

README

1. 개요

본 프로젝트는 예측 가능성과 안전성에 초점을 맞춘 ST MCU용 비선점형(Non-preemptive) 주기 기반 스케줄러입니다.

단순히 태스크를 실행하는 것을 넘어, 실시간 시스템에서 발생하는 자연과 예외 상황을 명확히 진단하고 대응하는 것을 목표로 합니다.

특징

이 스케줄러는 아래의 특징을 가집니다.

Deterministic Multi-Layer Diagnosis

단순히 태스크를 실행하는 것을 넘어 시스템의 건강 상태를 정밀하게 추적합니다.

- 이중 오버런 관리

각 태스크의 오버런 발생을 '현재 사이클'과 '전체 가동 시간' 단위로 개별 집계하여, 일시적인 부하와 구조적인 병목 구간을 명확히 구분합니다.

Fail-Safe Oriented Protection

시스템이 통제 불능 상태에 빠지는 것을 방지하는 강력한 보호 로직을 갖추고 있습니다. (선택사항)

- 임계치 기반 리셋

누적 오버런이 설정된 임계치(`THRESHOLD_CNT`)를 초과하면 정의되지 않은 동작을 방지하기 위해 시스템을 안전하게 초기화(Reset) 프로세스로 유도합니다.

- 위치독 동기화

모든 태스크가 정상 실행된 시점에만 위치독을 갱신하여, 특정 태스크의 데드락이나 논리 오류 발생 시 확실한 하드웨어 리셋을 보장합니다.

Hook System

디버거를 연결하여 CPU를 멈추지 않고도, 실제 가동 중인 시스템의 내부 상태를 실시간으로 추적할 수 있습니다.

- 타이밍 보존 추적

유저 흑(Hook) 함수를 통해 매 사이클의 상태를 UART 등으로 출력하여, 디버거 브레이크 포인트로 인한 시스템 멈춤 없이 흐름을 파악합니다.

- 생애 주기 통제

초기화, 에러 발생, 사이클 전환 등 핵심 이벤트 시점에 사용자 로직을 삽입하여 시스템 전체를 제어할 수 있습니다.

Non-Invasive Weak Symbol

라이브러리 내부 코드를 수정하지 않고도 사용자 환경에 최적화된 기능을 확장할 수 있습니다.

- 독립적 코드 관리

`_weak` 키워드를 활용한 콜백 구조를 통해 본체 코드는 유지한 채 사용자 정의 핸들러(로그, 비상 정지 등)를 자유롭게 연결합니다.

- 유연한 예외 처리

사용자가 하드웨어 환경에 맞춰 LED 인디케이터나 긴급 데이터 백업 로직을 쉽게 구현할 수 있도록 인터페이스를 제공합니다.

Flexible Macro-Based Customization

사용자의 시스템 환경과 리소스 상황에 맞춰 스케줄러의 기능을 자유롭게 조절할 수 있습니다.

- 기능 제어

`config` 헤더의 `#define` 설정만으로 오버런 탐지, 위치독 피딩, 사이클 측정 등의 핵심 기능을 개별적으로 켜거나 끌 수 있습니다.

- **오버헤드 관리**

불필요한 기능을 비활성화하여 MCU의 오버헤드를 최소화하고 시스템에 꼭 필요한 로직만 남길 수 있습니다.

2. 파일 구성

라이브러리는 총 4개의 파일로 구성됩니다.

tasksch.c / tasksch.h

스케줄러의 핵심 엔진입니다. 태스크 실행 로직, 오버런 탐지, 메이저 사이클 관리 및 시간 관리 로직이 포함되어 있으며, 라이브러리 본체이므로 가급적 수정을 권장하지 않습니다.

tasksch_config.c / tasksch_config.h

사용자 정의 설정 파일입니다. 실행할 태스크의 등록, 하드웨어 의존적인 로직(Watchdog, GPIO, ISR 제어 등) 및 스케줄러 옵션(매크로)을 설정합니다.

