ASTROCAMP 衛星開発ゼミ

事前学習 第1段階 ／ Week 4

リアルタイムシステム

テーマ　LED の点滅速度をスイッチで切り替える

目的①　リアルタイムシステムの型を学ぶ

目的②　シンプルなテーマにも型があることを学ぶ

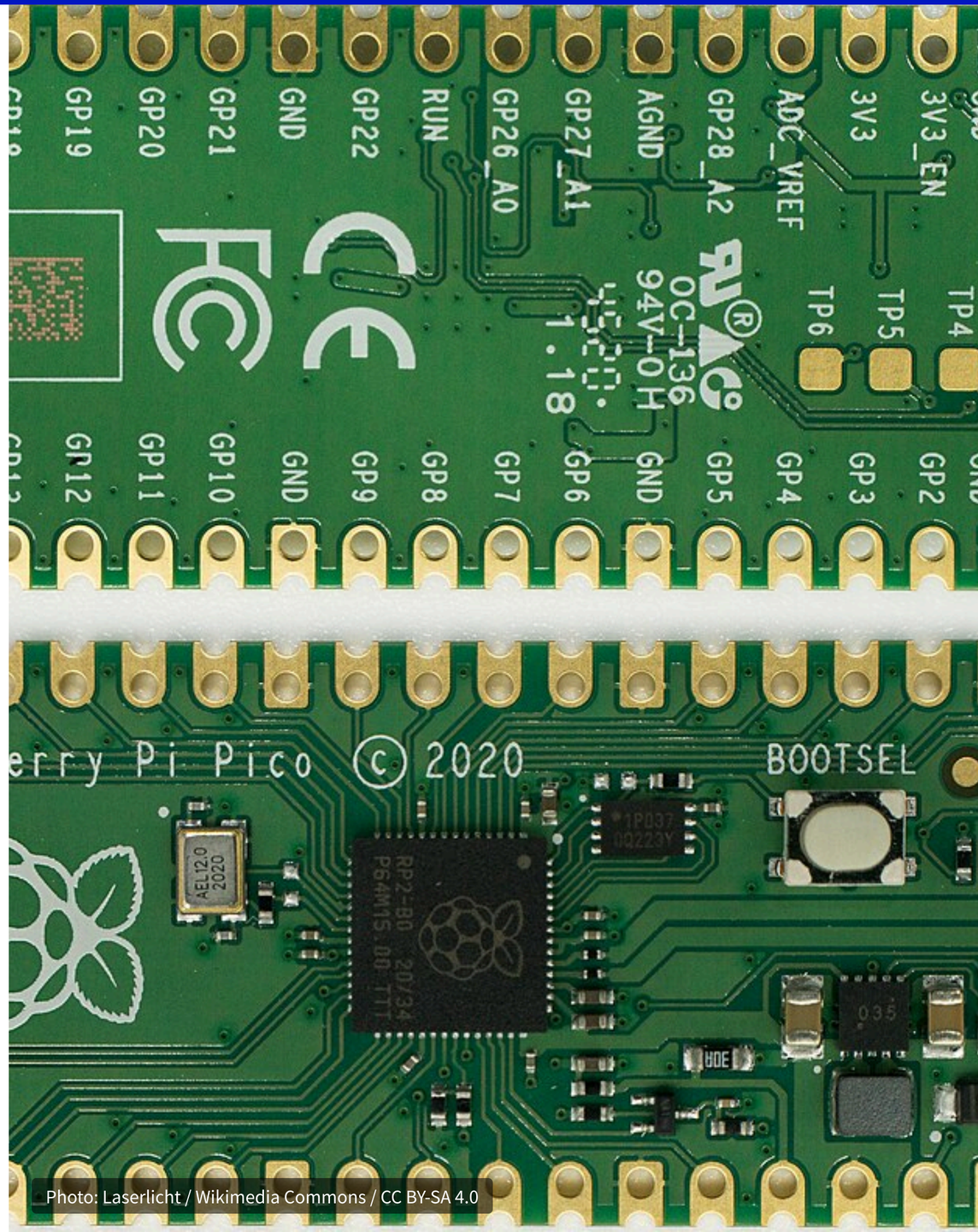
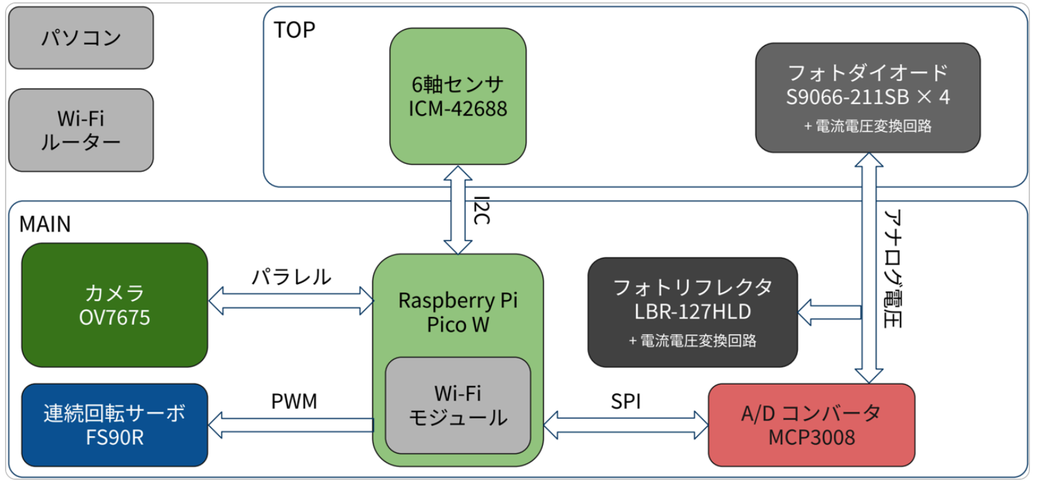


