https://automaticaddison.com/how-to-publish-wheel-encoder-tick-data-using-ros-and-arduino/

IRAM이란?

ESP32와 같은 마이크로컨트롤러에서 **IRAM(Internal RAM)**은 칩 내부에 포함된 **고속 메모리 영역**으로, 주로 CPU가 실행할 코드를 저장하거나 실행하는 데 사용됩니다. IRAM은 MCU의 성능을 최적화하는 데 중요한 역할을 합니다.

IRAM의 주요 특징

1. **속도**:
   - IRAM은 CPU와 가까운 고속 메모리로, 명령어 실행 및 데이터 처리 속도가 빠릅니다.
   - CPU는 외부 RAM보다 IRAM에 더 빠르게 접근할 수 있습니다.
2. **코드 실행**:
   - ESP32는 플래시 메모리에 저장된 프로그램 코드를 실행하지만, 실행 속도를 높이기 위해 중요한 코드나 데이터는 IRAM으로 로드됩니다.
   - 플래시 메모리는 속도가 느리므로 실행 시 IRAM에 복사해서 빠르게 실행합니다.
3. **크기 제한**:
   - IRAM의 용량은 제한적(ESP32의 경우 약 128KB)이며, 이 메모리는 CPU가 실행해야 할 **성능에 중요한 코드**와 **데이터**를 저장하는 데 사용됩니다.
4. **SRAM과의 차이**:
   - ESP32에는 IRAM뿐 아니라 **DRAM(SRAM)**도 있습니다.
   - IRAM은 실행 코드(텍스트 섹션)와 관련된 부분에 사용되며, DRAM은 일반 데이터 섹션(변수, 힙 등)에 사용됩니다.

IRAM의 활용

1. **속도 최적화**:
   - 타이밍이 중요한 코드나 ISR(Interrupt Service Routine) 같은 고성능 작업 코드를 IRAM에 저장합니다.
2. **플래시 메모리에서의 코드 실행**:
   - 플래시에서 실행되는 코드를 IRAM으로 복사해 실행하면 **Wait State**를 줄일 수 있습니다.
3. **ROS와의 연계**:
   - ESP32를 ROS 시스템의 센서 노드나 액추에이터로 사용할 때, 실시간 성능이 중요한 부분(예: 메시지 처리, 인터럽트 처리 등)을 IRAM에 배치하는 것이 좋습니다.

개발 팁

- ESP32에서는 GCC 컴파일러를 사용할 때, IRAM에 특정 코드를 할당하려면 함수 앞에 `IRAM_ATTR` 어트리뷰트를 사용합니다.

```
void IRAM_ATTR my_critical_function() {
    // 이 함수는 IRAM에서 실행됩니다.
}
```

- IRAM을 효율적으로 사용하려면 **크기 제한**을 고려하고, 정말 필요한 코드만 IRAM에 배치하세요.

IRAM 용량 확인

ESP32 개발 도구(예: Arduino IDE, ESP-IDF)를 사용하면 컴파일 후 메모리 사용량 보고서를 통해 IRAM의 사용량을 확인할 수 있습니다.

```
IRAM   : 32768 bytes (used 1648 bytes)
DRAM   : 81920 bytes (used 7632 bytes)
```

이 정보를 활용해 IRAM 최적화를 진행할 수 있습니다.

DRAM(SRAM)?

DRAM(SRAM)은 ESP32와 같은 마이크로컨트롤러에서 CPU가 사용할 수 있는 **데이터 저장용 메모리**로, 프로그램이 실행되는 동안 동적으로 생성되는 데이터를 저장하는 데 사용됩니다. ESP32에서 DRAM은 사실상 **SRAM(Static RAM)**이며, DRAM이라는 용어는 내부 메모리 구조의 구분을 위해 사용됩니다.

DRAM(SRAM)의 주요 특징

1. **역할**:
   - **DRAM**은 프로그램 실행 중 생성되는 데이터(변수, 스택, 힙 등)를 저장하는 데 사용됩니다.
   - 반면, **IRAM**은 실행할 코드나 성능에 중요한 데이터를 저장합니다.

