The ESP32 S3 Webserver Project

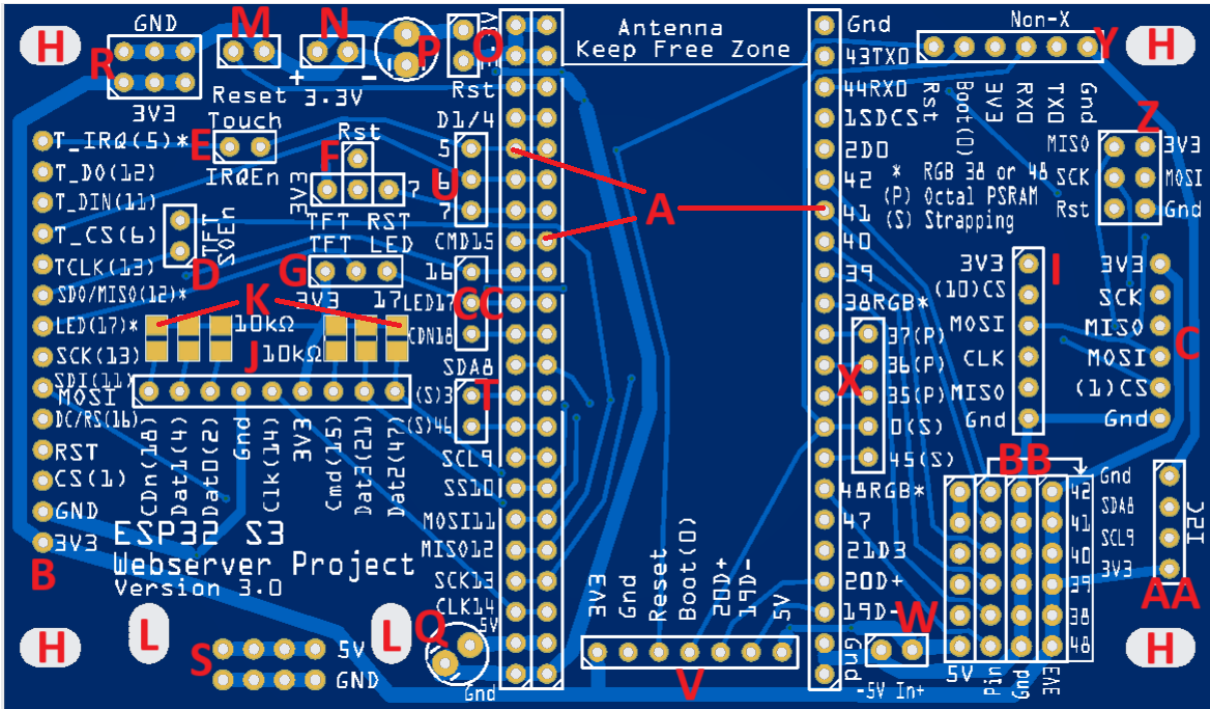
The ESP32 S3 Webserver Project was designed to provide a flexible base for building various web-enabled micro-controller projects on. Using the powerful ESP32 S3, it offers 240MHz and 320KB RAM. Depending on the ESP32 S3 module used, 4-32MB of Flash is available. The ESP32 S3 offers built-in WiFi and Bluetooth support. The asynchronous library used allows up to 8 WiFi clients to be connected simultaneously. It supports HTML and WebSockets.

The ESP32 S3 Webserver Project is divided into hardware and software sections which can each be used independently or combined to provide the synergy to get a project up and running quickly. The project provides the necessary infrastructure for a wide range of simple to complex IOT projects. Using **The ESP32 S3 Webserver Project** allows you to concentrate on the individual aspects of your project without having to worry about the web server infrastructure. **The ESP32 S3 Webserver Project** hardware can be used for rapid prototyping as well as in finished projects without the need of building the hardware twice.

While not absolutely necessary to use the software, **The ESP32 S3 Webserver Project** hardware makes it easier by providing breakouts and connectors for many things. The board has been designed as a standalone or to attach to the back of a 3.5 inch TFT Display, with or without a touchscreen. The board provides connectors for 5V in/out, 3.3V in/out, I2C, SPI SD/SDHC card (or connect to one already on the TFT display), SD/SDXC MMC card, TX/RX Port, USB Port, reset button, and boot button. A standard SPI port is also provided for additional SPI devices.

The ESP32 S3 Webserver Project software supports Over the Air (OTA) flashing, SD and/or LittleFS (file storage in flash memory), connectivity as an access point and WiFi, client-side webpage caching, a local timezone including daylight savings time where appropriate, time synching over the internet, an Admin, File Manager and Status webpage, Serial Monitor support over a webpage, WebSockets and sending/receiving Telegram messages.

The ESP32 S3 Webserver Project Hardware

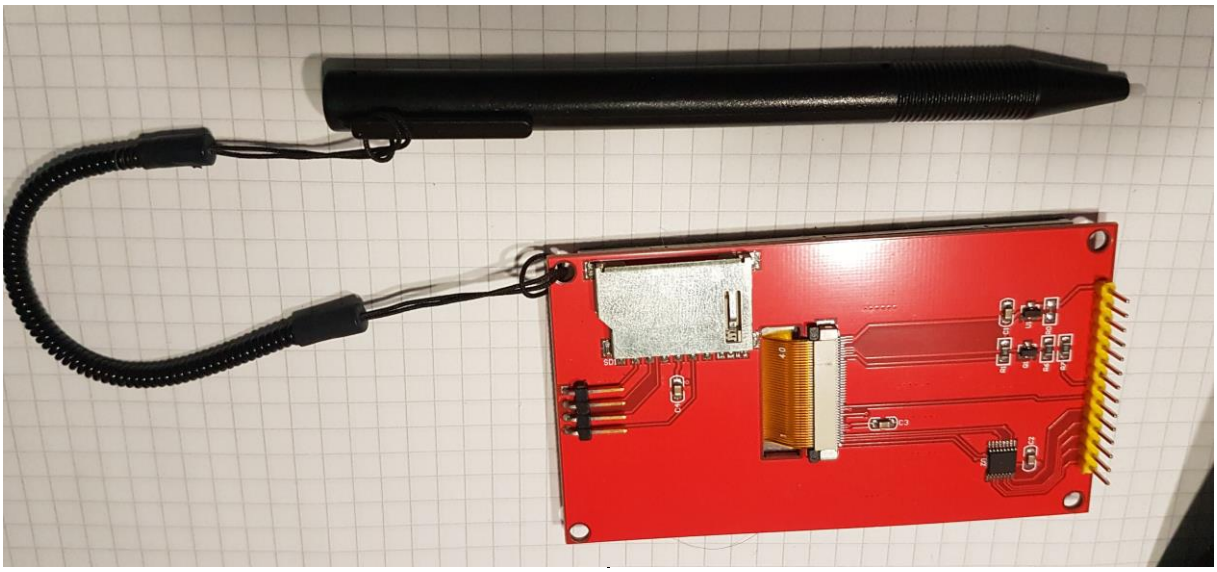


The ESP32 S3 Webserver Hardware PCB board

Note that all connectors and components other than two of the three A connectors are optional and can be mounted or left off according to the needs of your project and how you will be providing the required DC voltage.

A) Connectors for the ESP32 S3 DevKit module. I have seen these modules in two different widths, one being slightly wider than the other which is why there are two sockets on the left side. Use whichever of the two left sockets that fit your module. Some modules have only 20 pins along each side as opposed to the 22 pins shown in the above image. If your module only has 20 pins on each side, it is probably missing the top two pins (an extra 3.3V on the left and Gnd on the right) and the bottom two pins (extra Gnd on the left and right). If that is the case, you can safely insert your module into The **ESP32 S3 Webserver Project** hardware and simply leave each of the four corner pins empty.

B) The hardware has been designed for standalone use or for being attached to the back of a 3.5 inch TFT 480 x 320 display. If you are using such a TFT module this connector provides all of the connections between the ESP32 S3 and the TFT. Some of those modules also include an integrated touchpad. The pins T_IRQ, T_DO, T_DIN, T_CS and TCLK are used to connect the touchpad to the ESP32 S3. NOTE: Beginning with version 3.0, the CS pin for the TFT was moved to 1 to make the hardware more compatible with external examples “out of the box”.



TFT Module, 480 x 320 with Touchpad and SD Card

C) If using a TFT module, some of those modules also include an onboard SD card reader. This connector provides the connection between the ESP32 S3 and that SD card reader. Those TFT modules are, unfortunately, not standardized and appear to be available in two sizes. For one of those sizes, you can use the through hole connector. For others, you may have to use a surface mount connector and the pads on the bottom of **The ESP32 S3 Webserver Project** hardware are elongated to allow them to be soldered accordingly. Typically Gnd and 3V3 aren't available on the TFT boards 4 pin connector but are provided here to facilitate connecting an external module, if required. NOTE: Beginning with version 3.0, the CS pin was moved to 10 to make the hardware more compatible with external examples "out of the box".

D) Some TFT modules support reading from the TFT display memory, others don't. Modules using the ILI9488 chip pose a serious problem in that they don't tri-state the SDO pin (MISO) when the module is not in use (CS is high) which would cause interference for any other devices using the SPI bus. If using a TFT module with the ILI9488, do not bridge this connector and everything will work fine. If your TFT module does support reading from the TFT, you will have to bridge this connector.

E) If you will be using a touchpad on the TFT module, you may choose to use the IRQ pin or not. The IRQ can be used to execute an ISR (Interrupt Service Request) immediately when the TFT is touched. Most libraries have the IRQ as an option. If using the IRQ of a touchpad module, you will need to bridge this connector. GPIO pin 5 of the ESP32 S3 is then dedicated to detecting IRQ from the touchpad.

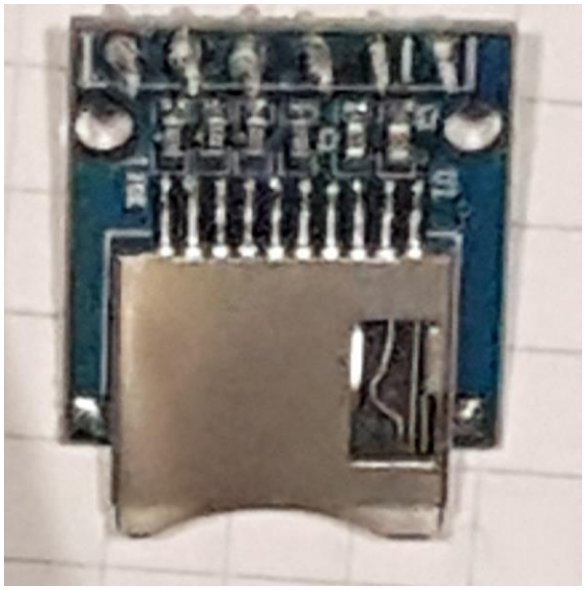
F) If using a TFT module, most support a hardware reset and most libraries will have the option of resetting under the control of the library. Using this connector and a jumper block, you can choose to connect the TFT reset pin either to 3V3, which would disable the hardware reset of the TFT module, or to Rst which would reset the TFT module simultaneously with an ESP32 S3 hardware reset, or to ESP32 S3 pin 7 which would then put the TFT reset under software control.

G) If using a TFT, you may wish to be able to control the brightness of the backlight through the software. This bridge connector allows you to connect the TFT backlight either directly to 3.3V (no dimming) or to pin 17. Note that the TFT does not power the backlight directly through the LED pin

on the connector, but provides an internal switch that is controlled by this pin. Therefore, you can use pin 17 with PWM to dim the TFT backlight under software control.

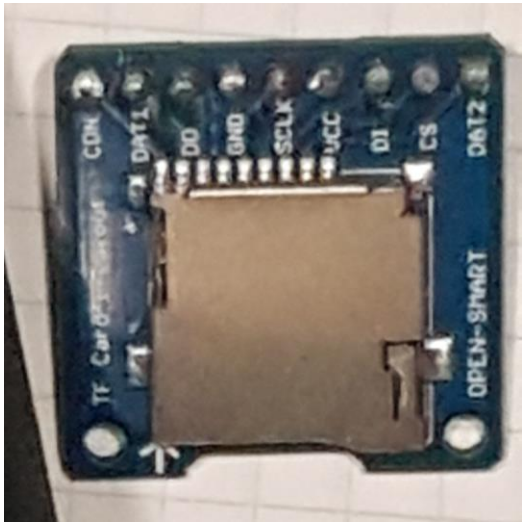
H) Mounting holes. These holes can be used to easily mount the hardware to the back of a TFT module and/or directly to the project case. The holes are elongated to allow for the various sized TFT modules. M2.5 or M3 screws should work fine.

I) If not using a TFT module, or the module doesn't contain an onboard SD, this connector can be used to connect a separate external SD module. NOTE: Beginning with version 3.0, the CS pin was moved to 10 to make the hardware more compatible with external examples "out of the box".



SD Card Module

J) MMC cards are faster and can have a much higher capacity than a standard SD card. This connector supports connecting to a separate MMC SD module.



SD_MMC Card Module

K) The ESP32 S3 datasheet states that the MMC data pins should be externally pulled high with 10K Ohm resistors. In practice, everything seems to work just fine without the pullup resistors. If you

experience trouble using an MMC card, these pullup resistors may help. Note that the CDn pin is for optional chip detection and not all MMC card modules support that feature. If you are using the CDn pin, then GPIO 18 of the ESP32 S3 is dedicated to this function.

L) Mounting holes for the MMC module, when used.

M) A connector for an external reset switch.

N) A connector for providing 3.3V from an external source. If not using an external power source, this connector can provide an additional 3.3V output.

