デモ

- SchemeインタプリタのREPLから、Wiiリモコンのボタン状態を読み取る
  - car : Wiimote-button
  - cdr : 押されているボタンのリスト

デモ

- 従来、ホストOSのAPIを呼び出して行っていたようなデバイスの制御が、S式の読み書きだけでできるようになった。
- インターフェースを知らなくてもopen/readでなんとなくわかる
- センサの読み取りや、モータの制御などへの応用を期待している。
  - `(read: port)`

    `=> (temperature 23)`；気温23度

ターゲット層

e.g.)
フィジカル-
コンピューティング

- ほんのちょっとのコードで、面白いことが出来る
  - Wiiリモコン鍵くらいならREPL上で
- 目の前に有るデバイスを繋ぐためのWeb-APIくらい手軽な方法。

Overview

本発表の内容：

- 人間にとって読みやすいS式から、パケットを適当に補完して構築する方法論を検討した。
  - プリミティブの検討(TLV, 構造体, 辞書, ...)
  - 処理系の実装
- これを実現するために、パケットフォーマットを記述するための言語を提案する。
- 現実のプロトコル(TCP/IP)を実装し、パフォーマンスを評価した。

agenda

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

agenda

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

本研究の動機

ユーザモードアプリケーション
によるデバイスの制御

例題:『PCにマウスを5台繋げたい』

解：
『OSのマウス関連APIを使わず、アプリケーションから直接USBパケットを発行して読み取る』

本発表のテーマ

ユーザモードアプリケーション
によるデバイスの制御

New!!

↑
↓

？

本発表のテーマ

ユーザモードアプリケーション
によるデバイスの制御

New!!

↑
↓

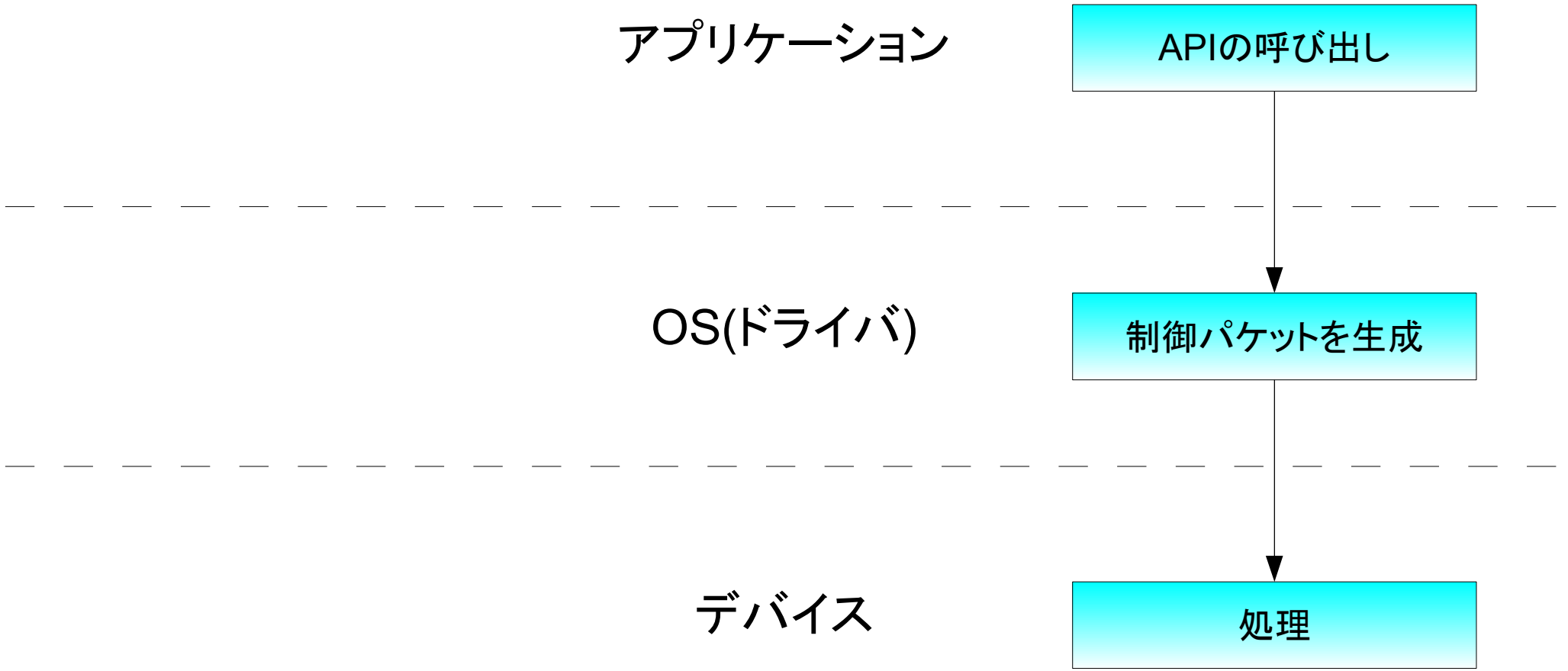
デバイスドライバを書いてデバイスを使う
(OSのAPIによる抽象化)

boring..

新旧手法の比較

New!!

boring..