3. 지원 환경

스케줄러의 안정적인 동작을 위해 권장되는 시스템 환경입니다.

Name	Description	Required
MCU	ST MCU	O
system CLK	72MHz	O
TIM	1ms period Interrupt	O
IWDG	For initialization when out of control of the system	X
GPIO	For measuring the cycle of Major Cycle	X

4. Quick Start

본 스케줄러를 프로젝트에 빠르게 적용하는 방법입니다.

a. 스케줄러 옵션 설정

`tasksch_config.h` 에서 시스템에 필요한 옵션들을 설정하세요

```
#define TASKSCH_WATCHDOG_ENABLE          (1)
#define TASKSCH_WATCHDOG_DISABLE          (0)
#define TASKSCH_TASK_WATCHDOG           (TASKSCH_WATCHDOG_ENABLE)      // NOTE : USER DEFINITION

#if (TASKSCH_TASK_WATCHDOG == TASKSCH_WATCHDOG_ENABLE)
// NOTE : USER DEFINE → Watchdog clock frequency, prescaler, timeout count
// INFO : LSI frequency can have an error of up to ±20%, so take the count with a margin of error.
#define TASKSCH_WATCHDOG_CLK_FREQ_HZ   (40000U)    // Consider LSI at 56khz
#define TASKSCH_WATCHDOG_PRESCALER    (256U)
#define TASKSCH_WATCHDOG_PRSCLAER_BITS (IWDG_PRESCALER_256)
#define TASKSCH_WATCHDOG_TIMEOUT_CNT  (2000U)

// Derived Macros → Do not modify
#define TASKSCH_WATCHDOG_1COUNT_MS    (TASKSCH_WATCHDOG_PRESCALER / TASKSCH_WATCH
```

```
DOG_CLK_FREQ_HZ) // 256 / 40000 = 6.4ms
#define TASKSCH_WATCHDOG_TIMEOUT_MS (TASKSCH_WATCHDOG_TIMEOUT_CNT * TASKSCH_WAT
CHDOG_1COUNT_MS) // 469 * 6.4ms = 3s
#endif
```

b. 태스크 함수 정의 및 등록

`tasksch_config.c` 파일에서 실행하고자 하는 유저 함수를 정의하고 `tasksch_init_RegTaskObj`에서 등록합니다.

```
// 1. Define Task Function
void UserTask_1ms(void) { /* logic */ }
void UserTask_10ms(void) { /* logic */ }

// 2. Registration
void tasksch_init_RegTaskObj(void) {
    vUserRegiTaskObj[0].regiTaskFunc_ptr = UserTask_1ms;
    vUserRegiTaskObj[0].regiTaskPeriod_ms = 1;
    vUserRegiTaskObj[0].regiTaskOffset_ms = 0;

    vUserRegiTaskObj[1].regiTaskFunc_ptr = UserTask_10ms;
    vUserRegiTaskObj[1].regiTaskPeriod_ms = 10;
    vUserRegiTaskObj[1].regiTaskOffset_ms = 5;
}
```

c. 타이머 인터럽트 연결

MCU의 1ms 타이머 인터럽트 서비스 루틴(ISR) 내에 스케줄러의 타임 매니저를 배치합니다.

```
void TIM1_UP_IRQHandler(void)
{
    tasksch_timeManager();
}
```

d. 스케줄러 시작

```
int main(void) {
    ...

    // 1. Initialization
    tasksch_init();
    // 2. executing Task
    tasksch_execTask()

    while (1) {
        ...
    }
}
```

e. 상태추적

시스템 상태를 추적하려면 `tasksch_userMajorCycleHook`을 구현하여 활용하세요.

```
// implementation in tasksch_config.c
void tasksch_userMajorCycleHook(void) {
    // Send current overrun count to UART every major cycle
    printf("Total Overrun: %d\n", tasksch_getOverRunCount());
}
```

5. 에러 처리

본 스케줄러는 시스템의 안정성을 최우선으로 하며, 에러 발생 시 상태 코드(`ErrorCode`)를 저장하고 안전한 중단(`Assert`)을 유도합니다.