2. **속도**:
    - DRAM은 플래시 메모리보다 훨씬 빠릅니다.
    - IRAM만큼 빠르지는 않지만, 일반적인 데이터 처리에 적합한 속도를 제공합니다.
3. **정적 메모리(SRAM)**:
    - ESP32의 DRAM은 실제로는 **SRAM**으로 구현되어 있으며, DRAM처럼 재충전이 필요하지 않아서 더 간단하고 빠릅니다.
4. **크기**:
    - ESP32의 DRAM 크기는 약 **320KB**로 제한적입니다(ESP32 모델에 따라 다름).
    - 이 메모리는 실행 중인 애플리케이션에서 사용하는 스택, 힙, 전역 및 정적 변수를 저장합니다.
5. **SRAM의 구조**:
    - ESP32는 내부 메모리를 **IRAM**과 **DRAM**으로 구분하지만, 하드웨어적으로는 동일한 SRAM 블록을 공유하며, 메모리 매핑에 따라 구분됩니다.

DRAM의 활용

1. **데이터 저장**:
    - 프로그램 실행 중 필요한 모든 동적 데이터(예: 변수, 버퍼, 힙 등)를 DRAM에 저장합니다.
2. **스택과 힙**:
    - 스택(Stack): 함수 호출 시 사용되는 메모리.
    - 힙(Heap): 동적으로 할당되는 메모리(예: `malloc`이나 `new`를 통해).
3. **ROS와의 연계**:
    - ESP32를 ROS 센서 노드로 사용할 때, 센서 데이터 버퍼링, 메시지 큐 등의 데이터를 DRAM에서 관리합니다.

DRAM 관련 메모리 관리

1. **정적 변수**:
    - `int a = 10;` 같은 전역 변수나 정적 변수는 DRAM에 할당됩니다.
2. **동적 메모리 할당**:
    - `malloc` 또는 `new`를 통해 동적으로 생성된 객체는 DRAM(Heap)에 저장됩니다.
3. **스택 메모리**:
    - 함수 내부에서 선언된 지역 변수는 스택 메모리에 할당되며, 함수 실행이 끝나면 자동으로 해제됩니다.

4. **SPI RAM**:
    - ESP32는 내부 DRAM이 부족할 경우 외부 **SPI RAM**(PSRAM)을 사용하여 추가 메모리를 제공할 수 있습니다.
    - 예: `CONFIG_SPIRAM_SUPPORT`를 활성화하면 힙 크기가 증가합니다.

DRAM 최적화 팁

1. **동적 메모리 사용 최소화**:
    - ESP32의 DRAM은 제한적이므로, 동적 메모리(`malloc`) 사용을 최소화하고 정적 변수 사용을 권장합니다.
2. **메모리 사용량 확인**:
    - ESP-IDF를 사용하면 런타임 중 DRAM 사용량을 확인할 수 있습니다.

```
#include "esp_system.h"

void print_free_memory() {
    printf("Free heap size: %d bytes\n", esp_get_free_heap_size());
}
```

3. **메모리 할당 실패 처리**:
    - 메모리 부족으로 인한 프로그램 오류를 방지하기 위해 항상 할당 실패를 처리하세요.

```
int* buffer = malloc(1024);
if (!buffer) {
    printf("Memory allocation failed!\n");
}
```

DRAM 용량 확인

컴파일 후 메모리 사용량을 확인하면 DRAM의 사용 상태를 확인할 수 있습니다. 예를 들어:

```
IRAM   : 32768 bytes (used 1648 bytes)
DRAM   : 81920 bytes (used 7632 bytes)
```

이 정보를 바탕으로 메모리를 효율적으로 관리하면, DRAM 부족으로 인한 문제를 방지할 수 있습니다.

Flash 메모리?

Flash 메모리는 ESP32와 같은 마이크로컨트롤러에서 프로그램 코드와 데이터를 **비휘발성**으로 저장하는 데 사용되는 메모리입니다. 전원이 꺼져도 데이터를 유지할 수 있는 특징을 가지고 있으며, 마이크로컨트롤러의 핵심 저장장치로 사용됩니다.

