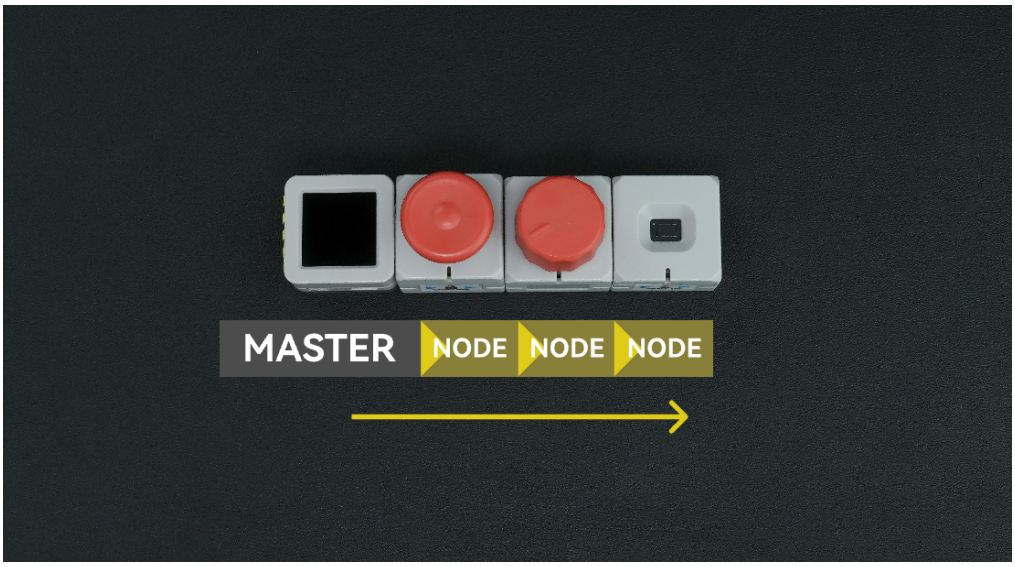
Chain シリーズデバイス Bus 通信使用チュートリアル

1. Chain Bus バス概要

M5Stack Chain Bus は、UART 通信インターフェースを採用したチェーン状のカスケード接続トポロジー構造であり、1 台のマスター（Master）と複数のノード（Node）で構成されます。マスターはリンクの起点として機能し、各ノードは同一バス上に順番に直列接続されます。システム動作中、各ノードはデータパケットを順次転送する方式でマスターと通信を行います。



マスター（Master）

マスターデバイスは **UART インターフェース** を介して Chain Bus と通信します。マスターは複数の Chain Bus バスを拡張することが可能で、その数は利用可能な UART インターフェースの数によって決まります。これにより、マルチバス・マルチノードの柔軟な拡張構成を実現できます。

ノード（Node）

Chain ノードは通常、コアコントローラーとして **STM32G031G8U6** を搭載し、**Chain シリーズ専用の UART シリアルカスケード通信プロトコル** を使用してデータのやり取りを行います。ノードには **2 個の HY2.0-4P 拡張インターフェース** が実装されており、それぞれ **データ入力（IN）** と **データ出力（OUT）** に使用されます。接続の際は方向に注意が必要で、マスターをノードのデータ入力（IN）に接続し、ノードのデータ出力（OUT）から次段の入力へと接続を続けます。

- Chain シリーズ マスター:
  - Chain DualKey
- Chain シリーズ コネクター:
  - Chain Bridge
  - Chain Return
- Chain シリーズ 入力デバイス:
  - Chain Joystick
  - Chain Key
  - Chain Angle
  - Chain Encoder
  - Chain ToF

Chain Bus のマスターとして、Chain DualKey または Atom シリーズのマスターに Atomic ToChain Base を組み合わせた構成を推奨します。

