工作原理：以太网帧 + RTC 同步

以太网帧结构

通用以太网帧 (Ethernet Frame) 结构图 (简化)

目的

目的地址 (Destination Address)

目的地址，通常是一个唯一的 MAC 地址，指定了数据帧应该发送到网络上的哪个特定设备。这个地址通常是硬件厂商分配给网络接口卡 (NIC) 的全球唯一标识符。

例如，一个 LED 灯带，它的 ESP32 控制器可能会分配一个 MAC 地址 `AA:BB:CC:DD:EE:01`。当一个运行在 Linux 上的 Python 脚本通过 `sendp()` 函数发送一个目的地址是 `AA:BB:CC:DD:EE:01` 的帧时，网络上的所有设备都会接收到这个帧，但只有 ESP32 控制器的网卡会识别出这个地址是发给它的，并将帧内容交给其固件处理。

这个地址是实现点对点通信的基础，它确保了数据帧能够精确地到达预期的接收方，而不会干扰网络上其他设备的正常工作。

源地址

源地址 (Source Address)

指定了数据帧的发送方。

例如，在上面的 LED 灯带例子中，运行 Python 脚本的 Linux 主机的网卡 MAC 地址可能是 `11:22:33:44:55:66`。

时间戳的载荷

时间戳的载荷

在这一个以太网帧中，主要的信息是时间戳。这个时间戳通常是一个 64 位的数值，表示从某个参考点（例如 1970 年 1 月 1 日 00:00:00 UTC）开始经过的纳秒数。

这个时间戳被放在以太网帧的数据部分 (Payload) 中。当 Python 脚本通过 `sendp()` 发送这个帧时，它会将这个时间戳作为数据包的内容。接收方的 ESP32 控制器在接收到这个帧后，会从帧的数据部分提取出这个时间戳，并用它来校准其内部的实时时钟 (RTC)，以实现精确的时间同步。

在这一个以太网帧中，主要的信息是时间戳。这个时间戳通常是一个 64 位的数值，表示从某个参考点（例如 1970 年 1 月 1 日 00:00:00 UTC）开始经过的纳秒数。

这个时间戳被放在以太网帧的数据部分 (Payload) 中。当 Python 脚本通过 `sendp()` 发送这个帧时，它会将这个时间戳作为数据包的内容。接收方的 ESP32 控制器在接收到这个帧后，会从帧的数据部分提取出这个时间戳，并用它来校准其内部的实时时钟 (RTC)，以实现精确的时间同步。

以太网帧 vs 传统数据包

以太网帧 vs 传统数据包 (IP/UDP)

	以太网帧	传统数据包 (IP/UDP)
目的地址	目标 MAC 地址	目标 IP 地址
源地址	源 MAC 地址	源 IP 地址
时间戳	时间戳	时间戳
数据	数据	数据
FCS	FCS	FCS

在这一个以太网帧中，主要的信息是时间戳。这个时间戳通常是一个 64 位的数值，表示从某个参考点（例如 1970 年 1 月 1 日 00:00:00 UTC）开始经过的纳秒数。

这个时间戳被放在以太网帧的数据部分 (Payload) 中。当 Python 脚本通过 `sendp()` 发送这个帧时，它会将这个时间戳作为数据包的内容。接收方的 ESP32 控制器在接收到这个帧后，会从帧的数据部分提取出这个时间戳，并用它来校准其内部的实时时钟 (RTC)，以实现精确的时间同步。

工作流程图解说明

在这一个以太网帧中，主要的信息是时间戳。这个时间戳通常是一个 64 位的数值，表示从某个参考点（例如 1970 年 1 月 1 日 00:00:00 UTC）开始经过的纳秒数。

这个时间戳被放在以太网帧的数据部分 (Payload) 中。当 Python 脚本通过 `sendp()` 发送这个帧时，它会将这个时间戳作为数据包的内容。接收方的 ESP32 控制器在接收到这个帧后，会从帧的数据部分提取出这个时间戳，并用它来校准其内部的实时时钟 (RTC)，以实现精确的时间同步。

