

Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services along side device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of “board support packages (BSP)” to a complete “project support package (PSP)”, a feature enabling projects to be greatly accelerated.

This document discusses the implementation and use of Global Software and Hardware Timers in the V1.3 release.

Introduction to the µTasker Global Timers

The µTasker Mono-stable Timers are discussed in the operating system introduction “*uTaskerV1.3.doc*” and the Global Timers are also introduced.

Here the advantages of this technique are discussed in more detail and the options to test these in the µTasker demo project are detailed.

Although it is often possible for a system to fulfil all requirements using the task mono-stables – one task has one unique mono-stable timer if this is defined in its configuration – there are certain circumstances where either multiple or high resolution timers are of advantage or necessity. These are made available by a special task called the Global Timer Task.

The Global Timer Task is activated by using the define GLOBAL_TIMER_TASK in config.h. It has its own mono-stable timer which is however used together with a timer queue to realise multiple timers. The amount of timers supported can be set by using the define TIMER_QUANTITY and the value should be chosen to suit the maximum parallel timer use by the system.

A demonstration of the Global Software Timers in action

The µTasker demo project includes a demonstration of using and testing such timers. This can be activated in the file application.c by commenting in the define TEST_GLOBAL_TIMERS.

When this define is set, several timers are started in parallel and their timeout events are handled so that their operation and capabilities are visible.

Three parallel timers are started in the routine fnStartGlobalTimers():

```

static void fnStartGlobalTimers(void)
{
    CONFIG_TIMER_TEST_LEDS();           // configure and light 2 test LEDs
    TIMER_TEST_LED_ON();
    TIMER_TEST_LED2_ON();
#ifdef GLOBAL_HARDWARE_TIMER
    uTaskerGlobalMonoTimer( (UTASK_TASK) (OWN_TASK | HARDWARE_TIMER),
        (CLOCK_LIMIT) (10*MILLISEC),  E_TIMER_TEST_10MS ); // start a 10ms timer
    uTaskerGlobalMonoTimer( (UTASK_TASK) (OWN_TASK | HARDWARE_TIMER),
        (CLOCK_LIMIT) (3*MILLISEC),   E_TIMER_TEST_3MS ); // start a 3ms timer
    uTaskerGlobalMonoTimer( (UTASK_TASK) (OWN_TASK | HARDWARE_TIMER),
        (CLOCK_LIMIT) (5*MILLISEC),   E_TIMER_TEST_5MS ); // start a 5ms timer
#else
    uTaskerGlobalMonoTimer( OWN_TASK, (CLOCK_LIMIT) (10*SEC),
        E_TIMER_TEST_10S ); // start a 10s timer
    uTaskerGlobalMonoTimer( OWN_TASK, (CLOCK_LIMIT) (3*SEC),
        E_TIMER_TEST_3S ); // start a 3s timer
    uTaskerGlobalMonoTimer( OWN_TASK, (CLOCK_LIMIT) (5*SEC),
        E_TIMER_TEST_5S ); // start a 5s timer
#endif
}

```

Note that the code will either test slow software timers (in the seconds range) or fast hardware timers (in the ms range) depending on the configuration of hardware timer support. Initially the discussion will concern Global Software Timers and afterwards treat the Global Hardware Timer option – note however that both types can also be used together.

The three `uTaskerGlobalMonoTimer()` calls are used to start three parallel timers belonging to the calling task. Each timer is defined to have a timeout period represented in seconds: eg.

```
(CLOCK_LIMIT) (3*SEC)
```

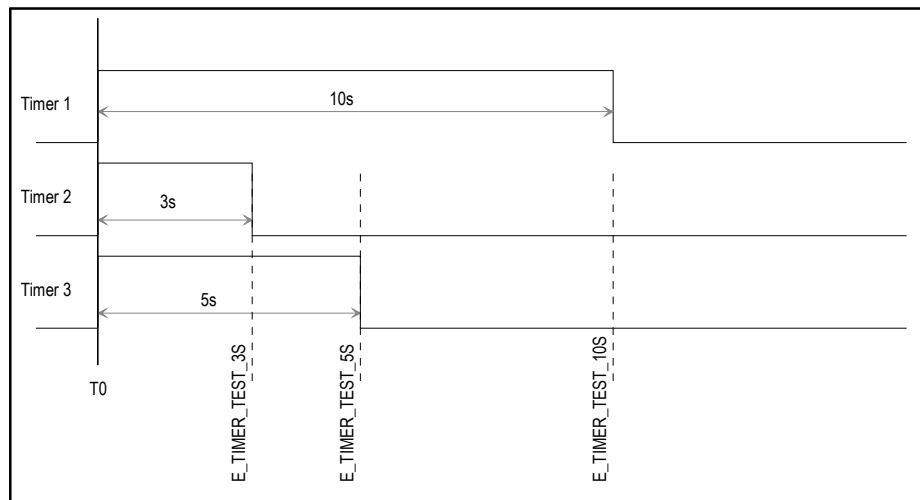
The limit is defined in `types.h` for a project, depending on the longest timeout which can be supported.

