1 ビット デジタル(ON/OFF)信号入出力（公開資料） (1Bit Digital Signal Input And Output)	作成 2022/1/17 ニューラルソフト有限会社 市来 博記

初版	－	2022/1/17
B	1 ビット デジタル信号の応答時間に関する記載を更新した。(計測プログラムを整理して再計測した為)	2022/2/2

概要

本書は、Linux OS 搭載の 2 台の Raspberry Pi で 1 ビット デジタル(ON/OFF)信号による通信を行う方法の調査の結果を記したものです。

調査内容

調査項目を示します。

1　ユーザ空間で動作するプログラムとカーネル空間ドライバの関係
2　開発環境への対応
3　Raspberry Pi の GPIO 入出力部のハードウェア
4　1 ビット デジタル信号を入出力する C/C++言語プログラムの作成方法
5　2 台の Raspberry Pi における 1 ビット デジタル信号の応答時間

情報源

Qiita(@wancom) : libgpiod の使い方
libgpiod/libgpiod.git : libgpiod のソース コード
#embeddedbits : Linux kernel GPIO user space interface
BeyondLogic : An Introduction to chardev GPIO and Libgpiod on the Raspberry PI
Lloyd Rochester - Using libgpiod to detect input events
hnw の日記 : Raspberry Pi の GPIO は起動直後から内部プルダウンされている

関連情報

Raspberry Pi リモート開発環境の構築

前提条件

調査における前提条件を示します。

- 調査対象の通信環境は、Raspberry Pi Model 4B と Raspberry Pi Model 3B の GPIO ピンをジャンパー線(約 20cm)で接続したネットワークとします。
- Raspberry Pi に搭載する OS は、以下の通りとします。
  Model 4B : Raspberry Pi OS 10 Buster 64BIT β

Model 3B：Raspberry Pi OS 10 Buster 32BIT

- 調査用プログラムの作成と動作確認は Visual Stdudio 2022 Community（C++による Linux 開発）を使用します。（Visual Stdudio 2022 を使用した Raspberry Pi Model 4B/3B のリモート開発環境は構築済みとします。構築方法は、「Raspberry Pi リモート開発環境の構築」を参照して下さい。）
- 開発用ホスト マシンは、Windows10（64 ビット）がインストールされた PC（i7-3930K，32GB メモリ搭載の PC）を使用します。
  開発/調査環境は図 1 の通りです。

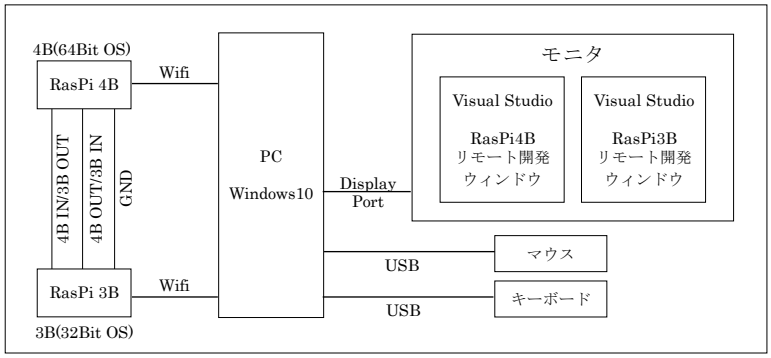


図 1 開発/調査環境

- 1 ビット デジタル信号の入出力は、libgpiod を使用して実現します。
- 本書の記載内容は、C/C++言語でのプログラム作成の経験がある方が対象です。

調査結果

〔ユーザ空間で動作するプログラムとカーネル空間ドライバの関係〕

linux などの OS 上で動作するユーザ プログラムからハードウェアにアクセスする場合、通常は特権モードで動作するカーネル空間ドライバ モジュールを介してアクセスすることになります。（ハードウェアのメモリマップド I/O 空間をユーザ空間にマッピングすることで、ユーザ プログラムからの直接アクセスを可能する機能も存在します。）カーネル空間ドライバ モジュールは、ハードウェアに直接アクセスする物理層(ポート)ドライバ、物理層ドライバの機能を利用してハードウェアにアクセスする論理層(クラス)ドライバ、論理層ドライバと物理層ドライバの仲介をする I/O フレーム ワークに分離・階層化されています。（ユーザ空間で動作するプログラムも論理層ドライバとなるものがあります。）

linux における GPIO に分類されるハードウェアに対するドライバのコンポーネントの呼称は以下の通りです。

- 物理層ドライバ：プロデューサ=ラインを生成するドライバ（カーネル空間モジュール）
- 論理層ドライバ：コンシューマ※1= ラインを消費するドライバ（カーネル/ユーザ空間モジュール）
- I/O フレーム ワーク：gpiolib（カーネル空間モジュール）

