Tutorial on Using eByte E220 with ESP32

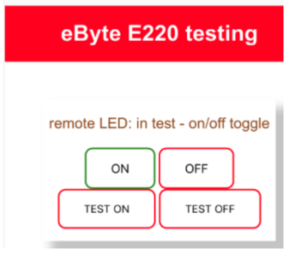
by Joe Margevicius **rev 1.0**

Table of Contents

Introduction

This is a tutorial about a very low power, battery-operated, remote ***receiver*** (a tutorial nothing like the great ones what Sara and Rui of Random Nerd Tutorials do, but since this topic has been bantered around, I know this forum is a good place for this kind of help)

As an incentive for doing this tutorial, at the very end is the code and hardware explanation for a *remote receiver LED* controlled by a *local transmitter* with serving up this web page:



So, building a remote low power battery-operated ***transmitter*** is not so hard -- the transmitter is asleep most of the time, and when programmed to wake up via an ultra-low-power real-time-clock, it reads sensor information (for example), and sends it to a home/office-based AC powered receiver, then goes back to sleep - a very low energy situation.

However, building a remote low power battery-operated ***receiver*** is much more complicated because, well, the receiver doesn't know when you'll talk to it, so it has to be alert all the time, draining any battery. The good news is that there are commercially available solutions using a bare-bones modem+microcontroller, both of which can be put to sleep. Both wake up for just milliseconds every second (or more), and listen for possible packet communication from the AC-powered transmitter. Basically, when you want to communicate with it, the home-based AC powered transmitter sends out a very, very long preambled packet (i.e. more than 1 second of preamble). When the receiver wakes up for that millisecond, it senses this and then fully wakes up and "pays attention". Otherwise the modem + microcontroller spend 99.9% of their time asleep, in very low power mode. This method is called Wake On Receive, or **WOR**. I have measured 20 uAmp current draw by the receiver (the modem+microcontroller in WOR mode +ESP32 microcontroller in sleep mode).

A low overhead, bare-bones, low power radio system is required. Unfortunately, Expressif's ESP32 microcontroller chip is not appropriate as a complete solution for this. It can be put to sleep, but it only internally supports a WiFi "radio" ... WiFi is terribly power-hungry with a seriously complex communication protocol because it has to "fight a lot" just to communicate in a very, very busy environment of cell phones, laptops, the neighbor's TV, etc, etc, etc -- "it's a jungle out there!!!" ... The good news is that after many years of development, WiFi is now robust, reliable, ubiquitous ... but power hungry. So, using the ESP32 chip alone for this isn't a good choice for a battery-operated receiver application, aside from the issue of needing a router or other network gear somewhere nearby.

There are at least 2 other solutions on the market that advertize a solution - they use proprietary very low overhead modems (LoRa = LOng RAnge spread-spectrum RF), and incorporate software (i.e. firmware in a flash memory) that can do implement the very low power Wake On Receive.

The chips used, incorporated into modules are: Texas Instruments' CC1310 modem+microcontroller chip, and Semtech's LLCC68 modem-only chip - both have internal low-power radio support and are sleepable. Much like the "dev" boards available to use Espressif's ESP32 chip, there are 2 other module companies that incorporate these chips into a module. They are eByte and Radio-Controlli.

For the TI chip, the company Radio-Controlli out of Italy has attempted to build a stand-alone WOR device (but from my experience, their WOR software is very unreliable and their unit is really a demo version in need of more work). For the Semtech chip, the company eByte out of China built a module with the modem chip, adding serious high-frequency antenna support, *a microcontroller*, and wide-range voltage input support -- see photo below. This module however *needs yet another microcontroller*, and using the ESP32 is a natural since it can be put to sleep also, even if it "costs" a tiny bit more power while asleep.

A web server application using the eByte and the ESP32 has been discussed a lot by William Lucid in this RNT blog. Although it is an interesting application, I found the project way too complex to get a good understanding of the relatively simple eByte unit. Further, use of a definitions file (i.e. a .h file) by another web blogger - Renzo Mischiante - makes it even more difficult to see the "forest for the trees". A good understanding is helpful for solving bugs and *allowing for creative uses*.

The intention of this tutorial is to get a basic understanding of the transmit/receive function of the eByte for low-power applications.

Below is a photo of the eByte E220 900Mhz module. The chip on the left is the RF switch; the one in the center is the Semtech LLCC68 modem; the one on top right is the Vin to 3.3V regulator; the next down is a CX32L003 microcontroller; below the modem is the 32 Mhz crystal oscillator that supports the modem and controller. The I/O pins are to the right - you'll notice resistors in between them and the microcontroller.

This module, although it has a nice 32bit ARM microcontroller inside, still needs an external microcontroller to manage it. So, using the ESP32, or a Raspberry Pi is a necessary component. I use the ESP32s in this discussion. They are sleep-able, and as a "dev" board, are easily programmed via the USB port. Of course they need that (bulky) C++ code. An alternative for the always-awake transmitter is to use the Raspberry Pi and the more sensible python code, but that's another project (The link to a RiPi based transmitter system for anyone interested is given later).

What follows is a tutorial based on the eByte E220 product and 2 ESP32 "dev" boards. In the end, you will have 2 units, one is labeled the Tx, and the other the Rx. Only the Rx eByte + ESP32 will go into sleep mode for very low power operation, while the transmitter will always be awake. For this tutorial, both the Rx and Tx units will be AC driven, since the ESP32s will be on a "dev" board which is USB-driven (i.e. driven via an AC "wall wart").

In this tutorial, I provide 9 program steps, providing a walk-through of using the 2 ESP32 dev boards each with their eByte E220 modules. Initially I show the very simple 8 wire connections needed for each side, then incorporating test code called *testBasicE220Wiring* into a Visual Code Studio + PlatformIO project. After testing simple "Normal" Tx/Rx communication, then Rx ESP sleeping operation, then Rx WOR operation, I get to eventually a "very reliable" very low-power Receiver driven by a Transmitter, with an OLED to show errors.

Reliability means the following:
- waking up the ESP32 reliably with a 2 packet transmission (the 1st packet is a "wakeup" packet, followed by the "message" packet)
- using an acknowledgement "handshake" system (i.e. an ACK) to have the receiver send back to the transmitter that it got the message properly.
- having the Tx re-transmit if it times out waiting for an ACK from the Rx, but for 2 cases:
  1) the Rx never sends the ACK
     1) the Rx never wakes up
     2) or the Rx misses the "wakeup" packet but rather wakes on the 2nd (message) packet. It then waits for another message packet, not a wakeup packet (so just the message packet needs to be resent).
  2) the Rx sends the ACK but the Tx doesn't see it.

Building in code to accommodate these cases, communication can be guaranteed. In a real application, it would be critical to have this. For example, to make sure the load is turned off and not draining the battery by staying on, or a water valve turned off and not staying on wasting water. In my testing, errors happen at a rate of about 1/1000 transmissions, which sounds like a little bit, but it's random and well, if it happens and is not dealt with, the system hangs.

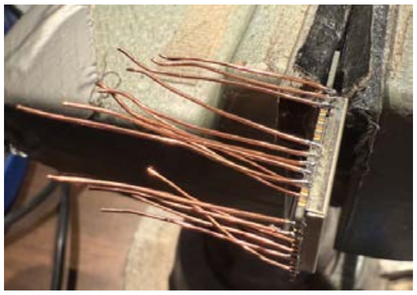
I will assume that you have downloaded these 9 C++ main.cpp code programs:
- *A-testBasicE220Wiring_main.cpp*
- *B-testNormalTxRx_main.cpp*
- *C-testRxESPwakeup_main.cpp*
- *D-testWORwithACK_main.cpp*
- *E-testWORandESPwakup_main.cpp*
- *F-testWORandESPwakeupWithOLED_main.cpp*
- *G-testReliableWORandESPwakeup_main.cpp*
- *H-Tx-SomethingUseful_main.cpp*
- *H-Rx-SomethingUseful_main.cpp*

~~~~~~~~~~~~~~~~~~~~~ **Practical Issues** ~~~~~~
~~~~~~~~~~~~~~~~~~~~~

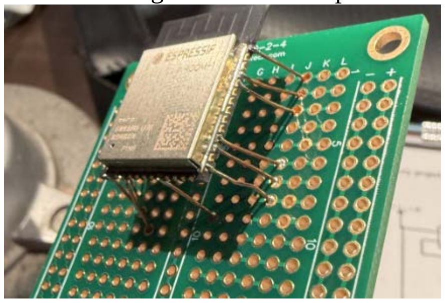
Dev boards versus ESP modules:
In a serious production-level low-power receiver, the "dev" boards wouldn't be used --- too much overhead with the USB, voltage converter, etc. **-** using the Espressif ESP32 *module for the Rx* would be the preferred solution, as shown in photos below. An ESP dev board is a power hog, and meant for, well, "development". However, with a solar panel, a dev board can be used, but care is needed to keep a chargeable USB battery from falling asleep (auto shut-down) due to low drain (one needs to use a "keep alive" device, like Lambda Nu's #CS-POWEREVER device**)**

Using an ESP32 module for the Rx is serious hardware tinkering as shown below !

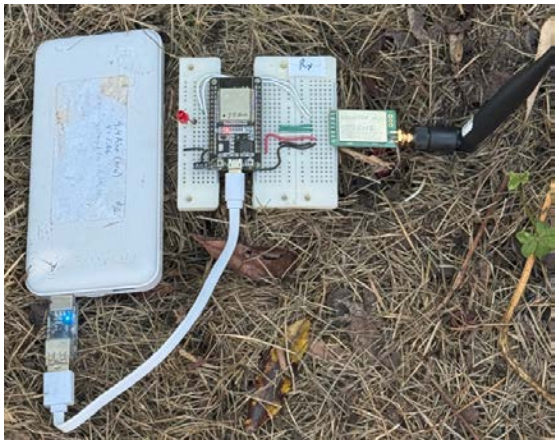
attaching thin #26 copper wires to select pins



attaching the ESP32 to a protoboard



Using an ESP32 Dev Board for the Rx is possible with a 5V cellphone backup battery, and a "keep alive" circuit *to keep the battery from going to sleep and dropping the voltage, (thinking that the "cellphone" is charged since so little current is drawn* **-** example: Lambda Nu's #CS-POWEREVER device).



Another issue: Power consumption by a "reliable" receiver has been carefully measured by me to be on the order of 20 microamps in the ESPasleep/eByte WOR mode (the module+eByte - not the dev board). For a 10Ahr Lithium battery, that would run for > 75 years ! It will be significantly less for an ESP Dev Board, but with Solar, it could last "forever".

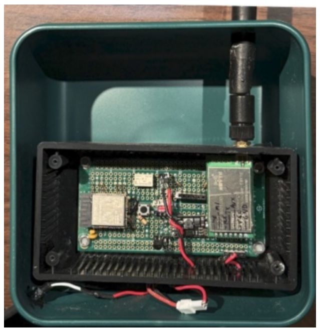
But of course, ***what does it do when it wakes up?*** What is to be controlled with this remote receiver? If it's turning on a big lamp or holding a big relay active, for example, the battery will drain out without a huge solar panel. And if the ESP/eByte system needs to stay on to control the power to the lamp, even more power is needed.

So some solutions:
1) Don't control constant high current devices (sorry, but remote well pumps will be a problem with batteries; use propane)

2) Control any lamps or valves using pulsed switches to turn them on/off (e.g. small latchable relays like Omron's G6KU-2P-Y DC3 for small loads like LEDs) ... or serious pulsed *big* relays for big loads like turning on that propane-driven well pump, or a gate lock or an irrigation valve (e.g. circuitry like implemented in Orbit's model 91566 irrigation valve - not simple to implement)
3) use big lithium batteries with solar
4) run an extension cord out to the receiver (and, with an extension cord, if WiFi is available, use that instead -- much more reliable!)

This tutorial uses ESP32 dev boards with USB power, not stand-alone ESP32 modules, and is meant as an intro to the eByte E220

Here is an example of a a low-power receiver, managing a low-power LED turned on twice a day (1 1/2 hr at sunrise; 1 1/2 hr at sunset) A 10 Ahr Lithium flat battery is under the black box; LED draws ~9ma at 9volts; unit operates ~ 4 months before recharge; forever with a solar panel.   (schematic and description of this project will be presented in a github link later).



