Preview

This article demonstrates how to use a power profiler to extend the battery life of embedded hardware and how I used the profiler for debugging.

Using a Power Profiler to speed development of embedded hardware

Matthew Oppenheim February 2025

Introduction

A power profiler is an instrument that accurately measures the power that your device under test (DUT) uses real-time. This allows us to see how the changes to the hardware and firmware on our latest embedded project affect the power that our device uses. In this article I explain how I used the Nordic Semiconductor Power Profiler Kit 2 (PPK2)[1] to significantly improve the battery life of a wearable assistive technology project. The profiler also played a major role in debugging when the hardware was misbehaving. A power profiler gives an extra channel of information as to what the hardware is doing in addition to our usual debugging tools. By monitoring how the current varies in a microcontroller, security experts such as Colin O'Flynn can even extract enough information as to what the internals of the microprocessor are doing to crack the encryption of supposedly secure data[2].

Enter the PPK2

Figure 1 Shows the PPK2 connected to a Lilygo T-Watch S3 that I used for the project presented in this article. More on this programmable smart-watch later. A friend introduced me to the PPK2. He is lucky enough to have his employer supply the instrument. I bought mine for around $100 so it is not prohibitively expensive. I found more expensive power profilers with extra features, such as the ability to vary the output voltage and current to mimic a discharging battery. I can't justify paying the extra for this functionality.

The PPK2 powers the DUT and connects to a PC or laptop by a USB cable. The PPK2 is controlled from your PC using a software graphical user interface (GUI). I'll explain how to download and install the software later. There is an extra power socket in the PPK2 to allow an external power supply to provide more power to the DUT than the PPK2 can source through the USB port that connects with your PC.

The PPK2 can supply up to 1A. The resolution of the current measurement varies from 100nA to 1mA depending on how much current is supplied. For the 25-120mA that the DUT I use draws, the current measurements look stable to a resolution of 0.1mA. The frequency that the current is

measured at is 100 kHz. The voltage that the PPK2 supplies to the DUT is set in the GUI within a range of 0.8 to 5V with a precision of 1mV. I typically supplied 3700mV from the PPK2 to the T-Watch S3. I verified that the value of voltage that I set in the GUI exactly matched with the reading on my multimeter. The voltage value of 3700mV was chosen to represent a partially discharged lithium battery. I need to know that the watch still works when the battery is partially discharged, not just when the battery is freshly charged.

Those of you who had a good look at the photo of the PPK2 may be asking what the ‘LOGIC PORT’ socket does. So far, I haven’t used this. According to the documentation, this port allows signals from your embedded device to be displayed on the current use display. This looks to be a useful tool which could be used to indicate when peripherals turn on or off so that we can clearly correlate this with any change in current.

