 时(24) 0 ~ 23
/// - minute: 分 0 ~ 59
/// - second: 分 0 ~ 59
/// - weekDay: 星期
/// - completion: 设置是否成功
public static func setTime(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    year: UInt16,
    month: UInt8,
    day: UInt8,
    hour: UInt8,
    minute: UInt8,
    second: UInt8,
    weekDay: YCWeekDay,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 设置星期
@objc public enum YCWeekDay : UInt8 {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
    case sunday
}

```

- 说明
 - 设备只支持公历时间，不能设置如佛历等其它时间类型。
 - SDK内部自动设置时间，一般不需要调用此API。
- 使用举例

```

// 2021/12/6 14:38:59 星期一
YCProduct.setTime(
    year: 2021,
    month: 12,
    day: 6,
    hour: 14,
    minute: 38,
    second: 59,

```

```

        weekDay: .monday) { state, response in
        if state == .succeed {
            print("success")
        } else {
            print("fail")
        }
    }
}

```

6.2 目标设置

- 方法

```

/// 设置步数目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - step: 步数目标(步)
///   - completion: 设置结果
public static func setDeviceStepGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    step: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 设置卡路里目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - calories: 卡路里目标(千卡)
///   - completion: 设置结果
public static func setDeviceCaloriesGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    calories: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 设置距离目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - calories: 距离目标(米)
///   - completion: 设置结果
public static func setDeviceDistanceGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    distance: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

```

)

/// 设置睡眠目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - hour: 时 0~ 23
///   - minute: 分 0 ~ 59
///   - completion: 设置结果
public static func setDeviceSleepGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8,
    minute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 运动时间目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - hour: 运动时间 小时部分
///   - minute: 运动时间 分钟部分
///   - completion: 设置结果
public static func setDeviceSportTimeGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8,
    minute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 有效步数目标
/// - Parameters:
///   - peripheral: 已连接设备
///   - effectiveSteps: 有效步数目标 (步)
///   - completion: 设置结果
public static func setDeviceEffectiveStepsGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    effectiveSteps: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明

- 如果参数赋值错误，或者是设备不支持目标设置，都会返回失败。

- 使用举例

```

// step goal
YCPProduct.setDeviceStepGoal(
    step: 10000) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// calories goal
YCPProduct.setDeviceCaloriesGoal(calories: 1000) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// distance goal
YCPProduct.setDeviceDistanceGoal(distance: 10000) { state, _ in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// sleep goal
YCPProduct.setDeviceSleepGoal(hour: 8, minute: 30) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// sport time goal
YCPProduct.setDeviceSportTimeGoal(hour: 1, minute: 20) { state, response in

    if state == .succeed {
        print("success")
    } else {

```

```

        print("fail")
    }
}

// effective step goal
YCProduct.setDeviceEffectiveStepsGoal(effectiveSteps: 8000) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
}

```

6.3 用户信息

▪ 方法

```

/// 用户信息设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - height: 身高 100 ~ 250cm
///   - weight: 体重 30 ~ 200 kg
///   - gender: 性别 YCDeviceGender
///   - age: 年龄 6 ~ 120
///   - completion: 设置结果
public static func setDeviceUserInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    height: UInt8,
    weight: UInt8,
    gender: YCDeviceGender,
    age: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 性别
@objc public enum YCDeviceGender: UInt8 {
    case male
    case female
}

```

▪ 说明

- 只针对用户的身体，体重，性别，年龄进行设置，注意每个参数的取值范围。

- 使用举例

```
YCProduct.setDeviceInfo(height: 180,
                        weight: 90,
                        gender: .male, age: 18) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.4 单位设置

- 方法

```
/// 单位设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - distance: 距离单位
///   - weight: 体重单位
///   - temperature: 温度单位
///   - timeFormat: 时间格式12小时制/24小时制
///   - bloodGlucose: 血糖单位 mmol/l mg/dl
///   - completion: 设置结果
public static func setDeviceUnit(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    distance: YCDeviceDistanceType = .km,
    weight: YCDeviceWeightType = .kg,
    temperature: YCDeviceTemperatureType = .celsius,
    timeFormat: YCDeviceTimeType = .hour24,
    bloodGlucose: YCDeviceBloodGlucoseType = .millimolePerLiter,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 距离单位
@objc public enum YCDeviceDistanceType: UInt8 {
    case km
    case mile
}

/// 体重单位
@objc public enum YCDeviceWeightType: UInt8 {
```

```

        case kg
        case lb
    }

    /// 温度单位
    @objc public enum YCDeviceTemperatureType: UInt8 {
        case celsius
        case fahrenheit
    }

    /// 时间格式
    @objc public enum YCDeviceTimeType: UInt8 {
        case hour24
        case hour12
    }

    /// 血糖单位
    @objc public enum YCDeviceBloodGlucoseType: UInt8 {
        case millimolePerLiter           // mmol/L
        case milligramsPerDeciliter      // mg/dl
    }

```

- 说明
 - 单位设置用于显示显示数值格式
- 使用举例

```

YCProduct.setDeviceUnit(distance: .km,
                        weight: .kg,
                        temperature: .celsius,
                        timeFormat: .hour24) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.5 久坐提醒

- 方法

```

/// 久坐提醒

```

```

/// - Parameters:
///   - peripheral: 已连接设备
///   - startHour1: 开始时间1 小时 0 ~ 23
///   - startMinute1: 开始时间1 分钟: 0 ~ 59
///   - endHour1: 结束时间1 小时 0 ~ 23
///   - endMinute1: 结束时间1 分钟: 0 ~ 59
///   - startHour2: 开始时间2 小时 0 ~ 23
///   - startMinute2: 开始时间2 分钟: 0 ~ 59
///   - endHour2: 结束时间2 小时 0 ~ 23
///   - endMinute2: 结束时间2 分钟: 0 ~ 59
///   - interval: 15 ~ 60 分钟
///   - repeat: 重复 YCDeviceWeekRepeat
///   - completion: 设置结果
public static func setDeviceSedentary(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    startHour1: UInt8,
    startMinute1: UInt8,
    endHour1: UInt8,
    endMinute1: UInt8,
    startHour2: UInt8,
    startMinute2: UInt8,
    endHour2: UInt8,
    endMinute2: UInt8,
    interval: UInt8,
    repeat: Set<YCDeviceWeekRepeat>,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 星期重复时间
@objc public enum YCDeviceWeekRepeat: UInt8 {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
    case sunday
    case enable
}

```

■ 说明

- 久坐提醒 只能设置设置两上个时间段
- 注意间隔时间的取值范围, 时间的取值都为24小时制。

- YCDeviceWeekRepeat 的最后一个值是时间使能开关，如果包含 YCDeviceWeekRepeat.enable 表示此参数有效，否则无效。
- 使用举例

```
YCProduct.setDeviceSedentary(startHour1: 9,
    startMinute1: 0,
    endHour1: 12,
    endMinute1: 30,
    startHour2: 13,
    startMinute2: 30,
    endHour2: 18,
    endMinute2: 00,
    interval: 15,
    repeat: [
        .monday,
        .tuesday,
        .wednesday,
        .thursday,
        .friday,
        .enable
    ]
) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.6 防丢

- 方法

```
/// 防丢设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - antiLostType: 防丢类型
///   - completion: 设置结果
public static func setDeviceAntiLost(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    antiLostType: YCDeviceAntiLostType = .middleDistance,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
```

```

    )

    /// 防丢类型
    @objc public enum YCDeviceAntiLostType: UInt8 {
        case off // 关闭
        case closeRange // 近距离
        case middleDistance // 中距离
        case longDistance // 远距离
    }

```

- 说明
 - 防丢指的是设备与手机连接信号变弱或断连后，手环会发生震动。防丢类型中的后三个取值效果是相同的。
- 使用举例

```

YCProduct.setDeviceAntiLost(antiLostType: .middleDistance) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.7 通知提醒开关

- 方法

```

/// 设置消息提醒类型
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnabled: 是否开启
///   - infoPushType: 类型
///   - completion: 设置结果
public static func setDeviceInfoPush(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool, infoPushType: Set<YCDeviceInfoPushType>,
    completion: (( _ state: YCProductState, _ response: Any?) -> ()))?
)

/// 通知提醒类型
@objc public enum YCDeviceInfoPushType: UInt {

```

```

    case call
    case sms
    case email
    case wechat
    case qq
    case weibo
    case facebook
    case twitter

    case messenger
    case whatsapp
    case linkedIn
    case instagram
    case skype
    case line
    case snapchat
    case telegram

    case other
    case viber
    case zoom
    case tiktok
    case kakaoTalk
}

```

■ 说明

- 设备中的通知提醒是基于iOS的ANCS服务来实现，API只能设置是否显示对应类型的消息。
- 设备第一次连接手机时，iOS会弹出两个界面，分别是是否允许配对及是否允许通知显示，全部要同意，否则通知提醒将无法使用。

■ 使用举例

```

// on
YCProduct.setDeviceInfoPush(isEnable: true,
                             infoPushType: [.call, .qq, .weChat] ) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
}

```

```
// off
YCProduct.setDeviceInfoPush(isEnable: false,
                             infoPushType: [.call, .qq, .weChat ] ) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
}
```

6.8 健康监测

- 健康监测指的是设备会在固定时间测量对应的数据并保存。
- 健康监测中，心率监测和温度监测是通用的功能，也可以满足约大部分的场景。SDK提供了一个独立的API用于设置(6.8.1，效果等同于同时调用6.8.2和6.8.3)，对于一般设备来说，只需要舍使用此API，其它如血氧监测API(6.8.4开始的API)只有个别设备才可能到到。

6.8.1 健康监测

- 方法

```
/// 健康监测
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnable: 是否使能
///   - interval: 监测间隔 1 ~ 60分钟
///   - completion: 设置结果
public static func setDeviceHealthMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool, interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明
 - 设备会按照自动的时间测量心率，血压，血氧，呼吸率，体温。时间越微短，功耗越大。
 - 调用此方法的作用与同时调用6.8.2和6.8.3的方法是相同的。

- 使用举例

```
YCProduct.setDeviceHealthMonitoringMode(isEnable: true, interval: 60) {
    state, response in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.8.2 心率监测 (废弃)

- 方法

```
/// 心率监测
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnable: 是否使能
///   - interval: 监测间隔 1 ~ 60分钟
///   - completion: 设置结果
public static func setDeviceHeartRateMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool, interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明
 - 心率监测开启后，设备会按照自动的时间测量心率，血压，血氧，呼吸率。时间越短，功耗越大。
- 使用举例


```
YCProduct.setDeviceHeartRateMonitoringMode(isEnable: true, interval: 60) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.8.3 温度监测 (废弃)

- 方法

```
/// 温度监测
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnable: 是否开启
///   - interval: 监测间隔 1 ~ 60分钟
///   - completion: 设置结果
public static func setDeviceTemperatureMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明

- 温度监测开启后，设备将自动按照设定的时间进行温度测量并记录。
- 建议将时间间隔与心率监测间隔保持一致。

- 使用举例

```
YCProduct.setDeviceTemperatureMonitoringMode(isEnable: true, interval: 60)
{ state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.9 健康值预警

6.9.1 心率告警

- 方法

```
/// 心率报警
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnabled: 是否开启
///   - maxHeartRate: 心率告警上限 100 ~ 240
///   - minHeartRate: 心率告警下限 30 ~ 60
///   - completion: 设置结果
public static func setDeviceHeartRateAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    maxHeartRate: UInt8,
    minHeartRate: UInt8 ,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明

- 心率告警指的是设备检测到心率值只要高于上限或低于下限都会开启，注意参数的取值范围。

- 使用举例

```
YCProduct.setDeviceHeartRateAlarm(isEnable: true,
                                   maxHeartRate: 100,
                                   minHeartRate: 50) { state, response in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.9.2 温度报警

- 方法

