YCProductSDK revision history

Version	Content description	Revision date	Revised by
1.0.0	1. Swift initial version	2021-12-30	Mark

1 Overview

This document will explain the APIs and usage scenarios related to the functions used in the Bluetooth device.This document is only applicable to wearable devices such as bracelets or watches made by Yucheng.

1.1 Applicable readers

For engineers who use this document for APP development, they should have the following basic skills:

1. Have basic iOS development experience
2. Need to master the Swift language and have the knowledge reserve of mutual calling between Objective-C and Swift.
3. Understand the basic process of Bluetooth development in iOS

1.2 Related terms

App: This article refers to applications running on mobile phones or tablets

Device: This article refers to wearable hardware devices: such as bracelets, watches, etc.

Upload: Refers to the device sending data to the App

Delivery: Refers to the app sending data to the device

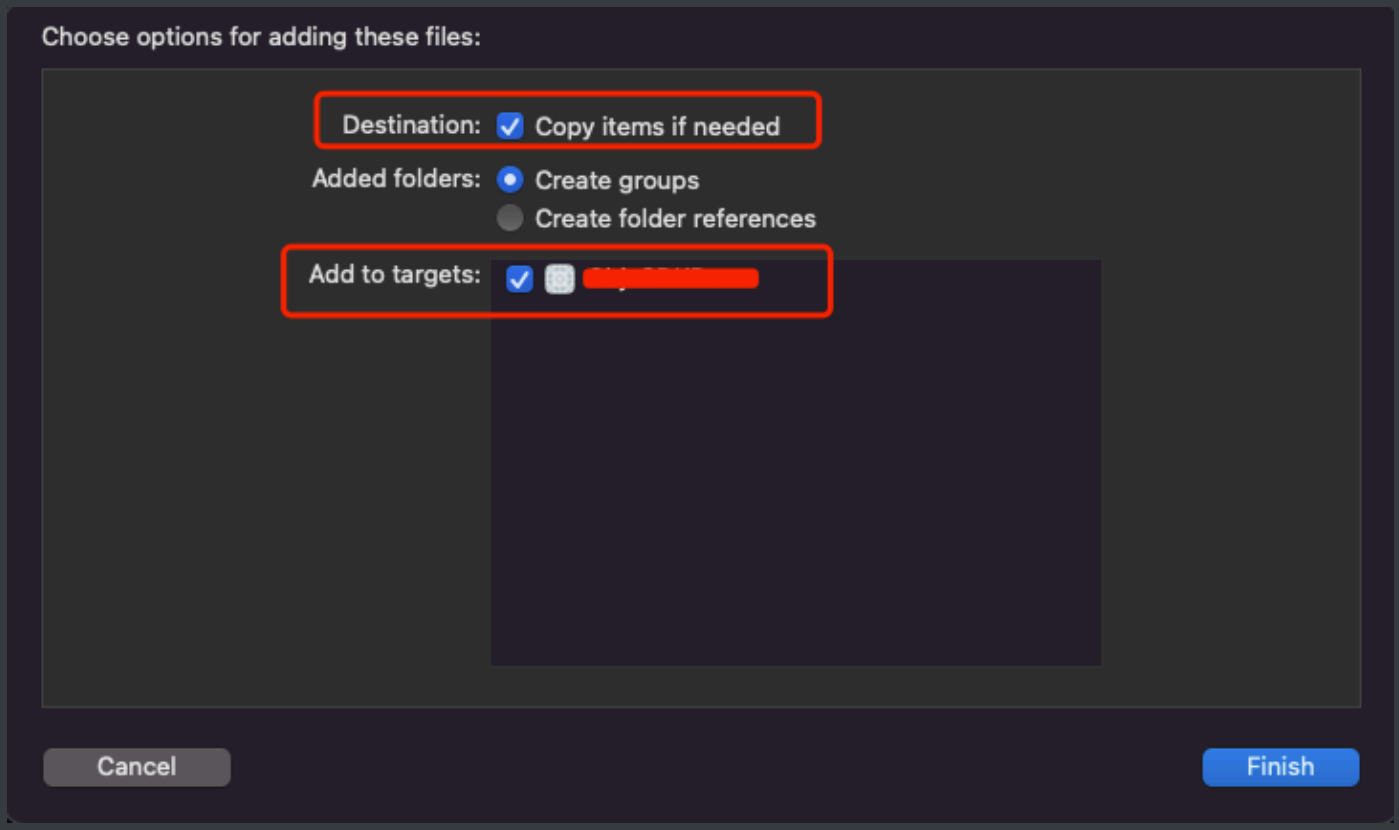
1.3 Description

1. All the APIs in the document will be demonstrated in the corresponding Demo. For the functions of the bracelet, you can refer to the SmartHeatlh application published in the AppStore for use. Combining the experience of using and reading this document will greatly improve efficiency.
2. The API in the document does not apply to all devices, that is, the device may only support some of the functions in the API, and it can be judged whether it is available through the return value or attribute of the API.
3. In the development and debugging stage, it is strongly recommended that you turn on the log switch to facilitate error information to help locate the problem.
4. This document only describes the Swift version. If you use Objective-C language to develop, you can combine the document and use Xcode's smart prompts to call the corresponding Method.
5. For the introduction of API, the principle is to explain according to the classification of instructions. If there are related functions, it may be put in a chapter or a mark that refers to the API will be introduced.

2. SDK integration instructions

2.1 Integrated SDK

1. Drag and drop the file `YCProduct.framework` directly into the project, or import using pod.




```
platform :ios, '9.0'

target 'Your app' do
  use_frameworks!

  pod 'YCProductSDK-Swift'

end
```

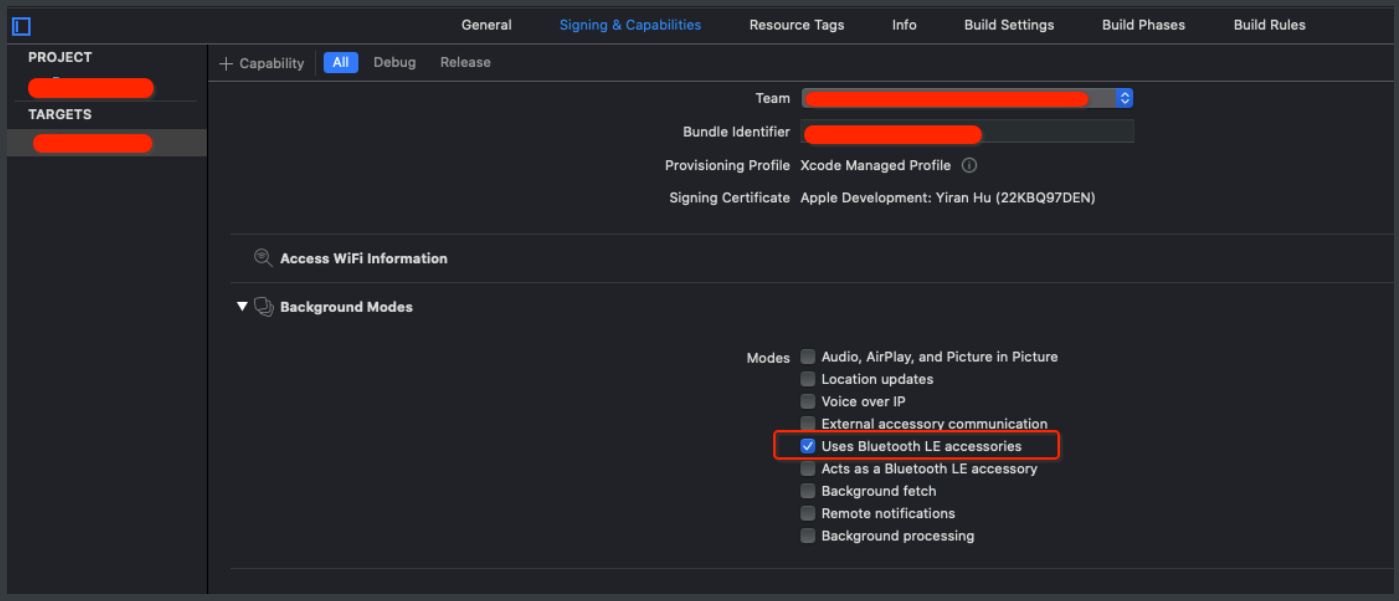
2. Configure the info.plist file

The use of Bluetooth needs to be explained in the info.plist, the keys used in the versions before and after iOS13 are different

NSBluetoothAlwaysUsageDescription (iOS 13+)

NSBluetoothPeripheralUsageDescription

3. Configure Bluetooth access



4. Project import

```swift
// Import of Swift project
import CoreBluetooth
import YCProductSDK
```

```objc
// Import of OC project
@import CoreBluetooth
@import YCProductSDK;
```

2.2 SDK log printing

Currently, the print function only supports opening and closing, and does not support multiple level settings for the time being.

```
/// Log level (currently only supports two, on and off)
@objc public enum YCProductLogLevel : Int {
    case off      // on
    case normal   // off
}

/// Log printing function, default is off
public static func setLogLevel(_ level: YCProductSDK.YCProductLogLevel =
.off)
```

2.3 Status code in SDK

- Each API callback will contain two parts, one is the status code, the other is the specific information, and the specific information is the Any type, mainly based on the basic data type and class.
- The specific information returned will be explained when it is used on the specific API, and all the states will be explained here.

```
@objc public enum YCProductState : Int {

    case unknow               // Bluetooth status is unknown
    case resetting            // Bluetooth reset
    case unsupported          // Does not support Bluetooth
    case unauthorized         // Bluetooth is not authorized
    case poweredOff           // Bluetooth off
    case poweredOn            // Bluetooth is on
    case disconnected         // Bluetooth disconnect
    case connected            // Bluetooth is connected
    case connectedFailed      // Bluetooth connection failed

    case succeed              // Success
    case failed               // Fail
```

```
    case unavailable          // API is not available, device does not
support
    case timeout              // time out
    case dataError            // data error
    case crcError             // crc error
    case dataTypeError        // Data type error
    case noRecord             // No record
    case parameterError       // Parameter error

    case alarmNotExist        // Alarm clock does not exist
    case alarmAlreadyExist    // Alarm already exists
    case alarmCountLimit      // The number of alarms reaches the limit
    case alarmTypeNotSupport  // Alarm clock type is not supported
}
```

2.4 SDK initialization

After integrating the SDK, you need to call the initialization Method (required). The SDK initialization Method will do some simple settings.

```
_ = YCProduct.shared
```

2.5 Supplement

1. There are a lot of type declarations in the API, which will only be explained when they are used.
2. If you find that some types are not given instructions when reading the document, you can use the global search Method, which may have been given in other places.
3. If the entire document does not give a definition, you can write this type directly in Xcode, and then use the Xcode jump function to jump to the internal definition of

the framework.

4. In the process of using the SDK, if you encounter a problem, you can first try to use our application or Demo to determine whether the device itself is faulty, or there is a bug in the SDK, etc., and give us feedback in time.

3. Device search and connection

3.1 Device status

- Method

```
/// Device status changes
public static let deviceStateNotification: Notification.Name

/// State key
public static let connecteStateKey: String
```

- Instruction
    - The SDK will monitor the connection status of the device and will send it in the form of Notification. The App can monitor this message globally.
    - The message in Notification is stored in the key of `connecteStateKey`, and the corresponding state can be obtained through the key.
- Examples of use

```
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 Search device

- Method

```
/// Start scanning for devices
/// - Parameters:
///   - delayTime: delay to stop searching, default 3 seconds
///   - completion: result
public static func scanningDevice(
  delayTime: TimeInterval = 3.0,
  completion: (([CBPeripheral],   NSError?) -> ())?
)
```

- Instruction
    - After calling the method search device, it will stop automatically. The search time is determined by delayTimeg. The default time is 3 seconds. You can also set any time. It is recommended to use the default value.
    - The searched device will be returned in the callback when it ends.
- Examples of use

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

3.3 Connect the device

- Method

```
/// Connect the device
/// - Parameter peripheral:the user selects the device that needs to be connected
public static func connectDevice(
  _ peripheral: CBPeripheral,
  completion: ((YCProductSDK.YCProductState, NSError?) -> ())?
)
```

- Instruction
    - You can select any device from the searched devices to connect.
    - After selecting the connected device, you need to wait for a while, and the connection result will be returned in the callback.
- Examples of use

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

3.4 Disconnect device

- Method

```
/// Disconnect device
/// - Parameter peripheral: currently connected device
public static func disconnectDevice(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral
)
```

- Instruction
    - The connected device, calling this method, will disconnect from the SDK. Note that it will not disconnect from the iOS system.
    - After disconnecting, the SDK will clear the back-connect mark, and the device will not be back-connected.
- Examples of use

```
YCProduct.disconnectDevice(devcie)
```

3.5 Get the currently connected device

```
/// Current connected peripheral
public  var currentPeripheral: CBPeripheral? { get }

/// All peripherals currently connected (reserved parameters, extended
use)
public  var connectedPeripherals: [CBPeripheral] { get }
```

- Instruction
    - The SDK will save the connected devices to the array, and at the same time save the last connected device to the current device for easy use.
- Examples of use

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

3.6 Device back and long connection

1. After the SDK is successfully connected, it will save the parameters of the last successfully connected device. Once the device is disconnected, the SDK will actively reconnect the device.
2. If you call the method(3.3) of the disconnected device in the SDK, the SDK will not connect back to the device.
3. If the App wants to realize the function of keeping the connection all the time, it is realized by the App itself, and the SDK does not realize the function of keeping the connection in the background.

4. Get health data

- Obtaining the health data of the device refers to querying the heart rate, blood pressure, sleep, blood oxygen and other data of the device through active or automatic detection.
- The types of health data in the device are not the same, please call according to the return value of the device and the function support switch.
- Among the data types supported by the device, the first 5 data types (step, sleep, heartRate, bloodPressure, combinedData) are supported by most devices, and the rest are only supported by customized versions.
- Note: Do not use the heart rate, steps, sleep, blood pressure obtained through combinedData.
- The data in the device `will not be actively deleted` , so after the App is obtained, it should be deleted actively, otherwise the same data will be obtained the next time it is obtained, and once the data in the device exceeds the storage size, it will be automatically deleted.

4.1 Retrieve data

- Method

```
/// Query the type definition of health data
@objc public enum YCQueryHealthDataType: UInt8 {

  case step                       // Step data
  case sleep                      // Sleep data
  case heartRate                  // Heart rate data
  case bloodPressure              // Blood pressure data
  case combinedData               // Combined data (blood oxygen,
respiration rate, temperature, body fat, hrv, cvrr)

  case bloodOxygen                // Blood oxygen data
  case temperatureHumidity        // Environmental temperature and
humidity data
  case bodyTemperature            // Body temperature data
  case ambientLight               // Ambient light data
  case wearState                  // Wearing status record
  case healthMonitoringData       // Health monitoring data
  case sportModeHistoryData       // Exercise history data
}

/// Query health data
/// - Parameters:
///   - peripheral: current device
///   - datatType:  YCQueryHealthDataType
///   - completion: result
public static func queryHealthData(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    datatType: YCQueryHealthDataType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the corresponding data is queried, the result will be saved in the response. The results returned by each type are different and need to be converted into corresponding objects. The specific types have been listed one by one, please refer to 4.3.
- Examples of use

```
// Query sleep
YCProduct.queryHealthData(datatType: YCQueryHealthDataType.sleep) {
state, response in

    if state == .succeed, let datas = response as? [YCHealthDataSleep] {
        for info in datas {
            print(info.startTimeStamp,
                  info.endTimeStamp,
                  info.lightSleepCount,
                  info.lightSleepMinutes,
                  info.deepSleepCount,
                  info.deepSleepMinutes,
                  info.sleepDetailDatas
            )
        }
    }
}

// Query combined data
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
                )
            }
        }
    }

    // For other types, please refer to Demo
```

4.2 Delete data

- Method

```
    /// Delete the type definition of health data
    @objc public enum YCDeleteHealthDataType: UInt8 {

        case step
        case sleep
        case heartRate
        case bloodPressure
        case combinedData

        case bloodOxygen
        case temperatureHumidity
        case bodyTemperature
        case ambientLight
        case wearState
        case healthMonitoringData
        case sportModeHistoryData
    }

    /// Delete health data
    /// - Parameters:
    ///   - peripheral: currently connected device
```

```
///   - datatType: YCDeleteHealthDataType
///   - completion: result
public static func deleteHealthData(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    datatType: YCDeleteHealthDataType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction：
    - When calling method, just pass in the specific type to the dataType. If the deletion is successful, the state will return succeed, otherwise it means the operation failed.
- Examples of use