新旧手法の比較

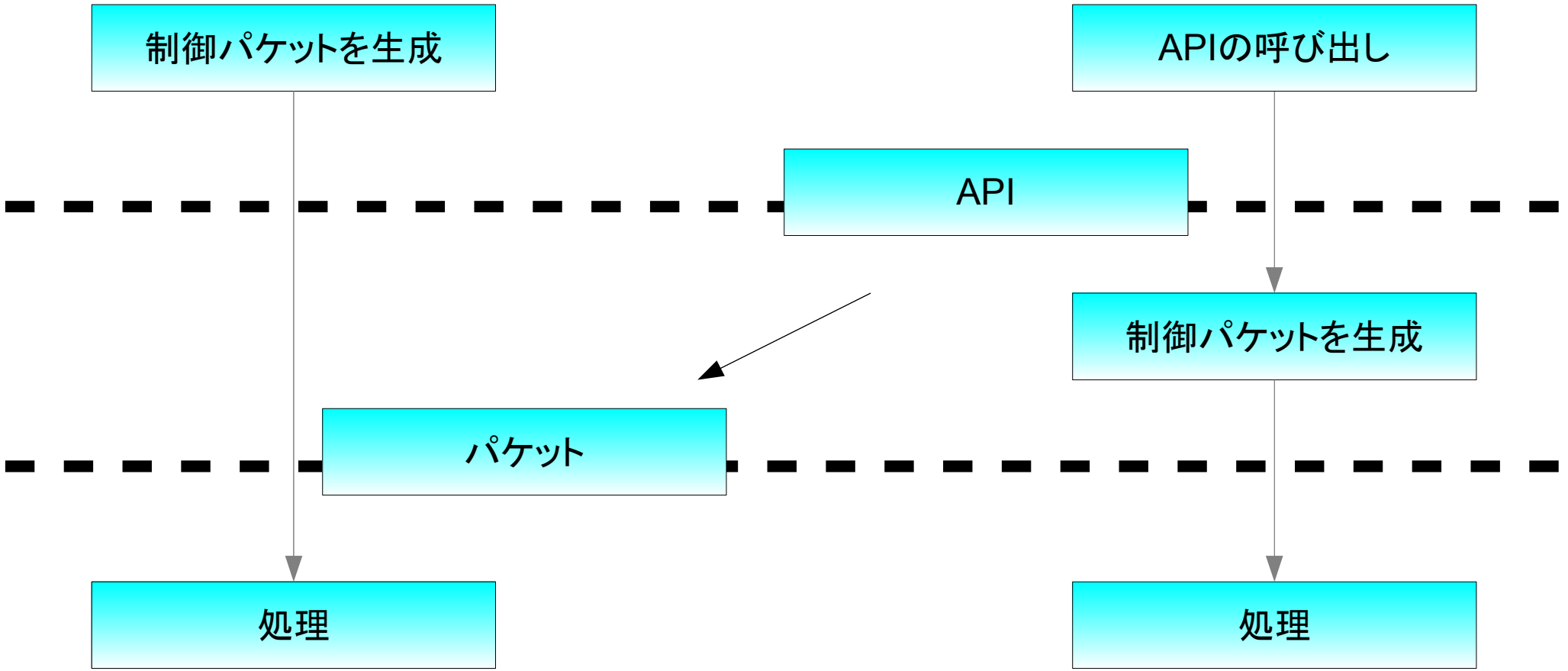
New!!

boring..

新旧手法の比較

New!!

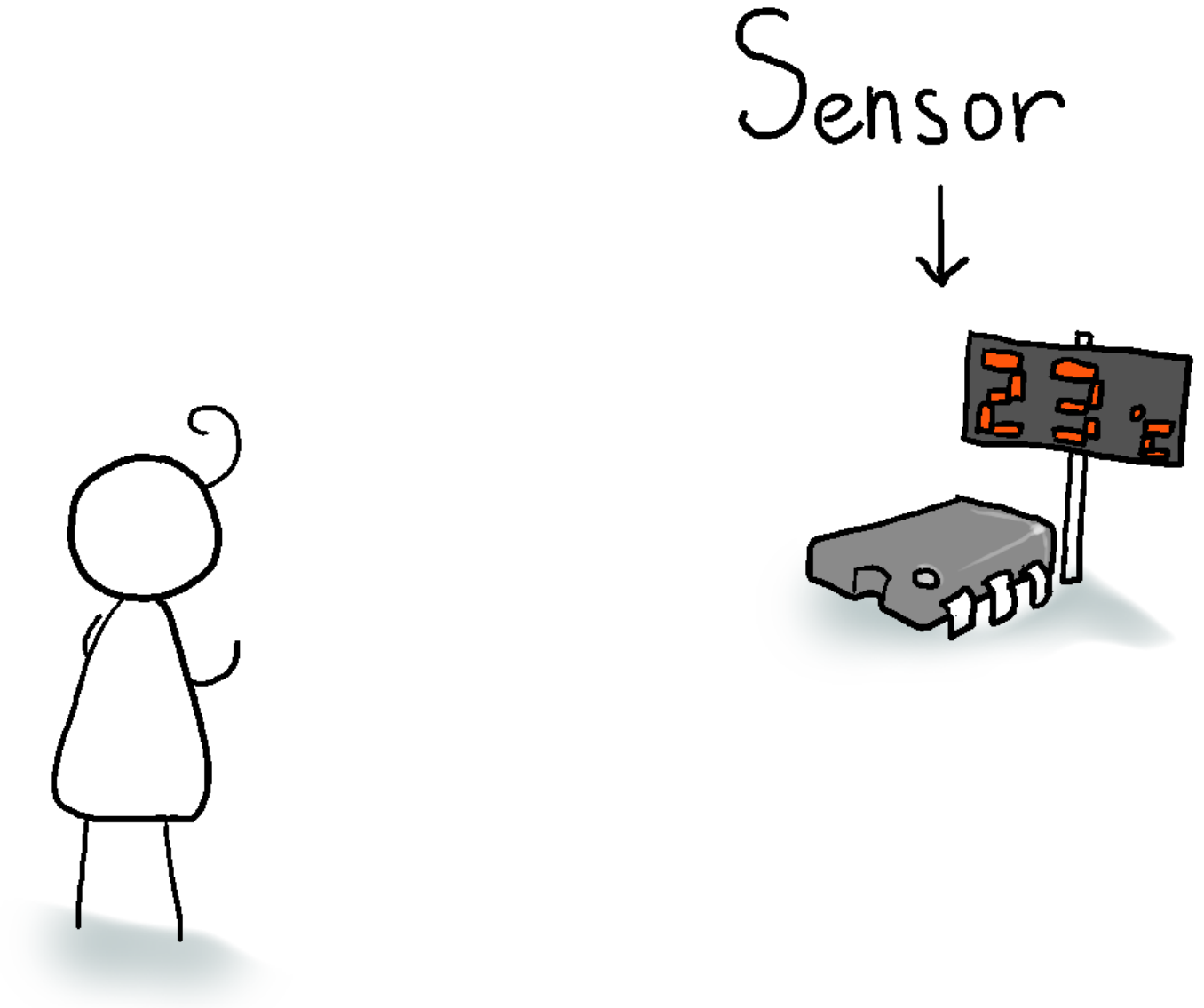
boring..