Photo: Laserlicht / Wikimedia Commons / CC BY-SA 4.0

Week 1〜3 で作れるようになった要素を、1個の CPU の上で回し続ける

- **Week 1〜3：**
  マイコンの原理・センサとの通信という要素技術
- **Week 4：**
  要素を同時に動かす統合の技術



センサ・アクチュエータ・通信のすべてが1つの Pico W にぶら下がる

Chapter 1　衛星の仕事は1つではない

姿勢制御・テレメトリ送信・コマンド受信という、周期も性質も違う仕事が1個の CPU で回り続ける

1個の CPU（Raspberry Pi Pico W）

姿勢制御

短い周期で測って動かし続ける

●●●●●●●● 密

テレメトリ送信

緩い周期で定期的に送る

● ● ● 疎

コマンド受信

不定期。来た瞬間を取りこぼせない

● ●● ● 不定

工学のリアルタイムは、処理が速いことではなく、決めた締め切り（デッドライン）を毎回守ること

速さでも平均でもなく、締め切りを毎回守れるか

	処理 A	処理 B
平均の速さ	1 ms（速い）	8 ms（遅い）
最悪のとき	まれに 1 秒	常に 8 ms
締め切り 10 ms を毎回守れるか	**×破る回がある**	**◯ 毎回守る**

締め切り違反の被害で区別する —— 事故に直結するならハード、品質低下で済むならソフト

ハードリアルタイム

違反1回で意味を失う

エアバッグの展開
数 ms の締め切りを1回でも破れば無意味

模擬衛星の**姿勢制御の周期**

ソフトリアルタイム

破っても品質が下がるだけ

動画のコマ落ち
不快だが破綻はしない

地上局の**表示遅れ**

「LED 点滅の速度をスイッチで切り替える」は、衛星と同じ周期タスクとイベントの最小セット

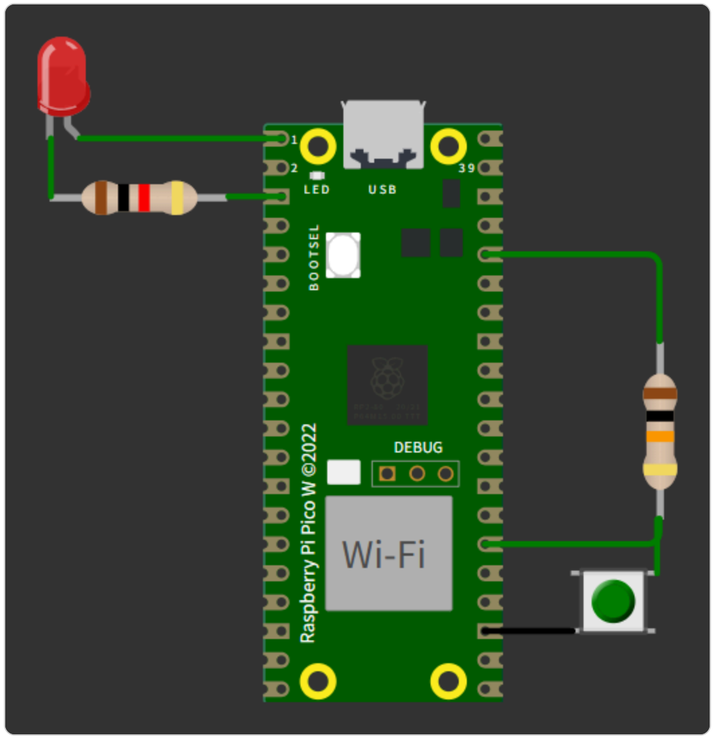
LED を点滅させる

「500 ms ごと」という締め切り付きの**周期タスク**
＝ 姿勢制御・テレメトリと同じ性質

スイッチを判定する

いつ来るか分からない**イベント**
＝ コマンド受信と同じ性質

回路の補足　**プルアップ抵抗：**スイッチは **GND に接続するかどうか**を選ぶだけ
接続しないとき H にするため必要（I2C と同じ）



LED＋抵抗と、外付けプルアップ抵抗つきスイッチの回路

周期の違う複数の仕事を1個の CPU で締め切り通りに回し続ける

覚える ①

リアルタイム≠動作が早い
リアルタイム＝締め切りを毎回守る

覚える ②

処理時間・周期の違う仕事を
マイコン1つで実行する方法を学ぶ

覚える ③

それを学ぶための今日のテーマが
LED 点滅周期のスイッチでの切替

Chapter 2　第1段：sleep_ms() で待つ素朴版

sleep_ms(500) で待ちながら LED を反転し、ループの末尾でスイッチを1回だけ読む一本道

```
// [Code-01] 第1段：素朴版（骨格）
while (true) {
    gpio_put(LED_PIN, 1);
    sleep_ms(interval_ms);        // 点灯のまま待つ
    gpio_put(LED_PIN, 0);
    sleep_ms(interval_ms);        // 消灯のまま待つ
    if (gpio_get(SW_PIN) == 0) {  // 確認は1周に1回
        interval_ms = (interval_ms == 500) ? 100 : 500;
    }
}
```

