BlueR

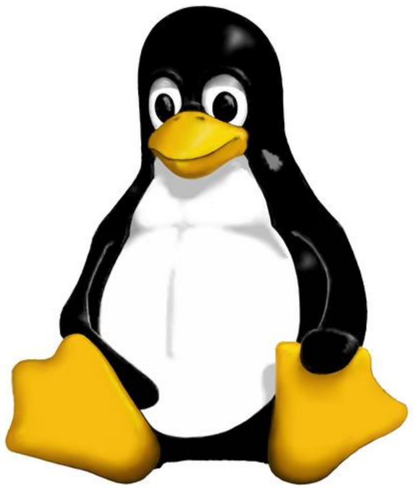
Rust bindings for the official Linux Bluetooth protocol stack

Dr. Sebastian Urban
surban@surban.net

About me

- MSc in Physics
  - fluid dynamics, partly using GPU computing (CUDA)
  - MATLAB
- PhD in Computer Science
  - thesis about activation functions in (deep) neural networks
  - Python / F# / CUDA
- Started using Rust in late 2018, commercially
  - RF test equipment, 3D image capture and processing
  - started by porting C++ code to Rust
  - now developing 90% in Rust
- Started 4 Rust open-source projects
  - `remoc`: Tokio’s channels but remotely (across process boundaries or network)
  - `bluer`: BlueZ bindings for Rust
  - `aggligator`: reliable multipath connections in user-space
  - `OpenEMC`: open management controller firmware and drivers

BlueZ

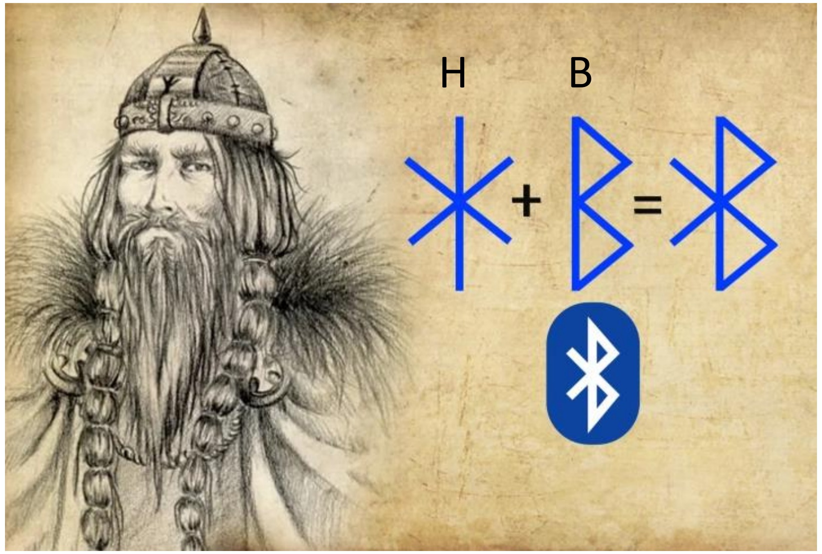


- Official Linux Bluetooth protocol stack
  - Started in 2001 at Qualcomm and released under GPL
  - First included in Linux 2.4.6
- Consists of two main parts
  - kernel code implementing device drivers and middle-level protocols
  - privileged `bluetoothd` process for management and high-level protocols
- User-space interfaces via sockets and D-Bus calls
  - BlueR exposes a subset of them to Rust
  - no logic in BlueR: almost every decision is made by the Bluetooth controller's firmware, Linux kernel or `bluetoothd`

BlueR

- BSD license
- Started as a fork Blurz, became full rewrite.
- Hosted by the BlueZ organization at https://github.com/bluez/bluer
- Requirements
  - Linux (!)
  - running `bluetoothd` for most functions
  - D-Bus development headers
    `sudo apt install libdbus-1-dev`
  - Tokio async runtime
- Add dependency
  - `cargo add bluer -F full`
- Install tools (optional)
  - `cargo install bluer-tools`