O) An ESP32 S3 DevKit is typically connected to an external 5V supply and uses the onboard converter to supply the required 3.3V. However, this could lead to problems supplying sufficient 3.3V to other parts of your project because the 3.3V is limited to around 500mA and the ESP32 S3 itself can draw half of that while transmitting. If you are using a TFT display, that also requires a significant portion of the available 3.3V. Using an external 3.3V supply of sufficient power would solve that problem, but could lead to another if 5V is also being supplied, for example driving addressable RGB LEDs. This bridge connector solves the problem. If using both a 3.3V and 5V external supplies, do not bridge this connector, otherwise bridge this connector. If left unbridged, the external 5V will supply 3.3V only to the DevKit module and the external 3.3V supplies the rest of the project.

P) An optional capacitor (10µF or larger is recommended) to help stabilize the 3.3V.

Q) An optional capacitor (10µF or larger is recommended) to help stabilize the 5V.

R) A connector which can be used to supply 3.3V and ground to other parts of your project such as sensors.

S) A connector which can be used to supply 5V and ground to other parts of your project IF you are providing the 5V through the external 5V connector.

T) A connector for ESP32 S3 data pins 3 and 46. These pins are strapping pins so don’t use them unless you understand their meaning to the ESP32 S3.

U) A connector for ESP32 S3 data pins 5, 6 and 7. If you are not using these pins for a touchpad (pin 5 is T_IRQ and 6 is T_CS) or the TFT (pin 7 may be the software reset), you can use this connector to provide additional data pins for your project.

V) A connector which can be used for an external USB connector (D+/pin 20 and D-/pin 19) as well as additional pins which may be convenient for use with an external USB connector.

W) A connector for providing 5V from an external source. The ESP32 S3 DevKit module uses this 5V input to provide the 3.3V required for the ESP32 S3.

X) A connector for ESP32 S3 data pins 0, 35, 37, 38 and 45. Pins 0 and 45 are strapping pins; do not use them in your project unless you understand their special meaning on the ESP32 S3. Pins 35, 36 and 37 cannot be used if your ESP32 S3 has an octal PSRAM as they are reserved for that purpose. If your ESP32 S3 does NOT have octal PSRAM, these pins can be used to provide additional data pins for your project. Some modules may have the onboard LED on pin 38.

Y) A connector for a serial port. This connector also includes other pins which may be convenient when using a serial port. Note that RX and TX are NOT connected as crossovers. RX from this connector would be connected to TX on the other side and vice versa.

Z) A standard SPI connector which can be used for additional SPI connectivity if your project requires it.

AA) An I2C connector which can be used to attach I2C devices in your project.

BB) Connectors providing connection to data pins 38, 39, 40, 41, 42 and 48 as well as 3.3V and Gnd. If you are providing an external 5V, it is also available on this connector which may be especially convenient, for example for addressable LEDs. Note that the ESP32 S3 DevKit module has its own addressable LED which may be attached to either pin 38 or 48 depending on your version of the DevKit.

CC) Beginning with version 2.2, **The ESP32 S3 Project** hardware has an additional connector for GPIO pins 16, 17 and 18 which can provide additional I/O pins for your project if these pins are not being used for either the TFT display or chip detection on the MMC connector.

Before assembling the Hardware:

Check the width of your DevKit module. These come in two different widths and that will determine which of the two left connectors (A) you will need. Refer to the description of these connectors earlier in this document.

Decide if you will be using the 480 x 320 TFT display and if you will be using the touchpad. If so, the 14 pin female display/touchpad connector (B) must be soldered onto the bottom of the board. If the TFT display supports MISO, the TFT SOEn connector (D) must be soldered in and/or bridged. The TFT RST (F) should be bridged to 3.3V (disabled) or Rst (reset of ESP32 S3) if you don't want a software reset of the TFT display and to pin 7 if you do. To control the backlight of the display through software, bridge the TFT LED connector (G) to pin 17. Bridge it directly to 3.3V if software dimming isn't required.

Decide if you will be using LittleFS, a SDHC or SDXC card. If using the SDXC card, the board must have the SDXC connector (J) soldered in. Some SDXC modules may not have the chip detect pin. If using the SDHC card built onto the TFT display, a four pin connector (C) must be soldered to the BOTTOM of the board. Refer to the description of the SD connector (C) earlier in this document. If using a SDHC without the TFT display, a 6 pin connector (I) must be soldered to the top of the board and properly connected to a SD module. Here are the results of some speed tests I made for the various options.

	1MB Write	1MB Read	Remark
SD	3670.768 mS	2402.870 mS	1GB SDSC, SPI @ 4 MHz
SD	2575.287 mS	694.873 mS	1GB SDSC, SPI @ 40 MHz
SD	4301.888 mS	1048.576 mS	16GB, SanDisk Ultra SDHC, SPI @ 4 MHz
SD*	2451.307 mS	681.579 mS	16GB, SanDisk Ultra SDHC, SPI @ 40 MHz
MMC	2205.380 mS	326.795 mS	16GB, SanDisk Ultra SDHC, 1 bit mode
MMC*	2027.529 mS	161.659 mS	16GB, SanDisk Ultra SDHC, 4 bit mode
MMC	869.579 mS	347.536 mS	1GB SDSC, 1 bit mode
MMC*	692.572 mS	184.180 mS	1GB SDSC, 4 bit mode

```
LittleFS  7066.146 mS   394.701 mS  Using DIO 80Mhz Flash
LittleFS  6896.746 mS   363.599 mS  Using QIO 80Mhz Flash
LittleFS* 8993.782 mS   342.727 mS  Using OPI Fast Read 80Mhz Flash

* = Best results (Note that MMC has different results for read and write based on
card)
All read/write tests were done using 512 byte blocks
```

If you will be using the I2C (AA), RX/TX (Y), USB port (V), or SPI devices (Z) respective connectors will be required.

If your project is to have an external pushbutton for reset, you will want to solder in connector (M). If you want an external Boot pushbutton, you can tap boot and ground the easiest using the serial connector (Y) but the USB connector (V) can also be used.

The board provides multiple connections for 5V out (S) and 3.3V out (R) with ground. If your project requires multiple connections, be sure to include them. The ESP32S3 can use up to 250mA when transmitting. If you are using a TFT display, that needs power too. Be sure your power supply is sufficient. You should not pull more than about 500mA through the onboard 5V to 3.3V regulator. If you need more power, use the external power inputs provided on **The ESP32 S3 Webserver Project** hardware.

As explained above, D, E, F, G and O are hardware configuration bridges to select various hardware features. Each of these bridges can have pins soldered in and a configuration block used for maximum flexibility of those options. Alternatively, you could also use solder to short the required bridges semi-permanently on the top side of the board.

Plan the internal cables for your project. Where on the case will external connectors, switches or LEDs go? Prevent running cables near the ESP32 S3 on board antenna as this may affect WiFi connectivity and range. It is, in my opinion, better to use header pins and wires that plug onto them, or, even better, screw terminals for attaching wires to the PCB. You could solder wires directly to the PCB but that can make assembly/disassembly harder and the wires can easily break off. Use the mounting holes to mount the PCB inside the case. If a TFT display is used, use spacers between **The ESP32 S3 Webserver Project** PCB and the TFT.

The ESP32 S3 Webserver Project Hardware Pinouts

Pin	ESP32 S3 Standard Use	ESP32 S3 Webserver Project Use
0	Strapping, Boot	
1		SD CS, beginning with version 3.0 TFT CS[1/2]
2	MMC Interface Data0	MMC Connector Data0[3]
3	Strapping	
4	MMC Interface Data1	MMC Connector Data1[3]
5		Touchpad IRQ[4]
6		Touchpad CS[4]
7		TFT Reset[4]
8	I2C SDA	I2C SDA
9	I2C SCL	I2C SCL
10	SPI CS/SS	TFT CS/SS, beginning with version 3.0 SD CS[1/3]
11	SPI MOSI	MOSI for TFT, Touchpad, SD, SPI Connector
12	SPI MISO	MISO for TFT, Touchpad, SD, SPI Connector
13	SPI SCK/CLK	SCK/CLK for TFT, Touchpad, SD, SPI Connector
14	MMC Interface CLK	MMC Connector CLK[3]
15	MMC Interface CMD	MMC Connector CMD[3]
16		TFT DC/RS[4]
17		TFT Backlight Dim Control[4]
18		MMC Connector Chip Detection[3]
19	USB D-	USB Connector D-
20	USB D+	USB Connector D+
21	MMC Interface Data3	MMC Connector Data3[3]
35	Octal PSRAM[5]	Octal PSRAM[5]
36	Octal PSRAM[5]	Octal PSRAM[5]
37	Octal PSRAM[5]	Octal PSRAM[5]
38	RGB (DevKit v.2)	Available on Data Connector / RGB (DevKit v.2)[6]
39		Available on Data Connector
40		Available on Data Connector
41		Available on Data Connector
42		Available on Data Connector
43	Serial Interface TX0	Serial Connector TX
44	Serial Interface RX0	Serial Connector RX
45	Strapping	
47	MMC Interface Data2	MMC Connector Data2[3]
48	RGB (DevKit v.1)	Available on Data Connector / RGB (DevKit v.1)[6]

1) Prior to version 3.0, pin 1 was CS for the SD and 10 was CS for the TFT. In version 3.0 these were reversed which makes the hardware more compatible with example coding found on the internet.
2) SD pins may be available as general purpose IO if no SD is used in the project. (MMC connector J is not a SD in this sense.)
3) MMC pins may be available as general purpose IO if connector J is not used to connect an MMC SD card in the project.
4) TFT Pins may be available as general purpose IO if no Touchpad is used in the project.
5) If the DevKit has octal flash or PSRAM, these pins are used internally for that purpose and should NOT be used in the project.
6) Some DevKits have the onboard addressable RGB LED attached to pin 38, some to 48. Some have a solderpad so that the LED can be removed from the circuit all together.

Peripheral Hardware

SD Card Module: TF Micro SD Card Module Mini SD Card Module Memory Module for Arduino ARM AVR

MMC Card Module: OPEN-SMART Micro SD / TF Card Breakout to DIP Board Module DIY Micro SD / TF Card Adapter Breakout Board Module for Arduino

SD Card Extender (move SD card to panel of case): 10CM 25CM 48CM 60CM SD card Female to TF micro SD Male SD to SD TF to TF Flexible Card Extension cable Extender Adapter reader

TFT Display w/SD & Touch: IPS SPI module 3.5 inch TFT LCD colorful display screen with Resistance touch panel ILI9486 ILI9488 drive IC controller

Screw Terminals: 5/10pcs KF128 2.54mm 0.1" Pitch Mini PCB Screw Terminal Block Connector for Wires 2P 3P 4P 5P 6P 7P 8P 9P 10P 12P 16P Terminal

Pushbuttons: 8mm Momentary Metal Horn Doorbell Bell Push Button Switch Waterproof Car Auto Engine PC Power Start Home appliance

Spacers: M2 M3 M4 M5 Black White Nylon Spacers Hollow Non Threaded Round Standoff Board Rack Washers Insulation Plastic Column Pillars

Panel Mount USB Connector: Type-c 2pin 4pin Panel Mounting Charging Interface Usb-c Connector Jack Soundboxes Charging Port Small Appliances Usb Socket - Connectors

Configuration Jumpers: 100PCS PitchJumper Shorted Cap Headers Wire Housings SHUNT Black Tool Accessories

The ESP32 S3 Webserver Project Software

The **ESP32 S3 Webserver Project** software has been specifically designed to fully utilize **The ESP32 S3 Webserver Project** hardware but can also be used without that specific hardware.

The ESP32 S3 Webserver Project software has been designed to offer various functionalities which may be selected and configured at compile time:

SDHC or SDXC cards. SDHC supports up to 4GB files and 32GB total while SDXC supports up to 2TB of storage. While SDHC can be relatively fast, here up to 40MHz, SDXC is faster yet. If selected, the SDXC used here runs in 4 bit mode which is the fastest mode available today.

Over The Air, or OTA flashing. This feature allows reflashing the software without a physical connection such as a USB cable. It should be noted that OTA will divide the program area of Flash into two sections, so to allow for up to 2MB of program size and use OTA, 4MB of Flash must be allocated. Note that the initial software load will have to be done over a physical connection because OTA is only implemented once OTA enabled software has been initially loaded.

LittleFS allows allocating a portion of the flash to be used as a file system. By doing so, a few MB of files or data can be placed in the flash memory where they can be accessed faster than from an SD card. This is the perfect place to store favicon.ico, small HTML files, etc. which are frequently accessed by the clients as the quick access will help keep the server responsive. Files that change often, such as a log or dynamic data should NOT be placed in flash as it will eventually fail after thousands of writes.

A status web page which shows the hardware configuration, memory usage, etc. can be enabled. This page can be useful for debugging.

An admin web page which allows configuring WiFi credentials, time zone, an internet time server, etc. is provided and is always included. The admin web page is logon and password protected and can optionally be limited only to clients on the local network.

A file manager webpage which can be used to view a list of files on the server as well as upload, download and delete files of any type is provided and can be included. The file manager is logon and password protected. While uploading large files a progress bar is displayed. In tests, a 2.77MB file was uploaded in around eleven seconds with the progress bar being updated about twice per second.

The webserver will be available locally through its' own access point and can also be available over the local WiFi network.

The webserver supports a local timezone, to include automatic adjustment for daylight savings time where appropriate. Synchronizing to a time server over the internet is also supported.

The webserver supports sending and/or receiving Telegram messages using one or more Telegram bots. See Appendix B for more information on Telegram bots and messaging.

Configuration

Configuration takes place at several levels.