```
// Delete steps
YCProduct.deleteHealthData(datatType: YCDeleteHealthDataType.step) {
state, response in

    if state == .succeed {

        print("Delete succeed")
    }
}

// Delete combined data
YCProduct.deleteHealthData(datatType:
YCDeleteHealthDataType.combinedData) { state, response in

    if state == .succeed {

        print("Delete succeed")
    }
}

// For other types, please refer to Demo
```

4.3 Health data type and corresponding return value type

4.3.1 Step

```
//  YCQueryHealthDataType.step
/// Steps information
@objcMembers public class YCHealthDataStep : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// End timestamp (seconds)
    public  var endTimeStamp: Int { get }

    /// Steps (steps)
    public  var step: Int { get }

    /// Distance (meter)
    public  var distance: Int { get }

    /// Calories (kcal)
    public  var calories: Int { get }
}
```

4.3.2 Sleep

```
// YCQueryHealthDataType.sleep

/// Sleep data
@objcMembers public class YCHealthDataSleep : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// End timestamp (seconds)
    public  var endTimeStamp: Int { get }

    /// Deep sleep times
    public  var deepSleepCount: Int { get }

    /// Total time of deep sleep (unit: minute)
    public  var deepSleepMinutes: Int { get }

    /// REM duration (unit: minute)
    public  var remSleepMinutes: Int { get }

    /// Light sleep times
    public  var lightSleepCount: Int { get }

    /// Total light sleep time (unit: minute)
    public  var lightSleepMinutes: Int { get }

    /// Sleep detailed data
    public  var sleepDetailDatas: [YCProductSDK.YCHealthDataSleepDetail]
{ get }
}

/// Deep sleep and light sleep types
@objc public enum YCHealthDataSleepType : Int {
    case unknow
```

```
    case deepSleep
    case lightSleep
}

/// Sleep detailed data
@objcMembers public class YCHealthDataSleepDetail : NSObject {

    /// Deep sleep or light sleep mark
    public  var sleepType: YCProductSDK.YCHealthDataSleepType { get }

    /// Sleep start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Sleep duration (unit: second)
    public  var duration: Int { get }
}
```

4.3.3 Heart rate

```
// YCQueryHealthDataType.heartRate
/// Heart rate health data
@objcMembers public class YCHealthDataHeartRate : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataMeasureMode { get }

    /// Heart rate value
    public  var heartRate: Int { get }
}
```

4.3.4 Blood pressure

```
// YCQueryHealthDataType.bloodPressure
/// Blood pressure health data
@objcMembers public class YCHealthDataBloodPressure : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataBloodPressureMode { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }
}

/// measurement method
@objc public enum YCHealthDataBloodPressureMode : UInt8 {
    case single         // Single measurement
    case monitor        // Automatic monitoring
    case inflated       // Accurate measurement
}
```

4.3.5 Combined data

```
// YCQueryHealthDataType.combinedData
/// Combined data
@objcMembers public class YCHealthDataCombinedData : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }
```

```swift
    /// Steps (steps)
    public  var step: Int { get }

    /// Heart rate value
    public  var heartRate: Int { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }

    /// Blood oxygen value
    public  var bloodOxygen: Int { get }

    /// Respiration rate value
    public  var respirationRate: Int { get }

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

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

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

    /// Is the temperature valid
    public  var temperatureValid: Bool { get }

    /// Body fat
    public  var fat: Double { get }
}
```

4.3.6 Blood oxygen

```
// YCQueryHealthDataType.bloodOxygen
/// Blood oxygen health data
@objcMembers public class YCHealthDataBloodOxygen : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataMeasureMode { get }

    /// Blood oxygen value
    public  var bloodOxygen: Int { get }
}

@objc public enum YCHealthDataMeasureMode : UInt8 {
    case single       // Single measurement
    case monitor      // Automatic monitoring
}
```

4.3.7 Temperature and humidity

```
// YCQueryHealthDataType.temperatureHumidity
/// Temperature and humidity
@objcMembers public class YCHealthDataTemperatureHumidity : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataMeasureMode { get }

    /// Temperature
```

```
    public  var temperature: Double { get }

    /// Humidity
    public  var humidity: Double { get }
}
```

4..3.8 Body temperature

```
// YCQueryHealthDataType.bodyTemperature
/// Body temperature
@objcMembers public class YCHealthDataBodyTemperature : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataMeasureMode { get }

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

4.3.9 Ambient light

```
// YCQueryHealthDataType.ambientLight
/// Ambient light
@objcMembers public class YCHealthDataAmbientLight : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Measurement method
    public  var mode: YCProductSDK.YCHealthDataMeasureMode { get }

    /// Ambient light
    public  var ambientLight: Double { get }
}
```

4.3.10 Wearing state

```
// YCQueryHealthDataType.wearState

/// Wear off data
@objcMembers public class YCHealthDataWearStateHistory : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// State
    public  var state: YCProductSDK.YCHealthDataWearState { get }
}

/// Wear off state
@objc public enum YCHealthDataWearState : UInt8 {
    case wear
    case fallOff
}
```

4.3.11 Health monitoring

```
// YCQueryHealthDataType.healthMonitoringData
/// Health monitoring data
@objcMembers public class YCHealthDataMonitor : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// Steps (steps)
    public  var step: Int { get }

    /// Heart rate value
    public  var heartRate: Int { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }

    /// Blood oxygen value
    public  var bloodOxygen: Int { get }

    /// Respiration rate value
    public  var respirationRate: Int { get }

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

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

    /// Temperature
```

```
    public  var temperature: Double { get }

    /// Is the temperature valid
    public  var temperatureValid: Bool { get }

    /// Humidity
    public  var humidity: Double { get }

    /// Ambient light
    public  var ambientLight: Double { get }

    /// Sport mode
    public  var sport: YCProductSDK.YCDeviceSportType { get }

    /// Distance (meter)
    public  var distance: Int { get }

    /// Calories (kcal)
    public  var calories: Int { get }
}
```

4.3.12 Sports history data

```
// YCQueryHealthDataType.sportModeHistoryData

/// Sports history data
@objcMembers public class YCHealthDataSportModeHistory : NSObject {

    /// Start timestamp (seconds)
    public  var startTimeStamp: Int { get }

    /// End timestamp (seconds)
    public  var endTimeStamp: Int { get }
```

```
    /// Steps (steps)
    public  var step: Int { get }

    /// Distance (meter)
    public  var distance: Int { get }

    /// Calories (kcal)
    public  var calories: Int { get }

    /// Sport mode
    public  var sport: YCProductSDK.YCDeviceSportType { get }

    /// Sport start method
    public  var flag: YCProductSDK.YCHealthDataSportModeStartMethod {
get }

    /// Heart rate value
    public  var heartRate: Int { get }
}

/// Sport start method
@objc public enum YCHealthDataSportModeStartMethod : UInt8 {
    case app
    case device
}
```

5. Get device information

Note: In this chapter, starting from 5.8, only customized devices are supported.

5.1 Support function

- Method

```
extension CBPeripheral {
    /// Support function list
    public  var supportItems: YCProductSDK.YCProductFunctionSupportItems
}
```

- Instruction
    - For customized equipment, the function is already fixed, it is not necessary to use this parameter. This parameter is valid only after the device is successfully connected.
    - This parameter contains too many attributes, which are not listed in this document. Readers can jump to the definition through Xcode to view it carefully.
- Examples of use

```
guard let device = YCProduct.shared.currentPeripheral else {
    return
}

if device.supportItems.isSupportStep {
    print("step")
}

if device.supportItems.isSupportBloodPressure {
    print("blood pressure")
}
```

5.2 Basic device information

- Method

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

/// Version Information
@objcMembers public class YCDeviceVersionInfo: NSObject {

    /// version
    public var version: String { get };

    /// major version
    public var majorVersion: UInt8 { get }

    /// subversion
    public var subVersion: UInt8 { get }
}

/// Basic Information
@objcMembers public class YCDeviceBasicInfo : NSObject {

    /// Device ID
    public  var deviceID: UInt16 { get }

    /// Firmware version
    public var mcuFirmware: YCProductSDK.YCDeviceVersionInfo { get }

    /// Battery status
    public  var batterystatus: YCProductSDK.YCDeviceBatterystate { get }

    /// battery power
    public  var batteryPower: UInt8 { get }
```

```
    /// Whether to bind (reserved parameters)
    public  var isBind: Bool { get }

    /// Do you need to synchronize (retain parameters)
    public  var needSync: Bool { get }

    /// Communication protocol version (internal use)
    public var innerProtocol: YCProductSDK.YCDeviceVersionInfo { get }

    /// Inflatable Blood Pressure Firmware Information
    public  var bloodPressureFirmware: YCProductSDK.YCDeviceVersionInfo
{ get }

    /// TP firmware information
    public  var touchPanelFirmware: YCProductSDK.YCDeviceVersionInfo {
get }
}

/// Battery charge status
@objc public enum YCDeviceBatterystate : UInt8 {
    case normal       // normal
    case low          // low power
    case charging     // charging
    case full         // be filled
}
```

- Instruction
    - Query basic information, you will get an object of type `YCDeviceBasicInfo` ,
- Examples of use

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

5.3 Mac address

- Method

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

/// mac address attribute
extension CBPeripheral {
    public  var macAddress: String
}
```

- Instruction
    - To get the mac address, you can get it directly by accessing its attributes. If the value in the attribute is empty, you can get it by calling method.
- Examples of use

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

5.4 Device model

- Method

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

- Instruction
    - this method is mainly used to obtain the type number of the device, and is used to distinguish the models of different products derived from the same series.
- Examples of use

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

5.5 Theme information

- Method

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

/// Theme information
@objcMembers public class YCDeviceTheme : NSObject {

    /// Number of themes
    public  var themeCount: Int { get }

    /// Current theme index
    public  var themeIndex: Int { get }
}
```

- Instruction
    - Get the theme index and the total number of themes currently displayed on the device
- Examples of use

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

5.6 Get chip information

- Method

```
/// Get the chip model
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
}
```

- Instruction
    - Obtaining chip information refers to the manufacturer brand and model of the MCU for obtaining the device
- Examples of use

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

5.7 Get user configuration information

- Method

```
/// Get user configuration information
public static func queryDeviceUserConfiguration(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// User configuration
@objcMembers public class YCProductUserConfiguration : NSObject {

  /// Step goal
  public  var stepGoal: Int { get }

  /// Calorie goal
  public  var calorieGoal: Int { get }

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

  /// Sleep goals (hour part)
  public  var sleepGoalHour: Int { get }

  /// Sleep goal (minute part)
  public  var sleepGoalMinute: Int { get }

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

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

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

```
  /// Age
  public  var age: Int { get }

  /// Distance unit
  public  var distanceUnit: YCProductSDK.YCDeviceDistanceType { get }

  /// Weight unit
  public  var weightUnit: YCProductSDK.YCDeviceWeightType { get }

  /// Temperature unit
  public  var temperatureUnit: YCProductSDK.YCDeviceTemperatureType {
get }

  /// Display time mode
  public  var showTimeMode: YCProductSDK.YCDeviceTimeType { get }

  /// Sedentary reminder
  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 }

  /// Anti-lost
  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 }

    /// Wearing position
    public  var wearingPosition: YCProductSDK.YCDeviceWearingPositionType
{ get }

    /// Heart rate alarm switch
    public  var heartRateAlarmEnable: Bool { get }

    /// Heart rate alarm value
    public  var heartRateAlarmValue: Int { get }

    /// Whether automatic monitoring is turned on
    public  var heartMonitoringModeEnable: Bool { get }

    /// Monitoring interval
    public  var monitoringInterval: Int { get }

    /// Language
    public  var language: YCProductSDK.YCDeviceLanguageType { get }

    /// Raise your wrist to turn on the screen switch
    public  var wristBrightenScreenEnable: Bool { get }

    /// Screen brightness
    public  var brightnessLevel:
YCProductSDK.YCDeviceDisplayBrightnessLevel { get }

    /// Skin tone settings
    public  var skinColor: YCProductSDK.YCDeviceSkinColorLevel { get }

    /// Resting time
    public  var breathScreenInterval:
YCProductSDK.YCDeviceBreathScreenInterval { get }
```

```
    /// Bluetooth disconnect reminder
    public  var deviceDisconnectedReminderEnable: Bool { get }

    ///  Upload reminder switch
    public  var uploadReminderEnable: Bool { get }

    /// Do not disturb
    public  var notDisturbEnable: Bool { get }
    public  var notDisturbStartHour: Int { get }
    public  var notDisturbStartMinute: Int { get }
    public  var notDisturbEndHour: Int { get }
    public  var notDisturbEndMinute: Int { get }

    /// Sleep reminder
    public  var sleepReminderEnable: Bool { get }
    public  var sleepReminderStartHour: Int { get }
    public  var sleepReminderStartMinute: Int { get }

    /// Schedule switch
    public  var scheduleEnable: Bool { get }

    /// Event reminder switch
    public  var eventReminderEable: Bool { get }

    /// Accident monitoring switch
    public  var accidentMonitorinEnable: Bool { get }

    /// Body temperature alarm switch
    public  var bodyTemperatureAlarm: Bool { get }
}
```

- Instruction
    - The API returns all the fields, but the configuration information supported by the specific device may be different.
    - Some attributes in the return value do not have specific type analysis, and

can be read in conjunction with the setting part.

- Examples of use

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

5.8 History summary

- Method

```
/// Get history summary information
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceHistorySummary(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Summary information
@objc public class YCDeviceHistorySummary : NSObject {

    /// Number of sleep records
    public  var sleepCount: Int { get }

    /// Total sleep time (minute)
    public  var sleepTime: Int { get }

    /// Number of heart rate records
```

```
    public  var heartRateCount: Int { get }

    /// Number of exercise records
    public  var sportCount: Int { get }

    /// Number of blood pressure records
    public  var bloodPressureCount: Int { get }

    /// Number of blood oxygen records
    public  var bloodOxygenCount: Int { get }

    /// Number of environmental temperature and humidity records
    public  var temperatureHumidityCount: Int { get }

    /// Number of temperature records
    public  var bodyTemperatureCount: Int { get }

    /// Number of ambient light records
    public  var ambientLightCount: Int { get }
}
```

- Instruction
    - Only the number of records will be returned, and all the information about the number of records will be stored in the YCDeviceHistorySummary type.
- Examples of use

```
YCProduct.queryDeviceHistorySummary { state, response in
    if state == .succeed,
    let info = response as? YCDeviceHistorySummary {
        print(info.sleepTime, info.sportCount, info.heartRateCount)
    }
}
```

5.9 Get current data

- This part of the content is the interface used by some customized equipment, most of the equipment does not need to be used.

5.9.1 Sport

- Method

```
/// Get current exercise data
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceCurrentExerciseInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Current sport
@objc public class YCDeviceCurrentExercise: NSObject {

    /// Steps (unit: step)
    public  var step: Int { get }

    /// Calories (unit: kcal)
    public  var calories: Int { get }

    /// Distance (unit: meter)
    public  var distance: Int { get }
}
```

- Instruction
    - Get the number of steps, distance, and calories of the current device.
    - Note: This API is only used for individual devices, to obtain the current exercise data, there is a general API, which will be listed in the following

chapter 7.8.

