mcr_camera_2026base
プログラム設計仕様書

GR-PEACH (RZ/A1H) マイコンカーラリー カメラクラス開発

バージョン　1.0

作成日　2026 年 2 月 19 日

作成者　Shiozawa

ステータス　ドラフト

本文書は `mcr_camera_2026base` プロジェクトのプログラム設計仕様書です。
ベアメタル環境における割り込み駆動アーキテクチャの詳細設計を記述します。

改訂履歴

版	日付	著者	変更内容
1.0	2026-02-19	Shiozawa	初版作成

目次

1 概要

GR-PEACH (RZ/A1H) を使用した、マイコンカーラリー (MCR) カメラクラス開発のためのベースプログラムである。

▷ 設計思想

- mbed OS を使用せず、ルネサス標準の `iodefine.h` を用いたベアメタル（レジスタ直接操作）環境で構築する
- 既存のスパゲッティコードを廃し、クラスベースの `init/update` パターンを採用したモダンなアーキテクチャを目指す

1.1 用語定義

用語	定義
MCR	マイコンカーラリーの略称
GR-PEACH	ルネサス RZ/A1H 搭載の開発ボード
MTU2	マルチファンクション・タイマ・ユニット 2。PWM 出力に使用
ラッチ方式	セッターで値を内部変数に保持し、`update()` 呼び出し時にのみハードウェアへ反映する方式
IModule	全モジュール共通の基底インターフェース (§ 7.1)

2 背景・課題

⚠ 開発環境の課題

1. **mbed OS サービス終了**: mbed のサービス終了に伴い、将来的な継続利用や保守が困難になる
2. **開発環境の老朽化**: 既存プロジェクトの e2 studio 環境やコンパイラバージョンが古く、最新の環境でビルドを通す必要がある
3. **ブラックボックス化の解消**: OS 内部処理への依存を減らし、デバッグを容易にする必要がある

3 目的

1. mbed OS に依存しない、RZ/A1H 用の C++ ベースプログラム環境の構築
2. **クラス化・モジュール化**による可読性・保守性の向上
3. 共通インターフェース (`IModule`) による統一的な制御フローの確立

4 システムアーキテクチャ

システムはタイマー割り込みによる定期実行（1 ループ/1ms を想定）で完全に同期して動作する。処理は 3 つのフェーズに分離される。

4.1 実行フロー図

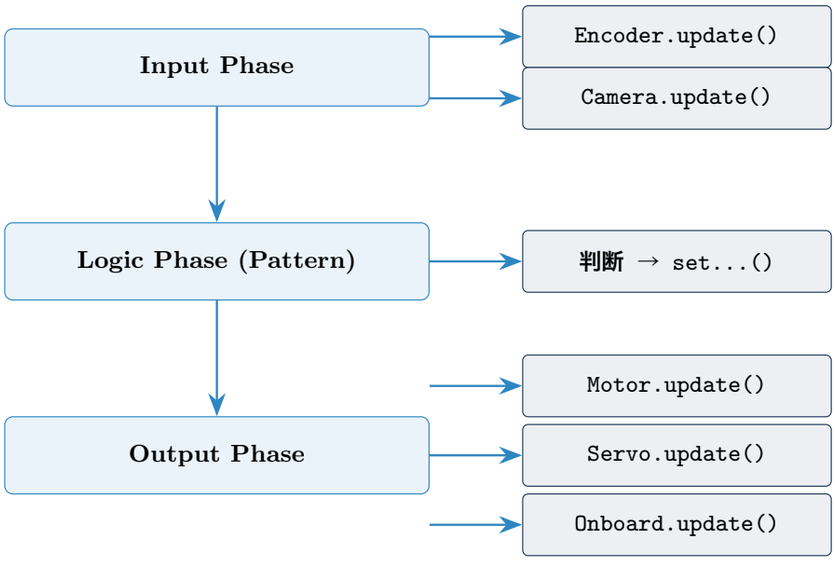

図 1: 割り込みハンドラ実行フロー

4.2 3 フェーズモデル

順序	フェーズ	内容
1	Input Phase	センサ値の読み取り（Encoder, Camera）
2	Logic Phase	判断ロジック実行、各ドライバの `set...()` を呼び出し（**この時点では出力しない**）
3	Output Phase	全ドライバの `update()` を呼び出し、ハードウェアへ反映