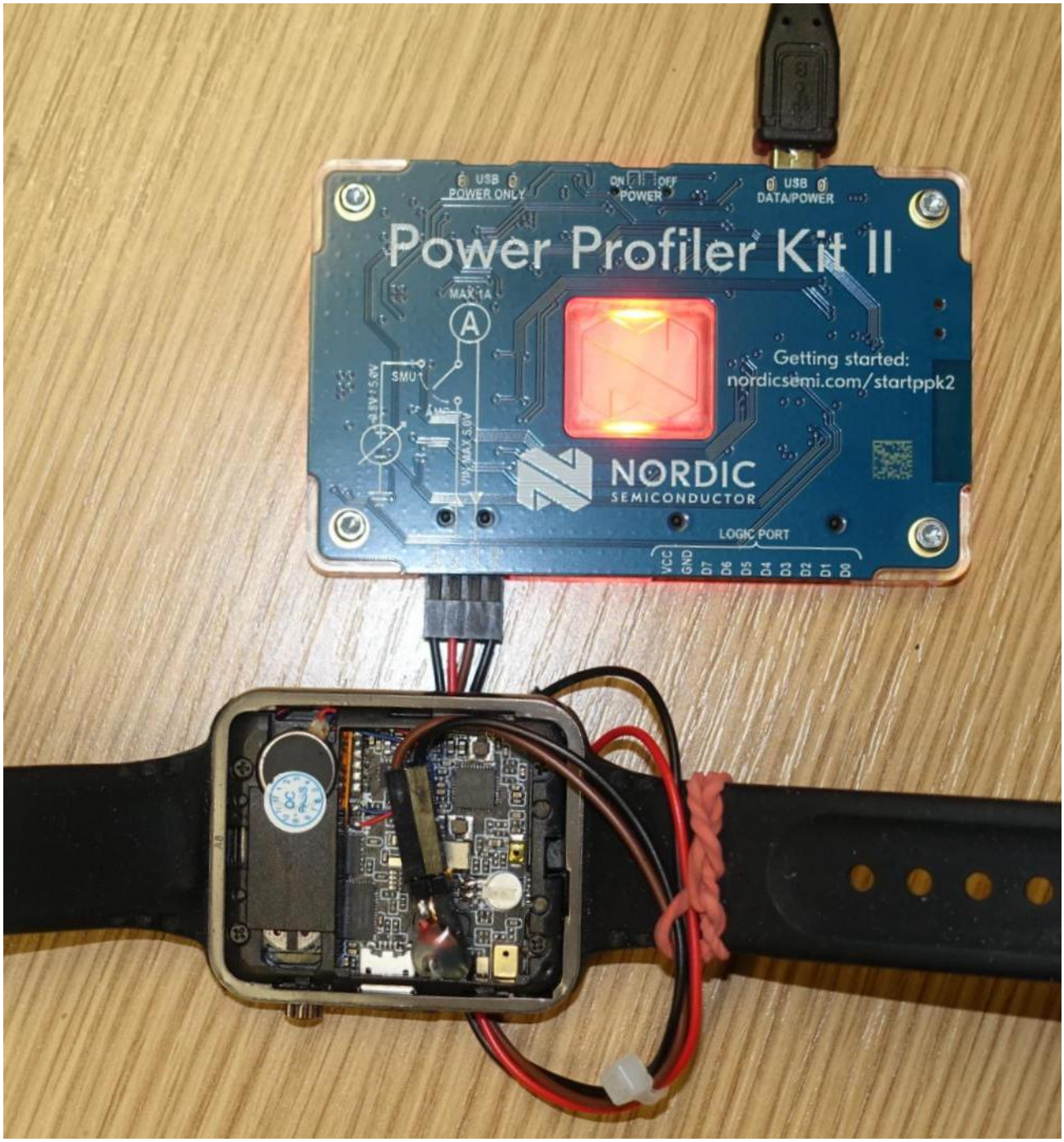

Figure 1: Power Profiler Kit II connected to Lilygo T-Watch S3. The red LED indicates that the PPK2 is powering the watch.

Software setup and use

One tip: Connect the PPK2 directly to your laptop or PC, not through a USB hub. Not all hubs are created equal. Connect your PC to the micro USB socket labelled ‘USB DATA/POWER’. Connect extra power supplies or battery banks to the socket labelled ‘USB POWER ONLY’ if you need to supply more current to your DUT than can be sourced through the data link USB port. There is also a tiny on/off switch in the middle of the top edge of the device which is easy to miss! When the PPK2 is powered a reassuring green light pulses in the middle.

To install the software that controls the PPK2 and displays the real-time current draw, head to this link and download the relevant software for your operating system: [3]. I use Linux and downloaded an appimage. I have not tried using the PPK2 with other operating systems. The appimage starts a graphical user interface (GUI) with options to install a number of packages for different nRF devices. Click on ‘Install’ next to the Power Profiler option. Connect your PPK2 to a USB socket and start the Power Profiler. I received a message ‘Jlink not detected’ the first time that I ran this. The nRF app is looking for driver software. Install this from Segger at the link in the resources[4]. Now kill and restart the appimage. Open the Power Profiler app. The appimage automagically detects updates and asks if you want to install them when you start up.

Click on ‘SELECT DEVICE’ in the top left corner of Power Profiler window. Your PPK2 should appear with a serial number. I usually click the ‘AUTO RECONNECT’ option at this stage. Click on the PPK2 and you should see a ‘PPK started’ and ‘PPK opened’ message in the log which indicates that the PPK2 is communicating with the GUI. The first time that you connect with your PPK2 you may be asked to program the device. Accept this offer.

There are a surprising amount of controls to set to power up your DUT! At the top left of the Power Profiler window you have options ‘Source meter’ and ‘Ampere meter’. I connect my PPK2 to my DUT then select ‘Source meter’. I set the supply voltage in mV and move the slider on the ‘Enable the power output’ to on. Then click the ‘Start button’. Now you should see a scrolling display showing the varying current draw of your DUT. You can zoom in and out using the scroll button on your mouse. The average current for the data shown in the window is shown at the bottom. This allows you to quickly compare regions where parameters have changed, for instance by zooming into an area where the radio is turned on and comparing the current used against a time period where the radio is turned off.

An example display is shown in Figure 2. The prominent elevated square on the display shows when the radio was active on the T-Watch S3. I was confused over what caused the smaller peaks on this display. The PPK2 allows you to measure time intervals. I found these peaks are exactly 1ms apart. What peripheral is turned on with a frequency of 1kHz? After some digging I found out that this is the refresh rate for the screen on the T-Watch S3.

For whatever reason, the reassuring softly pulsing green LED in the PPK2 turns to a frightening shade of red when it is powering your DUT. In this case, red is good.