「待って、待って、最後に1回読む」

1周のうち2つの sleep_ms() が大半を占め、スイッチ確認は末尾に一度だけ。読みやすく、点滅だけなら完璧に動く。

QUIZ 2.1 予想してみよう

素朴版でスイッチを短く押すと、点滅速度が変わるまでに何が起きるか？（あてはまるものをすべて選ぶ）

A

ほぼ遅れなし（押した瞬間に変わる）

B

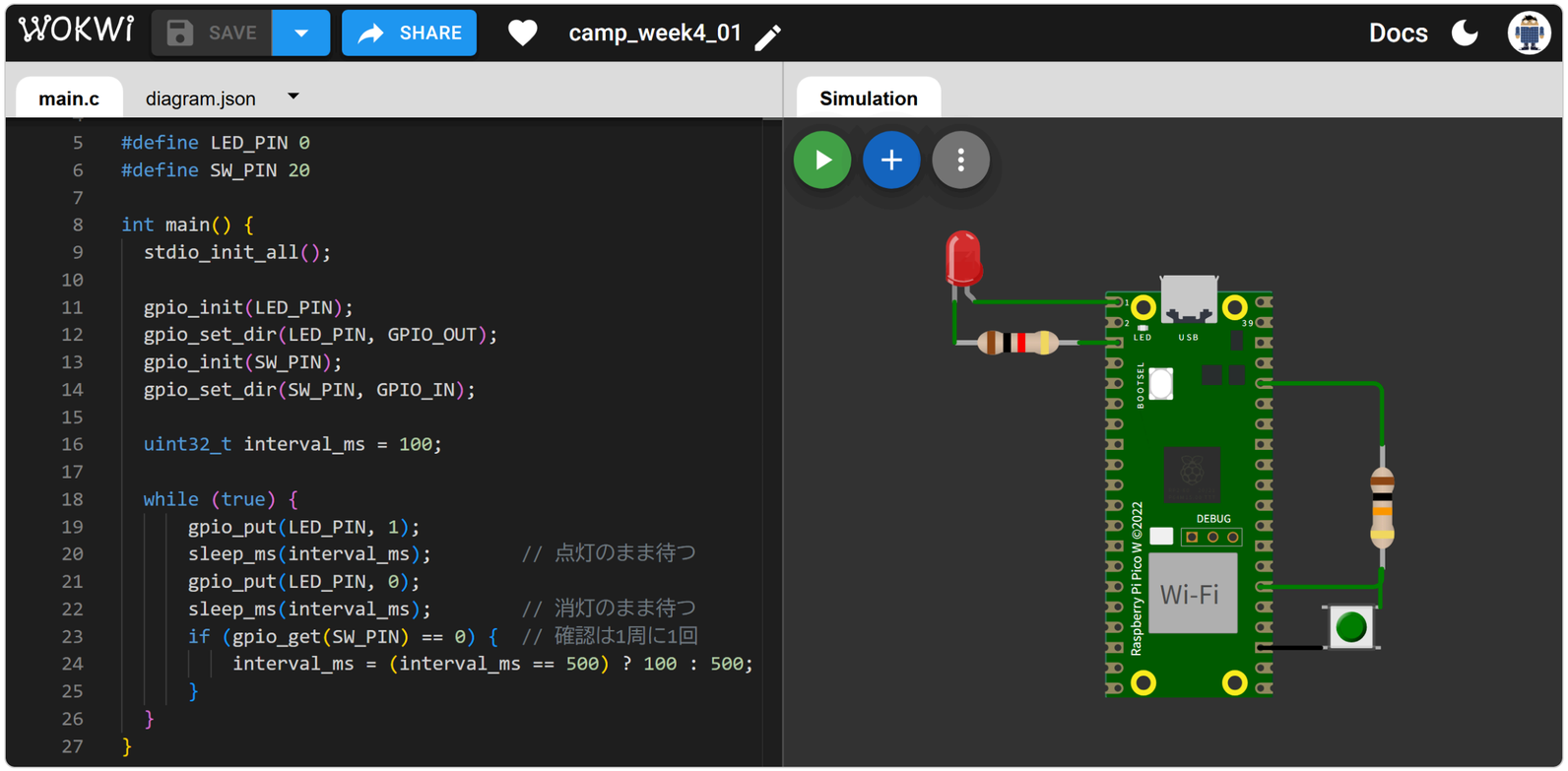
最大で約 1 秒 遅れることがある

C

そもそも反応しない

WORK 2.2 手を動かす①：Wokwi で試す

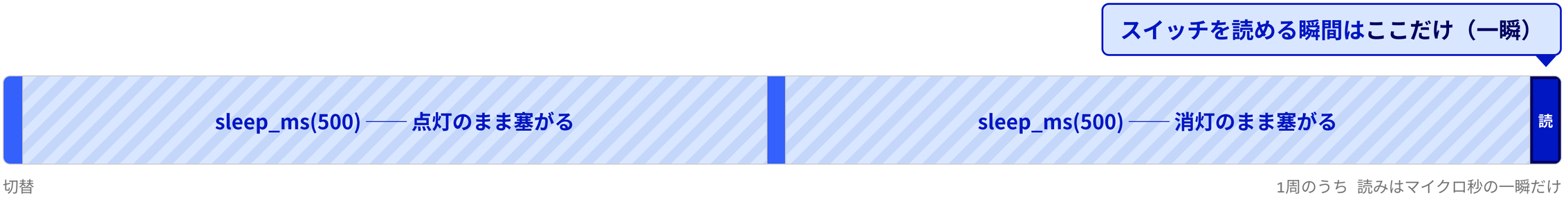
Wokwi のシミュレータで素朴版を動かし、ボタンを押した瞬間と点滅速度が変わる瞬間のズレ（遅れ）を観察する



