Department of Electrical and Computer Engineering
Technical University of Munich

Power Consumption Report

Atalay Özmen and Hande Şirikçi
TUM School of Computation, Information and Technology
July 17, 2023

1 Measurements with Different Configurations

1.1 Energy of Startup Phase

1.1.1 Measurements without ALS

Experiment no	80MHz	160MHz	240MHz
1	414.24 mAs	370.96 mAs	312.33 mAs
2	539.21 mAs	247.80 mAs	323.48 mAs
3	350.6 mAs	252.37 mAs	323.81 mAs
4	532.64 mAs	350.81 mAs	330.72 mAs
5	699.98 mAs	250.16 mAs	402.33 mAs
6	464.81 mAs	248.29 mAs	475.84 mAs
7	315.49 mAs	312.89 mAs	304.73 mAs
8	343.77 mAs	360.46 mAs	402.97 mAs
9	352.56 mAs	248.45 mAs	395.37 mAs
10	511.07 mAs	244.61 mAs	413.97 mAs
Average	451.9 mAs	288.1 mAs	367.9 mAs

In contrast to our intuition, increasing clock frequency does not always result with decreasing energy consumption. We thought being faster, while increasing the energy spent each second, would lower the total time spent, thus decreasing the total energy consumption. Increasing the clock period helps reducing execution time as long as the logic operations on registers able to finish their work before the next clock. So increasing clock frequency helps energy reduction to some extent, which is 160MHz in our case. It is no longer worth it to increase it further.

1.1.2 160 MHz with Automatic Light Sleep (ALS)

Experiment no	Acc. Energy
1	164.08 mAs
2	204.36 mAs
3	208.96 mAs
4	160.24 mAs
5	215.12 mAs
6	198.04 mAs
7	145.32 mAs
8	222.01 mAs
9	542.44 mAs
10	198.16 mAs
Average	225.8 mAs

Here we see an average consumption lower than the 160 MHz we saw previously, which shows that automatic light sleep actually works and affects the power consumption.

1.1.3 DFS (80, 240)

Experiment no	Acc. Energy
1	286.45 mAs
2	216.16 mAs
3	309.45 mAs
4	306.93 mAs
5	603.52 mAs
6	256.09 mAs
7	181.57 mAs
8	221.80 mAs
9	182.33 mAs
10	202.54 mAs
Average	276.2 mAs

DFS mode enables CPU to adapt to clock frequency in specified range. Since 276.2mAs is less than 288.1mAs, we can infer that the average optimum clock frequency is some value different than 80MHz, 160MHz, and 240MHz.

1.1.4 DFS (80, 240) + ALS

Experiment no	Acc. Energy
1	248.41 mAs
2	199.42 mAs
3	215.52 mAs
4	207.84 mAs
5	313.58 mAs
6	254.95 mAs
7	273.12 mAs
8	277.20 mAs
9	256.44 mAs
10	283.34 mAs
Average	252.958 mAs

ALS helps decreasing energy consumption since sleep modes uses less energy compared to normal executions. We see lower energy consumption compared to our previous measurement.

1.2 Energy of WiFi

`esp_wifi_set_ps(WIFI_PS_MAX_MODEM)`

1.2.1 Listen interval: 1

Experiment no	Acc. Energy
1	281.32 mAs
2	263.61 mAs
3	286.66 mAs
4	312.05 mAs
5	262.35 mAs
Average	280.8 mAs

1.2.2 Listen interval: 2

Experiment no	Acc. Energy
1	203.01 mAs
2	229.57 mAs
3	268.60 mAs
4	273.85 mAs
5	217.25 mAs
Average	238.2 mAs

1.2.3 Listen interval: 6

Experiment no	Acc. Energy
1	198.40 mAs
2	216.96 mAs
3	204.83 mAs
4	239.15 mAs
5	193.52 mAs
Average	210.7 mAs

1.2.4 Listen interval: 10

Experiment no	Acc. Energy
1	177.20 mAs
2	216.40 mAs
3	187.20 mAs
4	206.96 mAs
5	206.99 mAs
Average	198.95 mAs

1.2.5 Without power saving

`sp_wifi_set_ps(WIFI_PS_NONE)`

Experiment no	Acc. Energy
1	1270.24 mAs
2	1272.3 mAs
3	1278.20 mAs
4	1278.44 mAs
5	1284.64 mAs
Average	1278.2 mAs

As listen interval increases, the time between our checks also increases. As we check the WiFi less often, we consume less energy. We also saw that without power saving, we use huge amounts of energy compared to previous ones.

1.3 Energy of Calculation Phase

1.3.1 No ALS

For 80 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	1104.92 mAs : 37.18 sec	29.7
2	1104.92 mAs : 37.18 sec	29.8
3	1104.92 mAs : 37.18 sec	29.8
4	1103.40 mAs : 37.18 sec	29.7
5	1104.96 mAs : 37.18 sec	29.8
Average	1104.62 mAs : 37.18 sec	29.7

For 160 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	796.48 mAs : 18.58 sec	42,86
2	796.40 mAs : 18.58 sec	42,86
3	799.36 mAs : 18.58 sec	43
4	800.08 mAs : 18.58 sec	43
5	797.00 mAs : 18.58 sec	42,89
Average	797.864 mAs : 18.58 sec	42

For 240 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	812.16 mAs : 12.38 sec	65.6
2	812.48 mAs : 12.38 sec	65.6
3	812.76 mAs : 12.38 sec	65
4	815.15 mAs : 12.38 sec	65.8
5	812.40 mAs : 12.38 sec	65.6
Average	812.40 mAs : 12.38 sec	65.6

160 MHz clock frequency was the best in terms of power consumption again, just like in the startup phase.

1.3.2 160 MHz with ALS

Experiment no	Accumulated Energy and Time	mA/sec
1	792.80 mAs : 18.58 sec	42.6
2	789.04 mAs : 18.58 sec	42.46
3	785.72 mAs : 18.58 sec	42.2
4	788.72 mAs : 18.58 sec	42.2
5	785.04 mAs : 18.58 sec	42.25
Average	788.26 mAs : 18.58 sec	42.4

Here, compared to the 160 MHz without ALS we measured earlier, we only see a slight improvement in the power consumption. This could be because of that the calculation function is very CPU hungry and there is not much opportunity to go to sleep during execution.

1.3.3 DFS(80, 240) + ALS

Experiment no	Accumulated Energy and Time	mA/sec
1	753.32 mAs : 12.39 sec	60.8
2	751.40 mAs : 12.39 sec	60.64
3	752.88 mAs : 12.39 sec	60.76
4	753.48 mAs : 12.39 sec	60.81
5	752.48 mAs : 12.39 sec	60.73
Average	752.71 mAs : 12.39 sec	60.75

Since DFS did not make quite change, we could infer that the 160MHz is so close to optimum choice of clock frequency.

1.4 Parallelization of the computation to use both cores

Parallelization of the calculation function was done using the xTaskCreatePinnedToCore() function, where we gave half of the calculation work to one task, and other half to the other task, while configuring the cores 0 and 1 respectively.

For 80 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	661.52 mAs : 21.04 sec	31.4
2	661.36 mAs : 21.04 sec	31.4
3	661.76 mAs : 21.04 sec	31.4
4	664.64 mAs : 21.04 sec	31.5
5	662.12 mAs : 21.04 sec	31.4
Average	662.28 mAs : 21.04 sec	31.4

For 160 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	496.04 mAs : 11.51 sec	43.0
2	497.44 mAs : 11.51 sec	43.2
3	497.48 mAs : 11.51 sec	43.2
4	497.92 mAs : 11.51 sec	43.2
5	497.80 mAs : 11.51 sec	43.2
Average	497.33 mAs : 11.51 sec	43.16

For 240 MHz:

Experiment no	Accumulated Energy and Time	mA/sec
1	503.37 mAs : 8.34 sec	60.3
2	504.65 mAs : 8.34 sec	60.5
3	504.65 mAs : 8.34 sec	60.5
4	504.72 mAs : 8.34 sec	60.5
5	505.24 mAs : 8.34 sec	60.6
Average	504.52 mAs : 8.34 sec	60.4

It is seen that making the executions parallel lowered the execution time compared to the non-parallelized approach we have done previously.

1.5 Measuring power consumption in LightSleep with/without Display

While measuring light sleep, the mAs accumulated during 10 seconds were measured with and without display.

Experiment no	With Display	No Display
1	51.68 mAs : 10 sec	27.16 mAs : 10 sec
2	51.84 mAs : 10 sec	25.08 mAs : 10 sec
3	51.88 mAs : 10 sec	24.24 mAs : 10 sec
4	51.20 mAs : 10 sec	24.24 mAs : 10 sec
5	50.76 mAs : 10 sec	25.12 mAs : 10 sec
Average	51.47 mAs	25.16 mAs

1.6 Measuring power consumption in DeepSleep with/without Display

Experiment no	No Display	With Display
1	0.13 mAs : 10 sec	1.16 mAs : 10 sec
2	0.14 mAs : 10 sec	1.17 mAs : 10 sec
3	0.14 mAs : 10 sec	1.16 mAs : 10 sec
4	0.15 mAs : 10 sec	1.18 mAs : 10 sec
5	0.13 mAs : 10 sec	1.17 mAs : 10 sec
Average	0.138 mAs	1.168 mAs

2 Deep Sleep Implementation

In order to lower the energy consumption and increase the battery life, sleep mode could be implemented to deactivate sensors. In this section we are going to explain our approach of deep sleep. Here, we only execute app main to process the signals that we received when we were in deep sleep, and signals during deep sleep are put in a queue in the wakeup stub.

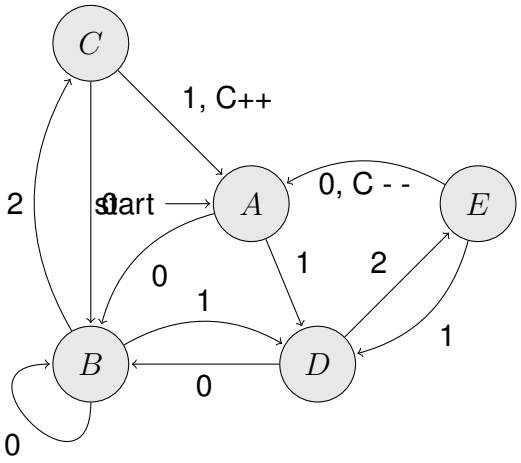
2.1 Our approach of Wakeup Stub

2.2 Changes in FSM

The state transition logic is changed since we are not able to receive 00 barrier status anymore. 00 was the symbol for no barriers activated. But since we are only able to receive signals that are 1, state of 00 is not captured anymore. We are only able to receive 01 11 and 10. And this caused changes in our finite state machine.

barrierStatus	
0	Only outer barrier broken
1	Only inside barrier broken
2	Both barriers activated

Table 1 `barrierStatus` definitions



With this new logic, we increase the count when we have 0 2 1, and decrease when we have 1 2 0. This new algorithm may not include all the edge cases we previously covered due to limitations.

2.3 Changes in `app_main`

For the first time when system starts, `app_main` is running and it checks whether this is the first running or not. `rtc_get_reset_reason(0) == DEEPSLEEP_RESET` statement is to check this condition. According to that, `app_main` takes different actions.

- `rtc_get_reset_reason(0) == DEEPSLEEP_RESET`
  This means the reason of invoking `app_main` is waking up from deep sleep. So we need to process receiving signals, if there is any.
- `rtc_get_reset_reason(0) != DEEPSLEEP_RESET`
  This means the reason of invoking `app_main` is running the system for the first time. We initialize the variables instead of processing the queue.

After this, we need to set the function that will run for the wake up stub.

```
esp_set_deep_sleep_wake_stub(&wake_stub)
```

We also need to define the wake up reason, which is gonna trigger this event. There are many options for wake-up source and we chose `ext1` to assign multiple pins, two different sensors in our case.

```
esp_sleep_enable_ext1_wakeup(
    ext_wakeup_pin_1_mask |
    ext_wakeup_pin_2_mask,
    ESP_EXT1_WAKEUP_ANY_HIGH)
```

`ESP_EXT1_WAKEUP_ANY_HIGH` enables waking up if any of the selected pins is high. But since this wake up source is implemented by RTC controller and RTC peripherals could be powered down, we have to request for RTC peripherals to be active while sleeping. This is done with the code below.

```
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH,
    ESP_PD_OPTION_ON)
```

Finally, deep sleep function `esp_deep_sleep()` is called and the system will be sleeping for the specified duration.

```
esp_deep_sleep(300000000LL);
```

2.4 wake_stub()

This function runs when the system wakes from deep sleep. Since we only have access to RTC slow and fast memory in deep sleep, we should place this function in RTC fast memory. This is possible via adding `RTC_IRAM_ATTR` attribute to the function initialization. And also adding variables to RTC slow memory is possible via adding `RTC_NOINIT_ATTR` attribute to the variable initialization.

We first set the `flag` variable to make sure that we only receive one `barrierStatus` signal. We also periodically call `REG_WRITE(TIMG_WDTFEED_REG(0), 1)` to make sure that the other processes can get enough time to execute.