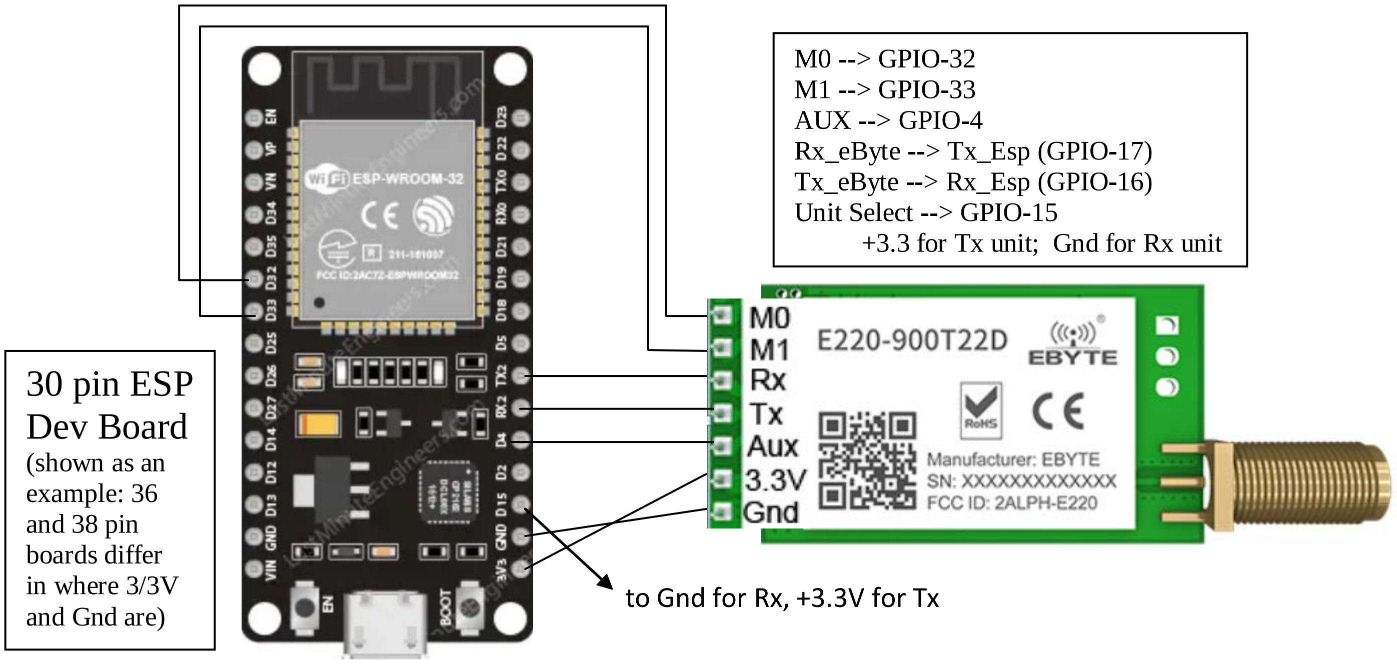
A - Assembly and Test a Tx and Rx unit

(*A-testBasicE220Wiring*)

A0 - Wiring up Tx and Rx systems

Parts needed:
2 ESP32 development boards (e.g. a DOIT ESP32 DEVKIT) ; 30 pin or 36 pin
2 eByte E220 Long Range (LoRa) modules: (e.g. the 900 Mhz E220-900T22D)
2 antennas (900Mhz)
2 USB cables to connect ESP boards to the computer
Bread boards (e.g. - see photos; 3 of the BB400 boards, cut as per photo .... or whatever allows mounting an ESP with an eByte)
#24 wire or jumpers

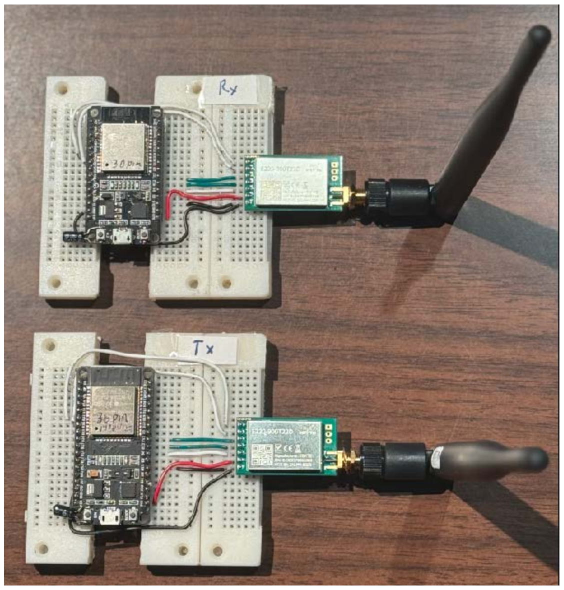
Wiring Diagram for a 30 pin dev module (36 and 38 pin modules have Gnd and 3.3V on different pins):



M0 --> GPIO-32
M1 --> GPIO-33
AUX --> GPIO-4
Rx_eByte --> Tx_Esp (GPIO-17)
Tx_eByte --> Rx_Esp (GPIO-16)
Unit Select --> GPIO-15
+3.3 for Tx unit; Gnd for Rx unit

30 pin ESP Dev Board
(shown as an example: 36 and 38 pin boards differ in where 3/3V and Gnd are)

my setup using "sliced" protoboards (Rx has a 30 pin ESP32 dev module, and Tx has a 36 pin ESP32 dev module - use whatever)



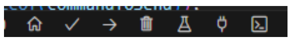
A1 - Software setup - using Visual Studio Code:

Because of variations in how people setup VSC, I provide only the main.ccp file for each step of this tutorial ... use this to buid a VSC project, as explained below. This writeup will lead you through the procedure to test at each stage (although going through each stage is not necessary if you just want to jump to the last section).

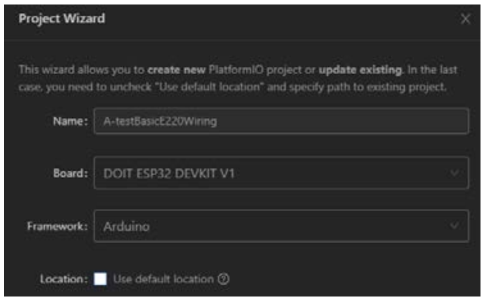
I assume familiarity with VSC (if not, check out Random Nerd Tutorial's xxxxxx)

As an example of how to construct a project in VSC, here is what to do for the *A-testBasicE220Wiring* project (the 1st one)

1) with VSC up, poke the "House" icon at the bottom of the screen, then hit "New Project"



- fill in the boxes:
    Name = ***A-testBasicE220Wiring***
    Board = DOIT ESP32 DEVKIT V1
        (to easily find this, type ESP32 in the Board box, and scroll down)
    Framework = Arduino
    Location = (2 options:)
      -Default (the "default place" = C:\Users\your_username\Documents\PlatformIO\Projects\)
      -Your own place: unselect Default and navigate to the directory where you'd
        put projects (this one could be called eByteTutorial)



Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name: A-testBasicE220Wiring

Board: DOIT ESP32 DEVKIT V1

Framework: Arduino

Location: ☐ Use default location

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Right called *A-testBasicE220Wiring*
 - go to the src line, then select *main.cpp* -- a simple template file will appear

3) open the *A-testBasicE220Wiring_main.cpp* file in something like notepad, or Sublime, or whatever you use for text editing (Word works too). Select the entire content of that file, and copy it.

4) back in VSC with the *main.cpp* template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the windows (Ctrl-V) .... you should now have the entire *testBasicE220Wiring* code in VSC. ... Ctrl-S to save

5) Now on the Left, click on the platformio.ini file, and at the end, add this statement: *monitor_speed = 115200*
  the code in the platformio.ini file should now look like this:

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

6) Finally, create a workspace file that will help you in the future locate all these projects:
 - go up to File/Save Workspace As, and type in a name like *eByteE220 Tutorial*
    (this allows you to work in other workspaces (e.g. *RNTutorials examples*),
      and switch back to eByteE220 Tutorial by going to *File/Open Workspace From File*)

You are now ready to test the Tx and Rx modules.

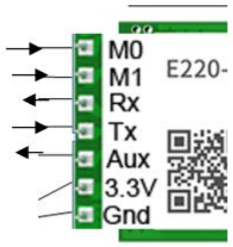
A2 - Code Description:

Overview of the module:

The eByte E220 module, of Chengdu Yibaite Electronic Technology Co., China., is based on the LLCC radio transceiver (i.e. modem) chip of Semtech, California. Chengdu added a microcontroller, firmware for WOR, the complex high frequency antenna support, local oscillator circuitry, and wide-range voltage input support.

The module has 4 modes of operation, selected by the 2 external *input* pins M0 and M1 of the module:

- Normal mode	[M1 = 0, M0 = 0]
- Tx WOR mode (Wake on Receive)	[M1 = 0, M0 = 1]
- Rx WOR mode	[M1 = 1, M0 = 0]
- Deep sleep mode	[M1 = 0, M0 = 1]



The module allows access to 8 eight-bit registers, accessed by the 2 external Rx/Tx UART pins of the module:

- High & Low module address registers (e.g. module address = 0x0102)
- Reg0, 1, 2, 3 -- configure operation of the module
- High & Low encryption support registers

When configuring the chip, and/or changing modes, an external *output* "Ready" pin, called AUX is provided (ready when HI).

These 5 pins, along with power and ground pins, are the interface in this module.

The Code for *testBasicE220Wiring*:

This code basically writes to the registers, and reads its auto-response - this tests all wiring, and if the wiring is done correctly, the responses will match what was written.

One option in software for dealing with the modes and registers of the module is to use the LoRa_E220.h file generated by Renzo Mischiante. Personally, I find his file and code incredibly complex for such a relatively simple chip, so I don't access any .h file other than the standard Arduino.h file. Instead I provide that info in this program.

The first part of the code defines the GPIO pin assignments for the ESP32. One is forced to use GPIO 16,17 for the 2nd UART (called Serial2) in the ESP32 (the 1st one, called Serial1, is used to download code from the USB port). The assignments for M0, M1 and AUX must be to pins supporting the RTC memory (this is for sleep functionality; more in the test program for ESP32 & eByte sleeping).

Next is a long list (almost 100 lines !) of all the register addresses, contents (aka "switches"), modes, and commands. This is a complete listing for the chip, *for reference purposes mainly*. In later test code, only a few of those declare statements are used (looks overwhelming, but consider it a reference for the chip, allowing you to understand all the things that can be selected, if you don't like the defaults).

Example: Here is a listing for *Register 0* -- although the chip can run at many UART rates/parity, I use the defaults:

```
  // REG0 content: "Switches"
              // NOTE: MUST OR these 3 categories e.g. (UART_RATE_9600 OR AIR_RATE_2400 OR EBYTE_UART_PARITY_8N1)
    // UART speeds - bits/second
#define EBYTE_UART_RATE_1200        0x00
#define EBYTE_UART_RATE_2400        0x20
#define EBYTE_UART_RATE_4800        0x40
#define EBYTE_UART_RATE_9600        0x60     // default
#define EBYTE_UART_RATE_19200       0x80
#define EBYTE_UART_RATE_38400       0xA0
#define EBYTE_UART_RATE_57600       0xC0
#define EBYTE_UART_RATE_115200      0xE0
    // UART 9 bit format
#define EBYTE_UART_PARITY_8N1       0x00     // default (see manual for other rarer parity options)
#define EBYTE_UART_PARITY_8O1       0x08
#define EBYTE_UART_PARITY_8E1       0x10
//#define EBYTE_UART_PARITY_8N1       0x18       // same as default
    // Tx/Rx RF Air rate - bits/second
//#define AIR_RATE_2400         0x00        // same as default -- was 300 in E22, but didn't work
//#define AIR_RATE_2400         0x01        // same as default -- was 1200 in E22, but didn't work
#define AIR_RATE_2400         0x02     // default
#define AIR_RATE_4800         0x03
#define AIR_RATE_9600         0x04
#define AIR_RATE_19200        0x05
#define AIR_RATE_38400        0x06
#define AIR_RATE_62500        0x07
```

As you can see, you really don't need all those #defines -- in this test code (*testBasicE200Wiring*), they aren't used at all -- they are for reference only.

Next is a declaration of an array to store eByte responses. When a register is written to, the eByte module writes to the memory, and then it reads that memory and automatically sends out the contents - a "response" to the writing.

Next is a write/read function for the eByte. *void eByteE220WriteReadMemory(byte REGtoWrite, byte contentsToWrite, int printResponse)*

- the printResponse variable is to allow printing (1) or not printing of the response (0). We will print out the response for this test, but future test programs may not clutter up the printing by printing these responses.

Again, the 4 modes of the eByte are as follows:

```
[M1 M0] = [0 0] for Normal mode
[M1 M0] = [0 1] for WOR Tx
[M1 M0] = [1 0] for WOR Rx
[M1 M0] = [1 1] for Configuration mode, and Sleep mode
```

The module should normally be in "normal mode", and when writing to a register, it must be put in Configuration Mode.
While in Config Mode, a write sequence would be something like this:

byte commandToSend[] = {WRITE_REG_CMD, REGtoWrite, 0x01, contentsToWrite};

where Write_Reg_CMD is defined above as 0xC1. 0x01 indicates writing to just 1 location.
example: *byte commandToSend[] = { C1 0 1 1 }* --- i.e. write to Reg0, 1 location (ie. 1 byte), and the contents are 0x1

Serial2 = UART2 = connected to the eByte, is used to send the message: *Serial2.write(commandToSend, sizeof(commandToSend));*
- the size of the message has to be defined in the Serial2 port transmission.

When a register is written to, the eByte writes to memory, and then it reads that memory and sends out the contents - a "response" to the writing. The next section of code wait for availability of a response, and then reads byte by byte - 4 bytes.

Finally, the chip is put back into Normal Mode.

In the Setup() section, the 2 UARTs are configured, and the GPIO's are configured.

In the Loop() section, the identification of Tx or Rx is made by reading the appropriate GPIO, and then the 8 registers are written to, and their response is read.