Wokwi プロジェクト camp_week4_01 LINK wokwi.com/projects/470318028620099585

sleep_ms(500) は戻るまで CPU を占有し続ける呼び出し（ブロッキング）。その間は他の仕事が進まない

1周 ≈ 1000 ms のタイムライン



× 押下への応答 は次に読まれるまで返らない。1周 ≈ 1000 ms のほぼ全部が塞がった時間 —— 衛星なら sleep 中に届いたコマンドへの反応が最大約 1 秒遅れる。

sleep 中に始まって終わる短い入力は、末尾の 1 回の読みに残らず取りこぼされる

原因は遅れと同じ。1 周に 1 回、しかも**「今このピンが押されているか」という現在のレベル**しか見ていない。だから押されていた瞬間の長さで結果が分かれる。

押しっぱなし

△ 遅れて拾われる

一瞬押す

× 取りこぼし

直し方の芽 レベルではなく**変化した瞬間（エッジ）**を捉えれば取りこぼさない —— その仕組みが第4段の**割り込み**（5章）。衛星では一瞬のコマンドや異常を取りこぼすのは、遅れよりむしろ恐い。

素朴版の「遅れ」も「取りこぼし」も、待っている間 CPU が塞がる ブロッキングという一つの問題から生じる

覚える ①

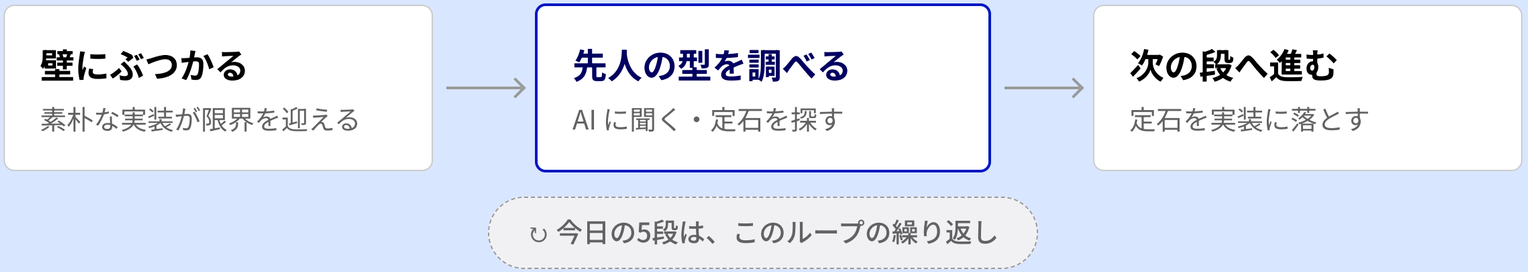
sleep_ms()は CPU を返さない処理（ブロッキング）

覚える ②

**塞がった時間は応答の遅れになり、
その間に始まって終わる短い入力は取りこぼされる**

Chapter 3　壁の前でやること：先人の型を調べる

「LED 点滅とスイッチ」という単純なお題にも、組み込み開発の確立された定石が既にそろっている



3.1 第2段：ノンブロッキング化（time_us_64）

現在時刻が締め切りを過ぎたら実行し、前回の締め切りに周期を足して次を予約する

```
// [Code-02] 第2段：ノンブロッキング版（骨格）
uint64_t next_toggle_us = time_us_64();
while (true) {
    uint64_t now = time_us_64();
    if (now >= next_toggle_us) {          // 締め切りが来たか
        led_on = !led_on;
        gpio_put(LED_PIN, led_on);
        next_toggle_us += interval_ms*1000; // 次を予約
    }
    if (gpio_get(SW_PIN) == 0) {          // 毎周確認できる
        interval_ms = (interval_ms == 500) ? 100 : 500;
    }
}
```

① **読む** 現在時刻

② **比べる** 締め切りと

③ **予約する** 前回の締め切りに周期を足す（ずれない）

sleep が1行もない。

⚠ スイッチの if は“レベル”判定。高速ループでは押している間ずっと発火 → 次スライドで修正

実際のコード
wokwi.com/projects/470353078971828225

高速ループで“今押されているか”を見ると、1回の押下の間に何千回も反転する。“押した瞬間”＝立下り（1→0）だけを1回拾う

```
// [Code-03] 立下りエッジを自前で検出
bool sw_prev = 1;                   // 前回の読み値（初期は H）
while (true) {
    // …点滅の締め切り判定は第2段のまま…
    bool sw = gpio_get(SW_PIN);     // 今回の読み値
    if (sw_prev == 1 && sw == 0)    // H→L の“その瞬間”だけ
        interval_ms = (interval_ms == 500) ? 100 : 500;
    sw_prev = sw;                   // 次回のために保存
}
```