- Examples of use

```
YCProduct.queryDeviceCurrentExerciseInfo { state, response in
    if state == .succeed,
    let info = response as? YCDeviceCurrentExercise {
        print(info.step, info.calories, info.distance)
    }
}
```

5.9.2 Heart rate

- Method

```
///  Get current heart rate
/// - Parameters:
///   -  peripheral: Connected device
///   - completion: Result
public static func queryDeviceCurrentHeartRate(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Heart rate
@objcMembers public class YCDeviceCurrentHeartRate : NSObject {

    /// Is it measuring
    public  var isMeasuring: Bool { get }

    /// Heart rate value
    public  var heartRate: UInt8 { get }
}
```

- Instruction
    - If there is a measurement value, it returns the corresponding heart rate, otherwise it returns 0. If this function is not supported, it also returns 0.
- Examples of use

```
YCProduct.queryDeviceCurrentHeartRate { state, response in
    if state == YCProductState.succeed,
       let info = response as? YCDeviceCurrentHeartRate {
        print(info.isMeasuring, info.heartRate)
    }
}
```

5.9.3 Blood pressure

- Method

```
/// Get current blood pressure
/// - Parameters:
///   -  peripheral: Connected device
///   - completion: Result
public static func queryDeviceCurrentBloodPressure(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Blood pressure
@objcMembers public class YCDeviceCurrentBloodPressure : NSObject {

    /// Is it measuring
    public  var isMeasuring: Bool { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: UInt8 { get }

    /// Diastolic blood pressure
```

```
    public  var diastolicBloodPressure: UInt8 { get }
}
```

- Instruction
    - If there is a measurement value, it will return the corresponding blood pressure, otherwise it will return 0. If this function is not supported, it will also return 0.
- Examples of use

```
YCProduct.queryDeviceCurrentBloodPressure { state, response in
    if state == YCProductState.succeed,
       let info = response as? YCDeviceCurrentBloodPressure {
        print(info.isMeasuring, info.systolicBloodPressure,
info.diastolicBloodPressure)
    }
}
```

5.9.4 Blood oxygen

- Method

```
///  Get current blood oxygen
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceBloodOxygen(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Current blood oxygen
@objcMembers public class YCDeviceBloodOxygen : NSObject {

    /// Is it measuring
    public  var isMeasuring: Bool { get }
```

```
    /// Blood oxygen
    public  var bloodOxygen: UInt8 { get }
}
```

- Instruction
    - If there is a measurement value, it will return the corresponding blood oxygen, otherwise it will return 0. If this function is not supported, it will also return 0.
- Examples of use

```
YCProduct.queryDeviceBloodOxygen { state, response in
  if state == .succeed,
  let info = response as? YCDeviceBloodOxygen {
    print(info.bloodOxygen)
}
```

5.9.5 Ambient light

- Method

```
/// Get the current ambient light
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceAmbientLight(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Current ambient light
@objcMembers public class YCDeviceAmbientLight : NSObject {

    /// Is it measuring
    public  var isMeasuring: Bool { get }
```

```
    /// Ambient light
    public  var ambientLight: UInt16 { get }
}
```

- Instruction
    - If there is a measurement value, it will return the corresponding ambient light, otherwise it will return 0. If this function is not supported, it will also return 0.
- Examples of use

```
YCProduct.queryDeviceAmbientLight { state, response in
    if state == .succeed,
    let info = response as? YCDeviceAmbientLight {
        print(info.ambientLight)
    }
}
```

5.9.6 Temperature and humidity

- Method

```
/// Get the current environment temperature and humidity
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceTemperatureHumidity(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Temperature and humidity
@objcMembers public class YCDeviceTemperatureHumidity : NSObject {

    /// Is it measuring
```

```
    public  var isMeasuring: Bool { get }

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

    /// Humidity
    public  var humidity: Double { get }
}
```

- Instruction
    - If there is a measurement value, it will return the corresponding measurement value, otherwise it will return 0. If this function is not supported, it will also return 0.
- Examples of use

```
YCProduct.queryDeviceTemperatureHumidity { state, response in
    if state == .succeed,
    let info = response as? YCDeviceTemperatureHumidity {
        print(info.temperature, info.humidity)
    }
}
```

5.10 Sensor sampling information

- Method

```
/// Get sensor sampling information
/// - Parameters:
///   -  peripheral: Connected device
///   - dataType: Type of collected data
///   - completion: Result
public static func queryDeviceSensorSampleInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
```

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

/// 采集类型
@objc public enum YCDeviceDataCollectionType : UInt8 {
    case ppg
    case acceleration
    case ecg
    case temperatureHumidity
    case ambientLight
    case bodyTemperature
    case heartRate
}

/// Sensor sampling information
@objcMembers public class YCDeviceSensorSampleInfo : NSObject {

    /// Whether to open
    public  var isOn: Bool { get }

    /// Single acquisition time unit: seconds
    public  var acquisitionTime: UInt16 { get }

    /// Collection interval unit: minutes
    public  var acquisitionInterval: UInt16 { get }
}
```

- Instruction
    - According to different types, different sampling information is returned, and the results are stored in the YCDeviceSensorSampleInfo type. Pay attention to the unit of each parameter and.
- Examples of use

```
YCProduct.queryDeviceSensorSampleInfo(
  dataType: YCDeviceDataCollectionType.ppg) { state, response in
    if state == .succeed,
    let info = response as? YCDeviceSensorSampleInfo {
        print(info.isOn, info.acquisitionTime, info.acquisitionInterval)
    }
}
```

5.11 Working mode

- Method

```
/// Get the current system working mode
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceWorkMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Working mode
@objc public enum YCDeviceWorkModeType : UInt8 {
    case normal         // normal
    case care           // care
    case powerSaving    // power saving
    case custom         // custom
}
```

- Instruction
    - The device is in normal working mode by default
- Examples of use

```
YCProduct.queryDeviceWorkMode { state, response in
    if state == .succeed,
    let info = response as? YCDeviceWorkModeType {
        print(info)
    }
}
```

5.12 Upload reminder configuration information

- Method

```
/// Get upload reminder configuration information
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceUploadReminderInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Upload reminder
@objcMembers public class YCDeviceUploadReminderInfo {

    /// Whether to open
    public  var isOn: Bool { get }

    /// Storage threshold 0 ~ 100
    public  var threshold: UInt8 { get }
}
```

- Instruction
    - The percentage of data that the device has currently transmitted during the process of uploading data.

- Examples of use

```
YCProduct.queryDeviceUploadReminderInfo { state, response in
    if state == .succeed,
       let info = response as? YCDeviceUploadReminderInfo {
        print(info.isOn, info.threshold)
    }
}
```

5.13 Bracelet reminder setting information

- Method

```
/// Get bracelet reminder setting information
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: Reminder type
///   - completion: Result
public static func queryDeviceRemindSettingInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCDeviceRemindSettingType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Reminder setting type
@objc public enum YCDeviceRemindType : UInt8 {
    case deviceDisconnected   // Device disconnected
    case sportsCompliance     // Sports standard
}

/// Set status
@objc public enum YCDeviceReminderSettingState: UInt8 {
    case off      // Disenable
```

```
    case on       // Enable
}
```

- Instruction
    - According to different reminder types, get the status of the current setting. The status value type is YCDeviceReminderSettingState.
- Examples of use

```
YCProduct.queryDeviceRemindSettingInfo(dataType: .deviceDisconnected) {
state, response in
    if state == .succeed,
       let state = response as? YCDeviceReminderSettingState {
        print(state == .on)
    }
}
```

5.14 Screen display information

- Method

```
/// Get screen display information
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceScreenDisplayInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Screen display information
@objc public class YCDeviceScreenDisplayInfo : NSObject {

    /// Display level
```

```
    public  var brightnessLevel:
YCProductSDK.YCDeviceDisplayBrightnessLevel { get }

    /// Resting time
    public  var restScreenTime:
YCProductSDK.YCDeviceBreathScreenInterval { get }

    /// Language
    public  var language: YCProductSDK.YCDeviceLanguageType { get }

    /// Working mode
    public  var workmode: YCProductSDK.YCDeviceWorkModeType { get }
}
```

- Instruction
    - There are a large number of data types in the information displayed on the screen, and specific definitions have been given elsewhere in the document.
- Examples of use

```
YCProduct.queryDeviceScreenDisplayInfo { state, response in
    if state == .succeed,
       let info = response as? YCDeviceScreenDisplayInfo {
        print(info.brightnessLevel, info.restScreenTime, info.language,
info.workmode)
    }
}
```

6. Set up the device

6.1 Time

- Method

```
/// time setting
/// - Parameters:
///   - peripheral: Connected device
///   - year:  2000+
///   - month:  1 ~ 12
///   - day:  1 ~ 31
///   - hour:  0 ~ 23
///   - minute:  0 ~ 59
///   - second:  0 ~ 59
///   - weekDay: week
///   - completion: Result
public static func setDeviceTime(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    year: UInt16,
    month: UInt8,
    day: UInt8,
    hour: UInt8,
    minute: UInt8,
    second: UInt8,
    weekDay: YCWeekDay,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Set the day of the week
@objc public enum YCWeekDay : UInt8 {
    case monday
    case tuesday
    case wednesday
    case thursday
```

```
    case friday
    case saturday
    case sunday
}
```

- Instruction
    - The device only supports Gregorian calendar time, and other time types such as Buddhist calendar cannot be set.
    - The time is automatically set internally in the SDK, and generally there is no need to call this API.
- Examples of use

```
// 2021/12/6 14:38:59 Monday
YCProduct.setDeviceTime(
    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 Goal setting

- Method

```
/// Set step goal
/// - Parameters:
///   - peripheral: Connected device
```

```
    ///   - step: Step goal (steps)
    ///   - completion: Result
    public static func setDeviceStepGoal(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        step: UInt32,
        completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )

    /// Set calorie goal
    /// - Parameters:
    ///   - peripheral: Connected device
    ///   - calories: Calorie goal (kcal)
    ///   - completion: Result
    public static func setDeviceCaloriesGoal(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        calories: UInt32,
        completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )

    /// Set distance goal
    /// - Parameters:
    ///   - peripheral: Connected device
    ///   - calories: Distance target (m)
    ///   - completion: Result
    public static func setDeviceDistanceGoal(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        distance: UInt32,
        completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )

    /// Set sleep goals
    /// - Parameters:
    ///   - peripheral: Connected device
    ///   - hour: 0 ~ 23
    ///   - minute: 0 ~ 59
    ///   - completion: Result
```

```
public static func setDeviceSleepGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8,
    minute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

///  Exercise time goal
/// - Parameters:
///   - peripheral: Connected device
///   - hour: 0 ~ 23
///   - minute: 0 ~ 59
///   - completion: Result
public static func setDeviceSportTimeGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8,
    minute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Effective step goal
/// - Parameters:
///   - peripheral: Connected device
///   - effectiveSteps: Effective step goal (steps)
///   - completion: Result
public static func setDeviceEffectiveStepsGoal(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    effectiveSteps: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - If the parameter assignment is wrong, or the device does not support the target setting, it will return failure.
- Examples of use

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

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

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

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

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

```
}

// sport time goal
YCProduct.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 User information

- Method

```
/// User information settings
/// - Parameters:
///   - peripheral: Connected device
///   - height:  100 ~ 250cm
///   - weight:  30 ~ 200 kg
///   - gender:  YCDeviceGender
///   - age:  6 ~ 120
///   - completion: Result
```

```
public static func setDeviceUserInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    height: UInt8,
    weight: UInt8,
    gender: YCDeviceGender,
    age: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

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

- Instruction
    - Set only for the user's body, weight, gender, and age. Pay attention to the value range of each parameter.
- Examples of use

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

6.4 Unit settings

- Method

```
/// Unit settings
```

```swift
    /// - Parameters:
    ///   - peripheral: Connected device
    ///   - distance: Distance unit
    ///   - weight: Weight unit
    ///   - temperature: Temperature unit
    ///   - timeFormat: Time format: 12-hour clock/24-hour clock
    ///   - completion: Result
    public static func setDeviceUnit(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        distance: YCDeviceDistanceType = .km,
        weight: YCDeviceWeightType = .kg,
        temperature: YCDeviceTemperatureType = .celsius,
        timeFormat: YCDeviceTimeType = .hour24,
        completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )

    /// Distance unit
    @objc public enum YCDeviceDistanceType: UInt8 {
        case km
        case mile
    }

    /// Weight unit
    @objc public enum YCDeviceWeightType: UInt8 {
        case kg
        case lb
    }

    /// Temperature unit
    @objc public enum YCDeviceTemperatureType: UInt8 {
        case celsius
        case fahrenheit
    }

    /// Time format
    @objc public enum YCDeviceTimeType: UInt8 {
```

```
    case hour24
    case hour12
}
```

- Instruction
    - The unit setting is used to display the display value format
- Examples of use

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

6.5 Sedentary reminder

- Method

```
/// Sedentary reminder
/// - Parameters:
///   - peripheral: Connected device
///   - startHour1:  0 ~ 23
///   - startMinute1:  0 ~ 59
///   - endHour1:  0 ~ 23
///   - endMinute1:  0 ~ 59
///   - startHour2: 0 ~ 23
///   - startMinute2:  0 ~ 59
///   - endHour2: 0 ~ 23
///   - endMinute2:   0 ~ 59
///   - interval: 15 ~ 45 minutes
```

```
///   - repeat:   YCDeviceWeekRepeat
///   - completion: Result
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?) -> ())?
)

/// Repeat time of the week
@objc public enum YCDeviceWeekRepeat: UInt8 {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
    case sunday
    case enable
}
```

- Instruction
    - Sedentary reminder can only be set for two time periods
    - Pay attention to the value range of interval time, and the value of time is in 24-hour format.
    - The last value of `YCDeviceWeekRepeat` is the time enable switch. If `YCDeviceWeekRepeat.enable` is included, this parameter is valid, otherwise it is invalid.

- Examples of use

```
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 Anti-lost

- Method

```
/// Anti-lost settings
/// - Parameters:
///   - peripheral: Connected device
///   - antiLostType: Anti-lost type
```

```
///   - completion: Result
public static func setDeviceAntiLost(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    antiLostType: YCDeviceAntiLostType = .middleDistance,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
 )

/// Anti-lost type
@objc public enum YCDeviceAntiLostType: UInt8 {
    case off              // closure
    case closeRange       // Close range
    case middleDistance   // Middle distance
    case longDistance     // Long distance
}
```

- Instruction
    - Anti-lost means that the bracelet will vibrate when the connection signal between the device and the mobile phone becomes weak or disconnected. The last three values in the anti-lost type have the same effect.
- Examples of use

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

6.7 Notification reminder switch

- Method

```
/// Set the message reminder type
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - infoPushType: Notification reminder type
///   - completion: Result
public static func setDeviceInfoPush(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
isEnable: Bool, infoPushType: Set<YCDeviceInfoPushType>,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

///  Notification reminder type
@objc public enum YCDeviceInfoPushType: UInt16 {

    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
}
```

- Instruction
    - The notification reminder in the device is implemented based on the ANCS service of iOS, and the API can only set whether to display the corresponding type of message.
    - When the device is connected to the phone for the first time, iOS will pop up two interfaces, which are whether to allow pairing and whether to allow notifications to be displayed. All of them must be agreed, otherwise the notification reminder will not be available.
- Examples of use

```
// 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 Health monitoring

- Health monitoring means that the device will measure and save the corresponding data at a fixed time.
- In health monitoring, heart rate monitoring and temperature monitoring are universal functions, which can also meet most of the scenarios.
- Others such as blood pressure monitoring, blood oxygen monitoring, etc. are used for special application scenarios of some customized equipment.

6.8.1 Heart rate monitoring

- Method

```
///  心率监测
/// - Parameters:
///    - peripheral: Connected device
///    - isEnable: Whether to enable
///    - interval: Monitoring interval 1 ~ 60 minutes
///    - completion: Result
public static func setDeviceHeartRateMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool, interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the heart rate monitoring is turned on, the device will automatically measure the heart rate, blood pressure, blood oxygen, and respiration rate according to the automatic time. The shorter the time, the greater the power consumption.
- Examples of use

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

6.8.2 Temperature monitoring

- Method

```
/// Temperature monitoring
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - interval: Monitoring interval 1 ~ 60 minutes
///   - completion: Result
public static func setDeviceTemperatureMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the temperature monitoring is turned on, the device will automatically measure and record the temperature according to the set time.
    - It is recommended to keep the time interval consistent with the heart rate monitoring interval.
- Examples of use

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

6.8.3 Blood pressure monitoring

- Method

```
///  Blood pressure monitoring mode setting
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to enable
///   - interval: Monitoring interval 1 ~ 60 minutes
///   - completion: Result
public static func setDeviceBloodPressureMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the heart rate monitor is turned on, the device will monitor blood pressure at the same time. Only a few older versions of the device need to use this method, and in general, there is no need to call this method.
    - The monitoring time is best to be consistent with the heart rate monitoring
- Examples of use

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

6.8.5 Blood oxygen monitoring

- Method

```
/// Blood oxygen monitoring
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - interval: Monitoring interval 1 ~ 60 minutes
///   - completion: Result
public static func setDeviceBloodOxygenMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - None
- Examples of use

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

6.8.6 Ambient light monitoring

- Method

```
/// Ambient light monitoring
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - interval: Monitoring interval 1 ~ 60 minutes
///   - completion: Result
public static func setDeviceAmbientLightMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Individual customized bracelets are supported, which are the same as other monitoring modes, and will be automatically measured and saved after being turned on.
- Examples of use

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

6.8.7 Environmental temperature and humidity monitoring

- Method

```
/// Environmental temperature and humidity monitoring
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - interval: Monitoring interval 1 ~ 60 minutes
///   - completion: Result
public static func setDeviceTemperatureHumidityMonitoringMode(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    interval: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - None
- Examples of use

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

6.9 Health warning

6.9.1 Heart rate alarm

- Method

```
/// Heart rate alarm
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - maxHeartRate: Heart rate warning upper limit 100 ~ 240
///   - minHeartRate: Heart rate lower limit 30 ~ 60
///   - completion: Result
public static func setDeviceHeartRateAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    maxHeartRate: UInt8,
    minHeartRate: UInt8 ,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Heart rate alarm means that the device detects that the heart rate value is higher than the upper limit or lower than the lower limit will be turned on,

pay attention to the parameter value range.

