ECE:4880 **Lab 1 Report** Shubhresh Jha
2022-10-03 Evan Hagen
Oliver Emery

I. Design Documentation

The Ninecent Thermal Monitoring System (TMS) consists of an 802.11b/g/n-enabled sensor unit (handset), and an on-premises or cloud-deployed webserver. The handset communicates with the webserver to log chronological temperature data to a database, which is presented to the user via a reactive web frontend.

The handset consists of a 4-digit 7-segment display and a detachable sensor cable. On startup, the handset connects to a configured wireless network and begins to update the webserver every 0.8 seconds with the temperature read from the sensor. A momentary pushbutton enables the handset display for local real-time temperature readout.

As the webserver receives updates from the handset, it logs the temperature and current time to a database. On page load, the web frontend requests the last hour of data from the webserver, then proceeds to display a scrolling real-time plot of temperature against time. Controls on the frontend allow remotely enabling or disabling the handset display.

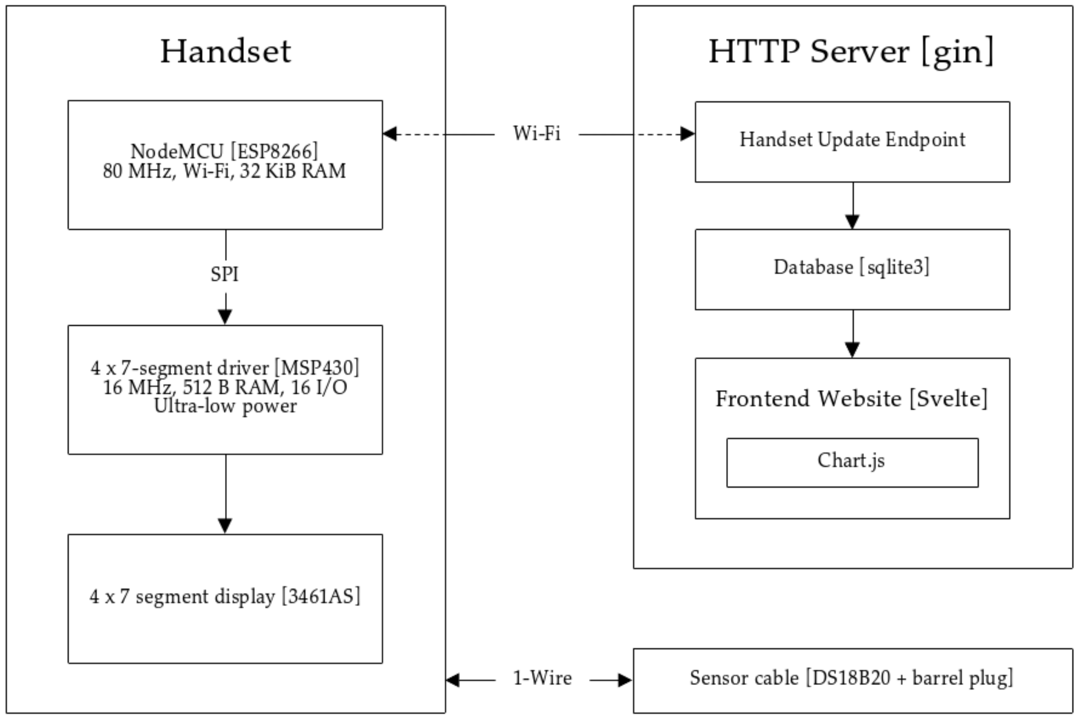

Handset
NodeMCU [ESP8266] 80 MHz, Wi-Fi, 32 KiB RAM
SPI
4 x 7-segment driver [MSP430] 16 MHz, 512 B RAM, 16 I/O Ultra-low power
4 x 7 segment display [3461AS]
Wi-Fi
HTTP Server [gin]
Handset Update Endpoint
Database [sqlite3]
Frontend Website [Svelte]
Chart.js
1-Wire
Sensor cable [DS18B20 + barrel plug]

Fig. 1 – System-level block diagram

Hardware

The handset is 120mm long, 65mm wide, and 22mm thick. Power is supplied via a standard 9V alkaline battery. A side-mounted switch controls power to the entire handset. The sensor cable connects via a simple and sturdy barrel interface. The top-mounted momentary pushbutton activates the mounted display while depressed.