× レベルで判定

`gpio_get() == 0`

押している間ずっと真 → 毎周反転して制御不能

◯ エッジで判定

`prev == 1 && now == 0`

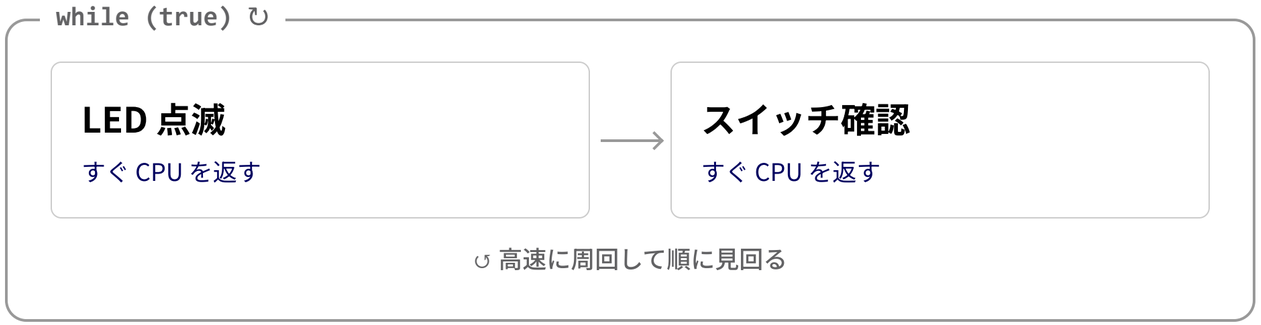
変化した1瞬間だけ真 → 1押下＝1回だけ反転

ハードに見張らせる割り込みは **5章** で。

実際のコード

wokwi.com/projects/470379759720116225

高速に回りながら各仕事の締め切りを確認する無限ループ＝スーパーループ（協調的マルチタスク）



各仕事が**「すぐ CPU を返す」行儀**を守ることで成立（だから協調的）。

組み込みの現場で広く使われる基本形。合宿のリアルタイム設計の出発点。

まず型を調べる。今回はスーパーループという型を学ぶ。

覚える ①

困ったら、まず先人の型を調べる（AI に聞いてよい）

覚える ②

ノンブロッキング＝待たずに経過を確認

覚える ③

その型がスーパーループ（協調的マルチタスク）

Chapter 4　第3段：タスクを足して同じ形に載せる

センサ読みもテレメトリ送信も、点滅と同じ「締め切りが来たかを確認して実行し、次を予約する」形で足せる

点滅

周期：短

締め切りが来たか？→実行→次を予約

センサ読み

周期：短

締め切りが来たか？→実行→次を予約

テレメトリ送信

周期：長

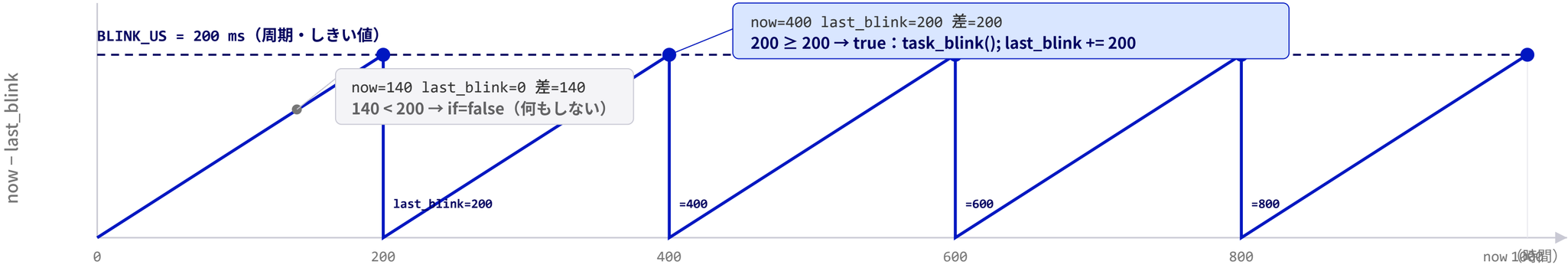
締め切りが来たか？→実行→次を予約

どのカードも同じ形 —— 周期の数字が違うだけ

タスクごとに周期と前回の締め切りを持たせると、タスクの追加は if ブロック1つで済む

```
while (true) {
    uint64_t now = time_us_64();
    if (now - last_blink >= BLINK_US) { last_blink += BLINK_US; task_blink(); }
    if (now - last_sense >= SENSE_US) { last_sense += SENSE_US; task_sense(); }
    if (now - last_telem >= TELEM_US) { last_telem += TELEM_US; task_telem(); }
    task_check_switch();              // イベントの確認は毎周
}
```

task_blink だけを追う —— 毎周 now − last_blink を測り、それが周期 BLINK_US に届いた瞬間だけ if が true になって実行する