Flash 메모리의 주요 특징

1. **비휘발성**:
   - 전원이 꺼져도 데이터를 유지할 수 있어 프로그램 코드(펌웨어)와 영구 데이터를 저장하는 데 적합합니다.
2. **저장 용량**:
   - ESP32에는 일반적으로 4MB ~ 16MB의 Flash 메모리가 내장되어 있습니다(모델에 따라 다름).
3. **저장 역할**:
   - **프로그램 코드**: 컴파일된 바이너리(펌웨어)는 Flash 메모리에 저장됩니다.
   - **데이터 저장소**: 파일 시스템(SPIFFS, LittleFS)이나 NVS(Non-Volatile Storage)를 통해 데이터를 저장합니다.
4. **속도**:
   - Flash 메모리는 DRAM이나 IRAM보다 속도가 느립니다.
   - 실행 속도를 높이기 위해 필요한 코드나 데이터를 IRAM으로 복사해서 실행합니다.
5. **쓰기/지우기 제약**:
   - Flash 메모리는 블록 단위로 데이터를 지우고, 페이지 단위로 데이터를 쓸 수 있습니다.
   - 쓰기와 지우기가 제한된 횟수(일반적으로 10,000 ~ 100,000회)가 있으므로, 자주 쓰기가 필요한 경우 주의가 필요합니다.

Flash 메모리의 주요 사용 사례

1. **펌웨어 저장**:
   - 컴파일된 애플리케이션 코드가 Flash에 저장되며, 부팅 시 CPU가 Flash에서 프로그램을 읽어 실행합니다.
2. **SPIFFS/LittleFS**:

- Flash 메모리의 일부를 파일 시스템으로 사용하여 설정 파일, 로그, 이미지 등 데이터를 저장할 수 있습니다.
- 예: `data.txt` 파일을 저장하거나 읽을 때 사용.

3. **NVS(Non-Volatile Storage)**:
   - 비휘발성 키-값 저장소로, 소량의 데이터를 저장할 때 사용합니다(예: Wi-Fi SSID 및 비밀번호).
4. **OTA(Over-The-Air) 업데이트**:
   - Flash 메모리를 이용해 새로운 펌웨어를 다운로드하고, 시스템을 업데이트할 수 있습니다.
5. **캠핑 데이터**:
   - 센서 데이터, 로그, 상태 정보를 저장하여 시스템 재부팅 후에도 데이터를 유지합니다.

Flash 메모리의 구조 (ESP32 기준)

ESP32에서 Flash 메모리는 여러 섹션으로 나뉘어 사용됩니다:

1. **부트로더(bootloader)**:
   - ESP32가 부팅할 때 실행되는 초기 코드.
2. **파티션 테이블(partition table)**:
   - Flash 메모리를 구분하는 정보가 저장됩니다(예: 펌웨어, OTA, NVS, SPIFFS).
3. **애플리케이션 코드**:
   - 실제 프로그램 코드가 여기에 저장됩니다.
4. **OTA 파티션**:
   - 펌웨어 업데이트를 지원하는 경우, 업데이트된 코드를 저장하는 파티션.
5. **파일 시스템**:
   - SPIFFS 또는 LittleFS를 사용하여 데이터를 저장.
6. **NVS**:
   - 영구적으로 데이터를 저장하는 키-값 저장소.

Flash 메모리 사용 예제

1. **SPIFFS 파일 시스템 사용**:
   - ESP32의 Flash를 파일 저장소로 사용하는 코드입니다.

```
#include <SPIFFS.h>

void setup() {
    Serial.begin(115200);

    // SPIFFS 초기화
    if (!SPIFFS.begin(true)) {
        Serial.println("SPIFFS Mount Failed");
        return;
    }

    // 파일 쓰기
    File file = SPIFFS.open("/test.txt", FILE_WRITE);
    if (file) {
        file.println("Hello Flash!");
        file.close();
    }

    // 파일 읽기
    file = SPIFFS.open("/test.txt");
    if (file) {
        while (file.available()) {
            Serial.write(file.read());
        }
        file.close();
    }
}

void loop() {}
```