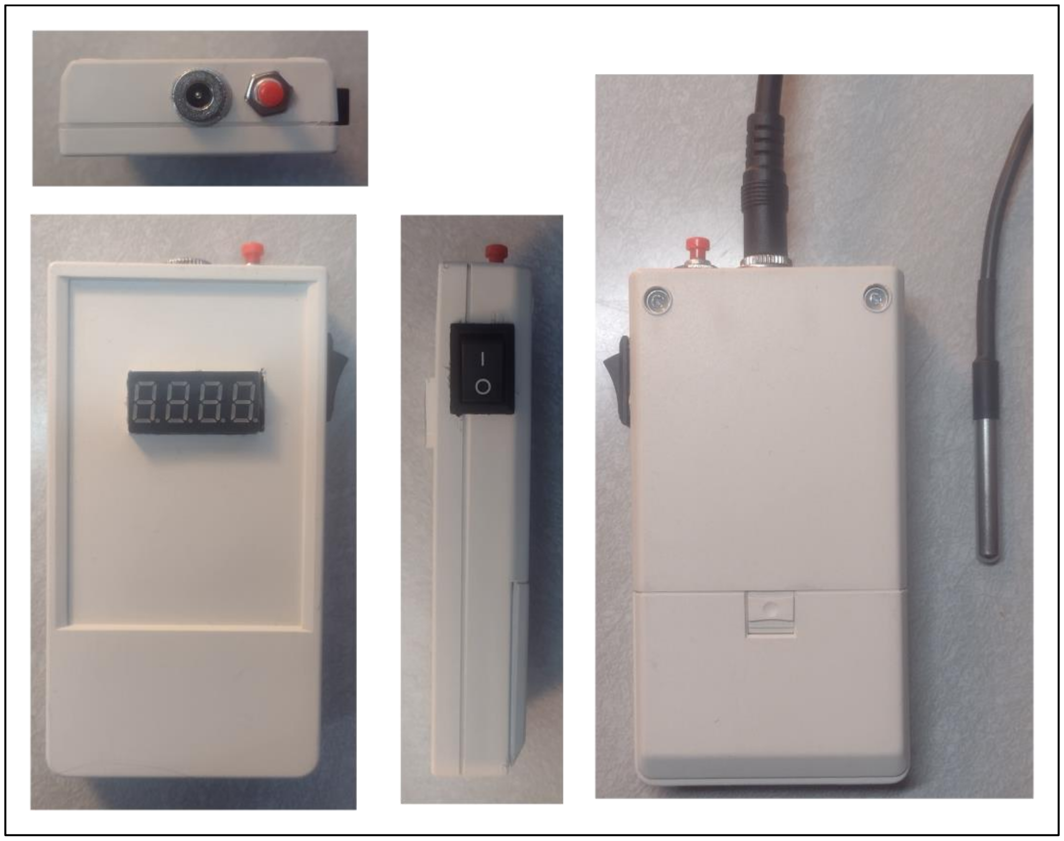


Fig. 2 – Handset exterior

The DS18B20 in the TMS sensor cable is configured in parasitic power mode, which allows delivering power to the sensor via the data line. This enables the use of a barrel jack which only provides two lines. With only one device on the bus and the internal pull-up resistors of the NodeMCU, the temperature sensor is stable even without the recommended external pull-up.