各タスクは**順番に実行** —— CPU は1周のなかで1つずつ処理するので、複数のタスクが同時に走って重複することはない。

QUIZ 4.2 考えてみよう

このスーパーループに、1回の実行に 50 ms かかる画像処理タスクを足すと、10 ms 周期のタスクはどうなるか？

A

スケジューラが途中で切り替えるので、
影響はほぼ出ない

B

画像処理の実行中は走れず、
締め切りを最大で約 50 ms 過ぎる

C

画像処理だけ自動的に後回しにされ、
10 ms 周期は保たれる

タスクは同じ形で増やせるが、1つでも実行の長い処理を挟むと、他のタスクの締め切りをまとめて壊す

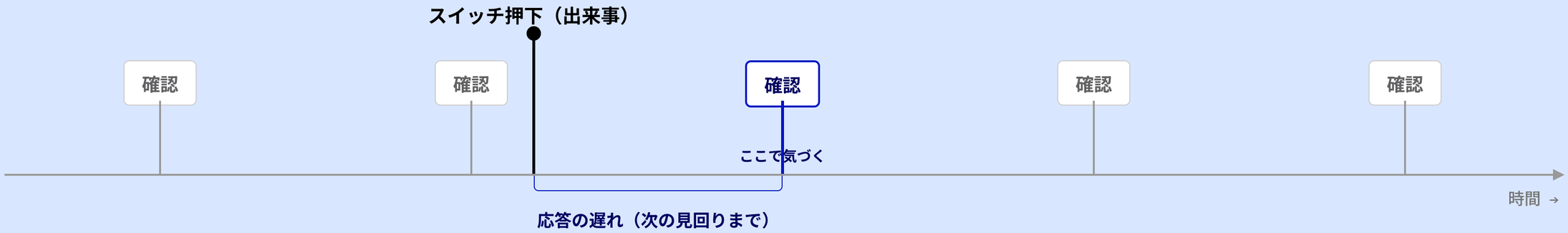
覚える ①

タスクは「周期＋前回の締め切り＋確認」の同じ形に載る

覚える ②

長い処理を1つ挟むと、他の全タスクが締め切りを落とす

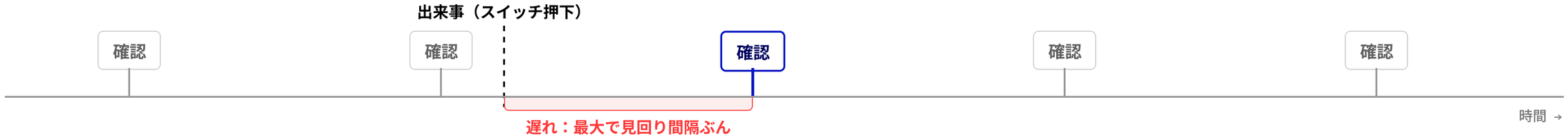
ソフトウェアが一定間隔で「来た？」と見に行くポーリングでは、イベントは確認の隙間で起こり、次の見回りまで気づけない



見回りの間隔が、そのまま応答の遅れの最大値になる。イベントを見逃すこともありうる。

同じ出来事でも、ポーリングは次の見回りまで遅れる。割り込みは瞬間に処理を中断して ISR へ飛ぶため、遅れが小さく一定になる

ポーリング：ソフトウェアが見に行く―― 隙間で起きた出来事は次の見回りまで待つ



割り込み：ハードウェアが知らせる―― 出来事の瞬間に中断して ISR へ飛ぶ

① 事前に「このピンが変化したら ISR を呼べ」と登録しておく

② 中断と復帰はハードが自動 ―― メインループは気づかない

WORK 5.2 手を動かす③：スイッチを割り込みで受ける

立下りエッジを割り込み要因として登録すると、押下の瞬間にメインの処理を中断して ISR が呼ばれる

```
// [Code-04] 第4段：GPIO 割り込み（骨格）

volatile uint32_t interval_ms = 500;   // 点滅間隔（ISR と共有）

void on_switch(uint gpio, uint32_t events) {
    interval_ms = (interval_ms == 500) ? 100 : 500;  // 間隔を切替
}

// main() で登録：押下＝GND への立下りエッジ
gpio_set_irq_enabled_with_callback(SW_PIN, GPIO_IRQ_EDGE_FALL, true, &on_switch);
```

共有変数に volatile

`volatile uint32_t interval_ms`

メインループにとって
知らないタイミングで値が変わりうることをコンパイラに示すため

実際のコード

wokwi.com/projects/470384930874677249

ハードウェアタイマ(によるタイマ割り込み)は、メインの処理と無関係に一定間隔で ISR を呼び続ける

```
// [Code-05] タイマ割り込みで点滅を刻む

bool on_blink_timer(repeating_timer_t *t) {
    led_on = !led_on;
    gpio_put(LED_PIN, led_on);
    return true;    // true で繰り返し継続
}
add_repeating_timer_ms(-500, on_blink_timer, NULL, &timer);
```

第1引数は負値で指定

`add_repeating_timer_ms(-500, …)`

Pico SDK の仕様。**負値は「前回の開始時刻」から**次を数えるため、ISR の実行時間を差し引いた**正確な 500 ms 周期**になる（正値は「前回の終了時刻」からで、処理時間の分だけ周期が延びる）

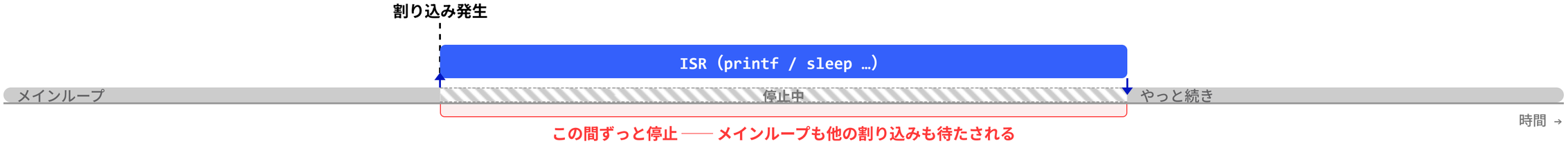
実際のコード

wokwi.com/projects/470386323683712001

ISR は状態を更新する数行にとどめ、重い処理はメインループ側で行う

◯ 短い ISR：状態を更新して即座に戻る —— メインループの乱れは最小

✕ 重い ISR（printf / sleep …）：長く居座り、その間すべてが止まる



5.6 この章のまとめ ／ ここは覚えて次へ

割り込みで応答の遅れは一定・極小になる。ただし ISR は短く保ち、共有変数には volatile を付ける

覚える ①

割り込み＝出来事の瞬間にハードウェアが CPU を呼ぶ仕組み

覚える ②

ISR は短く保ち、状態の更新だけにとどめる

覚える ③

ISR とループで共有する変数には `volatile` を付ける

CPU がやっていた定型の仕事をハードウェアに設定として渡すと、以降 CPU は関与せずに動き続ける

① ソフトのループ
CPU 関与：大（毎周確認）

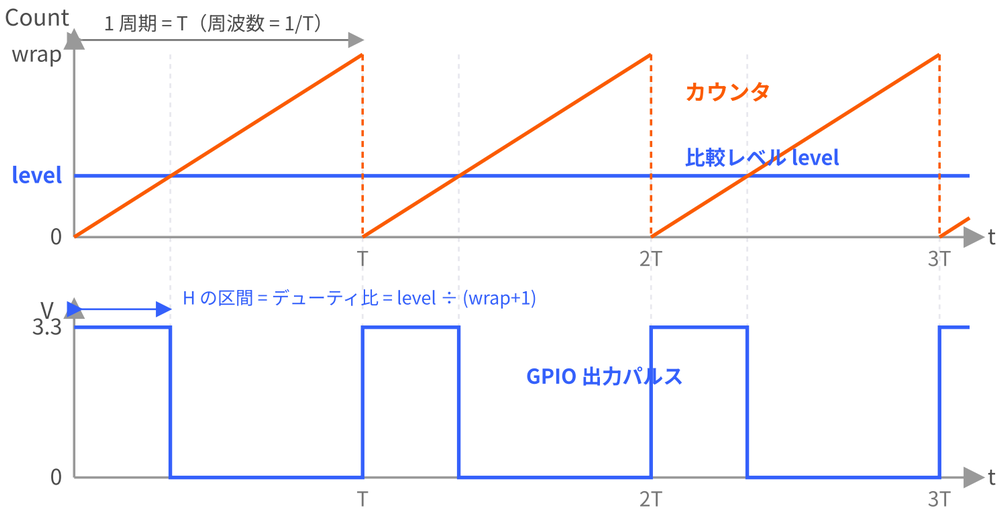
→

② タイマ割り込み
CPU 関与：小（切替時だけ）

→

③ ハードへ委譲（PWM）
CPU 関与：ゼロ（設定後）

カウンタが 0→wrap を1周する時間が「周期」、その中で H になる割合が「デューティ比」



カウンタ（count）が比較レベル（level）を超えると出力が H→L に切り替わる —— その幅がデューティ比。

周期（＝周波数）

カウンタが **0 → wrap** を1周する時間が1周期。周期が短いほど高い周波数で、**wrap（と clkdiv）**で決まる。

```
周波数 = クロック ÷ (clkdiv × (wrap+1))
```

デューティ比

1周期のうち出力が **H の割合**。比較レベル **level** で決まり、明るさやサーボ指令の「量」を表す。

```
デューティ比 = level ÷ (wrap+1)
```

数行の設定で委譲でき、以降 CPU は関与しない。
ただし速すぎて点滅には不向きで、明るさ制御やサーボ指令に使う。

```
// [Code-06] 第5段：PWM に任せる（骨格）

gpio_set_function(LED_PIN, GPIO_FUNC_PWM);
uint slice = pwm_gpio_to_slice_num(LED_PIN);
pwm_set_clkdiv(slice, 125.0f);   // 125 MHz → 1 MHz
pwm_set_wrap(slice, 999);        // 0〜999 で折り返し → 1 kHz
pwm_set_chan_level(slice, ch, 500);  // デューティ比 50%
pwm_set_enabled(slice, true);    // 以降 CPU は関与しない
```

実際のコード
wokwi.com/projects/470426236921829377

速すぎて点滅には不向き

16 ビットのカウンタと分周の上限で、基本的には高速動作。
1 Hz のゆっくりした点滅は PWM 単体では作れない。

LED の明るさ制御 —— 速い点滅がデューティ比＝明るさに融ける

サーボモータの指令 —— パルス幅で角度・量を伝える（模擬衛星：リアクションホイール）

定型の仕事をハードに任せる手段は複数ある —— PWM は最も簡単な入口で、実務の主流は DMA・PIO

今日扱う

PWM

周期的な H/L 波形を作る専用ペリフェラル。数行の設定だけで委譲できる。使い方は簡単。

主流・難しい

DMA

Direct Memory Access。CPU を介さず、メモリ⇄ペリフェラル間で大量のデータを運ぶ。取り込みの定番。

主流・難しい

PIO

Programmable I/O。RP2040 特有。任意の信号プロトコルをハードで生成・受信できる。自由度が高い。

共通点：いずれも「CPU を介さず、定型の仕事をハードに任せる」委譲。DMA・PIO は強力だが習得が難しいので、今日は概念の紹介までにとどめる。

定型の仕事はハードへ委譲して CPU を空ける。PWM はその簡単な入口で、実務の主流は DMA・PIO

覚える ①

ハードへの委譲＝定型の仕事をハードに任せ、CPU を本業に空ける（今日は PWM で体験）

覚える ②

委譲の主流は DMA・PIO（強力だが難しい）。CPU 時間は配分すべき有限の資源

他にも方法がある

マルチコア　物理的に複数のコアで真の同時実行（RP2040 は core0 / core1 の2コア）

これまで：並行（1コアを時分割）

1個の CPU を高速に切り替え、同時に見せる。スーパーループ・割り込みはこれ。

マルチコアで可能に：並列（真の同時）

2つのコアが文字どおり同時に走り、仕事を物理的に分担（姿勢制御と通信を別コアへ）。

代償：同じ変数への同時書き込みが起き、排他制御（スピンロック・ミューテックス）が要る。

RTOS　締め切りで強制的にタスクを切り替える OS（FreeRTOS が定番）

考え方：仕事をタスクに分け、それぞれに優先度を付ける。OS が実行するタスクを選び、より優先度の高いタスクの出番が来たら、実行中のタスクを途中で強制的に止めて切り替える（プリエンプション）。スーパーループのように各処理の「行儀」に頼らないのが違い。