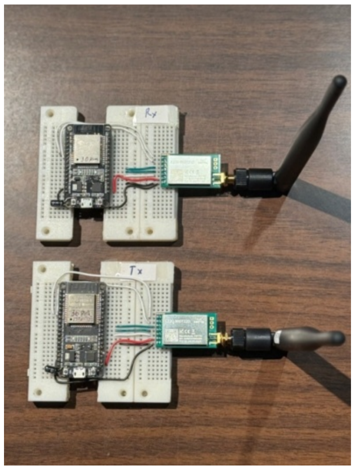
If the responses are what was written, the wiring is fine .... this is a good program to run later if you find problems in a more complicated project. You should get:

```
FYI -- wiring check: ...
this is a TRANSMITTER

now will write to 9 control memory locations, then read response

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG1 : C1 3 1 0  = response. Expect (C1 3 1 0)
REG2_RF_CHAN : C1 4 1 32  = response. Expect (C1 4 1 32)
REG3 : C1 5 1 0  = response. Expect (C1 5 1 0)
REG_CRYPT_H : C1 6 1 0  = response. Expect (C1 6 1 0)
REG_CRYPT_H : C1 6 1 0  = response. Expect (C1 6 1 0)

the end of a loop ... will wait 5 seconds before doing it again
```

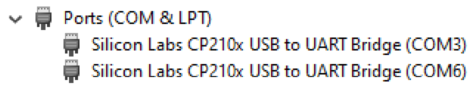
A3 - Testing the Wiring:

At this point, you should have the Tx and Rx modules wired up, and Visual Studio Code *A-testBasicWiring* project up. I will assume use of a Windows PC, (but using the MAC is only slightly different in its control keys)

Connect the 2 modules to your PC - assuming you have 2 USB ports (if not, you'll need a USB hub).
Here is my setup -- Rx using 30 pin ESP32 and Tx using 36 pin ESP32 (no difference in wiring - it's what I had)

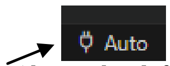


It's necessary to know what ports are now occupied. Go to Control Panel/Device Manager (or type in Device Manager in the Windows Start button at the bottom left). Under Ports, you'll see something like this:

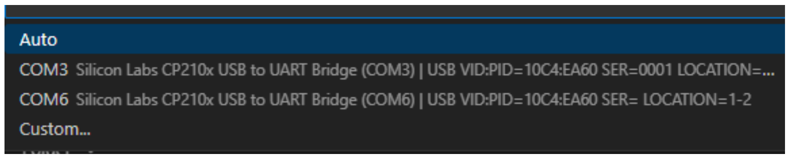

Ports (COM & LPT)
Silicon Labs CP210x USB to UART Bridge (COM3)
Silicon Labs CP210x USB to UART Bridge (COM6)

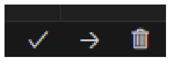
To know which one is the Tx port, remove the Rx USB, and observe the change in Device Manager.'

The code needs to be downloaded into each unit - the same code (the code detects whether it is a Tx or an Rx).



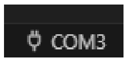
Click on the word "Auto" at the bottom of the screen (that is the default mode of selecting ports -- but you will want to know which module is being addressed, if only to know if something goes wrong). At the top will pop up a window indicating the ports available, like this:


Auto
COM3 Silicon Labs CP210x USB to UART Bridge (COM3) | USB VID:PID=10C4:EA60 SER=0001 LOCATION=...
COM6 Silicon Labs CP210x USB to UART Bridge (COM6) | USB VID:PID=10C4:EA60 SER= LOCATION=1-2
Custom...

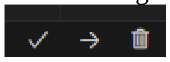


Select the port for the Tx module, then download the code by hitting the *forward arrow* at the bottom:
- if you don't have the capacitor on the boot button (or a more current ESP32 module), you'll have to hold the boot button down to initiate a download.

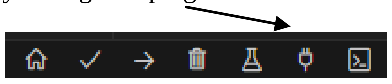
Assuming that was successfully loaded, change to the Rx port: the word "Auto" will be replaced by the Port being used, eg:



Hit that icon, and up at the top of the screen will again appear the ports available -- now select the Rx port, and download the code into that module using the foward arrow:.



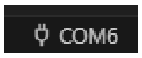
With the Rx connected, activate the *terminal* by hitting the "plug" at the bottom of the screen:



If the code is running properly, you should see the following display repeating every 5 seconds:

```
FYI -- wiring check: ...
this is a RECEIVER

now will write to 9 control memory locations, then read response

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG1 : C1 3 1 0  = response. Expect (C1 3 1 0)
REG2_RF_CHAN : C1 4 1 32  = response. Expect (C1 4 1 32)
REG3 : C1 5 1 0  = response. Expect (C1 5 1 0)
REG_CRYPT_H : C1 6 1 0  = response. Expect (C1 6 1 0)
REG_CRYPT_H : C1 6 1 0  = response. Expect (C1 6 1 0)

the end of a loop ... will wait 5 seconds before doing it again
```

If the responses are what are expected, all is wired up fine :--) ... if not, check the wiring.



Now switch to the Tx module by poking the Port icon at the bottom, and selecting the Tx USB port e.g.
- you should see the same display, except for this line:

```
FYI -- wiring check: ...
this is a TRANSMITTER
```

This completes this test --- come back to it if something doesn't work later in the tutorial --- it's a simple test of accessing the eByte.

B - Testing Normal Tx--Rx communication
(*B-testNormalTxRx*)

After successfully proving the wiring is fine, the next step is establishing communication between the Tx and Rx modules.

Building the project:
To do this, follow the procedure for Software Setup in the *testBasicE220Wiring* project, except when filling in the new project, use this:

Name = ***B-testNormal TxRx***
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called *B-testNormalTxRx*
- go to the src line, then select main.cpp -- a template file will appear

3) open the *B-testNormalTxRx_main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the *main.cpp* template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *B-testNormalTxRx* code in VSC. ... Ctrl-S to save

Again, make the addition of the final monitor_speed line to the platformio.ini code (click on it to open it):

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *testNormalTxRx*

~~~~~~

A note on a Send/Acknowledge protocol. The RF signal sent by either module is subject to corruption due to interference by other RF sources, and/or noise. In cases where it is ABSOLUTELY important that a transmission is correctly received, the receiver must check the expected contents, and if good, send back an ACKnowledgement to the transmitting module. ... examples would be a signal to turn off an irrigation valve - failure to do so will cause water to be wasted. or, failure to turn off a low-power light, will result in draining the battery.

I find that in the Normal Tx/Rx transmission (like this test), errors don't happen often, but without this in the more complicated and practical cases of WOR transmission, errors can occur and the system can hang without further code. When running the system in final Wake On Receive mode (presented in the last tests), an packet error rate of around 0.1% was measured by me (if it's critical that the receiver "get the message", then this ACK system is absolutely necessary). (If your connection to the cellphone tower were ~0.1 % packet error rate, you would look for another carrier! ... that's a really bad error rate in communication systems).

So, for the purpose of a building a robust Tx/Rx pair, using an ACK system is critical, and I introduce this Send/Ack protocol at this point in the testing, just to get used to it and understand how it performs.

~~~~~~

Code Description:
After the ESP32 pin assignments, there are the #define statements for the eByte module. Unlike in *testBasicE220Wiring*, we only need a few #define statements for the chip registers.

```
// eByte registers  --- see project testBasicE220Wiring for more complete info on registers of eByte
#define REG_MODULE_ADDR_H       0x00
#define REG_MODULE_ADDR_L       0x01

#define REG0                    0x02
#define EBYTE_UART_RATE_9600    0x60      // default
#define EBYTE_UART_PARITY_8N1   0x00      // default
```

```
#define AIR_RATE_9600                    0x04
#define REG0_Setup                       EBYTE_UART_RATE_9600 | EBYTE_UART_PARITY_8N1 | AIR_RATE_9600

#define REG2_RF_CHAN                     0x04
        // = RF channel select (# 0 to #80) --- RF Freq = 850.125Mhz + 1Mhz x Chan Number (i.e. chan 4  --> freq = 854.125Mhz

 // I/O commands
#define WRITE_REG_CMD           0xC0
```

Notice that the *REG0_Setup* variable is the result of "bitwise ORing" the 3 "switches" for that register.

The next set of code defines the message (packet contents) to be sent by the Tx, and the ACK sent by the Rx ("ACK" was chosen for convenience and clarity, but any word or group of words would work as well).

```
  // data to send by Tx
char TxMessage[] = "Hello ... how are you doing today?";
int TxMessageLength = sizeof(TxMessage);
  // data to send by Rx
char RxAck[] = "ACK";
int RxAckLength = sizeof(RxAck);
```

The function *void eByteE220WriteReadMemory(...)* is the same as in the *testBasicE220Wiring* project **--** sending out a packet, receiving the response, and printing that response if *printResponse* = 1).

Two further functions are defined: *initializeDevice()* and *blinkOnBoardLED()*
The initialize function is called in *setup()* code, and basically configures the eByte module registers.

- the module address: 0x0201
- REG0's 3"switches" - UART 9600baud, 8N1 parity, and 9600 bits/sec RF air rate
- the RF frequency as 854.125 Mhz

The *blinkOnBoardLED()* function is used to indicate a packet being sent out by the Tx, received by the Rx, and an ACK received by the Tx **-** it uses the blue LED on the ESP32 dev module.

Continuing with the code description, the *setup()* section will configure the 2 UARTS, the GPIO pins, and initialize the chip registers.

On entering the loop() (i.e. main) section of the code, a distinction is made between a Tx and an Rx.

For the Tx, it will immediately send out a " Hello ... how are you doing today?" packet, and then wait for an ACK reply. The statement: *Serial2.write(TxMessage, sizeof(TxMessage));* sends out the message, and the ESP32 onBoard blue LED is blinked 1x.

The Tx then waits for an ACK from the Rx, and on getting it, blinks the blue LED 3x.

The Tx then waits 4 seconds, and starts over with sending out the message again.

For the Rx, it will wait for a packet from the Tx, then check it's contents.
- the statement *if(strstr(dataPointer,"Hello ... how are you doing today?") != NULL*) will do the checking, letter by letter using the strstr() function (found in Arduino.h). If the comparison is a result that is not 0, the Rx will blink its ESP32 blue LED 2x, then send off the ACK, using this statement:

Serial2.write(RxAck, sizeof(RxAck));

The Rx then goes back to wait for a message from the Tx (who is sending out one every 4 seconds).

**~~~~~~**
Running the programs:

Loading the code:
Refer to the *testBasicE220Wiring* procedure to download the same *testNormalTxRx* code into the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~4 seconds.

Seeing the printouts:

To see the outputs, follow the procedure for selecting which USB port the VSC application is to select.

For the **transmitter,** you should see this printed out:

```
/***** this is what should appear in the TRANSMITTER terminal ***
 ... initializing eByte - write to 4 memory locations
C1 0 1 1  = response. Expect (C1 0 1 1)
C1 1 1 2  = response. Expect (C1 1 1 2)
C1 3 1 64  = response. Expect (C1 3 1 64)
C1 5 1 7  = response. Expect (C1 5 1 7)


Setting up the TRANSMITTER to send 1 message, then wait for ACK from receiver
now will transmit 35 bytes
message sent
..
ACK (received by Tx; sent from Rx after received message)

Setting up the TRANSMITTER to send 1 message, then wait for ACK from receiver
now will transmit 35 bytes
message sent
.
ACK (received by Tx; sent from Rx after received message)

*********/
```

For the **receiver,** you should see this printed out:

```
/******** this is what should appear in the RECEIVER terminal *****
 ... initializing eByte - write to 4 memory locations

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)

Setting up the RECEIVER to accept 1 message, then if good, send an ACK to the transmitter
...
packet that was sent by xmitter was: Hello ... how are you doing today?
packet just received by rcvr is:     Hello ... how are you doing today?
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx

Setting up the RECEIVER to accept 1 message, then if good, send an ACK to the transmitter
......
packet that was sent by xmitter was: Hello ... how are you doing today?
packet just received by rcvr is:     Hello ... how are you doing today?
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx

***********/
```

C - Testing Rx ESP32 wakeup by Normal Tx
(*C-testRxESPwakeup*)

After successfully demonstrating Tx/Rx reliable ACK'd communication, testing proceeds with putting the ESP32 to sleep and having the Tx wake it up.

Important note: I found that on wakeup of the ESP32 by an external signal (i.e. the eByte sending it a packet), the ESP32's Serial2 UART was not functional in time to read the contents of the packet. So, what works is to send a short "wakeup" packet (eg. the character "w"), then wait for the ESP32 Serial2 UART to be fully functional, then send the "message" packet. ***This requires attention to timing***, and is built into the code for this test.

~~~~~~

Building the project:
To do this, follow the procedure for Software Setup in the *testBasicE220Wiring* project, except when filling in the new project, use this:

Name = ***C-testRxESPwakeup***
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called *C-testRxESPwakeup*
 - go to the src line, then select main.cpp -- a template file will appear

3) open the *C-testRxESPwakeup_main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the main.cpp template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *C-testRxESPwakeup* code in VSC. ... Ctrl-S to save

Again, make the addition of the final monitor_speed line to the platformio.ini code (click on it to open it):

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *testRxESPwakeup*

~~~~~~

Code Description:
After the ESP32 pin assignments, there are the #define statements for the eByte module. They are identical to the statements in the *B-testNormalTxRx* .

The next set of code is also similar to the previous test code, except an additional array is needed to hold the "wakeup packet"

```
char wakeupPacket[] = "w";  // just a short packet to wakeup the ESP32
int wakeupPacketLength = sizeof(wakeupPacket);
```

The code will run repeatedly, like the previous test code. The Rx ESP32 is awaken repeatedly, every ~6 seconds, by the Tx in this code. So as to not have to keep configuring the eByte module again and again (it remembers its configuration, and doesn't need re-configure), it is helpful to identify the 1st time the chip is configured, using a "flag" stored in RTC Flash of the ESP32. This statement accomplishes that:

```
// info to store in flash for wake-up
RTC_DATA_ATTR bool firstTimeUp = true;  // start true to program the devices; then false for the Rx to process packet
```

This is not necessary (it's ok to repeatedly configure the eByte); it's a matter of "elegance" of the code (it can be left out).