초기화

시스템 가동 시 발생할 수 있는 설정 오류를 감지합니다. 에러 발생 시 `tasksch_init()` 과정에서 `TASKSCH_ASSERT_FUNC()` 가 호출되어 시스템 가동을 사전에 차단합니다.

Error Code	Description	Cause & Solution
<code>TASKSCH_INIT_ERR_TASK_CONFIG_FAIL</code>	태스크 구성 실패	TASKSCH_NUMBER 범위를 초과하거나 잘못된 파라미터 전달 시 발생
<code>TASKSCH_INIT_ERR_TASK_INFO_INVALID</code>	태스크 유효성 검사 실패	주기(Period)가 0이거나 함수 포인터가 NULL인 경우 발생
<code>TASKSCH_INIT_ERR_WATCHDOG_INIT_FAIL</code>	워치독 초기화 실패	하드웨어(IWDG) 초기화 중 HAL 에러 발생 시 발생
<code>TASKSCH_INIT_ERR_WATCHDOG_START_FAIL</code>	워치독 시작 실패	워치독 엔진 가동 실패 시 발생

런타임

시스템 운영 중에 발생하는 실시간 문제를 감지하여 보호 로직을 가동합니다.

Error Code	Description	Reaction
<code>TASKSCH_RUN_ERR_INIT_NOT_DONE</code>	미초기화 실행	초기화(init) 없이 execTask 호출 시 즉시 Assert 처리
<code>TASKSCH_RUN_ERR_INIT AGAIN</code>	중복 초기화	이미 가동 중인 스케줄러를 재초기화 시도 시 차단
<code>TASKSCH_RUN_ERR_WATCHDOG_FAIL</code>	워치독 피딩 실패	하드웨어 워치독 갱신 실패 시 시스템 리셋 유도
<code>TASKSCH_RUN_ERR_OVERRUN_CNT_EXCEEDED</code>	오버런 임계치 초과	시스템 부하 누적으로 THRESHOLD_CNT 초과 시 유저 훅(Hook) 호출 후 안전 모드 진입

에러 대응 방법

- Assert Logic

치명적인 에러 발생 시 `TASKSCH_ASSERT_FUNC()` 가 호출됩니다. 이는 `config` 설정에 따라 무한 루프에 빠지거나, 워치독 피딩을 중단하여 하드웨어 강제 리셋을 유발하도록 설계되었습니다.

- User Hook 활용

런타임 에러 발생 시 `tasksch_userOverRun_thrshldExceedHook` 등 유저 정의 함수를 통해 시스템이 멈추기 전 마지막 상태를 저장(EEPROM 등)하거나 로그를 남길 수 있습니다.

6. 트러블 슈팅

계속되는 초기화/리셋 문제

스케줄러가 자꾸만 리셋되어 시스템을 사용할 수 없다면 다음 사항들을 순서대로 확인해보세요.

- 타이머 ISR 확인:

`tasksch_timeManager()` 가 정확히 1ms마다 호출되는지 중단점을 걸어 확인하십시오.

- 클럭 주파수:

MCU 클럭이 너무 낮으면 태스크 처리가 밀려 오버런이 쌓이거나, 그 전에 워치독이 먼저 작동할 수 있습니다.

- **워치독 데드라인:**

메이저 사이클(=가장 긴 태스크 주기)이 워치독 타임아웃보다 훨씬 짧아야 합니다.

독점 태스크 식별

오버런은 보통 실행 시간이 긴 태스크에 의해 **밀려난** 후순위 태스크에서 발생합니다.

만약 원인을 찾기 어렵다면, **개별 오버런 횟수**(`indiOverRun_totalCnt`)가 가장 적으면서 주기가 짧은 태스크를 의심하십시오.