Bluetooth

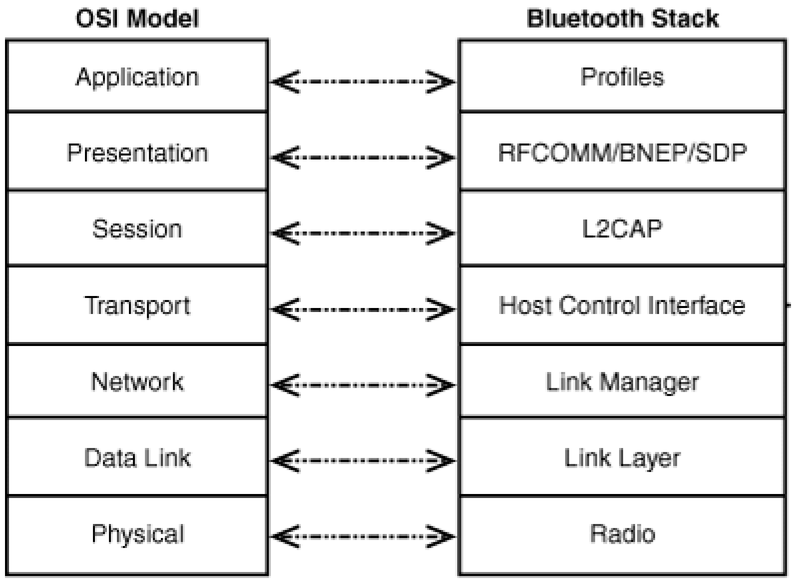


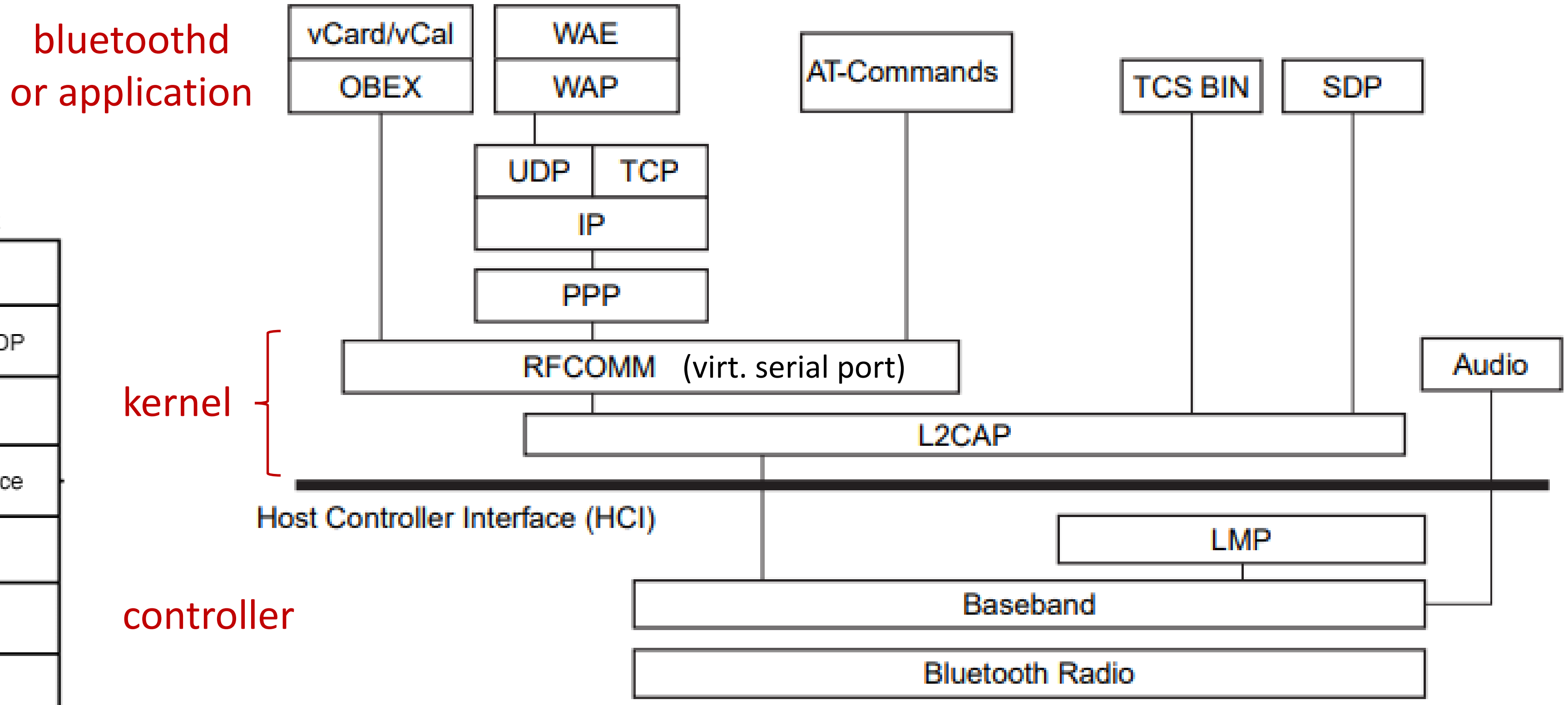
- Motivation: Wireless communication technology for short-range data exchange
- Named after Harald Bluetooth, 10th-century king who united Denmark and Norway
- Invented by Ericsson in 1994, first adopted in 1999
  - First usage: link an IBM ThinkPad with a GSM phone
- Operates on 2.4 GHz ISM frequency band (2400 - 2483.5 MHz)
- Device classes
  - Class 1: 100 mW power, up to 100 meters
  - Class 2: 2.5 mW power, up to 10 meters
  - Class 3: 1 mW power, up to 1 meter

Bluetooth BR/EDR vs. LE

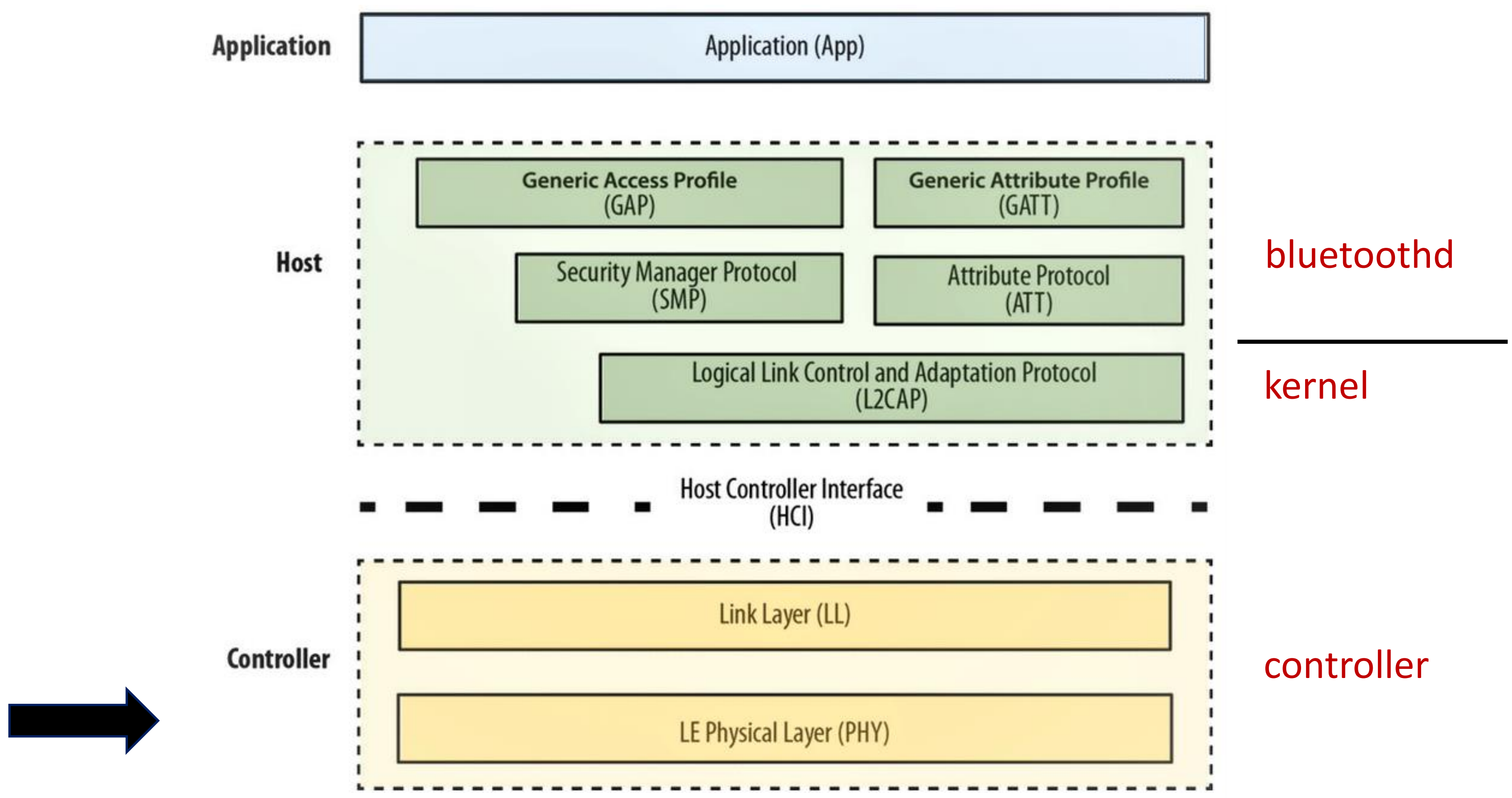
- BR/EDR (Basic Rate/Enhanced Data Rate):
  - Classic Bluetooth
  - Data rates of about 1-3 Mbps
  - Device to device connections
  - Uses Bluetooth profiles to provide services:
    File Transfer Profile (FTP), Headset Profile (HSP), Human Interface Device (HID)
- LE (Low Energy):
  - Introduced in Bluetooth 4.0 (2010)
  - Optimized for low power consumption
  - Lower data rates (up to 1 Mbps)
  - Ideal for intermittent data transfer (e.g., sensors, wearables)
  - Broadcasting data
  - Uses Generic Attribute Profile (GATT) to provide services:
    heart rate, wind speed, temperature, waist circumfence
- Coexistence: Dual-mode devices support both protocols