The 1st 3 functions are the same as in the previous test code:
- *eByteE220WriteReadMemory()*
- *initializeDevice()*
- *blinkOnBoardLED()*

The new function is *goToSleepESP()*

```
esp_sleep_enable_ext0_wakeup(gpio_eByte_AUX,0); // Enable Ext0 as wakeup/reboot: Logic Level defined = 0 = interrupt; must do this after a reboot
 //When in deep sleep, the general GPIO hardware is turned off and float. Only the RTC part and RTC-GPIOs are on.
gpio_hold_en(gpio_eByte_M0); // keep M0 and M1 from changing (if they change, they cause AUX to go low, and this will wake up the ESP)
gpio_hold_en(gpio_eByte_M1);

Serial.println("\nRECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)");
esp_deep_sleep_start();  // ESP32 goes to sleep
```

As is commented, when the ESP32 goes to sleep, normal ***GPIO pins revert to default values, which are not what we want*** **-** the eByte module needs to continue with the mode selected by M0, M1. So *gpio_hold_en* functions are needed, as shown. (the holds will be disabled on wakeup to allow continued programming of the eByte module if necessary).

The AUX line-out from the eByte module goes LO when a packet is received, and the 1st line of the function defines that signal as the "wakeup line" for the ESP, specifying that a "0" on the line is the interrupt (AUX is normally HI).

The setup() code distinguishes between Tx and Rx.
For Tx, it configures the UARTs and the GPIO pins, and does the full *initialize()* function for the eByte module. (It later does code in the *loop()* section).

For Rx, the UARTs and GPIOs pins are configured **--** when the ESP32 wakes up, the GPIOs pins *must be reassigned*, and the UARTs reconfigured. (I think this necessary reconfiguration of Serial2 is the reason for it being sluggish about being able to read the wakeup packet; I experimented with putting the reconfiguration only after reset, but found that unreliable. For reliable communications, use the "wakeup packet" then "message packet" sequence as provided in the code).

In my experiments, the soonest a message packet can be sent after a wakeup packet is 125 mSec. **--** it takes the ESP32 at least 120 mSec to have the 2nd UART fully functional. **--** that's about 140 characters for RF air speed at 9600 Baud !

On the 1st run after reset, after the UART and GPIO configuration and the chip is initialized, the *firstTimeUp* flag is made false. Then the ESP32 is put to sleep

```
initializeDevice();  // configures address and channel; ends up in Normal mode
firstTimeUp = false;
Serial.print("hello from firstTimeUp -- going to sleep now");
goToSleepESP();
```

The ESP32 is now asleep, waiting for an interrupt from the eByte.

When the "wakeup" packet is received by the eByte, the ESP32 wakes up and proceeds into *setup()* again. This time, it does the UART/GPIO configurations, but not the initialization of the module (since *firstTimeUp* is now false).

It moves on, removing the holds on the M0 M1 output pins of the ESP32, and sits waiting for the "message packet" (the "wakeup packet" was received **--** it's what woke up the ESP32). It waits via the *while(!Serial2.available())* statement.

Just like in the *testNormalTxRx* project, the Rx finally gets the "message packet", checks it thoroughly, and sends off an ACK to the transmitter if it's good.

Then it goes back to sleep, waiting for another "wakeup packet" from the Tx.

For Tx, it continues past the Rx code, and enters the *loop(),* and sends off the "wakeup packet". It waits, critically, 150 mSec, and then sends the "message packet", blinking the onBoard blue LED 1x.

As in the previous *testNormalTxRx* project, the Tx waits for an Ack from the Rx. On getting it, it waits 3 seconds before starting the *loop*() over again, transmitting a "wakeup packet".

With the delays, the cycle repeats every ~6 seconds.

~~~~~~
~~~~~~

Running the programs:

Loading the code:
Refer to the *testBasicE220Wiring* procedure to download the *C-testRxESPwakeup_main.cpp* code in the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~6 seconds.

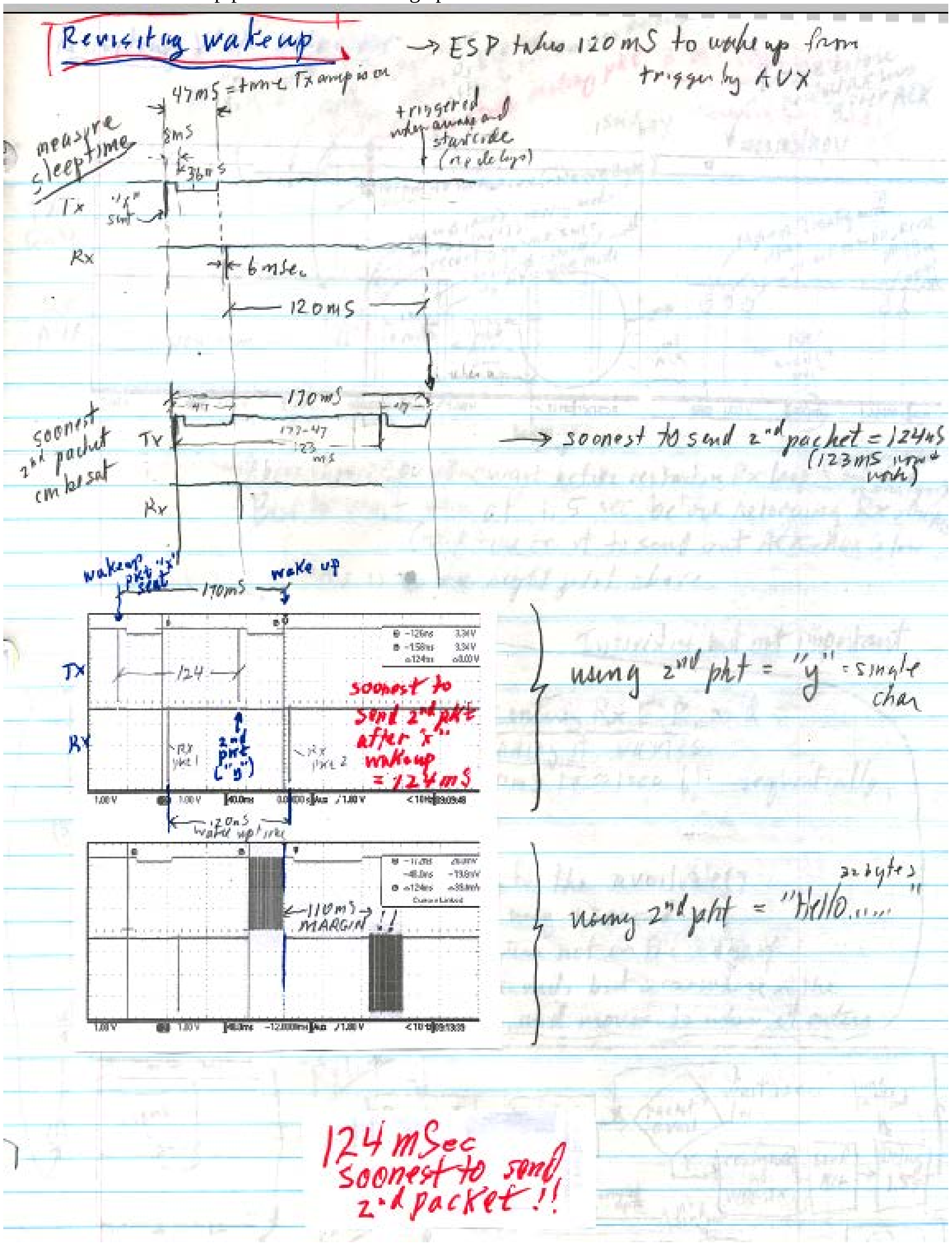
Seeing the printouts:
To see the outputs, follow the procedure for selecting which USB port the VSC application is to select.
For the **transmitter,** you should see this printed out:

```
/***** this is what should appear in the TRANSMITTER terminal ***
 ... initializing eByte - write to 4 memory locations
C1 0 1 1  = response. Expect (C1  1 )
C1 1 1 2  = response. Expect (C1  1 )
C1 3 1 64  = response. Expect (C1  1 d)
C1 5 1 7  = response. Expect (C1  1 )

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
..
ACK (received by Tx; sent from Rx after received message)

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
..
ACK (received by Tx; sent from Rx after received message)

*********/
```

For the **receiver,** you should see this printed out:

```
/******** this is what should appear in the RECEIVER terminal *****
 ... initializing eByte - write to 4 memory locations
C1 0 1 1  = response. Expect (C1 ␀ 1 ␁)
C1 1 1 2  = response. Expect (C1 ␁ 1 ␂)
C1 3 1 64  = response. Expect (C1 ␃ 1 d)
C1 5 1 7  = response. Expect (C1 ␅ 1 ␇)
hello from firstTimeUp -- going to sleep now
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)


RX JUST WOKE UP by wakeup packet; now to read message packet
.
packet that was sent by xmitter was: Hello ... how are you doing today?
packet just received by rcvr is:     Hello ... how are you doing today?
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx

RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)
*/
```

notes on need for wakeup packet before message packet.


Revisiting wakeup
→ ESP takes 120mS to wake up from trigger by AVX
47mS = time Tx amp is on
triggered wake up and start code (no delays)
measure sleeptime
3mS
36mS
Tx
"x" sent
Rx
6mSec
120mS
170mS
47
170-47
123 mS
soonest 2nd packet can be sent
→ soonest to send 2nd packet = 124mS (123mS won't work)
wakeup pkt "x" sent
wake up
124
soonest to send 2nd pkt after "x" wakeup = 124mS
Rx pkt 1
2nd pkt ("y")
Rx pkt 2
120mS wake up time
110mS MARGIN !!
using 2nd pkt = "y" = single char
using 2nd pkt = "Hello....." 32 bytes
124 mSec soonest to send 2nd packet !!

D - Testing Wake On Receive = WOR operation (*D-testWORwithACK*)

After successfully demonstrating Rx ESP32 sleep/wakeup, testing proceeds with the WOR mode of the eByte E220 module.

note: WOR as implemented in eByte is not a super reliable protocol. Care must be taken with necessary delays, and building in an acknowledgement "hand-shake" protocol into ESP code.

Wake On Receive (WOR) operation background:
To keep the receiver of a modem "alive" yet not consuming much energy, it is necessary to put the modem Rx into sleep mode most of the time, and allow it to wakeup for a very short time to "sniff" around for a packet to itself. This is the WOR sleep mode of the eByte module (i.e the modem chip and the tiny microcontroller chip in the eByte module). Since the eByte module requires an external microcontroller to manage it, and we're using the ESP32, it too must be put into sleep mode. So, the eByte in WOR mode with the sleep mode of the ESP32 is for the next testing project.

Software to perform the WOR mode is built into the by eByte tiny microcontroller. An example of the operation is that the modem Rx is put to sleep and awaken by software every 1 second for "a few milliseconds" ... this is putting the Rx in WOR_1 second configuration. To ensure that the Rx sees a Tx packet when it wakes up briefly, the Tx must send a very long packet (i.e. a long header) ... the Tx must be put into WOR_1 second mode also (although I've measure the Tx sending a 1.4 second long packet).

[the following is not supported in the E220; it was in the E22 module: We are already using the Send/ACK protocol to improve reliability, but to even further increase it, the Tx is put into a 4 second mode (i.e. send a 4 second long packet), while the Rx is in 1 second mode ... then an even longer packet will be sent to the "sniffing" Rx. These things have been implemented by me in a very reliable product (something for a later blog, if interested)]

~~~~~~

Building the project:
To do this, follow the procedure for Software Setup in the testBasicE220Wiring project, except when filling in the new project, use this:

Name = ***D-testWORwithACK***
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called *D-testWORwithACK*
- go to the src line, then select main.cpp -- a template file will appear

3) open the *D-testWORwithACK _main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the main.cpp template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *D-testWORwithACK* code in VSC. ... Ctrl-S to save

Again, make the addition of the final monitor_speed line to the platformio.ini code (click on it to open it):

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *D-testWORwithACK*

~~~~~~

Code Description:
After the ESP32 pin assignments, there are the #define statements for the eByte E220 module. They are a few additional #defines from the previous *B-testNormalTxRx* and *C-testRxESPwakeup* : ... Reg3 stores the WOR cycle duration:

```
#define REG3                    0x06  // needed for WOR implementation
#define WOR_CYCLE_1000MS        0x01
```

The 1st 3 functions are the same as in the previous test code:
- *eByteE220WriteReadMemory()*
- *initializeDevice()*
- *blinkOnBoardLED()*

The new functions are these 3:
- *putInNormal_mode()*
- *putInWOR_Tx_mode_1seconds()*
- *putInWOR_Rx_mode_1second()*

In these 3 new functions, the first thing done is to check the "eByte ready" line = AUX line. In developing this code, I found it helpful to printout where the AUX line is still not "ready". --- this is really a troubleshooting aid. For the Normal function, it will print "n1" or "n2" depending on where it is waiting, and in the WOR_Tx function, it will print "TxW1_1" or "TxW1_2", and in the WOR_Rx function, it will print "RxW1_1" or "RxW1_2".

Otherwise, the *putInNormal_mode()* function simply configures the eByte [M1 M0] lines as [0 0].

In the *putInWOR_Tx_mode_1second()* function, the eByte Reg3 must first be configured as WOR_CYCLE_1000MS (by writing to the registers in Configuration mode - [M1 M0] = [1 1] - done by the *eByteE220WriteReadMemory()* function), and then the eByte is put into Tx_WOR mode by setting [M1 M0] = [0 1].

Similarly, the *putInWOR_Rx_mode_1second()* function first configures Register 3, then puts the eByte into Rx_WOR mode by setting [M1 M0] = [1 0].

In the setup() code, first for both Tx and Rx, the UART and GPIO pins are configured, and a full *initialize()* of the registers is done. Then, f**or Rx,** it is put into a 1second WOR mode (i.e. it will "wake up and sniff for a few milliseconds" every ~1 second.
For Tx, it is put into a 1 second WOR mode (i.e. when it sends out a packet in this mode, the packet will last at least 1 second to allow the Rx to detect it).