※1 論理層ドライバを使用するユーザ プログラムもコンシューマになります。

GPIO に関連するモジュールの関係を図 2 に示します。

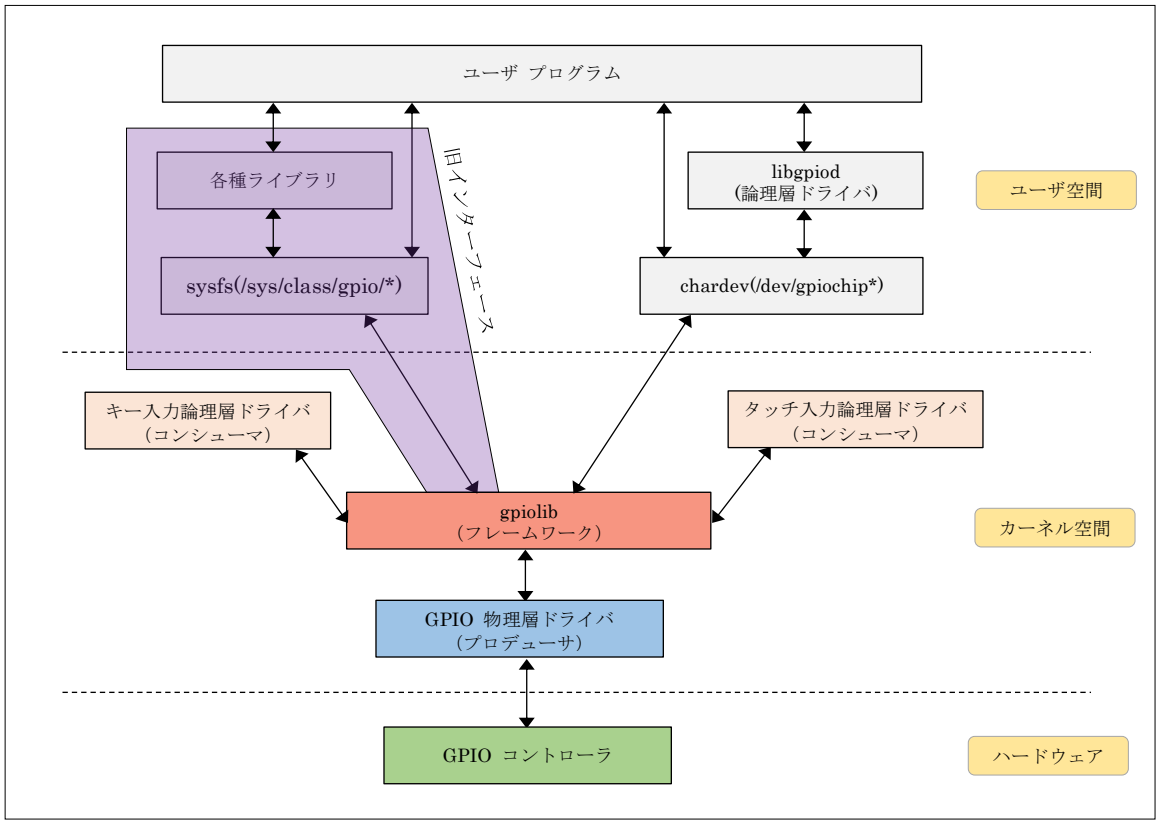


図 2 GPIO に関連するモジュールの関係

今回、1 ビット デジタル信号の入出力で使用する libgpiod は、ユーザ空間で動作する論理層ドライバであり、キャラクター デバイス ファイル経由で GPIO 入出力を実現しています。

〔開発環境への対応〕

次のコマンドで libgpiod の利用に必要なパッケージをインストールします。

```
sudo apt install libgpiod2 libgpiod-dev libgpiod-doc
```

2022 年 1 月 17 日現在、上記のコマンドでインストールされるパッケージの libgpiod モジュールは最新版ではないようです。（内部バイアスの設定がサポートされていません。）最新版の機能が必要な場合は、”https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/” からビルド環境一式をダウンロードできます。

libgpiod 最新版のビルド方法は以下の通りです。

```
sudo apt install autoconf-archive（autoconf-archive がインストール済みの場合は不要※1)
./autogen.sh --enable-tools=yes --prefix=インストール パス(絶対パス)
make
make install
```

※1 autoconf-archive がインストールされていない場合、autogen.sh の実行が異常終了します。
エラー メッセージ：

```
configure.ac:160: error: Unexpanded AX_ macro found. Please install GNU autoconf-archive.
```