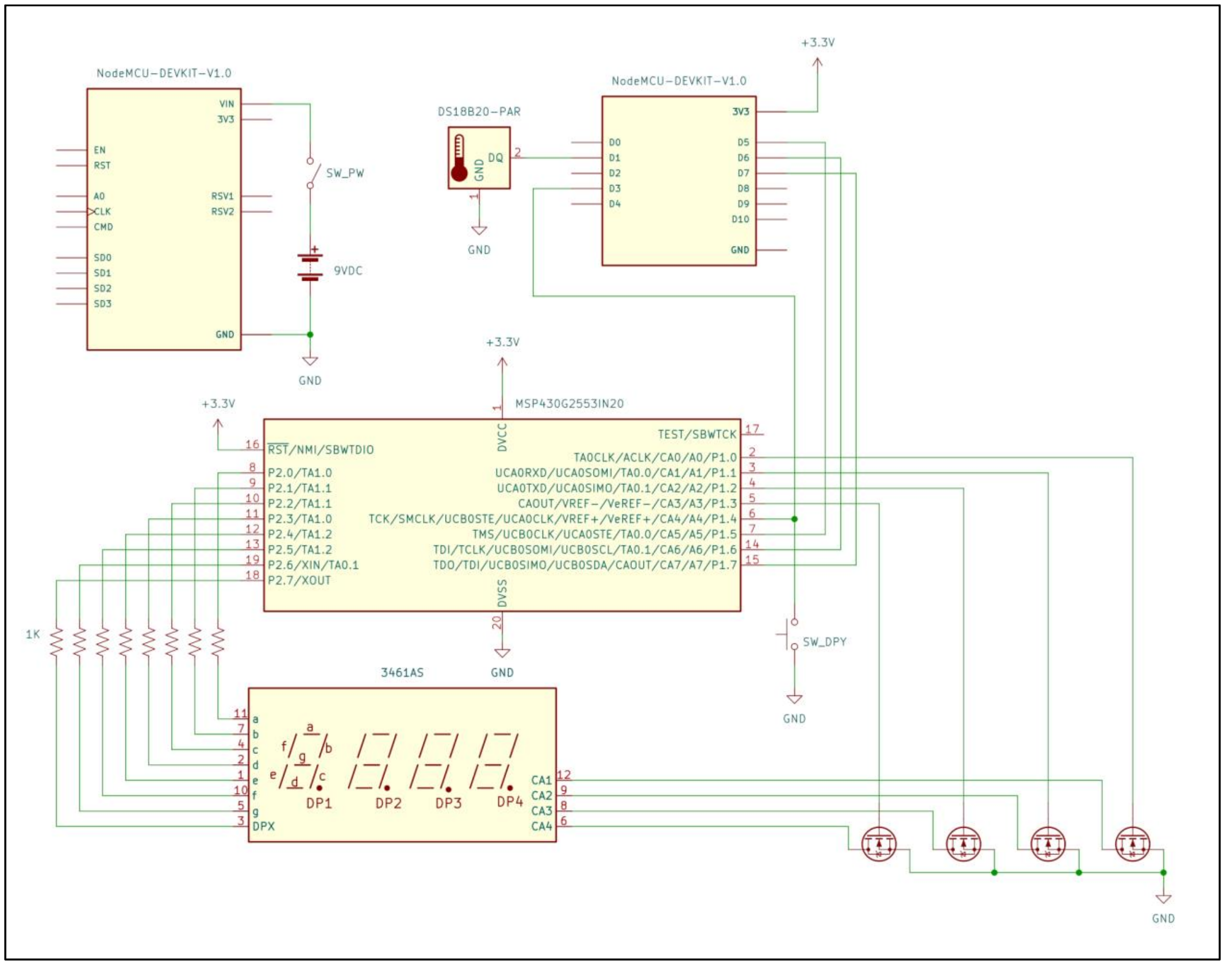


Fig. 3 – Hardware schematic

The NodeMCU has an onboard voltage regulator that produces 3.3 volts with an input voltage of up to 20 volts. The output of this regulator is used to power the display driver and display. The display driver (MSP430) cannot sink enough current to drive the display's digits. Thus, 4 NMOS transistors are used to switch the digit connections to ground.