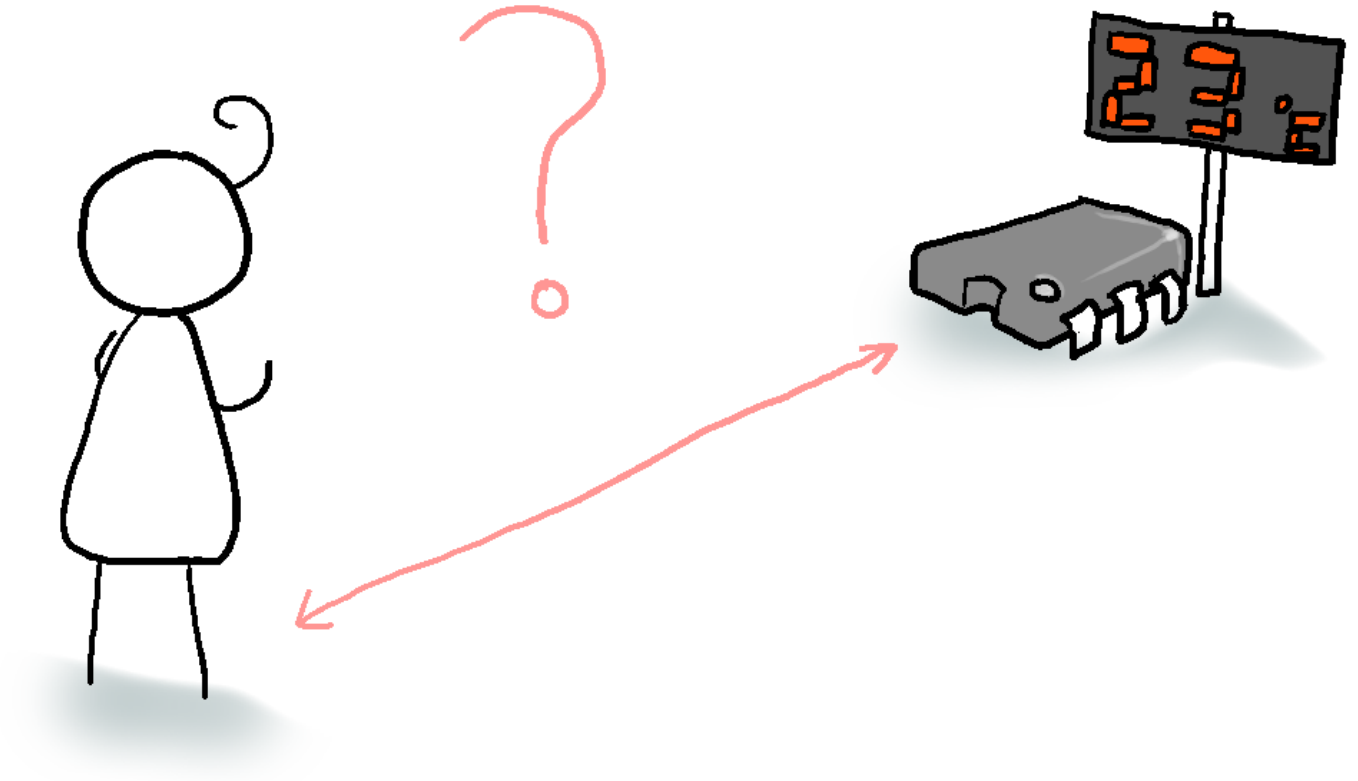
Why is API abstraction so boring?

→ 新しいことをする際に障害になる

Why is API abstraction so boring?



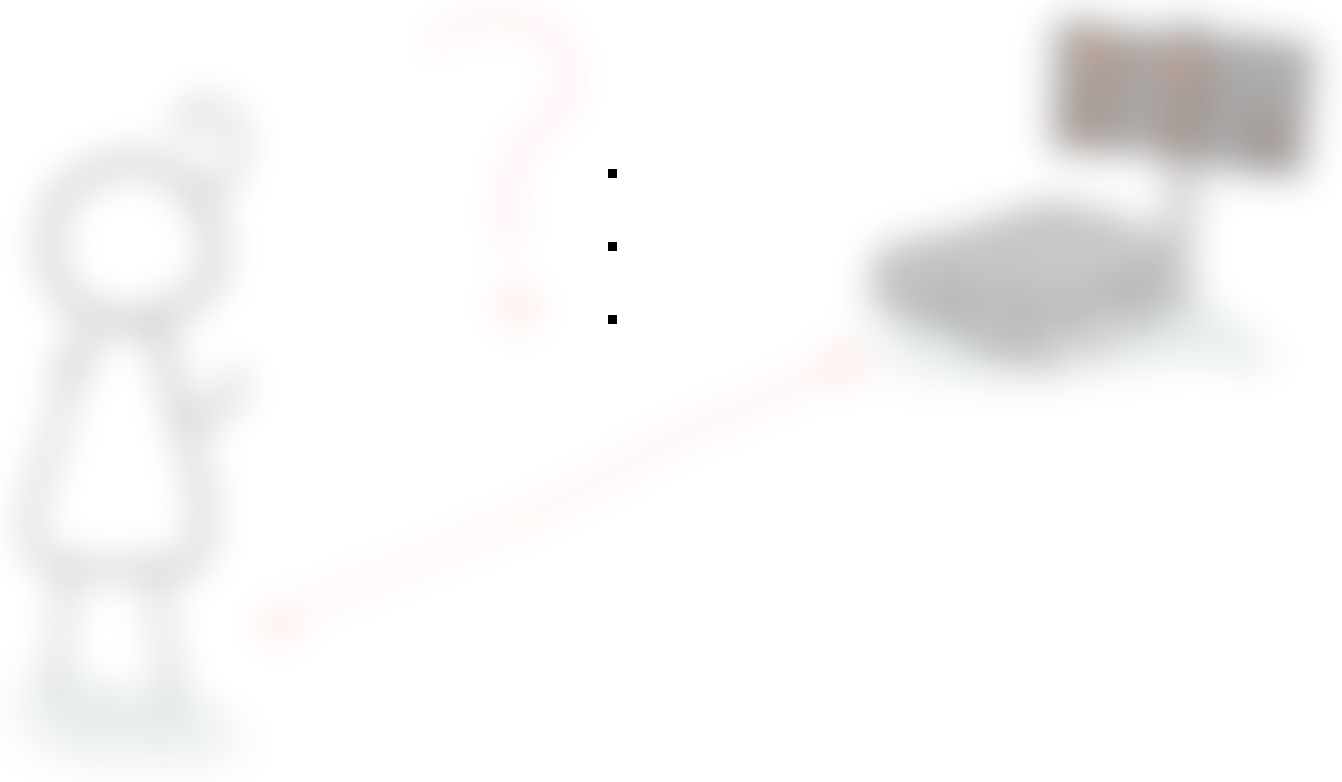
Why is API abstraction so boring?