- Examples of use

```
YCProduct.setDeviceHeartRateAlarm(isEnable: true,
                                  maxHeartRate: 100,
                                  minHeartRate: 50) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.9.2 Temperature alarm

- Method

```
/// Temperature alarm
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - highTemperatureIntegerValue: 36 ~ 100 Celsius
///   - highTemperatureDecimalValue: 0 ~ 9 Celsius
///   - lowTemperatureIntegerValue: -127 ~ 36 Celsius
///   - lowTemperatureDecimalValue: 0 ~ 9 Celsius
///   - completion: Result
public static func setDeviceTemperatureAlarm(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEnable: Bool,
  highTemperatureIntegerValue: UInt8,
  highTemperatureDecimalValue: UInt8,
  lowTemperatureIntegerValue: Int8,
  lowTemperatureDecimalValue: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
```

```
)
```

- Instruction
    - Like the heart rate alarm value, there are two values. Separate the integer and the decimal part. The value of the decimal part is 0-9. Like other methods, the temperature in the document is in degrees Celsius.
- Examples of use

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

6.9.3 Blood pressure warning

- Method

```
/// Blood pressure alarm setting
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - maximumSystolicBloodPressure: Maximum systolic blood pressure
///   - maximumDiastolicBloodPressure: Maximum diastolic blood pressure
///   - minimumSystolicBloodPressure: Minimum systolic blood pressure
///   - minimumDiastolicBloodPressure: Minimum diastolic blood pressure
```

```
///   - completion: Result
public static func setDeviceBloodPressureAlarm(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  isEnable: Bool,
  maximumSystolicBloodPressure: UInt8,
  maximumDiastolicBloodPressure: UInt8,
  minimumSystolicBloodPressure: UInt8,
  minimumDiastolicBloodPressure: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - When the device detects that the blood pressure is not within the set range, it will alarm and report the detection value at the same time. For the content of the detection value, please refer to Chapter 8.
- Examples of use

```
YCProduct.setDeviceBloodPressureAlarm(isEnable: true,
                                      maximumSystolicBloodPressure: 250,
                                      maximumDiastolicBloodPressure:
140,
                                      minimumSystolicBloodPressure: 160,
                                      minimumDiastolicBloodPressure: 90)
{ state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.9.4 Blood oxygen warning

- Method

```
/// Set blood oxygen alarm
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - minimum: Minimum blood oxygen level
///   - completion: Result
public static func setDeviceBloodOxygenAlarm(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool, minimum: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - When the device detects that the blood oxygen is lower than the set value, the device will alarm and report the detected value. For the content of the detection value, please refer to Chapter 8.
- Examples of use

```
YCProduct.setDeviceBloodOxygenAlarm(isEnable: true, minimum: 88) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.10 Do not disturb settings

- Method

```
///  Do not disturb settings
/// - Parameters:
///   - peripheral: Connected device
///   - isEable: Whether to enable
///   - startHour: 0 ~ 23
///   - startMinute: 0 ~ 59
///   - endHour: 0 ~ 23
///   - endMinute: 0 ~ 59
///   - completion: Result
public static func setDeviceNotDisturb(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEable: Bool,
  startHour: UInt8,
  startMinute: UInt8,
  endHour: UInt8,
  endMinute: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the bracelet enters the Do Not Disturb mode, all the reminder functions do not work.
- Examples of use

```
// 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 Reset

- Method

```
/// reset
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func setDeviceReset(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the device performs a factory reset, all data will be erased and the bracelet will be disconnected.
- Examples of use

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

6.12 Language setting

- Method

```
/// Language setting
/// - Parameters:
///   - peripheral: Connected device
///   - language: 语言
///   - completion: Result
public static func setDeviceLanguage(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    language: YCDeviceLanguageType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

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

    case english
    case chineseSimplified
    case russian
    case german
    case french
    case japanese
    case spanish
    case italian
    case portuguese
```

```
    case korean
    case poland
    case malay
    case chineseTradition
    case thai

    case vietnamese
    case hungarian
    case arabic
    case greek
    case malaysian
    case hebrew
    case finnish
    case czech
    case croatian

    case persian

    case ukrainian
    case turkish

    case danish
    case swedish
    case norwegian
    case romanian
}
```

- Instruction
    - The languages supported by each device are different, and the languages that are not supported may be displayed as English.
- Examples of use

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

6.13 Raise your wrist to turn on the screen switch

- Method

```
/// Raise your wrist to turn on the screen switch
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to enable
///   - completion: Result
public static func setDeviceWristBrightScreen(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - None
- Examples of use

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

6.14 Screen settings

6.14.1 Screen brightness

- Method

```
/// Screen brightness
/// - Parameters:
///   - peripheral: Connected device
///   - level: Brightness level
///   - completion: Result
public static func setDeviceDisplayBrightness(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    level: YCDeviceDisplayBrightnessLevel,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Brightness level
@objc public enum YCDeviceDisplayBrightnessLevel : UInt8 {
    case low
    case middle
    case high
    case automatic
    case lower
```

```
    case higher
}
```

- Instruction
    - The first 3 values of brightness level are universal, and the last three values are only supported by some customized devices.
- Examples of use

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

6.14.2 Resting time

- Method

```
/// Rest screen time setting
/// - Parameters:
///   - peripheral: Connected device
///   - interval: YCDeviceBreathScreenInterval
///   - completion: Result
public static func setDeviceBreathScreen(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    interval: YCDeviceBreathScreenInterval,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Interval
```

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

- Instruction
    - Note that the time interval is not a specific value, but `YCDeviceBreathScreenInterval`.
- Method

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

6.15 Skin tone settings

- Method

```
/// 肤色设置
/// - Parameters:
///   - peripheral: Connected device
///   - level: color
///   - completion: Result
public static func setDeviceSkinColor(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    level: YCDeviceSkinColorLevel,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

```
@objc public enum YCDeviceSkinColorLevel: UInt8 {
  case white
  case whiteYellow
  case yellow
  case brown
  case darkBrown
  case black
  case other
}
```

- Instruction
    - The skin color setting will affect the health data of the device and the ECG test. Generally, the darker the skin and the more hair users, the larger the value.
- Examples of use

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

6.16 Blood pressure level setting

- Method

```
/// Blood pressure range setting
/// - Parameters:
///   - peripheral: Connected device
///   - level: Blood pressure range
///   - completion: Result
public static func setDeviceBloodPressureRange(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
```

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

/// Blood pressure level
@objc public enum YCDeviceBloodPressureLevel: UInt8 {
    case low              // sbp < 90
    case normal           // sbp < 140
    case slightlyHigh     // spb < 160
    case moderatelyHigh   // spb < 180
    case severeHigh
}
```

- Instruction
    - When the measured photoelectric blood pressure has a large deviation from the actual blood pressure, the blood pressure level value of the device can be set for correction.
    - Note: If the device has a blood pressure calibration function, you do not need to use this function, and directly call blood pressure calibration (refer to 7.2).
- Examples of use

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

6.17 Bluetooth name setting

- Method

```
/// Set Bluetooth name
/// - Parameters:
///   - peripheral: Connected device
///   - name: New name
///   - completion: Result
public static func setDeviceBloodPressureRange(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    name: String,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The setting name is not allowed to exceed 12 bytes, and it is not recommended to use special characters.
    - This method is used for factory production, and it is not necessary to develop ordinary applications.
- Examples of use

```
YCProduct.setDeviceName(name: "YC2021") { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.18 Set the sensor sampling rate

- Method

```
/// Set the sensor sampling rate
/// - Parameters:
///   - peripheral: Connected device
///   - ppg: PPG sampling rate HZ
///   - ecg: ECG sampling rate HZ
///   - gSensor: G-Sensor sampling rate HZ
///   - tempeatureSensor: Temperature sensor sampling rate HZ
///   - completion: Result
public static func setDeviceSensorSamplingRate(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  ppg: UInt16,
  ecg: UInt16,
  gSensor: UInt16,
  tempeatureSensor: UInt16,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The sensor sampling rate is best to use the default, and do not modify it easily.
- Examples of use

```
YCProduct.setDeviceSensorSamplingRate(ppg: 250,
                                      ecg: 100,
                                      gSensor: 25,
                                      tempeatureSensor: 10) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.19 Theme settings

- Method

```
/// Theme settings
/// - Parameters:
///   - peripheral: Connected device
///   - index: Theme Index
///   - completion: Result
public static func setDeviceTheme(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    index: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The theme index starts from 0 and reaches the total number of themes-1。
- Examples of use

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

6.20 Reminder settings

6.20.1 Sleep reminder time

- Method

```
/// Sleep reminder time
/// - Parameters:
///   - peripheral:  Connected device
///   - hour: 0 ~ 23
///   - minute: 0 ~ 59
///   - repeat: YCDeviceWeekRepeat
///   - completion: Result
public static func setDeviceSleepReminder(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    hour: UInt8, minute: UInt8, repeat: Set<YCDeviceWeekRepeat>,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After the setting is successful, when the current time enters the reminder time, the device will vibrate and display the sleep reminder screen.
- Examples of use

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

6.20.2 Disconnect or exercise standard reminder settings

- Method

```
/// Device reminder type setting
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to enable
///   - remindType: YCDeviceRemindType
///   - completion: Result
public static func setDeviceReminderType(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  isEnable: Bool,
  remindType:YCDeviceRemindType,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - When the device reaches the set reminder conditions, it will vibrate.
- Examples of use

```
YCProduct.setDeviceReminderType(isEnable: true,
                                    remindType: .deviceDisconnected) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.20.3 Upload reminder

- Method

```
/// Upload reminder
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - threshold: Reminder value
///   - completion: Result
public static func setDeviceUploadReminder(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    threshold: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - It is used to indicate that when the storage data of the device reaches a specified ratio, an upload reminder is generated.
- Examples of use

```
YCProduct.setDeviceUploadReminder(isEnable: true, threshold: 50) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.21 Data collection configuration

- Method

```
/// Data collection configuration
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to enable
///   - dataType: Type of collected data
///   - acquisitionTime: Acquisition time unit: second, use 0 when
closed.
///   - acquisitionInterval: Collection interval unit: minute, use 0
when off.
///   - completion: Result
public static func setDeviceDataCollection(_ peripheral: CBPeripheral? =
YCProduct.shared.currentPeripheral,
                                           isEnable: Bool,
                                           dataType:
YCDeviceDataCollectionType,
                                           acquisitionTime: UInt8,
                                           acquisitionInterval: UInt8,
completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Data collection configuration in different working modes
/// - Parameters:
///   - peripheral: Connected device
///   - mode: work mode
///   - dataType: Type of collected data
///   - acquisitionTime: Acquisition time unit: second, use 0 when
closed.
///   - acquisitionInterval: Collection interval unit: minute, use 0
when off.
///   - completion: Result
public static func setDeviceWorkModeDataCollection(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
```

```
  mode: YCDeviceWorkModeType,
  dataType: YCDeviceDataCollectionType,
  acquisitionTime: UInt16,
  acquisitionInterval: UInt16,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Type of collected data
@objc public enum YCDeviceDataCollectionType: UInt8 {
    case ppg
    case acceleration
    case ecg
    case temperatureHumidity
    case ambientLight
    case bodyTemperature
    case heartRate
}
```

- Instruction
    - Not all devices support the types listed in the document
    - Pay attention to the unit and value of the last two parameters in method
    - There are two methods listed in the document. The second method is used for some customized devices, and some of the parameters of the two methods are different.
- Examples of use

```
YCProduct.setDeviceDataCollection(isEnable: true,
                                  dataType: .ppg,
                                  acquisitionTime: 90,
                                  acquisitionInterval: 60) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
```

```
    }
}

YCProduct.setDeviceWorkModeDataCollection(mode: .normal,
                                          dataType: .ppg, a
                                          cquisitionTime: 90,
                                          acquisitionInterval: 60) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.22 Working mode switch

- Method

```
/// Working mode
/// - Parameters:
///   - peripheral: Connected device
///   - mode: Working mode YCDeviceWorkModeType
///   - completion: Result
public static func setDeviceWorkMode(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

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

- Instruction
    - After the equipment enters different working modes, the monitoring time

or sampling frequency of the equipment will change.

- Examples of use

```
YCProduct.setDeviceWorkMode(mode: .normal) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.23 Accidental monitoring switch (reserved)

- Method

```
/// Accident monitoring switch
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - completion: Result
public static func setDeviceAccidentMonitoring(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

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

- Instruction
    - The interface is reserved, and there is no device support for the time being.
- Examples of use

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

6.24 Step counting time setting

- Method

```
/// Step counting time setting
/// - Parameters:
///   - peripheral: Connected device
///   - time: minutes
///   - completion: Result
public static func setDevicePedometerTime(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    time: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Set the step counting frequency of the device, the fixed value is 10, 5, 1, and the unit is minute.
- Examples of use

```
YCProduct.setDevicePedometerTime(time: 10) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.25 Bluetooth broadcast interval setting

- Method

```
/// Bluetooth broadcast transmission interval
/// - Parameters:
///   - peripheral: Connected device
///   - interval: 20 ~ 10240ms
///   - completion: Result
public static func setDeviceBroadcastInterval(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    interval: UInt16,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Note that the unit and value of the time interval in the parameter must be an integer multiple of 0.625ms。
- Examples of use

```
YCProduct.setDeviceBroadcastInterval(interval: 20) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.26 Bluetooth transmit power setting

- Method

```
/// Bluetooth transmit power setting
/// - Parameters:
///   - peripheral: Connected device
///   - power: DBM
///   - completion: Result
public static func setDeviceTransmitPower(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    power: Int8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The transmit power is best not to be a negative number, it needs to be greater than or equal to 0DBM.
- Examples of use

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

6.27 Exercise heart rate zone setting

- Method

```
/// Exercise heart rate zone setting
/// - Parameters:
///   - peripheral: Connected device
///   - zoneType: Exercise type
///   - minimumHeartRate: Maximum heart rate
///   - maximumHeartRate: Heart rate minimum
///   - completion: Result
public static func setDeviceExerciseHeartRateZone(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    zoneType: YCDeviceExerciseHeartRateType,
    minimumHeartRate: UInt8,
    maximumHeartRate: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Exercise type
@objc public enum YCDeviceExerciseHeartRateType: UInt8 {
    case retreat
    case casualwarmup
    case cardiorespiratoryStrengthening
    case reduceFatShape
```

```
    case sportsLimit
    case emptyState
}
```

- Instruction
    - According to different exercise types, set different heart rate ranges.
- Examples of use

```
YCProduct.setDeviceExerciseHeartRateZone(zoneType: .retreat,
                                         minimumHeartRate: 60,
                                         maximumHeartRate: 100) { state,
response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.28 Safety interface switch

- Method

```
/// Set to display the insurance interface
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - completion: Result
public static func setDeviceInsuranceInterfaceDisplay(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    isEnable: Bool,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - It only controls whether the insurance function interface of the device is displayed.
- Examples of use

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

6.29 Vibration time setting

- Method

```
/// Time required to set up the motor
/// - Parameters:
///   - peripheral: Connected device
///   - mode: Motor vibration type
///   - time: Duration in milliseconds
///   - completion: Result
public static func setDeviceMotorVibrationTime(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    mode: YCDeviceMotorVibrationType = .alarm, time: UInt32,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The vibration time of the motor can be modified. Note that the unit is milliseconds.
- Examples of use

```
YCProduct.setDeviceMotorVibrationTime(time: 2 * 1000) { state, response
in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.30 Alarm clock

6.30.1 Query alarm clock

- Method

```
/// Query alarm clock
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Alarm information
public static func queryDeviceAlarmInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Alarm information
@objc public class YCDeviceAlarmInfo : NSObject {

    /// Maximum number of alarms allowed by the device
    public  var limitCount: UInt8 { get }

    /// Alarm type
    public  var alarmType: YCProductSDK.YCDeviceAlarmType { get }

    /// Hour 0 ~ 23
    public  var hour: UInt8 { get }

    /// Minute 0 ~ 59
    public  var minute: UInt8 { get }

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

    /// Snooze time (minute)
    public  var snoozeTime: UInt8 { get }
}

/// Alarm type
```

```
@objc public enum YCDeviceAlarmType : UInt8 {
    case wakeUp
    case sleep
    case exercise
    case medicine
    case appointment
    case party
    case meeting
    case custom
}
```

- Instruction
    - After executing the query alarm, all the alarm information will be returned in the form of `YCDeviceAlarmInfo`.
- Examples of use

```
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 Add alarm clock

- Method

```
/// Add alarm clock
/// - Parameters:
///   - peripheral: Connected device
```

```
///   - alarmType: Alarm type
///   - hour: 0 ~ 23
///   - minute: 0 ~ 59
///   - repeat: Repeat time
///   - snoozeTime: Snooze time 0~59minutes
///   - completion: Result
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?) -> ())?
)
```

- Instruction
    - Only one alarm clock is allowed at the same time. Increase the number of alarm clocks that cannot exceed the device limit, generally 10.
- Examples of use

```
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 Modify alarm

- Method

```
/// Modify alarm
/// - Parameters:
///   - peripheral: Connected device
///   - oldHour: The original hour of the alarm clock
///   - oldMinute: The original minute of the alarm
///   - hour: Alarm clock new hour
///   - minute: Alarm new minute
///   - alarmType: Alarm type
///   - repeat: Repeat time
///   - snoozeTime: Snooze time
///   - completion: Result
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?) -> ())?
)
```

- Instruction
    - The alarm information can be modified according to the old time of the alarm clock
- Examples of use

```
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 Delete alarm

- Method

```
/// Delete alarm
/// - Parameters:
///   - peripheral: Connected device
///   - hour: 0 ~ 23
///   - minute: 0 ~ 59
///   - completion: Result
public static func deleteDeviceAlarm(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  hour: UInt8,
  minute: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The specified alarm can be deleted according to the time of the alarm

- Examples of use

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

6.31 Event

The basic handling and functions of events are similar to that of alarm clocks. The biggest difference between events and alarm clocks is that you can add a note name. Events are only supported by some customized devices.