2. サンプルプログラム

コンパイル要件

M5Stack ボードマネージャ バージョン >= 3.2.4
M5Chain ライブラリ バージョン >= 1.0.0

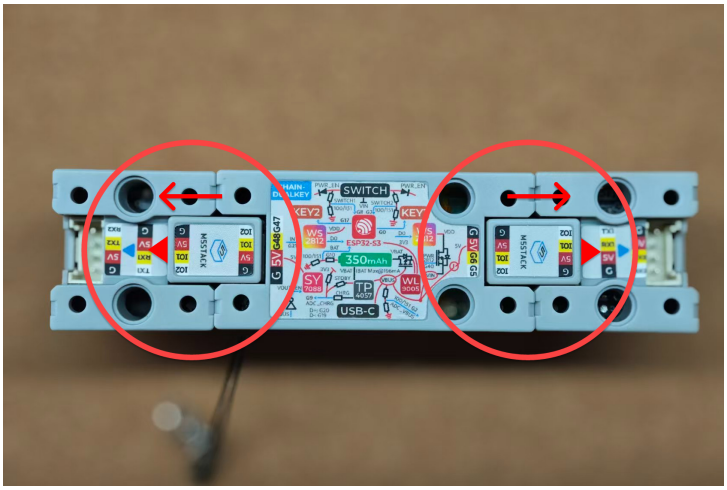
cpp

```cpp
#include "M5Chain.h"

#define RXD_PIN GPIO_NUM_5  // 47 for the other side of Chain DualKey
#define TXD_PIN GPIO_NUM_6  // 48 for the other side of Chain DualKey

Chain M5Chain;

chain_status_t chain_status;
device_list_t *device_list = NULL;
uint16_t device_count = 0;
uint8_t opr_status = 0;
uint8_t rgb_test[] = { 255, 255, 255 };

void setup() {
  Serial.begin(115200);
  delay(1000);
  Serial.println("======================");
  Serial.println("M5Stack Chain Bus Test");
  M5Chain.begin(&Serial2, 115200, RXD_PIN, TXD_PIN);
}

void loop() {
  Serial.println();
  delay(2000);

  if (!M5Chain.isDeviceConnected()) {
    Serial.println("No device connected");
    return;
  }

  chain_status = M5Chain.getDeviceNum(&device_count);
  if (chain_status == CHAIN_OK) {
    device_list = (device_list_t *)malloc(sizeof(device_list_t));
    device_list->count = device_count;
    device_list->devices = (device_info_t *)malloc(sizeof(device_info_t) * device_count);
  } else {
    Serial.printf("Get device count failed, chain status: %d\r\n", chain_status);
    return;
  }

  if (!M5Chain.getDeviceList(device_list)) {
    Serial.println("Get device list failed");
    return;
  }

  if (device_list == NULL) {
    Serial.println("Device list is NULL");
    return;
  }