Pre-compilation. Make the board settings. Settings are determined by your ESP32 S3 DevKit module and will resemble:

Board: ESP32S3 DevModule
CPU Freq: 240 MHz
Flash Mode: QIO 120MHz (or IAW your ESP 32 Module)
Flash Size: 8MB (or IAW your ESP 32 Module)
PSRAM: OPI PSRAM (or IAW your ESP 32 Module)
Partition Scheme: 8M Flash(2MB App, OTA, 3.5MB LittleFS) (or appropriate settings)

Note that The ESP32 S3 Webserver Project software, with all options selected, is slightly larger than 1MB. An App size of 1.5MB should be sufficient unless your customizations for your project include a large amount of code and/or libraries.

See **Appendix A** for instructions on how to create your own partition scheme, if necessary, to more perfectly match your modules Flash size and your project requirements.

In customize.h:

```
// ------------------------ Admin section ------------------------
```

The values in the Admin section are defaults only. These values will be burned onto the board the first time the sketch is booted. On subsequent boots, the settings will be detected and the values from the board will be used. Typically, these values are maintained by the admin using the admin web page.

To prevent duplicating information, most of these values are documented in more detail under the **Admin Webpage** section later in this document.

```
#define def_HostName      "WebServer"         // Default name of this server and access point
```

Refer to **Host Name** in the **Admin Web Page** section of this document.

```
#define def_APpassword    "TopSecret"         // Default password for this access point
```

Refer to **AP Password** in the **Admin Web Page** section of this document.

```
#define def_Beacon   true  // Default beacon for this access point (false hides AP)
```

Refer to **Broadcast AP** in the **Admin Web Page** section of this document.

```
#define def_SSID          ""                  // Local WiFi
#define def_Password      ""                  // Password for local WiFi
```

Refer to **WiFiName** and **WiFi Password** in the **Admin Web Page** section of this document.

```
#define def_AdminName     "Admin"             // Administrative admin logon
#define def_AdminPassword "Secret"            // Administrative admin password
```

Refer to Admin **Name** and **Admini Password** in the **Admin Web Page** section of this document.

```
#define def_Timezone      " EDT+5:00EST+4:00,M3.2.0/2:00,M11.1.1/2:00 "  // Local time zone
#define def_NTPServer     "us.pool.ntp.org"    // Name of default NTP server to synchronize to
```

Refer to **Timezone** and **Time Server** in the **Admin Web Page** section of this document.

```
#define def_FORCE   false  // Force these parameters to be used after the next boot
```

If set to true, at the next boot, any of the def_ values previously burned onto the board will be overwritten by the values designated in customize.h. This value is typically left at false and the values are maintained using the **Admin Web Page**.

```
#define onlyLocalAdmin    true  // admin.htm, fileman.htm, SerialWS and certain web
```

If false, anyone who knows the admin name and password can perform admin functions. If true, access to admin functions are limited to only being accessible from the local network. “Local network” means, someone logged onto the servers access point or the same WiFi as the server. “Admin function” means, the Admin Webpage, File Manager Webpage (if implemented), the SerialWS Webpage (if implemented) and certain security relevant websocket messages.

// ---------- Safety net section ----------

Values in the safety net section provide for an automatic reboot of the server if certain (undesired) situations are detected. Such conditions are either a good indicator that something is going wrong with the server and it might be a good time (or YOU decided) to reboot in order to prevent the server from crashing.

```
#define useSafetyNet true             // Safety net forces a reboot based on...
```

If **set** to true, the safety net is activated. A setting of false deactivates the safety net. If activated:

```
#define safetyMinFreeHeapKB    50   // Minimum heap that must be available.
```

The minimum memory heap size that must always be available, in KB. An entry of 0 disables this feature. The “heap” memory is used to store data during typical server process, such as when a client connects to the server.

```
#define safetyMinAllocHeapKB   5  // Minimum block of heap that must be available.
```

Similar to the minimum heap size, but indicates the minimum heap memory, in KB, that must be available as a single block. Heap memory can become fragmented, for example into 20 blocks of 1KB each. Though there would be a total of 20KB, if a variable needs to be defined with a size of 2KB, it can’t because there are no blocks of at least 2KB in size. An entry of 0 disables this feature.

```
#define safetyMaxRebootDays    0    // Force a reboot every n days.
```

This is the strictest form of the safety net. The server will be rebooted as a precaution just after 3:00 AM every so many days, even if everything appears to be running normal. An entry of 0 disables this feature.

```
// ------------------------ Mass storage section -------------------------
```

Every web server needs mass storage to store the files and data on. **The ESP32 S3 Webserver Project** software supports three types of storage.

```
#define SDType TypeMMC      // Possible values are TypeNoSD, TypeSD, or TypeMMC only
```

Set this entry to TypeSD to use standard SDHC cards. These cards can contain up to 32GB of data. Set it to TypeSDMMC to use more modern SDXC cards which are faster and can contain up to 2TB of data. If your project will not be using any SD card, set this entry to TypeNoSD. Note that the software and documentation may use SD, MMC, SDMMC, SDHC and SDXC as interchangeable terms for simplicity. In fact, the hardware is different. If using The **ESP32 S3 Webserver Project** hardware, the SD slots mounted to TFT boards are usually only SDHC, the connectors C and I can only be used for SDHC cards and an SDXC card can only be supported using the T connector.

Depending on the setting of SDType, additional definitions may be necessary.

```
#if (SDType == TypeSD)         // If using standard SD following defines are necessary

#define SDformatIfFail false   // True if SD card should be formatted if mounting fails
#define SDcs 1                 // Chip Select pin for standard SD card
#define SDMHz 40               // The SD lib has a speed of 4 but tests show 40MHz as stable
#define SDmaxOpenFiles 5       // The SD library has 5 as the default so good starting point.

#if (SDType == TypeMMC)        // If using MMC, following defines are necessary

#define MMCformatIfFail false  // True if SD card should be formatted if mounting fails
#define MMCmode1bit false      // True for 1 bit, false for 4 bit (faster)

#define MMCChipDetect 18       // Pin to detect if a SD card is inserted. -1 if not needed
#define MMCclk 14              // CLK pin for MMC card
#define MMCcmd 15              // CMD pin for MMC card
#define MMCd0   2              // Data0 pin for MMC card
  #if (!MMCmode1bit)           // If not using the slower 1 bit mode
    #define MMCd1   4          // Data1 pin for MMC card
    #define MMCd2  47          // Data2 pin for MMC card
    #define MMCd3  21          // Data3 pin for MMC card
  #else
    #define MMCd1 GPIO_NUM_NC  // Data1 pin for MMC card is not used in 1 bit mode
    #define MMCd2 GPIO_NUM_NC  // Data2 pin for MMC card is not used in 1 bit mode
    #define MMCd3 GPIO_NUM_NC  // Data3 pin for MMC card is not used in 1 bit mode
  #endif
#define MMCMHz BOARD_MAX_SDMMC_FREQ // SPI speed, in MHz, for MMC access.
#define MMCmaxOpenFiles 5      // The SD_MMC lib has 5 as the default so good starting point.
```

Note that for both SDTypeSD and SDTypeSDMMC, the defaults shown in customize.h match the required settings if you are using The ESP32 S3 Webserver Project hardware and only need to be changed if you have designed your own hardware.

As for the maxOpenFiles setting, the requisite libraries have a default of 5, so that is a good starting point. However, if a webpage calls for several files back to back (i.e. during initial load, requesting .css files, .js files, maybe images) then the 5 is quickly spent and may need to be raised.

```
#define UseLittleFS false      // Will LittleFS (files in flash) be used.
#define fmtLittleFS true       // If LittleFS is used & doesn't mount, should it be formatted?
```

Set `UseLittleFS` to true and LittleFS can be used for small files and data and may be faster than an SD. The size of LittleFS storage depends on the size of the Flash on the ESP32 chip, the size of the software, and if OTA is also used. The flash available for LittleFS is determined by the partition scheme you selected.

```
// ------------------------ Serial section ------------------------
```

Serial output can be used to display information during the initialization of **The ESP32 S3 Webserver Project** software and if/when errors or unusual events occur. **The ESP32 S3 Webserver Project**

software supports the standard, hardwired, Serial port but can also provide a Serial Monitor over a webpage.

```
#define serialEnabled true  // If true,serial will be enabled.If false,serial won't be enabled
#define serialBaud 115200   // Speed of serial, if implemented
```

Set serialEnabled to true and the correct baud rate if you intend to monitor the server over a hardwired Serial Port such as a USB connection. Note: If you are using the Serial Port for other purposes in your projrct, leave this at false and implement the Serial Port in your customization programming in order to prevent the server sending errors/warnings to your customizations.

```
#define setupVerbose false  // Controls serial output during setup.
```

If using a hardwired Serial Port, setting setupVerbose to true provides a lot of information during the initialization of the server. This causes lots of messages to be sent to the Serial monitor during the initialization of the WebServer. It's nice information during debugging but bloats the program size and isn't much use once debugging is completed so it will typically be set to false.

```
#define useSerialWS true        // If true, serial is available over the web
```

If useSerialWS is set to true, serial information will be available over the web page /SerialWS on this server. This is useful if you want to monitor the server but the hardware is remote, such as in the garden monitoring soil moisture levels in your vegetable beds.

If using SerialWS:

```
#define SerialWSrxSize  64        // This defines the size of the receive buffer
#define SerialWStxSize 256        // and the transmit buffer
```

SerialWS can be used to provide a serial monitor over a webpage. While this feature is of limited use during the bootup sequence (because the server isn't running yet), it can be used to monitor the server for errors that may occur once it is up and running. It should be noted that, if used, SerialWS will internally implement a second websocket handler because whenever serial is to be sent, it will be broadcast to ALL clients currently attached to that handler. If it didn't use its own dedicated handler, clients who weren't particularly interested in SerialWS output (such as the admin webpage, fileman webpage, your custom webpages, etc.) would also receive the SerialWS broadcast. This second websocket handler does add some overhead to the server.

The server software itself doesn't use serial input and thus only a minimal receive buffer is defined. If your project requires receiving data over SerialWS, you should increase the size of the receive buffer accordingly.

```
// ------------------------ Options section -------------------------
```

The ESP32 S3 Webserver Project software supports various optional functionality.

```
#define UseOTA true                   // Using OTA doubles the amount of Flash used for
```

Define if OTA should be used. If OTA is used, program space will be used twice from the Flash but OTA has the advantage of being able to reflash the software without the need of a physical USB connection and can be extremely convenient for projects mounted within a case and without a USB port.

```
#define UseStatus true                // /status is a dynamic webpage designed for debugging
                                      // purposes and displays a server status
```

The /status web page displays the hardware configuration and information about the available and used memory, file storage space etc. and can be useful during development to detect memory leaks.

```
#define UseFileManager true           // File Manager is a web page that allows for listing,
                                      // uploading and downloading of files to either SD/MMC
                                      // (as defined above) and/or LittleFS (as defined above)
```

Define if the file manager should be implemented or not. If it is used, `FMblockedPath` designates a path that is virtually invisible to the filemanager. The file manager can be accessed with `http://hostname/fileman`. (Where hostname is the name you gave the server.)

```
const char FMblockedPath[] ="/system/";
```

Defines a path that will be invisible and deemed nonexistent in the File Manager. Use this path for system files (html, css, js, images, etc.) that the File Manager cannot delete, edit or even see. This path will be excluded on all file systems used by File Manager (SD and LittleFS).

```
#define embedFileman true             // If true, fileman.htm will be embedded in the program.
```

If true, a GZipped version of fileman.htm will be embedded into the software. This is convenient to facilitate the initial load of the files but it also takes up several KB of program space so it is better to set this to false after at least fileman.htm has been loaded to the server.

```
// ------------------------ Webpage section -------------------------
```

This section is used to define where the various system pages are to be found.

```
#define StandardFiles onSD            // Defines where admin.htm and favicon.ico and
                                      // DefaultHTM are located. Possible values
                                      // are onSD or onLFS
```

Will standard files such as the default web page, favicon.ico, admin webpage, fileman webpage and SerialWS webpage be stored on the SD, or in LittleFS?

NOTE: <u>If you enable both an SD card and LittleFS</u> and a webpage makes reference to a file (i.e. /logo.jpg), the server software has no idea of whether that file is on LittleFS or on the SD card. For that purpose, special notation must be implemented when referring to such files. If the file is in LittleFS, use /f as a prefix (i.e. /f/logo.jpg). If the file is on SD, use /s as a prefix (i.e. /s/logo.jpg).

```
const char* cacheRule = "max-age=31536000, must-revalidate";    //31,536,000 = seconds in 365
days so max-age = 1 year.
```

The server will tell the client to cache the files locally in order to reduce the load on the server itself.

```
#define DefaultHTM "/main.htm"        // Default file when webserver is accessed (typically
                                      // index.htm, main.htm, etc.) Remember the preceding
                                      // slash!
#define FavIcon "favicon.ico"         // Default file for favicon (typically *.ico, *.jpg,
                                      // *.png, or *.bmp)
```

Define the filenames of the standard webpage and icon. Note that the names are case sensitive.

Note, main.htm is provided as a sample. It displays a clock but also contains samples of using websockets with JSON formatted parameters. Favicon.ico (The letter W arranged like the a in the @ symbol) is provided as a sample.

```
#define AdminHTM "/admin.htm"         // File for admin webpage
```

Note, admin.htm is provided as a sample. You might add to it in order to administer additions you implemented in your customizations but it is probably a better idea to just implement your own separate admin webpage for your customizations.

```
#define FilAdminHTM "/fileman.htm"  // File for fileman webpage if using the fileman option
#define SerialHTM "/SerialWS.htm"   // File for SerialWS webpage if using the SerialWS option
```

Note, fileman.htm and SerialWS.htm are provided.

```
// ------------------------ Telegram section ------------------------
```

The ESP32 S3 Webserver Project software provides functionality for sending and receiving Telegram messages using one or more Telegram Bots.

```
#define UseTelegram TelegramSendReceive // Possible values are noTelegram, TelegramSend,
                                        // TelegramReceive, or TelegramSendReceive only
```

Use this setting to define if Telegram should be used to receive and/or send messages.