Why is API abstraction so boring?

OSのAPIを使う場合

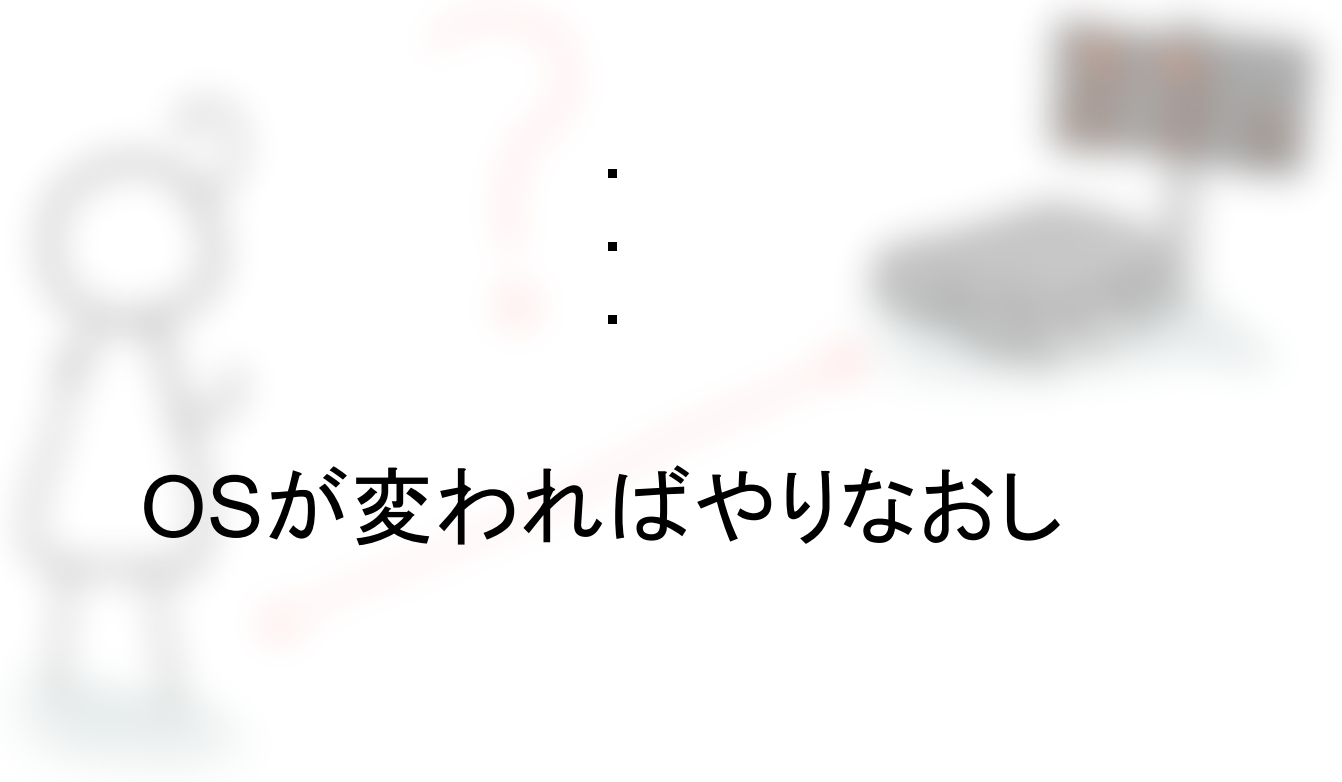
- OSに対応したデバイスドライバを書く
- デバイスを制御するAPIを学習する
  - (デバイスそのものではなく)



Why is API abstraction so boring?

OSのAPIを使う場合

- OSに対応したデバイスドライバを書く
- デバイスを制御するAPIを学習する
  - (デバイスそのものではなく)



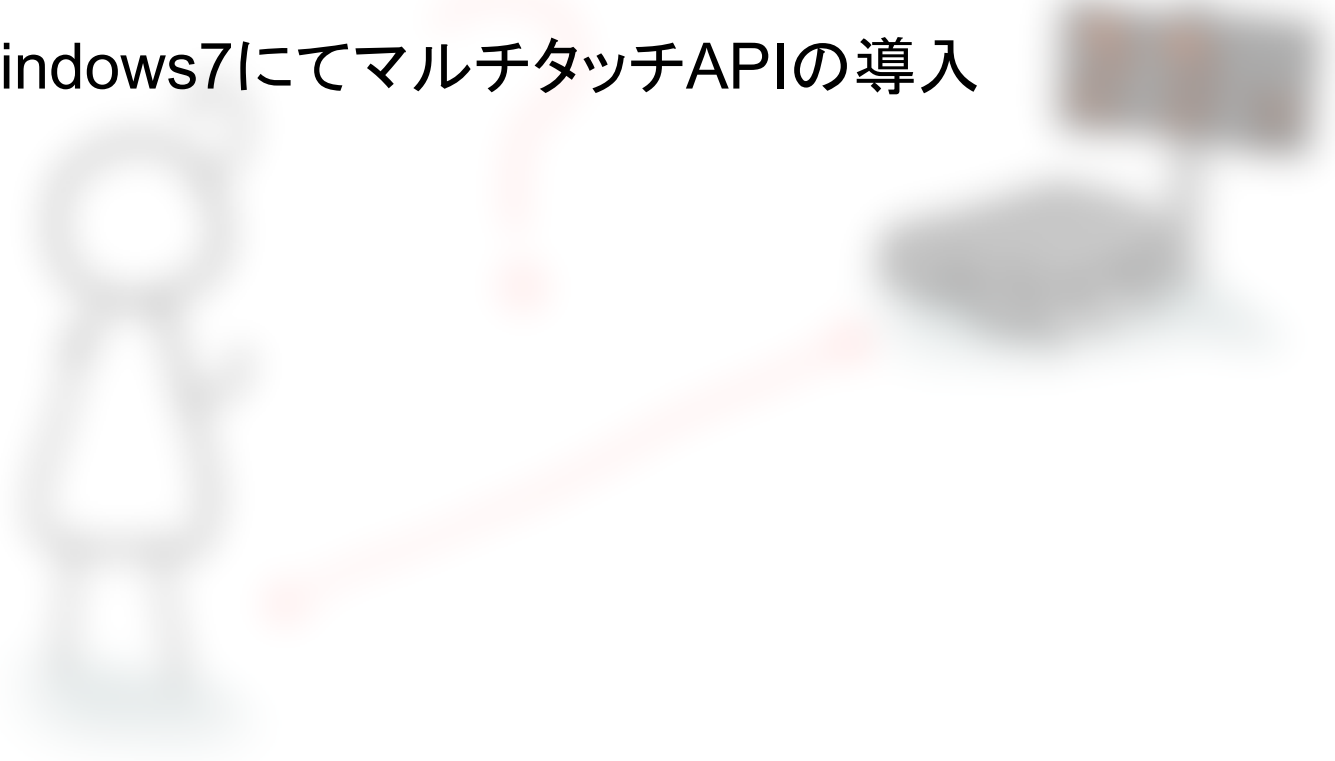
.
.
.