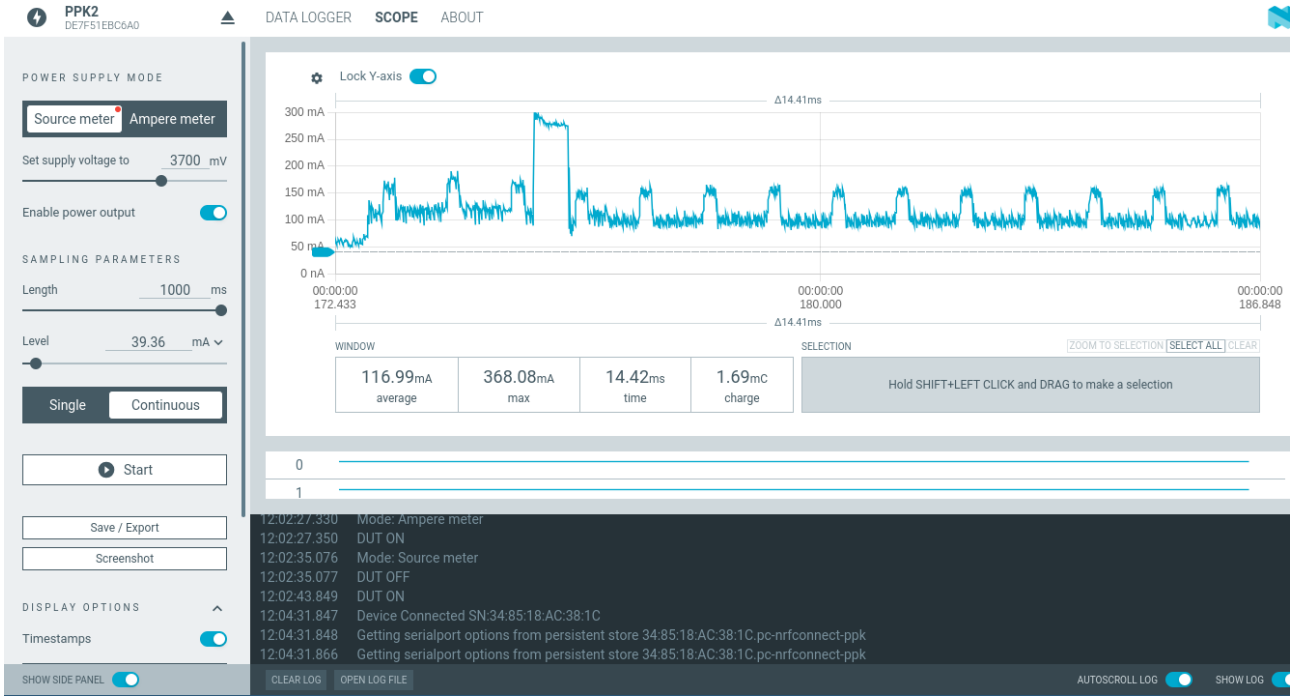

Figure 2: PPK2 display showing current supplied to the T-Watch S3, sampled at 100kHz

Handshake application

I am working on a project to help enable people who need to use software to create speech to be able to access this technology through hand motion[5]. I call this project handshake. Many people who are unable to speak use communication software, e.g. Sensory Software's Grid, to create speech. This software is often operated using buttons or joysticks and is known as switchable software. Some people are unable to use these controllers but are still able to make intentional hand movements. This project aims to detect this movement and use it to operate as a standard assistive technology switch.

I recently ported my home-made assemblies of PCBs to run on two off-the-shelf modules made by Lilygo. These are the Lilygo T-Watch S3[6] and the Lilygo T-Embed[7], see Figure 3. Moving from my home-made boards to manufactured devices in presentable cases enables the technology to be safely distributed for evaluation. A short video demonstrating how the modules are used to enable people to create speech is listed in Resources [8]. The T-Watch S3 is used to track hand movement and detect when an intentional movement is made. The T-Embed has a PCB added to enable it to act as an assistive technology switch to operate communication software when a radio trigger is received from the watch.

The battery inside the T-Watch S3 is connected to the watch by two soldered wires. I cut these wires one at a time so that the blades of my cutter didn't short out the battery then soldered on a connector to allow the PPK2 to power the T-Watch S3. Now the PPK2 powers the T-Watch S3 and measures the current supplied to the T-Watch S3. I used some hot-melt glue and a rubber band to add strain

relief to where the power wires are attached to the T-Watch PCB as I don’t want these tearing free from the board. Have a look at Figure 1 to see these features.

Figure 3: Lilygo T-Watch-S3 and Lilygo T-Embed modules used for the handshake project

The T-Embed is powered from a USB-C cable connected to a powered laptop or tablet. As the watch is battery powered and needs to be continuously active to monitor the accelerometer, configuring the firmware to maximise the battery life on the watch was critical. Initially, I only got about 20 minutes battery life! I needed to extend this or my project was dead in the water.

How I extended the battery life

The Lilygo T-Watch S3 is powered by a 400mAh rechargeable Lithium battery. I based my initial project setup on the code examples supplied by the manufacturer for the T-Watch S3[9]. The example code that comes with the watch demonstrates a graphical display and haptic feedback which are two of the features I need to implement. The examples use VSCode with the PlatformIO plug-in as the development platform. I often use VSCode as a programming IDE. Installing the PlatformIO plug-in is straight forward and well documented online. There is a choice of working with Arduino or ESP-IDF libraries. I chose the Arduino libraries as this is what the example code uses.