In the loop() code, Tx and Rx have their own programs that repeat over and over.
For Tx, after setup, or when re-doing the *loop(),* it is in 1 second WOR mode. It sends off a message packet, blinking the onBoard LED 1x. It then goes to Normal mode (no need to stay in WOR mode at this point. Both the Tx and Rx are programmed to move to Normal mode once the Rx receives the WOR packet; it sends the Ack in Normal mode) --- this improves reliability since WOR mode is inherently unreliable in my experience). The Tx, after sending the message, waits for an ACK from the receiver in Normal mode, then checks to see if the contents are "ACK", and if so, blinks the onBoard LED 3x. Finally, it switches back to 1 second WOR mode, and will wait an additional 2 seconds before starting over and sending out a message packet. With the 1 second delay of the 3x blinking LED, and other delays, the message is sent out every ~4 seconds)

For Rx, after setup, or when re-doing the *loop()*, it is in 1 second WOR mode waiting for a long 1 second packet from Tx.
Upon getting the packet, and checking its contents, it will switch to Normal mode, then send out an ACK, blinking the onBoard LED 2x. (If the contents aren't correct, it will stay in 1 second WOR mode, and be ready to "try again" and receive a Tx packet. In this test, there is no code to support "timers" and "retrys" --- for now, this complicates the basic ideas. It will be included later in the final full-blown low-power Rx/Tx communication system.

There are multiple *delay()* statements in the code -- these are necessary to coordinate between the state of the Tx and of the Rx. A lot of testing was done to get these "right".

note: after initiating a Serial2 write to the eByte, the eByte will be busy for the duration of the message. It is important to not ask the ESP to change modes of the ESP before an accommodating delay (e.g. changing back to Normal from WOR mode after the Rx sends out an ACK, a delay of ~100 mSec is necessary; if not, the eByte will abort sending)

The cycle of Send/Receive/ACK is about 4 seconds - i.e. every 4 seconds a new packet is sent out and the receiver responds with an ACK.

~~~~~~
~~~~~~

Running the programs:

Loading the code:
Refer to the procedure to download the same code in the Tx, and then the Rx.

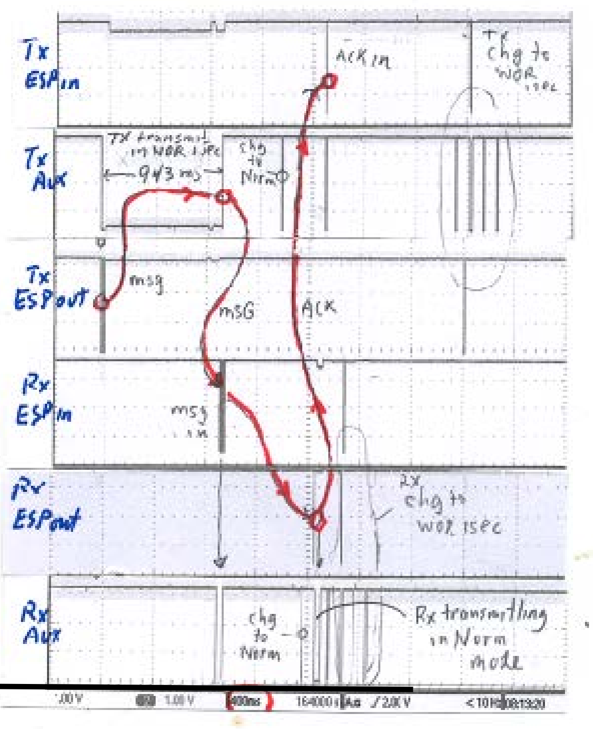
After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~4 seconds.

You should see the following printout:

```
/*   this is what should appear in the TRANSMITTER terminal
 ... initializing eByte - write to 4 memory locations

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)unit now in WOR_Tx Mode 1 second cycle mode

will transmit 35 bytes now in WOR_1 second mode - message is: Hello ... how are you doing today?
 n1 unit now in Normal Mode
...
ACK (received by Tx; sent from Rx after received message)

found ACK: ACK received by Tx from Rx after received message)
unit now in WOR_Tx Mode 1 second cycle mode
will now wait 3 seconds before transmitting a packet again (= 1 sec LED blinking + 2 sec delay)

will transmit 35 bytes now in WOR_1 second mode - message is: Hello ... how are you doing today?
 n1 unit now in Normal Mode
*/

/*   this is what should appear in the RECEIVER terminal
 ... initializing eByte - write to 4 memory locations

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)
unit now in WOR_Rx 1 second cycle Mode.......
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
will start looking for next packet

....
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
will start looking for next packet

....
*/
~~~~~~~~~~~~~~~~~
```



Waveforms: 4.16 second cycle period

E - Testing complete WOR + Rx ESP wakeup (E-testWORandESPwakeup)

After successfully demonstrating WOR mode, testing proceeds with both Rx ESP wakeup and WOR modes.

~~~~~~

Building the project:
To do this, follow the procedure for Software Setup in the testBasicE220Wiring project, except when filling in the new project, use this:

Name = ***E-testWORandESPwakeup***
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called *E-testWORandESPwakeup* - go to the src line, then select main.cpp -- a template file will appear

3) open the *E-testWORandESPwakeup_main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the main.cpp template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *E-testWORandESPwakeup* code in VSC. ... Ctrl-S to save

Again, make the addition of the final monitor_speed line to the platformio.ini code (click on it to open it):

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *E-testWORandESPwakeup*

~~~~~~

Code Description:
After the ESP32 pin assignments, there are the #define statements for the eByte E220 module, the same as used in previous test code.

The functions are also the same used in the previous test code:
- *eByteE220WriteReadMemory()*
- *initializeDevice()*
- *blinkOnBoardLED()*
- *putInNormal_mode()*
- *putInWOR_Tx_mode_1seconds()*
- *putInWOR_Rx_mode_1second()*
- *goToSleepESP()*

In the setup() code, as in the ESPsleep test code, the Rx does this when it wakes up, every time. The Tx only does it once.

For Rx, the UARTs and GPIOs pins are configured -- when the ESP32 wakes up, the GPIOs pins must be reassigned, and the UARTs reconfigured.
The following is similar to the **ESPsleep** and the **ESP_WOR** test code:
- on the 1st run after reset, after the UART and GPIO configuration and the chip is initialized, the *firstTimeUp* flag is made false, and it is put in WOR_1sec mode, then the ESP32 is put to sleep

```
 initializeDevice();  // configures address and channel; ends up in Normal mode
putInWOR_Rx_mode_1second();
firstTimeUp = false;
Serial.print("hello from firstTimeUp -- going to sleep now");
goToSleepESP()
```

The ESP32 is now asleep, waiting for an interrupt from the eByte.

When the "wakeup" packet is received by the eByte, the ESP32 wakes up and proceeds into this *setup()* again. This time, it does the UART/GPIO configurations, but not the initialization of the module (since *firstTimeUp* is now false).

It moves on, removing the holds on the M0 M1 output pins of the ESP32, and sits waiting for the "message packet" (the "wakeup packet" was received **--** it's what woke up the ESP32). It waits via the *while(!Serial2.available())* statement.

Just like in the *testNormalTxRx* project, the Rx finally gets the "message packet", checks it thoroughly, changes to Normal Mode (to improve reliability **-** WOR is not super reliable), and sends off an ACK to the transmitter if it's good.

Then it changes back to WOR_1second, and goes back to sleep, waiting for another "wakeup packet" from the Tx.

For Tx, it is initialized, then put into a 1 second WOR mode (i.e. when it sends out a packet in this mode, the packet will last at least 1 second to allow the Rx to detect it).

In the loop() code -- only the Tx does this (Rx goes to sleep in setup())

For Tx**,** after setup, or when re-doing the *loop(),* it is in 1 second WOR mode at the beginning of loop(). It starts by sending off the "wakeup packet". It waits, critically, 1 second (ostensibly to wait for the 1 second WOR wakeup packet, but as short a delay as 150 mSec works **-** due to the Rx ESP Serial2 port to wakeup and be able to record what the eByte is sending it). It then sends the WOR "message packet", blinking the onBoard blue LED 1x.

It now changes to Normal Mode to wait for an Ack from the Rx (going into Normal mode improves reliability; WOR mode is not perfectly reliable). On getting it, it changes back to WOR_1second mode, and then waits ~1 second before starting the *loop*() over again, transmitting a "wakeup packet".

With the delays, the cycle repeats every ~6 seconds.

**~~~~~~**
Running the programs:

Loading the code:
Refer to the testBasicE220Wiring procedure to download the same code in the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

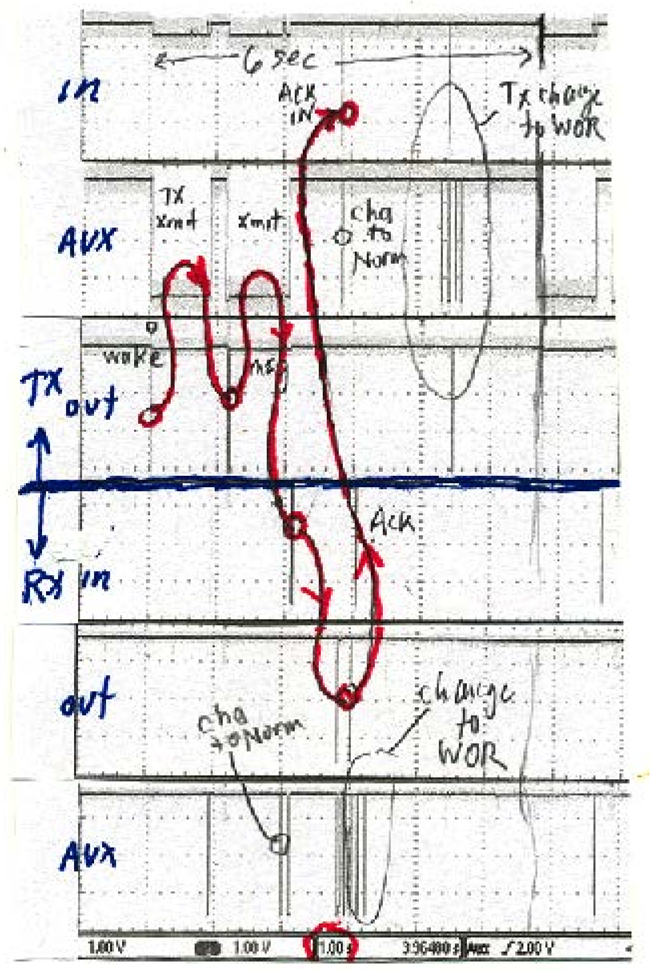
You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~6 seconds.

You should see the following printout:

```
/*   this is what should appear in the TRANSMITTER terminal
 now in tx setup
 ... initializing eByte - write to 4 memory locations

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)unit now in WOR_Tx Mode 1 second cycle mode

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
unit now in Normal Mode
.......................
ACK (received by Tx; sent from Rx after received message)

found ACK: ACK received by Tx from Rx after received message)
unit now in WOR_Tx Mode 1 second cycle mode
will wait 3 seconds before transmitting a packet again (= 1 sec LED blinking + 2 sec delay)
*/
```

```
/*   this is what should appear in the RECEIVER terminal
 REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)
unit now in WOR_Rx 1 second cycle Modehello from firstTimeUp -- going to sleep now
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)