〔Raspberry Pi の GPIO 入出力部のハードウェア〕

図 3 に Raspberry Pi の GPIO 入出力部のハードウェア ブロック構成を示します。

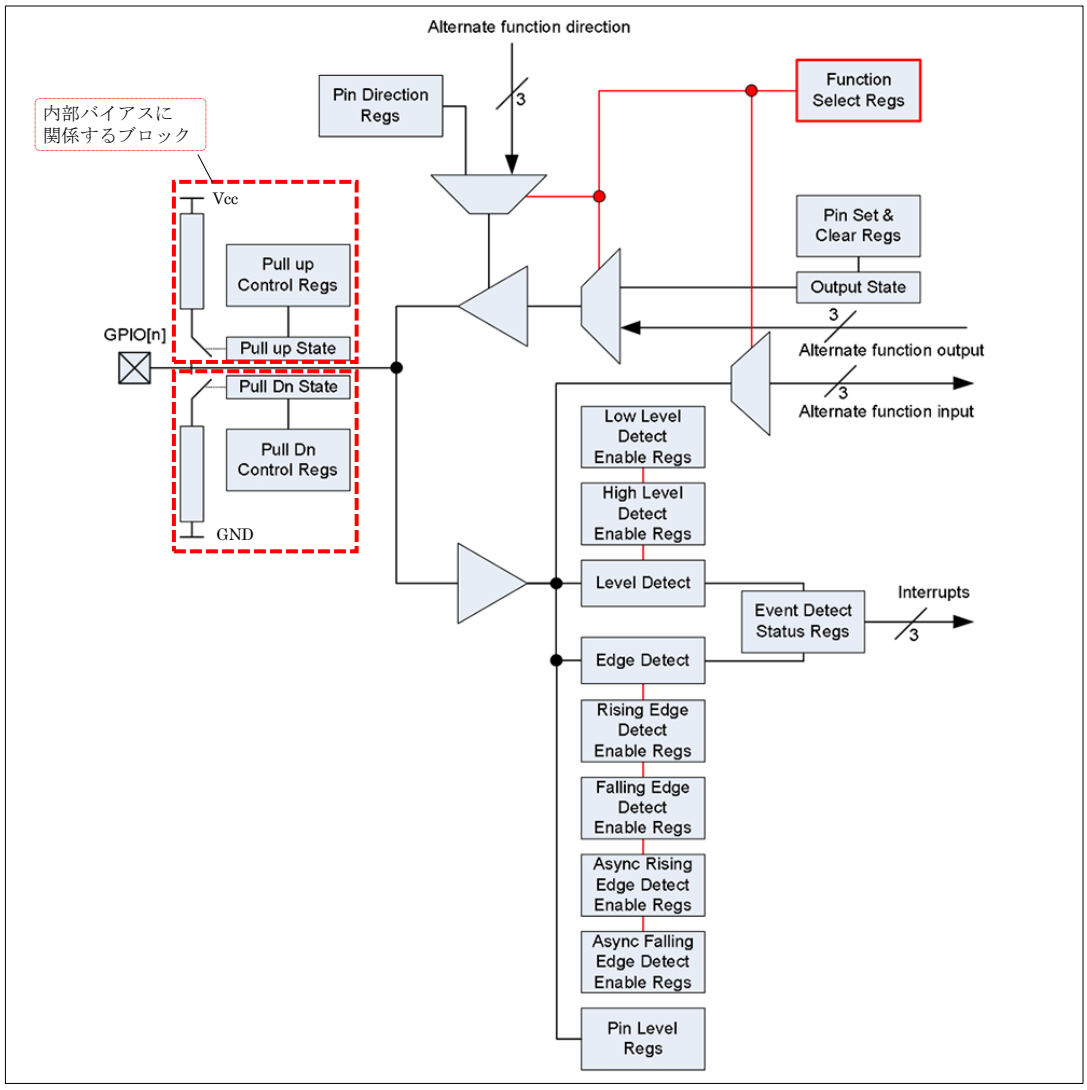


図 3 Raspberry Pi の GPIO 入出力部のブロック構成

（Raspberry Pi 公式資料：bcm2835-peripherals.pdf の Figure 6-1）

物理層のドライバを作成するのでなければ、GPIO 入出力部のハードウェアを意識する必要は殆どありません。しかし、入出力ピンと外部の回路を接続する際は、内部バイアスについて、知っておく必要があります。（初期状態のバイアスの状態と抵抗値については、「hnw の日記 ：Raspberry Pi の GPIO は起動直後から内部プルダウンされている」を参照して下さい。）

〔1 ビット デジタル信号を入出力する C/C++言語プログラムの作成方法〕

libgpiod を使用して 1 ビット デジタル信号を入出力するプログラムの流れは、以下のようになります。

図 4 libgpiod を利用した 1 ビット デジタル信号通信プログラムの流れ

以下に使用頻度が高いと思われる libgpiod の関数の使用方法を示します。

〔必要なインクルード ファイル〕

```
#include <gpiod.h>
```

〔GPIO チップのオープン/クローズ系〕

〔GPIO チップのオープン〕

キャラクタ デバイス ファイル名を指定して、gpiod_chip_open*関数を呼び出します。

```
// GPIOオープン
struct gpiod_chip* gpioc = gpiod_chip_open("/dev/gpiochip0");
  又は、
struct gpiod_chip* gpioc = gpiod_chip_open_by_name("gpiochip0");
  又は、
struct gpiod_chip* gpioc = gpiod_chip_open_by_number(0);
  又は、
struct gpiod_chip* gpioc = gpiod_chip_open_by_label(GPIOチップ ラベル※1);
  又は、
struct gpiod_chip* gpioc = gpiod_chip_open_lookup(上記のgpiod_chip_open*関数のパラメータの何れか);
if (gpioc == nullptr)
{
    //  エラー処理
}

※1 GPIOチップ ラベルはgpiod_chip_open 関数でオープン後、gpiod_chip_label関数で確認できます。
```

〔GPIO チップのクローズ〕

GPIO チップを指定して、gpiod_chip_close 関数を呼び出します。

```
// GPIOクローズ
gpiod_chip_close(gpioc); // 確保されたリソースも全て開放される
```

〔単一ライン操作系〕

〔ラインの取得〕

GPIO チップと GPIO 番号を指定して、gpiod_chip_get_line 関数を呼び出します。

```
// ラインの取得
struct gpiod_line* line = gpiod_chip_get_line(gpioc, GPIO番号);
if (line == nullptr)
{
    //  エラー処理
}
```

