Questo codice MicroPython implementa un semplice **servizio Bluetooth Low Energy (BLE) UART (Universal Asynchronous Receiver/Transmitter)** su un dispositivo MicroPython (come una scheda ESP32 o Raspberry Pi Pico W). In sostanza, trasforma il tuo dispositivo in un **periferico BLE** che può inviare e ricevere dati in modalità wireless, simulando una comunicazione seriale.

### Spiegazione Dettagliata

Analizziamo il codice passo dopo passo:

#### 1. Importazioni Iniziali

import bluetooth  
import random  
import struct  
import time  
from machine import Pin  
from ble\_advertising import advertising\_payload  
from micropython import const

* bluetooth: La libreria principale per la gestione del BLE in MicroPython.
* random, struct, time: Moduli standard di Python per funzionalità aggiuntive (sebbene random e struct non siano direttamente utilizzati nell'esempio demo, potrebbero essere usati in un'applicazione più complessa).
* from machine import Pin: Permette di controllare i pin GPIO (General Purpose Input/Output) del microcontrollore, in questo caso per un LED.
* from ble\_advertising import advertising\_payload: Una funzione helper (presumibilmente da un modulo separato, come suggerisce il commento PiCockpit.com) che aiuta a creare i dati di advertising BLE.
* from micropython import const: Una funzione speciale di MicroPython per definire costanti. Questo aiuta a ottimizzare l'uso della memoria e le prestazioni.

#### 2. Costanti BLE

\_IRQ\_CENTRAL\_CONNECT = const(1)  
\_IRQ\_CENTRAL\_DISCONNECT = const(2)  
\_IRQ\_GATTS\_WRITE = const(3)  
  
\_FLAG\_READ = const(0x0002)  
\_FLAG\_WRITE\_NO\_RESPONSE = const(0x0004)  
\_FLAG\_WRITE = const(0x0008)  
\_FLAG\_NOTIFY = const(0x0010)

Queste costanti definiscono i tipi di eventi e le proprietà dei caratteristici BLE:

* **Eventi IRQ (Interrupt ReQuest)**:
  + \_IRQ\_CENTRAL\_CONNECT: Viene generato quando un dispositivo centrale (ad esempio, uno smartphone) si connette al nostro periferico.
  + \_IRQ\_CENTRAL\_DISCONNECT: Viene generato quando un dispositivo centrale si disconnette.
  + \_IRQ\_GATTS\_WRITE: Viene generato quando un dispositivo centrale scrive un valore su un caratteristico del nostro periferico.
* **Flag di Permesso (Proprietà dei Caratteristici)**:
  + \_FLAG\_READ: Permette di leggere il valore del caratteristico.
  + \_FLAG\_WRITE\_NO\_RESPONSE: Permette di scrivere il valore senza una risposta di conferma.
  + \_FLAG\_WRITE: Permette di scrivere il valore con una risposta di conferma.
  + \_FLAG\_NOTIFY: Permette al periferico di inviare notifiche al centrale quando il valore del caratteristico cambia.

#### 3. Definizione del Servizio UART BLE

\_UART\_UUID = bluetooth.UUID("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")  
\_UART\_TX = (  
 bluetooth.UUID("6E400003-B5A3-F393-E0A9-E50E24DCCA9E"),  
 \_FLAG\_READ | \_FLAG\_NOTIFY,  
)  
\_UART\_RX = (  
 bluetooth.UUID("6E400002-B5A3-F393-E0A9-E50E24DCCA9E"),  
 \_FLAG\_WRITE | \_FLAG\_WRITE\_NO\_RESPONSE,  
)  
\_UART\_SERVICE = (  
 \_UART\_UUID,  
 (\_UART\_TX, \_UART\_RX),  
)

Questa sezione definisce la struttura del servizio UART personalizzato:

* \_UART\_UUID: L'UUID (Universally Unique Identifier) del servizio UART. Questo è un UUID standard per i servizi UART BLE.
* \_UART\_TX: Questo è il **caratteristico di trasmissione (TX)** dal punto di vista del periferico. Ha un suo UUID specifico e le proprietà \_FLAG\_READ (può essere letto) e \_FLAG\_NOTIFY (il periferico può inviare notifiche/dati al centrale).
* \_UART\_RX: Questo è il **caratteristico di ricezione (RX)** dal punto di vista del periferico. Ha un suo UUID e le proprietà \_FLAG\_WRITE e \_FLAG\_WRITE\_NO\_RESPONSE (il centrale può scriverci dei dati).
* \_UART\_SERVICE: Una tupla che combina l'UUID del servizio e i suoi caratteristici (\_UART\_TX e \_UART\_RX). Questa struttura è ciò che MicroPython si aspetta per registrare un servizio BLE.

#### 4. Classe BLESimplePeripheral

Questa classe incapsula tutta la logica per gestire il periferico BLE.

class BLESimplePeripheral:  
 def \_\_init\_\_(self, ble, name="mpy-uart"):  
 self.\_ble = ble  
 self.\_ble.active(True)  
 self.\_ble.irq(self.\_irq)  
 ((self.\_handle\_tx, self.\_handle\_rx),) = self.\_ble.gatts\_register\_services((\_UART\_SERVICE,))  
 self.\_connections = set()  
 self.\_write\_callback = None  
 self.\_payload = advertising\_payload(name=name, services=[\_UART\_UUID])  
 self.\_advertise()

* **\_\_init\_\_(self, ble, name="mpy-uart")**: Il costruttore della classe.
  + self.\_ble = ble: Salva l'istanza del gestore Bluetooth.
  + self.\_ble.active(True): Attiva l'interfaccia BLE.
  + self.\_ble.irq(self.\_irq): Registra il metodo \_irq come handler per gli eventi BLE. Ogni volta che si verifica un evento BLE (connessione, disconnessione, scrittura, ecc.), questo metodo verrà chiamato.
  + ((self.\_handle\_tx, self.\_handle\_rx),) = self.\_ble.gatts\_register\_services((\_UART\_SERVICE,)): Registra il servizio UART definito in precedenza. Questa chiamata restituisce i "handle" (identificatori numerici) per i caratteristici TX e RX, che verranno usati per interagire con essi (leggere/scrivere/notificare).
  + self.\_connections = set(): Un set per tenere traccia di tutte le connessioni attive. Un set è usato perché garantisce l'unicità degli handle di connessione.
  + self.\_write\_callback = None: Una variabile per memorizzare una funzione di callback che verrà chiamata quando i dati vengono ricevuti sul caratteristico RX.
  + self.\_payload = advertising\_payload(name=name, services=[\_UART\_UUID]): Crea il payload pubblicitario (i dati che il periferico trasmette per farsi trovare dagli altri dispositivi). Include il nome del dispositivo e l'UUID del servizio UART.
  + self.\_advertise(): Avvia il processo di advertising.

#### 5. Metodo \_irq (Handler Eventi)

def \_irq(self, event, data):  
 if event == \_IRQ\_CENTRAL\_CONNECT:  
 conn\_handle, \_, \_ = data  
 print("New connection", conn\_handle)  
 self.\_connections.add(conn\_handle)  
 elif event == \_IRQ\_CENTRAL\_DISCONNECT:  
 conn\_handle, \_, \_ = data  
 print("Disconnected", conn\_handle)  
 self.\_connections.remove(conn\_handle)  
 self.\_advertise() # Restart advertising after disconnection  
 elif event == \_IRQ\_GATTS\_WRITE:  
 conn\_handle, value\_handle = data  
 value = self.\_ble.gatts\_read(value\_handle)  
 if value\_handle == self.\_handle\_rx and self.\_write\_callback:  
 self.\_write\_callback(value)

Questo è il cuore della gestione degli eventi BLE:

* **\_IRQ\_CENTRAL\_CONNECT**: Quando un centrale si connette, l'handle della connessione (conn\_handle) viene aggiunto al set \_connections.
* **\_IRQ\_CENTRAL\_DISCONNECT**: Quando un centrale si disconnette, il suo handle viene rimosso dal set e l'advertising viene riavviato per permettere nuove connessioni.
* **\_IRQ\_GATTS\_WRITE**: Quando un centrale scrive su un caratteristico:
  + Vengono recuperati l'handle della connessione e l'handle del valore (value\_handle) del caratteristico scritto.
  + self.\_ble.gatts\_read(value\_handle) legge il valore effettivo scritto.
  + Se il value\_handle corrisponde al nostro caratteristico di ricezione (self.\_handle\_rx) e una funzione di callback per la scrittura (self.\_write\_callback) è stata definita, questa funzione viene chiamata con il valore ricevuto.