```
/// 温度报警
/// - Parameters:
```

```

/// - peripheral: 已连接设备
/// - isEnabled: 是否开启
/// - highTemperatureIntegerValue: 温度上限整数值 36 ~ 100 摄氏度
/// - highTemperatureDecimalValue: 温度上限小数值 0 ~ 9 摄氏度
/// - lowTemperatureIntegerValue: 温度下限 -127 ~ 36 摄氏度
/// - lowTemperatureDecimalValue: 温度下限小数值 0 ~ 9 摄氏度
/// - completion: 设置结果
public static func setDeviceTemperatureAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    highTemperatureIntegerValue: UInt8,
    highTemperatureDecimalValue: UInt8,
    lowTemperatureIntegerValue: Int8,
    lowTemperatureDecimalValue: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

```

- 说明

- 和心率报警值一样有两个值，要将整数和小数部分分开，小数部分的取值是0~9，和其它方法一样，文档中的温度都单位是摄氏度。

- 使用举例

```

YCProduct.setDeviceTemperatureAlarm(isEnabled: true,
                                     highTemperatureIntegerValue: 37,
                                     highTemperatureDecimalValue: 3,
                                     lowTemperatureIntegerValue: 35,
                                     lowTemperatureDecimalValue: 5) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.9.3 血压预警

- 方法

```

/// 血压报警设置
/// - Parameters:
/// - peripheral: 已连接设备

```

```

/// - isEnabled: 是否开启
/// - maximumSystolicBloodPressure: 最大收缩压
/// - maximumDiastolicBloodPressure: 最大舒张压
/// - minimumSystolicBloodPressure: 最小收缩压
/// - minimumDiastolicBloodPressure: 最小舒张压
/// - completion: 设置结果
public static func setDeviceBloodPressureAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    maximumSystolicBloodPressure: UInt8,
    maximumDiastolicBloodPressure: UInt8,
    minimumSystolicBloodPressure: UInt8,
    minimumDiastolicBloodPressure: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> (()))?
)

```

- 说明
 - 当设备检测到血压不在设置范围内将进行报警，同时将检测值上报，获取检测值的内容，请参考第8章节。
- 使用举例

```

YCProduct.setDeviceBloodPressureAlarm(isEnabled: true,
                                       maximumSystolicBloodPressure: 250,
                                       maximumDiastolicBloodPressure: 140,
                                       minimumSystolicBloodPressure: 160,
                                       minimumDiastolicBloodPressure: 90) {

state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.9.4 血氧预警

- 方法

```

/// 设置血氧报警
/// - Parameters:
///   - peripheral: 连接外设
///   - isEnabled: 是否开启
///   - minimum: 最低血氧值
///   - completion: 结果
public static func setDeviceBloodOxygenAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    minimum: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明

- 血氧报警 只有一个下限值，当测量值低于设置的报警值时，设备将会报警。测量值高于设定值时，设备将不报警。
- 此功能是否有效，可以由 `isSupportBloodPressureAlarm` 标志是否为 `true` 来判断。

- 使用举例

```

// 血氧低于90报警
YCProduct.setDeviceBloodOxygenAlarm(
    isEnabled: true,
    minimum: 90
) { state, response in
    if state == .succeed, let datas = response as? [YCDeviceEventInfo] {
        print("success")
    } else {
        print("fail")
    }
}

```

6.9.5 呼吸率报警

- 方法

```

/// 设置呼吸率报警
/// - Parameters:
///   - peripheral: 连接外设
///   - isEnabled: 是否开启
///   - maximum: 报警值上限

```

```

///    - minimum: 报警值下限
///    - completion: 结果
public static func setDeviceRespirationRateAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    maximum: UInt8,
    minimum: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

```

▪ 说明

- 呼吸率报警有上限和下限两个值，如果测量值在范围内，设备不会报警，否则设备报警。
- 此功能是否有效，可以由 isSupportRespirationRateAlarm 标志是否为 true 来判断。

▪ 使用举例

```

// 设置呼吸率报警值
YCProduct.setDeviceRespirationRateAlarm(
    isEnabled: true,
    maximum: 30,
    minimum: 6
) { state, response in
    if state == .succeed, let datas = response as? [YCDeviceEventInfo] {
        print("success")
    } else {
        print("fail")
    }
}

```

6.10 勿扰设置

▪ 方法

```

///    勿扰模式设置
///    - Parameters:
///    - peripheral: 已连接设备
///    - isEable: 是否使能
///    - startHour: 开始时间 小时 0 ~ 23
///    - startMinute: 开始时间 分钟 0 ~ 59

```

```

/// - endHour: 结束时间 小时 0 ~ 23
/// - endMinute: 结束时间 分钟 0 ~ 59
/// - completion: 设置结果
public static func setDeviceNotDisturb(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEable: Bool,
    startHour: UInt8,
    startMinute: UInt8,
    endHour: UInt8,
    endMinute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 手环进入勿扰模式后，所有的提醒类功能都不工作。
- 使用举例

```

// 9:30 ~ 12:00
YCProduct.setDeviceNotDisturb(isEable: true,
                                startHour: 9,
                                startMinute: 30,
                                endHour: 12,
                                endMinute: 0) { state, response in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.11 恢复出厂设置

- 方法

```

/// 恢复出厂设置
/// - Parameters:
/// - peripheral: 已连接设备
/// - completion: 结果
public static func setDeviceReset(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 设备执行恢复出厂设置后，将清除所有的数据，且手环会断开连接。
- 使用举例

```
YCProduct.setDeviceReset { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.12 设置语言

- 方法

```
/// 设置语言
/// - Parameters:
///   - peripheral: 已连接设备
///   - language: 语言
///   - completion: 设置结果
public static func setDeviceLanguage(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    language: YCDeviceLanguageType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 语言
@objc public enum YCDeviceLanguageType: UInt8 {

    case english           // 英语
    case chineseSimplified // 简体中文
    case russian           // 俄语
    case german            // 德语
    case french            // 法语
    case japanese          // 日语
    case spanish           // 西班牙语
    case italian           // 意大利语
    case portuguese        // 葡萄牙语
    case korean            // 韩语
    case poland            // 波兰语
    case malay             // 马来语
    case chineseTradition  // 繁体中文
```



```

case thai                // 泰语

case vietnamese          // 越南语
case hungarian           // 匈牙利语
case arabic              // 阿拉伯语
case greek               // 希腊语
case malaysian           // 马来西亚语
case hebrew              // 希伯来语
case finnish             // 芬兰语
case czech               // 捷克语
case croatian            // 克罗地亚语

case persian             // 波斯语

case ukrainian           // 乌克兰语
case turkish             // 土耳其语

case danish              // 丹麦语
case swedish             // 瑞典语
case norwegian           // 挪威语
case romanian            // 罗马尼亚语
}

```

- 说明
 - 每个设备支持的语言是不相同的，对于不支持的语言可能会显示为英语。
- 使用举例

```

YCProduct.setDeviceLanguage(language: .persian) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
}

```

6.13 抬腕亮屏开关

- 方法

```

/// 抬手亮屏开关
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnabled: 是否使能
///   - completion: 执行回调
public static func setDeviceWristBrightScreen(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 无
- 使用举例

```

YCProduct.setDeviceWristBrightScreen(isEnabled: true) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.14 屏幕设置

6.14.1 屏幕亮度

- 方法

```

/// 屏幕亮度设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - level: 亮度等级
///   - completion: 设置结果
public static func setDeviceDisplayBrightness(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    level: YCDeviceDisplayBrightnessLevel,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 亮度等级
@objc public enum YCDeviceDisplayBrightnessLevel : UInt8 {

```

```

        case low
        case middle
        case high
        case automatic
        case lower
        case higher
    }

```

- 说明
 - 亮度等级前3个值是通用，后三个值是某些定制设备才支持。
- 使用举例

```

YCProduct.setDeviceDisplayBrightness(level: .middle) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.14.2 息屏时间

- 方法

```

/// 息屏时间设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - interval: 时间间隔 YCDeviceBreathScreenInterval
///   - completion: 设置结果
public static func setDeviceBreathScreen(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    interval: YCDeviceBreathScreenInterval,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 时间间隔
@objc public enum YCDeviceBreathScreenInterval: UInt8 {
    case five // 5s
    case ten // 10s
    case fifteen // 15s
    case thirty // 30s
}

```

- 说明
 - 注意时间间隔不是具体的数值，而是 `YCDeviceBreathScreenInterval`。
- 方法

```
YCProduct.setDeviceBreathScreen(interval: .fifteen) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.15 肤色设置

- 方法

```
/// 肤色设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - level: 肤色
///   - completion: 设置结果
public static func setDeviceSkinColor(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    level: YCDeviceSkinColorLevel,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

@objc public enum YCDeviceSkinColorLevel: UInt8 {
    case white          // 白色
    case whiteYellow    // 白间黄
    case yellow          // 黄色
    case brown          // 棕色
    case darkBrown      // 褐色
    case black          // 黑色
    case other          // 其它
}
```

- 说明
 - 肤色设置会影响到设备检测健康数据与ECG测试等，一般皮肤越黑，毛发越多的用户，取值越大。
- 使用举例

```
YCProduct.setDeviceSkinColor(level: .yellow) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.19 主题设置

- 方法

```
/// 主题设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - index: 主题索引
///   - completion: 设置结果
public static func setDeviceTheme(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    index: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明
 - 主题索引从0开始，到主题总数 - 1。
- 使用举例

```
YCProduct.setDeviceTheme(index: 0) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.20 设置睡眠提醒时间

- 方法

```
/// 睡眠提醒时间
/// - Parameters:
///   - peripheral: 已连接设备
///   - hour: 小时 0 ~ 23
///   - minute: 分钟: 0 ~ 59
///   - repeat: 重复 YCDeviceWeekRepeat
///   - completion: 设置结果
public static func setDeviceSleepReminder(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8, minute: UInt8, repeat: Set<YCDeviceWeekRepeat>,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明

- 设置成功后，当前时间进入提醒时间时，设备将震动显示睡眠提醒画面。

- 使用举例

```
YCProduct.setDeviceSleepReminder(hour: 22,
                                   minute: 30,
                                   repeat: [.monday, .thursday, .wednesday,
                                           .enable]) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.30 闹钟

此功能不包含杰理平台，杰理平台的闹钟可以在手表上直接设置。

6.30.1 查询闹钟

- 方法

```
/// 查询闹钟
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 闹钟信息
public static func queryDeviceAlarmInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 闹钟信息
@objc public class YCDeviceAlarmInfo : NSObject {

    /// 设备允许的最多闹钟数量
    public var limitCount: UInt8 { get }

    /// 闹钟类型
    public var alarmType: YCProductSDK.YCDeviceAlarmType { get }

    /// 闹钟中的小时 (24小时制)
    public var hour: UInt8 { get }

    /// 闹钟中的分钟
    public var minute: UInt8 { get }

    /// 重复
    public var `repeat`: Set<YCProductSDK.YCDeviceWeekRepeat> { get }

    /// 贪睡时间 min
    public var snoozeTime: UInt8 { get }
}

/// 闹钟类型
@objc public enum YCDeviceAlarmType : UInt8 {
    case wakeUp           // 起床
    case sleep            // 睡眠
    case exercise         // 锻炼
    case medicine         // 吃药
    case appointment     // 约会
    case party            // 聚会
    case meeting          // 会议
}
```

```

        case custom                                // 自定义
    }

```

- 说明
 - 执行查询闹钟后，所有的闹钟信息将以YCDeviceAlarmInfo的形式返回。
- 使用举例

```

YCProduct.queryDeviceAlarmInfo { state, response in
    if state == .succeed,
        let datas = response as? [YCDeviceAlarmInfo] {

            for item in datas {
                print(item.hour, item.minute)
            }
        }
    }
}

```

6.30.2 增加闹钟

- 方法