得意：締め切りの厳しいタスクを優先して確実に間に合わせられる。タスクごとに独立して書けるので、処理が増えても構造が破綻しにくい。

代償：マルチコア同様、タスク間で共有するデータの競合が起きるため排他制御が要る。OS の分だけ学習コストとメモリ・実行時間のオーバーヘッドも増える。

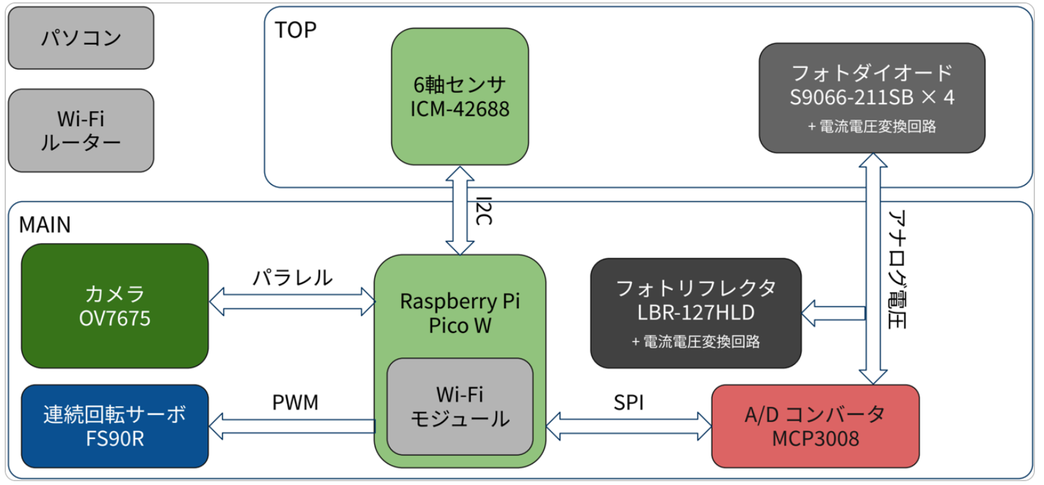
まとめ ／ 今日の解空間 —— 5つの型とその代償

まず解空間と設計の型を調べてから書き始める。知らない方法は選べない。

選択肢	得意	代償
スーパーループ	単純・見通しがよい	ジッタ・優先度に弱い
割り込み	出来事に遅れなく即応答	共有資源の競合
ハードへ委譲	定型仕事を CPU 抜きで継続	DMA・PIO は習得が難しい
マルチコア	真の並列で仕事を分担	排他制御が要る
RTOS	強制切り替えで締め切り厳守	複雑さ・競合が常駐

第2段階へ —— 宇宙機の要求と制約から開発の進め方を学ぶ

- ミッション要求とサクセスクライテリアの全体像を提示
- 制約がＶ字モデルやモデルベース開発を必然にする理由



Week 5 からは、この全体をどう設計・運用するかへ視点が上がる

宿題 10.2 チャタリング込みで、スイッチ切り替えを完成させる

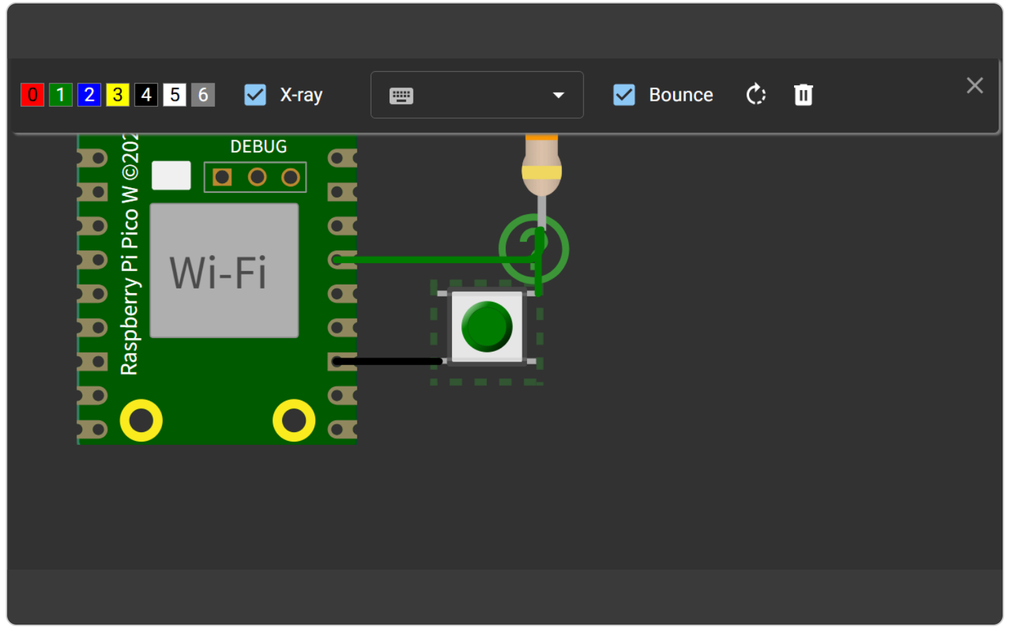
今日のお題「スイッチで点滅速度を切り替える」を、Wokwi の「Bounce」を有効にしたまま実装する

設定：Wokwi の「Bounce」（チャタリング模擬）を**有効（＝既定）のまま。**
1回の押下で接点が数十回開閉する

①スイッチ単体で printf で検証：LED は関係なくスイッチの H/L のみを printf 文で出力し、チャタリングの様子を観測する。

②チャタリング対策の実装：LED の点滅をスイッチ（チャタリングあり・対策あり）で切り替えるプログラムを実装する。

回答（提出）：①②のコード(Wokwiのリンク)に加え、
採用した手法／広げた選択肢／最終的な選定理由 の3点を書く



Wokwi の「Bounce」を有効にしたまま（画面上部のチェック）