OSが変わればやりなおし

Why is API abstraction so boring?
OSのAPIを使う場合

e.g.)

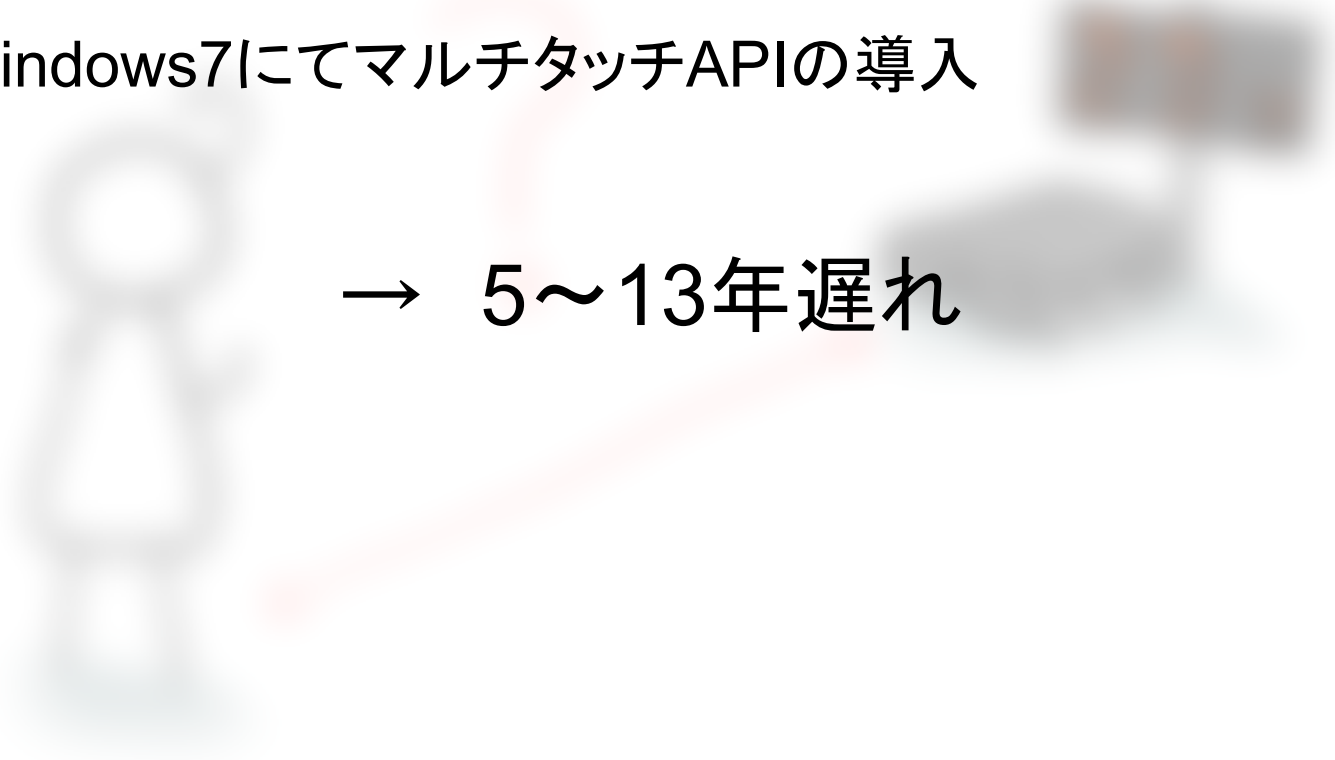
1996 - USBを使ってコンピュータにマウスを複数接続できるようになる
2001 - WindowsXPで、複数のマウスの読み取りに対応
2009 - Windows7にてマルチタッチAPIの導入



Why is API abstraction so boring?
OSのAPIを使う場合

e.g.)

1996 - USBを使ってコンピュータにマウスを複数接続できるようになる
2001 - WindowsXPで、複数のマウスの読み取りに対応
2009 - Windows7にてマルチタッチAPIの導入