▷ ラッチ方式の利点

Logic Phase でセットした値は即座にハードウェアへ反映されない。Output Phase で一括反映することで、割り込み内での出力タイミングを統一し、グリッチや不整合を防止する。

5 ファイル構成

5.1 ディレクトリ構造

```
mcr_camera_test1/
  src/
    mcr_camera_test1.cpp   ← メイン・割り込みハンドラ
    core/
      IModule.h   ← 基底インターフェース
    drivers/
      Camera.h / Camera.cpp   ← カメラ制御
      Encoder.h / Encoder.cpp   ← エンコーダ制御
      Motor.h / Motor.cpp   ← モータ制御
      Onboard.h / Onboard.cpp   ← LED・スイッチ制御
      SDCard.h / SDCard.cpp   ← SD カード制御
      Serial.h / Serial.cpp   ← シリアル通信
      Servo.h / Servo.cpp   ← サーボ制御
    modules/
      Trace.h / Trace.cpp   ← ライントレースロジック
    utils/   ← ユーティリティ (予定)
  generate/
    iodefine.h   ← レジスタ定義 (自動生成)
  doc/
    main.tex   ← 本仕様書
```

5.2 ディレクトリ役割

ディレクトリ	役割
src/core/	システム基盤（IModule インターフェース）
src/modules/	ロジックモジュール（Trace 等 — 未実装）
src/drivers/	ハードウェアドライバ（Motor, Servo, Encoder, Camera, Serial, SDCard, Onboard）
src/utils/	ユーティリティ関数
generate/	e2 studio 自動生成ファイル（iodefine.h 等）
doc/	ドキュメント

5.3 全ファイル一覧

ファイル	役割
src/mcr_camera_2026base.cpp	メインエントリ・割り込みハンドラ・Pattern ロジック
src/core/IModule.h	全モジュール共通の基底インターフェース
src/drivers/Motor.h/.cpp	モータ PWM 制御 (MTU2 ch3/4)
src/drivers/Servo.h/.cpp	サーボ PWM 制御 (MTU2 ch0)
src/drivers/Encoder.h/.cpp	ロータリーエンコーダ読取 (MTU2 位相計数)
src/drivers/Camera.h/.cpp	カメラ画像取得・ライン検出 (CEU/DMA)
src/drivers/Serial.h/.cpp	シリアル通信 (SCIF)
src/drivers/SDCard.h/.cpp	SD カードログ書込み (SPI)

ファイル	役割
src/drivers/Onboard.h/.cpp	オンボード LED・スイッチ制御 (GPIO)
src/modules/Trace.h/.cpp	ライントレース・PID 制御ロジック

5.4 機能とファイルの対応表

機能	メインファイル	関連ファイル
割り込みハンドラ	mcr_camera_2026base.cpp	全ドライバ・モジュール
モータ制御	Motor.cpp	IModule.h, iodefine.h
サーボ制御	Servo.cpp	IModule.h, iodefine.h
エンコーダ	Encoder.cpp	IModule.h, iodefine.h
カメラ	Camera.cpp	IModule.h, iodefine.h
シリアル通信	Serial.cpp	IModule.h, iodefine.h
SD カード	SDCard.cpp	IModule.h, iodefine.h
LED・スイッチ	Onboard.cpp	IModule.h, iodefine.h
ライントレース	Trace.cpp	IModule.h, Camera.h, Encoder.h

