

[Core Bluetooth](#) / Transferring Data Between Bluetooth Low Energy Devices

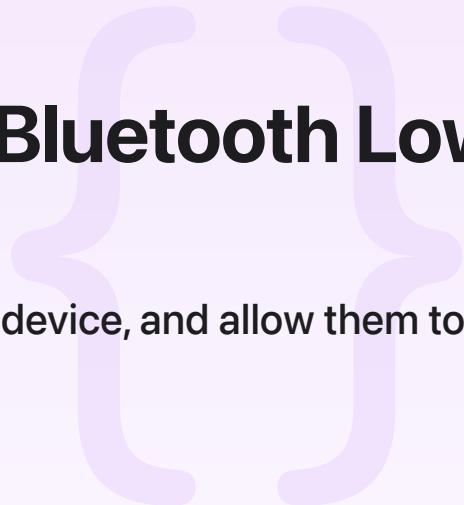
Sample Code

Transferring Data Between Bluetooth Low Energy Devices

Create a Bluetooth low energy central and peripheral device, and allow them to discover each other and exchange data.

[Download](#)

iOS 12.0+ | iPadOS 12.0+ | Xcode 11.1+



Overview

This sample shows how to transfer data between two iOS devices, with one acting as a Bluetooth central and the other as a peripheral, by using a [CBCharacteristic](#) on the peripheral side that changes its value. The value change is automatically picked up on the central side. The sample also shows how the central side can write data to a [CBCharacteristic](#) on the peripheral side.

This sample shows how to handle flow control in this scenario. It also covers a rudimentary way to use the Received Signal Strength Indicator (RSSI) value to determine whether data transfer is feasible.

Configure the Sample Code Project

1. Run the sample on two devices that support Bluetooth LE.
2. On one device, tap the “Central” button. This device will be the central mode device. The device will begin scanning for a peripheral device that is advertising the Transfer Service.
3. On the other device, tap the “Peripheral” button. This device will be the peripheral mode device.
4. On the peripheral mode device, tap the advertise on/off switch, to enable peripheral mode advertising of the data in the text field.

5. Bring the two devices close to each other.

Discover Bluetooth Peripherals and Connect to Them

The device running in central mode creates a `CBCentralManager`, assigning the `CentralViewController` as the manager's delegate. It calls `scanForPeripherals` to discover other Bluetooth devices, passing in the UUID of the service it's searching for.

```
centralManager.scanForPeripherals(withServices: [TransferService.serviceUUID],  
                                    options: [CBCentralManagerScanOptionAllowDuplicates])
```

When the central manager discovers a peripheral with a matching service UUID, it calls `centralManager(_:didDiscover:advertisementData:rssi:)`. The sample's implementation of this method uses the `rssi` (Received Signal Strength Indicator) parameter to determine whether the signal is strong enough to transfer data. RSSI values are provided as negative numbers, with a theoretical maximum of 0. The sample proceeds with transfer if the `rssi` is greater than or equal to -50. If the peripheral's signal is strong enough, the method saves the peripheral as the property `discoveredPeripheral` and calls `connect` to connect to it.

```
func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,  
                    advertisementData: [String: Any], rssi RSSI: NSNumber) {  
  
    // Reject if the signal strength is too low to attempt data transfer.  
    // Change the minimum RSSI value depending on your app's use case.  
    guard RSSI.intValue >= -50  
    else {  
        os_log("Discovered perhiperal not in expected range, at %d", RSSI.intValue)  
        return  
    }  
  
    os_log("Discovered %s at %d", String(describing: peripheral.name), RSSI.intValue)  
  
    // Device is in range – have we already seen it?  
    if discoveredPeripheral != peripheral {  
  
        // Save a local copy of the peripheral, so CoreBluetooth doesn't get rid of it.  
        discoveredPeripheral = peripheral  
  
        // And finally, connect to the peripheral.  
        os_log("Connecting to perhiperal %@", peripheral)  
        centralManager.connect(peripheral, options: nil)  
    }  
}
```

```
}
```

```
}
```

When the Peripheral Receives a Connection, Send the Data

The device running in peripheral mode creates a `CBPeripheralManager` and assigns its `PeripheralViewController` as the manager's delegate.