```

/// 添加闹钟
/// - Parameters:
///   - peripheral: 已连接设备
///   - alarmType: 闹钟类型
///   - hour: 小时 0 ~ 23
///   - minute: 分钟 0 ~ 59
///   - repeat: 重复时间
///   - snoozeTime: 贪睡时长 0~59分钟
///   - completion: 设置结果
public static func addDeviceAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

    alarmType: YCDeviceAlarmType,
    hour: UInt8,
    minute: UInt8,
    repeat: Set<YCDeviceWeekRepeat>,
    snoozeTime: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?)
)

```

- 说明

- 同一个时间的闹钟只允许一个，增加闹钟不能超过设备限制的数量，一般为10个。
- 使用举例

```
YCProduct.addDeviceAlarm(alarmType: .wakeUp,
                          hour: 6,
                          minute: 30,
                          repeat: [.enable, .sunday, .saturday],
                          snoozeTime: 0) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail \((String(describing: response) ?? ""))")
    }
}
```

6.30.3 修改闹钟

- 方法

```
/// 修改闹钟
/// - Parameters:
///   - peripheral: 已连接设备
///   - oldHour: 闹钟原来的小时
///   - oldMinute: 闹钟原来的分钟
///   - hour: 闹钟新的小时
///   - minute: 闹钟新的分钟
///   - alarmType: 闹钟类型
///   - repeat: 重复时间
///   - snoozeTime: 贪睡时长
///   - completion: 结果
public static func modifyDeviceAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    oldHour: UInt8,
    oldMinute: UInt8,
    hour: UInt8,
    minute: UInt8,
    alarmType: YCDeviceAlarmType,
    repeat: Set<YCDeviceWeekRepeat>,
    snoozeTime: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明

- 依据闹钟旧的时间就可以修改闹钟信息
- 使用举例

```
YCProduct.modifyDeviceAlarm(oldHour: 6,
                             oldMinute: 30,
                             hour: 11,
                             minute: 0,
                             alarmType: .meeting,
                             repeat: [.enable, .monday],
                             snoozeTime: 0) { state, response in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.30.4 删除闹钟

- 方法

```
/// 删除闹钟
/// - Parameters:
///   - peripheral: 已连接设备
///   - hour: 小时
///   - minute: 分钟
///   - completion: 结果
public static func deleteDeviceAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8,
    minute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明
 - 依据闹钟的时间就可删除指定的闹钟
- 使用举例

```

YCProduct.deleteDeviceAlarm(hour: 6, minute: 30) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

6.34 名片信息

- 方法

```

/// 发送名片到设备
/// - Parameters:
///   - peripheral: 连接设备
///   - businessCardType: 名片类型
///   - content: 内容
///   - completion: 结果
public static func sendBusinessCard(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    businessCardType: YCBusinessCardType,
    content: String,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 查询名片信息
/// - Parameters:
///   - peripheral: 已连接设备
///   - businessCardType: 名片类型
///   - completion: 结果
public static func queryBusinessCardContent(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    businessCardType: YCBusinessCardType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 名片类型

```

```

@objc public enum YCBusinessCardType: UInt8 {
    case wechat
    case qq
    case facebook
    case twitter
    case whatsapp
    case instagram

    /* ----- */
    case snCode
    case staticCode
    case dynamicCode
}

```

- 说明
 - 名片可以设置、可以查询。
 - 如果使用 snCode开始的参数，还会使用到状态变化的操作，参考8.6章节
- 使用举例

```

// 设置
YCProduct.sendBusinessCard(businessCardType: .wechat, content:
"https://u.wechat.com/xxxxxxx") { state, _ in

    if state == .succeed {
        print("succeed")
    } else {
        print("failed")
    }
}

// 查询
YCProduct.queryBusinessCardContent(businessCardType: .wechat) { state,
response in

    guard state == .succeed,
        let content = response as? String else {
        return
    }

    print(content)
}

```

7. 控制设备

7.1 找设备

- 方法

```
/// 找设备
/// - Parameters:
///   - peripheral: 已连接设备
///   - remindCount: 提醒次数 (1 ~ 10)
///   - remindInterval: 间隔秒数 (1 ~ 3)
///   - completion: 结果
public static func findDevice(_ peripheral: CBPeripheral? =
    YCProduct.shared.currentPeripheral,
                             remindCount: UInt8 = 5,
                             remindInterval: UInt8 = 1,
                             completion: ((_ state: YCProductState, _
response: Any?) -> ())?
)
```

- 说明
 - 调用 方法后，手环会生震动，虽然方法中提供了提醒参数设置，但建议使用SDK提供的默认值。
- 使用举例

```
YCProduct.findDevice { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.2 血压校准

- 方法

```

/// 血压校准
/// - Parameters:
///   - peripheral: 连接手环
///   - systolicBloodPressure: 收缩压
///   - diastolicBloodPressure: 舒张压
///   - completion: 结果
public static func deviceBloodPressureCalibration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    systolicBloodPressure: UInt8,
    diastolicBloodPressure: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明

- 血压校准是指校准光电血压，执行了血压校准后，不需要使用血压等级设置，即二者只用其中一个，且血压校准优先级高。

- 使用举例

```

YCProduct.deviceBloodPressureCalibration(
    systolicBloodPressure: 110,
    diastolicBloodPressure: 72) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

7.3 温度校准

- 方法

```

/// 温度校准
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 校准结果
public static func deviceTemperatureCalibration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 温度校准是用于生产设备时校准温度传感器，使温度测量更加准确，开发应用程序一般不需要使用此方法。
- 使用举例

```
YCProduct.deviceTemperatureCalibration { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.6 天气数据

- 方法

```
/// 发送天气
/// - Parameters:
///   - peripheral: 已连接设备
///   - isTomorrow: 今天天气还是明天天气
///   - lowestTemperature: 最低温度 摄氏度
///   - highestTemperature: 最高温度 摄氏度
///   - realTimeTemperature: 当前天气温度 摄氏度
///   - weatherType: 天气类型 YCWeatherCodeType
///   - windDirection: 风向
///   - windPower: 风力
///   - location: 城市
///   - moonType: 月相
///   - completion: 发送结果
public static func sendWeatherData(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isTomorrow: Bool = false,
    lowestTemperature: Int8,
    highestTemperature: Int8,
    realTimeTemperature: Int8,
    weatherType: YCWeatherCodeType,
    windDirection: String?,
    windPower: String?,
    location: String?,
    moonType: YCWeatherMoonType?,
    completion: (() -> Void)? = nil) {
```

```

        completion: ((_ state: YCProductState, _ response: Any?) -> (()))?
    )

    /// 天气编码类型
    @objc public enum YCWeatherCodeType: UInt8 {

        case unknow
        case sunny
        case cloudy
        case wind
        case rain
        case snow
        case foggy

        // ===== 个别设备定制类型
        case sunnyCustom
        case cloudyCustom
        case thunderShower
        case lightRain
        case moderateRain
        case heavyRain
        case rainSnow
        case lightSnow
        case moderateSnow
        case heavySnow
        case floatingDust
        case fog
        case haze
        case windCustom
    }

    /// 月相信息
    @objc public enum YCWeatherMoonType: UInt8 {

        case newMoon
        case waningMoon
        case theLastQuarterMoon
        case lowerConvexMoon
        case fullMoon
        case upperConvexMoon
        case firstQuarterMoon
        case crescentMoon
        case unknown
    }

```


- 说明
 - isTomorrow 用于确定是发送今日天气还是明天的天气，对于手环是否支持明天天气设置，可以依据功能属性或返回值来判断。
 - 天气中的温度都是摄氏温度
 - 天气类型 YCWeatherCodeType 只有前6个值是通用的，后续列举的取值是某些特殊定制的设备才能使用。
 - 剩余可选参数都是定制设备才能使用，其它设备一律使用 nil。
- 使用举例

```
YCProduct.sendWeatherData(lowestTemperature: -20,
                           highestTemperature: 36,
                           realTimeTemperature: 25,
                           weatherType: .sunny,
                           windDirection: nil,
                           windPower: nil,
                           location: nil,
                           moonType: nil) { state, _ in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.7 关机 复位 重启

- 方法

```
/// 系统操作
/// - Parameters:
///   - peripheral: 已连接设备
///   - mode: 模式
///   - completion: 结果
public static func deviceSystemOperator(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    mode: YCDeviceSystemOperator,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

/// 设备模式
@objc public enum YCDeviceSystemOperator: UInt8 {
    case shutDown = 1 // 关机
    case transportation // 运输模式
```

```

        case resetRestart                                // 复位
    }

```

- 说明
 - 使用不同的模式，设备会进入不同的状态，注意如果设置为运输模式，则必须使用充电才能退出。
- 使用举例

```

YCProduct.deviceSystemOperator(mode: .shutDown) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

7.8 获取实时数据

获取设备的实时数据比较特殊，开启与接收是在两个方法，而为了区分，返回数据类型与控制类型设置不完全一样，Demo会列出获取数据的过程与步骤，对于没有列举的内容，可能会在其它使用场景出现，如果整篇文档都没有出现则可能不支持或不需要此功能。

7.8.1 开启获取实时数据

- 方法

```

/// 实时数据上传
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnabled: 是否开启或关闭
///   - dataType: 数据类型
///   - interval: 间隔 1 ~ 240秒，建议使用2秒默认值
///   - completion: 设置结果
public static func realTimeDataUpload(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

    isEnabled: Bool,
    dataType: YCRealTimeDataType,
    interval: UInt8 = 2,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

```

```

/// 数据类型
@objc public enum YCRealTimeDataType: UInt8 {
    case step
}

```

- 说明
 - 这个方法使用场景较少，主要是SDK内部使用，个别情况需要使用到方法。
 - 使用此方法开启后，要主动接收设备的数据，参考7.8.2。
 - 整个过程使用结束后，建议对方法执行关毕操作，避免出现一些莫名奇妙的问题。
- 使用举例

```

YCProduct.realTimeDataUplod(isEnable: true,
                             dataType: YCRealTimeDataType.step) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

7.8.2 接收设备的上报的实时数据

- 方法

```

/// 接收实时数据的通知
public static let receivedRealTimeNotification: Notification.Name

/// 接收到实时数据类型
@objc public enum YCReceivedRealTimeDataType: UInt8 {
    case step
}

/// 接收到的实时数据的响应
@objcMembers public class YCReceivedDeviceInfo: NSObject {

    /// 收到的响应设备
    public var device: CBPeripheral? { get }

    /// 结果
    public var data: Any? { get }
}

```

```

/// 实时步数
@objcMembers public class YCReceivedRealTimeStepInfo: NSObject {

    /// 步数 (单位:步)
    public var step: UInt16 { get }

    /// 距离 (单位:米)
    public var distance: UInt16 { get }

    /// 卡路里 (单位:千卡)
    public var calories: UInt16 { get }
}

```

■ 说明

- receivedRealTimeNotification这个通知需要应用程序监听，依据不同的类型来解析，这里会给出所有的解析过程，在文档的其它地方可能会用到其中一部分。
- 如果没有对应的数据回放，说明没有对应的数据或当前场景无法获取。
- 注意：如果使用完这个方法，要主动执行一次关闭操作，否则可能会造成一些莫名其妙的问题。
- 所有的类型数据都会以YCReceivedDeviceReportInfo类型返回，要以YCReceivedRealTimeDataType类型作为key来取出。

■ 使用举例

```

NotificationCenter.default.addObserver(
    self,
    selector: #selector(receiveRealTimeData(_:)),
    name: YCProduct.receivedRealTimeNotification,
    object: nil
)

@objc private func receiveRealTimeData(_ notification: Notification) {

    guard let info = notification.userInfo else {
        return
    }

    if let response = info[YCReceivedRealTimeDataType.step.string] as?
YCReceivedDeviceReportInfo,
        let device = response.device,
        let sportInfo = response.data as? YCReceivedRealTimeStepInfo {

```

```

        print(device.name ?? "",
              sportInfo.step,
              sportInfo.calories,
              sportInfo.distance
        )
    }
}

```

7.10 ECG检测

ECG检测包含启与停止ECG，获取ECG的结果，关于绘制ECG的波形，请参考Demo的案例演示，文档会将相关的方法进行举例说明。ECG检测的开启和关闭都由App来完成，建议测量时间为60 ~ 90秒。测试过程会获取到测量的数据。同样，设备本身也能启动ECG测量，App可以获取到相关的信息。

7.10.1 获取电极位置

- 方法