BR/EDR protocol stack





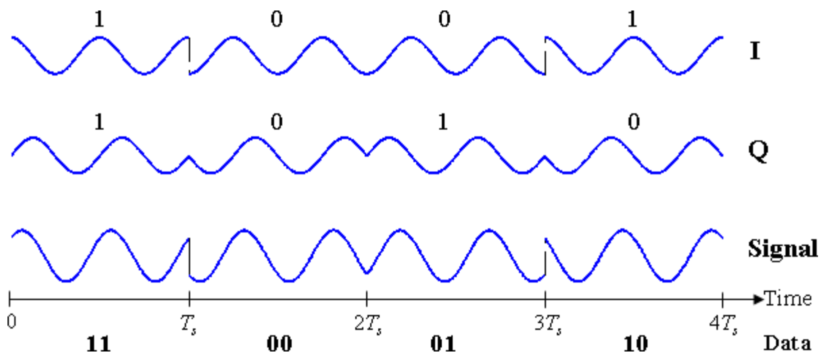
LE protocol stack



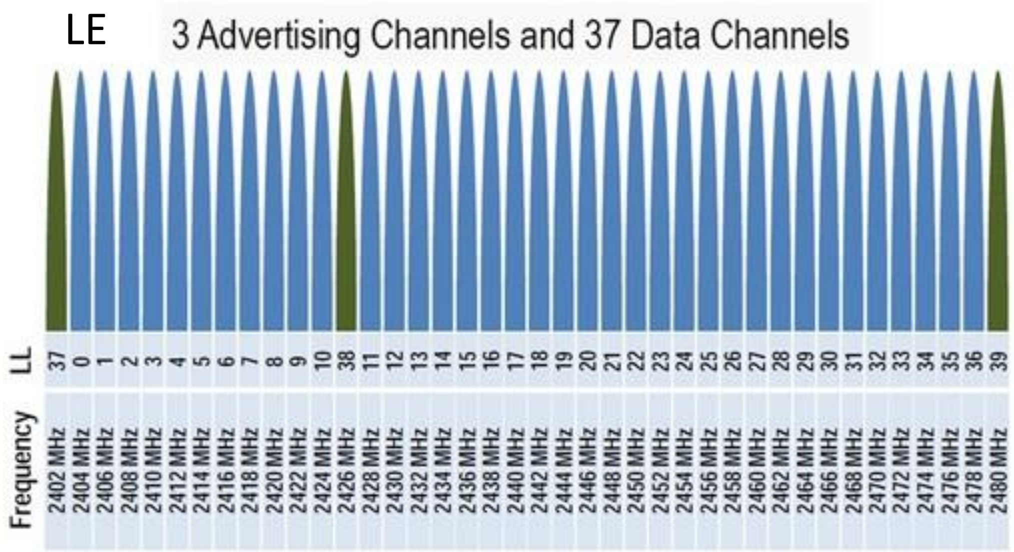
Bluetooth radio (PHY)

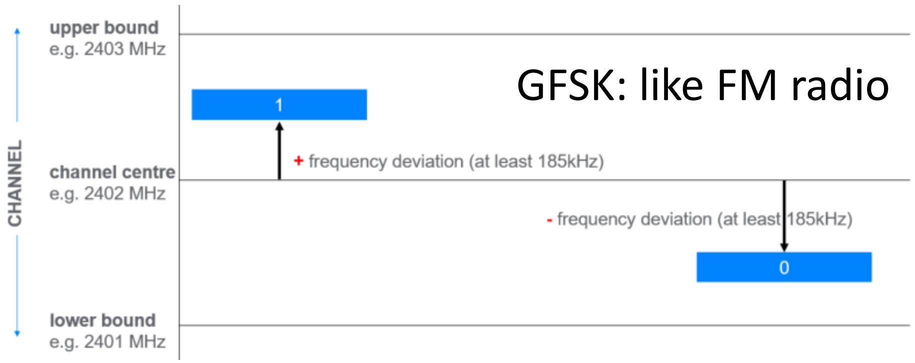
- Channels, starting at 2402 MHz
  - BR/EDR: 79 channels, each 1 MHz wide
  - LE: 40 channels, each 2 MHz wide
  - Adaptive Frequency Hopping (AFH) to minimize interference
- Half-duplex radio with time-division duplex (TDD)
- Modulation
  - BR/LE: Gaussian Frequency Shift Keying (GFSK)
  - EDR: Quadrature Phase-Shift Keying (QPSK)