When the `peripheralManagerDidUpdateState` method indicates that Bluetooth has powered on, the sample calls a private `setupPeripheral()` method to create a `CBMutableCharacteristic` called `transferCharacteristic`. It then creates a `CBMutableService` from the characteristic and adds the service to the `CBPeripheralManager`.

```
private func setupPeripheral() {  
  
    // Build our service.  
  
    // Start with the CBMutableCharacteristic.  
    let transferCharacteristic = CBMutableCharacteristic(type: TransferService.characteristicType,  
                                                          properties: [.notify, .writeWithoutResponse],  
                                                          value: nil,  
                                                          permissions: [.readable, .writeable])  
  
    // Create a service from the characteristic.  
    let transferService = CBMutableService(type: TransferService.serviceUUID, primary: true)  
    transferService.characteristics = [transferCharacteristic]  
  
    // Add the characteristic to the service.  
    transferService.characteristics = [transferCharacteristic]  
  
    // And add it to the peripheral manager.  
    peripheralManager.add(transferService)  
  
    // Save the characteristic for later.  
    self.transferCharacteristic = transferCharacteristic  
  
}
```

The user interface provides a `UISwitch` that starts or stops advertising of the peripheral's service UUID.

```

@IBAction func switchChanged(_ sender: Any) {
    // All we advertise is our service's UUID.
    if advertisingSwitch.isOn {
        peripheralManager.startAdvertising([CBAdvertisementDataServiceUUIDsKey: [Tran
    } else {
        peripheralManager.stopAdvertising()
    }
}

```

Once the central device discovers and connects to the peripheral, the peripheral side sends the data from the text field to the central. PeripheralViewController sends the data in chunks — each sized to the maximum value the central can receive — by setting the value of its transfer Characteristic property to the latest chunk. When finished, it sends the value EOM (for “end of message”).

When the Central Receives the Data, Update the User Interface

Back on the central device, a call to the central manager delegate’s `peripheral(_:didDiscoverCharacteristicsFor:error:)` tells the app that it has discovered the peripheral’s transfer characteristic. The sample’s implementation of this method calls `setNotifyValue(_:for:)` to start receiving updates to the characteristic’s value.

When the value does update — meaning text is available — the central manager calls the delegate method `peripheral(_:didUpdateValueFor:error:)`. The sample looks to see if the data is a chunk or an end-of-message marker. If the data is a chunk, the code appends the chunk to an internal buffer containing the peripheral’s message. If the data is an end-of-message marker, it converts the buffer to a string and sets it as the contents of the text field.

```

func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic: CBCharacteristic) {
    // Deal with errors (if any)
    if let error = error {
        os_log("Error discovering characteristics: %@", error.localizedDescription)
        cleanup()
        return
    }

    guard let characteristicData = characteristic.value,
          let stringFromData = String(data: characteristicData, encoding: .utf8) else {
        os_log("Received %d bytes: %@", characteristicData.count, stringFromData)
    }
}

```

```
// Have we received the end-of-message token?  
if stringFromData == "EOM" {  
    // End-of-message case: show the data.  
    // Dispatch the text view update to the main queue for updating the UI, because  
    // we don't know which thread this method will be called back on.  
    DispatchQueue.main.async() {  
        self.textView.text = String(data: self.data, encoding: .utf8)  
    }  
  
    // Write test data  
    writeData()  
} else {  
    // Otherwise, just append the data to what we have previously received.  
    data.append(characteristicData)  
}  
}
```

Once the transfer is complete, you can press the “Back” button on each device and reassign the central and peripheral roles, to perform the transfer in the opposite direction.