```

/// 获取心电电位位置
public static func queryDeviceElectrodePosition(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// 心电电极位置
@objc public enum YCDeviceElectrodePosition : UInt8 {
    case right // 右侧
    case bottom // 屏正下方
    case bothSides // 左侧两侧
    case fullEncirclement // 全包围
}

```

- 说明
 - 获取到设备的电极位置，可以提示用户在测量ECG时将手指放置在哪个地方。
- 使用举例

```
YCProduct.queryDeviceElectrodePosition { state, response in
    if state == .succeed,
    let info = response as? YCDeviceElectrodePosition {
        print(info.rawValue)
    }
}
```

7.10.2 设置穿戴位置

- 方法

```
/// 左右手佩戴设置
/// - Parameters:
///   - peripheral: 已连接设备
///   - wearingPosition: 佩戴位置
///   - completion: 设置结果
public static func setDeviceWearingPosition(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    wearingPosition: YCProductSDK.YCDeviceWearingPositionType = .left,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// 穿戴位置
@objc public enum YCDeviceWearingPositionType : UInt8 {
    case left        // 左手
    case right       // 右手
}
```

- 说明
 - 设备的佩戴位置与设置位置不匹配时，产生的波形是相反的。
- 使用举例

```
// 左手
YCProduct.setDeviceWearingPosition(wearingPosition: .left) { state,
response in
    if state == .succeed {

    }
}
```

7.10.3 开启与结束ECG测量

- 方法

```
/// 开始ECG测量
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 是否开启成功
public static func startECGMeasurement(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// 关闭ECG测量
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 是否关闭成功
public static func stopECGMeasurement(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- 说明

- 启动测量ECG后，需要将另一个手的手指放到与电极位置的地方，进行接触，测量开始后，设备会将上报测量的数据。

- 使用举例

```
// 开始测量
YCProduct.startECGMeasurement { state, _ in
    if state == .succeed {
    }
}

// 结束测量
YCProduct.stopECGMeasurement { state, _ in
    if state == .succeed {
    }
}
```

7.10.4 接收测量过程数据

- 方法

```
/// 接收实时数据的通知
public static let receivedRealTimeNotification: Notification.Name

/// 实时血压数据
@objcMembers public class YCReceivedRealTimeBloodPressureInfo : NSObject {

    /// 心率值
    public var heartRate: Int { get }

    /// 收缩压
    public var systolicBloodPressure: Int { get }

    /// 舒张压
    public var diastolicBloodPressure: Int { get }

    /// 血氧值
    public var bloodOxygen: Int { get }

    /// HRV
    public var hrv: Int { get }

    /// 温度
    public var temperature: Double { get }
}
```

- 说明

- 设备在启动测量ECG的过程中，会将心率，血压和ECG数据进行上报，部分定制设备会返回PPG数据。
- 通过监听同一个通知，依据不同的数据类型可以获得对应的数据。具体的数据处理，参考Demo。

- 使用举例

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(receiveRealTimeData(_:)),
    name: YCProduct.receivedRealTimeNotification,
    object: nil
)
```



```

@objc private func receiveRealTimeData(_ notification: Notification) {

    guard let info = notification.userInfo else {
        return
    }

    // 血压数据
    if let healthData =
        (info[YCReceivedRealTimeDataType.bloodPressure.toString] as?
        YCReceivedDeviceReportInfo)?.data as? YCReceivedRealTimeBloodPressureInfo
    {
        heartRate = healthData.heartRate
        systolicBloodPressure =
            healthData.systolicBloodPressure
        diastolicBloodPressure =
            healthData.diastolicBloodPressure
        if healthData.hrv > 0 {

        }
    }

    // ECG数据
    if let ecgData = (info[YCReceivedRealTimeDataType.ecg.toString] as?
        YCReceivedDeviceReportInfo)?.data as? [Int32] {
        print(ecgData)
    }

    // ppg数据
    if let ppgData = (info[YCReceivedRealTimeDataType.ppg.toString] as?
        YCReceivedDeviceReportInfo)?.data as? [Int32] {
        print(ppgData)
    }
}

```

7.10.5 获取ECG结果

SDK对于ECG的测量结果, 及测量过程计算到的参数, 提供一个YCECGManager工具类来处理。

7.10.5.1 初始化

- 方法

```

/// 全局对象
public static let shared: YCProductSDK.YCECGManager

/// 算法计算过程回调
public func setupManagerInfo(
    rr:((_ rr: Float, _ heartRate: Int) -> ()))?,
    hrv: ((_ hrv: Int) -> ())?
)

```

- 说明
 - 数据处理过程中会产生一些数据，会在 setupManagerInfo 方法的回调中体现。
- 使用举例

```

let ecgManager = YCECGManager()

ecgManager.setupManagerInfo { rr, heartRate in
    // 检查到RR间隔 计算出心率
    print("=== 播放音效")
} hrv: { [weak self] hrv in
    // HRV
}

```

7.10.5.2 接收ECG数据

- 方法

```

/// 处理ECG数据
public func processECGData(_ data: Int) -> Float

```

- 说明
 - 要将获得的ECG数据，逐个传入到YCECGManager工具类中。
- 使用举例

```

for data in datas {
    var ecgValue: Float = 0
    ecgValue = ecgManager.processECGData(Int(data))
    // ... 其它处理
}

```

7.10.5.3 获取ECG结果

- 方法

```
/// 获取ECG结果
/// - Parameters:
///   - peripheral: 已连接设备
///   - deviceHeartRate: 设备测量的心率
///   - deviceHRV: 设备测量的hrv
/// - completion: 测量结果
public func getECGMeasurementResult(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    deviceHeartRate: Int?,
    deviceHRV: Int?,
    completion: @escaping (_ result: YCECGMeasurementResult) -> ()
)

/// ECG测量结果
@objcMembers public class YCECGMeasurementResult : NSObject {

    /// 心率
    public var hearRate: Int { get }

    /// ECG结果
    public var ecgMeasurementType: YCProductSDK.YCECGResultType { get }

    /// hrv值
    public var hrv: Int { get }
}

/// ECG类型
@objc public enum YCECGResultType : UInt {
    case failed // 测量失败
    case atrialFibrillation // 房颤
    case earlyHeartbeat // 房性早搏
    case supraventricularHeartbeat // 室性早搏
    case atrialBradycardia // 心率过缓
    case atrialTachycardia // 心率过快
    case atrialArrhythmia // 窦性心率不齐
    case normal // 正常心电图
}
```

- 说明

- 获取结果中的的方法中有心率和HRV有两个参数，如果使用nil， 则最后的心率和HRV值将会使用算法计算的值，如果传入指定的值，最后的结果会使用传入的值。
- 建议使用设备测量的值，如果没有，使用算法的值。
- 关于ECG测量结果的表述，表格中已给出参考文案。

ECG结果 (YCECGResultType)	文字表述
failed	抱歉！本次测量信号欠佳，可能是皮肤干燥造成，请清洁或者润湿测试部位的皮肤后重新测试，测试过程中保持安静。
atrialFibrillation	QRS波形正常，正常P波消失，出现f波，R-R间距不规则。
earlyHeartbeat	QRS波形正常，变异P波提前出现，P-R > 0.12秒，代偿间歇不完全。
supraventricularHeartbeat	QRS-T波形宽大变形，QRS波形前无相关P波，QRS时限 > 0.12秒，T波方向与主波相反，完全性代偿间歇。
atrialBradycardia	QRS波形正常，R-R间距偏长。
atrialTachycardia	QRS波形正常，R-R间距偏短。
atrialArrhythmi	QRS波形正常，R-R间距变化偏大
normal	QRS波形形态时限振幅正常，P-R间期正常，ST-T无改变，Q-T间期正常。

- 使用举例

```
ecgManager.getECGMeasurementResult(
    deviceHeartRate: heartRate > 0 ? heartRate : nil,
    devieHRV: hrvValue > 0 ? hrvValue : nil) { result in
    print(result.hearRate,
        result.hrv,
        result.ecgMeasurementType == .normal
    )
}
```

7.10.5.4 获取身体等情绪指数 (保留)

- 方法

```
/// 获取身体指数
public func getPhysicalIndexParameters() -> YCBodyIndexResult

/// 身体结果参数
```

```
@objcMembers public class YCBodyIndexResult: NSObject {

    /// 是否可用
    public var isAvailable: Bool = false

    /// 负荷指数
    public var heavyLoad: Float = 0

    /// 压力指数
    public var pressure: Float = 0

    /// HRV指数
    public var hrvNorm: Float = 0

    /// 身体指数
    public var body: Float = 0
}
```

- 说明
 - 此方法可以能过返回值中的属性 isAvailable 来判断是否可用。
- 使用举例

```
let bodyInfo = ecgManager.getPhysicalIndexParameters()
if bodyInfo.isAvailable {
    print("heavyLoad = \(bodyInfo.heavyLoad), pressure = \(bodyInfo.pressure), hrvNorm = \(bodyInfo.hrvNorm), body = \(bodyInfo.body)")
}
```

7.10.6 关于绘制ECG波形

1. ECG测量过程中，如果有绘制图形的需要，还是先阅读Demo中的代码，同时需要了解iOS中的基本绘图知识。
2. Demo中给出的波形是正常标准化的画法，如果是其它类型，则需要在此基础上进行放大或缩小。

7.10.7 获取设备启动测量的ECG和PPG数据

1. 如果是设备启动的ECG测量，只能获取到测量的ECG或PPG数据，其它数据都没有。
2. 获取数据的相关操作，请参照第10章历史采集数据章节。

7.12 运动

```
/// 运动类型定义
@objc public enum YCDeviceSportType: UInt8 {

    case none                // 保留
    case run                  // 跑步（户外）
    case swimming             // 游泳
    case riding               // 户外骑行
    case fitness              // 健身

    case ropeskipping         // 跳绳
    case playball             // 篮球(打球)
    case walk                 // 健走
    case badminton            // 羽毛球

    case football             // 足球
    case mountaineering       // 登山
    case pingPang             // 乒乓球

    case indoorRunning        // 室内跑步
    case outdoorRunning       // 户外跑步
    case outdoorWalking       // 户外步行
    case indoorWalking        // 室内步行

    case indoorRiding         // 室内骑行
    case stepper              // 踏步机
    case rowingMachine         // 划船机
    case realTimeMonitoring   // 实时监控
    case situps               // 仰卧起坐
    case jumping              // 跳跃运动
    case weightTraining        // 重量训练
    case yoga                 // 瑜伽
}

/// 运动状态
@objc public enum YCDeviceSportState: UInt8 {
    case stop
    case start
}
```

7.3.1 运动启动与关闭

- 方法

```
/// 控制设备进入运动模式
/// - Parameters:
///   - peripheral: 已连接设备
///   - state: 运动状态
///   - sportType: 运动类型
///   - completion: 结果
public static func controlSport(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    state: YCDeviceSportState,
    sportType: YCDeviceSportType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明

- 目前，所有的设备只支持开启与结束。

- 使用举例

```
/// 开始跑步
YCProduct.controlSport(state: .start, sportType: .run) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

/// 结束跑步
YCProduct.controlSport(state: .stop, sportType: .run) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.3.2 运动数据的接收

由于设备运动数据上报方式进行了优化，所以接收数据的方式也随之发生了变化，后续的产品逐步迁移到新方式，新方式需要接收两组数据，旧方式只有一种。

7.3.2.1 判断设备传输数据的运动方式

- 方法

```
// 参考 5.1 节
public var isSupportSyncRealSportData: Bool = false
```

- 说明
 - 通过设备的属性来判断是新方式还是旧方式。
- 使用举例

```
if peripheral?.supportItems.isSupportSyncRealSportData {
    // 新方式
} else {
    // 旧方式
}
```

7.3.2.2 旧方式

- 方法