I wrote some firmware that demonstrated the basic functionality that the handshake project needs. This includes two-way radio communication between the T-Watch S3 and the T-Embed, graphical displays, file storage, real-time processing of accelerometer data and haptic feedback on the T-Watch S3. I had several project requirements to solve:

- Getting reliable radio communication between the T-Watch S3 and the T-Embed.
- Getting the graphical display to show what I wanted.
- Having the T-Watch S3 react to an intentional soft shaking motion when there is a background of large, unintended shakes.

One by one I solved how to implement these features. However, the battery life for the initial code was truly appalling, only around 20 minutes. Figure 4 shows the current use display from the PPK2 for the initial working code. A large current draw when the T-Watch D3 was waiting to detect a

handshake and an even larger current draw when the radio transmitted a signal to the T-Embed that a handshake gesture was detected. How could I reduce the current used by the T-Watch S3?

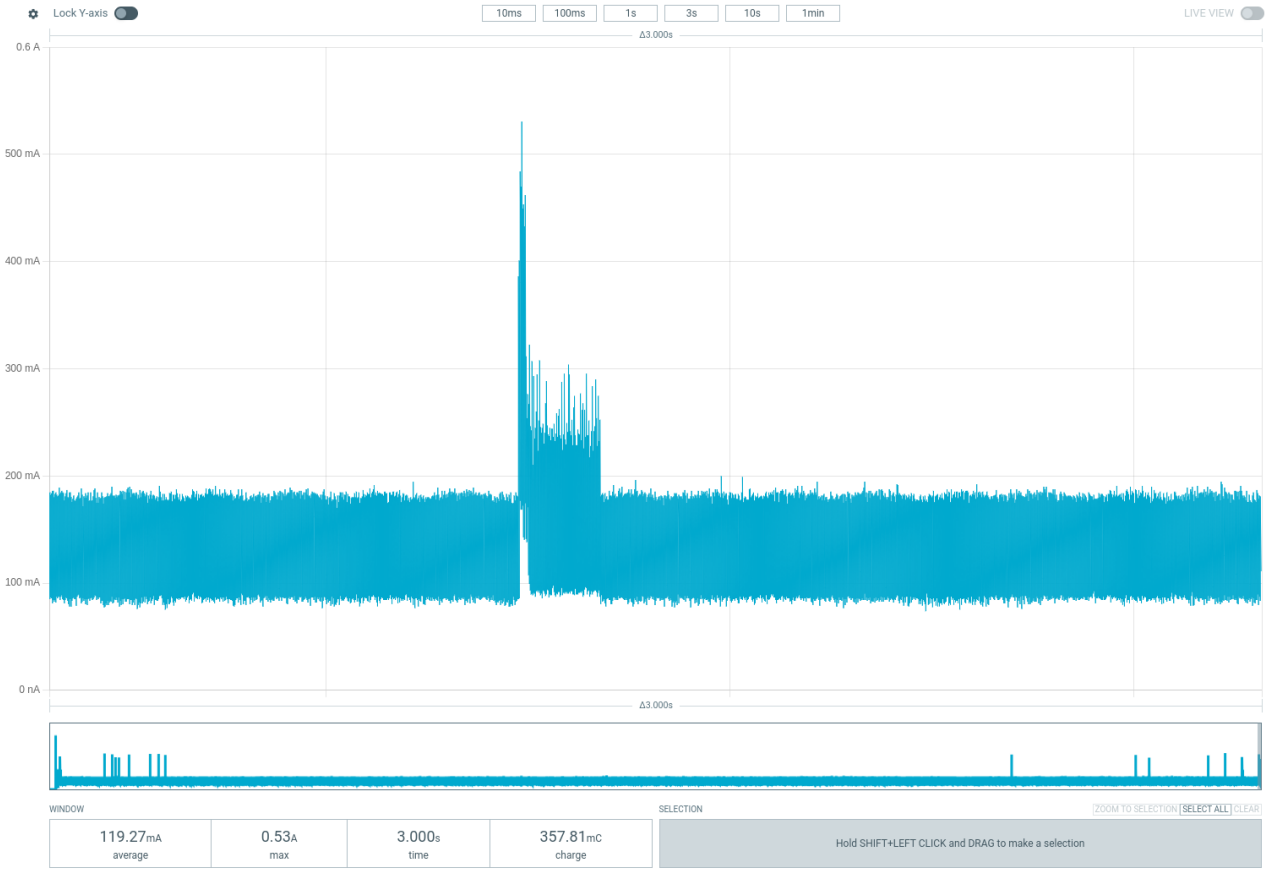

Figure 4: T-Watch S3 current draw for the initial firmware on the T-Watch S3, showing the radio turning on and off to transmit a handshake detection trigger to the T-Embed module

CPU clock frequency

The slower that we clock a CPU at, the less power it needs. So looking at what frequency the ESP32-S3 in the T-Watch S3 was running at seemed a good place to start. First of all, I had to find where this was set in the project setup files that I copied from the example code for the T-Watch S3.