QPSK: two bits per symbol

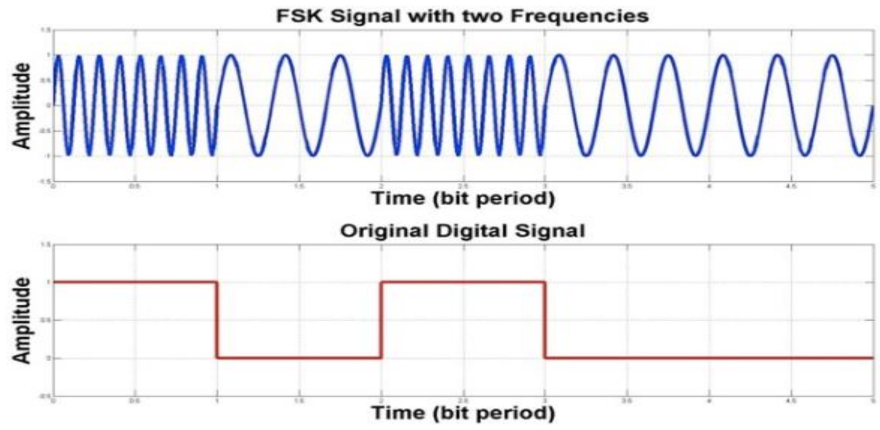


LE

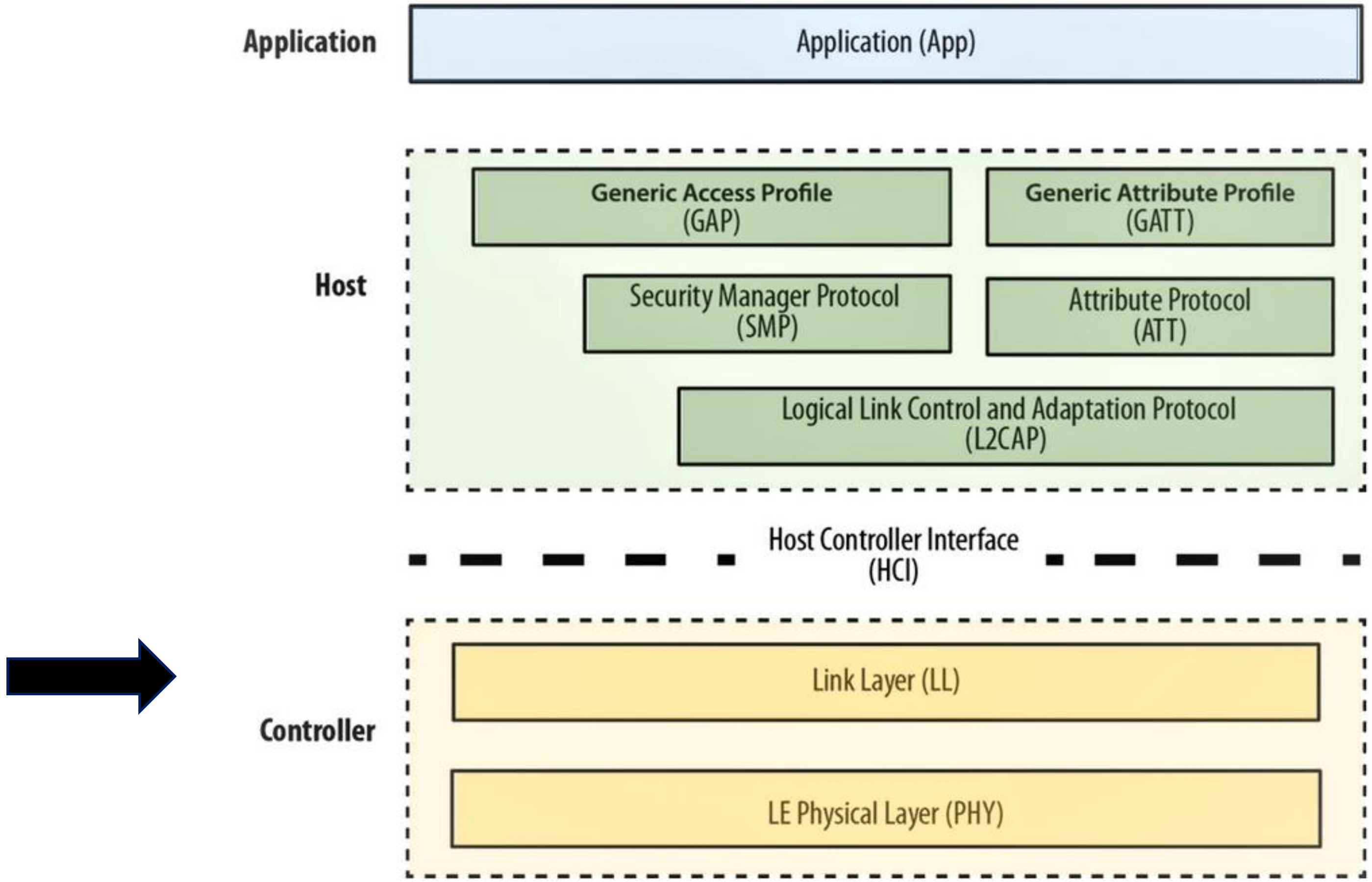




GFSK: like FM radio



LE protocol stack

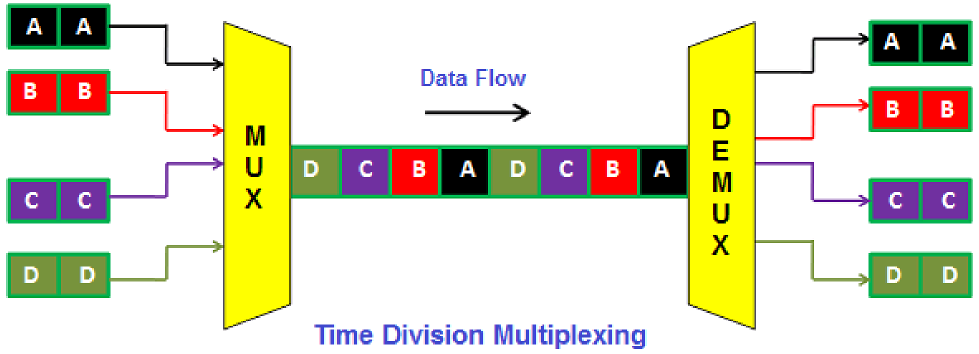


bluetoothd

kernel

controller

BR/EDR: Link Layer (LL)



- time-division multiplexing (TDM) provides multiple links
  - time slots of 625 µs
  - max 3 synchronous connection-oriented (SCO) links
    - fixed time-slots and therefore guaranteed bandwidth
    - used for real-time audio
  - one asynchronous connection-less (ACL) link
    - gets remaining, free time-slots
    - used for data and control
    - multiplexed in higher protocol layers (L2CAP)
- one master device
  - responsible for scheduling, allocation of time-slots
- multiple slaves (piconet)
  - slave can only transmit directly after receiving a packet from the master
- LM handles key management and exchange (pairing)

BR/EDR: link layer packet structure

LSB … MSB

Access Code	Header	BR Payload
68 or 72 bits	54 bits	0-2790 bits

Access Code

LSB … MSB

Preamble	Sync Word	Trailer
4 bits	64 bits	4 bits

Sync Word derived from
48-bit Bluetooth Device Address.

Header

LSB … MSB

LT_ADDR	Type	FLOW	ARQN	SEQN	HEC
3 bits	4 bits	1 bit	1 bit	1 bit	8 bits

ACL Payload

LSB … MSB

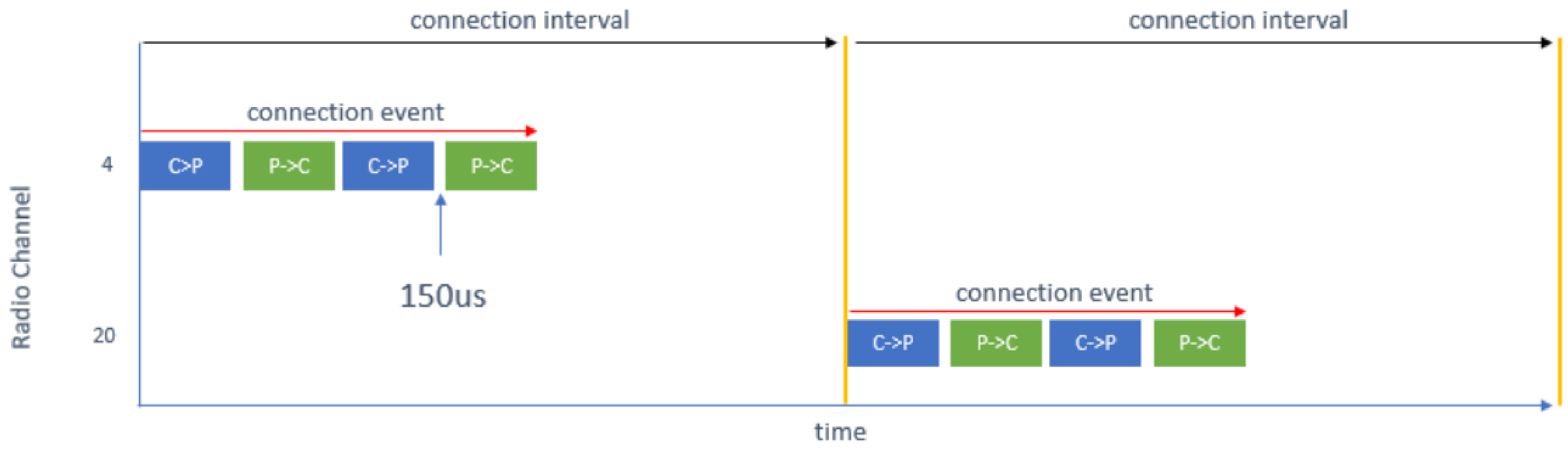
LLID	FLOW	Length
2 bits	1 bit	5 bits

+ data (L2CAP) + CRC

⇨ ordered, reliable connection

LE: Link Layer (LL)

