Karim Ait-Allaoua

200406520

ENSE481

April 10th, 2024

Ense481: Project Final Report

STM32F10[1,2,3] Low Power Analysis

github.com/kaitallaoua/ense481 - REV A April 3rd, 2024

Introduction

Overview

This project is split in two main areas. One for theoretical power analysis of the STM32F10[1,2,3] series microcontrollers (MCU's) (specifically the STM32F103RBT6 and its corresponding Nucleo Development board), and a Hardware Application of Self Power Measurements. Both use a Command Line Interface (CLI) that enables a user to perform specific tests for measurement under external meters, or the 1 ohm shunt resistor self measurement.

STM32CubeIde Considerations

All modes discussed were implemented with ST's provided Hardware Abstraction Layer (HAL). **All** sleep modes must be preceded by suspending the HAL's tick interrupt, which by default uses `TIM4` on the F103RB. Since ST's HAL uses `TIM4` as its internal timebase for peripherals such as the delay function, and timeouts for other HAL commands like `USART` the timer remains on even when all other peripherals are turned off.

Important note for those unfamiliar with using the STM32CubeIde is to watch for its auto generated code from what the user configures in the IOC file. There are certain sections where user code is to be added. If not followed on code regeneration, your changes will be overwritten. From personal experience I severely disliked this flow however have grown to enjoy it.

Low Power Modes

The range of MCU's under analysis have three standard modes of dedicated power saving operation: `Sleep`, `Stop`, and `Standby`. A commonality between these is the complete halting of the CPU. The choices between these modes come down to compromises between power

consumption, wake up time constraints and compatibility with wake up sources. `Sleep` mode has a peculiar feature that allows it to not only sleep immediately but also on exit from an ISR. The latter is not considered in this report, as it functionally is no different, but a matter of preference in project and ISR design. Additionally this report considers only interrupt wakeup not event wakeup. These modes are described in Table 1, a simplified table derived from RM0008 section 5.3.

Sleep

`Sleep` is the shallowest of the three modes. It is highlighted by zero latency, and simple bare-metal usage. It is also the most flexible to wake from as any interrupt acknowledged by the NVIC will do the job. This is at the cost of the least power savings (although still quite decent). This is accomplished via only turning only the CPU clock off, but all other peripherals including Cortex M3 ones on. This then looks like the following where the regulator parameter has no effect. On wake up, execution resumes immediately right after the enter sleep mode function invocation.

```
HAL_SuspendTick();
HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
```

Of note the EXTI callback function for the purposes of this project simply resumes the HAL's tick interrupt, which could immediately follow sleep commands, but is used here for code deduplication:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) { HAL_ResumeTick(); }
```

Stop

`Stop` is the next deepest sleep while still preserving register and SRAM contents. This means it is still compatible and in fact easier out of the box with freeRTOS. This is discussed further in the "FreeRTOS Considerations" section. It suffers in slightly higher wakeup latency where in the HSI "wakes up" and some clock reconfiguration must be done. This is due to the PLL, HSI and HSE being disabled. Further there is the option of configuring the internal regulator into low power mode, where there *seems* to be no downsides in enabling it. Most importantly, to wake up from `Stop`, any EXTI line must be used resulting in less flexibility.

```
HAL_SuspendTick();
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
// Must reconfigure system clocks as they were disabled
SystemClock_Config();
```

Standby

With `Standby` the pattern becomes obvious. Maximum power savings are achieved in this mode at the cost of maximum wake up latency and no runtime memories preserved. This means the only wakeup - better terminology being recovery is an MCU reset. Think of this mode as a shutdown command in a regular computer, where the MCU has completed its tasks and is assumed to not wake for long enough for the diminishing power savings returns to be worth it. It can be accomplished by the WKUP pin, NRST pin, RTC or IWDG.