Bot tokens would have been provided to you when you created your Bot in Telegram. You can define multiple Bots and determine if each Bot should be used to receive messages. Note: Each Bot that can receive messages must be periodically polled which will place a small load on this server as well as your local WiFi. Using multiple Bots can be useful. For example, the Telegram app on your phone supports muting and different ringtones for each chat so you could specifically set up one Bot to receive high priority alarms from your server.

```
  Bot Bots[] = {
    {"9999999999:AAAaaAAa_aAAaaA9aAaA9a9A9aaA9a-A9aa",true,0}
  };
```

The first (long and cryptic) value is the bot token, second = true/false indicating if this bot RECEIVES messages, third should always be zero.

```
#define CHAT_ID     "217889858"       // Primary Telegram user.
#define ALT_CHAT_ID "40676821"        // Alternate Telegram user.
```

Define a primary and, optionally, an alternate Telegram ChatID. Note that there are spammers out there that attack random Telegram Bots. This server will ignore any messages not coming from either the primary or alternate Chat ID. If you send a Telegram message of /status, the server will respond with a short status of the server, similar to the status webpage.

```
#define TelegramOnBoot true
#define TelegramAltOnBoot false
```

The server can send a Telegram message with status information to the primary and/or alternate Chat ID each time it boots.

```
#define TGRECEIVEFREQ 60           // How many seconds between polling for received messages?
```

Define how often Bots that receive messages will be polled. The lower the number (in seconds) the more responsive the Bot will be but it will also consume more network bandwidth.

See Appendix B for details on creating a Telegram Bot and obtaining chat IDs. The appendix also contains information on how you can integrate your own Telegram messaging into your customizations of **The ESP32 S3 Webserver Project** software.

Libraries Used

The ESP32 S3 Webserver Project software uses the following libraries which are not part of the standard Arduino ESP32 environment and will need to be loaded separately:

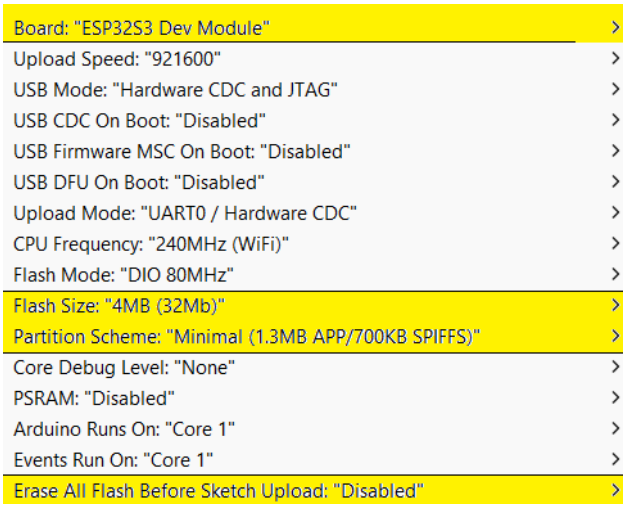
ESPAsyncWebServer, by lacamera, 3.1.0
ESPAsyncTCP, by dvarrel, 1.2.4
ArduinoJson, by Benoit Blanchon, 7.0.2
UniversalTelegramBot, by Brian Lough, 1.3.0

Quickstart

This quick start assumes you have made NO changes to customize.h. Therefore, this will be a minimal installation and:

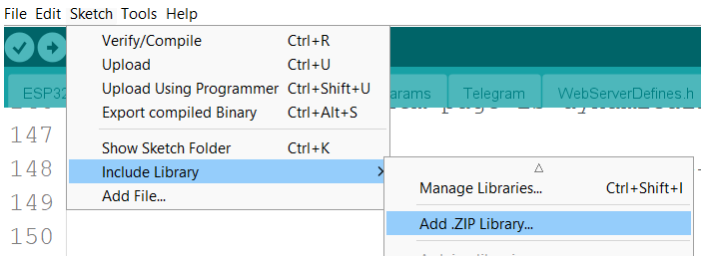
- The server name will be WebServer and create its' own access point
- The password to connect to the WebServer access point will be "TopSecret"
- The server will not connect to your local WiFi
- The local timezone will be set to US Eastern Time
- Administration will only be available from the WebServer access point
- The logon and password for administration will be "Admin" and "Secret"
- LittleFS (files on flash) will be enabled and may be formatted if necessary
- No SD card is used
- The Serial Monitor will be enabled and setup will be verbose (lots of messages)
- The file manager will be enabled (so you can easily load the required files later)
- SerialWS will be enabled (but not yet available until you load the files using fileman)
- Telegram messaging will be disabled

Set your board settings to the correct settings and be sure to select a partition that allows some LittleFS usage. For example:



The ESP32 S3 Webserver Project software uses my SerialWS library. If you have not previously installed it, you will need to do so before compiling the sketch. To do so:

1) From the Arduino menu, select Sketch -> Include Library -> Add ZIP Library
2) Locate the SerialWS.zip file I have included with the rest of the files for this project and open it.
3) The Arduino IDE will do the rest for you.

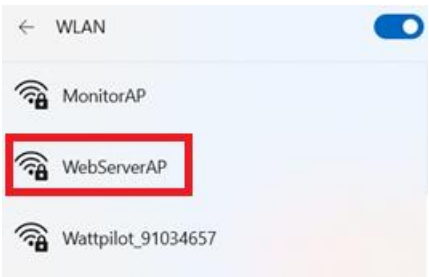


You should open the Serial Monitor before loading the sketch. That way, you will see the messages from the server as it starts up. If errors occur, you will see them too. You will receive warnings about missing files and the current time will be strange, but we will take care of those issues in the next steps. A sample Serial Monitor output is:

```
Initializing
Creating initial parameters
LittleFS mounted
Starting Access Point and WiFi
Hostname: WebServer
Access Point WebServerAP established on: 192.168.4.1 (public)
Local WiFi not defined
Initiating OTA
Starting OTA
mDNS responder started for: WebServer
Initiating Server
Warning: File /Main.htm is missing on LittleFS
Warning: File /favicon.ico is missing on LittleFS
Warning: File /admin.htm is missing on LittleFS
Warning: File /fileman.htm is missing on LittleFS
Warning: File /SerialWS.htm is missing on LittleFS
Starting server
Server is online
ESP32-S3
2 cores at 240MHz
32768KB Fast Read 80Mhz Flash
500KB of 1536KB free on LittleFS
8176KB of 8189KB PSRAM free
MCU Temp: 40.9°C
244KB of 334KB heap free
235KB maximum heap block
Webserver Version: 1.3
Wifi Strength: 0dBm 150%
Current Time: 1970/01/01 00:00:00
Uptime: 0 days, 0 hours, 0 minutes, 0 seconds
```

Once you have loaded the sketch onto your board you will be able to:

- See the access point in your WiFi settings and log onto it using the password you set with `#define def_APpassword` in customize.h- Go ahead and log on now. (Tip: Open the documentation pdf before logging onto the webserver access point)



- `ping webserver` from a command line and get a result.

```
C:\>ping webserver

Ping wird ausgeführt für WebServer.local [192.168.0.199] mit 32 Bytes Daten:
Antwort von 192.168.0.199: Bytes=32 Zeit=26ms TTL=255
Antwort von 192.168.0.199: Bytes=32 Zeit=31ms TTL=255
Antwort von 192.168.0.199: Bytes=32 Zeit=46ms TTL=255
Antwort von 192.168.0.199: Bytes=32 Zeit=63ms TTL=255

Ping-Statistik für 192.168.0.199:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0
    (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 26ms, Maximum = 63ms, Mittelwert = 41ms
```

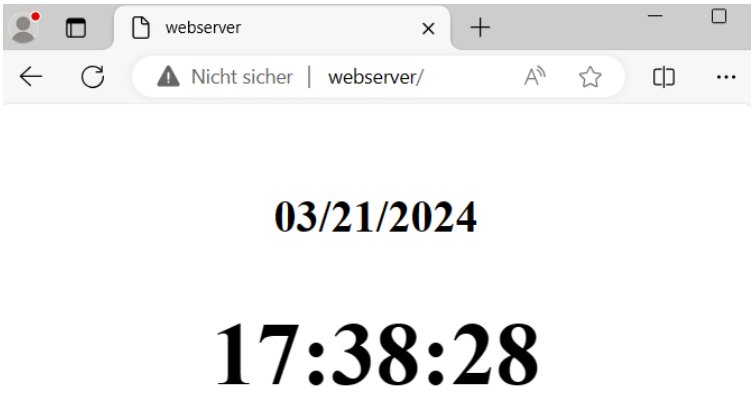
- Access http://webserver/fileman from your web browser.

Congratulations! The webserver is up and running! The first thing that needs to be done is to load some more files. Refer to **File Manager Webpage** later in this document for detailed instructions.

Main.htm: A sample webpage that is basically just a clock
admin.htm: The admin webpage for changing basic settings
fileman.htm: The file manager webpage which can be used to manage files
SerialWS.htm: The web based Serial Monitor
favicon.ico: A sample icon for the web page

Once the files have been created on the mass storage, you will be able to:

- Access http://webserver from your browser and get the clock wegpage contained in /main.htm or even enter simply webserver into your browser and get the same response



- Access http://webserver/status from your browser and get the Status wegpage described later in this document.
- Access http://webserver/SerialWS from your browser and get the SerialWS web based Serial Monitor described later in this document.
- Access http://webserver/admin from your browser and get the Admin wegpage described later in this document, but, before you do that:

Make an edit to customize.h and set def_FORCE to false. If you don't set def_FORCE to false, any changes you make to the configuration using the Admin webpage will be overwritten the next time the server boots. While you are making that change, you should also set embedFileman to false

because the embedded version is no longer needed now that you have loaded the fileman.htm file onto LittleFS and now it is only taking up program space. Reload the sketch to make the changes take effect.

Since OTA was enabled, you might take this opportunity to test loading the sketch via OTA instead of using the USB. You should see the webserver as a network port under tools.

First Steps

The first step you should take is to use the Admin webpage and configure YOUR project.

If you are using a SD or MMC card, connect it now. You may want to test the SD/MMC card using the examples included within the SD or SD_MMC libraries. If you are using The ESP32 S3 Webserver Project hardware, be sure to configure using the pins designated within this documentation for the SD/MMC card.

Once that is done, if you are using a SD or MMC card or you want to use Telegram messaging in your project, make those changes to customize.h and, yet again, reload the sketch to your board. Pay particular attention to the Serial Monitor for any warning or error messages and resolve those problems before going any further.

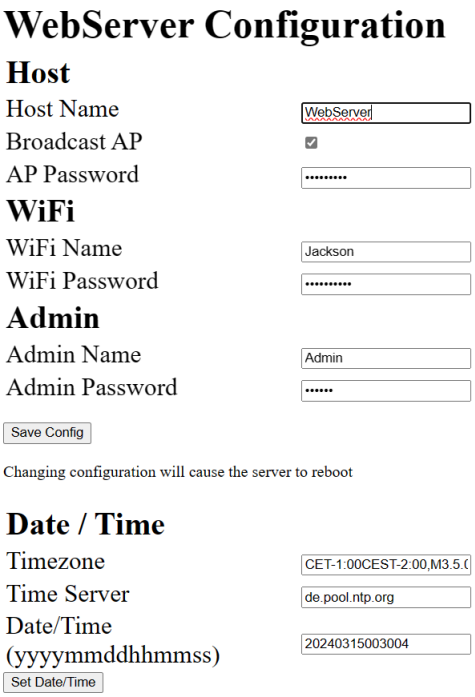
If you configured Telegram to send and receive and have configured the bot and chat id(s) properly, you may get a Telegram message when the server boots (if you configured that option) and may also be able to send "/status" in a Telegram message to your server to get the same status description as in the Status webpage.

It is **NOT recommended** to set serialEnabled to false until you are fairly sure that the server is running properly. Once everything is running properly, you could disable the Serial and use SerialWS

Once you have The ESP32 S3 Webserver Project (hardware and) software up and running, refer to **Customizing Your Own Project** later in this document for information on how to add your own coding and features to your webserver.

Admin Webpage

The Admin Webpage is in a file (admin.htm) and must be on the SD or LittleFS as specified by `#define StandardFiles` in customize.h. You can call it up in your browser with http://webserver/admin



The admin webpage is used to configure **The ESP32 S3 Webserver Project** software once the server is up and running. It also provides samples of using websockets with simple string array styled parameters.

You will notice that many options of **The ESP32 S3 Webserver Project**, such as the use of a SD/MMC card, LittleFS or Telegram cannot be configured from the admin webpage. This is intentional. By configuring those options within customize.h, the respective parts of the software can be included or excluded. By excluding unnecessary parts of the software, more space is available for LittleFS (though the Flash Partition may need to be tweaked).

When configuring your webserver:

Host Name: The name given here forms the address for webpages and OTA (if implemented) as well as for the access point (with AP added to the end). The host name must be 4 to 32 characters long. The first character must be A-Z or a-z.

Broadcast AP: If checked, anyone looking for networks in the area will see this access point. If set to false, the name of this access point will not be listed as an available network and they would have to know the name and type it in in order to attempt to access the access point.

AP Password: Must be between 8 and 16 characters. It must have at least one character from at least three of the following four groups: A-Z, a-z, 0-9 or a special character (such as !, ., +, -, or #).