6.31.1 Event enable

- Method

```
/// Event enable
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open
///   - completion: Result
public static func setDeviceEventEnable(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEnable: Bool,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Set whether the device event is valid, if it is false, all events are invalid.
- Examples of use

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

6.31.1 Add event

- Method

```
/// Add event
/// - Parameters:
///   - peripheral: Connected device
///   - name: <= 12 bytes, no more than 4 Chinese
///   - isEnable: Whether to enable
///   - hour:  0 ~ 23
///   - minute: 0 ~ 59
///   - interval: Repeat reminder interval
///   - repeat: Repeat week
///   - completion: Result
public static func addDeviceEvent(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    name: String,
    isEnable: Bool,
    hour: UInt8,
    minute: UInt8,
    interval: YCDeviceEventInterval,
```

```
    repeat: Set<YCDeviceWeekRepeat>,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

// Reminder interval
@objc public enum YCDeviceEventInterval: UInt8 {
    case none
    case ten
    case twenty
    case thirty
}
```

- Instruction
    - Please pay attention to the length of the event name. The time of the event interval uses enumerated values instead of specific values. Its unit is minutes.
    - After the setting is successful, the corresponding event id, 1 ~ 10 will be returned.
- Examples of use

```
YCProduct.addDeviceEvent(name: "party", isEnable: true, hour: 19,
minute: 50, interval: .ten, repeat: [.enable, .saturday]) { state,
respose in

                if state == .succeed,
                   let eventID = respose as? UInt8 {
                    print("success \(eventID)")
                } else {
                    print("fail")
                }
            }
```

6.32.2 Delete event

- Method

```
/// Delete event
/// - Parameters:
///   - peripheral: Connected device
///   - eventID: 1 ~ 10
///   - completion: Result
public static func deleteDeviceEvent(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    eventID: UInt8,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - It can be deleted directly by specifying the event number
- Examples of use

```
YCProduct.deleteDeviceEvent(eventID: 1) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.32.3 Modify event

- Method

```
/// Modify event
/// - Parameters:
```

```
    ///   - peripheral: Connected device
    ///   - name: Event name
    ///   - eventID: Event id
    ///   - isEnable: Whether to open
    ///   - hour: 0 ~ 23
    ///   - minute: 0 ~ 59
    ///   - interval: Time interval
    ///   - repeat: Repeat time
    ///   - completion: Result
    public static func modifyDeviceEvent(
        _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
        name: String,
        eventID: UInt8,
        isEnable: Bool,
        hour: UInt8,
        minute: UInt8,
        interval: YCDeviceEventInterval,
        repeat: Set<YCDeviceWeekRepeat>,
        completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )
```

- Instruction
    - The content of the event can be modified by specifying the id of the event
- Examples of use

```
YCProduct.modifyDeviceEvent(name: "sleep",
                            eventID: 1,
                            isEnable: true,
                            hour: 22,
                            minute: 30,
                            interval: .twenty,
                            repeat: [.enable, .monday, .tuesday,
.wednesday, .thursday, .friday]) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

6.32.4 Query event

- Method

```
/// Query event
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryDeviceEventnfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// event information
@objcMembers public class YCDeviceEventInfo : NSObject {

    /// Event ID
    public var eventID: UInt8 { get }
```

```
    /// Whether to open
    public var isEnable: Bool { get }

    /// Event hour 0 ~ 23
    public var hour: UInt8 { get }

    /// Event minutes 0 ~ 59
    public var minute: UInt8 { get }

    /// Event repeat
    public var `repeat`: Set<YCDeviceWeekRepeat> { get }

    /// Event interval
    public var interval: YCDeviceEventInterval { get }

    /// Event name
    public var name: String { get }
}
```

- Instruction
    - After executing the query event information, the result will be returned in the form of the `YCDeviceEventInfo` collection.
- Examples of use

```
YCProduct.queryDeviceEventnfo { state, response in

    if state == .succeed, let datas = response as? [YCDeviceEventInfo] {
        print("success")
        for item in datas {
            print(item.name, item.eventID,
                  item.hour, item.minute)
        }
    } else {

        print("fail")
    }
}
```

7. Controll device

7.1 Find device

- Method

```
/// Find device
/// - Parameters:
///   - peripheral: Connected device
///   - remindCount: 1 ~ 10
///   - remindInterval: 1 ~ 3
///   - completion: Result
public static func findDevice(_ peripheral: CBPeripheral? =
YCProduct.shared.currentPeripheral,
                              remindCount:UInt8 = 5,
                              remindInterval: UInt8 = 1,
                              completion: ((_ state: YCProductState, _
response: Any?) -> ())?
)
```

- Instruction
    - After calling Method, the bracelet will vibrate. Although the method provides reminder parameter settings, it is recommended to use the default values provided by the SDK.
- Examples of use

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

7.2 Blood pressure calibration

- Method

```
/// Blood pressure calibration
/// - Parameters:
///   - peripheral: Connected device
///   - systolicBloodPressure: Systolic blood pressure
///   - diastolicBloodPressure: Diastolic blood pressure
///   - completion: Result
public static func deviceBloodPressureCalibration(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  systolicBloodPressure: UInt8,
  diastolicBloodPressure: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Blood pressure calibration refers to the calibration of photoelectric blood pressure. After blood pressure calibration is performed, blood pressure level setting is not required, that is, only one of the two is used, and the priority of blood pressure calibration is high.
- Examples of use

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

7.3 Temperature calibration

- Method

```
/// Temperature calibration
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func deviceTemperatureCalibration(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Temperature calibration is used to calibrate temperature sensors in production equipment to make temperature measurement more accurate. Generally, this method is not required to develop applications.
- Examples of use

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

7.4 Axillary temperature measurement

After the device enters the axillary temperature measurement, the display will turn off and the interface will redisplay until the measurement result.

7.4.1 Start axillary temperature measurement

- Method

```
/// Axillary temperature
/// - Parameters:
/// - peripheral: Connected device
/// - isEnable: Whether to start axillary temperature measurement
/// - completion: Result
public static func deviceArmpitTemperatureMeasurement(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEnable: Bool,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - After starting the axillary temperature measurement, you need to actively obtain the measured temperature value. The recommended test time is 10 minutes. After the measurement is completed, you need to actively close the test.
- Examples of use

```
// Start measuring
YCProduct.deviceArmpitTemperatureMeasurement(isEnable: true) { state, _
in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// End measurement
YCProduct.deviceArmpitTemperatureMeasurement(isEnable: false) { state, _
in
```

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

7.4.2 Get temperature measurement

- Method

```
/// Get real-time temperature
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Measure temperature
public static func queryDeviceRealTimeTemperature(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - This method is only valid after the underarm temperature measurement is turned on, and the temperature will be returned in the type of Double.
- Examples of use

```
YCProduct.queryDeviceRealTimeTemperature { state, response in
    if state == .succeed,
       let temperature = response as? Double {
        print("success \(temperature)")
    } else {
        print("fail")
    }
}
```

7.5 Modify the color of the body temperature QR code

- Method

```
/// Modify the color of the body temperature QR code
/// - Parameters:
///   - peripheral: Connected device
///   - color: color
///   - completion: Result
public static func changeDeviceBodyTemperatureQRCodeColor(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  color: YCBodyTemperatureQRCodeColor,

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

@objc public enum YCBodyTemperatureQRCodeColor: UInt8 {
    case green
    case red
    case orange
}
```

- Instruction
    - For some individual customized devices, the color of the body temperature QR code can be modified
- Examples of use

```
YCProduct.changeDeviceBodyTemperatureQRCodeColor(color: .green) { state,
_ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.6 Weather data

- Method

```
/// Send weather
/// - Parameters:
///   - peripheral: Connected device
///   - isTomorrow: Today's weather or tomorrow's weather
///   - lowestTemperature: Minimum temperature Celsius
///   - highestTemperature: Maximum temperature Celsius
///   - realTimeTemperature: Current weather temperature Celsius
///   - weatherType: YCWeatherCodeType
///   - windDirection: Wind direction
///   - windPower: Wind force
///   - location: City
///   - moonType: Moon phase
///   - completion: Result
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: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Weather code type
@objc public enum YCWeatherCodeType: UInt8 {

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

    // ======== Individual equipment customization type
    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
}

/// Moon phase information
```

```
@objc public enum YCWeatherMoonType: UInt8 {

    case newMoon
    case waningMoon
    case theLastQuarterMoon
    case lowerConvexNoon
    case fullMoon
    case upperConvexMoon
    case firstQuarterMoon
    case crescentMoon
    case unknown
}
```

- Instruction
    - isTomorrow is used to determine whether to send today's weather or tomorrow's weather. Whether the bracelet supports tomorrow's weather setting can be judged based on function attributes or return values.
    - The temperature in the weather is all Celsius
    - Weather type `YCWeatherCodeType` Only the first 6 values are common, and the values listed later can only be used by some special customized equipment.
    - The remaining optional parameters can only be used on customized devices, and all other devices use nil.
- Examples of use

```
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 Shutdown reset restart

- Method

```
/// System operation
/// - Parameters:
///   - peripheral: Connected device
///   - mode: Shutdown reset restart
///   - completion: Result
public static func deviceSystemOperator(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  mode: YCDeviceSystemOperator,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// device mode
@objc public enum YCDeviceSystemOperator: UInt8 {
    case shutDown       = 1
    case transportation
    case resetRestart
}
```

- Instruction
    - Using different modes, the device will enter different states. Note that if it is set to transport mode, you must use charging to exit.
- Examples of use

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

7.8 Get real-time data

Obtaining the real-time data of the device is quite special. There are two methods for opening and receiving. In order to distinguish, the returned data type and the control type setting are not exactly the same. The Demo will list the process and steps of obtaining the data. For the content that is not listed, it may be It will appear in other usage scenarios. If the entire document does not appear, this function may not be supported or needed.

7.8.1 Enable real-time data acquisition

- Method

```
/// Real-time data upload
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to open or close
///   - dataType: YCRealTimeDataType
///   - interval: 1 ~ 240 seconds, it is recommended to use the default
value of 2 seconds
///   - completion: Result
public static func realTimeDataUplod(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  isEnable: Bool,
  dataType: YCRealTimeDataType,
```

```
  interval: UInt8 = 2,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// type of data
@objc public enum YCRealTimeDataType: UInt8 {
    case step
    case heartRate
    case bloodOxygen
    case bloodPressure
}
```

- Instruction
    - This method is used in fewer scenarios, mainly used internally by the SDK, and Method needs to be used in individual cases.
    - After using this method, you must actively receive data from the device, refer to 7.8.2.
    - After the whole process is used, it is recommended to close the method to avoid some inexplicable problems.
- Examples of use

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

7.8.2 Real-time data reported by the receiving device

- Method

```
/// Receive notifications of real-time data
public static let receivedRealTimeNotification: Notification.Name

/// Real-time data type received
@objc public enum YCReceivedRealTimeDataType: UInt8 {
    case step
    case heartRate
    case bloodOxygen
    case bloodPressure
    case ppg
    case ecg
    case realTimeMonitoringMode
}

/// Response to received real-time data
@objcMembers public class YCReceivedDeviceReportInfo: NSObject {

    /// Responding device received
    public  var device: CBPeripheral? { get }

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

/// Real-time steps
@objcMembers public class YCReceivedRealTimeStepInfo: NSObject {

    /// Steps (unit: step)
    public  var step: UInt16 { get }

    /// Distance (unit: meter)
    public  var distance: UInt16 { get }
```

```
    /// Calories (unit: kcal)
    public  var calories: UInt16 { get }
}

/// Real-time blood pressure data
@objcMembers public class YCReceivedRealTimeBloodPressureInfo: NSObject
{

    /// Heart rate value
    public  var heartRate: Int { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }

    /// Blood oxygen level
    public  var bloodOxygen: Int { get }

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

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

}

/// Real-time monitoring data
@objcMembers public class YCReceivedMonitoringModeInfo: NSObject {

    /// Start timestamp
    public  var startTimeStamp: Int { get }

    /// Heart rate value
```

```
    public  var heartRate: Int { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }

    /// Blood oxygen level
    public  var bloodOxygen: Int { get }

    /// Respiration rate value
    public  var respirationRate: Int { get }

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

    /// Real-time steps
    public  var realStep: Int { get }

    /// Real-time distance (unit: meter)
    public  var realDistance: UInt16 { get }

    /// Real-time calories (unit: kcal)
    public  var realCalories: UInt16 { get }

    /// Pattern steps
    public  var modeStep: Int { get }

    /// Mode distance (unit: meter)
    public  var modeDistance: UInt16 { get }

    /// Mode calories (unit: kcal)
    public  var modeCalories: UInt16 { get }
}
```

- Instruction
    - The notification of `receivedRealTimeNotification` needs to be monitored by the application and parsed according to different types. All the parsing processes are given here, and some of them may be used elsewhere in the document.
    - If there is no corresponding data playback, it means there is no corresponding data or the current scene cannot be obtained.
    - Note: If you finish using this Method, you must take the initiative to perform a shutdown operation, otherwise it may cause some inexplicable problems.
    - All type data will be returned in the `YCReceivedDeviceReportInfo` type, and be retrieved with the `YCReceivedRealTimeDataType` type as the key.
- Examples of use

```
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
            )
        }

        else if let response =
    info[YCReceivedRealTimeDataType.heartRate.string] as?
    YCReceivedDeviceReportInfo,
                let device = response.device,
                let heartRate = response.data as? UInt8 {

            print(device.name ?? "",
                  heartRate)
        }

       else if let response =
    info[YCReceivedRealTimeDataType.bloodOxygen.string] as?
    YCReceivedDeviceReportInfo,
                       let device = response.device,
                       let bloodOxygen = response.data as? UInt8 {

                    print(device.name ?? "",
                          bloodOxygen)
        }

        else if let response =
     info[YCReceivedRealTimeDataType.bloodPressure.string] as?
    YCReceivedDeviceReportInfo,
        let device = response.device,
        let bloodPressureInfo = response.data as?
    YCReceivedRealTimeBloodPressureInfo   {

            print(device.name ?? "",
                  bloodPressureInfo.systolicBloodPressure,
                  bloodPressureInfo.diastolicBloodPressure)
        }
```

```
}
```

7.9 Waveform upload control

- Method

```
/// Waveform upload control
/// - Parameters:
///   - peripheral: Connected device
///   - state: Whether to switch
///   - dataType: Wave type
///   - completion: Result
public static func waveDataUpload(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  state: YCWaveUploadState,
  dataType: YCWaveDataType,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Waveform type selection
@objc public enum YCWaveDataType: UInt8 {
    case ppg
    case ecg
    case multiAxisSensor
    case ambientLight
}

  @objc public enum YCWaveUploadState: UInt8 {
    case off = 0                    // Stop transfer
    case uploadWithOutSerialnumber  // No serial number transmission
    case uploadSerialnumber         // With 8-digit charge number
transmission
  }
```

- Instruction
    - This method is a method used internally by the SDK, and it may be required for special scenarios of some customized devices.
    - After the function is turned on, the device will report the waveform, and the SDK will send it as a notification after receiving it internally, and the notification needs to be monitored. For receiving data, please refer to the content in the previous section.
- Examples of use

```
// on
YCProduct.waveDataUpload(state: .uploadWithOutSerialnumber, dataType:
.ppg) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// off
YCProduct.waveDataUpload(state: .off, dataType: .ppg) { state, _ in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

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.ppg.string] as?
YCReceivedDeviceReportInfo,
     let device = response.device,
     let ppgData = response.data as? [Int32] {
        print(device.name ?? "", ppgData)

    } else if let response = info[YCReceivedRealTimeDataType.ecg.string]
as? YCReceivedDeviceReportInfo,
              let device = response.device,
              let ecgData = response.data as? [Int32] {
          print(device.name ?? "", ecgData)
      }
}
```

7.10 ECG measurement

ECG detection includes starting and stopping ECG, and obtaining the results of ECG. For drawing ECG waveforms, please refer to the demo case. The document will give examples of related Methods. The ECG detection is turned on and off by the App, and the recommended measurement time is 60 to 90 seconds. The measured data will be acquired during the test. Similarly, the device itself can also start ECG measurement, and the App can obtain relevant information.