- based on “connection events”
  - master and slave agree upon connection interval
  - master sends packet at start of each connection event
  - slave must reply immediately with one packet
  - if it has no data to send, it transmits an empty packet
- slave can miss connection events to save power



LE: link layer packet structure

LSB → MSB

Preamble	Access Address	Protocol Data Unit (PDU)	Cyclic Redundancy Check (CRC)	Constant Tone Extension
1 or 2 octets	4 octets	2-258 octets	3 octets	16 to 160 µs

Advertising PDU

LSB → MSB

Header	Payload
16 bits	1-255 octets

Header

LSB → MSB

PDU Type	RFU	ChSel	RFU	ChSel	Length
4 bits	1 bit	1 bit	1 bit	1 bit	8 bits

Data PDU

LSB → MSB

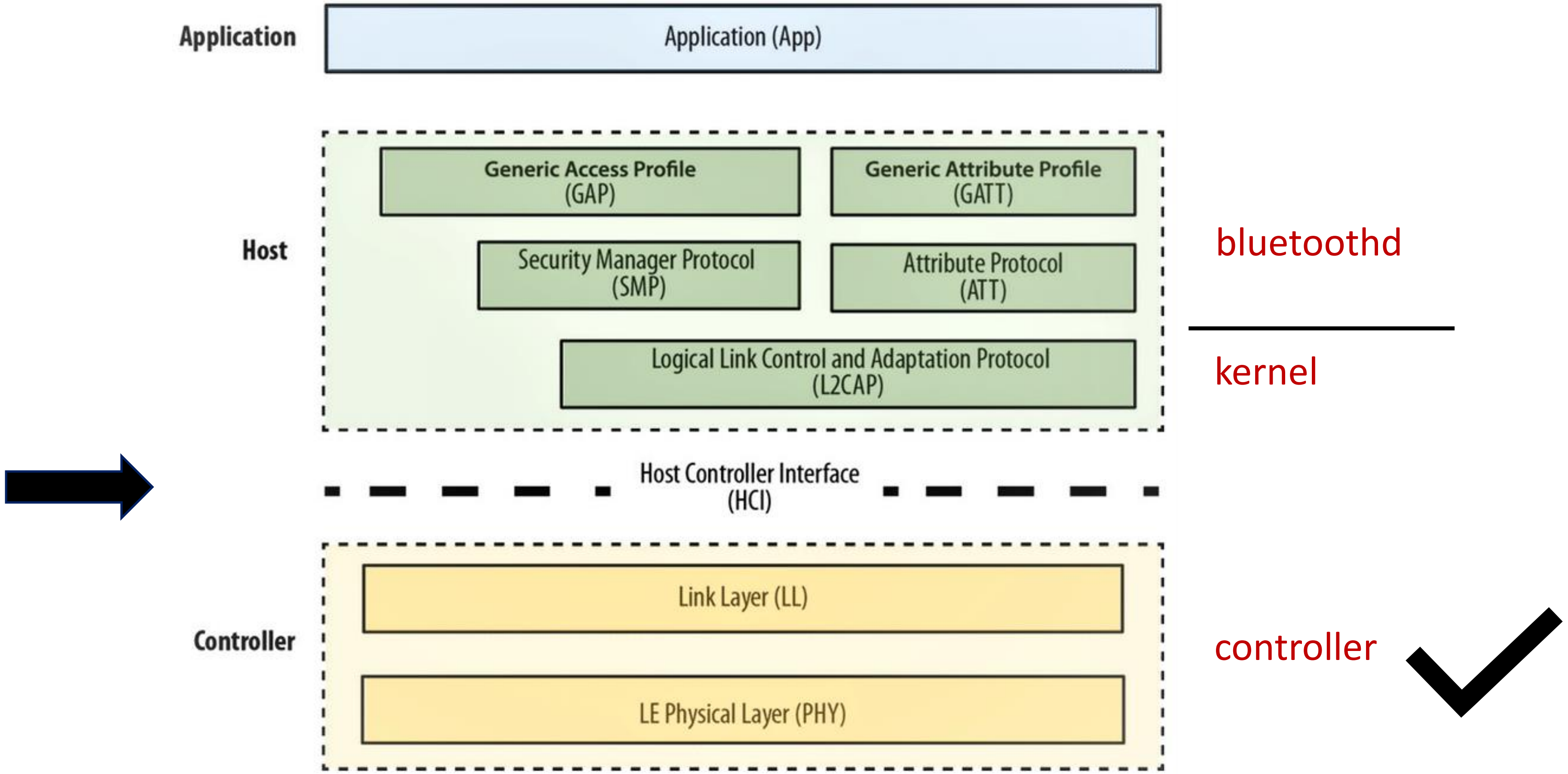
Header	Payload L2CAP	MIC
16 bits or 24 bits	0-251 octets	32 bits

Header

LLID	NESN	SN	MD	CP	RFU	Length	CTEInfo
2 bits	1 bit	1 bit	1 bit	1 bit	2 bits	8 bits	8 bits

⇨ ordered, reliable connection

LE protocol stack

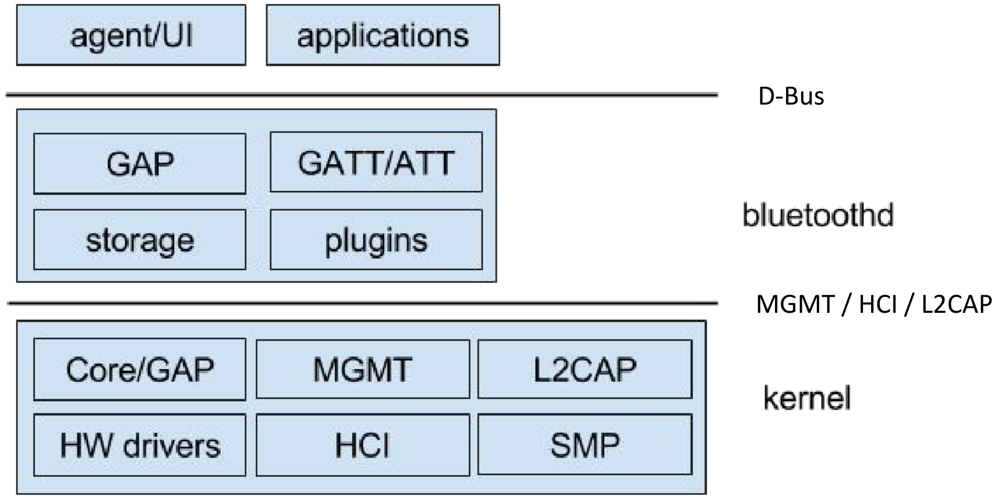


implemented in
controller firmware

⇨ would you trust it?