Qty	Part #	Description
1	NodeMCU-DEVKIT-V1	Wi-Fi enabled 32-bit MCU
1	MSP430G2553	ultra-low power 16-bit MCU
1	DSB1820	temperature sensor
1	3461AS	4 x 7-segment display
4	ZVNL110A	enhancement-mode NMOS
8	R1K	1K resistor
1		5.5 mm barrel plug
1		5.5 mm barrel jack
1		SPST switch
1		SPST momentary pushbutton
1		9V battery connector

Fig. 4 – Bill of materials

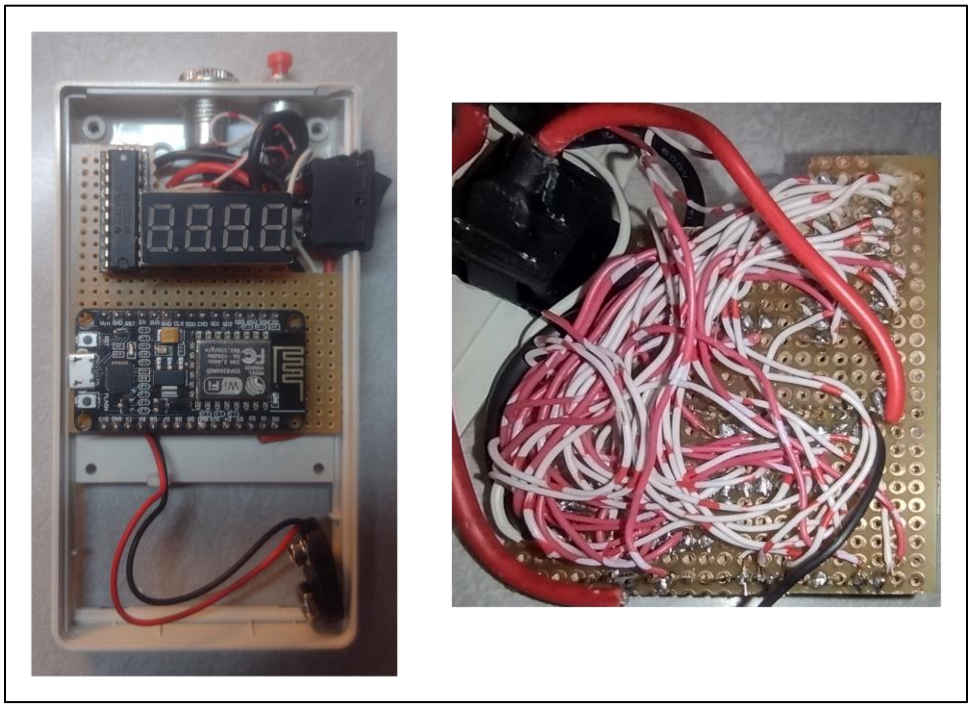


Fig. 5 – Handset internals

Resistors, transistors, and the NodeMCU are soldered directly to the protoboard. The display and its driver slot into sockets to enable replacement and firmware updates. Due to the depth of the NodeMCU, a socket is not viable, but the on-board micro-USB port allows for in-system reprogramming.

Firmware

NodeMCU

With standard firmware, the NodeMCU operates with an event-driven model, and is programmed in Lua. Many essential services (Wi-Fi, TCP/IP, HTTP) are abstracted and simplified to enable rapid development.

At power-on, the NodeMCU is programmed to connect to a configured wireless network (WPA2-PSK) with credentials loaded from flash. After a successful authentication and a DHCP lease is secured, the NodeMCU begins its main loop.