7.10.1 Get electrode position

- Method

```
/// Get the position of the ECG potential
public static func queryDeviceElectrodePosition(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// ECG electrode position
@objc public enum YCDeviceElectrodePosition : UInt8 {
    case right              // Right
    case bottom             // Directly below the screen
    case bothSides          // Left side and right side
    case fullEncirclement   // all
}
```

- Instruction
    - Obtaining the electrode position of the device can prompt the user where to place the finger when measuring ECG.
- Examples of use

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

7.10.2 Set wearing position

- Method

```
/// Set wearing position
/// - Parameters:
///   - peripheral: Connected device
///   - wearingPosition: Wearing position
///   - completion: Result
public static func setDeviceWearingPosition(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  wearingPosition: YCProductSDK.YCDeviceWearingPositionType = .left,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// Wearing position
@objc public enum YCDeviceWearingPositionType : UInt8 {
    case left   // Left hand
    case right  // Right hand
}
```

- Instruction
    - When the wearing position of the device does not match the setting position, the generated waveform is opposite.
- Examples of use

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

  }
}
```

7.10.3 Start and end ECG measurement

- Method

```
/// Start ECG measurement
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func startECGMeasurement(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// Stop ECG measurement
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func stopECGMeasurement(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- Instruction
    - After starting to measure ECG, you need to place the finger of the other hand where it is with the electrode to make contact. After the measurement starts, the device will report the measured data.
- Examples of use

```
// Start ECG measurement
YCProduct.startECGMeasurement { state, _ in
    if state == .succeed {
    }
}

// Stop ECG measurement
YCProduct.stopECGMeasurement { state, _ in
    if state == .succeed {
    }
}
```

7.10.4 Receive measurement process data

- Method

```
/// Receive notifications of real-time data
public static let receivedRealTimeNotification: Notification.Name

/// Real-time blood pressure data
@objcMembers public class YCReceivedRealTimeBloodPressureInfo : NSObject
{

    /// Heart rate value
    public  var heartRate: Int { get }

    /// Systolic blood pressure
    public  var systolicBloodPressure: Int { get }

    /// Diastolic blood pressure
    public  var diastolicBloodPressure: Int { get }

    /// Blood oxygen level
```

```
    public  var bloodOxygen: Int { get }

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

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

- Instruction
    - When the device starts to measure ECG, it will report heart rate, blood pressure and ECG data, and some customized devices will return PPG data.
    - By monitoring the same notification, corresponding data can be obtained according to different data types. For specific data processing, refer to Demo.
- Examples of use

```
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
        }

        // Blood pressure data
```

```
        if let healthData =
 (info[YCReceivedRealTimeDataType.bloodPressure.string] as?
YCReceivedDeviceReportInfo)?.data as?
YCReceivedRealTimeBloodPressureInfo  {
            heartRate = healthData.heartRate
            systolicBloodPressure =
                healthData.systolicBloodPressure
            diastolicBloodPressure =
                healthData.diastolicBloodPressure
            if healthData.hrv > 0 {

            }
        }

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

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

7.10.5 Get ECG results

SDK provides a `YCECGManager` tool class to process the ECG measurement results and the parameters calculated in the measurement process.

7.10.5.1 initialization

- Method

```
/// Global object
public static let shared: YCProductSDK.YCECGManager

/// Algorithm calculation process callback
public func setupManagerInfo(
  rr:((_ rr: Float, _ heartRate: Int) -> ())?,
  hrv: ((_ hrv: Int) -> ())?
)
```

- Instruction
    - Some data will be generated during data processing, which will be reflected in the callback of `setupManagerInfo` method.
- Examples of use

```
let ecgManager = YCECGManager.shared

ecgManager.setupManagerInfo { rr, heartRate in
    //  Check the RR interval and calculate the heart rate
    print("=== Play sound")
} hrv: {  [weak self] hrv in
     // HRV
}
```

7.10.5.2 Receive ECG data

- Method

```
/// Process ECG data
public func processECGData(_ data: Int) -> Float
```

- Instruction
    - To pass the obtained ECG data into the `YCECGManager` tool class one by one.
- Examples of use

```
for data in datas {
      var ecgValue: Float = 0
      ecgValue = ecgManager.processECGData(Int(data))
      // ... other processing
}
```

7.10.5.3 Get ECG results

- Method

```
///  Get ECG results
/// - Parameters:
///   - peripheral: Connected device
///   - deviceHeartRate: Heart rate measured by the device
///   - devieHRV: HRV measured by the device
/// - completion: Result
public func getECGMeasurementResult(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    deviceHeartRate: Int?,
    devieHRV: Int?,
    completion: @escaping (_ result: YCECGMeasurementResult) -> ()
)

/// ECG measurement results
@objcMembers public class YCECGMeasurementResult : NSObject {

    /// Heart rate
    public var hearRate: Int { get }
```

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

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

/// ECG type
@objc public enum YCECGResultType : UInt {
    case failed                         // Measurement failed
    case atrialFibrillation             // Atrial fibrillation
    case earlyHeartbeat                 // Premature sexual intercourse
    case supraventricularHeartbeat      // Ventricular premature bleeding
    case atrialBradycardia              // Slow heart rate
    case atrialTachycardia              // Fast heart rate
    case atrialArrhythmi                // Sinus arrhythmia
    case normal                         // Normal ECG
}
```

- Instruction
    - There are two parameters of heart rate and HRV in the Method in the obtained result. If nil is used, the final heart rate and HRV value will use the value calculated by the algorithm. If the specified value is passed in, the final result will use the passed in value.
    - It is recommended to use the value measured by the device, if not, use the value of the algorithm.
    - Regarding the presentation of ECG measurement results, the reference text has been given in the table.

ECG Result (YCECGResultType)	textual representation
failed	This measurement signal is not good, which may caused by dry-skin. Please clean or moisten the skin and retest. Keep quiet during the test.
atrialFibrillation	QRS waveform was normal, normal P wave disappeared, F wave appeared, R-R interval was irregular.
earlyHeartbeat	QRS waveform was normal, variant P wave appeared ahead of time, P-R > 0.12 s, compensation interval was incomplete.
supraventricularHeartbeat	The qrs-t waveform was wide and deformed. There was no related P wave before the QRS waveform. The QRS duration was > 0.12 seconds. The direction of T wave was opposite to that of the main wave.
atrialBradycardia	The QRS waveform was normal and the R-R interval was too long.
atrialTachycardia	The QRS waveform was normal and the R-R interval was short.
atrialArrhythmi	The QRS waveform was normal, and the R-R interval changed too much.
normal	The amplitude of QRS waveform was normal, P-R interval was normal, ST-T was not changed, and Q-T interval was normal.

- Examples of use

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

7.10.5.4 Get body and other emotional index (reserved)

- Method

```
/// Get body index
public func getPhysicalIndexParameters() -> YCBodyIndexResult

/// Physical outcome parameters
@objcMembers public class YCBodyIndexResult: NSObject {

    /// Available or not
    public  var isAvailable: Bool = false

    /// Load index
    public  var heavyLoad: Float = 0

    /// Stress index
    public  var pressure: Float = 0

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

    /// Body index
    public  var body: Float = 0
}
```

- Instruction
    - This method may be able to determine whether it is available through the attribute isAvailable in the return value。
- Examples of use

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

7.10.6 About drawing ECG waveforms

1. During ECG measurement, if you need to draw graphics, you still need to read the code in the Demo first, and you need to understand the basic drawing knowledge in iOS.
2. The waveform given in the Demo is a normal and standardized drawing method. If it is of other types, you need to zoom in or out on this basis.

7.10.7 Acquire ECG and PPG data for device startup measurement

1. If it is the ECG measurement initiated by the device, only the measured ECG or PPG data can be obtained, but no other data.
2. For related operations to obtain data, please refer to Chapter 10 Historical Data Collection.

7.11 Health data measurement

App start measurement is completed by two parts: start test and receive measurement data.

7.11.1 Turn measurement on and off

- Method

```
/// Health data measurement
/// - Parameters:
///   - peripheral: Connected device
///   - measureType: YCAppControlHealthDataMeasureType
///   - dataType: YCAppControlMeasureHealthDataType
///   - completion: Result
public static func controlMeasureHealthData(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  measureType: YCAppControlHealthDataMeasureType,
  dataType: YCAppControlMeasureHealthDataType,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Measurement method
@objc public enum YCAppControlHealthDataMeasureType: UInt8 {
    case off          // Close measurement
    case single       // Single measurement
    case monitor      // Reserved parameters
}

/// type of data
@objc public enum YCAppControlMeasureHealthDataType: UInt8 {
    case heartRate
    case bloodPressure
    case bloodOxygen
    case respirationRate
    case bodyTemperature
```

```
    case unknow
}
```

- Instruction
    - None
- Examples of use

```
// Start measurement
YCProduct.controlMeasureHealthData(measureType: .single, dataType:
.heartRate) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}

// End measurement
YCProduct.controlMeasureHealthData(measureType: .off, dataType:
.heartRate) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.11.2 Receive measured value

After the device starts the measurement, the value during the measurement will be actively reported. For the analysis of the received data, please view the code demonstration in 7.8.2.

7.11. 3 Receive measurement status

- Method

```
/// Device control notification
public static let deviceControlNotification: Notification.Name

/// Single measurement result
@objcMembers public class YCDeviceControlMeasureHealthDataResultInfo:
NSObject {
    public  var state: YCAppControlMeasureHealthDataResult { get }
    public  var dataTyp: YCAppControlMeasureHealthDataType { get }
}

//// Measurement result
@objc public enum YCAppControlMeasureHealthDataResult: UInt8 {
    case exit         // User exits measurement
    case success      // Successful measurement
    case fail         // Measurement failed
}
```

- Instruction
    - After the measurement starts, the user exits the measurement interface or the measurement ends, the device will report the status, and the SDK will send the status.
- Examples of use

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

```
@objc private func measureDataStateChanged(_ ntf: Notification) {
    guard let info = ntf.userInfo,
          let result =
((info[YCDeviceControlType.healthDataMeasurementResult.string]) as?
YCReceivedDeviceReportInfo)?.data as?
YCDeviceControlMeasureHealthDataResultInfo  else {
              return
          }
    print(result.state, result.dataType)
}
```

7.12 Sport

```
/// Sport type definition
@objc public enum YCDeviceSportType: UInt8 {

    case none
    case run
    case swimming
    case riding
    case fitness

    case ropeskipping
    case playball
    case walk
    case badminton

    case football
    case mountaineering
    case pingPang
```

```
    case indoorRunning
    case outdoorRunning
    case outdoorWalking
    case indoorWalking

    case indoorRiding
    case stepper
    case rowingMachine
    case realTimeMonitoring
    case situps
    case jumping
    case weightTraining
    case yoga
}

/// Sport state
@objc public enum YCDeviceSportState: UInt8 {
    case stop
    case start
}
```

7.3.1 Sport control

- Method

```
/// Control the device to enter the sport mode
/// - Parameters:
///   - peripheral: Connected device
///   - state: YCDeviceSportState
///   - sportType: YCDeviceSportType
///   - completion: Result
public static func controlSport(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  state: YCDeviceSportState,
  sportType: YCDeviceSportType,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Currently, all devices only support opening and closing.
- Examples of use

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

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

```
}
```

7.3.2 Receipt of exercise data

As the device motion data reporting method has been optimized, the method of receiving data has also changed accordingly. Subsequent products gradually migrate to the new method. The new method requires two sets of data to be received, and the old method only has one.

7.3.2.1 Determine the movement mode of the device to transmit data

- Method

```
// Refer to section 5.1
public  var isSupportSyncRealSportData: Bool = false
```

- Instruction
    - Judge whether it is the new way or the old way through the attributes of the device.
- Examples of use

```
if peripheral?.supportItems.isSupportSyncRealSportData {
    // New way
} else {
    // Old way
}
```

7.3.2.2 Old way

- Method

```
/// Receive notifications of real-time data
public static let receivedRealTimeNotification: Notification.Name
```

- Instruction
    - The device will return heart rate, steps, distance, calories, but not time.
    - Note that the number of steps, distance, and calories are returned to the cumulative result, so each time you get these three values, you should subtract the initial value obtained for the first time after entering the exercise.
    - In this way, it is impossible to obtain whether the device has exited the sports mode.
- Examples of use

```
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
            )
        }
        else if let response =
    info[YCReceivedRealTimeDataType.heartRate.string] as?
    YCReceivedDeviceReportInfo,
                let device = response.device,
                let heartRate = response.data as? UInt8 {
            print(device.name ?? "",
                heartRate)
        }
    }
```

7.3.2.3 New way

- Method

```
/// Receive notifications of real-time data
public static let receivedRealTimeNotification: Notification.Name

/// Sports status change notification
public let deviceControlNotification: Notification.Name

/// Sports status information
@objcMembers public class YCDeviceControlSportModeControlInfo: NSObject
{
    public  var state: YCDeviceSportState {get}
    public  var sportType: YCDeviceSportType {get}
}
```

- Instruction
    - In addition to obtaining the exercise data of the device, the new method may also obtain the status of whether the exercise is exited or not.

- Examples of use

```
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 Sports history data

- Most devices will not record motion-related data after starting up and running, and need to be processed by the App itself.
- For some customized devices, exercise data will be saved. If you want to obtain this part of information, please refer to 4.3.12.

7.13 Photograph

There are two ways to start taking pictures. One is to start the device and enter the camera mode, and the other is to start the App to enter the camera mode. After entering the photo mode, there are two ways to perform the photo action, one is to tap the App to take a photo, and the other is to tap the device to take a photo. The real photo is done on the mobile phone. If the device clicks to take the photo, you need to reply to the device after the photo is completed to see if the photo is successful.

7.13.1 App to turn on and off the camera mode

- Method

```
/// Mobile phone control to enter and exit photo mode
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Activate or deactivate photo mode
///   - completion: Result
public static func takephotoByPhone(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEnable: Bool,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - In this method, the mobile phone controls the device to enter or exit the camera mode, and all operations are performed on the mobile phone.

7.13.2 The device starts and exits to take pictures

- Method

```
/// Device control notification
public static let deviceControlNotification: Notification.Name

/// Photo mode status
@objc public enum YCDeviceControlPhotoState: UInt8 {
    case exit     // Exit photo
    case enter    // Enter to take a photo
    case photo    // Photograph
}
```

- Instruction
    - When the received status is a photo, the photo is still operated on the mobile phone, and the device is only responsible for reporting the current status.
- Examples of use

```
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 Cross operation

Once the interaction logic of taking pictures crosses, for example, the mobile phone starts and the device exits, or the device starts and the mobile phone exits. Pay attention to the change of state, and then call the corresponding interface.