해당 태스크가 CPU를 점유하여 전체 스케줄링을 방해하고 있을 가능성이 큽니다.

훅 함수 문자열 문제

만약 UART를 결정적인 이벤트에서 알림용으로 사용하는데, 문자열이 깨지는 문제가 발생하는 경우가 발생하였을 때 대처 할 수 있는 방법입니다.



```
VT COM10 - Tera Term VT
File Edit Setup Control Window Help
HBeding fail!
Hello Host!
HBeding fail!
Hello Host!
Heding fail!
Hello Host!
HBeding fail!
Hello Host!
HBeding fail!
Hello Host!
Heding fail!
Hello Host!
HBeding fail!
Hello Host!
HBeding fail!
Hello Host!
Heding fail!
Hello Host!
HBeding fail!
Hello Host!
HBeding fail!
```

위의 예시는 초기화 훅 함수에서 "Hello Host!" 를 출력시키고 일부러 워치독 피딩을 실패시켜서 훅함수가 제대로 호출되는지를 확인하는 도중 발생한 문제입니다.

- **현상:**

- 초기화 훅 함수(`tasksch_userInitCmpltHook`) 등에서 "feeding fail!"와 같은 메시지를 출력하도록 설정했으나, 원하는 문자열의 초반부에 전혀 다른 문자열이 섞이는 현상
- 원하는 문자열이 아예 출력되지 않거나 일부만 출력되고 끊기는 현상.

- **원인 분석:**

- Major cycle의 동작 순서에서, Major사이클 hook 함수가 실행되고 곧바로 워치독 feedingFail hook 함수가 실행되는 구조를 갖습니다.

Major cycle Hook 함수 외 DMA를 사용할 경우 HW 단에서 Major사이클 hook 함수에서 설정한 버퍼를 중간에 지우고 새 문자열을 채우기 때문에 이와 같은 현상이 발생합니다.

- PC와 MCU간의 보드레이트 설정 불일치로 인해 알아볼 수 없는 문자열을 수신받을 수도 있습니다.
- Hook 함수에서 DMA를 사용하는 경우, 곧바로 Assert에 빠지게 되면 문자열이 전부 수신되기 전에 시스템이 종료될 수도 있습니다.

- **해결책:**

- **Blocking 방식 사용:** 디버깅용 로그를 찍을 때는 인터럽트 방식보다는 전송이 완료될 때까지 기다리는 **Polling(Blocking)** 방식의 함수(예: `HAL_UART_Transmit` 의 Timeout 활용)를 사용하세요.

- 메인 사이클 혹은 함수에서 사용하는 문자열 버퍼와 실행되고 종료단계에 돌입하는 혹은 함수가 사용하는 문자열 버퍼를 분리 하십시오
- **Baud rate 확인:** 고속 통신 시 오차가 발생하기 쉬우므로, 115200bps 등 표준 속도를 사용하고 오실로스코프로 한 비트의 폭(Bit width)이 정확한지 확인하세요.

- 예시

```
void uart_send_staticStr(const char* str) // \r\n plz..
{
    memset((void*)uart_normal_bufferTx, 0, UART_BUFFER_SIZE);
    strncpy(uart_normal_bufferTx, str, UART_BUFFER_SIZE - 1);
    uart_send_DMA(uart_normal_bufferTx);
}

void uart_send_beforeOFF(const char* str)
{
    HAL_UART_AbortTransmit(UART_HANDLER);

    memset((void*) uart_event_bufferTx, 0, UART_BUFFER_SIZE);
    HAL_UART_Transmit(UART_HANDLER,(const uint8_t*)"\\r\\n",2,10);
    strncpy(uart_event_bufferTx, str, UART_BUFFER_SIZE - 1);
    HAL_UART_Transmit(UART_HANDLER,(const uint8_t*)uart_event_bufferTx,UART_BUFFER_SIZE,500);
}
```