Every 0.8 seconds, the DS18B20 is queried for the current temperature. If no 1-wire devices are detected on the bus (i.e., the sensor cable is unplugged), the NodeMCU registers this state. After querying the sensor, the temperature is converted to decimal digits and sent via SPI to the display driver, and the temperature or error state is sent via HTTP POST request to a webserver endpoint. The status code of the response indicates whether the handset display should be enabled or not. Temperature readout is also available over a USB serial connection.

Display Driver

The MSP430 has no packaged firmware; C code is compiled then run directly by the MCU.

The driver is configured as an SPI slave to the NodeMCU. When the NodeMCU sends the 4 display digits padded with a 1-byte header and trailer, the driver waits until the end of the transmission then updates a 4-byte display buffer.

8 IO pins directly control the individual segments of the display. 4 additional IO pins are dedicated to controlling the currently lit digit. A final IO pin connected to both the NodeMCU and the pushbutton enables or disables the display. A timer interrupt scheduled to trigger approximately every 4 milliseconds advances in a loop through each digit of the display to display the contents of the display buffer. To the human eye, all digits appear to be lit at once.

Software

Web Server

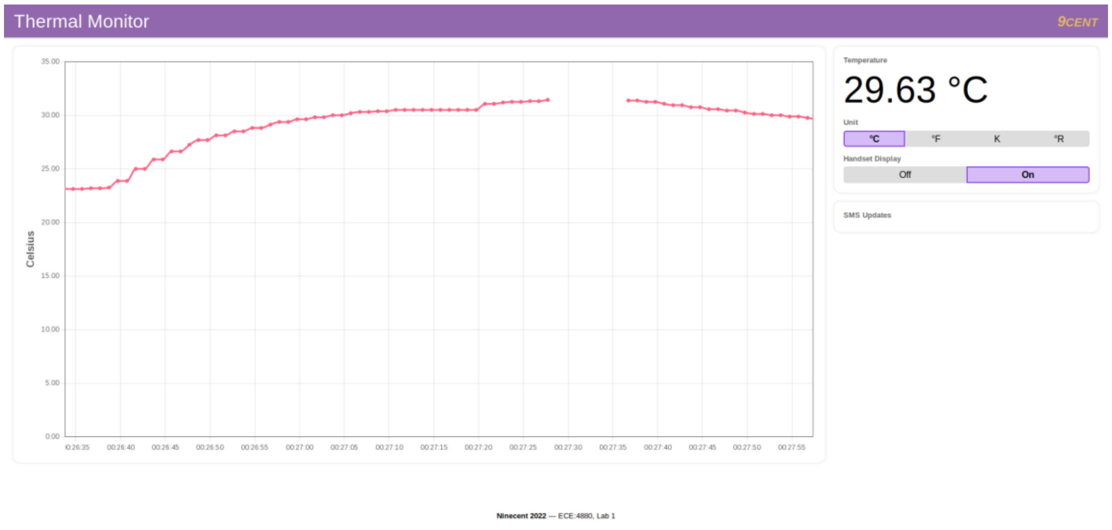
The webserver backend is written in Go using the gin web framework. A static endpoint hosts the frontend of the site.

An endpoint for the NodeMCU updates a sqlite3 database when POSTed with the current handset temperature readout. The status code of the server's response to this POST request controls the handset display power.

Two endpoints are exposed for the frontend, one to fetch the current temperature reading, and another to enable or disable the handset display.

Frontend User Interface

The webserver frontend is written in Svelte, and uses Chart.js to plot the temperature in real-time. On page load, the temperature readings for the past hour are loaded from the database via the backend and are used to prepopulate the plot. Every second, the current temperature is requested from the backend and added to the plot with the current timestamp. A user can pan and zoom the plot with a mouse. Straightforward buttons control the unit of the temperature displayed and the handset display.