```
/// 接收实时数据的通知
public static let receivedRealTimeNotification: Notification.Name
```

- 说明
 - 设备会返回心率，步数，距离，卡路里，不会返回时间。
 - 注意，步数，距离，卡路里是返回累加的结果，所以每次获取这三个值都应该减去进入运动后第一次获取到的初始值。
 - 这种方式无法获取设备是否退出运动模式。
- 使用举例

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(receiveRealTimeData(_:)),
    name: YCProduct.receivedRealTimeNotification,
    object: nil
```



```

)

@objc private func receiveRealTimeData(_ notification: Notification) {
    guard let info = notification.userInfo else {
        return
    }
    if let response = info[YCReceivedRealTimeDataType.step.string] as?
YCReceivedDeviceInfo,
        let device = response.device,
        let sportInfo = response.data as? YCReceivedRealTimeStepInfo {
        print(device.name ?? "",
            sportInfo.step,
            sportInfo.calories,
            sportInfo.distance
        )
    }
    else if let response =
info[YCReceivedRealTimeDataType.heartRate.string] as?
YCReceivedDeviceInfo,
        let device = response.device,
        let heartRate = response.data as? UInt8 {
        print(device.name ?? "",
            heartRate)
    }
}
}

```

7.3.2.3 新方式

- 方法

```

/// 接收实时数据的通知
public static let receivedRealTimeNotification: Notification.Name

/// 运动状态变化通知
public static let deviceControlNotification: Notification.Name

/// 运动状态信息
@objcMembers public class YCDeviceControlSportModeControlInfo: NSObject {
    public var state: YCDeviceSportState {get}
    public var sportType: YCDeviceSportType {get}
}

```

- 说明

- 新的方式除了可以获取设备的运动数据，也可能获取运动是否退出等状态。
- 使用举例

```
NotificationCenter.default.addObserver(  
    self,  
    selector: #selector(deviceDataStateChanged(_:)),  
    name: YCProduct.deviceControlNotification,  
    object: nil  
)
```

```
NotificationCenter.default.addObserver(  
    self,  
    selector: #selector(receiveRealTimeData(_:)),  
    name: YCProduct.receivedRealTimeNotification,  
    object: nil  
)
```

```
@objc private func receiveRealTimeData(_ notification: Notification) {  
  
    guard let info = notification.userInfo else {  
        return  
    }  
  
    if let response =  
info[YCReceivedRealTimeDataType.realTimeMonitoringMode.string] as?  
YCReceivedDeviceReportInfo,  
    let device = response.device,  
    let data = response.data as? YCReceivedMonitoringModeInfo {  
        print(device.name ?? "",  
            data.startTimeStamp,  
            data.modeStep,  
            data.modeCalories,  
            data.modeCalories  
        )  
    }  
}
```

```
@objc private func deviceDataStateChanged(_ ntf: Notification) {  
  
    guard let info = ntf.userInfo else {  
        return  
    }  
}
```

```

        if let response = info[YCDeviceControlType.sportModeControl.string] as?
        YCReceivedDeviceReportInfo,
            let device = response.device,
            let data = response.data as? YCDeviceControlSportModeControlInfo {
                print(device.name ?? "",
                    data.state,
                    data.sportType
                )
            }
    }
}

```

7.3.3 运动历史数据

- 绝大部分设备，由启动运行后，都不会记录运动的相关数据，需要App自己处理。
- 对于某些定制设备，会将运动数据进行保存，如果要获取这部分信息，请参考4.3.12 的内容。

7.13 拍照

启动拍照有两种方式，一种是设备启动进入拍照模式，一种是App启动进入拍照模式。进入拍照模式后，执行拍照动作也有两种方式，一种是点击App进行拍照，一种是点击设备进行拍照。真正的拍照都是在手机上完成，如果是设备点击执行拍照，则拍照完成后需要回复设备，拍照是否成功。

7.13.1 App开启与关闭拍照模式

- 方法

```

/// 手机控制进入与退出拍照模式
/// - Parameters:
///   - peripheral: 已连接设备
///   - isEnabled: 启动或关闭拍照模式
///   - completion: 结果
public static func takephotoByPhone(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

```

- 说明
 - 此方法是由手机来控制设备进入或退出拍照模式，所有的操作都是在手机端操作。

7.13.2 设备启动退出拍照

- 方法

```
/// 设备控制通知
public static let deviceControlNotification: Notification.Name

/// 拍照模式状态
@objc public enum YCDeviceControlPhotoState: UInt8 {
    case exit          // 退出拍照
    case enter         // 进入拍照
    case photo         // 拍照
}
```

- 说明

- 当接收到状态为拍照时，拍照依然是在手机上操作，设备只负责上报当前的状态。

- 使用举例

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(deviceDataStateChanged(_:)),
    name: YCProduct.deviceControlNotification,
    object: nil
)

@objc private func deviceDataStateChanged(_ ntf: Notification) {

    guard let info = ntf.userInfo else {
        return
    }

    if let response = info[YCDeviceControlType.photo.string] as?
YCReceivedDeviceReportInfo,
        let device = response.device,
        let state = response.data as? YCDeviceControlPhotoState {
        print(device.name ?? "",
              state
        )
    }
}
```

7.13.3 交叉操作

拍照的交互逻辑一旦出现交叉，比如手机启动，设备退出，或者设备启动，手机退出。要注意状态的变化，再调用对应的接口就可以了。

7.22 通讯录

通讯录功能，仅仅是将用户名称和号码，发送给设备进和保存，设备所能存储的最大数量为30个。传输通讯录的整个过程中，开启和退出同步只需要执行一次，而发送通讯数据，则需要重复执行，因为执行一次只能发送一条记录。

7.22.1 进入同步通讯录

- 方法

```
/// 开启通讯录同步
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 结果
public static func startSendAddressBook(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- 说明
 - 设备只有设置开始同步后，才能进入真正的同步数据。只需要执行一次。
- 使用举例

```
YCProduct.startSendAddressBook { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.22.2 发送通讯录数据

- 方法

```
/// 同步通讯录详情数据
/// - Parameters:
///   - peripheral: 已连接设备
///   - phone: 电话号码，不超过20个字符
///   - name: 用户名，不超过8个中文
///   - completion: 结果
public static func sendAddressBook(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    phone: String,
    name: String,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明

- 发送具体的通讯录信息到设备中，注意取传值长度。

- 使用举例

```
YCProduct.sendAddressBook(phone: "13800138000", name: "jack") { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.22.3 退出同步通讯录

- 方法

```
/// 退出通讯录同步
/// - Parameters:
///   - peripheral: 已连接设备
///   - completion: 结果
public static func stopSendAddressBook(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)
```

- 说明
 - 结束通讯录信息同步，只需要执行一次。
- 使用举例

```
YCProduct.stopSendAddressBook { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.22.4 杰理平台通讯录

杰理平台的通讯录的信息发送是使用的平台本身的API，是独立的API。

- 方法

```
/// 发送联系人信息到设备
/// - Parameters:
///   - datas: 联系人信息列表
///   - completion: 发送状态 isSuccess 成功还是失败, progress 0 ~ 1
public static func syncJLContactInfoToDevice(
    _ datas: [YCDeviceContactItem],
    completion: @escaping (_ isSuccess: Bool, _ progress: Float) -> ()
)

/// 查询设备联系人信息
/// - Parameter completion: 设备中存在的通讯录
public static func queryJLDeviceContactData(
    _ completion: @escaping (_ datas: [YCDeviceContactItem]) -> ()
)

/// 通讯录信息
@objcMembers open class YCDeviceContactItem: NSObject {

    /// 姓名
    open var name: String

    /// 电话
    open var phone: String
```

```

    /// 是否存在于设备中
    open var isExistDevice: Bool

    public init(name: String = "",
                phone: String = "",
                isExist: Bool = false)
}

```

- 说明
 - 杰理平台的通讯录支持查询与同步，通讯录信息是 `YCDeviceContactItem`，名称和电话都必须小于20个字节。
- 使用举例

```

/// 查询设备中手通讯信息
YCPProduct.queryJLDeviceContactData { deviceItems in
    for item in deviceItems {
        print("\(item.name) - \(item.phone)")
    }
}

/// 设置通讯录
YCPProduct.syncJLContactInfoToDevice([
    YCDeviceContactItem(name: "张三", phone: "18812345678"),
    YCDeviceContactItem(name: "李四", phone: "13685369726"),
]) { [weak self] isSuccess, progress in
    if isSuccess {
        print("success")
    } else {
        print("failed")
    }
}

```

7.23 血糖标定

- 方法

```

/// 血糖校准

```



```

/// - Parameters:
///   - peripheral: 连接设备
///   - bloodGlucoseInteger: 血糖整数
///   - bloodGlucoseDecimal: 血糖小数
///   - mode: 校准模式
///   - completion: 回调
public static func bloodGlucoseCalibration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    bloodGlucoseInteger: Int8,
    bloodGlucoseDecimal: Int8,
    mode: YCBloodGlucoseCalibrationaMode = .fasting,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// 血糖校准模式
@objc public enum YCBloodGlucoseCalibrationaMode: UInt8 {
    case fasting = 0 // 空腹 (餐前)
    case afterBreakfast // 早餐后 (餐后)
}

```

- 说明
 - 血糖检定是用于校准血糖测量的数据，执行此方法后，测试血糖结果会更加准确。
- 使用举例

```

// 空腹 4.6
YCProduct.bloodGlucoseCalibration(
    bloodGlucoseInteger: 4,
    bloodGlucoseDecimal: 6) { state, response in

    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

```

7.26 尿酸标定

- 方法

```
/// 尿酸校准
/// - Parameters:
///   - peripheral: 连接设备
///   - value: 尿酸值 (umol/L)
///   - completion: 校准结果
public static func uricAcidCalibration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    value: UInt16,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

```

- 说明
 - 用于校准尿酸测量，注意单位是 umol/L，如果是其它单位需要自己转换。校准范围是 90~1000 umol/L
- 使用举例

```
YCProduct.uricAcidCalibration(value: 300 ) { [weak self] state, response in

    if state == .succeed {

    } else {

    }

}
```

7.27 血脂标定

- 方法

```

/// 血脂校准
/// - Parameters:
///   - peripheral: 连接设备
///   - cholesterol: 胆固醇 (mmol/L) 1.0 ~ 9.5
///   - completion: 设置
public static func bloodFatCalibration(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    cholesterol: Double,
    completion: ((_ state: YCProductState, _ response: Any?) -> ()))?
)

```

- 说明
 - 用于校准血脂测量，注意单位是 mmol/L，如果是其它单位需要自己转换。校准范围是1.0~9.5 mmol/L
 - 调用接口后，需要等待手表测量结束。
- 使用举例

```

YCProduct.bloodFatCalibration(cholesterol: 5.1 ) { state, response in

    if state == .succeed {

        } else {

        }

    }

}

```

8. 接收设备响应

这部分指的是用户操作了设备或者设备监测到了某种信息，同时设备会将操作信息上报。SDK将统一监听设备的响应，依据不同的类型进行处理。SDK会以通知的形式发送设备的操作状态。应用程序需要进行监听并依据不同的类型进行解析。此外，有些内容在其它方已经列举，这里将不再出现。如果没有接收到对应的数据，可能是设备不支持此功能。

```

// 响应类型
@objc public enum YCDeviceControlType: UInt8 {

    case findPhone                // 找手机
    case photo                    // 拍照
    case stopRealTimeECGMeasurement // 停止ECG测量
    case sportModeControl         // 切换运动模式
    case switchWatchFace         // 切换表盘
}

/// 设备控制通知
public static let deviceControlNotification: Notification.Name

```

由于这部分比较统一，案例演示都写在一个演示中。

```

NotificationCenter.default.addObserver(
    self,
    selector: #selector(deviceDataStateChanged(_:)),
    name: YCProduct.deviceControlNotification,
    object: nil
)

@objc private func deviceDataStateChanged(_ ntf: Notification) {
    guard let info = ntf.userInfo else {
        return
    }

    // 找手机
    if let response = info[YCDeviceControlType.findPhone.toString] as?
    YCReceivedDeviceReportInfo,
        let device = response.device,
        let state = response.data as? YCDeviceControlState {
        print(device.name ?? "",
            state == .stop
        )
    }
}