〔ラインの確保〕

ライン、構成情報、初期状態値を指定して、gpiod_line_request 関数を呼び出します。

```
// ラインの確保
struct gpiod_line_request_config config = {
    .consumer = "コンシューマ名",
    .request_type = GPIOD_LINE_REQUEST_DIRECTION_※1 又は GPIOD_LINE_REQUEST_EVENT_※2,
    .flags = GPIOD_LINE_REQUEST_FLAG_OPEN_※3 [| GPIOD_LINE_REQUEST_FLAG_ACTIVE_LOW] [| GPIOD_LINE_REQUEST_FLAG_BIAS_※4]
};
sts = gpiod_line_request(line, &config, 0);
if (sts != 0)
{
    //  エラー処理
}

※1 AS_IS(入出力の方向に変更なし)/INPUT(入力)/OUTPUT(出力)
※2 FALLING_EDGE(立下りエッジ)/RISING_EDGE(立ち上がりエッジ)/BOTH_EDGES(両エッジ)
※3 DRAIN/SOURCE(入出力の方向が出力の場合のみ指定可能(core.c : line_request_values 関数内でチェックしている))
※4 libgpiod v1.5.3 以上で内部バイアスの設定が可能
```

〔入力用ラインの確保〕

ライン、コンシューマ名［、フラグ値］を指定して、gpiod_line_request_input*関数を呼び出します。

```
// 入力用ラインの確保
sts = gpiod_line_request_input[_flags](line, "コンシューマ名"[, フラグ値※1]);
if (sts != 0)
{
    //  エラー処理
}

※1 フラグ値は〔ラインの確保〕を参照のこと。
```

〔出力用ラインの確保〕

ライン、コンシューマ名［、フラグ値］、初期状態値を指定して、gpiod_line_request_output*関数を呼び出します。

```
// 出力用ラインの確保
sts = gpiod_line_request_output[_flags](line, "コンシューマ名"[, フラグ値※1], 0又は1);
if (sts != 0)
{
    //  エラー処理
}

※1 フラグ値は〔ラインの確保〕を参照のこと。
```

〔イベント入力用ラインの確保〕

ライン、コンシューマ名［、フラグ値］を指定して、以下の関数の何れかを呼び出す。

立ち上がりエッジ イベント：gpiod_line_request_rising_edge_events*

立ち下がりエッジ イベント：gpiod_line_request_falling_edge_events*

両エッジ イベント　　：gpiod_line_request_both_edges_events*

```
// イベント入力用ラインの確保
sts = gpiod_line_request_rising_edge_events[_flags](line, "コンシューマ名"[, フラグ値※1]);
  又は
sts = gpiod_line_request_falling_edge_events[_flags](line, "コンシューマ名"[, フラグ値※1]);
  又は
sts = gpiod_line_request_both_edges_events[_flags](line, "コンシューマ名"[, フラグ値※1]);
if (sts != 0)
{
    //  エラー処理
}
```

※1 フラグ値は〔ラインの確保〕を参照のこと。

〔ラインの開放〕

ラインを指定して、gpiod_line_release 関数を呼び出します。

```
// ラインの解放
gpiod_line_release(line);
```

〔信号出力〕

ライン、出力値を指定して、gpiod_line_set_value 関数を呼び出す。

```
// 信号出力
sts = gpiod_line_set_value(line, 0 又は 1);
if (sts != 0)
{
    //  エラー処理
}
```

〔信号入力〕

ラインを指定して、gpiod_line_get_value 関数を呼び出す。

```
// 信号入力
sts = gpiod_line_get_value(line); // 正常完了時、sts=0 又は 1（信号の状態）
if (sts == -1)
{
    //  エラー処理
}
```

〔イベント待ち〕

ライン、タイムアウトまでの時間を指定して、gpiod_line_event_wait 関数を呼び出す。

```
// イベント待ち
struct timespec timeout = {
    .tv_sec = 30,   // 秒単位の時間
    .tv_nsec = 0    // ナノ秒単位の時間
};
sts = gpiod_line_event_wait(line, &timeout); // sts=1 の時、イベント発生
if (sts < 1)
{
    //  sts=-1 の時はエラー処理、sts=0 の時はタイムアウト処理
}
```

〔イベント取得〕

ライン、イベント格納領域を指定して、gpiod_line_event_read 関数を呼び出す。

イベントが発生していない状態で、この関数を呼び出すとイベントが発生するまでブロックされます。

```
// イベント取得
gpiod_line_event ev;
sts = gpiod_line_event_read(line, &ev);
if (sts != 0)
{
    //  エラー処理
}
```

〔複数イベント取得〕

ライン、イベント格納用配列、最大イベント数を指定して、gpiod_line_event_read_multiple 関数を呼び出す。

```
// イベント取得
gpiod_line_event events[最大イベント数];
sts = gpiod_line_event_read_multiple(line, events, countof(events));
if (sts != 0)
{
    //  エラー処理
}
```