I found a file called LilyGoWatch-S3.json. This has various set up parameters. This line sets the CPU frequency:
"f_cpu": "240000000L" [Editor note: use code font]
The CPU is set to a frequency of 240MHz. Why so fast?
Another way to set the CPU frequency is in the platformio.ini file, with e.g..
board_build.f_cpu = 160000000L # 160MHz [Editor note: use code font]
Changing the platormio.ini file over rules the LilyGoWatch-S3.json file. However, changing either leads to lengthy full compile of the firmware when you next build the project.

To display the CPU frequency on a terminal connected through a serial port with the T-Watch S3 I use this command in the firmware:

Serial.printf("CPU frequency : %d MHz\r\n", ESP.getCpuFreqMHz()); [Editor note: use code font]

So I have two places I can set the frequency that are not immediately visible when I look at the code. I don’t like this.

Rather than using either of the configuration files to set the CPU frequency, I explicitly set it in my main.cpp file using the command setCpuFrequencyMhz [Editor note: use code font for setCpuFrequencyMhz]. This over rules whatever is set in the LilyGoWatch-S3.json and platormio.ini files. Using this command in the code allows me to make it apparent what the CPU frequency is and enables me to change the CPU frequency as the program is running. More on this later.

Lilygo are good enough to publish the schematic of the T-Watch S3 on their GitHub site[9]. According to this schematic, the ESP32-S3 uses a 40MHz crystal to generate the clock for the CPU. Figure 5 shows the relevant segment of the T-Watch S3 schematic supplied by Lilygo.

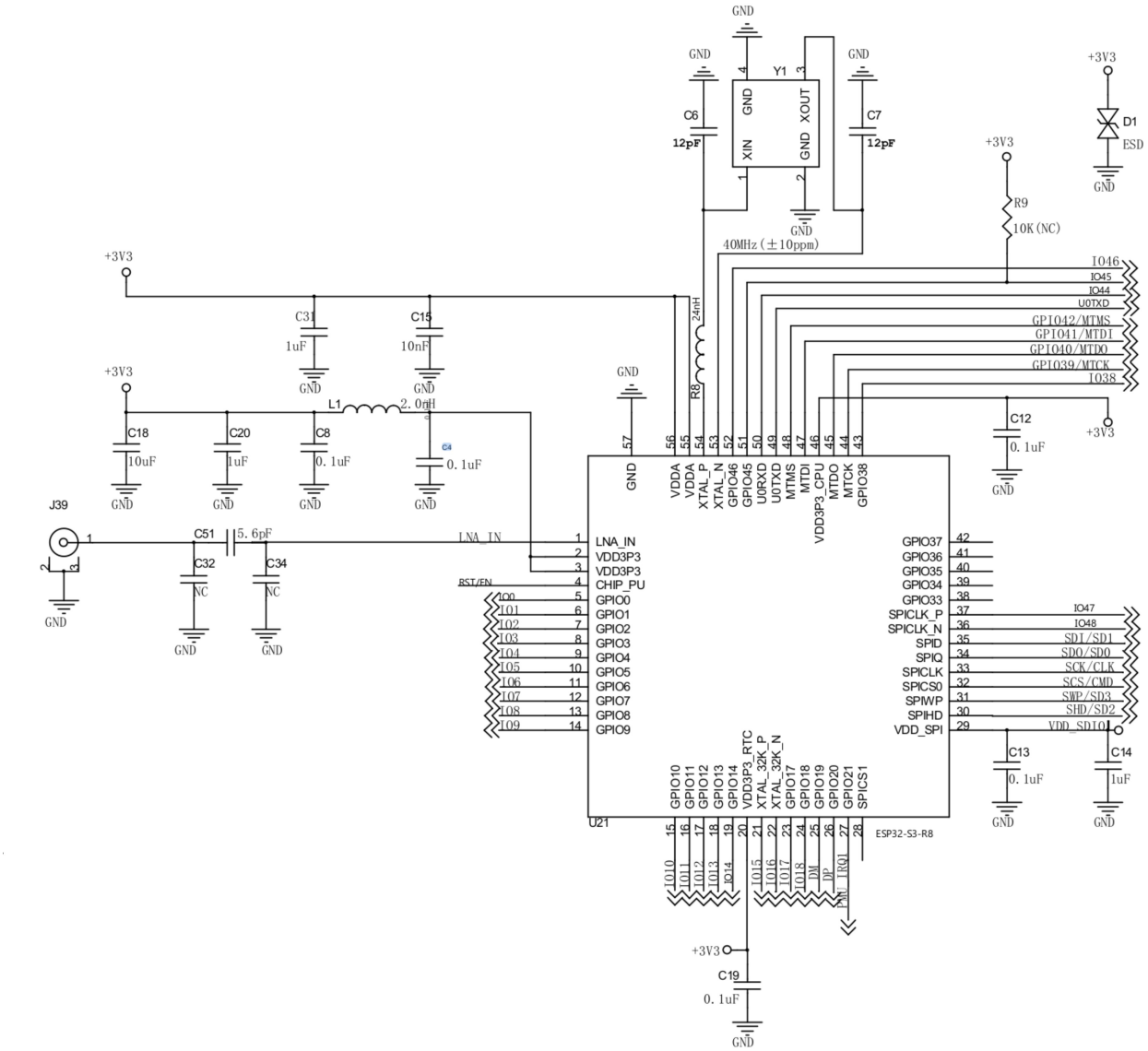

Figure 5: Schematic showing the ESP32-S3 and 40 MHz crystal in the T-Watch S3, from the Lilygo GitHub