WiFi Name: Must, of course, match your local WiFi name for this server to access your local WiFi. If blank, no WiFi connection will be made and the webserver will only be available over the access point it provides. Without a WiFi connection, time synchronization and Telegram are, of course, also not available.

WiFi Password: Will be ignored if no WiFi name is provided.

Admin Name: Must be 6 to 16 characters long. The first character must be A-Z or a-z. (Yes, the default of "Admin" violates the rules und therefore MUST be changed. This is intentional!) The same logon and password are used for the file manager and SerialWS web pages if they are implemented.

Admin Password: The rules are the same as for AP Password. (Again, the default value of "Secret" violates the rules and therefore MUST be changed and is intentional!)

Timezone: The time zone defines your local time zone and daylight savings time implementation. The entry must be 4 to 50 characters long. Further rules apply simply because timezones have a specific format of how they are expressed. (NOTE: Don't blame me for the syntax of the entry. I didn't invent it! It is an international standard.)

If you live in the Eastern Time zone of the US, your entries might be:

EDT+5:00EST+4:00,M3.2.0/2:00,M11.1.1/2:00 and us.pool.ntp.org

In Germany, our timezone data is `CET-1:00CEST-2:00,M3.5.0/2:00,M10.5.0/3:00`

That entry is interpreted as:

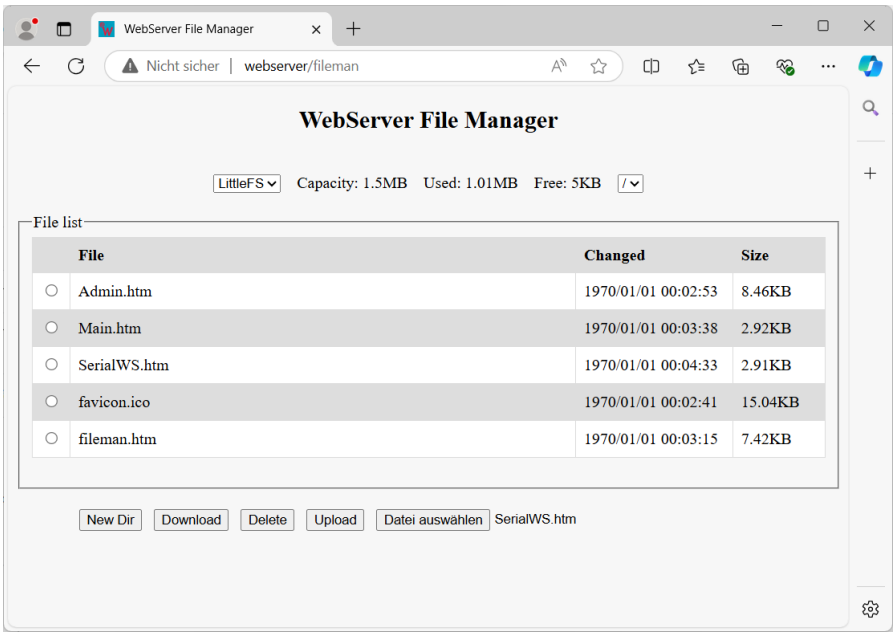
CET and CEST represent the name of the time zone during daylight savings and standard.
-1:00 and -2:00 are the offsets from this time zone to UTC/Greenwich during daylight savings and standard. M3.5.0/2:00 and M10.5.0/3:00 indicates that time changes in month 3 (March) and 10 (October) on the last (1-4 indicates which occurrence and 5 indicates the last occurrence) Sunday (0, Monday is 1, Saturday is 6) at 2:00 or 3:00 AM (24 hour clock).

Time Server: Must be between 8 and 35 characters long. Of course, in order for time synchronization to work, it has to be a valid internet time server. It is strongly recommended to make this entry a "pool" of time servers (indicated by the presence of "pool" in the server name). If in doubt, pool.ntp.org will always work but may be less accurate than using a server closer to your location. Tip, your WiFi router may even provide a time server of its own!

Date/Time: Must be exactly 14 numeric digits expressing the date and time in yyyymmddhhmmss format. (Year, month, day, hour, minutes, seconds). If a WiFi is available, the date and time will be synchronized to the Time Server periodically so this setting only makes sense if no WiFi is available. If there is no WiFi, this setting will need to be manually made each time the server boots.

File Manager Webpage

The File Manager Webpage is in a file (fileman.htm) and, if enabled in customize.h (`#define UseFileManager true`) must be on the SD or LittleFS as specified by `#define StandardFiles` in customize.h. You can call it up in your browser with http://webserver/fileman



The file manager webpage can be used to list, delete, upload and download files and create new directories on/to/from a SD card and/or LittleFS. When uploading, a progress bar is displayed. This webpage contains samples of using websockets with JSON formatted parameters.

The webpage is rather intuitive to use. The files can be sorted by name, date, or size, in ascending or descending order by clicking on the appropriate header in the file listing. The first click will sort using that column in ascending order, the second click in descending order, and the third click will return to the "natural" order (the order the files are found when reading the directory).

To download a file, simply select the desired file using the radio button to the left of the filename and click the Download button.

To delete a file, simply select the desired file and click the Delete button. WARNING: There is no request for confirmation. You click, you lose. (Which is why you might want to put the system files in the area protected from the file manager. See `FMblockedPath` above.)

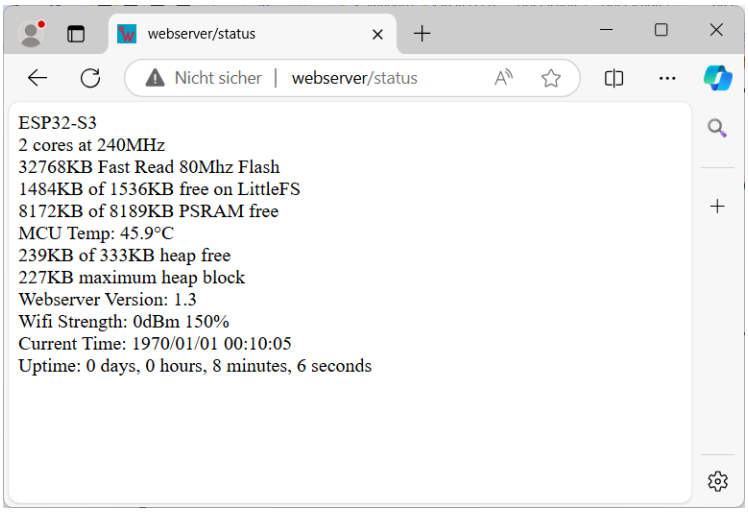
To upload a new file into the displayed directory, first select the file by clicking on the Select File button (my example above is in German and says "Datei auswählen" because the browser uses the international settings of the computer the browser is on). Then, click on the Upload button. There will be a progress bar displayed, but it really only makes graphical sense when uploading larger files.

You can switch between SD and LittleFS (if enabled in customize.h) using the dropdown. You can move between directories using that dropdown. Note that the directory ".." is used to move up one directory level.

You can create a new directory by clicking the New Dir button. A popup will appear where you enter the name of the new directory.

Status Webpage

The status webpage is dynamically generated. There is no corresponding .htm file. If enabled in customize.h (`#define UseStatus true`) it can be called up in your browser with http://webserver/status

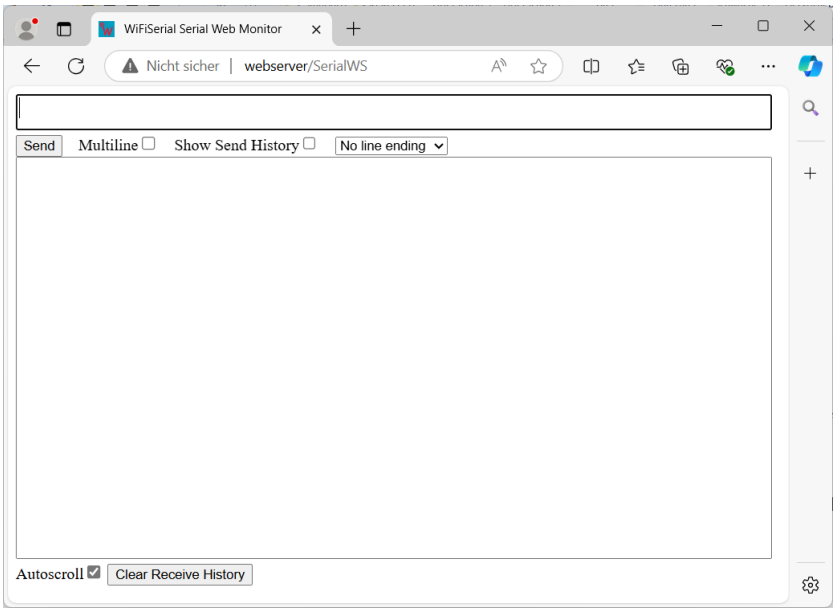


The page provides information about the server hardware, LittleFS storage, SD card storage, etc. Of particular interest is the amount of heap that is free. (Simply put, heap is a region of memory reserved for special purposes, usually temporary, as the program runs.) If this continually gets smaller, it is a sign of memory leak within the programming and the server will eventually crash and automatically reboot. The status also includes the Wifi Strength which could be helpful in placing the server where it has a good connection.

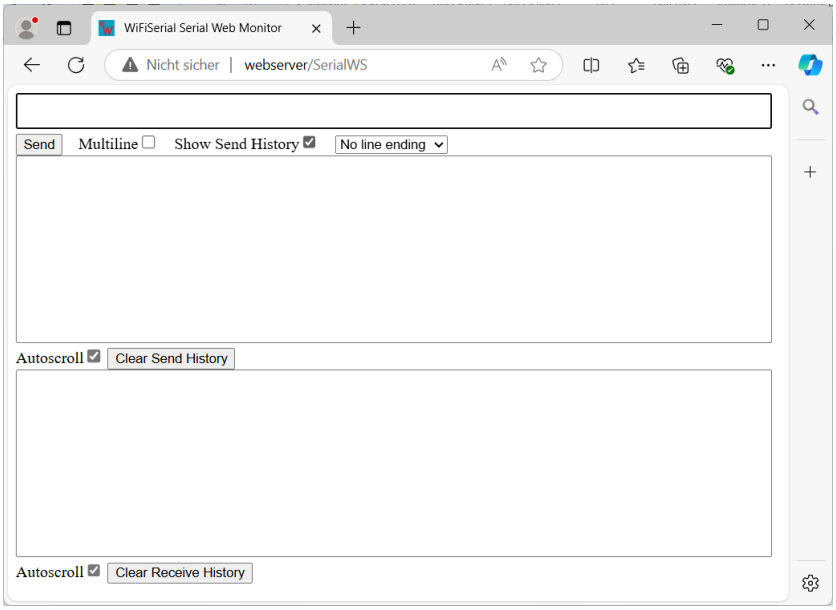
SerialWS Webpage

The SerialWS Webpage is in a file (SerialWS.htm) and, if enabled in customize.h (`#define UseSerialWS true`) must be on the SD or LittleFS as specified by `#define StandardFiles` in customize.h. You can call it up in your browser with http://webserver/SerialWS. The main advantage of having SerialWS is that you can use the "Serial Monitor" without having to plug a USB cable into your project. This goes hand in hand with the usefulness of OTA updates.

By default, SerialWS is similar to the Serial Monitor in the Arduino IDE except that some of the buttons and labels are in different places.



However, it also has an additional feature. If you check “Show Send History”, it shows not only everything that had been previously received, but also what has been sent.



Customizing Your Own Project

Once you get **The ESP32 S3 Webserver Project** up and running, the real fun begins. You can easily implement your own customizations to create your own projects. Whether you are designing a weather station, home automation or whatever, **The ESP32 S3 Webserver Project** handles the meat of the infrastructure and provides the flexibility to realize the project. (My first project involved monitoring the solar panels, battery charge, and electrical usage in my house. It sends Telegram messages to remind us that it might be a good time to run the dishwasher, washer, dryer or charge the electric car as the batteries reach capacity and excess power is available. It also has a display so we can see the power status at a glance.)

Always start with a **COPY** of **The ESP32S3 Webserver Project** software. Open the project and begin by doing a File->Save As… to the name for your new project.

While you are certainly welcome to poke around in the programming of **The ESP32 S3 Webserver Project** software, you don’t have to. You can simply add your projects’ programming into the custom tab. The custom tab (actually custom.ino within the sketch) has several sections to it. The first few sections are designed for general things like including libraries, defining global variables, designating interrupt service routines, etc. The rest of the sections contain an empty function that is called from the main routines at designated times. The various sections are documented below.

But, first, let’s take a look at some resources built into main software that you may find useful when coding your customizations.

SD or SD_MMC: Normally, you have to use the SD or SD_MMC library depending on which type of SD card interface your hardware uses. Unfortunately, those libraries are nearly exact but are different enough that the programming has to be different depending on which library used. For example, SD.begin() and SD_MMC.begin() are functionally identical. It is inconvenient to have to reprogram simply because you switch SD modules. For this reason, **The ESP32 S§ Webserver Project** software

redefines SD_MMC so that you can use, for example, SD.begin() even though you are using the SD_MMC library. So, regardless of which type of card module you are using, you CAN use the simpler syntax.