```
HAL_SuspendTick();
HAL_PWR_EnterSTANDBYMode();
```

Compared to `Stop` mode, where both can use hardware level wake up, if your application can stand the wake latency from a system reset and being put in a reset is tolerable to the application, you should choose `Standby`. If neither is appropriate and low power is a major priority , use `Stop`. In projects I have designed and can envision, the flexibility and ease of use of `Sleep` mode makes it a very preferable mode for most applications. Reserving `Stop` and `Standby` for very power conscious applications like running on battery power.

Mode Name	Awoken From	Register+SRAM contents preserved?	Power Savings (relative)	Wakeup Latency
`Sleep`	Any interrupt	Yes	Low	None
`Stop`	Any EXTI line	Yes	Medium	HSI wakeup+clock reconfigure
`Standby`	*WKUP pin, RTC, external NRST or IWDG	No	High	MCU reset

Table 1: Simplified Sleep mode summary

*Awoken is not accurate here, since the MCU must be reset to “wake”

Alternative Power Reduction Means

Although these methods were not compared in detail, they are still relevant ways to reduce power. The tests were done with a 24MHz clock.

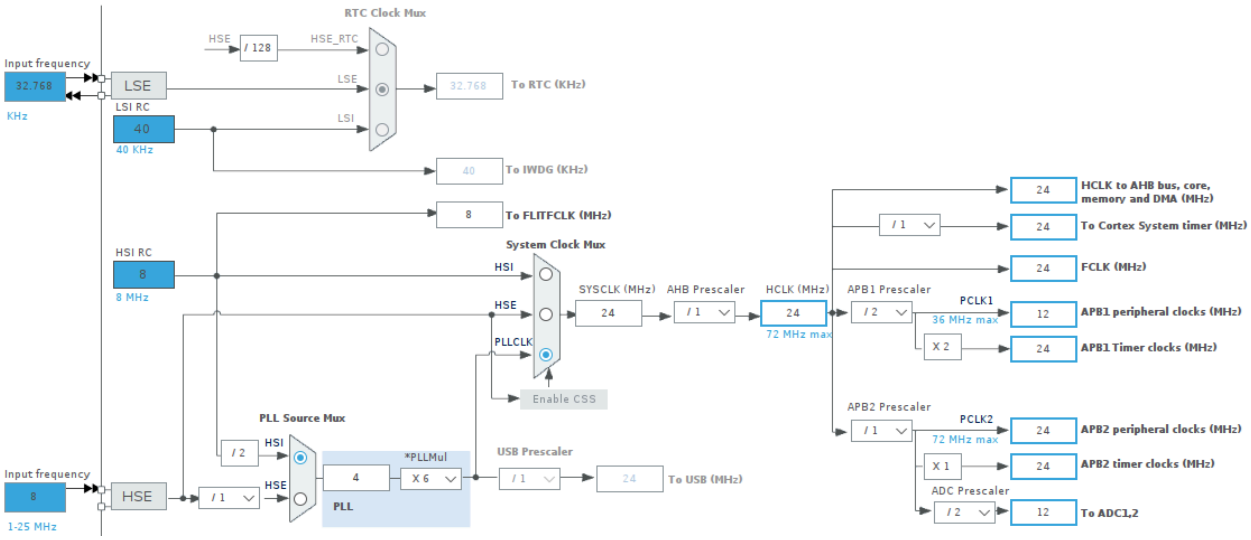

Figure 1: Clock Tree Configuration for Hardware Application

- Slowing down clocks of CPU and peripherals
- Gating (turn off) peripheral clocks when not in use e.g. APB and AHB

FreeRTOS Considerations

My original bare metal code automatically wakes up from `sleep` mode (within 1ms), from the systick interrupt. Unfortunately, getting sleep mode working with freeRTOS is not as simple as bare metal. Unexpectedly, `stop` mode works out of the box for my hardware application where I enter `Stop` mode from a user task directly. From initially reading forum posts it did not seem to be officially supported, but since register and and sram contents are still preserved in `stop` mode, it theoretically should work. ~~I think this kind of a "dirty" setup as the kernel is effectively frozen, and unaware of itself being frozen so any guarantees in task timing requirements are thrown out of the window.~~ Further investigation from community forums and ST's lab video, there seems to be two proper possible directions a user could take, including one seemingly "improper" one:

- Use `Sleep` for kernel in idle, have hardware timer in background wake just before systick overflow and compensate for it, enter sleep again (see Figure 2) (STMicroelectronics, 2021)
- Use `Stop` in kernel for idle, no timer needed, kernel frozen (EyalG, 2022)
- `Stop` in user task directly

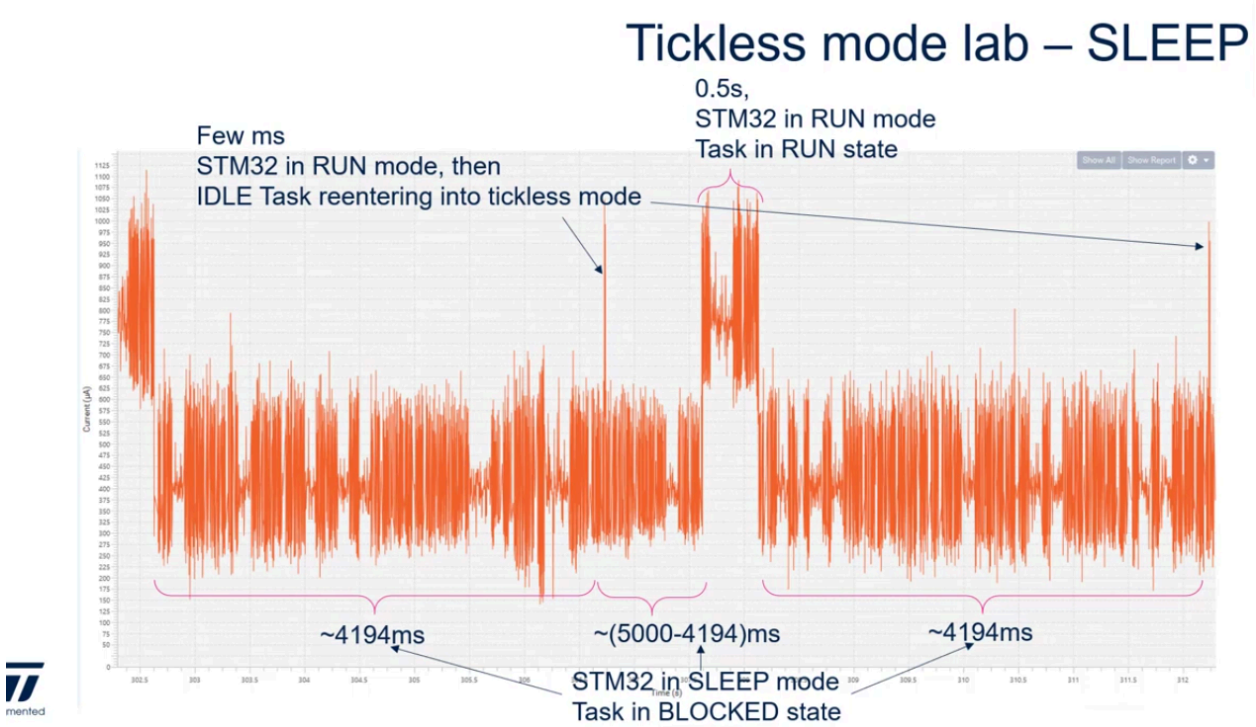

Figure 2: Snippet from ST's Lab video on freeRTOS tickless lab where task in RUN mode is dumb/spinning/delay function.

Figure 2 shows a single task that busy waits for 500ms and uses a low power timer to account for Systick overruns. Of course, `standby` mode has the same implications, an MCU reset is needed.

Results

Figure 3: Fractional Power Savings Comparing Different Modes

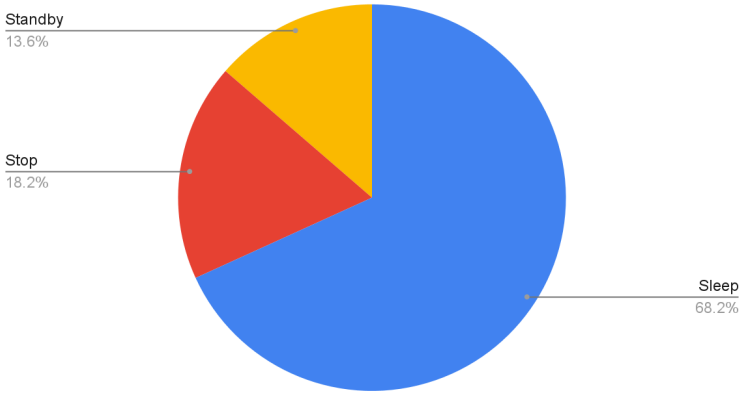

Figure 4: Fractional Lifetime of Panasonic CR2032

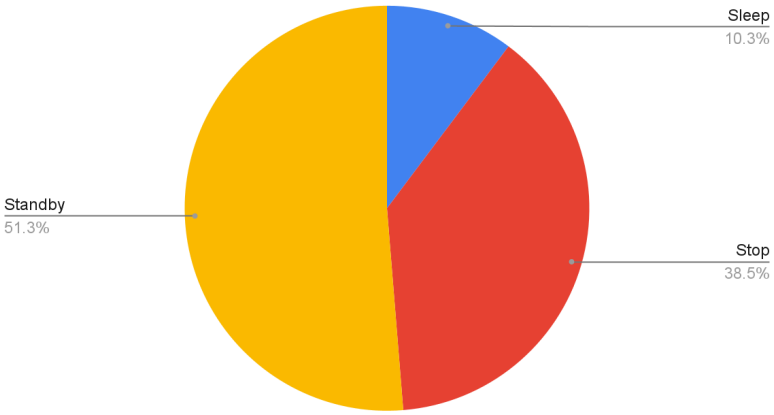

Figure 5: Incremental Power Consumption of Periphrials

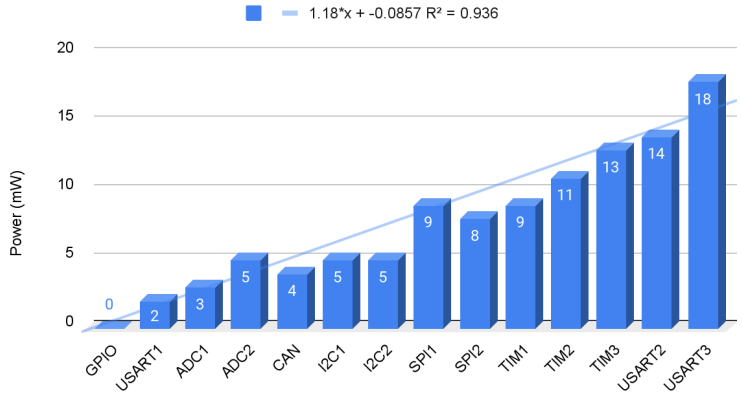

Figure 6: Lifetime of Panasonic CR2032 in Hours

Assuming never leaves sleep mode once entered, actual depends on duty cycle of app

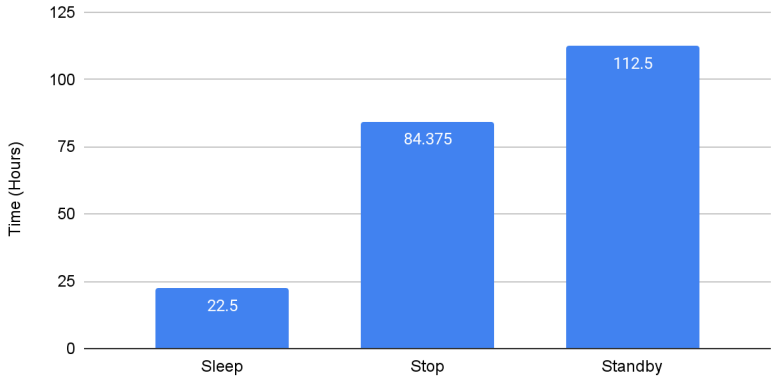

Figure 3 shows the relative power savings obtained by each mode. Note that using `Sleep` at all makes a huge difference (CLI idle absolute power of MCU = 77mW) (in `Sleep` 30 mW), then seeing diminishing returns moving towards `Stop` (8mW absolute mcu), then finally `Standby` (6mW absolute MCU). Then I decided it would be good to run though some theoretical runtime with my application over a Panasonic CR2032 with an energy storage of 675mWh (Master-Instruments, 2024). Figure 4 shows a fractional benefit of the various power modes in Figure 6, where we can see the majority of the benefit is obtained in `Standby` at 112

hours. Interestingly, `Stop` does not seem that far off at 84 hours. Clearly in `Sleep` a mere 22.5 hours is surprising to me, not even a full day! **Most importantly**, this shows that `Stop` should certainly be chosen if used in a battery application and can be awoken from EXTI. It's so worth it! Compared to `Standby` is not fair as the register and SRAM contents are not