〔ラインの構成確認〕

ラインを指定して、以下の関数を呼び出す。

GPIO 番号確認	: gpiod_line_offset	信号名確認	: gpiod_line_name
コンシューマ確認	: gpiod_line_consumer	入出力方向確認	: gpiod_line_direction
正/負論理確認	: gpiod_line_active_state	バイアス確認	: gpiod_line_bias
使用中確認	: gpiod_line_is_used	出力形態確認	: gpiod_line_is_open_drain
			: gpiod_line_is_open_source

```
// GPIO 番号確認
int lineNum = gpiod_line_offset(line);
// 信号名確認
const char* name = gpiod_line_name(line);
// コンシューマ確認
const char* consumer = gpiod_line_consumer(line);
// 入出力方向確認
int direction = gpiod_line_direction(line);  // direction : GPIOD_LINE_DIRECTION_INPUT / GPIOD_LINE_DIRECTION_OUTPUT
// 正/負論理確認
int actState = gpiod_line_active_state(line); // actState : GPIOD_LINE_ACTIVE_STATE_HIGH / GPIOD_LINE_ACTIVE_STATE_LOW
// バイアス確認
int bias = gpiod_line_bias(line); // bias : GPIOD_LINE_BIAS_PULL_UP / GPIOD_LINE_BIAS_PULL_DOWN
                                  //        GPIOD_LINE_BIAS_DISABLE / GPIOD_LINE_BIAS_AS_IS(unknown)
// 使用中確認
bool used = gpiod_line_is_used(line);
// 出力形態確認
bool openDrain = gpiod_line_is_open_drain(line);
bool openSouece = gpiod_line_is_open_source(line);
```

〔ラインの構成フラグ変更〕

ライン、構成フラグを指定して、gpiod_line_set_flags 関数を呼び出します。

```
// ラインの構成フラグ変更
int flags = GPIOD_LINE_REQUEST_FLAG_OPEN_※1 [| GPIOD_LINE_REQUEST_FLAG_ACTIVE_LOW] [| GPIOD_LINE_REQUEST_FLAG_BIAS_※2];
sts = gpiod_line_set_flags(line, flags);
if (sts != 0)
{
    //  エラー処理
}

※1 DRAIN/SOURCE(入出力の方向が出力の場合のみ指定可能(core.c : line_request_values 関数内でチェックしている))
※2 libgpiod v1.5.3 以上で内部バイアスの設定が可能
```

〔ラインの入出力方向変更〕

入力への変更時、ラインを指定して、gpiod_line_set_direction_input 関数を呼び出します。
出力への変更時、ラインと出力値を指定して、gpiod_line_set_direction_output 関数を呼び出します。

```
// ラインを入力へ変更
sts = gpiod_line_set_direction_input(line);
// ラインを出力へ変更
sts = gpiod_line_set_direction_output(line, 0 又は 1);
if (sts != 0)
{
    //  エラー処理
}
```

〔複数ライン操作系〕

〔複数ラインの取得〕

GPIO チップ、GPIO 番号の配列、GPIO 番号の配列数を指定して、gpiod_chip_get_lines 関数を呼び出します。

```
// 複数ラインの取得
unsigned int gpioNums[] = { GPIO番号0, GPIO番号1, ･･･ GPIO番号n };
struct gpiod_line_bulk lines = {};
int sts = gpiod_chip_get_lines(gpioc, gpioNums, countof(gpioNums), &lines);
if (sts != 0)
{
    //  エラー処理
}
```

〔複数ライン(構造体)の操作〕

複数ラインと操作の対象を指定して、以下の関数を呼び出します。

初期化　　：gpiod_line_bulk_init

ラインの追加　　：gpiod_line_bulk_add（操作の対象：追加するライン）

ラインの取得　　：gpiod_line_bulk_get_line（操作の対象：ラインのインデックス番号）

ライン数の取得　　：gpiod_line_bulk_num_lines

```
// 複数ラインの初期化
struct gpiod_line_bulk lines;
gpiod_line_bulk_init(&lines);

// ラインの追加
struct gpiod_line* line = gpiod_chip_get_line(gpioc, GPIO番号);
gpiod_line_bulk_add(&lines, line);

// ライン数の取得
int linesNum = gpiod_line_bulk_num_lines(&lines);

// ラインの取得
struct gpiod_line* lineX = gpiod_line_bulk_get_line(&lines, 0～linesNum-1のインデックス番号);
```

〔複数ラインの確保〕

複数ライン、構成情報、初期状態値の配列を指定して、gpiod_line_request_bulk 関数を呼び出します。

```
// 複数ラインの確保
struct gpiod_line_request_config config = {  // 複数ライン共通の構成情報
    .consumer = "コンシューマ名",
    .request_type = GPIOD_LINE_REQUEST_DIRECTION_※1 又は GPIOD_LINE_REQUEST_EVENT_※2,
    .flags = GPIOD_LINE_REQUEST_FLAG_OPEN_※3 [| GPIOD_LINE_REQUEST_FLAG_ACTIVE_LOW] [| GPIOD_LINE_REQUEST_FLAG_BIAS_※4]
};
int defVals[] = { 0, 0, ･･･ 0 };  // countof(defVals) >= lines.num_lines
sts = gpiod_line_request_bulk(&lines, config, defVals);
if (sts != 0)
{
    //  エラー処理
}

※1 AS_IS(入出力の方向に変更なし)/INPUT(入力)/OUTPUT(出力)
※2 FALLING_EDGE(立下りエッジ)/RISING_EDGE(立ち上がりエッジ)/BOTH_EDGES(両エッジ)
※3 DRAIN/SOURCE(入出力の方向が出力の場合のみ指定可能(core.c : line_request_values 関数内でチェックしている))
※4 libgpiod v1.5.3 以上で内部バイアスの設定が可能
```