7.14 Health parameters, warning information

- Method

```
/// Send health parameters
/// - Parameters:
///   - peripheral: Connected device
///   - warningState: YCHealthParametersState
///   - healthState: YCHealthState
///   - healthIndex: 0 ~ 120
///   - othersWarningState: Is the warning for relatives and friends effective?
///   - completion: Result
public static func sendHealthParameters(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    warningState: YCHealthParametersState,
    healthState: YCHealthState,
    healthIndex: UInt8,
    othersWarningState: YCHealthParametersState,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

```
/// Warning status
@objc public enum YCHealthParametersState: UInt8 {
    case off
    case effect
    case invalid
}

/// health status
@objc public enum YCHealthState: UInt8 {
    case unknow
    case excellent
    case good
    case general
    case poor
    case sick
    case invalid
}
```

- Instruction
    - After sending the warning message, the device will vibrate after reaching the specified conditions.
- Examples of use

```
YCProduct.sendHealthParameters(warningState: .off, healthState: .good, healthIndex: 100, othersWarningState: .off) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.15 News from relatives and friends

- Method

```
/// Show relatives and friends
/// - Parameters:
///   - peripheral: Connected device
///   - index: Emoticon number 0 ~ 4
///   - hour: Send time 0 ~ 23
///   - minute: Send time 0 ~ 59
///   - name: Name of relatives and friends
///   - completion: Result
public static func deviceShowFriendMessage(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  index: UInt8,
  hour: UInt8,
  minute: UInt8,
  name: String,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - This method is only supported by a customized device
- Examples of use

```
YCProduct.deviceShowFriendMessage(index: 1,
                                  hour: 10,
                                  minute: 23,
                                  name: "俺是大宝二") { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.16 Data write-back

This part is only used for the unique functions of some customized devices, and other devices can be ignored.

7.16.1 Health value write-back

- Method

```
/// Health value write-back
/// - Parameters:
///   - peripheral: Connected device
///   - healthValue: Health value
///   - statusDescription: Description
///   - completion: Result
public static func deviceHealthValueWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  healthValue: UInt8,
  statusDescription: String,

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

- Instruction
    - None
- Examples of use

```
YCProduct.deviceHealthValueWriteBack(healthValue: 50, statusDescription:
"Good") { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.16.2 Sleep data write back

- Method

```
/// Sleep data write back
/// - Parameters:
```

```
    ///   - peripheral: Connected device
    ///   - deepSleepHour: 0 ~ 23
    ///   - deepSleepMinute:   0 ~ 23
    ///   - lightSleepHour: 0 ~ 23
    ///   - lightSleepMinute: 0 ~ 59
    ///   - totalSleepHour: 0 ~ 23
    ///   - totalSleepMinute: 0 ~ 59
    ///   - completion: Result
    public static func deviceSleepDataWriteBack(
      _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

      deepSleepHour: UInt8,
      deepSleepMinute: UInt8,
      lightSleepHour: UInt8,
      lightSleepMinute: UInt8,
      totalSleepHour: UInt8,
      totalSleepMinute: UInt8,
      completion: ((_ state: YCProductState, _ response: Any?) -> ())?
    )
```

- Instruction
    - None
- Examples of use

```
YCProduct.deviceSleepDataWriteBack(
  deepSleepHour: 2,
  deepSleepMinute: 30,
  lightSleepHour: 4,
  lightSleepMinute: 0,
  totalSleepHour: 8,
  totalSleepMinute: 0) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.16.3 Personal information write back

- Method

```
/// Personal information write back
/// - Parameters:
///   - peripheral: Connected device
///   - infoType: User information type
///   - information: Description
///   - completion: Result
public static func devicePersonalInfoWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  infoType: YCPersonalInfoType,

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

```
/// Personal type
@objc public enum YCPersonalInfoType: UInt8 {
    case insurance
    case vip
}
```

- Instruction
    - None
- Examples of use

```
/// Personal data write-back
/// - Parameters:
///   - peripheral: Connected device
///   - infoType: User Info
///   - information: Description
///   - completion: Result
public static func devicePersonalInfoWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  infoType: YCPersonalInfoType,

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

7.16.4 Upgrade progress write back

- Method

```
/// Upgrade reminder
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: Whether to enable reminder
///   - percentage: Current progress 0 ~ 100
///   - completion: Result
public static func deviceUpgradeReminderWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  isEnable: Bool,
  percentage: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - None
- Examples of use

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

7.16.5 Sport data write back

- Method

```
/// Sport data write back
```

```swift
/// - Parameters:
///   - peripheral: Connected device
///   - step: Exercise steps
///   - state: Sport state
///   - completion: Result
public static func deviceSportDataWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

  step: UInt32,
  state: YCDeviceExerciseHeartRateType,

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

/// Exercise state type
@objc public enum YCDeviceExerciseHeartRateType: UInt8 {
    case retreat
    case casualwarmup
    case cardiorespiratoryStrengthening
    case reduceFatShape
    case sportsLimit
    case emptyState
}
```

- Instruction
    - None
- Examples of use

```
YCProduct.deviceSportDataWriteBack(step: 10000, state: .reduceFatShape)
{ state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.16.6 Calculate heart rate synchronization

- Method

```
/// Calculate the heart rate and send it to the device
/// - Parameters:
///   - peripheral: Connected device
///   - heartRate:  Calculate heart rate
///   - completion: Result
public static func sendCaclulateHeartRate(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  heartRate: UInt8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - None
- Examples of use

```
YCProduct.sendCaclulateHeartRate(heartRate: 78) { state, response in
  if state == .succeed {
      print("success")
  } else {
      print("fail")
  }
}
```

7.16.7 Measurement data write-back

- Method

```
/// Health data write back
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: Measurement data type
///   - values: Measured value collection
///   - completion: Result
public static func deviceMeasurementDataWriteBack(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  dataType: YCMeasurementDataWritebackType,
  values: [UInt8],
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Measurement data type
@objc public enum YCMeasurementDataWritebackType: UInt8 {

    case heartRate          = 0
    case bloodPressure
    case bloodOxygen
    case respirationRate
    case hrv
```

```
}
```

- Instruction
    - Note that except for the blood pressure value, the others are all one value. For blood pressure, the systolic blood pressure is written in the front and the diastolic blood pressure is written in the back. No matter how many values it is, it must be passed in the form of an array.
- Examples of use

```
YCProduct.deviceMeasurementDataWriteBack(
  dataType: .bloodPressure,
  values: [110, 220]) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.17 Sensor data storage switch control

- Method

```
/// Sensor data storage switch control
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: Sensor type
///   - isEable: Whether to open
///   - completion: Result
public static func deviceSenserSaveData(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  dataType: YCDeviceSenserSaveDataType,
  isEable: Bool,
```

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

/// Sensor type
@objc public enum YCDeviceSenserSaveDataType: UInt8 {
    case ppg    = 0
    case acceleration
    case ecg
    case temperatureHumidity
    case ambientLight
    case bodyTemperature
}
```

- Instruction
    - Whether the control equipment records relevant data is only valid for a special equipment.
- Examples of use

```
YCProduct.deviceSenserSaveData(dataType: .temperatureHumidity,
                               isEable: false) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.18 Send mobile phone model

- Method

```
/// Send mobile phone model
/// - Parameters:
///   - peripheral: Connected device
///   - mode: Phone model such as iPhone 13
///   - completion: Result
public static func sendPhoneModeInfo(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,

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

- Instruction
    - None
- Examples of use

```
YCProduct.sendPhoneModeInfo(mode: "iPhone13 Pro Max") { state, response
in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.19 Send warning message

- Method

```
/// Warning information
/// - Parameters:
///   - peripheral: Connected device
```

```
///   - infoType: Information type
///   - message: information
///   - completion: Result
public static func sendWarningInformation(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    infoType: YCWarningInformationType,
    message: String?,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Warning message type
@objc public enum YCWarningInformationType: UInt8 {
    case warnSelf       // Alert yourself
    case warnOthers     // Alert others
    case highRisk       // High risk of exercise
    case nonHighRisk    // Non-high risk
}
```

- Instruction
    - Only when infoType is `YCWarningInformationType.warnOthers`, message is the name of the person who warned. For other types, this parameter is invalid, and it is always nil.
    - This method is only valid for special customized equipment.
- Examples of use

```
YCProduct.sendWarningInformation(infoType: .warnSelf, message: nil) {
state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.20 Message push

- Method

```
/// send Message
/// - Parameters:
///   - peripheral: Connected device
///   - index: Information label 0 ~ 6
///   - content: information
///   - completion: Result
public static func sendShowMessage(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    index: UInt8,
    content: String?,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - The sent message has a fixed value, only when the index is 6, the content will pass the value, otherwise it will pass nil.

index	Display content
0	A new weekly report is generated, please check it on the APP.
1	A new monthly report is generated, please check it on the APP.
2	If you receive information from relatives and friends, please check it on the APP.
3	It’s been a long time since I measured it. Let’s measure it.
4	You have successfully booked a consultation.
5	Your appointment will start in one hour.
6	content

- Examples of use

```
YCProduct.sendShowMessage(index: 1, content: nil) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.21 Ambient temperature and humidity calibration (reserved)

- Method

```
/// Temperature and humidity calibration
/// - Parameters:
///   - peripheral: Connected device
///   - temperaturerInteger: Temperature integer
///   - temperaturerDecimal: Temperature decimal
///   - humidityInteger: Humidity integer
///   - humidityDecimal: Humidity decimal
///   - completion: Result
public static func deviceTemperatureHumidityCalibration(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  temperaturerInteger: Int8,
  temperaturerDecimal: Int8,
  humidityInteger: Int8,
  humidityDecimal: Int8,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction

        - Calibrate related sensors to make the measurement more accurate.
    - Examples of use

```
YCProduct.deviceTemperatureHumidityCalibration(
  temperaturerInteger: 36,
  temperaturerDecimal: 5,
  humidityInteger: 43,
  humidityDecimal: 4) { state, response in
    if state == .succeed {
        print("success")
    } else {
        print("fail")
    }
}
```

7.22 Address book

The address book function only sends the user name and number to the device for storage and storage. The maximum number that the device can store is 30. During the entire process of transmitting the address book, opening and exiting synchronization only need to be executed once, while sending communication data requires repeated execution, because only one record can be sent at a time.

7.22.1 Enter sync address book

- Method

```
/// Turn on address book synchronization
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func startSendAddressBook(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Only after the device is set to start synchronization, can it enter the real synchronization data. It only needs to be executed once.
- Examples of use

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

7.22.2 Send address book data

- Method

```
/// Synchronize address book details data
/// - Parameters:
///   - peripheral: Connected device
///   - phone: Phone number, no more than 20 characters
///   - name: Username, no more than 8 Chinese
///   - completion: Result
public static func sendAddressBook(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  phone: String,
  name: String,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - Send specific address book information to the device, pay attention to the length of the value.
- Examples of use

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

7.22.3 Exit sync address book

- Method

```
/// Exit sync address book
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func stopSendAddressBook(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - To end the synchronization of address book information, it only needs to be executed once.
- Examples of use

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

8. Receive device response

This part refers to that the user operates the device or the device monitors some kind of information, and the device will report the operation information at the same time. The SDK will uniformly monitor the response of the device and process it according to different types. The SDK will send the operating status of the device in the form of a notification. The application needs to monitor and parse according to different types. In addition, some content has already been listed by other parties and will no longer appear here. If the

corresponding data is not received, it may be that the device does not support this function.

```
// Response type
@objc public enum YCDeviceControlType: UInt8 {

    case findPhone                      // Find cellphone
    case photo                          // Photograph
    case sos                            // SOS
    case allowConnection                // Whether to allow connection
    case sportMode                      // Switch sport
    case reset                          // Reset
    case stopRealTimeECGMeasurement     // Stop ECG measurement
    case sportModeControl               // Switch sport mode
    case switchWatchFace                // Switch watch face
    case healthDataMeasurementResult    // Start device test
    case reportWarningValue             // Warning value
}

/// Device control notification
public static let deviceControlNotification: Notification.Name
```

Since this part is relatively uniform, the case demonstrations are all written in one demonstration.

```
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
  }
```

```
  // Find cellphone
  if let response = info[YCDeviceControlType.findPhone.string] as?
YCReceivedDeviceReportInfo,
     let device = response.device,
     let state = response.data as? YCDeviceControlState {
      print(device.name ?? "",
            state == .stop
      )
  }

  // sos
  if let response = info[YCDeviceControlType.sos.string] as?
YCReceivedDeviceReportInfo,
     let device = response.device {
      print(device.name ?? "",
            "sos"
      )
  }

  // Whether to allow connection
  if let response = info[YCDeviceControlType.allowConnection.string] as?
YCReceivedDeviceReportInfo,
     let device = response.device,
     let state = response.data as? YCDeviceControlAllowConnectionState {
      print(device.name ?? "",
            state == .agree
      )
  }

  // reset
  if let response = info[YCDeviceControlType.reset.string] as?
YCReceivedDeviceReportInfo,
     let device = response.device {
      print(device.name ?? "",
            "reset"
```

```
        )
    }

    // Warning value
    if let response = info[YCDeviceControlType.reportWarningValue.string]
as? YCReceivedDeviceReportInfo,
       let device = response.device,
       let value = response.data as? YCDeviceControlReportWarningValueInfo
{
        print(device.name ?? "",
              value
        )
    }
}
```

8.1 Find cellphone

The device will report the status when it starts searching for the mobile phone or stops searching for the mobile phone.

```
// Find the status of the phone
@objc public enum YCDeviceControlState: UInt8 {
    case stop
    case start
}
```

8.2 SOS

When the device enters the SOS mode, it will report the SOS status, and there is no specific value.

8.3 Whether to allow connection

```
@objc public enum YCDeviceControlAllowConnectionState: UInt8 {
    case agree
    case refuse
}
```

8.4 Receive monitoring and warning values

```
@objcMembers public class YCDeviceControlReportWarningValueInfo:
NSObject {

    /// Warning type
    public  var dataType: YCAppControlMeasureHealthDataType {get}

    /// Warning value
    public  var values: [Int] {get}
}

// If the measurement type is blood pressure, the first element in
values is systolic blood pressure, and the second is diastolic blood
pressure.
```

9. Watch face download

Watch face download includes querying the watch face information in the device, App switching watch faces, deleting watch faces, device operation watch faces, and App custom watch faces.