2. **NVS 사용**:
    - ESP-IDF에서 NVS를 사용해 데이터를 저장/읽는 코드입니다.

```
#include "nvs_flash.h"
#include "nvs.h"

void app_main() {
    nvs_flash_init();

    // NVS 핸들 열기
    nvs_handle_t nvs_handle;
    nvs_open("storage", NVS_READWRITE, &nvs_handle);

    // 데이터 쓰기
    int32_t value = 42;
```

```
    nvs_set_i32(nvs_handle, "key", value);
    nvs_commit(nvs_handle);

    // 데이터 읽기
    int32_t read_value = 0;
    nvs_get_i32(nvs_handle, "key", &read_value);
    printf("Stored Value: %d\n", read_value);

    nvs_close(nvs_handle);
}
```

Flash 메모리와 다른 메모리와의 비교

특성	Flash	IRAM	DRAM(SRAM)
속도	느림	매우 빠름	빠름
비휘발성	예	아니오	아니오
저장 용량	수 MB	수십 KB	수백 KB
주요 용도	펌웨어, 데이터 저장	실행 코드 저장	동적 데이터 저장

Flash 메모리는 ESP32의 중요한 구성 요소로, 코드를 저장하고 비휘발성 데이터를 관리하며, 효율적인 관리가 필수적입니다. 프로젝트에 따라 Flash 메모리를 효율적으로 사용하는 방법을 선택하세요.

ESP32의 또 다른 메모리

ESP32에는 Flash, IRAM, DRAM(SRAM) 외에도 몇 가지 추가 메모리 유형이 존재합니다. 이 메모리들은 특정 목적에 맞게 설계되었으며, 효율적인 메모리 관리를 통해 ESP32의 성능을 극대화할 수 있습니다. 아래는 ESP32에서 사용할 수 있는 주요 메모리 종류와 그 특징들입니다.

---

1. RTC 메모리 (RTC RAM)

RTC 메모리는 ESP32의 저전력 모드에서 데이터를 유지할 수 있는 특수한 메모리입니다.

- **용도**:
  - **저전력 모드**(예: Deep Sleep)에서 데이터를 유지하기 위해 사용됩니다.
  - 전원이 꺼져도 RTC가 활성 상태라면 데이터가 보존됩니다.
- **특징**:
  - 용량: 약 **8KB**.
  - RTC가 활성화된 상태에서만 데이터를 유지합니다.
  - 주로 센서 데이터나 상태 정보를 저장하여 Deep Sleep 모드에서 복구 시 활용됩니다.
- **사용 방법**:
  - ESP-IDF에서 RTC 메모리를 사용하려면 `rtc_memory` API를 활용합니다.

```
RTC_DATA_ATTR int count = 0; // RTC RAM에 저장
void app_main() {
    count++;
    printf("Count: %d\n", count); // Deep Sleep 후에도 count 값 유지
    esp_deep_sleep(1000000); // 1초 후 Deep Sleep 모드로 진입
}
```

---

2. PSRAM (Pseudo SRAM)

PSRAM은 ESP32 외부에 추가될 수 있는 메모리로, 메모리 용량을 확장하고 대용량 데이터를 처리하는 데 사용됩니다.

- **용도**:
  - 대규모 데이터 처리(이미지 버퍼, 네트워크 데이터 버퍼 등).
  - 머신러닝 모델, 그래픽 데이터 등 고용량 작업에 적합.
- **특징**:
  - 용량: **4MB ~ 16MB**(ESP32 모듈 및 외부 칩에 따라 다름).
  - ESP32-WROVER, ESP32-S3 같은 모델에서 기본적으로 PSRAM이 포함되어 있음.
  - DRAM처럼 동작하지만 속도는 IRAM이나 내부 DRAM보다 느림.
- **활성화**:
  - ESP-IDF 설정에서 **PSRAM 지원**을 활성화해야 합니다.
  - `CONFIG_SPIRAM_SUPPORT` 옵션을 설정하면 사용할 수 있습니다.
- **사용 예**:

```c
#include <stdlib.h>
void app_main() {
    int* large_array = (int*) heap_caps_malloc(10000 * sizeof(int),
MALLOC_CAP_SPIRAM);
    if (large_array == NULL) {
        printf("Failed to allocate PSRAM memory\n");
    } else {
        printf("PSRAM allocation successful\n");
        free(large_array);
    }
}
```