```

8.1 找手机

设备开始找手机或停止找手机，都会上报状态。

```
// 找手机的状态
@objc public enum YCDeviceControlState: UInt8 {
    case stop
    case start
}
```

9. 表盘下载

表盘下载包含查询设备中的表盘信息，App切换表盘，删除表盘，设备操作表盘，App自定义表盘。

9.1 查询设备表盘信息

- 方法

```
/// 查询表盘信息
/// - Parameters:
///   - peripheral: 连接的设备
///   - completion: 查询结果
public static func queryWatchFaceInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// 表盘的断点信息
@objcMembers public class YCWatchFaceBreakCountInfo : NSObject {

    /// 表盘数据
    public var dials: [YCProductSDK.YCWatchFaceInfo] { get }

    /// 支持的最大数量
    public var limitCount: Int { get }
```

```

    /// 本地已存储的数量
    public var localCount: Int { get }
}

/// 表盘信息
@objcMembers public class YCWatchFaceInfo : NSObject {

    /// 表盘id
    public var dialID: UInt32

    /// 表盘断点值
    public var blockCount: UInt16

    /// 是否支持删除
    public var isSupportDelete: Bool { get }

    /// 表盘版本
    public var version: UInt16 { get }

    /// 是否为自定义表盘
    public var isCustomDial: Bool { get }

    /// 是否为当前显示表盘
    public var isShowing: Bool { get }
}

```

- 说明
 - 如果查询成功，返回的结果是 [YCWatchFaceBreakCountInfo] 类型
- 使用举例

```

YCProduct.queryWatchFaceInfo { state, response in
    if state == YCProductState.succeed,
    let info = response as? YCWatchFaceBreakCountInfo {
        if info.localCount > 0 {
            for item in info.dials {
                print(item.dialID)
            }
        }
    }
}

```

9.2 App删除表盘

- 方法

```
/// 删除表盘
/// - Parameters:
///   - peripheral: 已连接设备
///   - dialID: 表盘ID
///   - completion: 删除结果
public static func deleteWatchFace(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dialID: UInt32,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- 说明
 - 只要指定删除的表盘ID就可以删除表盘。
- 使用举例

```
let dialID: UInt32 = 2147483539
YCProduct.deleteWatchFace(dialID: dialID) { state, _ in
    if state == .succeed {
        print("delete success")
    }
}
```

9.3 App切换表盘

- 方法

```
/// 切换表盘
/// - Parameters:
///   - peripheral: 已连接设备
///   - dialID: 表盘ID
///   - completion: 结果
public static func changeWatchFace(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dialID: UInt32,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- 说明
 - 切换表盘与删除表盘的参数都是相同的
- 使用举例

```
let dialID: UInt32 = 2147483539
YCProduct.changeWatchFace(dialID: dialID) { state, _ in
    if state == .succeed {
        print("change success")
    }
}
```

9.4 设备切换或删除表盘

- 方法

```
/// 设备控制通知
public static let deviceControlNotification: Notification.Name
```

- 说明
 - 设备删除或切换了表盘，会主动上报最终显示的表盘ID, 可以通过查询到的表盘信息进行匹配，找到完整的表盘信息。
 - 设备上报的信息可以通过类型来取出
- 使用举例

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(watchFaceChanged(_:)),
    name: YCProduct.deviceControlNotification,
    object: nil
)

@objc private func watchFaceChanged(_ ntf: Notification) {
    guard let info = ntf.userInfo,
          let dialID = ((info[YCDeviceControlType.switchWatchFace.string])
as? YCReceivedDeviceInfo)?.data as? UInt32 else {
        return
    }
    print("dialID: \(dialID)")
}
```


9.5 下载表盘

- 方法

```
/// 下载表盘
/// - Parameters:
///   - peripheral: 连接的设备
///   - isEnabled: 开启或关闭
///   - data: 表盘数据
///   - dialID: 表盘id
///   - blockCount: 表盘断点
///   - dialVersion: 表盘版本
///   - completion: 下载进度
public static func downloadWatchFace(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnabled: Bool,
    data: NSData,
    dialID: UInt32,
    blockCount: UInt16,
    dialVersion: UInt16,
    completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// 下载进度信息
@objcMembers public class YCDownloadProgressInfo : NSObject {

    /// 进度 (0 ~ 1.0)
    public var progress: Float { get }

    /// 已下载数据大小
    public var downloaded: Int

    /// 下载总数据的大小
    public var total: Int { get }
}
```

- 说明

- 下载表盘中的参数，会通过返回值来表示下载是否正常，下载进度。

- 使用举例

```
guard let path = Bundle.main.path(forResource: "customE80.bin", ofType:
nil),
    let dialData = NSData(contentsOfFile: path) else {
```

```

        return
    }

    let dialID: UInt32 = 2147483539

    // 删除表盘
    YCProduct.deleteWatchFace(dialID: dialID) { state, response in

        // 直接下载
        YCProduct.downloadWatchFace(
            isEnabled: true,
            data: customDialData,
            dialID: dialID,
            blockCount: 0,
            dialVersion: 1) { state, response in
                if state == .succeed,
                    let info = response as? YCDownloadProgressInfo {
                        print(info.downloaded, info.progress)
                    } else {

                }
            }
        }
    }
}

```

9.6 自定义表盘

- 方法

```

/// 生成自定义表盘数据
/// - Parameters:
///   - dialData: 原始表盘数据
///   - backgroundImage: 背景图片
///   - thumbnail: 缩略图片
///   - timePosition: 时间显示位置坐标
///   - redColor: 0 ~ 255
///   - greenColor: 0 ~ 255
///   - blueColor: 0 ~ 255
///   - isFlipColor: 是否要翻转颜色
/// - Returns: 表盘数据
public static func generateCustomDialData(
    _ dialData: Data,
    backgroundImage: UIImage?,
    thumbnail: UIImage?,

```

```

        timePosition: CGPoint,
        redColor: UInt8,
        greenColor: UInt8,
        blueColor: UInt8,
        isFlipColor: Bool
    ) -> Data

    /// 查询表盘文件中的BMP信息
    public static func queryDeviceBmpInfo(_ dialData: Data) ->
    YCProductSDK.YCWatchFaceDataBmpInfo

    /// 表盘中的图片信息
    @objcMembers public class YCWatchFaceDataBmpInfo : NSObject {

        /// 背景图片的宽度
        public var width: Int { get }

        /// 背景图片的高度
        public var height: Int { get }

        /// 背景图片的大小(字节)
        public var size: Int { get }

        /// 背景图片的半径
        public var radius: Int { get }

        /// 缩略图片的宽度
        public var thumbnailWidth: Int { get }

        /// 缩略图片的高度
        public var thumbnailHeight: Int { get }

        /// 缩略图片的大小(字节)
        public var thumbnailSize: Int { get }

        /// 缩略图片的半径
        public var thumbnailRadius: Int { get }
    }

```

■ 说明

- 自定义表盘是基于厂商提供的自定义表盘源文件，进行图片与文字颜色的修改，产生一个新的表盘文件，下载到设备中。
- 如果不修改图片，可以传入nil，SDK会保持原来的背景图片与缩略图片。

- 生成了新的表盘文件，直接调用表盘下载的方法，下载到设备中去即可。
- 文档中给出了一个表盘BMP查询信息的方法，可能在App开发界面,或生成缩略图片时要用到相关的参数。
- 使用举例

```
let customDialData =
    YCProduct.generateCustomDialData(
        dialData as Data,
        backgroundImage: UIImage(named: "test"),
        thumbnail: UIImage(named: "test"),
        timePosition: CGPoint(x: 120, y: 120),
        redColor: 255,
        greenColor: 0,
        blueColor: 0,
        isFlipColor:
            YCProduct.shared.currentPeripheral?.supportItems.isFlipCustomDialColor ??
            false
    ) as NSData
```

9.7 杰理表盘API

杰理平台使用的是平台本身提供的API，所以前面章节列举的表盘API绝大部分不可用。

9.7.1 查询表盘信息

- 方法

```
/// 查询杰理的当前显示表盘
/// - Parameter completion: 表盘名称
public static func queryJLDeviceCurrentWatchFace(_ completion: ((_ name:
String) -> ()))?)

/// 查询杰理表盘的所有表盘信息
/// - Parameter completion: isSuccess: 查询是否成功, dials: 表盘名称,
customDials: 自定义表盘名称
public static func queryJLDeviceLocalWatchFaceInfo(
    _ completion: (
```

```

        ( _ isSuccess: Bool, _ dials: [String], _ customDials: [String]) ->
    ())?
)

/// 查询表盘的版本信息
/// - Parameters:
///   - dials: 本地表盘
///   - completion: 表盘的版本与表盘id信息
public static func queryJLDeviceWatchVersionInfo(
    _ dials: [String],
    completion: ((_ info: [YCJLDeviceWatchFaceVersionInfo]) -> ()))?
)

/// 表盘版本信息
@objcMembers public class YCJLDeviceWatchFaceVersionInfo: NSObject {

    /// 表盘名称
    public var name: String = ""

    /// 表盘的id
    public var dialID: String = ""

    /// 表盘的版本
    public var version: String = ""

    init(name: String, dialID: String, version: String)
}

```

- 说明

- 表盘查询支持9.1中的API，此外本章节中也列举了三个，分别查询设备中的表盘信息与表盘对应的版本与表盘id，其中查询到的表盘名称都是 WATCH 开头，自定义表盘都是 BGP 开头。

- 使用举例

```

/// 查询当前表盘
YCProduct.queryJLDeviceCurrentWatchFace { name in
    print("==== 当前表盘: \(name)")
}

/// 查询设备中的所有表盘信息

```

```

YCProduct.queryJLDeviceLocalWatchFaceInfo { isSuccess, dialCaches,
customDialCaches in

    guard isSuccess else {
        return
    }

    print("\(dialCaches), \(customDialCaches)")
// FIXME: - 暂时不需要读取杰理表盘ID信息
//          YCProduct.queryJLDeviceWatchVersionInfo(dialCaches) { infos
in
//
//
//          }
}

```

9.7.2 设置表盘

- 方法

```

/// 设置杰理表盘
/// - Parameters:
///   - watchFaceName: 表盘名称
///   - isCustomWatchFace: 是否为自定义表盘
///   - completion: 设置是否成功
public static func settingJLDeviceWatchFace(
    _ watchFaceName: String,
    isCustomWatchFace: Bool = false,
    completion: ((_ isSuccess: Bool) -> ())?
)

```

- 说明

- 设置表盘也是依据名称来区分，包含设置普通表盘和自定义表盘，设置自定义表盘会在自定义表盘中单独说明。

- 使用举例

```
// 设置WATCH2为当前表盘
YCProduct.settingJLDeviceWatchFace("WATCH2") { [weak self] isSuccess in

    if isSuccess {
        print("success")
    } else {
        print("failed")
    }
}
```

9.7.3 删除表盘

- 方法

```
/// 删除表盘
/// - Parameters:
///   - watchFaceName: 表盘名称
///   - compleiton: 删除是否成功
public static func deleteJLDeviceWatchFace(
    _ watchFaceName: String,
    completion: ((_ isSuccess: Bool) -> ())?
)
```

- 说明
 - 和设置表盘一样，直接使用表盘名称做参数。
- 使用举例

```
// 删除 WATCH2
YCProduct.deleteJLDeviceWatchFace("WATCH2") { isSuccess in
    if isSuccess {
        print("success")
    } else {
        print("failed")
    }
}
```

9.7.4 下载表盘

- 方法

```
/// 安装杰理表盘
/// - Parameters:
///   - watchFaceName: 表盘名称
///   - dialData: 表盘数据
///   - completion: 安装进度progress 0 ~ 1.0 与结果 state
public static func installJLDeviceWatchFace(
    _ watchFaceName: String,
    dialData: Data,
    completion: @escaping ((_ state: JLDeviceWatchFaceState, _ progress:
Float) -> ()))
)