Temperature: 29.63 °C

Unit: °C

Handset Display: On

Fig. 6 – Web frontend

II. Design Process and Experimentation

Many components of the final design were selected primarily based on availability as well as group members' previous experience. Nevertheless, we compared and tried several options for critical components such as the microcontroller. Our design group already had several of the project boxes pictured above, as well as switch and display hardware.

The size of our project box influenced all of our hardware choices.

Display

We compared a 4 x 7-segment display to a 2 x 20-character LCD. With the digital interface of the LCD and a plethora of options to configure it, implementation seemed as it would be much more complex than that of the 7-segment display. This turned out to be a moot point, as the LCD would have taken up nearly the entirety of our enclosure.

We selected the 3461AS display for its simplicity and small footprint.

Microcontroller Selection

Although the design specification did not specify a wireless handset, we envisioned ours as wireless since it was to be battery-powered anyway. Even so, we examined the tradeoffs of non-wireless solutions.

	I/O	**RAM**	**Clock**	**Size (mm)**	**Wi-Fi**	**Cost**
Arduino UNO R3	13	2 KB	16 MHz	70 x 50	No	$10.00
MSP430G2553	16	512 B	16 MHz	30 x 15	No	$0.50
ESP8266 (NodeMCU)	17	32 KB	80 MHz	50 x 25	Yes	$3.00
ESP32 (NodeMCU)	24	320 KB	160 MHz	50 x 25	Yes	$6.00
Raspberry Pi Pico W	23	264 KB	133 MHz	50 x 25	Yes	$14.00

Not tabulated above, both NodeMCU boards and the Pico W also include firmware that greatly simplify the development process.

We initially planned to drive the display with a single MCU, so we knew we needed at least 12 IO pins for the display and an additional pin for the temperature sensor. Upon researching the NodeMCU (ESP8266), we learned that a library for the DS18B20 already existed. As the cheapest wireless option with plenty of I/O and great usability, we selected the ESP8266-based NodeMCU.

Display Driver

We soon determined that it was infeasible to drive the display directly with the NodeMCU without a significant amount of effort. With a single core primarily dedicated to networking, we were plagued with out-of-memory errors. The lack of usable hardware interrupts and the alternative, event-driven approach taken by the NodeMCU firmware prevented smooth operation of the display.

We first tried a 74HC595 shift register, as we had previously programmed an ATMega to drive the same display in the same manner in Embedded Systems. This also failed; while individual digits were easy to produce, the timing for alternating digits was inconsistent and resulted in an unusable flicker.

Next, we pivoted to the MSP430 we initially considered as a primary MCU option. With hardware timers and SPI, driving the display with it would require very little power or additional code. Though we experienced several SPI headaches, we ultimately achieved a smooth display output, with temperatures provided to the MSP430 from the NodeMCU via SPI.

Software Stack

As none of the group was terribly familiar with web development, we researched several reactive web frameworks to implement our frontend with. The top contenders were React and Svelte. React is a much older framework with hundreds of thousands StackExchange questions and wide community support. On the other hand, Svelte is much newer, and requires much less boilerplate code.

Since the group was a relatively blank slate and sans biases with regard to web design, we selected Svelte for ease of use.

Some group members had previous experience with web servers. We considered Node.js and Go. While Node.js is well-documented and supported, we determined that it would exhibit an unnecessarily large memory and storage footprint for such a simple server. A server written in Go would compile to native code and involve many less software components and libraries behind the scenes.

III. Project Retrospective

Ninecent's final product for Lab 1 might be considered a "semi-success." While completely ill-prepared for the checkoff of 29 September, by 30 September we had a working, up-to-spec solution, with the exception of SMS notifications. Our hardware design resulted in a compact, sturdy handset with a no-nonsense display and comfortable ergonomics, and our web design efforts produced a beautiful responsive single-page web application.

However, many aspects of our design were rushed or not carefully thought out due to our general lack of time and project management. We met at least once per week, but did not use any framework to organize our efforts. The MSP430 display driver was a surprisingly successful last-minute addition, but one that, without our previous experience, may have ended in further failure.

