Parámetros de compilación:

1. **#define _TASK_MICRO_RES:**

Compilará la biblioteca con una resolución de programación de microsegundos, en lugar de la resolución predeterminada de milisegundos.
Todos los parámetros relevantes para el tiempo como intervalo de ejecución, retraso, etc. se tratarán como microsegundos, en lugar de milisegundos.
NOTA: El modo de suspensión SLEEP_MODE_IDLE (ver más abajo) se desactiva automáticamente para una resolución de microsegundos. Las constantes de tiempo TASK_SECOND, TASK_MINUTE y TASK_HOUR se ajustan por una duración de microsegundos.

2. **#define _TASK_TIMECRITICAL:**

Compilará la biblioteca con una opción de seguimiento de tiempo crítico habilitada.
La opción de tiempo-crítico realiza un seguimiento de cuándo tuvo lugar la ejecución actual en relación con cuándo fue programada y dónde cae el tiempo de ejecución de la próxima tarea. Dos métodos proporcionan esta información:

Método Task::getStartDelay(): devuelve el número de milisegundos (o microsegundos) entre la hora actual del sistema (millis/micros) y el momento en el que se programó el inicio de la tarea. Un valor de 0 (cero) indica que la tarea comenzó justo a tiempo según la programación.

Método Task::getOverrun(): si getOverrun devuelve un valor negativo, el siguiente momento de ejecución de esta tarea ya pasó y la tarea está retrasada. Lo más probable es que esto signifique que el tiempo de ejecución del método de devolución de llamada de la tarea es demasiado largo o que el intervalo de ejecución es demasiado corto (y, por lo tanto, la programación es demasiado agresiva).
Un valor positivo indica que la tarea está según lo programado y que los métodos de devolución de llamada tienen tiempo suficiente para finalizar antes del siguiente paso programado.

3. **#define _TASK_SLEEP_ON_IDLE_RUN:**

Compilará la biblioteca con la opción de suspensión habilitada (solo placas AVR).
Cuando está habilitado, el programador pondrá el microcontrolador en estado SLEEP_MODE_IDLE si ninguno de los métodos de devolución de llamada de las tareas se activó durante la ejecución. El estado INACTIVO es interrumpido por temporizadores una vez cada 1 ms. Poner el microcontrolador en estado IDLE ayuda a conservar energía. Un dispositivo en SLEEP_MODE_IDLE se activa con todas las interrupciones de hardware y temporizador, por lo que la programación se mantiene actualizada.
NOTA: Esta opción de compilación no está disponible con la opción de resolución de microsegundos.

4. **#define _TASK_STATUS_REQUEST:**

Compilará **TaskScheduler** con soporte para el objeto **StatusRequest**. Las solicitudes de estado son objetos que permiten que las tareas esperen un evento y se indiquen entre sí la finalización del evento.
NOTA: tas de la versión 2.2.1, cada tarea tiene un objeto **StatusRequest** interno, que se activa en el momento en que se habilita la tarea y se activa en el momento en que se deshabilita la tarea. Estos eventos podrían ser utilizados por otras tareas para la ejecución basada en eventos.

5. #define _TASK_WDT_IDS

Compilará TaskScheduler con soporte para ID de tareas y puntos de control. A cada tarea se le puede asignar (y de forma predeterminada) se le asigna una ID, que podría usarse para identificar la tarea en caso de que haya un problema con ella. Además, dentro de la tarea, se podrían definir puntos de control para ayudar aún más a identificar áreas problemáticas potenciales. Por ejemplo, las tareas que tienen que ver con recursos externos (sensores, comunicaciones en serie, cualquier cosa que dependa del hardware) pueden bloquearse (o colgarse) debido a un hardware fallido. En este caso, se podría emplear un temporizador de vigilancia para detectar dicha tarea fallida e identificar cuál (por ID de tarea) y dónde exactamente dentro de la tarea (por un punto de control) es probable que se encuentre el problema.
NOTA: de forma predeterminada, los ID de conversación se asignan secuencialmente (1, 2, 3,…) a las tareas a medida que se crean. El programador puede asignar una identificación de tarea específica. Los identificadores de tareas son números enteros sin signo.
Los puntos de control proporcionan una forma de identificar posibles puntos problemáticos dentro de una tarea. Los puntos de control también son números enteros sin signo. Tenga en cuenta que solo hay un punto de control por tarea y se establece en cero cuando se invoca el método de devolución de llamada de la tarea (esto se hace para evitar que el punto de control 'desviado' de las tareas anteriores confunda las cosas).
El ejemplo n.° 7 contiene una prueba de la funcionalidad de ID de tarea y puntos de control.