RX JUST WOKE UP by wakeup packet; now to wait for message packet
.............
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)
....
*/
```

These are detailed waveforms captured with a scope, starting with the Tx sending out the wakeup packet:



F - Testing WOR + Rx ESP wakeup with OLED tracking errors (*F-testWORandESPwakeupWithOLED*)

After successfully demonstrating WOR+ESPsleep mode, testing proceeds with using an OLED to track errors made (useful to get an idea of how reliable the WOR protocol is that eByte implemented with the Semtech modem chip).

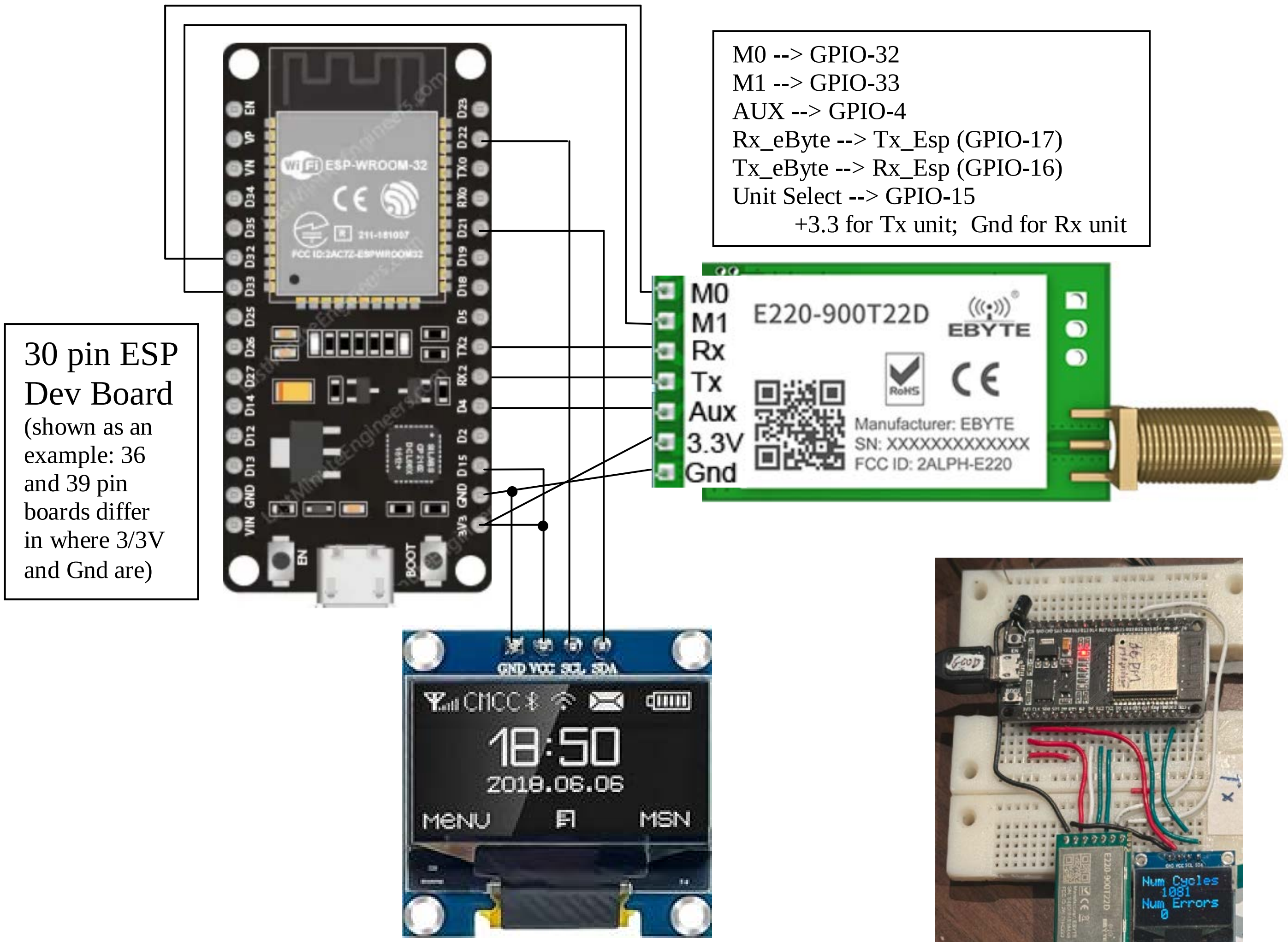
This testing requires adding the OLED, and minor addition to the code -- see hardware setup next. If you are not familiar with the OLED, refer to this tutorial by Rui and Sara.

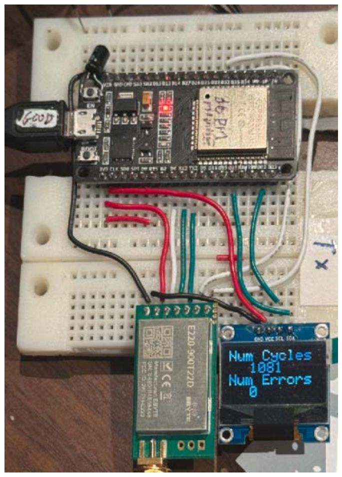
Wiring up the OLED to the Tx unit:

Additional Parts needed:

SSD1306 OLED 128x64 display I2C connectivity --- example --->
#24 wire or jumpers - 4 pcs.

Wiring Diagram for a 30 pin dev module (36 and 38 pin modules have Gnd and 3.3V on different pins):





my setup using "sliced" protoboards
(Tx has a 36 pin ESP32 dev module - use whatever) --- ***OLED only put on Tx, not Rx***

~~~~~~
~~~~~~

Building the project in Visual Code Studio:
To do this, follow the procedure for Software Setup in the testBasicE220Wiring project, except when filling in the new project, use this:

Name = F-testWORandESPwakeupWithOLED
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called
F-testWORandESPwakeupWithOLED
- go to the src line, then select main.cpp -- a template file will appear

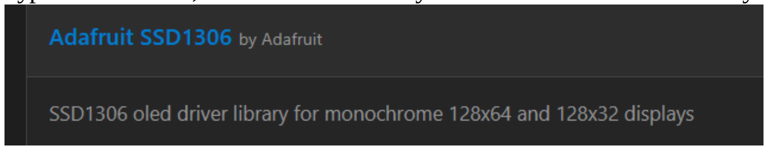
3) open the *F-testWORandESPwakeupWithOLED_main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the main.cpp template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *F-testWORandESPwakeupWithOLED* code in VSC. Ctrl-S to save

Again, make the addition of the final monitor_speed line to the *platformio.ini* code (click on it to open it)
AND type in the reference to the OLED library:

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = adafruit/Adafruit SSD1306@^2.5.15
```

(alternatively, to add this OLED library, click on the "House icon" at the bottom left in Visual Code Studio, then select "Libraries".
Type in SSD1306, and chose the library for monochrome SSD1306 by Adafruit, as shown here:


Adafruit SSD1306 by Adafruit
SSD1306 oled driver library for monochrome 128x64 and 128x32 displays

Then click "add to project", and it will basically modify the *platform.ini* code to look like what is shown above)

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *F-testWORandESPwakeupWithOLED*

~~~~~~

Code Description:

This is the same as in *E-testWORandESPwakeup*, with these additons:
1) adding 2 variables to track progress:

```
 // variables for testing for errors -- used in Tx only
int numTxCycles = 0;
int numErrors = 0;
```

2) adding support for the OLED in the #define area:

```
 // OLED stuff
#define SCREEN_WIDTH 128 // OLED display width, in pixels = columns (x)
#define SCREEN_HEIGHT 64 // OLED display height, in pixels = rows (y)
#define OLED_RESET     4 // Reset pin # (or -1 if sharing Arduino reset pin)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);  // i2c interface used
```

3) adding OLED setup in *setup()* just for Tx (only Tx has the OLED)

```
// OLED setup      - SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
   if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
     Serial.println(F("SSD1306 allocation failed"));
     while(1) {Serial.print("...can't connect to OLED display"); delay(1000);} } // Don't proceed, loop forever}
   Serial.println("OLED is setup\n");
   display.clearDisplay();
   display.display();
```
~~~~~~

4) incrementing the Number of Cycles, and if ACK is not recognized, incrementing the Number of Errors

note: the error tracked is only if the Tx receives a response from the Rx, but doesn't pass the filtering test.

it does not include the error when the receiver doesn't wake up or misreads the Tx transmission (this is address in G)

```
  numErrors ++;
}
numTxCycles++;
Serial.printf("number of Tx cycles so far: %d\n", numTxCycles);
```

5) reporting the Number of Cycles and the Number of Errors on the OLED:

```
// OLED 0.96 inch Display – 128 across × 64 down
   // TEXT SIZE = 2  --> allows 4 space character rows (15 pixels hight each), and 10 characters across (6.4 pixels each)
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(0,0);  // 1st char row: column 0, row = 0 (top right-hand pixel of a character)
  display.println("Num Cycles"); // will print 10 characters across = 128/10 = 12.8 pixels/character
  display.setCursor(30,16); // 2nd char row: column 60 (skipping 2 char: 2x15 = 30 pixels), row = down 1 char = 16 pix
  display.println(numTxCycles);
  display.setCursor(0,32); // 3rd char row: column 0, row = skip 2 character heights = 30 pixels
  display.println("Num Errors");
  display.setCursor(30,48); // 4th char row: column 30 (skipping 2 char: 2x15 = 30 pixels), row = down 3 char = 48 pixels
  display.println(numErrors);
  display.display();
   - This uses the Text Size 2, which allows 3 lines.
```

After the ESP32 pin assignments, there are the #define statements for the eByte E220 module, the same as used in previous test code.

The functions are also the same used in the previous test code:
- *eByteE220WriteReadMemory()*
- *initializeDevice()*
- *blinkOnBoardLED()*
- *putInNormal_mode()*
- *putInWOR_Tx_mode_1seconds()*
- *putInWOR_Rx_mode_1second()*
- *goToSleepESP()*

In the setup() code, as in the ESPsleep test code, the Rx does this when it wakes up, every time. The Tx only does it once.

For Rx, the UARTs and GPIOs pins are configured **--** when the ESP32 wakes up, the GPIOs pins must be reassigned, and the UARTs reconfigured.

The following is similar to the **ESPsleep** and the **ESP_WOR** test code:
- on the 1st run after reset, after the UART and GPIO configuration and the chip is initialized, the *firstTimeUp* flag is made false, and it is put in WOR_1sec mode, then the ESP32 is put to sleep

```
 initializeDevice();  // configures address and channel; ends up in Normal mode
putInWOR_Rx_mode_1second();
firstTimeUp = false;
Serial.print("hello from firstTimeUp -- going to sleep now");
goToSleepESP()
```

The ESP32 is now asleep, waiting for an interrupt from the eByte.

When the "wakeup" packet is received by the eByte, the ESP32 wakes up and proceeds into this *setup()* again. This time, it does the UART/GPIO configurations, but not the initialization of the module (since *firstTimeUp* is now false).

It moves on, removing the holds on the M0 M1 output pins of the ESP32, and sits waiting for the "message packet" (the "wakeup packet" was received **--** it's what woke up the ESP32). It waits via the *while(!Serial2.available())* statement.

Just like in the *testNormalTxRx* project, the Rx finally gets the "message packet", checks it thoroughly, changes to Normal Mode (to improve reliability - WOR is not super reliable), and sends off an ACK to the transmitter if it's good.

Then it changes back to WOR_1second, and goes back to sleep, waiting for another "wakeup packet" from the Tx.

For Tx, it is initialized, then put into a 1 second WOR mode (i.e. when it sends out a packet in this mode, the packet will last at least 1 second to allow the Rx to detect it).

In the loop() code -- only the Tx does this (Rx goes to sleep in setup())
For Tx**,** after setup, or when re-doing the *loop(),* it is in 1 second WOR mode at the beginning of loop(). It starts by sending off the "wakeup packet". It waits, critically, 1 second (ostensibly to wait for the 1 second WOR wakeup packet, but as short a delay as 150 mSec works **-** due to the Rx ESP Serial2 port to wakeup and be able to record what the eByte is sending it). It then sends the WOR "message packet", blinking the onBoard blue LED 1x.

It now changes to Normal Mode to wait for an Ack from the Rx (going into Normal mode improves reliability; WOR mode is not perfectly reliable). On getting it, it changes back to WOR_1second mode, and then waits ~1 second before starting the *loop*() over again, transmitting a "wakeup packet".

If the Tx doesn't get an ACK, it registers an error incrementing the *numErrors* variable. The OLED then is programmed as a 4 line display, as shown:

```
// OLED 0.96 inch Display – 128 across × 64 down
   // TEXT SIZE = 2  --> allows 4 space character rows (15 pixels hight each), and 10 characters across (6.4 pixels each)
  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(0,0);  // 1st char row: column 0, row = 0 (top right-hand pixel of a character)
  display.println("Num Cycles"); // will print 10 characters across = 128/10 = 12.8 pixels/character
  display.setCursor(30,16); // 2nd char row: column 60 (skipping 2 char: 2x15 = 30 pixels), row = down 1 char = 16 pix
  display.println(numTxCycles);
  display.setCursor(0,32); // 3rd char row: column 0, row = skip 2 character heights = 30 pixels
  display.println("Num Errors");
  display.setCursor(30,48); // 4th char row: column 30 (skipping 2 char: 2x15 = 30 pixels), row = down 3 char = 48 pixels
  display.println(numErrors);
  display.display()
```

With the delays, the cycle repeats every ~6 seconds.

~~~~~~

Running the programs:

Loading the code:
Refer to the *testBasicE220Wiring* procedure to download the same code in the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

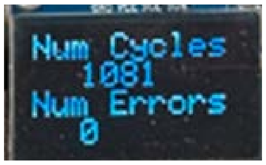
You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~5 1/2 seconds.

You should see the following printout:

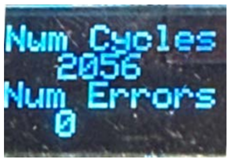
```
/*  TRANSMITTER
        now in tx setup
         ... initializing eByte - write to 4 memory locations

        REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
        REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
        REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
        REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)unit now in WOR_Tx Mode 1 second cycle mode
        OLED is setup

        This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
        wakeup packet sent ... now will transmit the message packet of 35 bytes
        message sent
        unit now in Normal Mode
        .........................
        ACK (received by Tx; sent from Rx after received message)

        found ACK: ACK received by Tx from Rx after received message)
        number of Tx cycles so far: 1
        unit now in WOR_Tx Mode 1 second cycle mode
        will wait 3 seconds before transmitting a packet again (= 1 sec LED blinking + 2 sec delay)
        */
```
~~~~~~

```
/*  RECEIVER
        doing fullInit for Rx - 1st time only; then go to sleep awaiting a wake-up packet and msg packet
         ... initializing eByte - write to 4 memory locations

        REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
        REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
        REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
        REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)
        unit now in WOR_Rx 1 second cycle Modehello from firstTimeUp -- going to sleep now
        RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)

        RX JUST WOKE UP by wakeup packet; now to wait for message packet
        ..............
        Hello ... how are you doing today? (received by Rx)
        found complete message: "Hello ... how are you doing today?"

        will now send Ack to Tx
        unit now in Normal Mode
         ... ACK sent

        unit now in WOR_Rx 1 second cycle Mode
        RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)
```

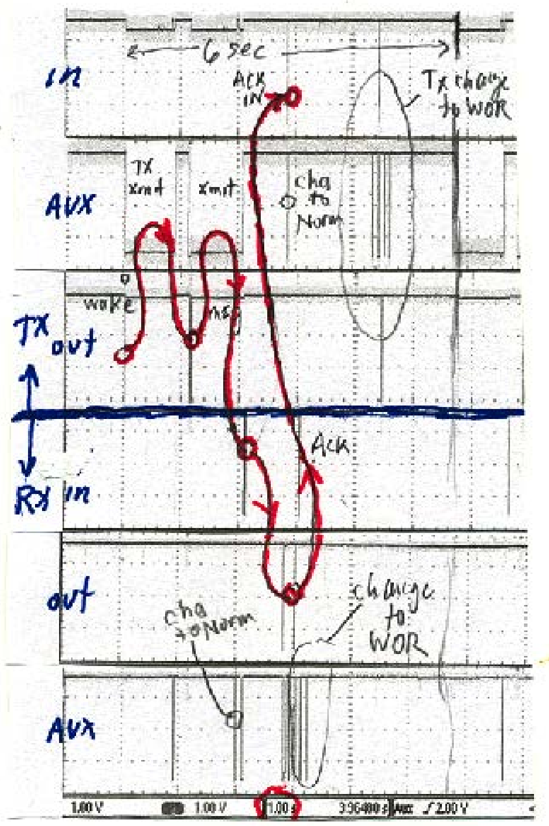
You'll see this on the OLED, for example:


Num Cycles
1081
Num Errors
0

975 cycles later, ***the Rx hangs***
OLED *not showing errors* since the Tx was just waiting for the Rx to send an ACK
(this is improved in the next test example).


Num Cycles
2056
Num Errors
0

Details of timing via scope Waveforms:



NOTE: *if you get an error, and the system hangs quickly, it might be your USB port's voltage. The eByte is sensitive to voltage, and when sending out a packet, it can load down the USB port resulting in a lower voltage, and the system hangs. (the recovery from this behavior is added in the next testing: G)*

G - Testing Reliable WOR + Rx ESP wakeup
(*G-testReliableWORandESPwakeup*)

The previous test basically stops if the Tx doesn't accept the ACK from the Rx, or if the Rx doesn't receive the proper messages. This new test adds in the *complexity of time-outs* (when the Tx can't find the ACK) and the complexity of *sending only the message* (in the case where the Rx didn't wake up with the "wakeup packet", but rather the message packet, thus waiting for another message packet). This complexity allow retries by the Tx, and recovery.   ***This finally makes this communication system very reliable***, and adaptable to critical projects (like remote water valve on/off (i.e. not to leave water on due to failure to get the command to shut off), or remote lamp on/off (i.e. not to drain the battery due to not getting the command to shut off the lamp).

~~~~~~

Building the project in Visual Code Studio:
To do this, follow the procedure for Software Setup in the *testBasicE220Wiring* project, except when filling in the new project, use this:

Name = ***G-testReliableWORandESPwakeup***
Board = DOIT ESP32 DEVKIT V1
(to easily find this, type ESP32 in the Board box, and scroll down)
etc.

As described previously:

2) after hitting ok, and ok'ing the "trust me" popup, you'll have a project appear on the Left called
G-testReliableWORandESPwakeup
- go to the src line, then select main.cpp -- a template file will appear

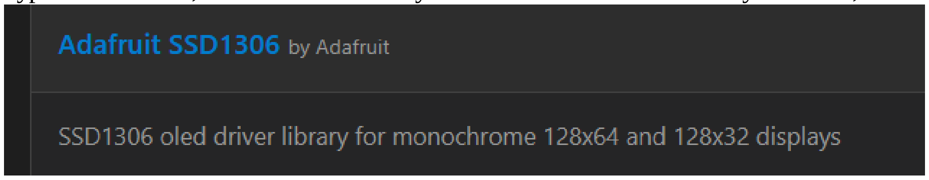
3) open the *G-testReliableWORandESPwakeup _main.cpp* file you downloaded. ... using a text editor like notepad, or Sublime, or whatever you use for text editing (Word works too), select the entire content of that file, and copy it (Ctrl-C).

4) back in VSC with the main.cpp template file open, click anywhere and select the entire file (Ctrl-A), then paste the "new" file into the window (Ctrl-V) .... you should now have the entire *G-testReliableWORandESPwakeup* code in VSC.   Ctrl-S to save

Again, make the addition of the final monitor_speed line to the *platformio.ini* code (click on it to open it)
AND type in the reference to the OLED library:

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = adafruit/Adafruit SSD1306@^2.5.15
```

(alternatively, to add this OLED library, click on the "House icon" at the bottom left in Visual Code Studio, then select "Libraries".
Type in SSD1306, and chose the library for monochrome SSD1306 by Adafruit, as shown here:


Adafruit SSD1306 by Adafruit

SSD1306 oled driver library for monochrome 128x64 and 128x32 displays

Then click "add to project", and it will basically modify the *platform.ini* code to look like what is shown above)

Finally, add this project to the workspace: *File/Add Folder to Workspace*

Ready to go ... but first, a description of the code *main.cpp* of *G-testReliableWORandESPwakeup*

~~~~~~

Code Description:

This is the similar to *F-testWORandESPwakeupWithOLED*, with ***no difference for the Rx code,*** *however*, the Tx code has several differences:
- adds new variables to track progress
- adds an ACK timeout to allow for a resend
- allows for sending just the message, not the wakeup (in the case where the Rx missed the wakeup packet, and used the message packet to wakeup; then it stays waiting for a message packet.

Tx specific changes:
1) add 5 new variables - badACKctr, noACKctr , waitForACKctr, sendWakeup, and cycleNumberWhenBadACK

```
 // variables for testing for errors -- used in Tx only
int numTxCycles = 0;
int noACKctr = 0;     // fails to find an ACK, and times out after looking for 4 characters from the Serial2 port
int badACKctr = 0;  // fails filter test of finding the word "ACK" .... rare, but possible
int waitForACKctr = 0;  // a counter used for the timeout of finding ACK characters
bool sendWakeup = true; // normally the wakeup packet is sent, then the message; false if needs only the message again.
int cycleNumberWhenBadACK = 0; // use to track the cycle when Bad Ack (i.e. a bad character), but doesn't register when No Ack
```

2) add "timeout" feature while waiting for ACK:

```
 // ********************** ACK section -- TX waits for ACK ***********************
waitForACKctr = 0;  // initialize the wait counter
char ignoreThisChar = Serial2.read();  // for some reason, a character can be stuck in Serial2 of the ESP, so this "cleans it out"
char incomingByteACK[RxAckLength] = {}; // zero the array;
for (int i=0; i<RxAckLength; i++) {  // for "ACK", RxAckLength = 4 (last char = CReturn or Linefeed)
  while(!Serial2.available()){ // wait for a character ... WAIT HERE !
     Serial.print("A?");
     delay(10);
     waitForACKctr ++;
     if (waitForACKctr == 200){  // wait total of 2 seconds for response; ACK not received
       waitForACKctr = 0;        // reset the counter
       Serial.print("w");
       sendWakeup = false;  // assume Rx is waiting for the msg; need to send it again, just the msg
       break;
     }}

  incomingByteACK[i] = Serial2.read();
  Serial.printf("\nincomingByteACK[%d] = %c ", i, incomingByteACK[i]);
}
```

3) if the ACK is not received and Tx times out, the n*oACKctr* is incremented and the Tx will try sending just the message again (i.e. not the wakeup packet). The *sendWakeup* flag is set to false. This is the normal case of an error: the Rx wakes up to the 2nd packet (the message) instead of the 1st packet (the wakeup), and is waiting and waiting for the message packet again. So in this case just the message is sent.

4) if the ACK is received, but it is corrupted (e.g. AdK instead of ACK), the *sendWakeup* flag stays true, and the Tx resends both a wakeup and a message. In my experience, the error is *nearly always* that the ACK is not received i.e. case 3) above (I have seen a corrupted ACK received only once after filtering in extensive testing; that is the purpose of the badACKctr **-** it registers only if the ACK is received but corrupted **-** this counter is nearly always 0).

5) the OLED displays number of Cycles in test (numTxCycles), the number of missed ACKs (noACKctr), the number of corrupted ACKs received (badACKctr), and on which cycle the error occurred (this is obviously for testing) and gives confidence that the system continues to reliably run despite these errors.

The Tx loop repeats every ~6seconds (i.e. a new wakeup/msg pair of packets are sent out)

Running the programs:

Loading the code:
Refer to the *testBasicE220Wiring* procedure to download the same code in the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

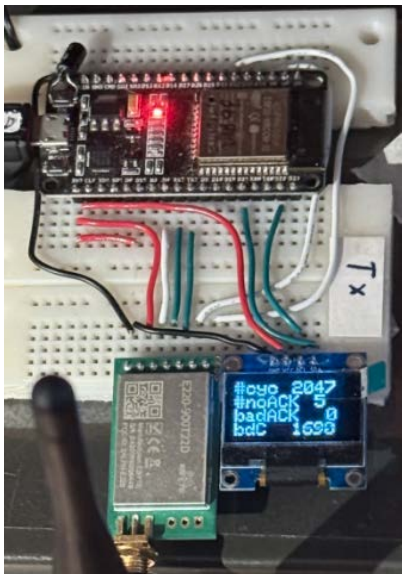
To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

You should see the Tx LED blink 1x, then the Rx LED blink 2x, then the Tx LED blink 3x ... and repeat every ~ 6 seconds.

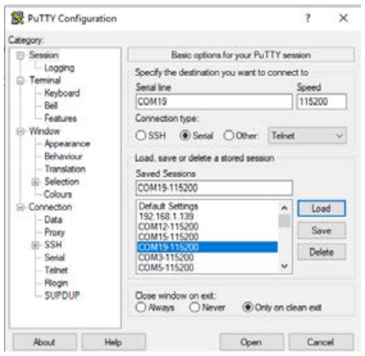
You should see the following printout:

```
/*   TRANSMITTER
     this is what should appear in the TRANSMITTER terminal

 ... initializing eByte - write to 4 memory locations
REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)unit now in WOR_Tx Mode 1 second cycle mode
OLED is setup


This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
unit now in Normal Mode
A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A
?A?A?A?A?A?A?A?A?A?A?A?A?
incomingByteACK[0] = A
incomingByteACK[1] = C
incomingByteACK[2] = K
incomingByteACK[3] =
number of Tx cycles so far: 1

ACK (= message recorded in Tx)

found ACK: ACK received by Tx from Rx after received message)
               unit now in WOR_Tx Mode 1 second cycle mode
               will wait 2 seconds before transmitting a packet again (= 1 sec LED blinking + 1 sec delay)

/*   RECEIVER
   this is what should appear in the TRANSMITTER terminal

 REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)
unit now in WOR_Rx 1 second cycle Modehello from firstTimeUp -- going to sleep now
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)
m?m?m?m?m?m?
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)
m?m?m?m?m?m?
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

*/
```

You'll see this on the OLED, for example:
   2,047 cycles - 5 errors = 0.2% error rate
(errors due to Tx not getting an ACK)



This is my monitor display, using "putty" ssh software to display the 2 USB connections (e.g. COM19 at 115200 baud)



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: COM19
Speed: 115200

Connection type: ( ) SSH [✓ Serial] ( ) Other: Telnet

Load, save or delete a stored session

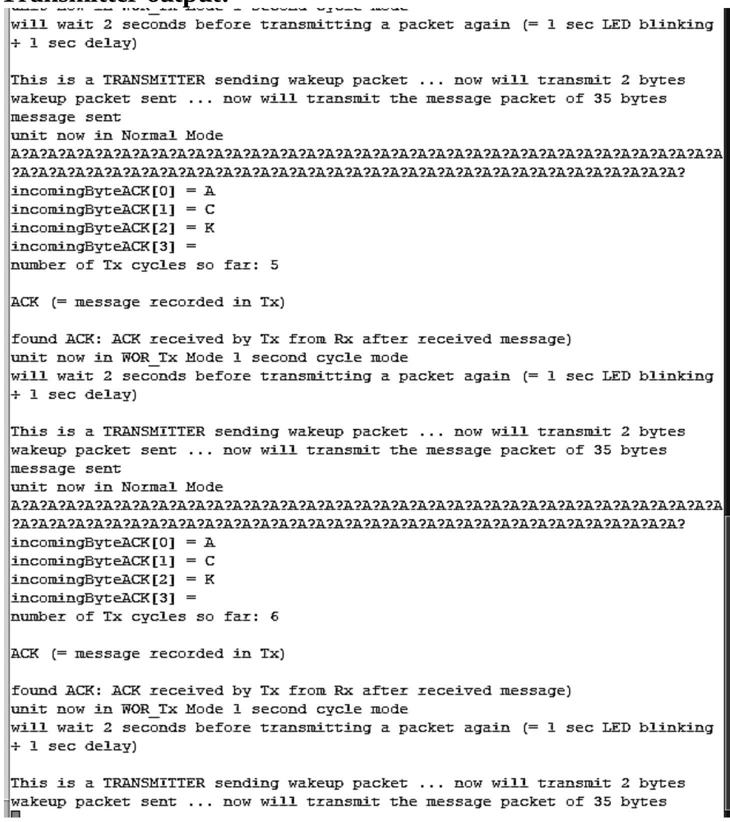
Saved Sessions: COM19-115200