6 技術スタック

項目	内容
ターゲットボード	GR-PEACH (ルネサス RZ/A1H, ARM Cortex-A9)
開発環境	Renesas e2 studio
言語	C++ (ベアメタル、mbed OS 不使用)
コンパイラ	GCC ARM (e2 studio 標準)
レジスタ定義	iodefine.h (e2 studio 自動生成)

7 API 仕様

本章では各クラスの公開 API をデータシート形式で記述する。全クラスは IModule インターフェース (§ 7.1) を継承する。

7.1 IModule インターフェース

IModule — 基底インターフェース

ファイル: src/core/IModule.h **種別:** 純粋仮想クラス

概要

全てのドライバおよびロジックモジュールが実装すべき共通インターフェース。init() で初期化、update() で周期処理を行う。

機能一覧

- 全モジュール共通の初期化・周期処理メソッドを定義
- 仮想デストラクタによる安全なポリモーフィズム

関数ドキュメント

◆ init()

```
virtual void init() = 0
```

初期化処理。レジスタ設定・変数初期化を行う。システム起動時に 1 回のみ呼び出される。

◆ update()

```
virtual void update() = 0
```

周期処理。割り込みハンドラまたはメインループから定期的に呼び出される。

実装例

```
class IModule {
public:
  virtual ~IModule() {}
  virtual void init() = 0;    // 初期化処理
  virtual void update() = 0; // 周期処理
};
```

Listing 1: IModule インターフェース定義

7.2 Encoder クラス

Encoder — エンコーダドライバ

ファイル: src/drivers/Encoder.h, Encoder.cpp **継承:** IModule Input Phase

概要

MTU2 (チャネル 1) の位相計数モード 2 を使用し、エンコーダのパルスをカウントする。フィルタリング処理を行い、走行距離と現在の速度を算出・管理する。

機能一覧

- トータル走行距離・コース走行距離の管理
- 現在速度の算出
- 距離の任意設定（クリア機能含む）

内部定数

定数名	値	説明
Filter_N	2	フィルタバッファサイズ
DIST_PER_PULSE	(定数)	1 パルスあたりの移動距離 [mm]

内部変数

変数名	型	初期値	説明
_totalDistance	float	0.0	トータル走行距離
_courseDistance	float	0.0	コース走行距離
_speed	float	0.0	現在速度
_deltaCount	int	0	今回のパルス変動量
_filter_buf[]	int[]	0	フィルタ用バッファ

関数ドキュメント

◆ init()

```
void init(void)
```

MTU2 の位相計数モード 2 を初期化する。GPIO 設定およびタイマスタートを行う。

◆ update()

```
void update(void)
```

MTU2 カウンタ値を読み取り、前回値との差分から距離・速度を更新する。

◆ getTotalDistance()

`float getTotalDistance(void)` — トータル走行距離 (_totalDistance) を返す。

◆ getCourseDistance()

`float getCourseDistance(void)` — コース走行距離 (_courseDistance) を返す。

◆ getSpeed()

`float getSpeed(void)` — 現在速度 (_speed) を返す。

◆ setTotalDistance(dist)

void setTotalDistance(float dist) — トータル走行距離を任意の値に設定する (0 でクリア)。

◆ setCourseDistance(dist)

void setCourseDistance(float dist) — コース走行距離を任意の値に設定する (0 でクリア)。

7.3 Camera クラス

Camera — カメラ制御および画像処理ドライバ

ファイル: src/drivers/Camera.h, Camera.cpp **継承:** IModule Input Phase

概要

関数ドキュメント (画像処理)

◆ resize(src, dst, percent)

void resize(uint8_t *src, uint8_t *dst, double percent)

(旧: ImageReduction) — 画像を縮小する。

◆ extractBrightness(src, dst)

void extractBrightness(uint8_t *src, uint8_t *dst)