6. #define _TASK_LTS_POINTER

Compilará TaskScheduler con soporte para el puntero de almacenamiento de tareas local (LTS). LTS es un puntero genérico (void*) que podría configurarse para hacer referencia a una variable o estructura específica de una tarea en particular. Un método de devolución de llamada puede obtener acceso a variables específicas al obtener una referencia a una tarea actualmente en ejecución desde el programador y luego convertir el puntero LTS (void*) al tipo de puntero apropiado.
NOTA: los parámetros anteriores están DESACTIVADOS de forma predeterminada y deben habilitarse explícitamente colocando declaraciones #define apropiadas delante de la declaración #include para el archivo de encabezado de TaskScheduler.

7. #define _TASK_PRIORITY

Compilará TaskScheduler con soporte para priorización de tareas en capas. La priorización de tareas se logra creando varios programadores y organizándolos en capas de prioridad. Las tareas se asignan a programadores correspondientes a su prioridad. Las tareas asignadas a las capas 'superiores' se evalúan para su invocación con mayor frecuencia y se les da prioridad en la ejecución en caso de coincidencia de programación. Más información sobre la priorización por capas en la documentación de la API y los ejemplos de TaskScheduler.

8. #define _TASK_STD_FUNCTION

Compilará TaskScheduler con soporte para funciones estándar (esp8266, esp32 y algunos otros marcos las admiten).

9. #define _TASK_DEBUG

Compilará TaskScheduler con todos los métodos y variables privados y protegidos expuestos como públicos. No debe usarse en producción.

10. **#define _TASK_INLINE**

Compilará TaskScheduler con todos los métodos declarados en línea permitiendo que el compilador optimice.

11. **#define _TASK_TIMEOUT**

Compilará TaskScheduler con soporte para tiempos de espera generales de tareas. Se puede configurar el tiempo de espera de cualquier tarea después de un cierto período de tiempo, y el tiempo de espera se puede restablecer (por lo que el tiempo de espera podría usarse como una especie de temporizador de vigilancia de una tarea individual).

12. **#define _TASK_OO_CALLBACKS**

Compilará TaskScheduler con soporte para enlace dinámico. Esto es útil si prefiere implementar Tareas como clases derivadas de la clase Tarea. Un ejemplo de este enfoque está aquí: biblioteca PainlessMesh.

13. **#define _TASK_EXPOSE_CHAIN**

Compilará TaskScheduler con soporte para acceder a métodos de cadena de programación y tareas en la cadena.

14. **#define _TASK_SCHEDULING_OPTIONS**

Compilará TaskScheduler con soporte para diferentes opciones de programación de tareas. De forma predeterminada, las tareas se invocan para mantener la programación original. Por ejemplo, si una tarea está programada para ejecutarse cada 10 segundos y comienza a las 9:00:00, el programador intentará invocar tareas lo más cerca posible de las 9:00:10, 9:00:20, independientemente de cuándo se realizó la tarea anterior. la invocación de la tarea realmente ocurrió. Esto significa que el intervalo entre invocaciones de tareas podría ser inferior a 10 segundos. Esta es una opción TASK_SCHEDULE. Por la misma razón, es posible que la tarea necesite 'ponerse al día' con la programación original si una tarea se retrasó más allá de uno o varios puntos de invocación, por lo que puede ver invocaciones rápidas hasta que la tarea se 'ponga al día' con la programación. Esto generalmente se debe a métodos de bloqueo (por ejemplo, retardo()) o métodos de devolución de llamada de larga duración que se comportan mal. TASK_SCHEDULE es una opción de programación predeterminada. Otra opción es TASK_SCHEDULE_NC, que tiene un comportamiento similar a TASK_SCHEDULE, pero sin 'ponerse al día'. En este escenario, se invoca una tarea en el siguiente punto programado, pero es posible que el número de iteraciones no sea correcto. Por ejemplo, una tarea programada para ejecutarse cada 6 segundos debería tener 10 iteraciones en 1 minuto. Si dicha tarea se bloquea durante 20 segundos, no podrá realizar las 10 iteraciones en 1 minuto. La última opción, TASK_INTERVAL, programa la siguiente invocación con prioridad a un 'período'. Por ejemplo, una tarea programada para las 9:00:00 con un intervalo de 10 segundos que en realidad se invocó a las 9:00:06 se programará para la siguiente invocación a las 9:00:16.

15. **#define _TASK_DEFINE_MILLIS**

Forzará la declaración directa del estilo 'C' de millis() y micros().

16. **#define _TASK_EXTERNAL_TIME**

Forzará el uso de métodos externos millis() y micros() como fuente de reloj. El desarrollador debe proporcionar los métodos _task_millis() y _task_micros().

17. **#define _TASK_THREAD_SAFE**

Utilizará un mutex interno para proteger los métodos de programación de tareas contra apropiaciones y comportamientos inesperados. Esta es una opción recomendada para esp32 y/u otras MCU que ejecutan TaskScheduler bajo un programador preventivo como FreeRTOS.