→ 5～13年遅れ

Why is API abstraction so boring?
OSのAPIを使う場合

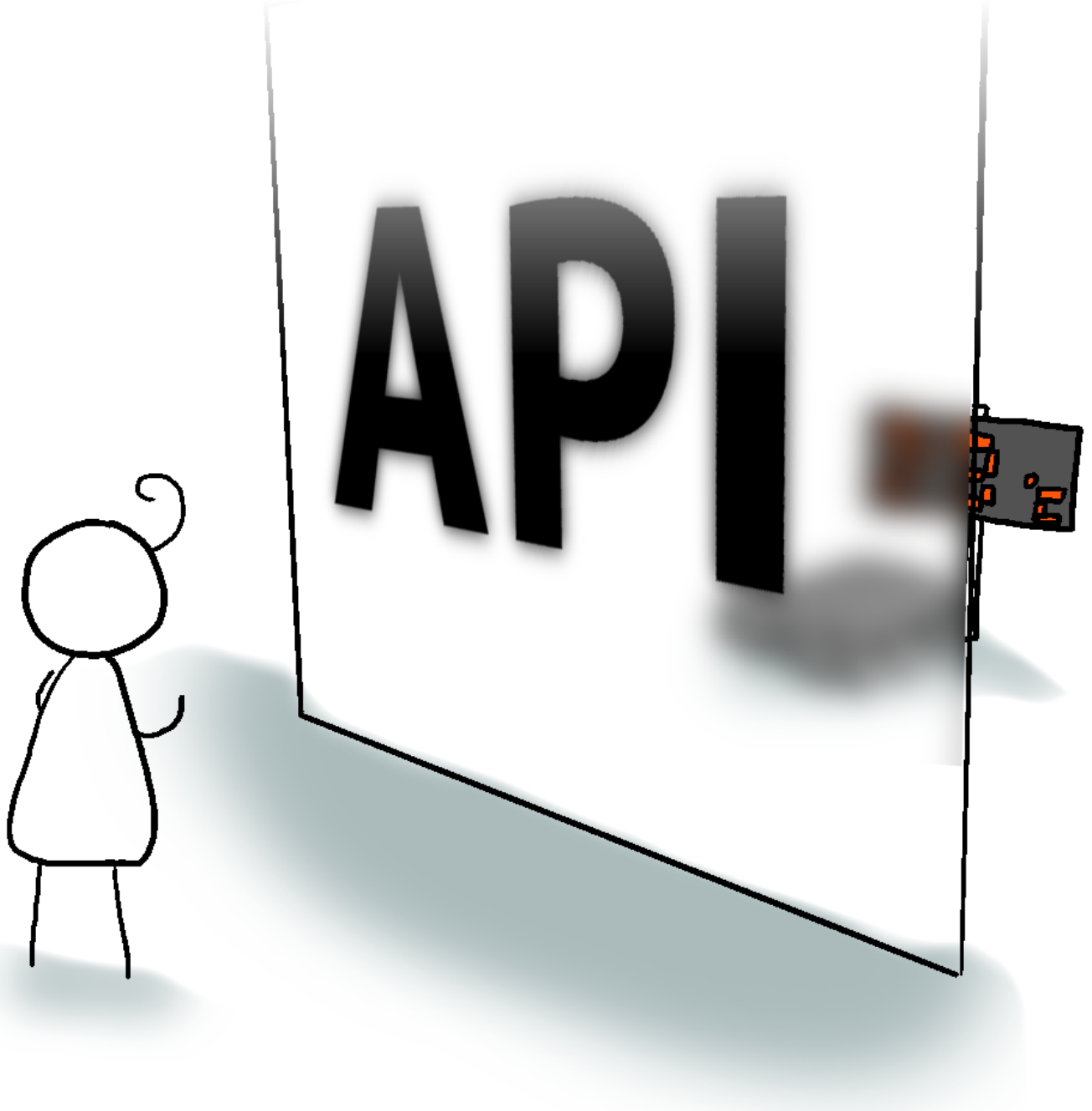
e.g.)

1996 - USBを使ってコンピュータにマウスを複数接続できるようになる
2001 - WindowsXPで、複数のマウスの読み取りに対応
2009 - Windows7にてマルチタッチAPIの導入

→ 5～13年遅れ

アプリケーションの進化はOSのAPIに規定される

Why is API abstraction so boring?



まとめ

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

OSのAPIよりも自由にプログラミングできる

agenda

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

提案する開発手法

特徴：

- 仕様記述言語によるコンポーネントの共通化
- SchemeとS式を利用
- 現実的なプロトコルを広範に記述可能
  - SCSI, TCP/IP, Bluetooth, USB, …

提案する開発手法

特徴：

- 仕様記述言語によるコンポーネントの共通化
- SchemeとS式を利用
- 現実的なプロトコルを広範に記述可能
  - SCSI, TCP/IP, Bluetooth, USB, …

→ デバイスの制御はS式の読み書きと同程度に簡単に

提案する開発手法

特徴：

- 仕様記述言語によるコンポーネントの共通化
- SchemeとS式を利用
- 現実的なプロトコルを広範に記述可能
  - SCSI, TCP/IP, Bluetooth, USB, …

→ デバイスの制御はS式の読み書きと同程度に簡単に
(≒XML)

→ Web-APIと同程度に簡単に

提案する開発手法

要約：

- APIはOpen/Read/Writeだけ
- デバイス固有の情報はS式で与える

→ cf.) HTTP RESTful API (POST/GET)

API v.s. Web-API

APIを学ぶ手法は静的なものしかない
(マニュアルを読む、ソースコードを参照する)

↑
↓

？

API v.s. Web-API

APIを学ぶ手法は静的なものしかない
(マニュアルを読む、ソースコードを参照する)

Web-APIは動いているものを見て学ぶことが出来る。

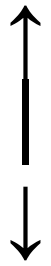
API v.s. Web-API

APIを学ぶ手法は静的なものしかない
(マニュアルを読む、ソースコードを参照する)

Web-APIは動いているものを見て学ぶことが出来る。
→　本研究はこの特徴をより多くの場所に提案したい

デバイス v.s. Web-API

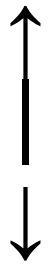
WebAPI　＝　人間に読めるデータを扱う
cf.) HTTP Query



デバイスのパケットは人間に読めない

デバイス v.s. Web-API

WebAPI　＝　人間に読めるデータを扱う
cf.) HTTP Query



デバイスのパケットは人間に読めない
→ プロトコル記述言語

先行研究

- “プロトコル記述言語”が研究されている。
- プロトコルを記述/実装するためのDSL
  - コンポーネントの共通化
  - パケットフォーマットの記述

- (しかし、本研究の領域をカバーできるものは知られていない)

先行研究

- プロトコル実装用
  - Prolac [Kohler99]
  - Preccs [服部05]
- 規格定義/実装用
  - ASN.1 BER/DER
- そのほか構造体定義
  - Hachoir
  - Binspect
  - Binpac [pang06]

先行研究

- プロトコル実装用
  - Prolac [Kohler99]
  - Preccs [服部05]
- 規格定義/実装用
  - ASN.1 BER/DER
- そのほか構造体定義
  - Hachoir
  - Binspect
  - Binpac [pang06]