Each timer is defined a unique event, eg.

```
E_TIMER_TEST_3S
```

Note that if a timer belonging to a particular task is started with the same event number as one which is already in operation, this results in the original one being ‘retriggered’. A new event number will cause a further parallel timer to be started assuming that there is enough space in the Global Timer Queue pool (see `TIMER_QUANTITY`). Note also that an event with the value 0 is not allowed so the event number must always be a non-zero value from 1..255.

As it stands, the following actions have thus been initiated:



The Global Timer Task is responsible for the management of the Global Timers in the system and when one (or more) timers time out, it sends a corresponding timer event to the owner task. This timer event is in fact compatible with the local Mono-stable timer event and is received in the task's input buffer as discussed in the µTasker operating system document "**uTaskerV1.3.doc**".

The demo project handles timer events in the routine fnHandleGlobalTimers().

```
// Test timer event handler
//
static void fnHandleGlobalTimers(unsigned char ucTimerEvent)
{
    switch (ucTimerEvent) {
        case E_TIMER_TEST_3S:
            TIMER_TEST_LED_OFF();
#ifdef GLOBAL_HARDWARE_TIMER
            uTaskerGlobalMonoTimer( (UTASK_TASK)(OWN_TASK | HARDWARE_TIMER),
                (CLOCK_LIMIT)(3*MILLISEC), E_TIMER_TEST_3MS ); // restart 3ms timer
#else
            uTaskerGlobalMonoTimer( OWN_TASK,
                (CLOCK_LIMIT)(3*SEC), E_TIMER_TEST_3S ); // restart 3s timer
#endif
            break;

        case E_TIMER_TEST_5S:
            TIMER_TEST_LED_ON();
#ifdef GLOBAL_HARDWARE_TIMER
            uTaskerGlobalStopTimer( (UTASK_TASK)(OWN_TASK | HARDWARE_TIMER),
                E_TIMER_TEST_3MS); // kill the 3ms timer
            uTaskerGlobalMonoTimer( (UTASK_TASK)(OWN_TASK | HARDWARE_TIMER),
                (CLOCK_LIMIT)(4*MILLISEC),E_TIMER_TEST_10MS ); // shorten 10 timer
            uTaskerGlobalMonoTimer( (UTASK_TASK)(OWN_TASK | HARDWARE_TIMER),
                (CLOCK_LIMIT)(4*MILLISEC),E_TIMER_TEST_4MS );// start a new 4ms timer
#else
            uTaskerGlobalStopTimer( OWN_TASK, E_TIMER_TEST_3S); // kill the 3s timer
            uTaskerGlobalMonoTimer( OWN_TASK,
                (CLOCK_LIMIT)(4*SEC), E_TIMER_TEST_10S ); // shorten 10s timer
            uTaskerGlobalMonoTimer( OWN_TASK, (CLOCK_LIMIT)(4*SEC),
                E_TIMER_TEST_4S ); // start a new 4s timer
#endif
            break;

        case E_TIMER_TEST_10S:
            TIMER_TEST_LED_OFF();
            break;

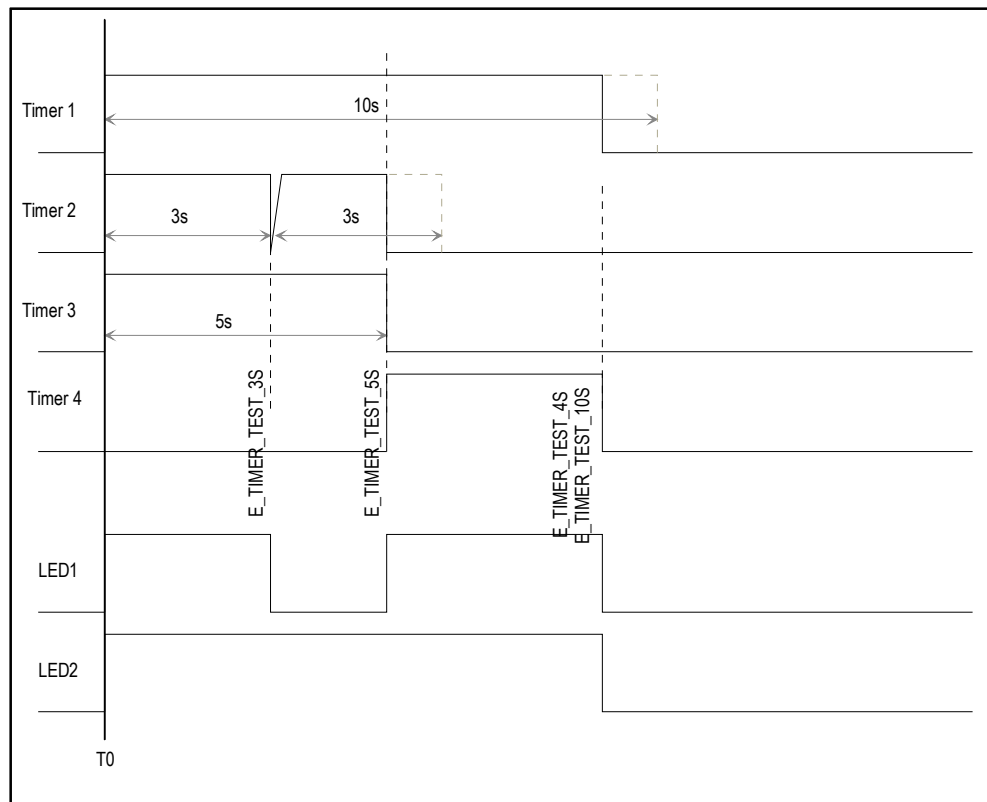
        case E_TIMER_TEST_4S:
            TIMER_TEST_LED2_OFF();
            break;

        default:
            break;
    }
}
```