```

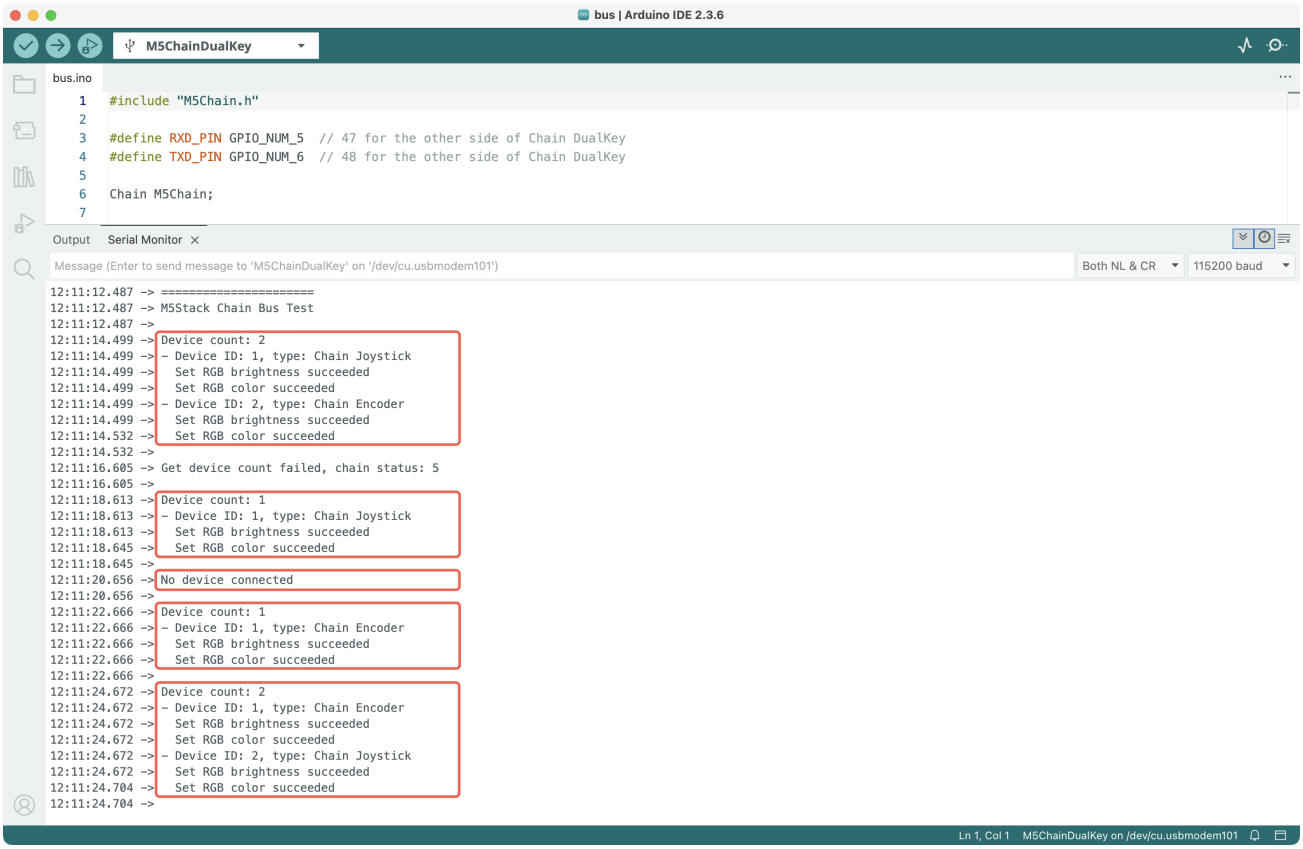
```
  Serial.printf("Device count: %d\r\n", device_list->count);

  for (int i = 0; i < device_list->count; i++) {
    Serial.print("- Device ID: ");
    Serial.print(device_list->devices[i].id);
    Serial.print(", type: ");
    switch (device_list->devices[i].device_type) {
      case CHAIN_UNKNOWN_TYPE_CODE:
        Serial.println("Unknown");
        break;
      case CHAIN_ENCODER_TYPE_CODE:
        Serial.println("Chain Encoder");
        break;
      case CHAIN_ANGLE_TYPE_CODE:
        Serial.println("Chain Angle");
        break;
      case CHAIN_KEY_TYPE_CODE:
        Serial.println("Chain Key");
        break;
      case CHAIN_JOYSTICK_TYPE_CODE:
        Serial.println("Chain Joystick");
        break;
      case CHAIN_TOF_TYPE_CODE:
        Serial.println("Chain ToF");
        break;
        // case CHAIN_UART_TYPE_CODE:
        //   Serial.println("Chain UART");
        //   break;
        // case CHAIN_SWITCH_TYPE_CODE:
        //   Serial.println("Chain Switch");
        //   break;
        // case CHAIN_PEDAL_TYPE_CODE:
        //   Serial.println("Chain Pedal");
        //   break;
        // case CHAIN_PIR_TYPE_CODE:
        //   Serial.println("Chain PIR");
        //   break;
        // case CHAIN_MIC_TYPE_CODE:
        //   Serial.println("Chain Mic");
        //   break;
        // case CHAIN_BUZZER_TYPE_CODE:
        //   Serial.println("Chain Buzzer");
        //   break;
    }

    // Device ID, LED brightness (0-100), operation status pointer
    chain_status = M5Chain.setRGBLight(device_list->devices[i].id, 100, &opr_status);
    if (chain_status == CHAIN_OK && opr_status) {
      Serial.println("  Set RGB brightness succeeded");
    } else {
      Serial.printf("  Set RGB brightness failed, chain status: %d, operation status: %d\r\n", chai
    }