preserved, you are not really waking up. Finally, using my max CLI mode I slowly enabled all peripherals (except USB) and recorded the power at each step. I initially tried to turn one on, hold, then off but had a tough time discerning the increased power over noise. So I quickly decided to just incrementally turn everything on and linearly regress with a line of best fit to determine an approximate, per peripheral power. As can be seen on Figure 5, the slope of the line is about **1.2 mW/peripheral.** Which is to say hand-wavy: each peripheral uses 1.2mW when enabling their clocks. Of course the power will vary in actually using each one, but that would need a detailed analysis for each. It then makes sense why clocks for these peripherals are disabled on boot: nearly all applications will not use all of them, and their power consumption is certainly noticeable!

Hardware Application: Self Power Measurements

Theory

An (assumed constant) 1 ohm generic 250mW resistor is used as a shunt resistor for measuring the current through the MCU. Two ADC1 channels are used to sense the supply 3.3V and after the resistor as the MCU supply, essentially measuring the voltage across the resistor. From this the simple math is implemented in the `print_pwr` function. This is nice as it provides a simple recipe for self-telemetry in future projects.

Usage

To primarily readout the shunt resistor current and voltage for power measurement. Also includes the demonstration of the various incremental power consumption of peripherals, maximum and minimum power states and various sleep modes. For all modes the user is expected to connect external power metering devices to measure the setup. It follows the bare metal design in `ense481/project/code`. For freertos-tickless usage see `ense481/project/code` which contains a WIP of getting freeRTOS into tickless mode. The latter project's CLI is not discussed here as it is changing too frequently as I attempt to understand a proper implementation.

Command Line Interface

- max : power consumption test enabling all clocks and peripherals
- typ : power consumption test for this program: USART1, ADC1, TIM3