Host Controller Interface (HCI)

- The Host Controller Interface (HCI) is a standardized interface that enables communication between the host and the Bluetooth controller.
  - Of course, there are vendor extensions.
- HCI can use various transport protocols:
  - UART (serial), for example on Raspberry Pi
  - USB (Universal Serial Bus)
  - SPI (Serial Peripheral Interface)
- Used for exchanging both data (L2CAP) and control messages.
- Host side
  - implemented by Linux kernel: `hciX` device
  - exposed to user-space via Bluetooth management interface
  - `bluetoothd` is main user and provides access to unprivileged processes via D-Bus



Bluetooth LE advertising

- Advertising channels: 37, 38 and 39
  - Advertiser broadcasts periodically on these channels.
  - Scanner is passive and listens periodically on these channels.
- Stochastic process
  - If advertiser and scanner are on same channel simultaneously, the advertisement is received.
  - If unlucky discovery of a device can take a long time.
- Advertising data (27 bytes, extensions permit 250 bytes)
  - address, flags, device name, TX power level, etc.
  - services provided by device (e.g. thermometer), encoded as UUIDs
  - service data (e.g. temperature), limited by advertising packet size
- Handled by controller firmware, managed by `bluetoothd`

Advertising in BlueR

```rust
use bluer::adv::{self, Advertisement}; use futures::future; use std::collections::BTreeMap; use uuid::uuid;

#[tokio::main]
async fn main() -> bluer::Result<()> {
    let session = bluer::Session::new().await?;
    let adapter = session.default_adapter().await?;
    adapter.set_powered(true).await?;

    let mut service_data = BTreeMap::new();
    service_data.insert(uuid!("123e4567-e89b-12d3-a456-426614174000"), vec![0x1, 0x02, 0x03]);

    let advertisement = Advertisement {
        advertisement_type: adv::Type::Peripheral,
        service_data,
        discoverable: Some(true),
        local_name: Some("le_advertise".to_string()),
        ..Default::default()
    };
    let _handle = adapter.advertise(advertisement).await?;

    future::pending().await
}
```

Scanning in BlueR

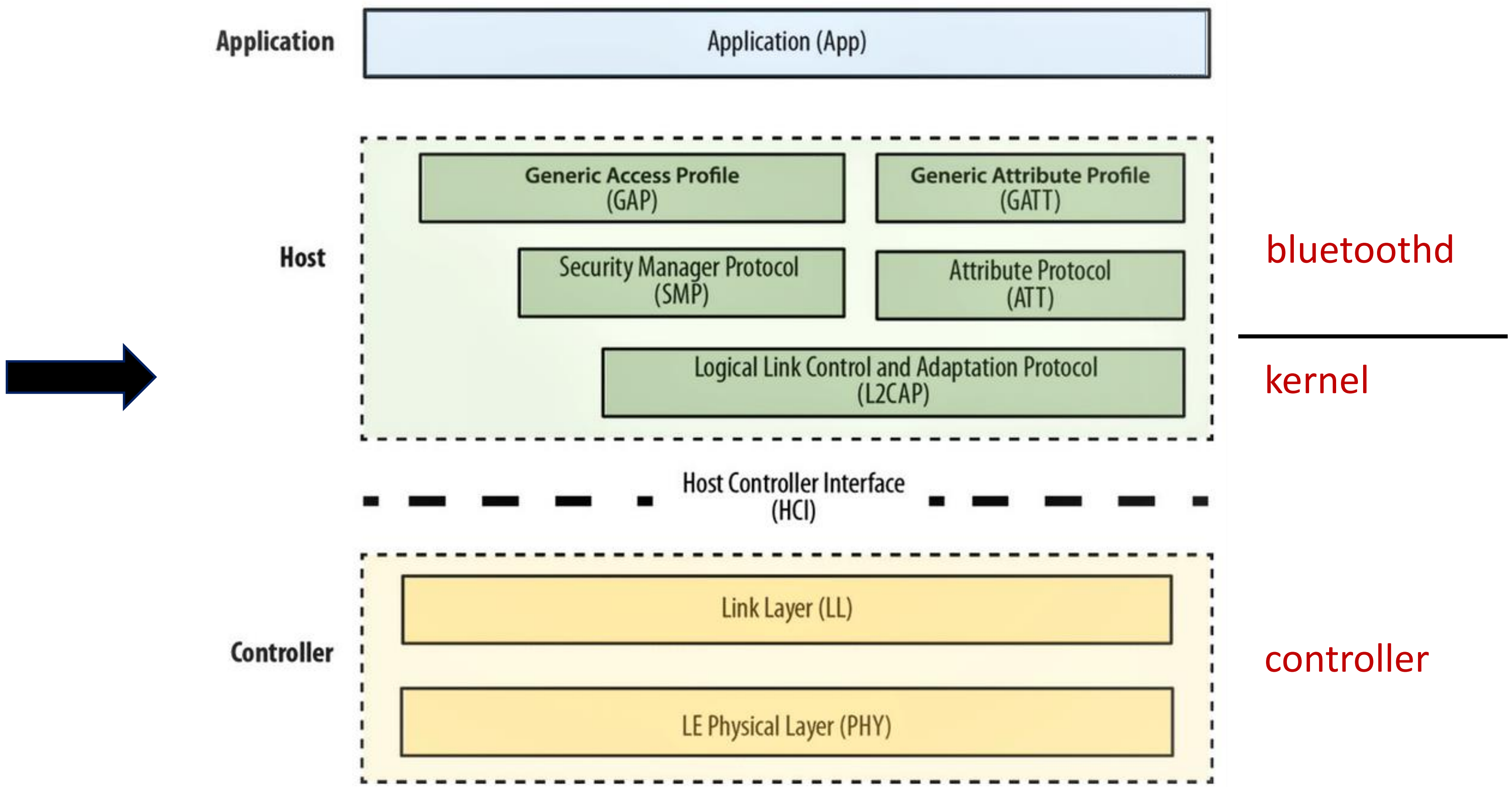

```rust
use bluer::{Adapter, AdapterEvent, Address}; use futures::{pin_mut, StreamExt};

#[tokio::main]
async fn main() -> bluer::Result<()> {
    let session = bluer::Session::new().await?;
    let adapter = session.default_adapter().await?;
    adapter.set_powered(true).await?;

    let device_events = adapter.discover_devices_with_changes().await?;
    pin_mut!(device_events);

    while let Some(device_event) = device_events.next().await {
        match device_event {
            AdapterEvent::DeviceAdded(addr) => {
                println!("Device added / changed: {addr}");
                let _ = query_device(&adapter, addr).await;
            }
            AdapterEvent::DeviceRemoved(addr) => {
                println!("Device removed: {addr}");
            }
            _ => (),
        }
    }

    Ok(())
}
```

Scanning in BlueR


```
async fn query_device(adapter: &Adapter, addr: Address) -> bluer::Result<()> {
    let device = adapter.device(addr)?;
    if device.name().await?.as_deref() != Some("le_advertise") {
        return Ok(());
    }

    println!("    Address type:       {}", device.address_type().await?);
    println!("    Name:               {:?}", device.name().await?);
    println!("    Connected:          {:?}", device.is_connected().await?);
    println!("    Paired:             {:?}", device.is_paired().await?);
    println!("    Trusted:            {:?}", device.is_trusted().await?);
    println!("    RSSI:               {:?}", device.rssi().await?);
    println!("    TX power:           {:?}", device.tx_power().await?);
    println!("    Services:           {:?}", device.uuids().await?.unwrap_or_default());
    println!("    Service data:       {:?}", device.service_data().await?);
    Ok(())
}
```