〔入力用複数ラインの確保〕

複数ラインとコンシューマ名［、フラグ値］を指定して、gpiod_line_request_bulk_input*関数を呼び出します。

```
// 入力用複数ラインの確保
sts = gpiod_line_request_bulk_input[_flags](&lines, "コンシューマ名"[, フラグ値※1]);
if (sts != 0)
{
    //  エラー処理
}

※1 フラグ値は〔複数ラインの確保〕を参照のこと。
```

〔出力用複数ラインの確保〕

複数ライン、コンシューマ名［、フラグ値］、初期状態値を指定して、gpiod_line_request_bulk_output*関数を呼び出します。

```
// 出力用複数ラインの確保
int defValues[] = { 0, 0, ··· };  // countof(defValues) >= lines.num_lines
sts = gpiod_line_request_bulk_output[_flags](&lines, "コンシューマ名"[, フラグ値※1], defValues);
if (sts != 0)
{
    //  エラー処理
}

※1 フラグ値は〔複数ラインの確保〕を参照のこと。
```

〔イベント入力用複数ラインの確保〕

複数ライン、コンシューマ名［、フラグ値］を指定して、以下の関数の何れかを呼び出す。

立ち上がりエッジ イベント　：gpiod_line_request_bulk_rising_edge_events_flags*

立ち下がりエッジ イベント　：gpiod_line_request_bulk_falling_edge_events_flags*

両エッジ イベント　　：gpiod_line_request_bulk_both_edges_events_flags*

```
// イベント入力用複数ラインの確保
sts = gpiod_line_request_bulk_rising_edge_events[_flags](&lines, "コンシューマ名"[, フラグ値※1]);
  又は
sts = gpiod_line_request_bulk_falling_edge_events[_flags](&lines, "コンシューマ名"[, フラグ値※1]);
  又は
sts = gpiod_line_request_bulk_both_edges_events[_flags](&lines, "コンシューマ名"[, フラグ値※1]);
if (sts != 0)
{
    //  エラー処理
}

※1 フラグ値は〔複数ラインの確保〕を参照のこと。
```

〔複数ラインの開放〕

複数ラインを指定して、gpiod_line_release_bulk 関数を呼び出します。

```
// 複数ラインの解放
gpiod_line_release_bulk(&lines);
```

〔信号出力〕

複数ライン、出力値配列を指定して、gpiod_line_set_value_bulk 関数を呼び出す。

```
// 信号出力
int values[] = { 0, 1, ··· }; // countof(values) >= lines.num_lines
sts = gpiod_line_set_value_bulk(&lines, values);
if (sts != 0)
{
    //  エラー処理
}
```

〔信号入力〕

複数ライン、入力値格納用配列を指定して、gpiod_line_get_value_bulk 関数を呼び出す。

```
// 信号入力
int values[ライン数]; // ライン数 >= lines.num_lines
sts = gpiod_line_get_value_bulk(&lines, values);
if (sts != 0)
{
    //  エラー処理
}
```

〔イベント待ち〕

複数ライン、タイムアウトまでの時間、イベント発生ライン格納領域（複数ライン）を指定して、gpiod_line_event_wait_bulk 関数を呼び出す。

```
// イベント待ち
struct timespec timeout = {
    .tv_sec = 30,   // 秒単位の時間
    .tv_nsec = 0    // ナノ秒単位の時間
};
struct gpiod_line_bulk event_bulk = {};
sts = gpiod_line_event_wait_bulk(&lines, &timeout, &event_bulk);
    // sts=1 の時、少なくとも 1 ラインでイベント発生
    // event_bulk にイベントが発生したライン数とライン情報が格納される
if (sts < 1)
{
    //  sts=-1 の時はエラー処理、sts=0 の時はタイムアウト処理
}
```

〔複数ラインの構成フラグ変更〕

複数ライン、構成フラグを指定して、gpiod_line_set_flags_bulk 関数を呼び出します。

```
// 複数ラインの構成フラグ変更
int flags = GPIOD_LINE_REQUEST_FLAG_OPEN_※1 [| GPIOD_LINE_REQUEST_FLAG_ACTIVE_LOW] [| GPIOD_LINE_REQUEST_FLAG_BIAS_※2];
sts = gpiod_line_set_flags_bulk(&lines, flags);
if (sts != 0)
{
    //  エラー処理
}

※1 DRAIN/SOURCE(入出力の方向が出力の場合のみ指定可能(core.c : line_request_values 関数内でチェックしている))
※2 libgpiod v1.5.3 以上で内部バイアスの設定が可能
```

