README_EN

1. Overview

This project is a **non-preemptive, period-based scheduler for ST microcontrollers**, designed with a primary focus on **predictability and system safety**.

Rather than simply executing tasks at fixed intervals, the scheduler is intended to **detect, diagnose, and respond to delays and exceptional conditions** that commonly arise in real-time embedded systems.

The core philosophy of this scheduler is not to eliminate all timing violations, but to **make such violations observable, traceable, and recoverable**.

Features

The scheduler provides the following core capabilities.

Deterministic Multi-Layer Diagnosis

Beyond basic task execution, the scheduler continuously tracks the overall health of the system.

- **Dual-Level Overrun Tracking**

  Overrun events are accumulated per task at two levels:

  - within the current major cycle, and
  - across the total system runtime.

  This separation allows transient load spikes to be clearly distinguished from persistent structural bottlenecks.

Fail-Safe Oriented Protection

The scheduler includes optional protection mechanisms designed to prevent the system from entering an uncontrollable state.

- **Threshold-Based Reset**

  When the accumulated overrun count exceeds the configured threshold (`THRESHOLD_CNT`), the system deliberately transitions into a controlled reset sequence to avoid undefined behavior.

- **Watchdog Synchronization**

  The watchdog is refreshed **only after all registered tasks have executed successfully**.

  This ensures a reliable hardware reset in the presence of task deadlocks, infinite loops, or logical faults.

Hook System

The internal state of a running system can be observed in real time **without halting execution using a debugger**.

- **Timing-Preserving Tracing**

  User-defined hook functions allow system state information to be emitted (e.g., via UART) at well-defined execution points, enabling runtime observation without disturbing system timing through breakpoints.

- **Lifecycle-Level Control**

  User logic can be injected at key lifecycle events—such as initialization completion, error detection, or major cycle transitions—allowing system-level behavior to be customized without modifying the scheduler core.

Non-Invasive Weak-Symbol Extension

Scheduler behavior can be extended without modifying internal library code.

- **Isolated User Code**

  A callback-based design using `__weak` symbols allows user-defined handlers (logging, emergency stop routines, status indicators, etc.) to be attached while keeping the core engine intact.

- **Flexible Exception Handling**

  The provided interfaces make it straightforward to implement hardware-specific responses, such as LED indicators or emergency data backup routines, tailored to the target system.

Flexible Macro-Based Customization

Scheduler functionality can be selectively enabled or disabled to match system constraints and available resources.

- **Feature Selection**

  Core features—such as overrun detection, watchdog feeding, and cycle measurement—can be individually controlled using `#define` options in the configuration header.

- **Overhead Control**

  Unused features can be disabled to minimize runtime overhead, leaving only the logic required for the target system.

---

2. File Structure

The scheduler library consists of four files.

`tasksch.c` / `tasksch.h`

These files contain the **core scheduler engine**, including task execution logic, overrun detection, major cycle management, and time management.

As these files form the core of the scheduler, **modification is strongly discouraged**.

`tasksch_config.c` / `tasksch_config.h`

These are **user configuration files**.

They are used to:

- register application tasks,
- configure hardware-dependent behavior (watchdog, GPIO, ISR control), and
- enable or disable scheduler features via macros.

---

3. Supported Environment

The following system configuration is recommended for stable operation.

Name	Description	Required
MCU	ST microcontroller	Yes
System Clock	72 MHz	Yes
Timer	1 ms periodic interrupt	Yes
IWDG	System recovery when out of control	Optional
GPIO	Major cycle timing measurement	Optional

---

4. Quick Start

This section describes the minimum steps required to integrate the scheduler into a project.

a. Scheduler Option Configuration

Configure the required options in `tasksch_config.h`.

```
#define TASKSCH_WATCHDOG_ENABLE          (1)
#define TASKSCH_WATCHDOG_DISABLE         (0)
#define TASKSCH_TASK_WATCHDOG            (TASKSCH_WATCHDOG_ENABLE)// User-defined

#if (TASKSCH_TASK_WATCHDOG == TASKSCH_WATCHDOG_ENABLE)
// User-defined watchdog parameters
// NOTE: LSI frequency may vary by up to ±20%; allow sufficient margin.
#define TASKSCH_WATCHDOG_CLK_FREQ_HZ    (40000U)
#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_WATCHDO
G_CLK_FREQ_HZ)
#define TASKSCH_WATCHDOG_TIMEOUT_MS     (TASKSCH_WATCHDOG_TIMEOUT_CNT * TASKSCH_WATC
HDOG_1COUNT_MS)
#endif
```

b. Task Definition and Registration

Define application tasks in `tasksch_config.c` and register them using `tasksch_init_RegiTaskObj`.

```
voidUserTask_1ms(void)  {/* task logic */ }
voidUserTask_10ms(void) {/* task logic */ }

voidtasksch_init_RegiTaskObj(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. Timer Interrupt Integration

Call the scheduler time manager from a 1 ms timer ISR.