DEMO

LE protocol stack



Logical Link Control and Adaptation Protocol (L2CAP)

- Data stream multiplexing: L2CAP uses channel identifiers (CIDs) to distinguish between different data streams.
  - Connection-oriented channels (COCs): reliable, in-order data transmission
  - Connectionless channels: unreliable, unordered data transmission
- Protocol and service multiplexing: a PSM value (like TCP port number) is used to request a specific service when opening a connection.
- Segmentation and reassembly: break down and reassemble packets too large for link layer.
- Optional, credit-based flow control.
- Implemented in Linux kernel and exposed via socket interface.
  - **domain**: `AF_BLUETOOTH`, **protocol**: `BTPROTO_L2CAP`
  - `SocketAddr { addr: [u8; 6], ty: u8, psm: u16, cid: u16 }`

L2CAP client in BlueR

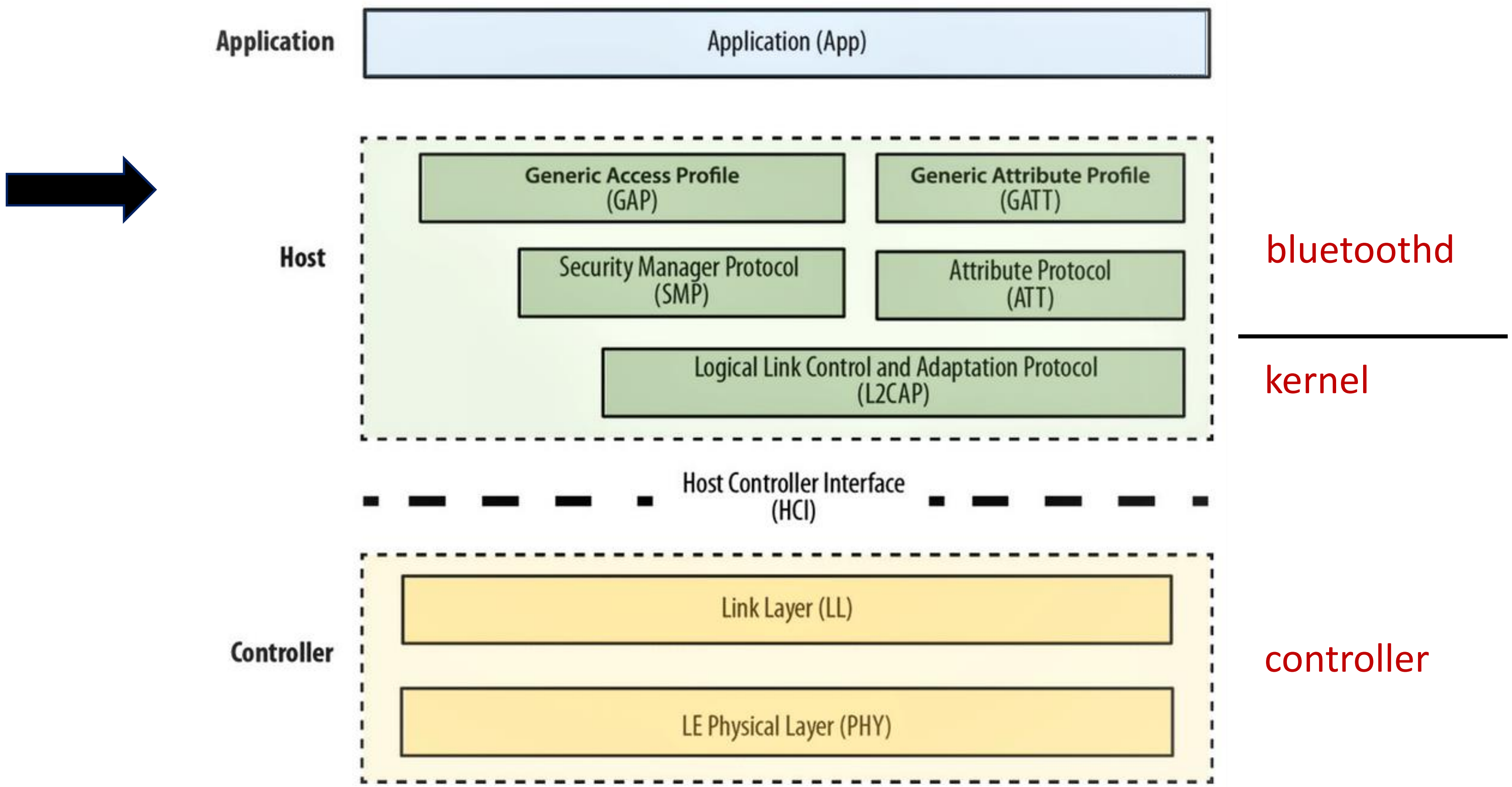
```rust
use bluer::{l2cap, Address, AddressType, Result};
use tokio::{io::AsyncWriteExt, time::sleep};

#[tokio::main]
async fn main() -> Result<()> {
    let remote = Address::new([0xe4, 0x5f, 0x01, 0x49, 0xdd, 0xd8]);
    let sa = l2cap::SocketAddr::new(remote, AddressType::LePublic, 240);

    // TODO: start discovery and wait for device

    let mut stream = l2cap::Stream::connect(sa).await?;
    sleep(std::time::Duration::from_millis(100)).await;

    stream.write_all(b"hello l2cap").await?;

    Ok(())
}
```

L2CAP server in BlueR

```
use bluer::{adv::Advertisement, l2cap::{SocketAddr, StreamListener}, Address, AddressType};
use tokio::io::AsyncReadExt;

#[tokio::main]
async fn main() -> bluer::Result<()> {
    let session = bluer::Session::new().await?;
    let adapter = session.default_adapter().await?;
    adapter.set_powered(true).await?;

    let _adv_handle = adapter.advertise(Advertisement { discoverable: Some(true), ..Default::default() }).await?;

    let sa = SocketAddr::new(Address::any(), AddressType::LePublic, 240);
    let listener = StreamListener::bind(sa).await?;

    let (mut stream, remote_sa) = listener.accept().await?;
    println!("Connection from {remote_sa:?}");

    loop {
        let mut buf = [0; 256];
        let n = stream.read(&mut buf).await?;
        if n == 0 {
            break;
        }
        println!("{}", String::from_utf8_lossy(&buf[..n]));
    }

    Ok(())
}
```

DEMO

LE protocol stack



(G)ATT: (Generic) Attribute Protocol

- ATT: Attribute Protocol
  - On LE runs over fixed L2CAP channel 4, can also be used over BR/EDR (SDP).
  - Allows reading, writing and change notifications of **attributes** consisting of
    - unique 16-bit **handle** (stable)
    - UUID defining attribute **type** (can repeat multiple times)
    - **value** of variable length
  - Can trigger LE advertising if disconnected and a notification is pending.
- GATT: Generic Attribute Profile
  - Defines how attributes are organized so that they become self-describing.