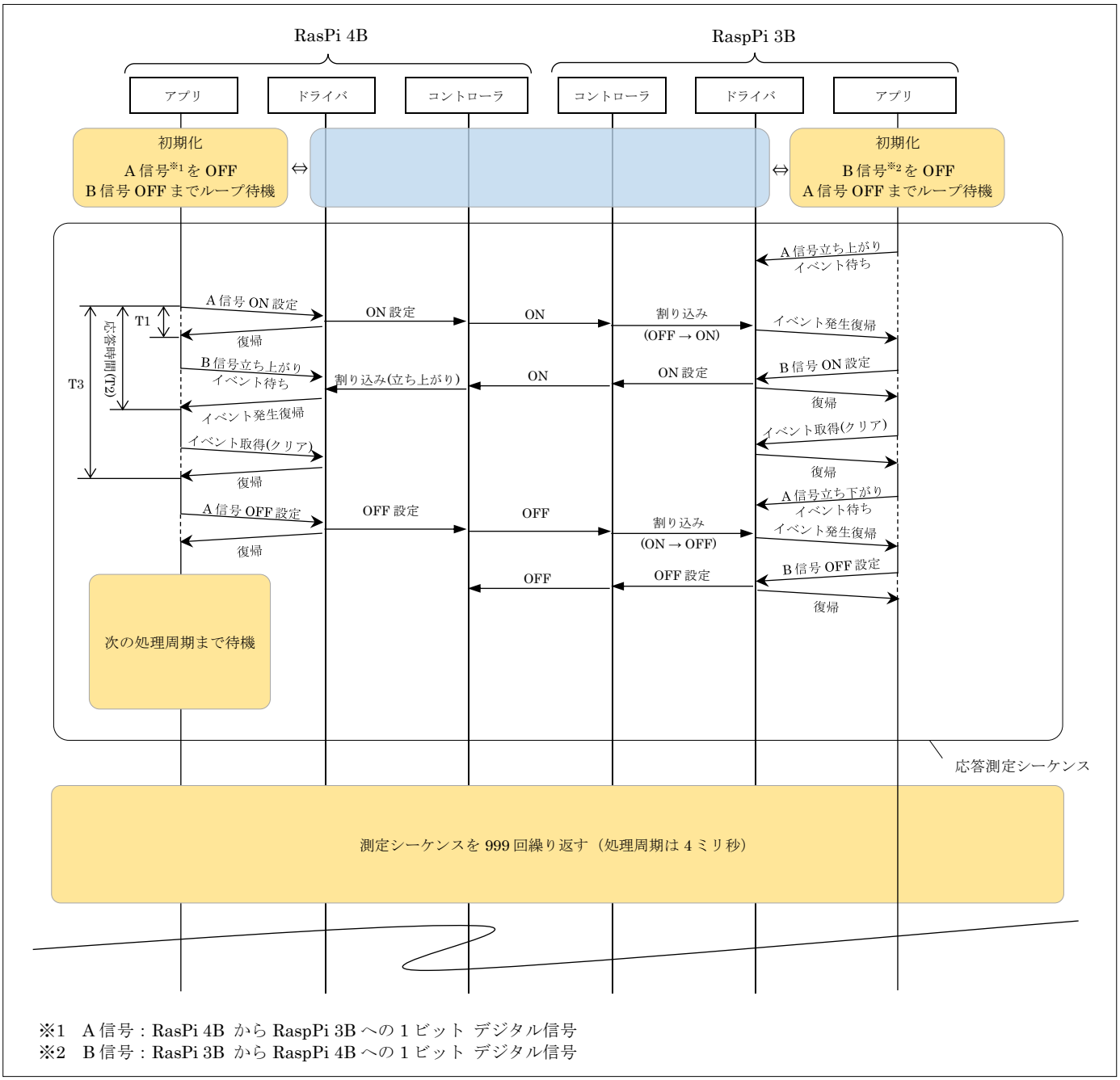
〔ラインの入出力方向変更〕

入力への変更時、複数ラインを指定して、gpiod_line_set_direction_input_bulk 関数を呼び出します。
出力への変更時、複数ラインと出力値配列を指定して、gpiod_line_set_direction_output_bulk 関数を呼び出します。

```
// 複数ラインを入力へ変更
sts = gpiod_line_set_direction_input_bulk(&lines);
// 複数ラインを出力へ変更
int values[] = { 0, 0, … };  // countof(values) >= lines.num_lines
sts = gpiod_line_set_direction_output_bulk(&lines, values);
if (sts != 0)
{
    //  エラー処理
}
```