Time to crack out the ESP32-S3 Technical Reference Manual[10] (version 1.4). At a mere 1505 pages this is light bedtime reading. On page 525 I found the critical sentence “Wi-Fi and Bluetooth LE can work only when CPU_CLK uses PLL_CLK as its clock source”. CPU_CLK is the clock signal used by the CPU. PLL_CLK is the clock signal generated by using the 40MHz signal from the external crystal with a phase-locked loop (PLL) inside of the ESP32. There are other potential clock sources for the CPU but only PLL_CLK can be used if you want to use the Wi-Fi/Bluetooth radio module. More information on how to set the clock can be found in the online api-reference[11].

I use ESP-NOW as the radio communication method between the T-Watch S3 and the T-Embed module. This leverages the Wi-Fi/Bluetooth module, so has the same clock restrictions. According to table 7-3 in the same esteemed tome of technical knowledge, the only CPU frequencies that are allowed to enable the radio module that provides Wi-Fi/Bluetooth/ESP-NOW are 240/160/80MHz.

I ran some simple tests with the CPU frequency set to 240 MHz, 160 MHz and 80 MHz. Table 1 shows the results. The values are accurate to +/- 0.1 mA. I took the average current values from the PPK2 GUI display. In some ways I was disappointed that reducing the CPU frequency by a factor of four only reduced the current by 10.3 mA. Still, this is about a 9% reduction in current use. I take what I can get in this game. According to the manual, setting the CPU frequency to an illegal value and using the radio causes the CPU frequency to be set to 240 MHz. I got this behaviour when I tried setting the CPU frequency to 16 MHz. The clock frequency was instead set to 240 MHz. However, the T-Watch S3 continuously reset when I tried setting the CPU frequency to 40 MHz rather than set the clock frequency to 240 MHz.

Table 1: Handshake current draws for different CPU frequencies with the PPK2 supplying 3700mV.

CPU frequency (MHz)	**Average current (mA) supply voltage=3700mV**
240	116.8
160	109.4
80	106.5

Turn off the radio module

Looking at the ESP32-S3 Series Datasheet [12] (I used Version 1.9), Table 5-7, the current draw during transmit is ~285 mA. This is a lot of current for a battery operated device! To enable a usable battery life, I need to run the T-Watch S3 with the radio turned off as much as practical. The radio module can be turned off using this code:

```
WiFi.disconnect(true);
WiFi.mode(WIFI_OFF);
```

[Editor note: use code font for WiFi.disconnect(true) and WiFimode(WIFI_OFF)].

I tried turning the Wi-Fi on and off and displayed the current. Figure 6 shows the current draw by the T-Watch S3 when the Wi-Fi module goes from being off to being on. By zooming in on the on and off states I found that the average current draw with the Wi-Fi module off was 66.4 mA. A further reduction of 40.1 mA. I run with the radio module coming on for 10% of the time to catch any messages from the T-Embed module. I use the T-Embed module to change the sensitivity of the T-Watch S3 remotely then send an updated value to the T-Watch S3 over ESP-NOW so that I don't have to interfere with whoever is wearing the watch. However, 66.4 mA is still a large current draw for a battery powered embedded device. What else can I do to reduce the current?