---

3. ROM (Read-Only Memory)

ESP32에는 칩에 고정된 **ROM**이 포함되어 있으며, 제조 시 미리 저장된 데이터를 포함합니다.

- **용도**:
  - 부트로더 코드 저장.
  - 기본 드라이버와 시스템 함수 제공.
- **특징**:
  - 변경 불가능.
  - 사용자가 직접 접근하거나 수정할 수 없음.
- **내장된 기능**:
  - UART 다운로드 부트로더.
  - 기본적인 플래시 메모리 읽기/쓰기 루틴.

---

4. Cache 메모리

ESP32는 Flash와 CPU 간의 속도 차이를 줄이기 위해 **캐시 메모리**를 사용합니다.

- **용도**:
  - Flash에서 실행되는 코드와 데이터를 캐싱하여 실행 속도를 높임.
  - IRAM으로 복사되지 않은 코드나 데이터를 처리.
- **특징**:
  - Flash 메모리 접근 지연 시간을 줄이기 위해 필수적.
  - CPU가 반복적으로 사용하는 데이터를 저장하여 성능 향상.

메모리 구조 요약

메모리 유형	역할	용량	휘발성 여부	특별한 사용 목적
Flash	펌웨어 및 영구 데이터 저장	4MB ~ 16MB	비휘발성	코드 저장, NVS, SPIFFS, OTA
IRAM	실행 코드 저장 및 빠른 접근	약 128KB	휘발성	성능 중요한 코드 실행
DRAM(SRAM)	동적 데이터 저장	약 320KB	휘발성	힙, 스택, 변수 저장
RTC RAM	저전력 모드에서 데이터 유지	약 8KB	조건부 휘발성	Deep Sleep 데이터 유지
PSRAM	외부 확장 메모리 (대용량 데이터 처리)	4MB ~ 16MB	휘발성	이미지 처리, 머신러닝, 고용량 데이터 처리
ROM	고정된 부트로더 및 시스템 함수 저장	제조 시 설정됨	비휘발성	기본 시스템 기능 제공
Cache	Flash와 CPU 간의 속도 차이 보정	내부 설정됨	휘발성	Flash 접근 성능 최적화

프로젝트에 따라 메모리 선택 가이드

1. **성능이 중요한 코드** → **IRAM**에 배치.
2. **동적 데이터** → **DRAM(SRAM)** 사용.
3. **대규모 데이터 처리** → **PSRAM** 사용.
4. **저전력 모드 데이터 유지** → **RTC RAM** 사용.
5. **펌웨어, 영구 데이터** → **Flash** 사용.

ESP32의 다양한 메모리 유형을 이해하고 활용하면, 메모리 부족 문제를 해결하고 애플리케이션의 성능을 최적화할 수 있습니다!

---

mcu.launch.py?

실행 시 예상 동작

1. **micro-ROS 에이전트가 실행**:
   - `micro_ros_agent`는 ROS 2 네트워크와 ESP32 MCU(예: NodeMCU) 간의 통신을 중계합니다.
   - ESP32에서 실행 중인 micro-ROS 클라이언트와 연결을 시도합니다.
2. **ESP32와 ROS 2 간 데이터 전송**:
   - ESP32에서 ROS 2 네트워크로 메시지를 전달하거나, ROS 2 네트워크에서 ESP32로 명령을 보낼 수 있는 상태가 됩니다.
3. **로그 출력**:
   - 에이전트의 실행 상태 및 통신 정보를 화면에 출력합니다. 예를 들어, 연결 성공/실패 메시지가 나타날 수 있습니다.