```

```
    rgb_test[0] = random(0, 256);  // [0, 255]
    rgb_test[1] = random(0, 256);  // [0, 255]
    rgb_test[2] = random(0, 256);  // [0, 255]

    // Device ID, LED start index, LED count, RGB color, size of RGB color, operation status pointe
    chain_status = M5Chain.setRGBValue(device_list->devices[i].id, 0, 1, rgb_test, 3, &opr_status);
    if (chain_status == CHAIN_OK && opr_status) {
      Serial.println("  Set RGB color succeeded");
    } else {
      Serial.printf("  Set RGB color failed, chain status: %d, operation status: %d\r\n", chain_sta
    }
  }
}
```

Chain Bridge コネクタを使用して、メインコントローラの Chain DualKey と各 Chain シリーズ入力デバイスを接続します。接続時は方向に注意し、三角形の矢印がメインコントローラ Chain DualKey から外側を指すようにしてください。次の図を参照:



上記のプログラムをコンパイルしてデバイスに書き込みます。プログラムは 2 秒ごとに Chain DualKey の G5、G6 ピン側の Chain Bus に接続されているデバイスを検出し、シリアルモニタに出力します。また、RGB ライトの色をランダムに変更します。プログラムの実行中に複数の Chain シリーズデバイスをホットプラグすることができます。



Chain DualKey の両側インターフェースを同時に使用する場合、プログラム内で 2 つの Chain クラスのインスタンスを作成する必要があります。それぞれのインスタンスが、対応するインターフェースに接続されたデバイスを管理します。

3.API

操作結果ステータスコード

```
typedef enum {
    CHAIN_OK                  = 0x00,  // Operation successful
    CHAIN_PARAMETER_ERROR     = 0x01,  // Parameter error
    CHAIN_RETURN_PACKET_ERROR = 0x02,  // Return packet error
    CHAIN_BUSY                = 0x04,  // Device is busy
    CHAIN_TIMEOUT             = 0x05   // Operation timeout
} chain_status_t;
```

begin

関数プロトタイプ:

```
void begin(HardwareSerial *serial, unsigned long baud = 115200, int8_t rxPin = -1, int8_t txPin = -1);
```

機能説明:

- Chain Bus 通信バスを初期化する

入力パラメータ:

- `HardwareSerial *serial`
  - シリアル通信に使用するハードウェアシリアルオブジェクトへのポインタ

- `unsigned long baud`
  - シリアル通信のボーレート（デフォルト：115200）
- `int8_t rxPin`
  - 受信信号用 GPIO 番号
- `int8_t txPin`
  - 送信信号用 GPIO 番号

戻り値:

- null

isDeviceConnected

関数プロトタイプ:

```
bool isDeviceConnected();
```

機能説明:

- Chain Bus 上にデバイスが接続されているかどうかを確認する

入力パラメータ:

- null

戻り値:

- `bool`
  - `true`：デバイスが接続されている
  - `false`：デバイスが接続されていない

getDeviceNum

関数プロトタイプ:

```
chain_status_t getDeviceNum(uint16_t *deviceNum);
```

機能説明:

- Chain Bus に接続されているデバイス数を取得する

入力パラメータ:

- `uint16_t *deviceNum`
  - 接続されているデバイス数を格納するためのポインタ

戻り値:

- `chain_status_t`
  - 操作結果ステータスコード

getDeviceList

関数プロトタイプ:

```
bool getDeviceList(device_list_t *list);
```

機能説明:

- Chain Bus に接続されているデバイス一覧を取得する

入力パラメータ:

- `device_list_t *list`
  - デバイス一覧を格納するためのポインタ。構造体にはデバイス数と、各デバイスの詳細情報を含む配列が含まれる。

```
typedef struct {
    uint16_t count;          // Number of devices
    device_info_t *devices;  // Array of devices
} device_list_t;
```

戻り値:

- `bool`
  - `true`: 取得成功
  - `false`: 取得失敗

デバイスの詳細情報

`getDeviceList` 関数で取得できるデバイス一覧は `device_list_t` 型であり、この構造体にはデバイス数と各デバイスの詳細情報配列が含まれている。各デバイスの詳細情報は `device_info_t` 型で表され、構造体にはデバイス ID とデバイスタイプが含まれる。

```
typedef struct {
    uint16_t id;                      // Device ID
    chain_device_type_t device_type;  // Device type
} device_info_t;
```

- **デバイス ID**

1 本の Chain Bus に接続されている各デバイスには、個別の識別および制御に使用される固有のデバイス ID が割り当てられる。各デバイスの ID 値はシステムによって自動的に割り当てられ、**コントローラ側から外側へ**の接続順に**1 から順番に**増加していく:

```
Main Controller -> 1 -> 2 -> 3 -> ...
```

- **デバイスタイプ**

デバイスタイプの列挙値は次のとおり:

```
typedef enum {
    CHAIN_UNKNOWN_TYPE_CODE  = 0x0000,  // Unknown device type
    CHAIN_ENCODER_TYPE_CODE  = 0x0001,  // Chain Encoder
    CHAIN_ANGLE_TYPE_CODE    = 0x0002,  // Chain Angle
    CHAIN_KEY_TYPE_CODE      = 0x0003,  // Chain Key
    CHAIN_JOYSTICK_TYPE_CODE = 0x0004,  // Chain Joystick
    CHAIN_TOF_TYPE_CODE      = 0x0005,  // Chain ToF
    CHAIN_UART_TYPE_CODE     = 0x0006,  // Chain UART
    CHAIN_SWITCH_TYPE_CODE   = 0x0007,  // Chain Switch
    CHAIN_PEDAL_TYPE_CODE    = 0x0008,  // Chain Pedal
    CHAIN_PIR_TYPE_CODE      = 0x0009,  // Chain PIR
    CHAIN_MIC_TYPE_CODE      = 0x000A,  // Chain Microphone
    CHAIN_BUZZER_TYPE_CODE   = 0x000B,  // Chain Buzzer
} chain_device_type_t;
```

setRGBLight

関数プロトタイプ:

```
chain_status_t setRGBLight(uint16_t id, uint8_t rgbBrightness, uint8_t *operationStatus);
```

機能説明:

- デバイス上の RGB LED の明るさを設定する

入力パラメータ:

- `uint16_t id`
  - デバイス ID
- `uint8_t rgbBrightness`
  - 明るさの値（0〜100）
- `uint8_t *operationStatus`
  - 操作結果を格納するためのポインタ（0＝操作失敗、1＝操作成功）

戻り値:

- `chain_status_t`
  - 操作結果ステータスコード

setRGBValue

関数プロトタイプ:

```
chain_status_t setRGBValue(uint16_t id, uint8_t index, uint8_t num, uint8_t *rgb, uint8_t size, uint8_t
```

機能説明:

- デバイス上の RGB LED の色を設定する

入力パラメータ:

- `uint16_t id`
  - デバイス ID
- `uint8_t index`

- 制御する LED のインデックス（0 から開始）

- `uint8_t num`
  - 制御する LED の数
- `uint8_t *rgb`
  - RGB カラー配列へのポインタ。形式：`[R0, G0, B0, R1, G1, B1, ...]`
- `uint8_t size`
  - RGB カラー配列の長さ（`num * 3` と等しくなる必要がある）
- `uint8_t *operationStatus`
  - 操作結果を格納するためのポインタ（0 = 操作失敗、1 = 操作成功）