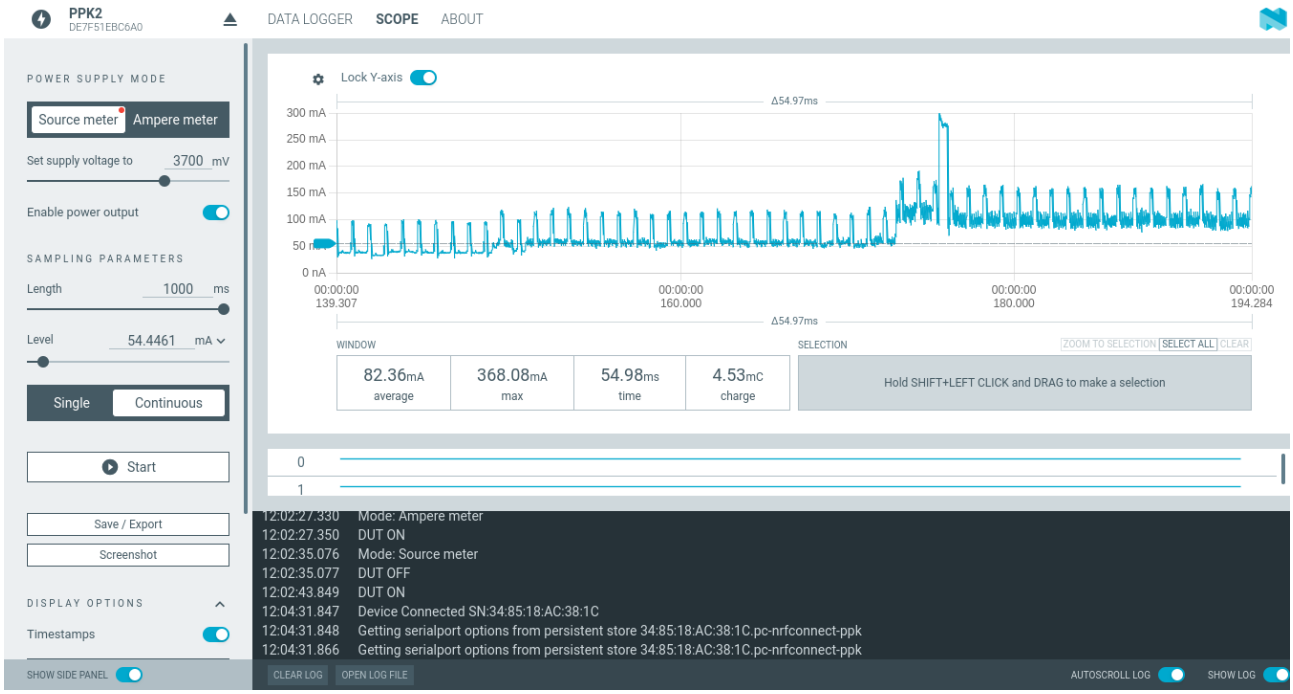

Figure 6: PPK2 display for the T-Watch S3 showing the Wi-Fi module turning on

The minimum that the CPU can be clocked at with the Wi-Fi module active is 80 MHz but with the Wi-Fi module turned off I can further reduce the CPU frequency. With the 40 MHz crystal as the clock source, how low can I set the CPU clock frequency? After some digging, I found the answer in the file esp32-hal-cpu.h. This has the function prototype for setCpuFrequencyMhz which is the function used to set the CPU frequency. According to the comments in this file, with the 40 MHz external crystal, I can set the CPU frequency to be 40, 20 or 10 MHz. I'm still trying to find why these values are the only ones that are valid in the Technical manual. A few more nights of light reading may uncover the answer. So, I set the CPU frequency to be 10 MHz as this is reported to be the lowest frequency available with the hardware setup that I am using.

At 10 MHz, the Wi-Fi module will not work, but everything else should. However, the T-Watch S3 started to randomly reset when I turned the CPU frequency down! What's going on?

Initially, I was turning the Wi-Fi module on and off using timers. I convinced myself that maybe one of the timer interrupts was being repurposed by one of the libraries I was using so re-wrote the code without timers, implementing a simple state-machine approach. Guess what? Same problem.

This is when the PPK2 entered the fray as a debugging tool. Looking at the current draw I could see that the Wi-Fi module was indeed turning off and on as it should, but only for around 5-7 cycles, then the T-Watch S3 would reset itself. The PPK2 display clearly showed when the device was resetting. However, the number of cycles that the Wi-Fi module was active for before the T-watch S3 reset varied. I knew from the PPK2 display that the functions I wrote to turn the Wi-Fi module on and off, which I called esp_now_start [Editor note: use code font for esp_now_start] and esp_now_stop [Editor note: use code font for esp_now_stop] were running. See Listing 1 for what I now have for these methods. Listing 2 shows the original incorrect code for esp_now_stop()[Editor note: use code font for esp_now_stop]. This probably gives you a clue as to where I found the error. I got esp_now_start() [Editor note: use code font for esp_now_start] working correctly, then did a copy and paste for esp_now_stop()[Editor note: use code font for esp_now_stop], deleting the lines I didn't need and editing the ones I did. But I didn't change the order of the operations. Can you see what's wrong?

It is essential to turn the Wi-Fi module off **before** you lower the CPU frequency. Similarly, increase the CPU frequency before turning the Wi-Fi module on. The order of operation swaps between turning the Wi-Fi module on and off. Obvious in hind sight. Most errors are.

```
// CPU frequency with the radio enabled, minimum available
is 80MHz, max is 240MHz
#define CPU_FREQUENCY_MHZ_RADIO 80
// CPU frequency with radio off, minimum available using
external crystal is 10MHz
#define CPU_FREQUENCY_MHZ_NO_RADIO 10

// Turn off ESP-NOW
void esp_now_stop(){
  WiFi.disconnect(true);
  WiFi.mode(WIFI_OFF);
  // turn down CPU frequency to save power
  setCpuFrequencyMhz(CPU_FREQUENCY_MHZ_NO_RADIO);
}

// Start ESP-NOW & configure callbacks
// ondata_recv is a function pointer to the OnDataRecv
function called when data is received
void esp_now_start(uint8_t receiver_mac[],
void(*ondata_recv)(const uint8_t* , const uint8_t*, int),
esp_now_peer_info_t peerInfo){
  // ensure that the CPU frequency is at least 80MHz for
the radio to work
  setCpuFrequencyMhz(CPU_FREQUENCY_MHZ_RADIO);
  // setup wifi
  WiFi.disconnect(false);
  WiFi.mode(WIFI_MODE_STA);
  // check that the ESP-NOW is initialised
  if  (esp_now_init() != ESP_OK) {
    LOG_ERROR("Error initializing ESP-NOW");
    return;
  }
```

```
  // Register the send callback function to call when data
is sent
  esp_now_register_send_cb(OnDataSent);

  // Register the receive callback function to call when
data is received
  esp_now_register_recv_cb(ondata_recv);

  // Register peer
  memcpy(peerInfo.peer_addr, receiver_mac, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  // ****** ALWAYS THROWS AN ERROR
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    LOG_DEBUG("Failed to connect to peer\r");
    return;
  }
}
```

