TIMER in esp32

Jaydeep Shah-7Apr22

The ESP32 has two timer groups, each one with two general purpose hardware timers. All the timers are based on **64 bits counters** and **16 bit prescalers**.

Timer base frequency is 80 mHz.All timer have this bae frequency.

The timer counters can be configured to count up or down and support automatic reload and software reload. They can also generate alarms when they reach a specific value, defined by the software. The value of the counter can be read by the software program.

NOTE: Below functions are only works with ESP chips in Arduino IDE software, Board = ESP32 related board must be selected; otherwise you will get the error during compilation time for Arduino or other boards. Library and Chip design varys with chip to chip / mfg to mfg

Function related to Timer:

1) **timerBegin() – For configure and start the timer**

Syntax: hw_timer_t * timerBegin(uint8_t num, uint16_t divider, bool countUp);

This function will return timer structure/pointer if configuration is successful. If NULL is returned, error occurs and the timer was not configured.

num: select timer number

divider: Prescaller -- select timer divider

countUp: If this parameter is true then Timer value increase with each tick.

2) **timerAttachInterrupt() – Attach ISR function when timer flag set**

Syntax: void timerAttachInterrupt(hw_timer_t *timer, void (*fn), bool edge);

return: void

timer: First parameter is timer pointer (need to pass timer structure)

fn: Second parameter is function pointer - funtion to be called when interrupt is triggered

edge: True if edge trigger, false = level trigger

1. **Level Triggering:** In level triggering the circuit will become active when the gating or clock pulse is on a particular level. This level is decided by the designer. We can have a negative level triggering in which the circuit is active when the clock signal is low or a positive level triggering in which the circuit is active when the clock signal is high.

2. **Edge Triggering:** In edge triggering the circuit becomes active at negative or positive edge of the clock signal. For example if the circuit is positive edge triggered, it will take input at exactly the time in which the clock signal goes from low to high. Similarly input is taken at exactly the time in which the clock signal goes from high to low in negative edge triggering. But keep in mind after the the input, it can be processed in all the time till the next input is taken.

3) timerDetachInterrupt( ) : This function is used to detach interrupt.

Syntax: void timerDetachInterrupt(hw_timer_t *timer);

return: void (nothing).

timer: Pass timer structure pointer argument which you want to detach.

4) **timerStart() : For start timer counter.**

Syntax: void timerStart(hw_timer_t *timer);

return: void / nothing

timer: Pass timer structure pointer argument which you want to start.

5) **timerStop()**: **For stopping timer**.

Syntax: void timerStop(hw_timer_t *timer);

return: void / nothing

timer: Pass timer structure pointer argument which you want to stop.

6) **timerAlarmWrite(): to specify the counter value in which the timer interrupt will be generated.**

Syntax: void timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload);

Return: nothing /void

This function receives as **first input** the pointer to the timer, as **second** the value of the counter in which the interrupt should be generated (ARR), and as **third a flag** indicating if the timer should automatically reload upon generating the interrupt.

6) **timerAlarmEnable()** : **This function is used to enable generation of timer alarm events.**

Syntax: void timerAlarmEnable(hw_timer_t *timer);

Return: void

Argument: timer structure/pointer

7) **timerEnd(): This function is used to end timer**
Syntax: void timerEnd(hw_timer_t *timer);

Note: There are other functions related to timer operation available which deals with counter direction / counter relaod method and prescaller individually.Here are the syntax of all that type of functions.

void timerSetAutoReload(hw_timer_t *timer, bool autoreload);

void timerSetCountUp(hw_timer_t *timer, bool countUp);

ISR

Interrupt Service Routine in esp32:

The interrupt handling routine should have the **IRAM_ATTR** attribute, in order for the compiler to place the code in IRAM. Also, interrupt handling routines should only call functions also placed in IRAM.

That then begs the question, what would you put in RAM that can be read from the instruction bus? The answer is (if I understand correctly) ... instructions (executable code).

When we compile a C source file we end up with an object file that is then linked to produce an executable. During compilation, the different "sections" of the compiled C are placed in different "sections" of the object file. For example, code goes into the ".text" section and initialized data goes into the ".data" section. By flagging a piece of code with the "IRAM_ATTR" we are declaring that the compiled code will be placed in a section called ".dram.text" (I'm making that up as I don't have a reference to hand). What this means is that instead of an executable having just ".text" and ".data" sections, there are additional sections. The ESP32 bootloader, upon startup, will copy those ".dram.text" sections into real RAM at startup before giving control to your application. The RAM is then mapped into the instruction area address space (> 0x4000 0000). This means that control can be passed to this code (as normal) from within your running app and it will "work" because the code lives in the instruction bus address space.

What now remains is "why" you would want to do this? The answer is to consider the alternative. If the code you want to run is NOT in RAM, then where else could it be? The answer is "flash" ... if it is in flash, then when a request to execute that code is received, the code has to be executed from there. Flash on the ESP32 is much slower than RAM access ... so there is a memory cache which can be used to resolve some of that ... however we can't be assured that when we branch to a piece of code that it will be present in cache and hence may need a

slow load from flash.

And now we come to the kicker ... if the code we want to run is an interrupt service routine (ISR), we invariably want to get in and out of it as quickly as possible. If we had to "wait" within an ISR for a load from flash, things would go horribly wrong. By flagging a function as existing in RAM we are effectively sacrificing valuable RAM for the knowledge that its access will be optimal and of constant time.

Note:

The interrupt service routine needs to be a function that returns void and receives no arguments.

Practical code Example: