Case Study

Data Structures in Wearable Health Devices, Pacemakers, and DSP Applications

1. Introduction

Modern wearable health devices, pacemakers, and digital signal processing (DSP) systems rely heavily on **efficient data management**. Real-time monitoring, signal filtering, and data logging require careful selection of **data structures** to ensure **speed, memory efficiency, and reliability**.

Data structures are used to **store, access, and process sensor data** efficiently. Choosing the right data structure can mean the difference between a device that performs in real-time and one that fails to deliver accurate results.

Key Areas Covered in This Case Study:

1. Pacemaker data buffering
2. Data structures in wearable health device firmware
3. DSP filter coefficient management

2. Pacemaker Data Buffering

2.1 Overview

Pacemakers are implantable devices that monitor a patient’s heart rhythm and provide electrical pulses to maintain a regular heartbeat. These devices must **store recent heart activity data** to detect irregularities and ensure proper pacing.

2.2 Data Structures Used

1. **Circular Buffer (Ring Buffer)**
   - Stores incoming heart rate readings temporarily.

- When the buffer is full, new data **overwrites the oldest data**.
- Ensures memory is **used efficiently**, preventing overflow.

2. **Queue**
   - Can be used for event logging (e.g., arrhythmia detection).
   - FIFO (First In First Out) ensures **earliest events are processed first**.
3. **Structs**
   - Group related data (e.g., timestamp, heart rate, pulse amplitude).
   - Makes processing more organized.

2.3 Implementation Example (Pseudocode)

```
#define BUFFER_SIZE 100

typedef struct {
    int heartRate;
    int pulseAmplitude;
    char timestamp[20];
} HeartData;

HeartData buffer[BUFFER_SIZE];
int start = 0, end = 0;

void insertData(HeartData data) {
    buffer[end] = data;
    end = (end + 1) % BUFFER_SIZE;
    if (end == start) {
        start = (start + 1) % BUFFER_SIZE; // overwrite oldest
    }
}

HeartData readData() {
    HeartData data = buffer[start];
```

```
    start = (start + 1) % BUFFER_SIZE;
    return data;
}
```

2.4 Advantages

- Efficient memory usage
- Real-time data processing
- Prevents data loss for latest readings

2.5 Limitations

- Limited buffer size
- Older data is lost when buffer overflows

3. Data Structures in Firmware for Wearable Health Devices

3.1 Overview

Wearable devices like smartwatches and fitness trackers monitor multiple parameters (heart rate, steps, oxygen level) and process data in **real-time**. Firmware must efficiently store, access, and process this continuous data stream.

3.2 Common Data Structures

1. **Array**
    - Stores sequential sensor readings (heart rate, steps, SpO2).
    - Allows **fast indexed access**.
2. **Queue / Circular Buffer**
    - Temporarily holds **streaming sensor data** for processing or display.
3. **Linked List**
    - Useful for **event-driven logging**, e.g., sudden heartbeat spikes.
    - Allows **dynamic insertion and deletion** of events.

4. **Structs**
    - Organizes related sensor data into a **single unit**:

```
typedef struct {
    int heartRate;
    int steps;
    float oxygenLevel;
    char timestamp[20];
} SensorData;
```

3.3 Real-Time Processing

- Sensor data must be **processed without delay**.
- Using **efficient data structures**, devices perform **calculations, filtering, and event detection** quickly.

3.4 Advantages

- Fast access to sensor data
- Low memory usage for continuous monitoring
- Easy to manage dynamic events

3.5 Limitations

- Limited memory on wearable devices
- Must balance **speed vs memory efficiency**

4. DSP Filter Coefficient Handling Using Data Structures

4.1 Overview

Digital Signal Processing (DSP) filters are used in wearable devices to **filter noise** and extract meaningful data (like heart signals). Filters use **coefficients** to perform calculations on signal samples.

4.2 Data Structures Used

1. **Array / Matrix**
   - Stores **filter coefficients**.
   - Supports **efficient arithmetic operations** on signals.
2. **Circular Buffer**
   - Maintains **recent input samples** needed for convolution or filtering.
   - Ensures **oldest samples are automatically replaced**.
3. **Structs**
   - Organize filter parameters: coefficients, order, states, etc.

4.3 Example: FIR Filter Coefficient Handling

```
#define N 5
float coeff[N] = {0.1, 0.15, 0.5, 0.15, 0.1};
float buffer[N] = {0};

float applyFilter(float newSample) {
    for (int i = N-1; i > 0; i--)
        buffer[i] = buffer[i-1];
    buffer[0] = newSample;

    float result = 0;
    for (int i = 0; i < N; i++)
        result += coeff[i] * buffer[i];
    return result;
```

```
}
```

4.4 Advantages

- Efficient signal processing
- Minimal memory usage
- Supports **real-time calculations**

4.5 Limitations

- Filter size increases memory usage
- Complex filters may require **more CPU cycles**

Comparison Table of Data Structures in These Applications

Application	Data Structure Used	Purpose / Benefit
Pacemaker Data Buffering	Circular Buffer, Queue, Struct	Real-time heart data storage, overwrite safely
Wearable Health Device Firmware	Array, Queue, Linked List, Struct	Manage sensor data, dynamic event logging
DSP Filter Coefficient Handling	Array, Circular Buffer, Struct	Store coefficients, manage recent samples

- Data structures play a **critical role in embedded systems and wearable devices**.
- **Circular buffers, arrays, linked lists, and structs** are widely used for **efficient memory and real-time processing**.
- Proper selection of data structures ensures:
  - Real-time performance

  - Memory efficiency
  - Accuracy and reliability in health monitoring devices

Future Scope:

- Integration with **AI algorithms** for predictive health monitoring
- Advanced **dynamic memory management** for more complex wearable devices