(旧: Extraction_Brightness) — YCbCr 画像から輝度成文 (Y) を抽出する。

◆ binarize(src, dst, threshold)

void binarize(uint8_t *src, uint8_t *dst, int threshold)

(旧: Binarization) — 画像を指定閾値で二値化する。

◆ percentileMethod(src, percent)

int percentileMethod(uint8_t *src, int percent)

(旧: Percentile_Method) — パーセンタイル法による閾値決定を行う。

◆ discriminantAnalysis(src)

int discriminantAnalysis(uint8_t *src)

(旧: DiscriminantAnalysis_Method) — 判別分析法による閾値決定を行う。

◆ extractPart(src, x, y, dst, w, h)

```
void extractPart(uint8_t *src, int x, int y, uint8_t *dst, int w, int h)
```

(旧: Image_part_Extraction) — 画像の一部を切り出す。

◆ standardDeviation(...)

```
double standardDeviation(uint8_t *data, double *devi, int w, int h)
```

(旧: Standard_Deviation) — 標準偏差を算出する。

◆ covariance(...)

```
double covariance(double *deviA, double *deviB, int w, int h)
```

(旧: Covariance) — 共分散を算出する。

◆ matchPattern(...)

```
void matchPattern(uint8_t *src, ImagePartPattern *tmplt, ...)
```

(旧: PatternMatching_process) — パターンマッチングを行う。

⚠ 実装上の注意

- DisplayBase ライブラリのセットアップ手順は複雑なため、既存実装 (init_Camera) を参考に移植する。
- 画像処理関数は重いため、処理時間オーバーに注意が必要。

7.4 Trace クラス

Trace — ライントレースモジュール

ファイル: src/modules/Trace.h, Trace.cpp **継承:** IModule Logic Phase

概要

カメラ画像からライン位置を検出し、PID 制御によりステアリング角度を算出する。update() 内で検出→PID 計算の一連処理を実行する。

機能一覧

- 画像二値化によるライン検出
- PID 制御によるステアリング角度算出
- ゲインパラメータの動的設定
- ステアリング角度のリミット処理

内部定数

定数名	値	説明
THRESHOLD	未定	二値化閾値
STEER_MAX	未定	ステアリング角度上限
STEER_MIN	未定	ステアリング角度下限

内部変数

変数名	型	初期値	説明
_kp	float	0.0f	比例ゲイン
_ki	float	0.0f	積分ゲイン
_kd	float	0.0f	微分ゲイン
_integral	float	0.0f	積分値
_prevErr	float	0.0f	前回偏差
_steer	float	0.0f	算出ステアリング角度
_linePos	float	0.0f	検出ライン位置
_lineFound	bool	false	ライン検出成否

関数ドキュメント

◆ init()

`void init(void)`

PID パラメータおよび内部状態変数を初期化する。

◆ update()

`void update(void)`

ライン検出→ PID 計算の一連処理を実行する。内部で `binarize()` → `detectLine()` → `calcPid()` → `clampSteer()` の順に呼び出す。

◆ getSteer()

`float getSteer(void)` — 算出されたステアリング角度を返す。

◆ getLinePos()

`float getLinePos(void)` — 検出されたライン位置を返す。

◆ isLineFound()

`bool isLineFound(void)` — ラインが検出されたかどうかを返す。

◆ setKp(v)

void setKp(float v) — 比例ゲインを設定する。($v \geq 0$)

◆ setKi(v)

void setKi(float v) — 積分ゲインを設定する。($v \geq 0$)

◆ setKd(v)

void setKd(float v) — 微分ゲインを設定する。($v \geq 0$)

◆ detectLine(data) [内部]

float detectLine(uint8_t* data)

二値化済みの画像データからライン中心位置を算出する。

引数: data (uint8_t*) — 二値化済み画像データ
戻り値: float — ライン中心位置

◆ calcPid(error) [内部]

float calcPid(float error)