```
voidTIM1_UP_IRQHandler(void)
{
   tasksch_timeManager();
}
```

d. Starting the Scheduler

```
intmain(void)
{
   tasksch_init();

while (1)
   {
      tasksch_execTask();
   }
}
```

e. System State Monitoring

System-level status can be monitored by implementing `tasksch_userMajorCycleHook`.

```
voidtasksch_userMajorCycleHook(void)
{
printf("Total Overrun Count: %d\n", tasksch_getOverRunCount());
}
```

5. Error Handling

The scheduler prioritizes system safety.

When an error is detected, an internal error code is recorded and a controlled stop (assert) is triggered.

Initialization Errors

Configuration errors detected during system startup.

If an error occurs during `tasksch_init()`, `TASKSCH_ASSERT_FUNC()` is invoked to prevent the system from entering an undefined state.

Error Code	Description	Cause
`TASKSCH_INIT_ERR_TASK_CONFIG_FAIL`	Task configuration failure	Invalid parameters or `TASKSCH_NUMBER` exceeded
`TASKSCH_INIT_ERR_TASK_INFO_INVALID`	Task validation failure	Zero period or NULL function pointer
`TASKSCH_INIT_ERR_WATCHDOG_INIT_FAIL`	Watchdog initialization failure	HAL error during IWDG initialization
`TASKSCH_INIT_ERR_WATCHDOG_START_FAIL`	Watchdog start failure	Watchdog failed to start

Runtime Errors

Errors detected during normal operation.

Error Code	Description	Reaction
`TASKSCH_RUN_ERR_INIT_NOT_DONE`	Execution before initialization	Immediate assert
`TASKSCH_RUN_ERR_INIT_AGAIN`	Re-initialization attempt	Blocked
`TASKSCH_RUN_ERR_WATCHDOG_FAIL`	Watchdog feeding failure	Hardware reset
`TASKSCH_RUN_ERR_OVERRUN_CNT_EXCEEDED`	Overrun threshold exceeded	User hook invoked, then safe shutdown

Error Response Strategy

- **Assert Logic**

  On critical errors, `TASKSCH_ASSERT_FUNC()` is invoked.

Depending on configuration, the system either enters an infinite loop or intentionally stops watchdog feeding to force a hardware reset.

- **User Hooks**

  Before shutdown, user-defined hooks (e.g., `tasksch_userOverRun_thrshldExceedHook`) can be used to record final system state or emit diagnostic logs.

---

6. Troubleshooting

Repeated Initialization or Reset

If the system repeatedly resets and fails to run, check the following.

- **Timer ISR**
  - Verify that `tasksch_timeManager()` is called exactly every 1 ms.
- **Clock Frequency**
  - Insufficient MCU clock speed may cause task execution delays and premature watchdog resets.
- **Watchdog Deadline**
  - The major cycle (longest task period) must be significantly shorter than the watchdog timeout.

Identifying Bottleneck Tasks

Overruns often appear in lower-order tasks that are **pushed back** by long-running tasks.

If the root cause is unclear, inspect tasks with:

- short periods, and
- unexpectedly low individual overrun counts (`indiOverRun_totalCnt`).

Such tasks often monopolize CPU time and disrupt overall scheduling.

Corrupted UART Output in Hook Functions

Corrupted or truncated UART output may occur when logging from hook functions.

Common causes include:

- buffer overwrites between consecutive hooks,
- baud rate mismatches, and
- DMA-based transmissions interrupted by asserts or watchdog resets.

Recommended Practices

- Use blocking (polling) UART transmission for diagnostic logs.
- Use separate buffers for major-cycle hooks and shutdown-phase hooks.
- Verify baud rate accuracy (e.g., 115200 bps) using an oscilloscope if necessary.