/// 表盘安装状态
@objc public enum JLDeviceWatchFaceState: Int {
    case noSpace                = 0 // 没有空间
    case installing              // 正在安装
    case success                 // 安装成功
    case failed                  // 安装失败
}
```

- 说明

- 安装表盘需要表盘名字与表盘文件数据作为参数。
- 安装成功后，需要主动设置一下安装表盘，才会显示出来。

- 使用举例


```
// 安装表盘 WATCH2
guard let dialData = NSData(contentsOfFile: "表盘文件路径") else {
    return
}

YCProduct.installJLDeviceWatchFace(
    "WATCH2",
    dialData: dialData as Data) { state, progress in
    if state == .success {
        YCProduct.settingJLDeviceWatchFace("WATCH2") { [weak self]
isSuccess in
        }
    }
}
}
```

9.7.5 设备表盘切换

- 方法

```
/// 杰理表盘切换通知
public static let jlDeviceWachFaceChangeNotification: Notification.Name

/// 杰理表盘切换有key
public static let jlDeviceWatcFaceChangeKey: String
```

- 说明

- 手表切换或删除了表盘会将当前界面显示的表盘名称发送出去，App只需要监听通知就可以了。

- 使用举例

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(receiveJLWatchFaceChange(_:)),
    name: YCProduct.jlDeviceWachFaceChangeNotification,
    object: nil
)

@objc private func receiveJLWatchFaceChange(_ ntf: Notification) {
```

```

        guard let info = ntf.object as? [String: Any],
              let name = info[YCProduct.jlDeviceWatchFaceChangeKey] as? String
        else {
            return
        }
        print("\(name)")
    }
}

```

9.7.6 自定义表盘

▪ 方法

```

/// 转换自定义表盘
/// - Parameters:
///   - watchName: 表盘名称
///   - backgroundImage: 背景图片
///   - completion: customWatchName - 自定义表盘名称, dialData - 表盘数据
public static func convertJLCustomWatchFaceInfo(
    _ watchName: String,
    backgroundImage: UIImage,
    completion: @escaping (_ customWatchName: String, _ dialData : Data?) -
>()
)

```

▪ 说明

- 自定义表盘是基于表盘文件变换背景图片，也就是说只是更改了绑定的背景图片。
- 表盘转换成功后，再下载到手表中，下载表盘API与普通表盘是相同的，只是表盘名称变为了自定义的表盘名称。
- 下载完成后，需要设置表盘才会显示出来，设置表盘接口与普通表盘相同，只是参数变化。
- 注意：杰理的自定义表盘设置的前提是表盘已存在于手表中，否则不能更换图片

▪ 使用举例

```

// 下载自定义表盘
YCProduct.convertJLCustomWatchFaceInfo(
    "WATCH100",
    backgroundImage: UIImage(named: "test")) { [weak self] customWatchName,
dialData in

```

```

YCPProduct.installJLDeviceWatchFace(
    customWatchName,
    dialData: dialData ?? Data()) { state, progress in

        if state == .success {

            YCPProduct.settingJLDeviceWatchFace(customWatchName,
isCustomWatchFace: true) { isSuccess in
                printLog("\(isSuccess)")
            }
        }
    }
}

```

10. 历史数据采集

注意：这部分主要是获取ECG和PPG数据，其它类型暂时不支持。

```

/// 采集数据类型
@objc public enum YCCollectDataType : UInt8 {
    case ecg
    // 心电图数据
    case ppg
    // PPG数据
}

```

10.1 查询信息记录

- 方法

```

/// 查询本地历史采集数据的基本信息
/// - Parameters:
///   - peripheral: 已 已连接设备
///   - dataType: 数据类型
///   - completion: 信息记录
public static func queryCollectDataBasicinfo(
    _ peripheral: CBPeripheral? = YCPProduct.shared.currentPeripheral,
    dataType: YCCollectDataType,

```

```

        completion: ((_ state: YCProductState, _ response: Any?) -> (()))?
    )

    /// 历史采集数据的基本信息
    @objcMembers public class YCCollectDataBasicInfo : NSObject {

        /// 类型
        public var dataType: YCProductSDK.YCCollectDataType { get }

        /// 序号
        public var index: UInt16 { get }

        /// 时间戳(秒)
        public var timeStamp: UInt32 { get }

        /// 采样率
        public var sampleRate: UInt16 { get }

        /// 样本位数
        public var samplesCount: UInt8 { get }

        /// 总字节数
        public var totalBytes: UInt32 { get }

        /// 总包数
        public var packages: UInt16 { get }
    }

```

- 说明
 - 调用此方法，可以获取到当前设备有几条记录，每条记录的基本信息
- 使用举例

```

YCProduct.queryCollectDataBasicinfo(dataType: .ecg) { state, response in
    guard state == .succeed,
        let datas = response as? [YCCollectDataBasicInfo] else {
        return
    }
    print(datas)
}

```

10.2 获取具体的数据

- 方法

```
/// 通过索引获取数据
/// - Parameters:
///   - peripheral: 已 已连接设备
///   - dataType: 数据采集类型
///   - index: 索引
///   - uploadEnable: 是否上报数据
///   - completion: 结果
public static func queryCollectDataInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCCollectDataType,
    index: UInt16 = 0,
    uploadEnable: Bool = true,
    completion: ((_ state: YCProductState, _ response: Any?) -> (()))?
)

/// 历史采集数据信息
@objcMembers public class YCCollectDataInfo: NSObject {

    /// 基础信息
    public var basicInfo: YCCollectDataBasicInfo { get }

    /// 同步进度
    public var progress: Float { get }

    /// 是否传输完成
    public var isFinished: Bool { get }

    /// 响应数据
    public var data: [Int32] { get }

    /// 衍生数据 (基于data)
    public var processedData: [Any] {get}
}
```

- 说明

- 此方法的返回值是 YCCollectDataInfo 类型，包含数据，进度等信息。

- 使用举例

10.3 删除数据

11. 固件升级

11.1 获取设备升级相关信息

11.1.1 芯片型号

固件升级前，需要确认设备使用的硬件平台，可以通过5.6节来获取设备使用的主控芯片型号来调用不同的升级库文件。nrf52832使用的是Nordic的升级库，rtk8762c或rtk8762d使用的是Realtek的升级库。

11.1.2 固件版本

可以通过5.2节来获取设备使用的固件版本信息包含主版本与子版本，判断版本号的大小时要注意，如果主版本不相同则主版本值越大表示版本越高。如果主版本相同则比较子版本，子版本的数字越大代表版本越高。例如固件版本1.10就要比1.1版本更高，不能从数学数值来判断。

11.2 Nordic固件升级

11.2.1 导入库文件

Nordic固件升级库可以直接在github中获取，获取地址是 <https://github.com/NordicSemiconductor/IOS-DFU-Library>。

建议使用Pod来安装

```
target 'YourAppTargetName' do
  use_frameworks!
  pod 'iOSSDFULibrary'
end
```

11.2.2 实现固件升级

方法中的API参数在Nordic升级库有详细的解释说明

```
import iOSSDFULibrary

class YCFirmwareUpgradeViewController: UIViewController {

    /// NRF升级控制
```

```

private var dfuController: DFUServiceController?

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)

    _ = dfuController?.abort()
}

}

// MARK: - 固件升级
extension YCFirmwareUpgradeViewController: DFUServiceDelegate,
DFUProgressDelegate {

    /// NRF固件升级（自定义实现）
    /// - Parameters:
    ///   - filePath: 固件文件地址
    ///   - device: 需要升级的设备
    private func startNRFFirmwareUpgrade(_ filePath: URL,
                                         device: CBPeripheral) {

        guard let dfuFirmware = DFUFirmware(urlToZipFile: filePath) else {
            printLog("固件不存在")
            return
        }

        let initiator =
            DFUServiceInitiator(queue: DispatchQueue.main,
                               delegateQueue: DispatchQueue.main,
                               progressQueue: DispatchQueue.main,
                               loggerQueue: DispatchQueue.main
                                )
        initiator.delegate = self
        initiator.progressDelegate = self
        initiator.forceDfu = false
        initiator.alternativeAdvertisingNameEnabled = false
        initiator.enableUnsafeExperimentalButtonlessServiceInSecureDfu =
true

        _ = initiator.with(firmware: dfuFirmware)
        dfuController = initiator.start(target: device)
    }

    /// 状态变化

```



```
func dfuStateDidChange(to state: DFUState) {
    switch state {

        case .disconnecting:
            break

        case .connecting:
            break

        case .starting:
            break

        case .enablingDfuMode: // 进入升级状态
            break

        case .completed: // 固件升级结束
            break

        case .aborted:
            break

        default:
            break
    }

    /// 升级出错
    func dfuError(_ error: DFUError, didOccurWithMessage message: String) {

    }

    /// 升级进度
    func dfuProgressDidChange(for part: Int, outOf totalParts: Int, to
    progress: Int, currentSpeedBytesPerSecond: Double, avgSpeedBytesPerSecond:
    Double) {

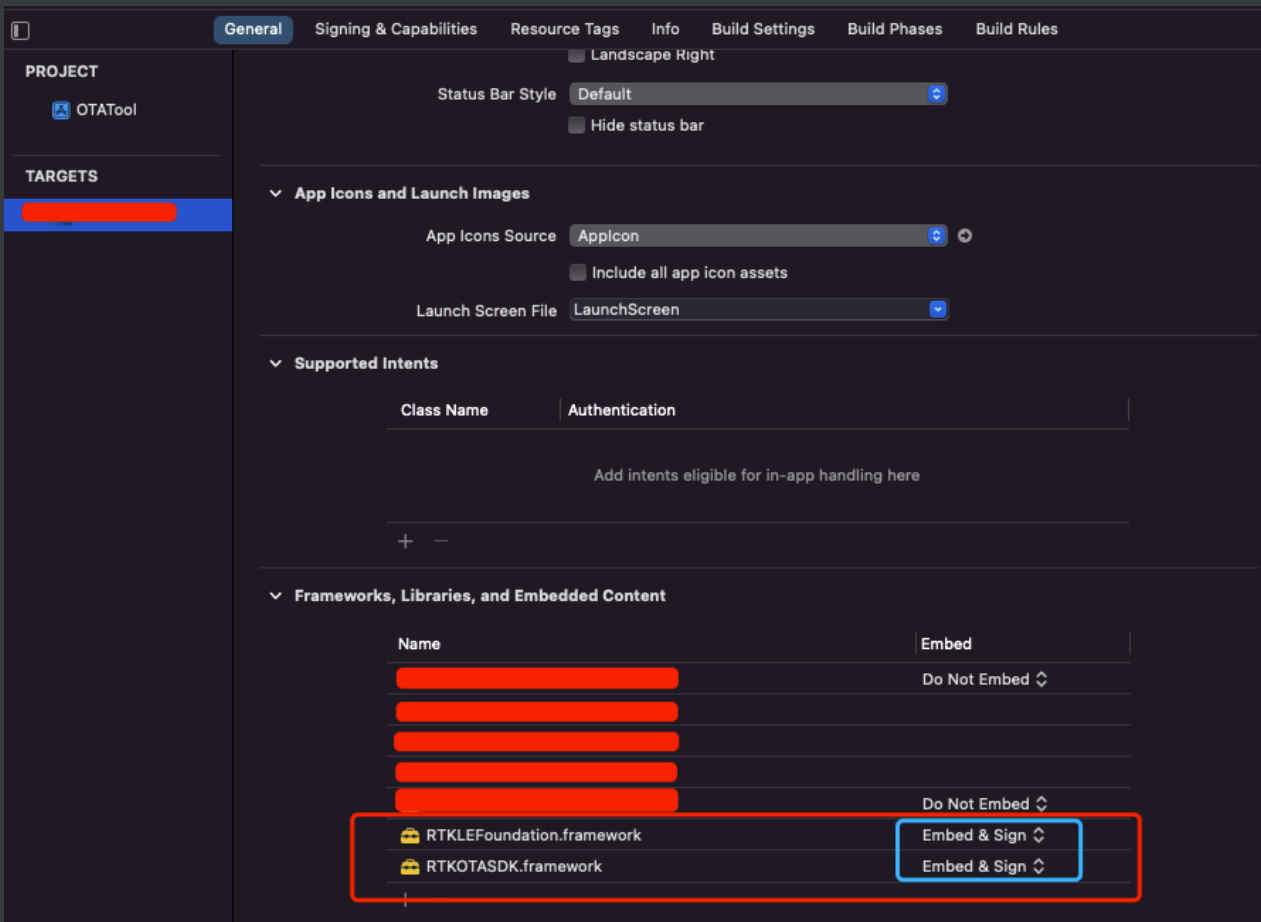
    }
}
```