PID 制御量を計算する。積分値 (_integral) と前回偏差 (_prevErr) を内部で更新する。

引数: error (float) — 目標位置との偏差
戻り値: float — PID 制御量

◆ binarize(data) [内部]

void binarize(uint8_t* data)

画像データを閾値 (THRESHOLD) に基づいて二値化する。入力バッファを直接書き換える。

引数: data (uint8_t*) — 画像データ（in-place 変換）

◆ clampSteer(val) [内部]

float clampSteer(float val)

ステアリング角度を STEER_MIN〜STEER_MAX の範囲にクランプする。

引数: val (float) — クランプ前のステアリング角度
戻り値: float — クランプ後のステアリング角度

7.5 Motor クラス

Motor — モータ PWM ドライバ

ファイル: src/drivers/Motor.h, Motor.cpp **継承:** IModule Output Phase

概要

MTU2 チャネル 3/4 を使用したリセット同期 PWM モードで左右モータを制御する。ラッチ方式を採用し、setPower() でデューティ比を内部保持、update() でレジスタへ反映する。

機能一覧

- MTU2 リセット同期 PWM モードによるモータ駆動
- 左右独立のデューティ比制御 ($-1.0 \sim 1.0$)
- 符号による正転/逆転の方向制御
- ラッチ方式による安全な出力タイミング

内部定数

定数	値	説明
PWM_CYCLE	33332	PWM 周期カウント値 (1ms @ P0ϕ/1)

内部変数

変数名	型	初期値	説明
_pwmLeft	float	0.0f	左モータデューティ比 ($-1.0 \sim 1.0$)
_pwmRight	float	0.0f	右モータデューティ比 ($-1.0 \sim 1.0$)

関数ドキュメント

◆ init()

```
void init(void)
```

GPIO 方向ピンおよび MTU2 チャネル 3/4 のレジスタを初期化する。リセット同期 PWM モードを設定し、カウント動作を開始する。

◆ update()

```
void update(void)
```

内部変数のデューティ比をハードウェアへ反映する。内部で setDirection() → calcDuty() → applyPwm() の順に呼び出す。

◆ setPower(l, r)

```
void setPower(float l, float r)
```

左右モータのデューティ比を内部変数に保存する（ラッチ方式）。この時点ではハードウェアへの反映は行わない。

- `l` (`float`) — 左モータデューティ比 ($-1.0 \sim 1.0$、負値で逆転)
- `r` (`float`) — 右モータデューティ比 ($-1.0 \sim 1.0$、負値で逆転)

◆ calcDuty(pwm) [内部]

```
uint16_t calcDuty(float pwm)
```

デューティ比（浮動小数点数）を PWM カウント値に変換する。$duty = \lfloor |pwm| \times \texttt{PWM\_CYCLE} \rfloor$

引数: `pwm` (`float`) — デューティ比 ($-1.0 \sim 1.0$)
戻り値: `uint16_t` — PWM カウント値

◆ setDirection(l, r) [内部]

```
void setDirection(float l, float r)
```

デューティ比の符号に基づき、方向ピン (GPIO P4_6, P4_7) を設定する。正値で正転 (Low)、負値で逆転 (High)。

- `l` (`float`) — 左モータデューティ比
- `r` (`float`) — 右モータデューティ比

◆ applyPwm(l, r) [内部]

```
void applyPwm(uint16_t l, uint16_t r)
```

MTU2 レジスタ (TGRA_4/TGRC_4, TGRB_4/TGRD_4) へ PWM 値を書き込む。

- `l` (`uint16_t`) — 左モータ PWM カウント値
- `r` (`uint16_t`) — 右モータ PWM カウント値

ハードウェアマッピング

信号	ピン	レジスタ	説明
左モータ PWM	P4_4 (TIOC4A)	TGRA_4 / TGRC_4	デューティ比
右モータ PWM	P4_5 (TIOC4B)	TGRB_4 / TGRD_4	デューティ比
左モータ 方向	P4_6 (GPIO)	GPIO.P4 bit6	Low: 正転 / High: 逆転
右モータ 方向	P4_7 (GPIO)	GPIO.P4 bit7	Low: 正転 / High: 逆転