For example, within the context of our codebase, integrating SMS notifications should have been a simple 1.5 hour task. Twilio, a provider of an SMS API, publishes libraries for all major languages (including Go). Registering an account, importing the library, and adding 20-30 lines of code is not a daunting task. What we lacked was a centralized "birds-eye view" of our project and its status.

One could describe our project management strategy as "event-driven," as our work efforts surged when contacted by Master's Student McIntyre to schedule check-offs. Hours prior to the checkoff, our initial protoboard met an extremely untimely fate in a face-off with a fully-heated soldering iron. Of course, had we planned our time wisely, there would not have been a fully-heated soldering iron anywhere near our prototype within hours of the checkoff.

In wake of the litany of mishaps and procrastinations, Ninecent is fully committed to adopting a more structured organizational approach to the next lab assignment. Next time, we will design and experiment first, then break the overall project into discrete assignable tasks that can be completed by one or two group members.

IV. Test Report

Test	**Expected Outcome**	**Result**
Sensor		
Measure sensor cable length	2.0m ± 0.1m	Pass
Place sensor in ice water for 10s	Functional sensor	Pass
Handset		
Operate without external power	Handset functions normally	Pass
Connect and disconnect sensor	User encounters no issues	Pass
Drop from 3 feet	Handset functions normally	Pass
Depress pushbutton	Display activates for duration of press	Pass
Close power switch	Handset powers up and begins normal operation	Pass
Open power switch	Handset powers off and ceases communication	Pass
Error condition indication with unplugged sensor	Display shows "Err" when activated	Pass
Web Interface		
Load site	Plot is populated with last 300s of sensor data, if it exists	Pass
Operate handset display control buttons	Handset display state synchronizes to control state within 1s	Pass
Observe plot and readout	Plot scrolls to left and displays temperature readout in real-time	Pass
Resize plot	Plot adjusts to desired size	Fail
Configure and enable SMS updates	SMS received when configured limits are exceeded	Fail
Disconnect sensor	Plot scrolls with no data and "Sensor Unplugged" message is displayed	Pass
Change unit	Temperature readout value converted properly	Pass

V. Appendix & References

Datasheets

- NodeMCU development board (https://github.com/nodemcu/nodemcu-devkit-v1.0)
- NodeMCU firmware (https://nodemcu.readthedocs.io/en/release/)
- 3461AS display (http://www.xlitx.com/datasheet/3461AS.pdf)
- MSP430G2553 (https://www.ti.com/product/MSP430G2553)
- ZVNL110A (https://www.diodes.com/assets/Datasheets/ZVNL110A.pdf)

Documentation

- Gin (https://gin-gonic.com/docs/)
- Svelte (https://svelte.dev/docs)

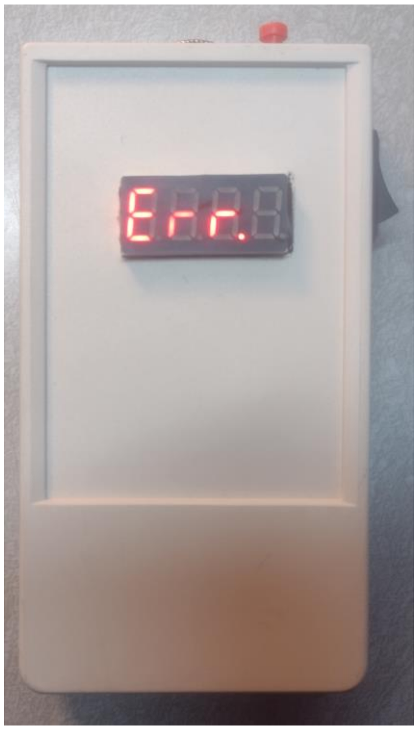


Fig. 7 – Handset with enabled display and unplugged sensor