11.3 Realtek固件升级

Realtek的升级操作与Nordic相比复杂的多

11.3.1 导入库文件

在SDK文件目录下提供了两个Realtek库文件 RTK0TASDK.framework 和 RTKLEFoundation.framework，将这两个库导入到工程中去。如果已导入请忽略。



11.3.2 实现固件升级

```
import UIKit
import RTK0TASDK

class YCFirmwareUpgradeViewController: UIViewController {

    /// RTK使用 任何操作都需要的管理对象
    private var rtkDFUUpgrade: RTKDFUUpgrade?

    /// RTL升级的固件镜像
    private var rtkImages: [RTK0TAUpgradeBin]?

}
```

```

// MARK: - 固件升级
extension YCFirmwareUpgradeViewController {

    /// 升级RTK
    /// - Parameters:
    ///   - device: rtkDevice
    ///   - filePath: 固件文件地址
    private func startRTKFirmwareUpgrade(_ device: CBPeripheral, filePath:
String) {

        #if DEBUG
        RTKLog.setLogLevel(.warning)
        #else
        RTKLog.setLogLevel(.off)
        #endif

        rtkDFUUpgrade = RTKDFUUpgrade(peripheral: device)
        rtkDFUUpgrade?.delegate = self

        // 严格检查版本
        rtkDFUUpgrade?.usingStrictImageCheckMechanism = true

        // 电量检查
        rtkDFUUpgrade?.batteryLevelLimit = RTKDFUBatteryLevel.zero

        // 加载文件
        guard let zipFile = try?
RTKOTAUpgradeBin.imagesExtracted(fromMPPackFilePath: filePath),
            zipFile.count == 1,
            zipFile.last?.isDetermined == false else {

            return
        }

        rtkImages = zipFile
        rtkDFUUpgrade?.prepareForUpgrade()

    }
}

// MARK: - 升级相关代理
extension YCFirmwareUpgradeViewController: RTKDFUUpgradeDelegate {

    /// 升级进度

```

```

func dfuUpgrade(_ task: RTKDFUUpgrade, withDevice connection:
RTKProfileConnection, didSendBytesCount length: UInt, ofImage image:
RTKOTAUpgradeBin) {
    print(task.progress.fractionCompleted)
}

/// 升级进度 结束
func dfuUpgrade(_ task: RTKDFUUpgrade, withDevice connection:
RTKProfileConnection, didCompleteSendImage image: RTKOTAUpgradeBin) {
    print(task.progress.fractionCompleted)
}

/// 准备升级
func dfuUpgradeDidReady(for task: RTKDFUUpgrade) {

    guard let deviceInfo = task.deviceInfo,
        let upgradeBins = rtkImages else {
        print("升级失败")
        return
    }

    // 准备升级
    upgradeBins.last?.assertAvailable(forPeripheralInfo: deviceInfo)
    rtkDFUUpgrade?.upgrade(withImages: upgradeBins)
}

/// 升级结束
func dfuUpgrade(_ task: RTKDFUUpgrade, didFinishUpgradeWithError error:
Error?) {
    if err == nil {
        print("升级成功")
    } else {
        print("升级失败")
    }
}

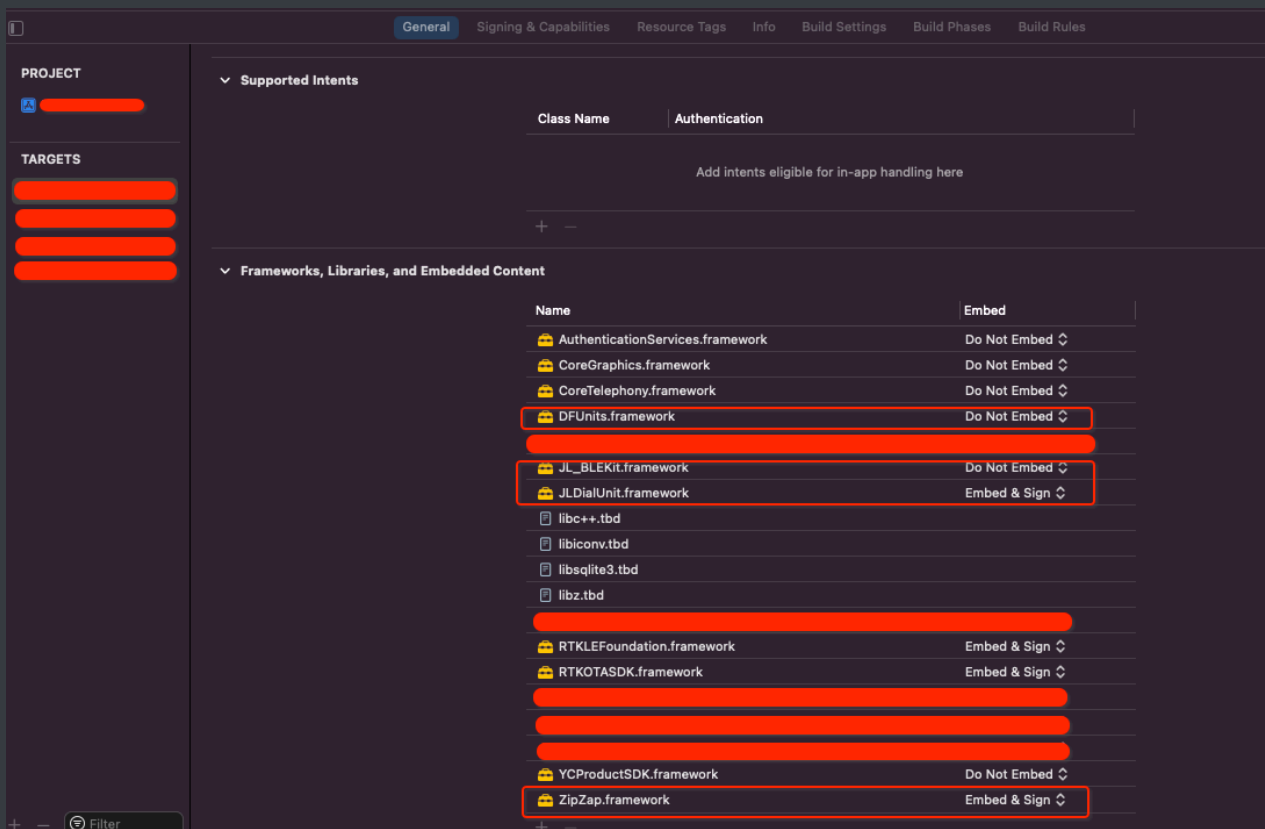
/// 升级出错（开始）
func dfuUpgrade(_ task: RTKDFUUpgrade, couldNotUpgradeWithError error:
Error) {
    print("升级失败")
}
}

```

11.4 杰理固件升级

11.4.1 导入库文件

如果已导入，请忽略。



11.4.2 杰理固件升级

杰理固件升级的过程比较多，主要分为以下四个步骤：

1. 判断当前升级状态，如果是正常状态，则按顺序进行后续步骤(2 ~ 3)，如果是强制状态则直接进入最后一步(第d三步)。
2. 更新UI文件 (更新结束后，设备会断开连接，需要手动重新连接，进入下一步。)
3. 更新固件

```
/// 杰理固件升级状态
@objc public enum JLDeviceUpdateState: Int {
    case start = 0 // 开始
    case resourceUpdating // 正在更新资源
    case updateResourceFinished // 更新资源完成
    case uiUpdating // 正在更新UI
```

```

        case updateUIFinished // UI升级完成
        case upgrading // 正在ota升级
        case success // 整个升级完成
        case failed // 失败
    }

    /// 是否强制升级
    /// - Returns: true 强制升级
    public static func isJLDeviceForceOTA() -> Bool

    /// 杰理设备升级文件
    /// - Parameters:
    ///     - filePath: 升级文件路径
    ///     - completion: state 升级状态, progress: 进度 0 ~ 1.0, didSend: 已发送字节数
    public static func jlDeviceUpgradeFirmware(
        filePath: String,
        completion: ((_ state: JLDeviceUpdateState, _ progress: Float, _
        didSend: Float) -> ())?
    )

```

```

import UIKit
import YCProductSDK
import JLDialUnit
import JL_BLEKit

/// 重复扫描次数
private let REPEAT_SCAN_JL_FORCE_OTA_COUNT = 5

/// 重复扫描次数
private var repeatScanJLCount: Int = 0

/// 升级文件
private var upgradeFilePath: String = ""

/// 回连地址
private var reconnectMacAddress: String = ""

/// 设备列表
private var deviceArray = [CBPeripheral]()

```

```

// MARK: - OTA升级
extension YCDeviceUpgradeManager {

    /// 固件升级
    /// - Parameters:
    ///   - filePath: 文件
    ///   - device: 设备
    func startJLFWUpgrade(_ filePath: String,
                          device: CBPeripheral) {

        upgradeFilePath = filePath
        reconnectMacAddress = device.macAddress

        jlDeviceOTA(device, filePath: filePath)
    }

    /// 杰理OTA
    /// - Parameters:
    ///   - device: <#device description#>
    ///   - filePath: <#filePath description#>
    private func jlDeviceOTA(
        _ device: CBPeripheral,
        filePath: String
    ) {

        YCProduct.jlDeviceUpgradeFW(device, filePath: filePath) {
[weak self] state, progress, didSend in

            switch state {

                case .start:
                    // 开始
                    break

                case .success:
                    // 升级成功
                    break

                case .failed:
                    // 升级失败

```

```

        break

        case .uiUpdating:
            // 升级UI
            break

        case .upgrading:
            // 正在升级
            break

        case .updateUIFinished:
            self?.reconnectWithMacAddr()

        default:
            break
    }
}

/// OTA回连设备
func reconnectWithMacAddr() {
    // 需要等设备重启
    usleep(2500_000)
    repeatScanJLCount = 0
    scanJLForceOtaDevice()
}

/// 扫描设备
@objc private func scanJLForceOtaDevice() {

    repeatScanJLCount += 1
    if repeatScanJLCount >= REPEAT_SCAN_JL_FORCE_OTA_COUNT {

        // 没有找到设备，升级失败。
        return
    }

    // 搜索设备
    YCProduct.scanningDevice(delayTime: 4.0 ) { devices, error in
        for device in devices where deviceArray.contains(device) ==
false {
            deviceArray.append(device)
        }
    }
}

```



```

        // 搜索结束 准备连接
        perform(
            #selector(connectForceOtaDevice),
            with: nil,
            afterDelay: 4.5
        )
    }

    /// 重连设备
    @objc private func connectForceOtaDevice() {

        // 查找设备
        for device in deviceArray {

            if device.macAddress.uppercased() ==
reconnectMacAddress.uppercased() {

                YCProduct.connectDevice(device) { [weak self] state, error
in

                    if state == .connected {

                        // 回连成功, 准备升级
                        self?.startJLFirmwareUpgrade(
                            upgradeFilePath,
                            device: device
                        )

                    } else {

                        // 回连失败, 重新搜索。
                        self?.scanJLForceOtaDevice()
                    }
                }

                return
            }
        }

        // 没有找到设备, 重新搜索。
        scanJLForceOtaDevice()
    }

```