7.6 Servo クラス

Servo — サーボ PWM ドライバ

ファイル: src/drivers/Servo.h, Servo.cpp **継承:** IModule Output Phase

概要

MTU2 チャネル 0 を PWM モード 1 で駆動し、ステアリングサーボの角度を制御する。ラッチ方式を採用し、setAngle() で角度を内部保持、update() でレジスタへ反映する。

機能一覧

- MTU2 PWM モード 1 によるサーボ駆動
- 角度→パルス幅の自動変換
- ラッチ方式による安全な出力タイミング

内部定数

定数名	値	説明
SERVO_CENTER	3090	サーボ中心位置のパルス幅カウント値
HANDLE_STEP	23	1 度あたりのカウント値

内部変数

変数名	型	初期値	説明
_angle	float	0.0f	サーボ角度 (度)

関数ドキュメント

◆ init()

```
void init(void)
```

GPIO および MTU2 チャネル 0 のレジスタを初期化する。PWM モード 1 を設定し、カウント動作を開始する。

◆ update()

```
void update(void)
```

内部変数の角度値をハードウェアへ反映する。内部で calcPulse() → applyServo() の順に呼び出す。

◆ setAngle(angle)

```
void setAngle(float angle)
```

サーボ角度を内部変数に保存する（ラッチ方式）。0 が中心、正負で左右に振れる。

引数: angle (float) — サーボ角度 (度、機体依存)

◆ calcPulse(angle) [内部]

```
uint16_t calcPulse(float angle)
```

角度をパルス幅カウント値に変換する。$val = \lfloor angle \times 23 \rfloor + \texttt{SERVO\_CENTER}$

引数: angle (float) — サーボ角度 (度)
戻り値: uint16_t — パルス幅カウント値

◆ applyServo(val) [内部]

```
void applyServo(uint16_t val)
```

MTU2 レジスタ (TGRB_0 / TGRD_0) へパルス幅を書き込む。

引数: val (uint16_t) — パルス幅カウント値

ハードウェアマッピング

信号	ピン	レジスタ	説明
サーボ PWM	P4_0 (TIOC0A)	TGRB_0 / TGRD_0	パルス幅 (立ち下がりエッジ)

7.7 Onboard クラス

Onboard — オンボード LED / スイッチ ドライバ

ファイル: src/drivers/Onboard.h, Onboard.cpp **継承:** IModule Output Phase

概要

GR-PEACH ボード上の RGB LED (3 個) + ユーザー LED (1 個) およびユーザースイッチ (1 個) を制御する。LED はラッチ方式、スイッチは即時読み取り。

機能一覧

- RGB LED 3 個 + ユーザー LED 1 個の制御
- ラッチ方式による LED 出力
- ユーザースイッチの即時読み取り

内部変数

変数名	型	初期値	説明
_ledState[4]	int[4]	{0,0,0,0}	各 LED 状態 (0: 消灯, 1: 点灯)

関数ドキュメント

◆ init()

`void init(void)`

GPIO 初期化を行う。全 LED を消灯し、スイッチを入力モードに設定する。PIBC6 の入力バッファ有効化を含む。

◆ update()

`void update(void)`

内部変数の LED 状態を GPIO へ反映する。内部で `writeLed()` を各 LED ID に対して呼び出す。

◆ setLed(id, val)

`void setLed(int id, int val)`

指定 LED の状態を内部変数に保存する（ラッチ方式）。

- `id` (`int`) — LED 番号 (0:Red, 1:Green, 2:Blue, 3:User)
- `val` (`int`) — 0: 消灯, 1: 点灯

◆ sw()