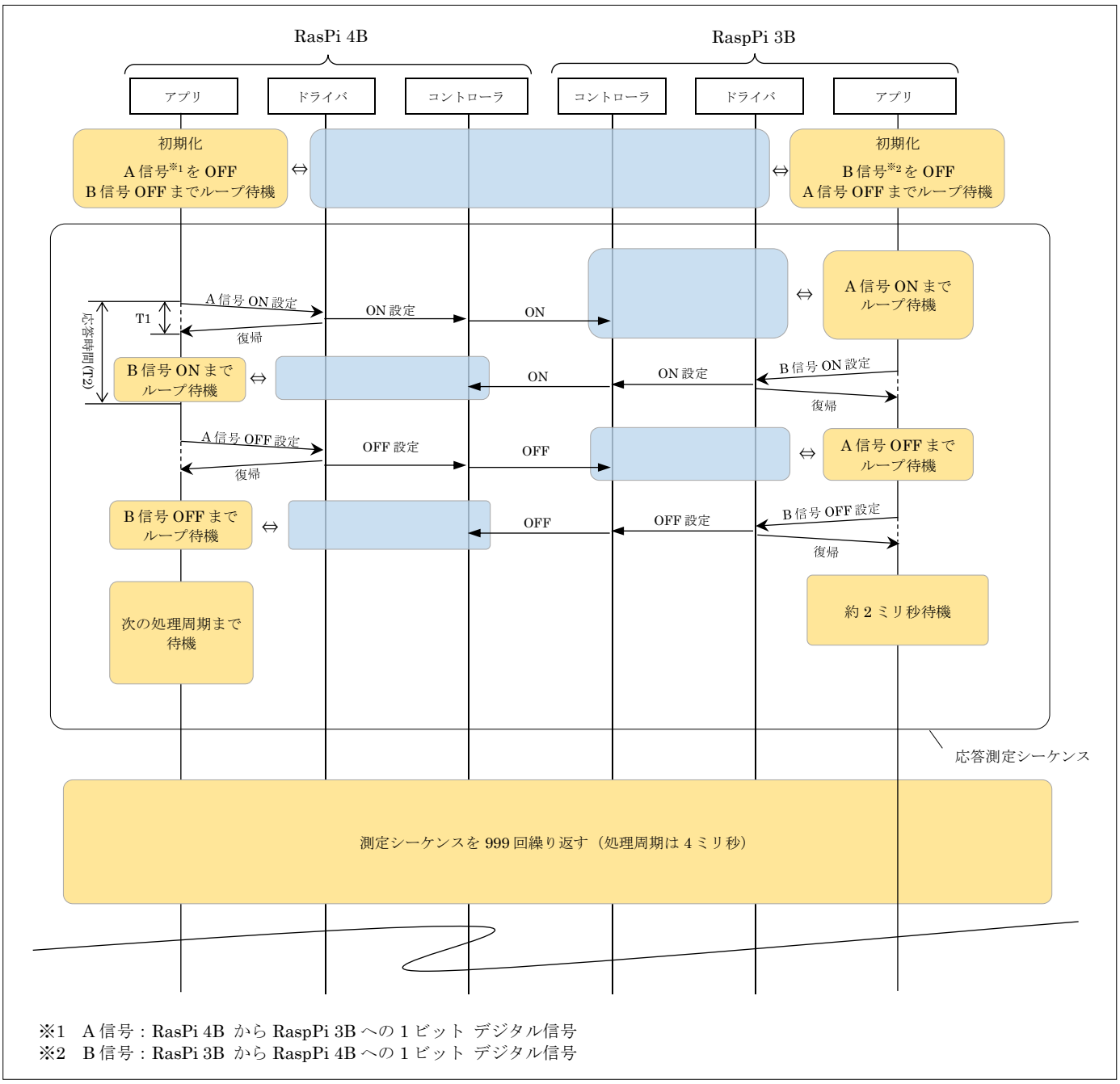
〔基本シーケンスにおける応答時間〕

応答時間測定の対象となる通信シーケンスを図 5 と図 6 に示します。



※1　A 信号：RasPi 4B から RaspPi 3B への 1 ビット デジタル信号
※2　B 信号：RasPi 3B から RaspPi 4B への 1 ビット デジタル信号

図 5 通信シーケンス 1（イベントを使用）



※1　A 信号：RasPi 4B から RaspPi 3B への 1 ビット デジタル信号
※2　B 信号：RasPi 3B から RaspPi 4B への 1 ビット デジタル信号

図 6 通信シーケンス 2（イベント未使用）

図 5 と図 6 の応答測定シーケンスを約 4 ミリ秒周期で 1000 回繰り返して応答時間を計測（clock_gettime 関数を使用）した結果を以下に示します。

〔計測の共通設定〕

CPU の周波数設定　　：cpufreq-set -g コマンドで performance を設定
計測プログラムのスレッド数　：シングル
プロセスの優先度　　：-10
スレッドのスケジュール設定　：ポリシー、優先度ともに変更なし

〔通信シーケンス 1（イベントを使用)〕

結果

```
        Ave,   Max,   Min
t1 =     2,     13,     2[usec]
t2 =    53,    107,    50[usec] (Ave と Min の差 3usec])
t3 =    56,    112,    52[usec]
```

〔通信シーケンス 2（イベント未使用)〕

結果

```
        Ave,   Max,   Min
t1 =     2,     13,     2[usec]
t2 =     5,     17,     4[usec] (Ave と Min の差 1usec])
```

アプリケーションで外部の信号変化を監視する場合、75 マイクロ秒程度（通信シーケンス 1 における最長時間×0.5×1.4）で信号変化を検知できると思われる。(恐らくタスクの優先順位のカスタマイズが必要)

所感

C/C++言語で 1 ビット デジタル(ON/OFF)信号入出力を行う場合、使用するライブラリの選択肢が複数あります。その中でも libgpiod ライブラリは、複数の信号を纏めて制御する機能が既に実装されていることや、C++言語のクラス ライブラリもあることから、アプリケーションを楽に実装できると思います。又、linux 推奨であることから将来的な互換性の面でも安心して使えるライブラリだと思います。ただ、イベントを使用する場合、オーバー ヘッドが大きいと感じるかも知れません。(libgpiod の問題ではなく、OS コア部分のスケジューリング機能か、タクス切り替え処理そのものに問題があるのかも知れません。)