Yaccのようなスタイルで、パーサとC言語部分、CSPによる制御部分を混合して記述する。

→ TCP/IPネットワークに特化

本研究のように、デバイス制御に適用可能なものは知られていない。

先行研究との違い

- トランスポート中立

  Preccsは、TCP/IPに依存している。

  本研究は任意のpacketベースのトランスポートをサポートする。

- 言語中立

  Preccsは、C言語に依存している。

- 多くのプリミティブ

  Preccsは、正規表現による一般的な表現をサポートしている。

  本研究はより多くのプリミティブを定義することで、より静的にコンパイルできる可能性が有る。

まとめ

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

Web-APIの特性をデバイスの制御にも提供

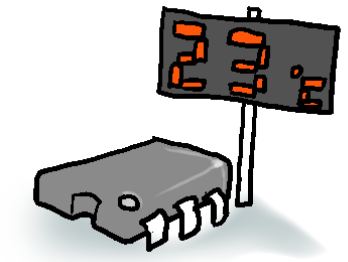
実現のためには独自の言語が必要

agenda

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

パケットとS式の相互変換

処理のイメージ：



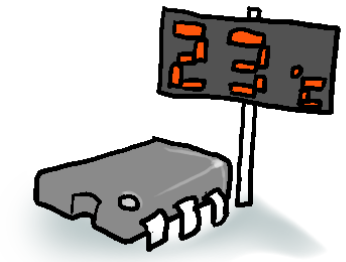
```
01 23 00 00 0e ec
```

↑

センサから送信されてくるパケット

パケットとS式の相互変換

処理のイメージ：



```
01 23 00 00 0e ec
```

↑

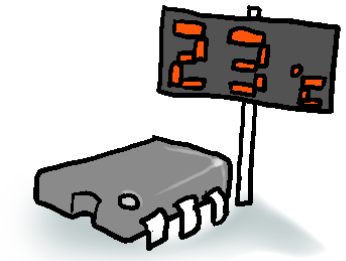
センサから送信されてくるパケット

デバイスからのデータは数値の羅列

→ そのままでは扱いづらい

パケットとS式の相互変換

処理のイメージ：



```
01 23 00 00 0e ec
```

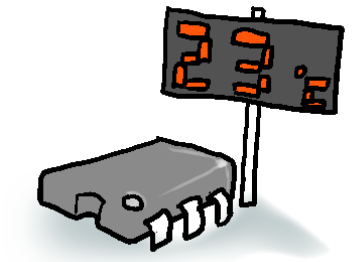
↓

```
(temp 23)
```

→ フォーマットの定義により、
人間に可読なS式に変換する

パケットとS式の相互変換

フォーマットの定義(C言語) :

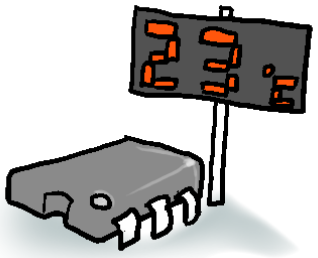


```
01 23 00 00 0e ec
```

```
struct packet {
   char        sensor_id;
   char        value_bcd[3];
   short int   crc16;
} __attribute__((packed));
```

パケットとS式の相互変換

フォーマットの定義(C言語) :

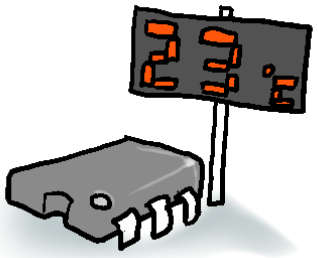


```
01 23 00 00 0e ec
```

```
struct packet {
  char        sensor_id;
  char        value_bcd[3];
  short int   crc16;
} __attribute__((packed));
```

パケットとS式の相互変換

フォーマットの定義(C言語)：



```
01 23 00 00 0e ec
```

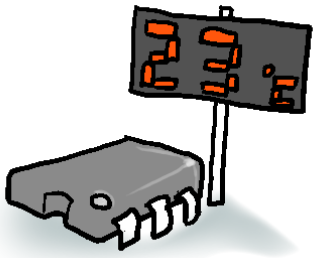
(temp 23)

?

```
struct packet {
  char        sensor_id;
  char        value_bcd[3];
  short int crc16;
} __attribute__((packed));
```

パケットとS式の相互変換

フォーマットの定義(C言語)：



```
01 23 00 00 0e ec
```

(temp 23)

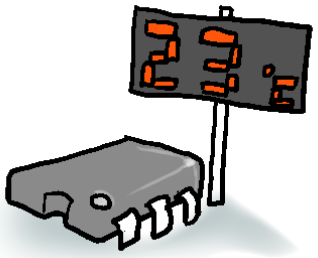
```
struct packet {
  char        sensor_id;
  char        value_bcd[3];
  short int   crc16;
} __attribute__((packed));
```

endian?

alignment?

パケットとS式の相互変換

フォーマットの定義(C言語) :



```
01 23 00 00 0e ec
```

endian?

alignment?

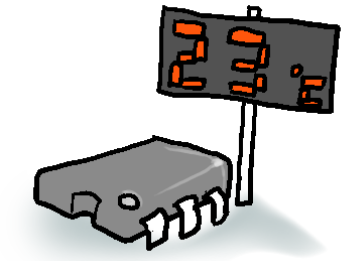
```
struct packet {
  char          sensor_id;
  char          value_bcd[3];
  short         checksum;
} __attribute__ ((packed));
```

C言語の構造体ではこれらを記述できない

(→ 新しい標準が必要)

パケットとS式の相互変換

フォーマットの定義(本研究) :

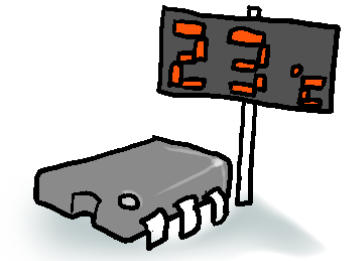


```
01 23 00 00 0e ec
```

```
(define-packet-format sens
  [sensor-id u8]
  [/         u24 (bcd)]
  [crc16     u16])
```

パケットとS式の相互変換

フォーマットの定義(本研究) :



```
01 23 00 00 0e ec
```

```
(define-packet-format sens
  [sensor-id u8]
  [/         u24 (bcd)]
  [crc16     u16])
```

→ C言語の**struct**の機能＋α

パケットフォーマット記述機能

```
(define-packet-format sens
  [sensor-id u8]
  [/         u24 (bcd)]
  [crc16     u16])
```

→ C言語の**struct**の機能＋α

パケットフォーマット記述機能

- (C言語の**struct**に相当する機能)
- 可変長配列
  - TLV(Type-Length-Value)構造
- 辞書(**enum**に相当)
- 1bit単位の型
- エンディアンの記述
- パディング
- オフセットポインタ

これらをR6RS Schemeのライブラリとして記述/利用可能

パケットフォーマット記述機能

- (C言語の`struct`に相当する機能)
- 可変長配列
  - TLV(Type-Length-Value)構造
- 辞書(`enum`に相当)
- 1bit単位の型
- エンディアンの記述
- パディング
- オフセットポインタ