`int sw(void)`

ユーザースイッチの状態を即時読み取りで返す。ラッチ方式**ではなく**、呼び出し時に GPIO を直接読む。内部で `readSw()` を呼び出す。

戻り値: `int` — 1: 押下, 0: 解放

◆ writeLed(id, val) [内部]

`void writeLed(int id, int val)`

指定 LED の GPIO を直接操作する。RGB LED は Low Active、ユーザー LED は High Active。

- `id` (`int`) — LED 番号
- `val` (`int`) — 論理的な点灯/消灯状態

◆ readSw() [内部]

`int readSw(void)`

SW 用 GPIO (P6_0) を読み取り、論理値に変換して返す。Low = 押下。

戻り値: `int` — 1: 押下, 0: 解放

ハードウェアマッピング

デバイス	ID	ピン	論理	説明
LED Red	0	P6_13	Low Active	赤色 LED
LED Green	1	P6_14	Low Active	緑色 LED
LED Blue	2	P6_15	Low Active	青色 LED
LED User	3	P6_12	High Active	ユーザー LED
SW User	–	P6_0	Low = 押下	プルアップ、GND 接続

⚠ GPIO 注意事項

- RGB LED は **Low Active**（GPIO Low で点灯）
- スイッチ読み取りには `PIBC6` の入力バッファ有効化が必須
- `sw()` はラッチ方式**ではなく**、呼び出し時に直接 GPIO を読む

7.8 Serial クラス

Serial — シリアル通信ドライバ

ファイル: `src/drivers/Serial.h`, `Serial.cpp` **継承:** `IModule` Output Phase

概要

SCIF (Serial Communication Interface with FIFO) を使用してデバッグ用シリアル通信を行う。送信はバッファリングし、`update()` で順次送出する。

機能一覧

- SCIF によるシリアル通信
- リングバッファによる送信バッファリング
- 書式付き出力 (`printf`)
- `update()` による非ブロッキング送出

内部定数

定数名	値	説明
`BAUD_RATE`	115200	ボーレート
`TX_BUF_SIZE`	256	送信バッファサイズ (bytes)

内部変数

変数名	型	初期値	説明
_txBuf[]	char[]	0	送信リングバッファ
_txHead	uint16_t	0	書込みポインタ
_txTail	uint16_t	0	読出しポインタ

関数ドキュメント

◆ init()

`void init(void)`

SCIF の初期化・ボーレート設定を行う。送信バッファをクリアする。

◆ update()

`void update(void)`

送信バッファの内容を FIFO へ順次送出する。FIFO 空き状況に応じて部分送出する。内部で `flushTx()` を呼び出す。

◆ print(str)

`void print(const char* str)`

文字列を送信バッファに追加する。バッファ満杯時は古いデータを上書き。

引数: `str` (`const char*`) — 送信文字列

◆ printf(fmt, ...)

`void printf(const char* fmt, ...)`

書式付き文字列を送信バッファに追加する。`vsnprintf` を内部で使用。

引数: `fmt` (`const char*`) — 書式文字列（`printf` 互換）

◆ putChar(c) [内部]

`void putChar(char c)` — 1 文字を FIFO へ書き込む。

◆ isTxReady() [内部]

`bool isTxReady(void)` — 送信 FIFO に空きがあるかを返す。

◆ flushTx() [内部]

`void flushTx(void)` — バッファ内容を可能な限り一括送出する。

7.9 SDCard クラス

SDCard — SD カードログドライバ

ファイル: src/drivers/SDCard.h, SDCard.cpp **継承:** IModule Output Phase

概要

SPI 経由で SD カードへログデータを書き込む。データはバッファリングし、update() でブロック単位で書き出す。

機能一覧

- SPI 経由の SD カードアクセス
- バッファリングによる効率的なブロック書込み
- ログ文字列の追記
- 明示的フラッシュ・クローズ操作

内部定数

定数名	値	説明
BLOCK_SIZE	512	SD ブロックサイズ (bytes)
WRITE_BUF_SIZE	1024	書込みバッファサイズ (bytes)