Current time: There is a global variable of type tm named currentTime. It is accurate to about half a second. It will be local time, including daylight savings time where used, as defined by the local timezone. Because it is of type tm, it has the following structure:

Member	Type	Meaning	Range
currentTime.tm_sec	int	seconds after the minute	0–59
currentTime.tm_min	int	minutes after the hour	0–59
currentTime.tm_hour	int	hours since midnight	0–23
currentTime.tm_mday	int	day of the month	1–31
currentTime.tm_mon	int	months since January	0–11
currentTime.tm_year	int	years since 1900	
currentTime.tm_wday	int	days since Sunday	0–6
currentTime.tm_yday	int	days since January 1	0–365
currentTime.tm_isdst	int	Daylight Saving Time flag	*

Note that tm_year is not something like 2024 but that you have to add 1900 to the value to get the true year. So, 124 is the value of tm_year in calendar year 2024. Note also that tm_mon is not what you would normally expect. January is 0, not 1. So, you have to add 1 to tm_mon to get the calendar month. Don't blame me, this standard has been around since the early 1970's where every bit and byte of storage was expensive and limited. The DST flag is <0 if DST data is unavailable, 0 if DST is not in effect and >0 if DST is in effect. On very rare occasions, the seconds may be 60 or 61 (for an instant when the atomic clock has to be adjusted). Sheldon Cooper Fun Facts: Between 1972 and 2024, this has occurred 27 times, most recently on December 31, 2016. It occurs on either June 30 or December 31 at midnight GMT.

void **textToHTML**(char txt[], int len): This is a function to convert simple text, which may contain symbols or non-English characters not compatible with the HTML protocol into HTML encoded UTF-8 text so the text can easily be displayed on a webpage. For example, "72°F & 80% Humidity" would be converted to "72° F & 80% Humidity". "El-Niño" would convert to "El-NiÑo" and "André" would convert to "André". The first variable must be a char array that contains the text to be converted and will be overwritten with the converted text. The second variable is the maximum size of the converted text. As you can see, the converted text can be substantially longer so be sure to allow space in the variable to convert.

SerialWS: If you want to print to the Serial port, you normally use Serial.print("text"). When using SerialWS, the corresponding syntax would be SerialWS.print("text"). The ESP32 S3 Webserver Project software supports both Serial and SerialWS optionally. That means, you would have to issue two print statements and, even worse, would have to change the programming if you decided later to change which Serial options you wanted to use. To avoid that, use the internal functions serialPrint() and serialPrintln() instead. Those functions will already know which serial options you are using and issue Serial.print() and/or SerialWS.print() statements for you. For simplicity sake, only printing of char arrays and int have been implemented.

```
void serialPrint(char buff[], bool immediate)
void serialPrint(int val, bool immediate)
void serialPrintln(char buff[], bool immediate)
void serialPrintln(int val, bool immediate)
```

The first parameter is what to print, the second indicates if the data should be immediately pushed to clients (true) or if it can be buffered until later (false). If issuing multiple prints back to back, it is best to not push them until the final print is issued. Don't worry, no data gets lost. Automatic pushes are performed as necessary to prevent overflowing the transmit buffer.

You can also use SerialWS as you would the Serial port, for example with SerialWS.print(), SerialWS.available(), SerialWS.read(), etc. SerialWS.send() forces an immediate broadcast of any data in the transmit buffer.

Custom Code

The various sections of custom.ino where you will place the code for your project customizations are as follows:

```
// ------------------------- Library section -------------------------
// Libraries
// Put any libraries necessary for your customizations here
// Sample: eSPI_TFT
// #include <eSPI_TFT.h>
```

The **Library section** is used to #include any libraries necessary for your customizations. This is what you normally do at the beginning of any sketch.

```
// ------------------------- Global section -------------------------
// Global
// Put any global variables, #defines, structure definitions, etc.
// necessary for your customization here
// Sample: A pushbutton
// #define button 42                // The pushbutton is on pin 42
// bool buttonPress = false;        // Will be changed to true within the ISR
```

The **Global section** is used to #define constants, designate global data structures, instantiate global variables, etc. This is the same as you would normally do at the beginning of any sketch.

```
// ------------------------- ISR section -------------------------
// Interrupt service routines
// Put any ISR necessary for your customization here
// Example:
// void IRAM_ATTR ISR_button(){buttonPress = true;}       // ISR to handle button
presses. IRAM_ATTR keeps it in memory.
```

The **ISR section** is a convenient place to define your own interrupt service routines. Define ISRs here as you would in any other sketch in which they were required.

```
// ------------------------- Setup section -------------------------
// Setup
// Put any additional setup requirements for your customization here
```

The **Setup section** is sub-divided into two subsections.

```
void customEarlySetup()
```

customEarlySetup() will be called before most of the server resources are made available. Use customEarlySetup for things that may need to be done more or less immediately upon boot, such as ensuring that a buzzer is not on or blanking a display (maybe display a flash screen).

```
void customLateSetup()
```

customLateSetup() will be called after all of the basic webserver setup is complete which means that things like local time, the SD card, Telegram, etc. would already be initialized and available within your customization.

```
// ------------------------ Webpage section ------------------------
void customPages()
```

customPages() is called once during server initialization and is used to register any custom webpages so that they will automatically be served when a client requests them. The registration can be handled in a number of ways.

See ESPAsyncWebServer documentation for ways to register webpages. [ESPAsyncWebServer]

One of the most simple ways to register is:

```
  server.on("/stat", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SD, "/stats.htm");
  });
```

This would serve the file /stats.htm from the SD card when the webpage /stat were requested.

You could use a function built into the software to do the same thing except that the file would also be cached on the client which reduces network traffic, thus improving performance, the next time the webpage were requested.

```
  server.on("/stat", HTTP_GET, [](AsyncWebServerRequest *request){
    responseWithCaching(request, SD, "/stats.htm");
  });
```

If the webpage is considered an administrative function such that you want the admin rules (`onlyLocalAdmin, AdminName, AdminPassword`) to apply:

```
  server.on("/stat", HTTP_GET, [](AsyncWebServerRequest *request){
    if(adminAllowed(request))responseWithCaching(request, SD, "/stats.htm");
  });
```

This will enforce the rules and send a 403 Forbidden error to the client if they are not met.

Some pages may require no setup at all as they may be picked up by the standard server.serveStatic() routine if they are in the location defined for stdFiles (though the full name would be required. Like "/sensors.htm"). Only add them here if you are getting page not found errors when accessing your pages.
If using both LittleFS and a SD card, remember that the prefix of /f for LittleFS or /S for SD may be required for files to be served. (Refer to the documentation within customize.h for further details.)

```
// ------------------------ Status section ------------------------
void customServerStatus(char stat[]){
```

customServerStatus() is called whenever the server generates a status message (on boot, for the /status webpage, or when /status is received by Telegram). Use `customServerStatus()` to add any messages to the server status that you may require. Remember to end messages with a newline ("\n"). Note, the standard status set by the server is usually less than 400 characters long. The status may NEVER be more than 999 characters long so keep that in mind when creating your own messages!

```
// ------------------------ Loop section -------------------------
bool customLoop(bool didOne){
```

customLoop() is called once during each iteration of loop() within the webserver sketch. Add code to this section just as you would in any normal Arduino loop() function. You need to be aware that if any iteration of the loop() takes too long, the RTOS operating system used by the ESP32 will force a reset. For that reason, avoid running any code at all possible if the parameter didOne is true. Likewise, if you do any custom processing within this function, set and return customDidOne true so that the servers main loop function is aware that processing has taken place and will not try to do any avoidable further processing of its own. Remember also to only execute your custom features periodically, such as every few seconds, or your customizations will block lower priority in the main webserver loop and, for example, the time would never be synchronized to the internet.

customLoop() will be called AFTER higher priority processing within the webserver loop (such as maintaining clients, polling for Telegram messages, etc.) but before handling lower priorities (such as time synchronization). customLoop() will NOT be called at all during an OTA update.

```
// ------------------------ Websocket section -------------------------
void customWS_EVT(AsyncWebSocket * server, AsyncWebSocketClient * client,
AwsEventType type, void * arg, uint8_t *data, size_t len)
```

customWS_EVT() will be called when a websocket data request is received that is NOT handled by the standard webserver processes themselves. For further details on the parameters, see the documentation for the ESP Asynchronous Web Server library

When the server receives a websocket request, it has no idea how to respond to it. There has to be some data embedded within the request so that while handling the request the servers programming can figure out what is being requested. There are endless ways to do that and two are actually used within **The ESP32S3 Webserver Project** software.

One method involves putting all the necessary data into separate lines of a text block where the first line contains the “name” of the request. The other involves using JSON encoding where a value of “request” is part of the JSON. Both methods are demonstrated in the sample `customWS()` function below.

```
void customWS_EVT(AsyncWebSocket * server, AsyncWebSocketClient * client,
       AwsEventType type, void * arg, uint8_t *data, size_t len){

  switch(type){
  case WS_EVT_DATA:  // EVT_DATA type is probably the only one you need to handle
    AwsFrameInfo * info = (AwsFrameInfo*)arg;
```

```
    if(info->opcode == WS_TEXT){  // Note: Apps only see WS_TEXT and WS_BINARY.
      // Assuming a maximum operand length of 24 plus terminating 0.
      // Increase if necessary. (Or use shorter names!)
      char operand[25];
      // Rudimentary test to determine if the message is in JSON format
      bool JSON = data[0]=='{';
      if(JSON){
        char* reqAddr = strstr((char*)data,"request");
        // Does the data contain "request" in the first ten characters? If so..
        if((reqAddr != NULL) && (reqAddr - (char*)data < 10)){
           JsonDocument doc;                          // Allocate the JSON document
           // Test if parsing succeeds and deserialize the JSON document
           DeserializationError Jerror = deserializeJson(doc, data);
          if(!Jerror){                                // Test if parsing succeeded
            const char* request = doc["request"];
            // Sample: a request "setVal"
            // if(strcmp(request,"setVal") == 0){
            //   doSetVal(doc);   // Additional parameters are in doc.
            // }
          }
        }
      } else { // The message is not JSON, get first line of a text block
        getOperand(operand, (char*)data, sizeof(operand));
        // Sample: a request "getVal"
        // if(strcmp(request,"getVal") == 0){
        //   doGetVal(data);   // Additional parameters are in data.
        // }
      }
    }    break;
  }
}
```

Now, you may be wondering how those websocket requests can be generated from within your own webpages. The following Javascript was used from within a webpage that periodically requested random text messages and a picture from the server. The websocket requests are formatted without JSON.

```
<script>
var Socket,initialized=false,intervalId;
var nextMsg0="", nextMsg1="", nextMsg2="";
var nextImage=new Image();
nextImage.src="http://"+window.location.hostname+"/DEFAULT.JPG";
Refresh();
establishWS();
function establishWS(){
 try{
  Socket=new WebSocket("ws://"+window.location.hostname+"/ws");
  Socket.onopen=function(evt){WSonOpen(evt)};
  Socket.onclose=function(evt){WSonClose(evt)};
  Socket.onmessage=function(evt){WSonMessage(evt)};
  Socket.onerror=function(evt){WSonError(evt)};
 }
 catch(err){
  alert(err.message);
 }
}
function WSonOpen(evt){
 if(!initialized){
  intervalId=setInterval(function(){Update();},13000);
```

```
  initialized=true;
 }
}
function WSonClose(evt){setTimeout(function(){establishWS();},1000);}
function WSonMessage(evt){
 var s=evt.data,tl=s.split("\r");  // Each line ends with \r, split into an array
 switch(tl[0]){                    // The first line is the name of the request
  case "NewDataset":
   if(tl.length==5){
    nextMsg0=tl[1];
    nextMsg1=tl[2];
    nextMsg2=tl[3];
    nextImage.src="http://"+window.location.hostname+"/"+tl[4];
   }
   break;
  default:break;
 }
}
function WSonError(evt){}
function WSdoSend(msg){
 try{Socket.send(msg);}
 catch(err){}
}
function Refresh(){
 document.getElementById("AATTAMsg0").innerHTML=nextMsg0;
 document.getElementById("AATTAMsg1").innerHTML=nextMsg1;
 document.getElementById("AATTAMsg2").innerHTML=nextMsg2;
 document.getElementById("AATTAImage").src=nextImage.src;
}
function requestNewDataset(){WSdoSend("RefreshMe");}
function Update(){Refresh();requestNewDataset();}
</script>
```

The fileman.htm webpage uses websocket requests that are formatted with JSON. An excerpt of the Javascript is:

```
function elID(el){
 return document.getElementById(el);
}
function sendJSON(obj){
 try{Socket.send(JSON.stringify(obj)+"\0");}
 catch(e){alert(e.message);}
}
// FmgetDir sends a websocket request with parameters in JSON
function FMgetDir(){
 sendJSON({request:"FMgetDir",                    // request = "FMgetDir"
    filesys:elID("fsSel").value,dir:getPath()});} // gsSel = selected file system
}
function WSonMsg(evt){  // Called when websocket message received
 var s=evt.data;
  let msgJ=JSON.parse(evt.data);
  switch(msgJ.answer){   // Parameter answer tells us what the data is
  case"FMgetBasic":FMgotBasic(msgJ);break;
  case"FMgetFS":FMgotFS(msgJ);break;
  case"FMgetDir":FMgotDir(msgJ);break;
  case"FMuploadStatus":upStat(msgJ);
  default:break;
  }
}
function FMgotFS(msgJ){
```

```
 elID("capa").innerHTML=msgJ.capacity;
 elID("used").innerHTML=msgJ.used;
 elID("free").innerHTML=msgJ.free;
 FMgetDir();
}
```

Coding that sends websocket data back to the client might look like this:

```
char rsvp[200] = "theAnswerName\nFirstValue\nNextValue\nLastValue"
client->text(rsvp);
```

Or, for JSON formatted data:

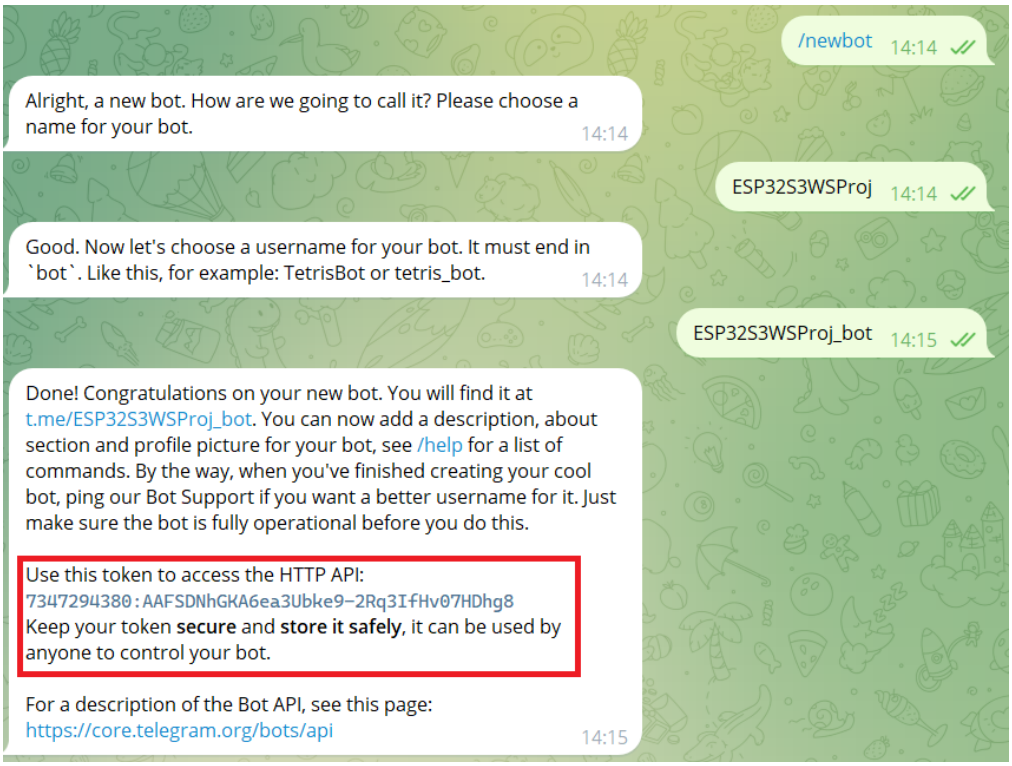
```
char rsvp[400] = "";
char buff[100];
strcpy(rsvp,"{\"answer\":\"theAnswerName\",\"answerValue\":\"");
strcat(rsvp,someCharVariable);
strcat(rsvp,"\",\"dataArray\":[");
for(int i = 0; i < numElems; i++){
  sprintf(buff,"\"%s\"%s",elems[i].name, i==numFS-1 ? "" : ",");
  strcat(rsvp,buff);
}
strcat(rsvp,"]}");
client->text(rsvp);


// ------------------------ Telegram section ------------------------
bool customTelegramRcvd(int botNum, UniversalTelegramBot bot, struct
telegramMessage tMsg)
```

The **Telegram section** handles received Telegram Messages. Be sure to set `UseTelegram` in customize.h to enable receiving Telegram messages!

You will need to have Telegram installed on your phone or computer. Your account has a specific chatID associated with it. Google shows different ways to get your Telegram chatID, one popular method is to send a message of /getid to @IDBot. The ID shown needs to be entered into customize.h as either:

```
  #define CHAT_ID     "999999999"       // Primary Telegram user.
  #define ALT_CHAT_ID "88888888"        // Alternate Telegram user.
```

You will also need a bot token for Telegram bot. To create your own bot, send /newbot to @BotFather and follow the directions.



Enter the bot token into customize.h (Note: It is nearly impossible to type it correctly. Use copy/paste!)

```
Bot Bots[] = {
  {"7347294380:AAFSDNhGKA6ea3Ubke9-2Rq3IfHv07HDhg8 ",true,0}
};
```

As previously stated, use true if this bot will be receiving messages. You can make as many entries as you want, but keep in mind that each bot that receives messages will be periodically polled to see if any new messages have been received and this causes a load on the server and your local WiFi and uses bandwidth from your internet provider.

When a message has been received for a given bot, customTelegramRcvd() will be called. Parameter botNum indicates which of the bots received the message. Bots are numbered in the order you defined them in customize.h, beginning with 0. The parameter bot is the bot itself and can be used to respond, etc.

The internal structure of a Telegram message is defined as:

```
struct telegramMessage{
  String text;                // The actual UTF-8 text of the message.
                              // Commands begin with /. For example, /lightOff
  String chat_id;             // The chat_id of the receiving person.
  String chat_title;          // Title, for supergroups, channels and group chats
  String from_id;             // The chat_id of the person sending the message
  String from_name;           // The first name of the person sending the message
  String date;                // Date the message was sent (in Unix time)
  String type;                // "channel_post", "callback_query",
                              // "edited_message", or "message"
  String file_caption;        // If hasDocument, the message caption
  String file_path;           // If hasDocument, the file path of the document
  String file_name;           // If hasDocument, the file name of the document
  bool hasDocument;           // Does this message contain a document?
```

```
  long file_size;             // If hasDocument, the size of the document
  float longitude;            // Longitude as defined by sender
  float latitude;             // Latitude as defined by sender
  int update_id;              // Internal use. Message id of last received message.
  int message_id;             // Unique message identifier inside this chat
  int reply_to_message_id;    // If this message is a reply to a message,
                              // the id of the message being replied to
  String reply_to_text;       // If this message is a reply to a message,
                              // the text of the message being replied to
  String query_id;            // Only given when type = "callback_query".
                              // Then it is the message_id
};
```

The fields listed above are the information about a given message available to you.

```
  // See documentation for examples.
  bool msgHandled = false;
  switch(botNum){                   // Depending on which bot received the message
    case 0:                         // If it was bot 0
      // Sample:
      // if(tMsg.text == "/power"){
      //   sendPowerMsg(tMsg);       // Info (ie as WHO sent the msg) is in tMsg.
      //   msgHandled = true;
      // }
      break;
  }
  return msgHandled;
}
```

See Appendix B for a simple example customization which adds a water detection alarm that would notify you when water is detected.

Appendix C expands on the sample from Appendix B and contains information on how to add Telegram messaging to your project.

See Appendix D for an example customization which implements graphics using an ILI9488 TFT Display. This customization realizes the synergy of using both **The ESP32 S3 Webserver Project** software and hardware.

Appendix A – Creating a Custom Partition Scheme

Adapted from: #222 More Memory for the ESP32 without Soldering (How-to adapt partition sizes) - YouTube. Thanks to Andreas Spiess, "the guy with the Swiss accent" for the video and information!

1. Know the size of Flash the chip has. Most boards are sold with 4MB, better ones with 8MB and the max is 32MB. Note the correct maximum addressing space from this table:

2MB	1F FFFF
4MB	3F FFFF
8MB	7F FFFF
16MB	FF FFFF
32MB	1FF FFFF

2. Decide which options you want. Some memory is required at a fix size. How big do you think your sketches will be (Arduino tells you this when you compile)? Do you want to implement OTA so you can update over the air (this will require double the sketch space). Will you require SPIFFS/LittleFS to store files in the Flash?

OTA	Will require APP1 (see below)
APP1	If using OTA, there must be an APP1 sized identically to APP0
SPIFFS	Required for storing files in Flash (SPIFF or LittleFS)

3. Create a csv file in `C:\Users\Yourname\AppData\Local\Arduino15\packages\esp32\hardware\esp32\2.0.9\tools\partitions` (2.0.9 is version and may differ)
   1st Line: `# Name, Type, SubType, Offset, Size, Flags`
   2nd Line: `nvs, data, nvs, 0x9000, 0x5000,`
   3rd Line: `otadata, data, ota, 0xe000, 0x2000,`
   Last Line: `coredump, data, coredump,0x7F0000,0x10000,`

   On the last line, what is shown in the above sample corresponds to an 8MB Flash. Notice that the 7F corresponds with 7F from the table in step 1 which defines the Flash size. Thus, for a 4MB Flash, the 7F would be 3F instead.

   The rest is trickier. The above lines have already allocated the first and last 64KB, 128KB total. You get to divide the rest. If you are using OTA, you will need equal amounts of space for both APP0 and APP1. Without OTA, there will be no APP1. The following table shows common memory amounts you may want to allocate.

MB	Decimal	Hex
1	1,048,576	10 0000
1.25	1,310,720	14 0000
1.5	1,572,864	18 0000
1.75	1,835,008	1C 0000
2	2,097,152	20 0000
3	3,145,728	30 0000
4	4,194,304	40 0000

   Assuming you want to allocate 1.25MB for your sketch AND are using OTA, add the following two lines after the OTA line and before the coredump line:

   app0, app, ota_0, 0x10000, 0x140000,

app1, app, ota_1, 0x150000,0x140000,

(0x140000 is 1.25MB, app1 begins at 0x150000 because app0 began at 0x10000 and was allocated 0x14000 of space.)

If you are using SPIFFS or LittleFS, add another line after the app lines and above the coredump line. SPIFFS should be allocated the entire space between the apps and the coredump. In our examples above, app1 started at 0x150000 and was allocated 0x140000 of space so SPIFFS would start at 0x290000 and have a length of (assuming the 8MB Flash in this example) 0x7F0000 – 0x290000 or 0x560000 (about 5.5MB) and the entry would be:

spiffs, data, spiffs, 0x290000, 0x560000,

4. If you are experiencing an error "Not enough space to save core dump!", you can increase the space allocated for a core dump by reducing the length of the SPIFFS allocation and decreasing the beginning of the core dump area as well as increasing its length. It is typically 0x10000 (64KB).

5. In `C:\Users\Yourname\AppData\Local\Arduino15\packages\esp32\hardware\esp32\2.0.9\boards.txt`, create menu entries for the partition you just designed. (2.0.9 is version and may differ)

   The partition is configured for each board. Assuming the ESP32S3 Dev Module, the three lines would be:

   esp32s3.menu.PartitionScheme.MB8OTA5MBSPIFF=**8M Flash(1MB App, OTA, 6MB SPIFFS)**
   esp32s3.menu.PartitionScheme.MB8OTA5MBSPIFF.build.partitions=**8MBOTA5MBSPIFF**
   esp32s3.menu.PartitionScheme.MB8OTA5MBSPIFF.upload.maximum_size=**1310720**

   MB8OTA5MBSPIFF is simply a unique name (unique within esp32s3). The name must start with a letter!
   8M Flash(1MB App, OTA, 6MB SPIFFS) is the name that will show up in the partition scheme of the IDE.
   8MBOTA5MBSPIFF is the name of the csv file created in step 3 above.
   1310720 is actually the decimal representation of 0x140000 which was previously determined to be the size of APP0 (and possibly APP1 if using OTA). Refer to the table in step 3.

If Arduino is open, you may need to close it and reopen it for the new partition scheme to become available.

Appendix B – Sample Customization For A Water Detection Alarm

This example will:

- Utilize a digital pin to detect water. The pin is usually pulled high and goes low when shorted by water. Pin 39 will be used.
- Utilize an active piezo buzzer attached to pin 40 to sound the alarm. It buzzes when the pin is high.
- Utilizes an ISR (Interrupt Service Routine) to ensure that even temporary water will be detected, not just if it happens to be there when we get around to looking for it.

This example requires no changes or configurations in customize.h.

In custom.ino

Global section: Add #defines for the two pins and a bool variable that water has been detected.

```
#define waterDetect 39
#define waterAlarm  40
bool volatile waterDetected  = false; // Variables used in ISR must be volatile!
bool alarmOn = false;      //Var to track alarm. Global because in several routines
```

ISR section: Add the ISR to set waterDetected = true when the interrupt occurs.

```
void IRAM_ATTR ISR_water(){waterDetected = true;}
```

Setup section: Set up the pins. Ensure the buzzer is OFF immediately as the server starts. Setting up the waterDetect pin as an input and attaching the interrupt doesn't have to be done immediately.

```
void customEarlySetup(){
  pinMode(waterAlarm, OUTPUT);
  digitalWrite(waterAlarm, LOW);
}

void customLateSetup(){
  // NOTE: Attach an external 10K resistor to pull this pin to 3.3V!
  pinMode(waterDetect, INPUT);
  // ISR_water will be called when waterDetect starts to go low
  attachInterrupt(waterDetect,ISR_water, FALLING);
}
```

Loop section: Turn the piezo buzzer on/off depending on waterDetected. This is a high priority task and doesn't take long so it should be done even if other processing took place.

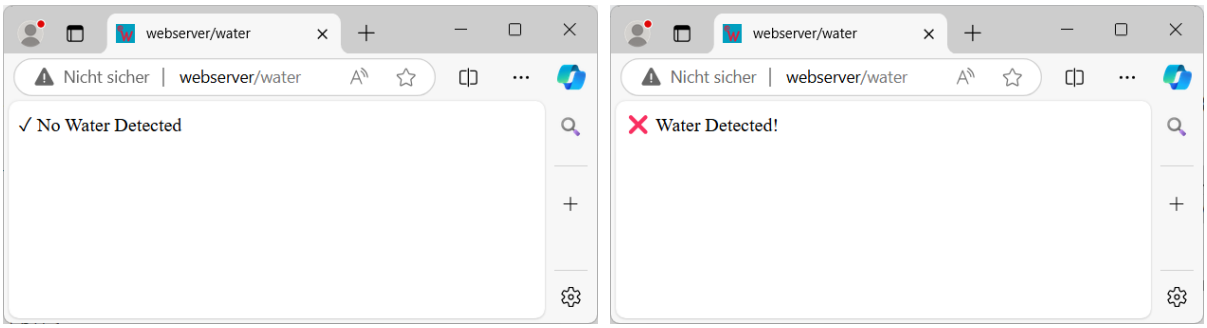
```
bool customLoop(bool didOne){

  bool customDidOne = false;        // Set to true if any custom processing is done

  if(alarmOn                        // Alarm is on
  && (!digitalRead(waterDetect)){   // but water is no longer there
    digitalWrite(waterAlarm, LOW);  // Turn piezo buzzer OFF
    waterDetect = false;            // Reset waterDetect so we see it next time
    alarmOn = false;                // Remember that the alarm is OFF
    customDidOne = true;            // We did some processing
  } else if((!alarmOn)              // Alarm is not on
  && waterDetect){                  // but water was detected
    digitalWrite(waterAlarm, HIGH); // Turn piezo buzzer ON
    alarmOn = true;                 // Remember that the alarm is ON
    customDidOne = true;            // We did some processing
```

```
  }

  if(didOne || customDidOne)          // The rest of this is no priority
    return customDidOne;              // So get out of here

  // Low priority processes go here

  return customDidOne;                // Let WebServer know if we did a process
}
```

Because this is, after a webserver, let's add a simple webpage that displays the alarm status (for when you're not close enough to hear that buzzer). We will access that webpage by calling up webserver/water in our browser.