Handle	UUID (short form)	Value (decoded)	Description
0x0100	**0x2800**	UUID 0x1816	**Primary service**: thermometer
0x0101	**0x2803**	UUID 0x2A2B Value handle: 0x0102	**Characteristic**: temperature
0x0102	0x2A2B	20 degrees	Temperature value
0x0104	0x2A1F	Celsius	**Descriptor**: unit
0x0105	**0x2902**	0x0000 (off)	**Client characteristic configuration descriptor** (writable, per-client, controls notifications)
0x0110	**0x2803**	UUID 0x2A08 Value handle: 0x0111	**Characteristic**: date/time
0x0111	0x2A08	1/1/1980 12:00	Date/time value

GATT server in BlueR


```rust
use bluer::{
    adv::Advertisement, id,
    gatt::local::{
        Application, Characteristic, CharacteristicRead, CharacteristicWrite, CharacteristicWriteMethod, Service,
        CharacteristicReadRequest, CharacteristicWriteRequest, ReqResult
    },
};
use futures::{FutureExt, future};
use std::{
    sync::atomic::{AtomicBool, Ordering},
};


#[tokio::main]
async fn main() -> bluer::Result<()> {
    let session = bluer::Session::new().await?;
    let adapter = session.default_adapter().await?;
    adapter.set_powered(true).await?;

    let _adv_handle = adapter
        .advertise(Advertisement {
            local_name: Some("heater".to_string()),
            service_uuids: [id::Service::HealthThermometer.into()].into_iter().collect(),
            discoverable: Some(true),
            ..Default::default()
        })
        .await?;
```

GATT server in BlueR


```
    let app = Application {
        services: vec![Service {
            uuid: id::Service::HealthThermometer.into(), primary: true,
            characteristics: vec![
                Characteristic {
                    uuid: id::Characteristic::Temperature.into(),
                    read: Some(CharacteristicRead {
                        read: true,
                        fun: Box::new(|req| read_temperature(req).boxed()),
                        ..Default::default()
                    }), ..Default::default()
                },
                Characteristic {
                    uuid: id::Characteristic::DigitalOutput.into(),
                    write: Some(CharacteristicWrite {
                        write: true, write_without_response: true,
                        method: CharacteristicWriteMethod::Fun(Box::new(|value, req| write_heater_on(value, req).boxed())),
                        ..Default::default()
                    }), ..Default::default()
                },
            ], ..Default::default()
        }], ..Default::default()
    };
    let _app_handle = adapter.serve_gatt_application(app).await?;

    future::pending().await
}

static HEATER_ON: AtomicBool = AtomicBool::new(false);
async fn read_temperature(_req: CharacteristicReadRequest) -> ReqResult<Vec<u8>> {
    if HEATER_ON.load(Ordering::SeqCst) { Ok(vec![60]) } else { Ok(vec![20]) }
}
async fn write_heater_on(value: Vec<u8>, _req: CharacteristicWriteRequest) -> ReqResult<()> {
    let on = value[0] != 0; println!("Heater switches {}", if on { "on" } else { "off" });
    HEATER_ON.store(on, Ordering::SeqCst); Ok(())
}
```

GATT client in BlueR 1/2

```rust
use bluer::{id, AdapterEvent, Device};
use futures::{pin_mut, StreamExt};
use std::time::Duration;
use tokio::time::sleep;

#[tokio::main]
async fn main() -> bluer::Result<()> {
    let session = bluer::Session::new().await?;
    let adapter = session.default_adapter().await?;
    adapter.set_powered(true).await?;

    let device_events = adapter.discover_devices_with_changes().await?;
    pin_mut!(device_events);

    while let Some(device_event) = device_events.next().await {
        if let AdapterEvent::DeviceAdded(addr) = device_event {
            let device = adapter.device(addr)?;
            if device.uuids().await?.unwrap_or_default().contains(&id::Service::HealthThermometer.into()) {
                test_device(device).await?;
                break;
            }
        }
    }

    Ok(())
}
```

GATT client in BlueR

2/2

```rust
async fn test_device(device: Device) -> bluer::Result<()> {
    println!("Testing device {}", device.address());
    if !device.is_connected().await? {
        device.connect().await?;
    }

    for service in device.services().await? {
        if service.uuid().await? != id::Service::HealthThermometer.into() {
            continue;
        }

        let mut temperature = None;
        let mut heater_on = None;
        for char in service.characteristics().await? {
            if char.uuid().await? == id::Characteristic::Temperature.into() {
                temperature = Some(char);
            } else if char.uuid().await? == id::Characteristic::DigitalOutput.into() {
                heater_on = Some(char);
            }
        }
        let temperature = temperature.unwrap();
        let heater_on = heater_on.unwrap();

        heater_on.write(&[0]).await?;
        sleep(Duration::from_millis(100)).await;
        println!("Temperature with heater off is {:?}", temperature.read().await?);

        heater_on.write(&[1]).await?;
        sleep(Duration::from_millis(100)).await;
        println!("Temperature with heater on is {:?}", temperature.read().await?);
    }

    Ok(())
}
```

DEMO

Other functionality in BlueR

- BR/EDR functionality
  - RFCOMM sockets (virtual serial ports)
    - very similar to L2CAP sockets
  - Profile registration and discovery
- Authorization agent
  - query user for PIN during pairing
- Service id and manufacturer database

- **BlueR-Tools**: Collection of command-line tools using BlueR functionality.
  - `blumon`: real-time Bluetooth device monitor
  - `gattcat`: inspect GATT services and read/write/notify characteristics
  - `l2cat`: netcat for L2CAP sockets
  - `rfcat`: netcat for RFCOMM sockets

BlueR-Tools demo: shell over Bluetooth

- Using L2CAP PSM 240.
- Server:
  `l2cat serve -p 240 bash`
- Client:
  `l2cat connect -r E4:5F:01:49:DD:D8 240`
- Arguments
  - `-p` allocates a PTY, so that we get a normal shell prompt and ncurses work.
  - `-r` enables raw mode, so that keyboard input is passed unprocessed.

DEMO

BlueZ functionality missing in BlueR

- Useful. Please send pull requests!
  - purely passive Advertisement Monitoring API
  - Bluetooth Mesh API (PR pending)
  - Admin Policy API
- Mostly legacy
  - Battery API
  - Health API
  - Human Interface Device (HID) API
  - Media Player API
  - Bluetooth Network API
  - Object Exchange (OBEX) API
  - Thermometer API
- Low-level HCI, management interface and audio (SCO)

Summary

- Bluetooth provides setup-free, short-range radio for applications across all platforms.
  - broadcasting
  - bi-directional data exchange
- Radio communication and low-level protocols are implemented in proprietary controller firmware.
- Advertising, scanning and connections are managed by privileged `bluetoothd`, which can be controlled over D-Bus by BlueR.
- L2CAP streams are accessible via BlueR and behave like TCP streams.
- Application-level GATT is handled by `bluetoothd` and fully controllable using BlueR.