#### 6. Metodi di Comunicazione e Stato

def send(self, data):  
 for conn\_handle in self.\_connections:  
 self.\_ble.gatts\_notify(conn\_handle, self.\_handle\_tx, data)  
  
 def is\_connected(self):  
 return len(self.\_connections) > 0  
  
 def \_advertise(self, interval\_us=500000):  
 print("Starting advertising")  
 self.\_ble.gap\_advertise(interval\_us, adv\_data=self.\_payload)  
  
 def on\_write(self, callback):  
 self.\_write\_callback = callback

* **send(self, data)**: Invia dati a tutti i dispositivi centrali connessi. Utilizza self.\_ble.gatts\_notify per inviare una notifica sul caratteristico \_handle\_tx.
* **is\_connected()**: Restituisce True se ci sono connessioni attive, False altrimenti.
* **\_advertise(self, interval\_us=500000)**: Avvia (o riavvia) l'advertising BLE. L'interval\_us è l'intervallo tra le trasmissioni pubblicitarie in microsecondi.
* **on\_write(self, callback)**: Permette di registrare una funzione di callback personalizzata che verrà eseguita quando i dati vengono ricevuti dal centrale.

#### 7. Funzione demo()

Questa funzione dimostra l'uso della classe BLESimplePeripheral.

def demo():  
 led\_onboard = Pin("LED", Pin.OUT) # Initialize the onboard LED  
 ble = bluetooth.BLE()  
 p = BLESimplePeripheral(ble)  
  
 def on\_rx(v):  
 print("RX", v)  
  
 p.on\_write(on\_rx)  
  
 i = 0  
 while True:  
 if p.is\_connected():  
 led\_onboard.on() # Turn on LED if connected  
 for \_ in range(3):  
 data = str(i) + "\_"  
 print("TX", data)  
 p.send(data)  
 i += 1  
 time.sleep\_ms(100) # Wait 100ms

* **led\_onboard = Pin("LED", Pin.OUT)**: Inizializza il pin del LED di bordo come output. Su Raspberry Pi Pico W, "LED" si riferisce al LED integrato.
* **ble = bluetooth.BLE()**: Crea un'istanza dell'oggetto BLE.
* **p = BLESimplePeripheral(ble)**: Crea un'istanza del nostro periferico BLE personalizzato.
* **def on\_rx(v): print("RX", v)**: Definisce una semplice funzione di callback per la ricezione dei dati, che stampa i dati ricevuti sulla console.
* **p.on\_write(on\_rx)**: Registra la funzione on\_rx come callback per la scrittura (quando il centrale invia dati).
* **Ciclo while True**:
  + Controlla se il periferico è connesso (p.is\_connected()).
  + Se connesso, accende il LED di bordo (led\_onboard.on()).
  + Invia tre messaggi (data = str(i) + "\_") al dispositivo centrale utilizzando p.send(data). Il numero i viene incrementato per ogni messaggio.
  + Pausa per 100 millisecondi (time.sleep\_ms(100)).
  + **Nota**: Il LED rimane acceso finché c'è almeno una connessione. Se si disconnette, il LED dovrebbe spegnersi (dato che non c'è una logica esplicita per spegnerlo quando non è connesso, potrebbe rimanere acceso se non è spento prima di led\_onboard.on() - un miglioramento potrebbe essere led\_onboard.off() quando non è connesso).

#### 8. Esecuzione del Codice

if \_\_name\_\_ == "\_\_main\_\_":  
 demo()

Questa riga assicura che la funzione demo() venga chiamata solo quando lo script viene eseguito direttamente (non quando viene importato come modulo).