`thisIsTheStart` is a flag to know whether this is the first execution after `app_main` runned or `wake_stub()` already runned after last run of `app_main`. We initialize starting time and also `debouncing_check` to eliminate noise in receiving data.

After the initialization of values, we have to check which signal we have received. For this purpose, we receive the mask specifying which RTC pin triggered the wakeup in `pinMaskS`. For each branch, we check the pins and also whether any of the other branches has already chosen.

Next step is to check whether the signal received for a time duration that is longer than debouncing threshold. For our case this threshold is 0,3 seconds. If the conditions are satisfied, we place the event in our customized queue, which is an array manipulated to function as a queue. There are functions named as `enqueueEvent` and `dequeueEvent` to achieve this functionality.

An example code for enqueing a signal when both barriers are blocked is like below:

```
if (!flag && pinMaskS & (uint32_t)1 <<
    interruptPinInRTC && pinMaskS & (uint32_t
    )1 << interruptPinOutRTC)
{
    if (now - debouncing_check > 300000)
    {
        enqueueEvent((event_t){.timeStamp
     = now, .barrierStatus = 2});
        debouncing_check = now;
    }
    flag = true;
}
```

After receving the signal, we check whether the system should invoke the `wake_stub()` or `app_main` after the deep sleep. In order to decide this, we have to check whether the time passed since the wake up stub starting time exceeded five minutes or not. If it exceeded, then we shouldn't set the `wake_stub()` as the function to be invoked after deep sleep. Not setting this function will set the `app_main` as the invoking function after deep sleep by default. After make the correct set of invoking function, program sleeps again. We use the code below to check if 5 minutes have passed or not, and set our function.

```
    // Set the pointer of the wake stub
function.
if (now - timeAtStart < 300000000)
{
    REG_WRITE(RTC_ENTRY_ADDR_REG, (
uint32_t)&wake_stub);
    deepsleep_for_us(300000000LL - (now -
 timeAtStart));
}
else
{
    timeAtStart = now;
    // Go into deepsleep
    CLEAR_PERI_REG_MASK(
RTC_CNTL_STATE0_REG, RTC_CNTL_SLEEP_EN);
    SET_PERI_REG_MASK(RTC_CNTL_STATE0_REG
, RTC_CNTL_SLEEP_EN);
}
```

3 Estimation with 1.400 mAs battery

To estimate how long our machine can run with 1.400 mAs battery, we had to do some extra measurements.

First of all, we needed to know how much energy is spent during the wakeup stub execution. To find this out, we looked at the energy consumption during deep sleep and wake up stubs. Because this measurement did not consist of a full deep sleep, we used channel 1 to do our measurements. We did our measurements five times. Each time we spent 30 seconds, and there was enter/exit event for 4 times. A small problem of this approach was the fact that while wakeup stubs energy consumption was calculated right, the deep sleep parts were overestimated, since in channel 1 we can by minimum have 800 uA as the shunt current, which is what is recorded in the accumulated energy part of our Amperemeter. In reality, we saw that this value is way lower, as low as 0.013 mA = 13 uA.

Experiment no	mAs
1	64.32 mAs : 30 sec
2	60.35 mAs : 30 sec
3	56.48 mAs : 30 sec
4	56.64 mAs : 30 sec
5	57.68 mAs : 30 sec
Average	0.138 mAs

We also noted that the shunt current while wakeup stub is running is 14 mA on average. Considering that an average entrance or exit takes one second, where at least one of the signals are blocked, we can say that cost of entrance is 14 mA. Since 14 times 4 is 56, the measurements we did for 4 entrances verify this, since they are close to 56.

Another thing to note when thinking about how long our device will last, is the power consumption during the running of app main, where we run the display again, connect to the wifi, send measurements to IoT platform etc. . In our previous calculations, we have found the average energy consumption for the startup phase to be 252.958 mAs in it's lowest form, where we used DFS (80, 240) + ALS.

Now we can make some assumptions. Let's say we will wake up from deep sleep to run app main every 20 minutes. That means every 20 minutes, we will spend 253 mAs on average. Second assumption can be that every 20 minutes, we will have 10 enter/exit events, that take 1 second on average. This means that every twenty minutes, we would need to use 10 times 14 = 140 mAs. When combined together, we can say that our device will use 253 + 140 = 393 mAs every 20 minutes.

We can simplify this and say that we use 393 * 20 = 1179 mAs per hour. Our battery is 1400 mAh, which would be equal to 5040000 mAs. Therefore, 5040000 / 1179 = 4274.8, which means that we can use it for approximately 4275 hours, which is equal to approximately 178 days.