Listing 1, correct code to stop and start the Wi-Fi module

```
// Turn off ESP-NOW
void esp_now_stop(){
  // turn down CPU frequency to save power
  setCpuFrequencyMhz(CPU_FREQUENCY_MHZ_NO_RADIO);
  WiFi.disconnect(true);
  WiFi.mode(WIFI_OFF);
}
```

Listing 2, incorrect code to stop the Wi-Fi module

Now I had the T-Watch S3 running as intended with the CPU clock at 10 MHz when the Wi-Fi module is turned off and at 80 MHz when the Wi-Fi module is turned on. With code turning the Wi-Fi module on for 0.5 s, then running with the Wi-Fi module turned off for 5 s, the average current draw is 31.19 mA. When the Wi-Fi module is turned off, the current draw averages 21.68 mA.

I tried a couple of other things to further reduce the current use.

A few more ideas

The display intensity can be varied using the watch.setBrightness(SCREEN_BRIGHTNESS) [Editor note: use code font for watch.setBrightness(SCREEN_BRIGHTNESS)] command where SCREEN_BRIGHTNESS is an integer with a value of 0 (screen off) to 9 (maximum brightness). In testing I found there was only a 0.6 mA increase in current going from fully off to fully bright on the display. I expected more of a difference but that’s what I measured. I tried turning the screen backlight off using watch.disableALD02 [Editor note: use code font for watch.disableALD02]. The current dropped by about 2 mA. A small gain. I found that having the screen turning on and off was distracting and made it look like the T-Watch S3 was having a hardware fit. I’ll spend the extra 2 mA to keep the display active all of the time.

The accelerometer data sheet claims that the accelerometer draws 150 uA when powered up in ‘performance mode’. I didn’t see any point in turning this off when not in use as the gain in current is so low.

I found that whenever I implemented any type of sleep mode on the ESP32-S3 to try and reduce current consumption that the T-Watch S3 needed a hard reset, which involves turning off the T-Watch S3 using the side button then using the master reset button on the circuit board. This requires opening the T-Watch S3 case and finding a pair of needle nosed tweezers or equivalent to press the tiny reset button.

Discussion

By using the PPK2 to monitor current draw on the T-Watch S3 as I changed the firmware, I extended the watch battery life from a handful of minutes to a handful of hours. This is sufficient to use the T-Watch S3 for testing with the target user group to see if the T-Watch S3 is of use as assistive technology. If the testing is successful, I have ideas on how to further extend the battery life.

I’m porting the firmware from the Arduino framework to the ESP-IDF framework to enable me to access finer grained controls and implement my own scheduler as well as try out FreeRTOS. This may enable me to use low power sleep modes without the T-Watch S3 needing a reset afterwards.

I may be able to use the external 32.7kHz crystal to clock the CPU when the Wi-Fi module is off.

The ESP32-S3 has a third ultra low-power core that stays awake even when the device is in hibernation. Perhaps I can leverage this when the T-Watch S3 is in a hibernation mode to monitor the accelerometer. This core is programmed using a small set of assembly like instructions. If I’m successful, I’ll be sure to include details in a future article.

Please find further details on my handshake project on my website[13].

Resources

[1] https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2
[2] Circuit Cellar issue 344 p48-51, Side-Channel Power Analysis by Colin O’Flynn
[3] https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop/Download#infotabs
[4] https://www.segger.com/downloads/jlink
[5] https://mattoppenheim.com/projects/2018/02/handshake/
[6] https://lilygo.cc/products/t-watch-s3
[7] https://lilygo.cc/products/t-embed
[8] https://youtu.be/LRmNCu1pQ6I?si=-g8biW_ReXGueP4M
[9] https://github.com/Xinyuan-LilyGO/TTGO_TWatch_Library/tree/t-watch-s3
[10] https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

[11] https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/system/power_management.html
[12] https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf
[13] https://mattoppenheim.com/handshake/

Author Profile

Matthew Oppenheim works in marine geophysical survey and offshore construction. He loiters at InfoLab21, Lancaster University when not working at sea. At InfoLab21, Matthew works on assistive technology projects and how to deploy them into the Real World. Visit www.mattoppenheim.com for more details of these projects. Please email matt@mattoppenheim.com with any comments.