戻り値:

- `chain_status_t`
  - 操作結果ステータスコード

4. 通信メカニズム

Chain シリーズのデバイスは、複数・多種類のデバイスのホットスワップおよびデイジーチェーン通信に対応しています。これは、デバイス検出、ID 割り当て、コマンド転送などで構成される Chain Bus 通信メカニズムによって実現されています。

デバイス ID

同一の Chain Bus 上に接続された各デバイスには、個別の識別と制御のために一意のデバイス ID が割り当てられます。ID はシステムによって自動的に付与され、ホスト側から外側に向かう接続順に 1 から順に増加します。

デバイス種類

デバイス種類 `chain_device_type_t` は、`CHAIN_ENCODER_TYPE_CODE` や `CHAIN_KEY_TYPE_CODE` など、デバイスの種類を区別するために使用されます。

データ転送とレスポンス

ホスト（主控）から Chain Bus に送信されるパケットは、ホスト側のシリアルインターフェースから出力され、デバイスチェーンに入ります。チェーンの先頭にあるデバイスは、パケットに含まれるターゲット ID を確認します。`ID == 1` の場合、そのデバイス自身がターゲットであり、パケットをそのまま処理します。`ID > 1` の場合、デバイスは `ID = ID - 1` を実行し、次のデバイスへパケットを転送します。この処理は、`ID == 1` となるターゲットデバイスに到達するまで繰り返され、対象デバイスがパケットを処理します。

処理が完了すると、デバイスはレスポンスパケットを逆方向（ホスト側）へ返送します。各デバイスはレスポンスを転送する前に `ID = ID + 1` を実行し、最終的にレスポンスパケットがホストに戻ることで、ホストはどのデバイスからの応答であるかを判別できます。

末端デバイス

Chain シリーズの各デバイスは、起動時の初期化プロセスにおいて、自身をデフォルトで「末端デバイス」とみなします。チェーン内の次のデバイスから最初の完全なデータパケットを受信すると、自身の状態を更新し、「末端デバイスではない」と判断します。

Heartbeat Packet

ハートビートパケットは、Chain Bus 内で定期通信を行うための特別なパケットです。チェーン上の各デバイスは、1 秒に 1 回次のデバイスにハートビートパケットを送信します。次のデバイスは、受信したハートビートパケットをそのまま返送し、応答とします。

あるデバイスが 3 回連続でハートビートパケットを送信しても応答を受け取れない場合、その次のデバイスがオフラインと判断し、自身の状態を「末端デバイス」に更新します。

ホスト（主控）もチェーンにハートビートパケットを送信します。有効な応答が返ってきた場合は、チェーン上にデバイスが接続されていることを示し、応答がない場合はデバイスが接続されていないことを示します。

列挙 Packet

列挙パケットは、Chain Bus の接続構造を更新するための特別なパケットです。新しいデバイスがチェーンに接続されると、チェーン構造が変化したことをホストに知らせるため、180 ms 間隔で 3 回自動的に列挙要求を送信します。ホストが列挙要求を受信すると、デバイス列挙プロセスを再実行し、接続構造およびデバイス一覧を更新します。

逆に、チェーンのどこかが断線した場合、断線地点の手前にあるデバイスはハートビート応答を受け取れず、自身の状態を「非末端デバイス」から「末端デバイス」に更新し、チェーン構造が変化したことを知らせるために 3 回の列挙要求を送信します。

ホストが開始するデバイス列挙指令に対しては、チェーンの末端デバイスが列挙処理の終了を担当し、結果パケットをホストへ返送することでチェーン全体の列挙を完了します。

5. 参考リンク

- M5Chain ライブラリ - GitHub