- min : power consumption test all peripherals disabled (except TIM4)
- pwr : free-running print of power readings
- ver : print version info
- hel : this message
- sle : enter sleep mode
- sto : enter stop mode
- sta : enter standby mode

Issues

- I used USART1 instead of the default USART2 that the nucleo uses. I had difficulties removing the solder bridges to switch USART2's to connect at the morpho connector. This could be important if the end application needs the fastest communication of 4.5Mbps which is only achievable on USART1 since its clocked with PCLK2 (72 MHz max)
- I attempted to use the USB peripherals however I was not able to resolve the clock conflicts when adding it with all other peripherals. More time should be spent understanding it.
- Initially for external power profiling I attempted to use the Power Profiler Kit 2 by Nordic, but could not get a current measurement working. Thankfully I was able to use a Joulescope JS110 which fit my needs better as it provided a USB port for convenient plug and measuring since my project needed to be assembled and disassembled every time I wanted to work on the project. A ammeter module could be inserted for direct mcu current measurement.

Future Recommendations

- Since wakeup latency is a consideration, it would be useful to know quantitatively the actual timing differences between `Stop` and `Standby`. For the latter it would depend on what the reset/initialization is.

References

EyalG. (2022, June 27). *Working with FreeRTOS in low power tickless mode for long stop periods*. FreeRTOS Community Forums. https://forums.freertos.org/t/working-with-freertos-in-low-power-tickless-mode-for-long-stop-periods/15332

Master-Instruments. (n.d.). *CR2032/BN*. Retrieved April 3, 2024, from https://www.master-instruments.com.au/products/54908/CR2032/BN.html

STMicroelectronics. (2021). FreeRTOS on STM32 v2 - 21a Low power modes (tickless mode) - Lab [Video]. In *YouTube*. https://www.youtube.com/watch?v=hVlpf1Rm4Qo

Tickless Low power features in FreeRTOS. (2018, September 20). FreeRTOS. https://www.freertos.org/low-power-tickless-rtos.html

ST's datasheets/user manuals/reference manuals/programming manuals

- AN2629
- RM0008
- UM1724
- PM0056