9.1 Query device dial information

- Method

```
/// Query device dial information
/// - Parameters:
///   - peripheral: Connected device
///   - completion: Result
public static func queryWatchFaceInfo(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)


/// Breakpoint information of the watch face
@objcMembers public class YCWatchFaceBreakCountInfo : NSObject {

    /// Dial data
    public  var dials: [YCProductSDK.YCWatchFaceInfo] { get }

    /// Maximum number supported
    public  var limitCount: Int { get }

    /// Locally stored quantity
    public  var localCount: Int { get }
}
```

```
/// Dial information
@objcMembers public class YCWatchFaceInfo : NSObject {

    /// Dial id
    public  var dialID: UInt32

    /// Dial breakpoint value
    public  var blockCount: UInt16

    /// Support delete
    public  var isSupportDelete: Bool { get }

    /// Dial version
    public  var version: UInt16 { get }

    /// Whether it is a custom watch face
    public  var isCustomDial: Bool { get }

    /// Whether it is the current display dial
    public  var isShowing: Bool { get }
}
```

- Instruction
    - If the query is successful, the returned result is `[YCWatchFaceBreakCountInfo]` type
- Examples of use

```
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 delete watch face

- Method

```
/// Remove watch face
/// - Parameters:
///   - peripheral: Connected device
///   - dialID: Dial ID
///   - completion: Result
public static func deleteWatchFace(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  dialID: UInt32,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- Instruction
    - You can delete the watch face as long as you specify the watch face ID to be deleted.
- Examples of use

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

9.3 App switch watch face

- Method

```
/// App switch watch face
/// - Parameters:
///   - peripheral: Connected device
///   - dialID: Dial ID
///   - completion: Result
public static func changeWatchFace(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  dialID: UInt32,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)
```

- Instruction
    - The parameters for switching the dial and deleting the dial are the same
- Examples of use

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

9.4 Device switch or delete watch face

- Method

```
/// Device control notification
public static let deviceControlNotification: Notification.Name
```

- Instruction
    - If the device deletes or switches the watch face, it will take the initiative to report the finally displayed watch face ID, and the complete watch face information can be found by matching the watch face information that is queried.
    - The information reported by the device can be retrieved by type
- Examples of use

```
 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?
YCReceivedDeviceReportInfo)?.data as? UInt32  else {
              return
          }
    print("dialID: \(dialID)")
}
```

9.5 Download watch face

- Method

```
/// Download watch face
/// - Parameters:
///   - peripheral: Connected device
///   - isEnable: On or off
///   - data: Dial data
///   - dialID: Dial ID
///   - blockCount: Dial breakpoint
///   - dialVersion: Dial version
///   - completion: Download progress
public static func downloadWatchFace(
  _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
  isEnable: Bool,
  data: NSData,
  dialID: UInt32,
  blockCount: UInt16,
  dialVersion: UInt16,
  completion: ((YCProductSDK.YCProductState, Any?) -> ())?
)

/// Download data progress information
@objcMembers public class YCDownloadProgressInfo : NSObject {

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

    /// Downloaded data size
    public  var downloaded: Int

    /// The size of the total downloaded data
    public  var total: Int { get }
}
```

- Instruction
    - The parameters in the download dial will indicate whether the download is normal and the download progress through the return value.
- Examples of use

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

let dialID: UInt32 = 2147483539

// Remove watch face
YCProduct.deleteWatchFace(dialID: dialID) { state, response in

    // Download
    YCProduct.downloadWatchFace(
      isEnable: 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 Custom watch face

- Method

```
/// Generate custom watch face data
/// - Parameters:
///   - dialData: Original dial data
///   - backgroundImage: Background picture
///   - thumbnail: Thumbnail
///   - timePosition: Time display position coordinates
///   - redColor: 0 ~ 255
///   - greenColor: 0 ~ 255
///   - blueColor: 0 ~ 255
/// - Returns: Dial data
public static func generateCustomDialData(
  _ dialData: Data,
  backgroundImage: UIImage?,
  thumbnail: UIImage?,
  timePosition: CGPoint,
  redColor: UInt8,
  greenColor: UInt8,
  blueColor: UInt8
) -> Data

/// Query the BMP information in the dial file
public static func queryDeviceBmpInfo(_ dialData: Data) ->
YCProductSDK.YCWatchFaceDataBmpInfo

/// Picture information in the watch face
@objcMembers public class YCWatchFaceDataBmpInfo : NSObject {

    /// The width of the background image
    public  var width: Int { get }

    /// The height of the background image
    public  var height: Int { get }
```

```
    /// The size of the background image (bytes)
    public  var size: Int { get }

    /// The radius of the background image
    public  var radius: Int { get }

    /// The width of the thumbnail
    public  var thumbnailWidth: Int { get }

    /// The width of the thumbnail
    public  var thumbnailHeight: Int { get }

    /// Thumbnail size (bytes)
    public  var thumbnailSize: Int { get }

    /// The radius of the thumbnail
    public  var thumbnailRadius: Int { get }
}
```

- Instruction
    - The custom dial is based on the custom dial source file provided by the manufacturer, the picture and text color are modified, and a new dial file is generated and downloaded to the device.
    - If you don’t modify the picture, you can pass in nil, and the SDK will keep the original background picture and thumbnail.
    - A new dial file is generated, directly call the method downloaded by the dial, and download it to the device.
    - The document gives a method for querying information of the dial BMP, which may be used in the App development interface or when generating thumbnails.
- Examples of use

```
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
) as NSData
```

10. Historical data collection

Note: This part is mainly to obtain ECG and PPG data, other types are not supported temporarily.

```
/// Type of collected data
@objc public enum YCCollectDataType : UInt8 {
    case ecg                                // ECG data
    case ppg                                // PPG data
    case triaxialAcceleration               // Three-axis acceleration
data
    case sixAxisSensor                      // Six-axis sensor data
    case nineAxisSensor                     // Nine axis sensor data
    case triaxialMagnetometer               // Three-axis magnetometer
data
    case inflationBloodPressure             // Inflation blood pressure
data
}
```

10.1 Query information record

- Method

```
/// Query basic information of local historical collected data
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: YCCollectDataType
///   - completion: information record
public static func queryCollectDataBasicinfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCCollectDataType,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Basic information of historical collected data
@objcMembers public class YCCollectDataBasicInfo : NSObject {

    /// Collection type
    public  var dataType: YCProductSDK.YCCollectDataType { get }

    /// Serial number
    public  var index: UInt16 { get }

    /// Timestamp (seconds)
    public  var timeStamp: UInt32 { get }

    /// Sampling Rate
    public  var sampleRate: UInt16 { get }

    /// Number of samples
    public  var samplesCount: UInt8 { get }

    /// Total bytes
```

```
    public  var totalBytes: UInt32 { get }

    /// Total number of packages
    public  var packages: UInt16 { get }
}
```

- Instruction
    - By calling this method, you can obtain several records of the current device and the basic information of each record.
- Examples of use

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

10.2 Get specific data

- Method

```
/// Get data through index
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: Data collection type
///   - index: Serial number
///   - uploadEnable: Whether to report data
///   - completion: Result
public static func queryCollectDataInfo(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCCollectDataType,
```

```
    index: UInt16 = 0,
    uploadEnable: Bool = true,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)

/// Historical collection data information
@objcMembers public class YCCollectDataInfo: NSObject {

    /// Basic information
    public  var basicInfo: YCCollectDataBasicInfo { get }

    /// Sync progress
    public  var progress: Float { get }

    /// Whether the transfer is complete
    public  var isFinished: Bool { get }

    /// Response data
    public  var data: [Int32] { get }
}
```

- Instruction
    - The return value of this method is of type YCCollectDataInfo, which contains information such as data and progress.
- Examples of use

```
YCProduct.queryCollectDataInfo( dataType: .ecg,
                                index: 0,
                                uploadEnable: true) { state, response in
    if state == .succeed,
    let info = response as? YCCollectDataInfo,
       info.isFinished {
        print(info.data, info.progress)
    }
}
```

10.3 Delete data

- Method

```
/// Delete by index
/// - Parameters:
///   - peripheral: Connected device
///   - dataType: Data collection type
///   - index: Serial number
///   - completion: Result
public static func deleteCollectData(
    _ peripheral: CBPeripheral? = YCProduct.shared.currentPeripheral,
    dataType: YCCollectDataType,
    index: UInt16,
    completion: ((_ state: YCProductState, _ response: Any?) -> ())?
)
```

- Instruction
    - If you do not actively delete the data, this data will always exist, and once the storage limit of the device is reached, it will be deleted by the device.
- Examples of use

```
YCProduct.deleteCollectData(dataType: .ecg,
                            index: 0) { state, response in

}
```

11. Firmware upgrade

11.1 Get information about device upgrades

11.1.1 Main control chip model

Before firmware upgrade, you need to confirm the hardware platform used by the device. You can obtain the main control chip model used by the device through section 5.6 to call different upgrade library files. nrf52832 uses Nordic's upgrade library, rtk8762c or rtk8762d uses Realtek's upgrade library.

11.1.2 Firmware version

You can use section 5.2 to obtain the firmware version information used by the device, including the major version and the sub-version. When judging the size of the version number, pay attention to it. If the major version is not the same, the larger the major version value, the higher the version. If the main version is the same, compare the sub-versions. The larger the number of the sub-version, the higher the version. For example, the firmware version 1.10 is higher than version 1.1, which cannot be judged from mathematical values.

11.2 Nordic firmware upgrade

11.2.1 Import library file

The Nordic firmware upgrade library can be obtained directly from github. The address is https://github.com/NordicSemiconductor/IOS-DFU-Library

It is recommended to use Pod to install

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

11.2.2 Realize firmware upgrade

The API parameters in the method are explained in detail in the Nordic upgrade library

```
import iOSDFULibrary

class YCFirmwareUpgradeViewController: UIViewController {

    /// NRF upgrade control
    private var dfuController: DFUServiceController?

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

        _ = dfuController?.abort()
    }

}

// MARK: - Firmware upgrade
extension YCFirmwareUpgradeViewController: DFUServiceDelegate,
DFUProgressDelegate {
```

```
    /// NRF firmware upgrade (custom implementation)
    /// - Parameters:
    ///   - filePath: Firmware file address
    ///   - device: Equipment that needs to be upgraded
    private func startNRFFirmwareUpgrade(_ filePath: URL,
                                         device: CBPeripheral) {

        guard let dfuFirmware = DFUFirmware(urlToZipFile: filePath) else
{
            printLog("Firmware does not exist")
            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)
    }

    /// State change
  func dfuStateDidChange(to state: DFUState) {
      switch state {

      case .disconnecting:
```

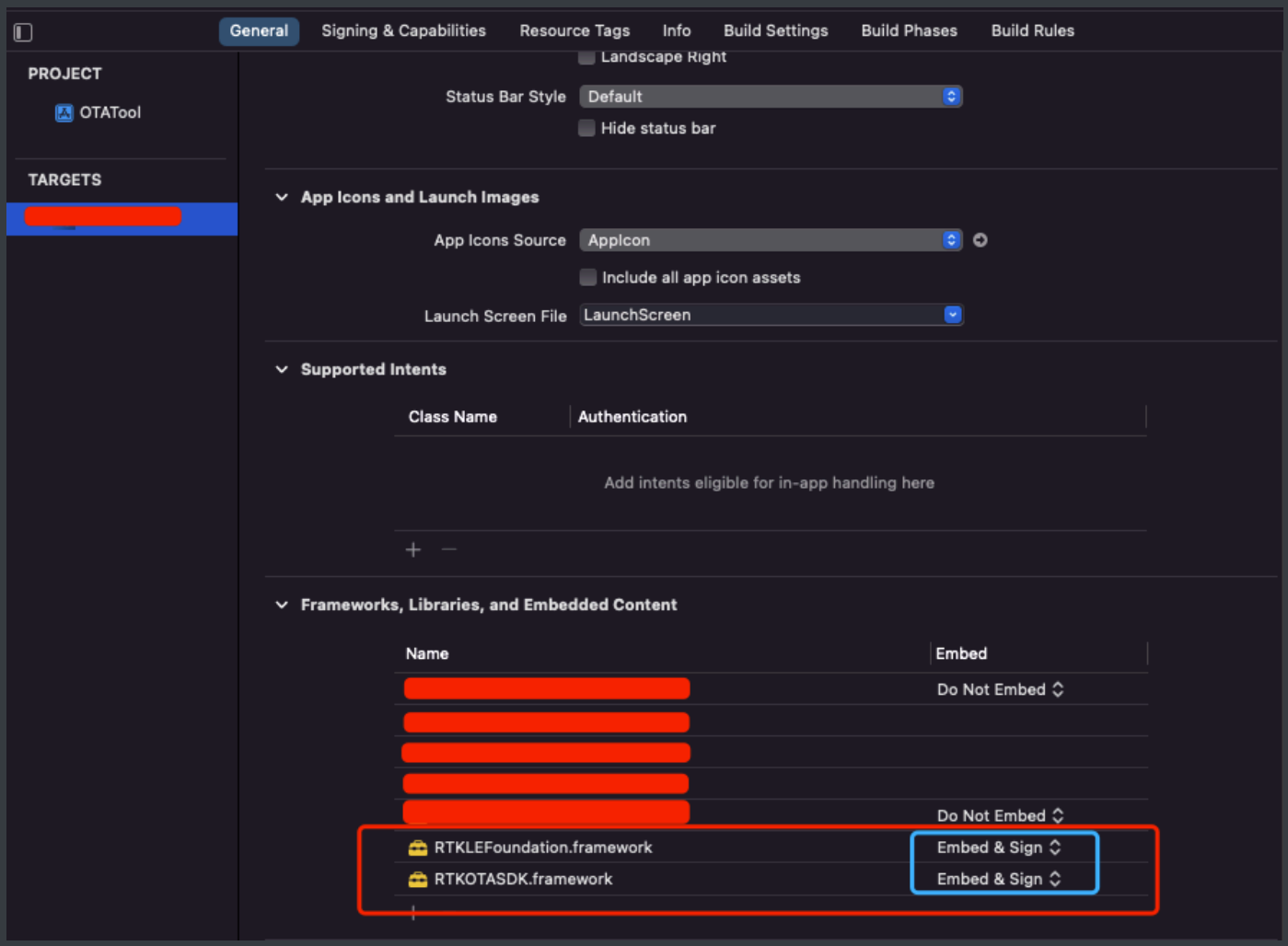
```
        break

      case .connecting:
        break

      case .starting:
        break

      case .enablingDfuMode: // Enter the upgrade state
        break

      case .completed:  // End of upgrade
        break

      case .aborted:
        break

      default:
        break
    }

    /// Upgrade error
    func dfuError(_ error: DFUError, didOccurWithMessage message:
String) {

    }

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

    }
}
```

11.3 Realtek firmware upgrade

Realtek's upgrade operation is more complicated than Nordic

11.3.1 Import library file

Two Realtek library files `RTKOTASDK.framework` and `RTKLEFoundation.framework` are provided in the SDK file directory, and these two libraries are imported into the project.



11.3.2 Realize firmware upgrade

```
import UIKit
import RTKOTASDK

class YCFirmwareUpgradeViewController: UIViewController {

    /// Firmware storage path
    private var firmwarePath: String = ""

    /// Upgrade equipment
    private var rtkDevice: CBPeripheral?

    /// RTK uses management objects required for any operation
    private var rtkProfile: RTKOTAProfile?

    /// RTK uses current equipment
    private var rtkPeripheral: RTKOTAPeripheral?

    /// DFU device entered by RTK
    private var rtkDfuPerpheral: RTKMultiDFUPeripheral?

    /// RTK upgraded firmware image
    private var rtkImages: [RTKOTAUpgradeBin]?
}

// MARK: -Firmware upgrade
extension YCFirmwareUpgradeViewController {

    /// Upgrade RTK
    /// - Parameters:
    ///   - device: rtkDevice
    private func startRTKFirmwareUpgrade(_ device: CBPeripheral) {

        firmwarePath = "Firmware storage path"
```

```
        RTKLog.setLogLevel(.off)  // Turn off log printing
        rtkProfile = RTKOTAProfile()
        rtkProfile?.delegate = self

        if rtkPeripheral != nil {
            rtkProfile?.cancelConnection(with: rtkPeripheral!)
        }

        rtkPeripheral = rtkProfile?.otaPeripheral(from: device)

        if rtkPeripheral != nil {
            rtkProfile?.connect(to: rtkPeripheral!)
        }

    }
}

// MARK: - Upgrade related agents
extension YCFirmwareUpgradeViewController: RTKLEProfileDelegate,
RTKDFUPeripheralDelegate {

    /// Connect the device (twice) Pay attention to the operation logic
of this part
    func profile(_ profile: RTKLEProfile, didConnect peripheral:
RTKLEPeripheral) {

          if peripheral.cbPeripheral.identifier == rtkDevice?.identifier
&&
        peripheral != rtkDfuPerpheral {

            if let zipFile = try?
RTKOTAUpgradeBin.imagesExtracted(fromMPPackFilePath: firmwarePath),
               zipFile.count == 1,
               zipFile.last?.icDetermined == false,
               rtkPeripheral != nil {
```

```
                rtkImages = zipFile
                rtkImages?.last?.assertAvailable(for: rtkPeripheral!)

            }

            if rtkPeripheral != nil,
               let dfuDevice = rtkProfile?.dfuPeripheral(of:
rtkPeripheral!) {

                dfuDevice.delegate = self
                profile.connect(to: dfuDevice)
                rtkDfuPerpheral = dfuDevice as? RTKMultiDFUPeripheral

                return
            }

        } else if peripheral == rtkDfuPerpheral {

            if let zipFile = rtkImages {
                rtkDfuPerpheral?.upgradeImages(zipFile, inOTAMode:
false)
            }
        }
    }

    /// Upgrade progress
    func dfuPeripheral(_ peripheral: RTKDFUPeripheral, didSend length:
UInt, totalToSend totalLength: UInt) {

    }

    /// End of upgrade
    func dfuPeripheral(_ peripheral: RTKDFUPeripheral,
didFinishWithError err: Error?) {

        if err == nil {
```

```
                print("update successed")
            } else {
                print("Upgrade failed")
            }
        }

        /// Device disconnected
        func profile(_ profile: RTKLEProfile, didDisconnectPeripheral
peripheral: RTKLEPeripheral, error: Error?) {

        }

        /// Connection failed
        func profile(_ profile: RTKLEProfile, didFailToConnect peripheral:
RTKLEPeripheral, error: Error?) {

        }

        /// State change
        func profileManagerDidUpdateState(_ profile: RTKLEProfile) {

        }
    }
```