步骤一：生成时间戳并发送

步骤一：生成时间戳并发送

Python 脚本

```
# 伪代码 - 运行在 Linux 上
import time
import scapy.all as scapy

# 1. 生成时间戳 (纳秒)
timestamp = int(time.time_ns())

# 2. 构造以太网帧
# 目的地址是 ESP32 的 MAC 地址
# 源地址是 Linux 网卡的 MAC 地址
# 数据部分是时间戳
frame = Ether(dst="AA:BB:CC:DD:EE:01", src="11:22:33:44:55:66") / Raw(load=timestamp)

# 3. 发送帧
sendp(frame, iface="eth0")
```

Linux 网卡

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 Python 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 ESP32 控制器有数据包到达
```

ESP32 网卡

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

ESP32 控制器

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

RTC 时钟

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

步骤二：接收时间戳并校准 RTC

步骤二：接收时间戳并校准 RTC

Linux 网卡

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 ESP32 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 Python 脚本有数据包到达
```

Python 脚本

```
# 伪代码 - 运行在 Linux 上
# 1. 接收到来自 ESP32 的帧
# 2. 解析帧，提取时间戳
# 3. 将时间戳写入 RTC 寄存器
# 4. 校准 RTC 时钟
```

ESP32 网卡

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

ESP32 控制器

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

RTC 时钟

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

步骤三：同步完成

步骤三：同步完成

Linux 网卡

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 ESP32 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 Python 脚本有数据包到达
```

Python 脚本

```
# 伪代码 - 运行在 Linux 上
# 1. 接收到来自 ESP32 的帧
# 2. 解析帧，提取时间戳
# 3. 将时间戳写入 RTC 寄存器
# 4. 校准 RTC 时钟
```

ESP32 网卡

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

ESP32 控制器

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

RTC 时钟

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

总结图解

总结图解

```
# 伪代码 - 运行在 Linux 上
import time
import scapy.all as scapy

# 1. 生成时间戳 (纳秒)
timestamp = int(time.time_ns())

# 2. 构造以太网帧
# 目的地址是 ESP32 的 MAC 地址
# 源地址是 Linux 网卡的 MAC 地址
# 数据部分是时间戳
frame = Ether(dst="AA:BB:CC:DD:EE:01", src="11:22:33:44:55:66") / Raw(load=timestamp)

# 3. 发送帧
sendp(frame, iface="eth0")
```

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 Python 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 ESP32 控制器有数据包到达
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 ESP32 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 Python 脚本有数据包到达
```

```
# 伪代码 - 运行在 Linux 上
# 1. 接收到来自 ESP32 的帧
# 2. 解析帧，提取时间戳
# 3. 将时间戳写入 RTC 寄存器
# 4. 校准 RTC 时钟
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

```
# 伪代码 - 运行在 Linux 内核
# 1. 接收到来自 ESP32 的帧
# 2. 检查目的地址是否匹配本机 MAC 地址
# 3. 如果匹配，将帧内容复制到内核缓冲区
# 4. 通知 Python 脚本有数据包到达
```

```
# 伪代码 - 运行在 Linux 上
# 1. 接收到来自 ESP32 的帧
# 2. 解析帧，提取时间戳
# 3. 将时间戳写入 RTC 寄存器
# 4. 校准 RTC 时钟
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 接收到来自 Linux 网卡的帧
// 2. 检查目的地址是否匹配本机 MAC 地址
// 3. 如果匹配，将帧内容复制到内存缓冲区
// 4. 通知 ESP32 控制器有数据包到达
```

```
// 伪代码 - 运行在 ESP32 固件
// 1. 从内存缓冲区读取帧
// 2. 解析帧，提取时间戳
// 3. 将时间戳写入 RTC 寄存器
// 4. 校准 RTC 时钟
```

```
// 伪代码 - 运行在 ESP32 硬件
// 1. 接收到来自 ESP32 控制器的时间戳
// 2. 将时间戳作为新的时间基准
// 3. 开始计时
```

总结图解

总结图解