Again this handler routine supports either Global Software or Global Hardware timers for demonstration purposes. It shows the following functions:

- A timer can be 'retriggered' by calling `uTaskerGlobalMonoTimer()` with an existing owner task/event number combination.
- A timer can be stopped by calling `uTaskerGlobalStopTimer()`.

The result of this code produces the following sequence which can be verified using the µTasker or on the target hardware by monitoring the test LED states.



1. The 3s timer fires and distinguishes the first test LED. It restarts itself immediately with another 3s period.
2. 2s later the 5s timer fires and turns on the first test LED again. It kills the 3s timer (which would otherwise have fired 1s later). It retriggers the 10s timer with a 4s delay and starts a new 4s timer.
3. 4s later (9s after the start of the test) the timers with events E_TIMER_TEST_4S and E_TIMER_TEST_10S fire at the same time. E_TIMER_TEST_10S distinguishes the first test LED and E_TIMER_TEST_4S extinguishes the second Test LED.

There are some checks which can be made to ensure that the test has indeed fulfilled its aims.

1. The 3s timer should never fire a second time since it was killed before its second timeout occurred.
2. The 10s timer was shortened to 9s.
3. The 10s and 4s timers fired together.

Note that although a fourth timer is illustrated this is probably using the queue resources freed by the killed timer 2 or the fired timer 3 and so there are in fact a maximum of 3 timer resources used throughout the test.

A demonstration of the Global Hardware Timers in action

If the define GLOBAL_HARDWARE_TIMER is enabled in config.h the demonstration code will use hardware timers and much shorter timeout periods. This demonstrates the use of these timers and their advantages over the Global Software Timers.

Note that the Global Hardware Timer version uses events like E_TIMER_TEST_3MS but these are in fact the same values as used by the second equivalents.

When starting a Global Hardware Timer there are two differences in the call.

The owner task is defined by (UTASK_TASK)(OWN_TASK | HARDWARE_TIMER) – which sets a flag to indicate that the hardware timer resources are to be used – and the timeout period is defined in ms rather than seconds. (CLOCK_LIMIT)(4*MILLISEC)

The reason for using milliseconds is that the Hardware Timers are generally used where short timeout periods with high resolution are required and these tend to be in this range where as the Software Timers tend to be used for longer delays in the seconds range (but can of course also be used in the sub-second range by calling with for example (CLOCK_LIMIT)(0.1*SEC).