TLV構造とは


```
   Type = 01
 Length = 00 03 (3 bytes)
  Value = 12 34 56
```

- 可変長フィールドの表現手法の一つ
- Type, Length, Valueの3フィールドからなるパケット構造の通称
- TCPのオプションやUSBのデスクリプタなどで広範に利用される

数値←→シンボル


```
  Type = 01
Length = 00 03 (3 bytes)
 Value = 12 34 56

(define-dict sensdata
   (1 temp)
   (2 humid) ...
```

- define-dictによる辞書を用いる
- 典型的にはTLVのTypeフィールド

パケット内容の補完


```
   Type = temp (01)
 Length = 00 03 (3 bytes)
  Value = 12 34 56
```

- TLVにおけるLengthは自明
- ユーザに見せる必要は無い
- define-packet-typeにおける属性として指示

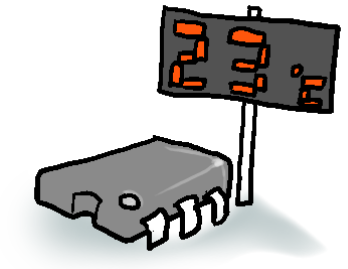
パケット内容の補完

パケットに含まれる情報のうち、以下のようなフィールドは
ユーザに表示されず、パケットを生成するときに自動的に補完。

- TLVにおける長さフィールド
- チェックサムやCRC
- パディング

→　以上で、”S式とパケットの相互変換”が完成

Overview

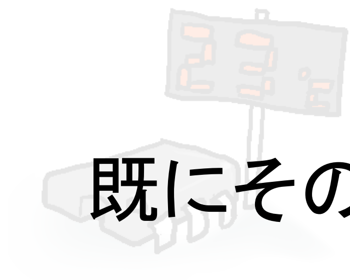


```
(define-packet-format pkt
  [type u8 (tlv-type)]
  [l
  [/
```

```
(define-dict pkt/type
  [temp  1]
  [humid 2] ...)
```

- **define-packet-type**と**define-dict**でデバイスのパケットフォーマットを記述する
- R6RSライブラリにコンパイルする
- REPLからS式を観察する
- プログラムを書く

Overview



```
(define-packet-format pkt
```

```
pkt/type
  [temp  1]
  [humid 2] ...)
```

既にそのデバイス向けのライブラリが提供されていれば
前半の手順は不要

- **define-packet-type**と**define-dict**でデバイスのパケットフォーマットを記述する
- R6RSライブラリにコンパイルする
- REPLからS式を観察する
- プログラムを書く

まとめ

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

C言語の`struct`＋プロトコルの記述に不可欠な機能

agenda

- 本研究の動機
  - デバイス直接制御の必要性
- 提案する開発手法
- 言語の機能
- 評価

評価(TCP/IP)

- TCP/IPプロトコルを実装
  - NIC(RTL8150) : 281行
  - ARP : 64行
  - Ethernet+IP+TCP : 822行

- TCPオプションはMTUのみ
- 静的ARP、静的ルーティング
- Discardサーバ(データは単に捨てる)

評価(TCP/IP) - 記述

- TCPでは疑似ヘッダが必要なため、パケットに含まれる情報だけではパケットの内容を補完できない。

評価(TCP/IP) - 記述

- TCPでは疑似ヘッダが必要なため、パケットに含まれる情報だけではパケットの内容を補完できない。

→ コネクション状態(疑似)フィールドを導入

  - 実際のパケットでは省略されるが、値としては与える必要のあるフィールド。
  - TCPの場合は、送信するデータ以外に、シーケンス番号やIPアドレスも含める。
  - これらのデータはユーザからは隠蔽される。(クロージャに含まれる)

評価(TCP/IP) - パフォーマンス

- 3.0Mbps (230 packets/sec)
- I/O bound
- Windows上での利用と遜色ないパフォーマンス

評価(TCP/IP) - パフォーマンス

- 3.0Mbps (230 packets/sec)
- I/O bound
- Windows上での利用と遜色ないパフォーマンス

→ NICがUSB 1.1接続だった
(想定用途には十分)

評価(TCP/IP) - パフォーマンス

- 負荷の殆どはGC (Boehm GC)
  - GC_mark_from + GC_header_cache_missで実行時間の5割程度

- 多数のアロケーションが発生するためと考えられる

→ より効率的なバッファ管理が必要

Future work

- パケットフォーマット記述の再利用
  - C言語ライブラリの生成
  - Linux kernel moduleへの応用
- `packet-format`リポジトリの構築
  - wikiなどへの展開
- コンパイラの生成するコードの改善
  - メモリコピーの削減
  - ネイティブコード化、SIMD化

Conclusion

- “人間に読める”データでデバイスを制御する手法について検討した。
- TLV構造の抽象化等、典型的なデバイスプロトコルの実現に必要な要素を抽出し、処理系として実装した。
- 提案する言語でTCP/IPを実装し、その実用性を検討した。

予備スライド

- 著しく予備

予備スライド

- うっとりと予備

ユーザモード制御の一般性

- 例：
  - DisplayLinkのUSB-VGAアダプタはWindows上でlibusbを使っている。
  - iPhone上のBluetooth
  -
- 各種方法論
  - TUN/TAP device
  - libusb (FreeBSD 8から標準)
  -

OSは不要？

- NO！
  - 現状では、デバイスを共有することが出来ない。
- Yes ..?
  - OSはデバイスに対するpacketの送受信だけを面倒みれば十分だと考えている(ala microkernel, “thin” OS)
  -

未解決の問題

- 公開
  - 次のmoshリリースでpreview版を同梱予定
- ユーザビリティテスト
- mux / demux (aka OSの提供する素晴しい機能)
  - デバイスの共有は想定外。
  - たとえば抽象化デバイスを作るなど。。
  - ...

推論の完全性

- うまくいく
  - bluetooth (デザイン基準)
  - USB
  - 各種気象センサ (SHT-x1 - 本来の目的)
- うまくいかない
  - TCP
  - ELF – どちらもunionが無いため
  - …

実用的には十分と期待

wireshark等との違い

the ONLY portable & synthesisable protocol description language.

“実行可能Wikipedia”(devicepedia?)を目指している。

- 全て既存のプログラミング言語に依存している。
  - wireshark – C
  - binspect – Ruby
  - Hachoir – Python