Webpage section: Register the /water webpage to display results which will depend on if water has been detected or not.

```
void customPages(){
  // Register the webpage "/water"
  server.on("/water", HTTP_GET, [](AsyncWebServerRequest *request){
    char alarmTxt[50];
    if(alarmOn || waterDetected){
      // Text will be an "X" symbol + "Water Detected!"
      strcpy(alarmTxt, "&#x274C; Water Detected!");}
    else {
      // Text will be a checkmark + "No Water Detected"
      strcpy(alarmTxt, "&#x2713; No Water Detected");}
    AsyncWebServerResponse *response =
      request->beginResponse(200, "text/html", alarmTxt);
    request->send(response);
  });
}
```



Appendix C –Telegram Messaging

In Appendix B, we created a water detection system that sounds an acoustic alarm and provides a webpage that shows if water has been detected or not. That's ok, but we aren't always near the acoustic alarm and we don't want to walk around constantly refreshing our webpage either.

The ESP32 S3 Webserver Project software provides for Telegram messages, so let's implement some Telegram messaging that notifies us of the alarm. While we're at it, we'll add a sump pump that can manually be turned on/off via Telegram messages.

The server will send a message whenever water is detected and when water is no longer detected. That same message can be manually requested by the user by sending a message with "/water" in it to the server. The messages will have two buttons to turn the sump pump on/off.

In customize.h, you will need to have set `UseTelegram` to `TelegramSendReceive` so that we can both send and receive Telegram messages. You will have to have entered your bot token (from Telegrams bot father when you created the bot) and your chat id (that identifies YOU as the Telegram user). You will also have to define how often Telegram should be checked for new messages. (Don't set this too low! It does cause a certain load on the server and your local WiFi and will use badwidth from your Internet Provider!) Telegram messages will be ignored if they are not from one of the two chat ids defined in customize.h! There are a lot of people out there trying to break into your bot and this provides a certain level of security.

```
  Bot Bots[] = {
    {"9999999999:AAA9aABc_dZZaaZ9aZzA9z9A9abC9z-Z9aa",true,0}
  };

  #define CHAT_ID     "888888888"       // Primary Telegram user.
  #define ALT_CHAT_ID "999999999"       // Alternate Telegram user.

  #define TGRECEIVEFREQ 60              // How many seconds between polling
```

In custom.ino

Global section: Add another #define for the pump pin and a bool variable for pump status

```
#define waterDetect 39
#define waterAlarm  40
#define waterPump   41
bool volatile waterDetected  = false; // Variables used in ISR must be volatile!
bool alarmOn = false;      //Var to track alarm. Global because in several routines
bool pumpOn  = false;      // Var to show if pump on/off.
```

Setup section: Set up the additional pin. Ensure the pump is OFF immediately as the server starts.

```
void customEarlySetup(){
  pinMode(waterAlarm, OUTPUT);
  digitalWrite(waterAlarm, LOW);
  pinMode(waterPump, OUTPUT);
  digitalWrite(waterPump, LOW);}
```

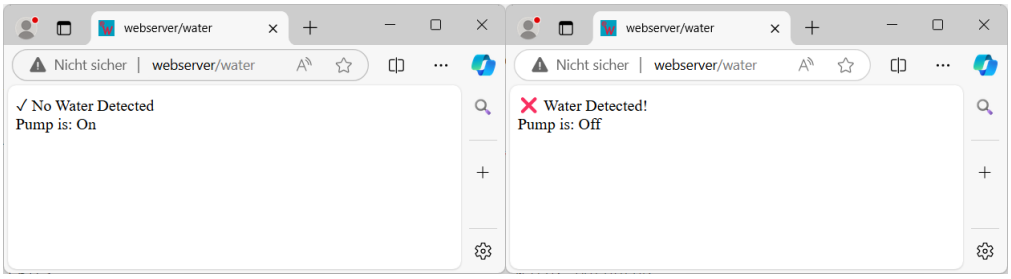
Loop section: Send messages when water is detected and when problem is solved

```
bool customLoop(bool didOne){

  bool customDidOne = false;        // Set to true if any custom processing is done
```

```
  if(alarmOn                          // Alarm is on
  && (!digitalRead(waterDetect)){     // but water is no longer there
    digitalWrite(waterAlarm, LOW);    // Turn piezo buzzer OFF
    waterDetect = false;              // Reset waterDetect so we see it next time
    alarmOn = false;                  // Remember that the alarm is OFF
    sendWaterMsg();                   // Notify user that problem was resolved
    customDidOne = true;              // We did some processing
  } else if((!alarmOn)                // Alarm is not on
  && waterDetect){                    // but water was detected
    digitalWrite(waterAlarm, HIGH);   // Turn piezo buzzer ON
    alarmOn = true;                   // Remember that the alarm is ON
    sendWaterMsg();                   // Notify user that water was detected
    customDidOne = true;              // We did some processing
  }

  if(didOne || customDidOne)          // The rest of this is no priority
    return customDidOne;              // So get out of here

  // Low priority processes go here

  return customDidOne;                // Let WebServer know if we did a process
}
```

Webpage section: We'll add the pump status to our /water webpage.

```
void customPages(){
  // Register the webpage "/water"
  server.on("/water", HTTP_GET, [](AsyncWebServerRequest *request){
    char alarmTxt[100];
    if(alarmOn || waterDetected){
      // Text will be an "X" symbol + "Water Detected!"
      strcpy(alarmTxt, "&#x274C; Water Detected!");}
    else {
      // Text will be a checkmark + "No Water Detected"
      strcpy(alarmTxt, "&#x2713; No Water Detected");}
    // Add in pump status
    strcat(alarmTxt, "<br>Pump is: ");
    if(pumpOn)
      strcat(alarmTxt, "On");
    else
      strcat(alarmTxt, "Off");
    AsyncWebServerResponse *response =
      request->beginResponse(200, "text/html", alarmTxt);
    request->send(response);
  });
}
```



Telegram section: Add handling of received Telegram messages

```
bool customTelegramRcvd(int botNum, UniversalTelegramBot bot, struct telegramMessage tMsg){
  bool msgHandled = false;
```

```
  switch(botNum){                          // Depending on which bot received the message
    case 0:                                // If it was bot 0
      if(tMsg.text == "/water"){           // User wants a water status
        sendWaterMsg();                    // Send the water status message
      }
      if(tMsg.text == "/pumpOn"){          // User wants to turn pump on
        digitalWrite(waterPump, HIGH);     // Turn pump ON
        pumpOn = true;                     // Remember that pump is on
      }
      if(tMsg.text == "/pumpOff"){         // User wants to turn pump off
        digitalWrite(waterPump, LOW);      // Turn pump OFF
        pumpOn = false;                    // Remember that pump is off
      }
      break;
  }
  return msgHandled;
}
```

This code calls a function sendWaterMsg() which you can put at the bottom of the custom.ino. This function will handle sending the message to the user.

```
void sendWaterMsg(){
  char msgTxt[100];              // Variable for text of message
  int numBtns = 2;               // Msg will have two buttons
  TelegramBtn btns[numBtns] = {  // Define the two buttons
    {"Pump On", "/PumpOn"},      // Says "Pump On", sends "/PumpOn"
    {"Pump Off", "/PumpOff"}     // when pressed. 2nd button is similar
  };
  // Build a text message based on water detection
  if(true || alarmOn || waterDetected){
    strcpy(msgTxt, "Water Detected!");}
  else {
    strcpy(msgTxt, "No Water Detected");}
  // Add pump status to message
  strcat(msgTxt, "\nPump is: ");
  if(!pumpOn)
    strcat(msgTxt, "On");
  else
    strcat(msgTxt, "Off");
  // Send Telegram Msg with keyboard (the 2 buttons)
  sendTelegramMsgKbd(0, tgChatID, msgTxt, btns, numBtns);
}
```

Appendix D – Sample Customization To Add Graphics

The ESP32S3 Webserver Project hardware was specifically designed so that the PC board could optionally be attached to the back of a 3.5 inch TFT display which is, in my case, equipped with an ILI9488 graphics driver. This example will add a graphic display to the water detection example shown in example B and C.

There are no required settings in customize.h that affect graphics.

In custom.ino

Library section: Add a graphics library.

```
#include <TFT_eSPI.h>
#define GFXFF 1
```

Note that this particular library requires some customization of its' own to define which pins are used to control the display. Those customizations are made in user_setup.h in the folder where the library is. The pins used for **The ESP32 S3 Webserver Project** hardware were all listed in the hardware section of this document, but for your convenience here is a summary of the changes to make to the TFT_eSPI library in the user_setup.h file.

```
#define ILI9488_DRIVER // TFT uses ILI9488 graphics chip
#define TFT_MOSI 11    // Standard SPI MOSI pin
#define TFT_MISO 12    // Standard SPI MISO pin
#define TFT_SCLK 13    // Standard SPI CLK pin
#define TFT_CS   10    // Chip select control pin
#define TFT_DC   16    // Data Command control pin
#define TFT_RST   7    // Reset pin
#define TFT_BL   17    // LED back-light
#define TOUCH_CS  6    // Chip select pin (T_CS) of touch screen
```

Global section: Instantiate the graphic display.

```
#define waterDetect 39
#define waterAlarm  40
#define waterPump   41
bool volatile waterDetected  = false; // Variables used in ISR must be volatile!
bool alarmOn = false;      //Var to track alarm. Global because in several routines
bool pumpOn  = false;      // Var to show if pump on/off.
TFT_eSPI tft = TFT_eSPI();    // Invoke TFT_eSPI library with default width and height
```

Setup section: Initiate the graphic display

```
void customEarlySetup(){
  pinMode(waterAlarm, OUTPUT);
  digitalWrite(waterAlarm, LOW);
  pinMode(waterPump, OUTPUT);
  digitalWrite(waterPump, LOW);
  initGraphics();
}
```

Loop section: Update the graphic display every 15 seconds

```
bool customLoop(bool didOne){

  static unsigned long lastDisplay = -15000;  // last display update
  bool customDidOne = false;      // Set to true if any custom processing is done.
```

```
  if(alarmOn                          // Alarm is on
  && (!digitalRead(waterDetect))){    // but water is no longer there
    digitalWrite(waterAlarm, LOW);    // Turn piezo buzzer OFF
    waterDetected = false;            // Reset waterDetected so we see it next time
    alarmOn = false;                  // Remember that the alarm is OFF
    sendWaterMsg();                   // Notify user that problem was resolved
    customDidOne = true;              // We did some processing
  } else if((!alarmOn)                // Alarm is not on
  && waterDetect){                    // but water was detected by the interrupt
    digitalWrite(waterAlarm, HIGH);   // Turn piezo buzzer ON
    alarmOn = true;                   // Remember that the alarm is ON
    sendWaterMsg();                   // Notify via Telegram that there is a problem
    customDidOne = true;              // We did some processing
  }

  if(didOne || customDidOne)          // The rest of this is no priority
    return customDidOne;              // So get out of here

  if((millis() - lastDisplay) > 15000){  // Update display every 15 seconds
    displayStatus();        // Update display
    lastDisplay = millis(); // Remember when display last updated
    customDidOne = true;    // Remember that we did processing
  }
  return customDidOne;      // Let WebServer know if we did a process
}
```

This code calls functions initGraphics() and displayStatus() which you can put at the bottom of the custom.ino.

This function will handle initializing the graphic display.

```
void initGraphics(){
  tft.begin();                        // Initialize TFT
  // By default the TFT_eSPI Library will turn the Backlight on.
  // Change the pin to analog and dim the screen
  pinMode(TFT_BL, OUTPUT);
  analogWrite(TFT_BL, 128);
  tft.setRotation(3); // 0&2 are portrait, 1&3 are landscape. If 0/1 is upside down use 2/3
}
```

This function will handle displaying the status on the graphic display.

```
void displayStatus(){
  char buff[100];
  int h = 29;                             // Height of a single line of text
  tft.fillScreen(TFT_BLACK);              // Clear the screen to black
  tft.setTextDatum(BL_DATUM);             // Bottom/Left oriented text
  tft.setFreeFont(&FreeSansBold12pt7b);   // Set 12 point, bold, Sans font
  tft.setTextColor(TFT_WHITE, TFT_BLACK); // Text is white on black

  tft.drawString("Water:", 0, h, GFXFF);
  if(alarmOn || waterDetected)
    tft.drawString("Detected!", 100, h, GFXFF);
  else
    tft.drawString("Not detected", 100, h, GFXFF);
  tft.drawString("Pump:", 0, h*2, GFXFF);
  if(pumpOn)
    tft.drawString("On", 100, h*2, GFXFF);
  else
    tft.drawString("Off", 100, h*2, GFXFF);
}
```