The MILLISEC conversion allows fixed delays to be calculated at compiler time and so achieve highest calculation efficiency and is configured to suit the underlying hardware.

The test runs simple faster and so it is best to monitor the test LEDs with an oscilloscope to verify that they are indeed accurate.

Hardware Details

The Global Software Timers operate using the system TICK, which requires one hardware timer on the processor. Processors often have a simple timer called the PIT (Periodic Interrupt Timer) which is designed especially for this purpose.

The Global Hardware Timers require a second hardware timer to operate. This timer is not a periodic timer like the TICK but instead is set up and used only when a delay is really required. When more that one Global Hardware Timer is active the Global Timer Task coordinates the use of the hardware timer so that always the shortest timeout period is programmed and reuses it to generate remaining delays for other outstanding timeouts.

The timer used for this second timer has to be available and free. Its characteristics can also vary greatly between processors types.

Depending on the capability and flexibility of timer available for use, the µTasker usually offers a configuration to achieve either greatest resolution of longest timeout period. This can be set in the project hardware header – eg. app_hw_m5223x.h

```
// Global Hardware Timer setup
//
#ifdef GLOBAL_TIMER_TASK
    #define MILLISEC    LOW_RES_MS    // longest period
    //#define MILLISEC    MED_RES_MS    // compromise between resolution and period
    //#define MILLISEC    HIGH_RES_MS    // highest resolution with shortest max. delay
#endif
```

Here the most suitable setting can be chosen depending on the projects requirements.

The following table shows the timers used and their limitations in the μTasker projects which support the Global Timer operation.

Processor and project configuration	Hardware Timer used	Hardware resolution	Max. delay (approx.)
M5223X – LOW_RES_MS (60MHz)	DMA Timer 3 (32 bit)	71us	81 hours
M5223X – MED_RES_MS (60MHz)	DMA Timer 3 (32 bit)	4us	305 minutes
M5223X – HIGH_RES_MS (60MHz)	DMA Timer 3 (32 bit)	0.26us	19 minutes
NE64 – LOW_RES_MS (50MHz)	T.B.D	T.B.D	T.B.D
NE64 – MED_RES_MS (50MHz)	T.B.D	T.B.D	T.B.D
NE64 – HIGH_RES_MS (50MHz)	T.B.D	T.B.D	T.B.D
SAM7X – LOW_RES_MS (47.92MHz)	TIMER 2 (16 bit)	22us	1.42s
SAM7X – MED_RES_MS (47.92MHz)	TIMER 2 (16 bit)	3us	177ms
SAM7X – HIGH_RES_MS (47.92MHz)	TIMER 2 (16 bit)	1us	44ms
STR91XF – LOW_RES_MS (96MHz)	TIMER 2 (16 bit)	2.6us	174ms
STR91XF – MED_RES_MS (96MHz)	TIMER 2 (16 bit)	0.66us	43.7ms
STR91XF – HIGH_RES_MS (96MHz)	TIMER 2 (16 bit)	0.33us	21.8ms
LPC23XX – LOW_RES_MS (72MHz)	T.B.D	T.B.D	T.B.D
LPC23XX – MED_RES_MS (72MHz)	T.B.D	T.B.D	T.B.D
LPC23XX – HIGH_RES_MS (72MHz)	T.B.D	T.B.D	T.B.D

1. Note that processors with 32 bit hardware timers enable very flexible hardware timer delays. Processors with only 16 bit timers tend to be suited only to short but accurate delays.

2. Note that when simulating hardware timers with the μTasker simulator the accuracy of delays is limited to the TICK rate of the simulator. Generally this is not a restriction since the simulator can still verify basic software operation but this limitation should be understood and tests at full speed on the target are often unavoidable.

Conclusion

This document has explained the use of the Global Software and Hardware timers support in the μTasker project. Global Hardware timers are typically important when short but accurate delays are required whereas the Global Software timer (as are the task's own Mono-stable timers) is best suited to longer delays which require only a resolution equal to the system TICK resolution.

Depending on the hardware platform the longest delay possible with a hardware timer is restricted by the hardware capabilities – these have been represented in a table for easy comparison.

The μTasker demo project has Global Timer support test code which can be activated if required. It illustrates the use and capabilities of these multiple parallel timers and serves as test case for verification of correct functionality.