内部変数

変数名	型	初期値	説明
_writeBuf[]	uint8_t[]	0	書込みバッファ
_bufPos	uint16_t	0	バッファ書込み位置
_isReady	bool	false	SD 初期化完了フラグ
_blockAddr	uint32_t	0	現在の書込みブロックアドレス

関数ドキュメント

◆ init()

```
void init(void)
```

SPI 初期化および SD カードの初期化シーケンス (CMD0 → CMD8 → ACMD41) を実行する。初期化成功で _isReady を true に設定。

◆ update()

`void update(void)`

バッファが 1 ブロック分 (512B) 以上溜まっている場合、SD カードへブロック単位で書き出す。内部で `writeBlock()` を呼び出す。

◆ log(str)

`void log(const char* str)`

ログ文字列をバッファに追加する。

引数: `str` (`const char*`) — ログ文字列

◆ flush()

`void flush(void)`

バッファ内容を即時書き出す。残りデータが 512B 未満でもパディングして書き出す。

◆ close()

`void close(void)`

`flush()` を呼び出した後、ファイル終端処理を行う。SD 通信を終了し、`_isReady` を `false` に設定する。

◆ spiWrite(byte) [内部]

`uint8_t spiWrite(uint8_t byte)`

SPI 1 バイト送受信を行う。送信と同時に受信データを返す。

引数: `byte` (`uint8_t`) — 送信バイト
戻り値: `uint8_t` — 受信バイト

◆ sendCmd(cmd, arg) [内部]

`uint8_t sendCmd(uint8_t cmd, uint32_t arg)`

SD カードコマンドを送信し、レスポンスを返す。

- `cmd` (`uint8_t`) — SD コマンド番号
- `arg` (`uint32_t`) — コマンド引数

戻り値: `uint8_t` — R1 レスポンス

◆ writeBlock(data) [内部]

`bool writeBlock(const uint8_t* data)`

512B ブロックをアドレス `_blockAddr` に書き込む。成功で `_blockAddr` をインクリメント。

引数: `data` (`const uint8_t*`) — 書込みデータ (512B)
戻り値: `bool` — 書込み成功/失敗

◆ waitReady() [内部]

`bool waitReady(void)`

SD カードのビジー状態が解除されるまで待機する。タイムアウト時は `false` を返す。

戻り値: `bool` — ビジー解除成功/タイムアウト

8 実行モデル詳細

8.1 割り込みハンドラ

割り込みハンドラ関数内で各モジュールの `update()` を呼び出す。Pattern 処理はクラス化せず、ハンドラ内にベタ書き（または関数化）する。

```
// グローバルインスタンス
static Onboard onboard;
static Motor motor;
static Servo servo;

void interrupt_handler() {
    // --- Input Phase ---
    // encoder.read();
    // camera.read();

    // --- Logic Phase (Pattern) ---
    int sw = onboard.sw();
    if (sw) {
        motor.setPower(0.2f, 0.2f);  // PWM 20%
        servo.setAngle(10.0f);       // 度10
        onboard.setLed(0, 1);        // Red ON
        onboard.setLed(1, 1);        // Green ON
        onboard.setLed(2, 1);        // Blue ON
    } else {
        motor.setPower(0.0f, 0.0f);
        servo.setAngle(0.0f);
        onboard.setLed(0, 1);        // Red ON
        onboard.setLed(1, 0);        // Green OFF
        onboard.setLed(2, 0);        // Blue OFF
    }

    // --- Output Phase ---
    motor.update();
    servo.update();
    onboard.update();
    // serial.update();
    // sd.update();
}
```

Listing 2: 割り込みハンドラ実装例

8.2 ラッチ方式ドライバの動作原理

図 2: ラッチ方式ドライバのデータフロー