```
Default Settings
192.168.1.139
COM12-115200
COM15-115200
COM19-115200
COM3-115200
COM5-115200
```

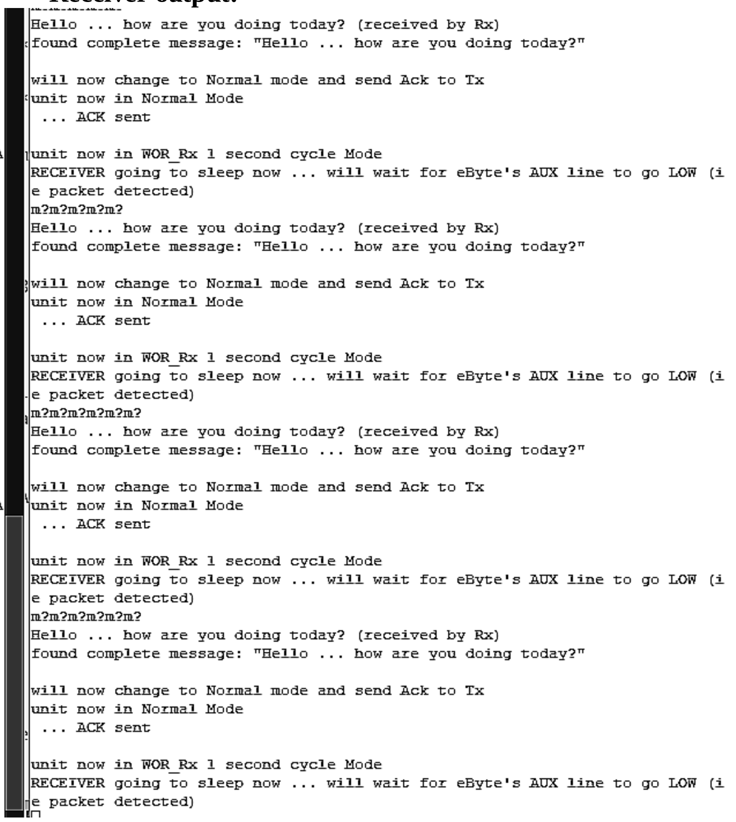
Close window on exit: ( ) Always ( ) Never [✓ Only on clean exit]

Transmitter output:

Receiver output:



```
will wait 2 seconds before transmitting a packet again (= 1 sec LED blinking
+ 1 sec delay)

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
unit now in Normal Mode
A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A
?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?
incomingByteACK[0] = A
incomingByteACK[1] = C
incomingByteACK[2] = K
incomingByteACK[3] =
number of Tx cycles so far: 5

ACK (= message recorded in Tx)

found ACK: ACK received by Tx from Rx after received message)
unit now in WOR_Tx Mode 1 second cycle mode
will wait 2 seconds before transmitting a packet again (= 1 sec LED blinking
+ 1 sec delay)

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
message sent
unit now in Normal Mode
A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A
?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?
incomingByteACK[0] = A
incomingByteACK[1] = C
incomingByteACK[2] = K
incomingByteACK[3] =
number of Tx cycles so far: 6

ACK (= message recorded in Tx)

found ACK: ACK received by Tx from Rx after received message)
unit now in WOR_Tx Mode 1 second cycle mode
will wait 2 seconds before transmitting a packet again (= 1 sec LED blinking
+ 1 sec delay)

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the message packet of 35 bytes
```



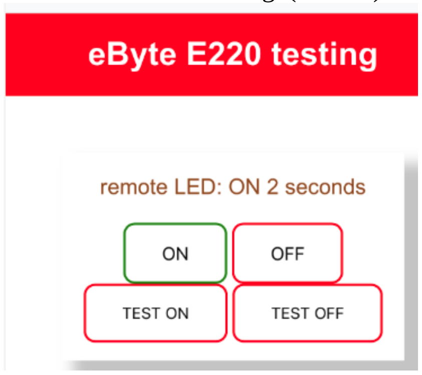
```
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i
e packet detected)
m?m?m?m?
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i
e packet detected)
m?m?m?m?m?
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i
e packet detected)
m?m?m?m?m?
Hello ... how are you doing today? (received by Rx)
found complete message: "Hello ... how are you doing today?"

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i
e packet detected)
```

H - "Something Useful" with the eByte E220
(H-Rx and H-Tx-SomethingUseful)

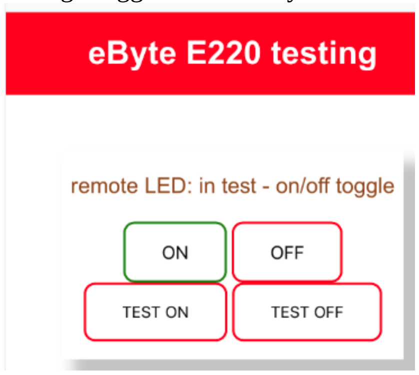
All the previous testing was to get used to how the E220 works, and the code. ... but it's not really useful for doing something remotely. This project provides a way to light a remote LED on the Rx, using a web browser, and/or to continuously run an on/off test to gain confidence in the error handling / reliability of the pair. There is separate code for the Tx (with the web browser), and the Rx (to interpret commands).

Here is the web page:

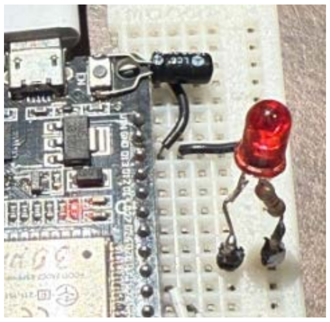
manual switching (turn on)

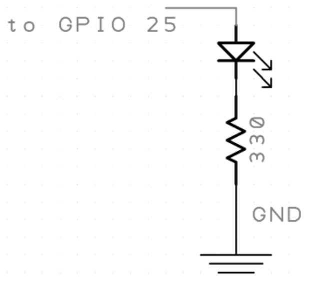


testing - toggle on/off every 10sec



Additionally, 2 red LEDs need to be hooked up to GPIO 25 (with a series resistor - 330 ohms works fine) - 1 on Tx, 1 on Rx.





~~~~~~

Building the project in Visual Code Studio: -- ***please follow the instructions in G-testReliableWORandESPwakeup***

Tx Code Description:

The #includes now have RTOS (real time OS) support to allow several things to run at the same time (i.e. the web browser, the on/off LED code, and the test). ... this is the collection:

```
#include <Arduino.h>
#include <Adafruit_SSD1306.h>
#include <stdio.h>
#include "sdkconfig.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
```

In addition to the variables in the G1 test, 2 more are added to support a continuous test, if selected via the browser.

```
bool stopTest = false;  // allows a test to run, if desired
int tempCtr = 0; // supports blink time of the LED in testing if used.
```
~~~~~~

The WiFi credentials need to be added:

```
// Replace with your network credentials
const char* ssid = "your SSID here";
const char* password = "your password here";
```

The code to allow several things to run at once is in the loop() section - they are transmitCommands, and continualTestingTxRx:

```
xTaskCreate(transmitCommands,      // Entry function of the task
        "transmit Commands",     // Name of the task
        100000,      // The number of words to allocate for use as the task's stack (arbitrary size enough for this task)
        NULL,        // No parameter passed to the task
        taskPriority, // Priority of the task
        NULL);       // No handle

 xTaskCreate(continualTestingTxRx,      // Entry function of the task
        "testing Tx Rx link",     // Name of the task
        10000,      // The number of words to allocate for use as the task's stack (arbitrary size enough for this task)
        NULL,        // No parameter passed to the task
        taskPriority, // Priority of the task
        NULL);       // No handle
```

The code for supporting a web server is adapted from Rui and Sara's 2_1_Output project in their book "Build Web Servers". Once the Tx ESP "serves up" the html code to the browser, there are 4 things (via buttons) that it can do via buttons: LED on, LED off, Test on, Test off.

Global variables (i.e. sensed by other functions in the code) are used to start/stop things:

- webONrequest, webOFFrequest, testON, testOFF ... the web support code sets those variables.

```
// setting up the webServer
 Serial.print("\n\n setting up the webserver\n");

 server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) { // Route for root / web page
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
 });

 server.serveStatic("/", SPIFFS, "/");

 server.on("/on", HTTP_GET, [](AsyncWebServerRequest *request) {   // Route to set GPIO state to HIGH
  if(webOFFrequest == true){webOFFrequest = false;}  // cancel an OFF request if it was made
  webONrequest = true; // this triggers the Tx to send out the ON command
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
 });

 server.on("/off", HTTP_GET, [](AsyncWebServerRequest *request) { // Route to set GPIO state to LOW
  if(webONrequest == true){webONrequest = false;}  // cancel an ON request if it was made
  webOFFrequest = true; // this triggers the Tx to send out the OFF command
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
 });

 server.on("/test_on", HTTP_GET, [](AsyncWebServerRequest *request) {   // Route to set GPIO state to HIGH
  testOFF = false;
  testON = true;
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
 });

 server.on("/test_off", HTTP_GET, [](AsyncWebServerRequest *request) {   // Route to set GPIO state to HIGH
  testON = false;
  testOFF = true;
  request->send(SPIFFS, "/index.html", "text/html", false, processor);
 });

 Serial.print("now will start the server \n");
 server.begin();
 Serial.print("\nfinished setup\n");
}
```

The functions that respond to these global variables are ***transmitCommands()*** and ***continualTesting().***

The easy one first: *continualTestingTxRx()*

```
void continualTestingTxRx(void *pvParameters){
  Serial.print("\nstarting continualTestingTxRx task\n\n");
  while(true){
    if(testON == true){
      webONrequest = true; // this triggers the Tx to send out the ON command
      delay(10000); // 10 sec delay
      //webONrequest = false; // transmit function will set this false, if an ACK was received
      webOFFrequest = true; // this triggers the Tx to send out the OFF command
      delay(10000); // 10 sec delay
      //webOFFrequest = false; // transmit function will set this false, if an ACK was received
    }
    else{delay(100);} // delay a little before checking again
  }
}
```

This code senses the value of *testON* only (changed by the web support code). If *testON* is set true in the web section (via the button in the browser's web page), then it goes into a loop that triggers the remote LED (and local Tx LED) to turn on for 10 seconds, and off for 10 seconds ... it does this by manipulating the global variables *webONrequest and webOFF request*, which are sensed by the other function: transmitCommands(). (as described below, the LED is only on for 2 seconds)

Now for the *transmitCommands()* code:

This is essentially a modification to the G-testReliableWORand ESPwakeup code. However, instead of being in the loop() section of code, it is a separate function called by the RTOS. But with the *while(true)* statement at the beginning, it is effectively in a loop, waiting for the value of ***webONrequest*** or ***webOFFrequest*** to go true (changed by the web support code, which senses a browser's button was pushed). It does this loop check every 100 milliseconds with the *delay(100)* command at the end of the function.

Once ***webONrequest*** or ***webOFFrequest*** go true, a wakeup is sent to the Rx. Then, depending on which one was set true, either "***onCmd"*** or the "***offCmd"*** is send to the Rx .... and waits for an ACK.

At the Rx end, it senses the command, and reads it. If it's *onCmd*, then it turns on its red LED (programmed with a delay of 2 seconds to see it come on, before going to sleep and shutting it off). It then sends an ACK to the Tx, and goes back to sleep. Similarly for the *offCmd*

At the Tx end, once an ACK is received, the code determines what it had sent, and if it sent *onCmd* then it turns on its LED - a visual for the user at the "main station" (i.e. the Tx side), AND changes the *webONrequest* to false. Similarly for *offCmd.* If an ACK isn't received, the Tx times out and resends just the command (not the wakeup), just like in the G-test... program. If the ACK received is corrupted, the wakeup and command are resent ... again just like in the G-test, and the *webONrequest* will stay true, allowing the transmission to happen again.

The OLED does the same as in G-test...

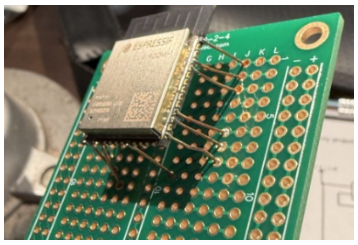
Rx Code Description:

The Rx code is very similar to the G-test.. code, except it has to read the message and determine which command was sent: ***onCmd*** or ***offCmd****.* Based on what it receives it turns on or off its red LED. Since the object of this project is a low-power receiver, keeping the LED driven by the ESP means the ESP is awake and using power ! ... so even though the remote Rx is told to turn on for 10 seconds, the code only turns on for a few seconds before making the ESP go to sleep - this shuts off the LED (the GPIO output goes lo on sleep).

The Rx behaves just like it did in the *G-testReliableWORandESPwakeup* code -- blinks 2x when a proper command is received, and is otherwise asleep.

From a practical point of view, as a low-power Rx, a necessary component is a latchable relay that gets pulsed on/off, and allows the ESP+eByte to go back to sleep. A good latchable relay is the Omron G6KU-2P-Y DC3 (which runs on 3.3V 33mA; the ESP32 typical pin output max spec is 40mA, so no problem pulsing it) - see https://components.omron.com/eu-en/products/relays/G6K. (my LadyLiberty project uses this -- details in another github entry). The code would "poke" the relay on/off, and go to sleep - the relay is latched on/off and its coil draws no current after latched.

Of course, really, a low-power Rx ideally is made from just an ESP32 module, not a power-hungry dev board ... as described in the introduction. But using a solar panel and the "anti-sleep" device between a cellphone backup battery pack and the dev board, this can work (example - Lambda Nu's #CS-POWEREVER device)



Using the module along requires some hardware experience like this mounting.

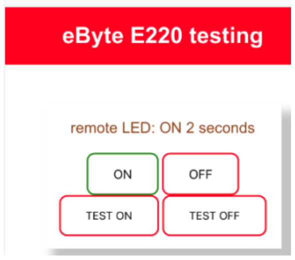
Running the programs:

Loading the code:
Refer to the *testBasicE220Wiring* procedure to download the same code in the Tx, and then the Rx.

After modules are loaded, it's necessary to synchronize the Tx/Rx pair (this could be done automatically with a more code, but complicates learning these chips).

To synchronize the Tx/Rx pair, hold the reset buttons down on each ESP32 module, and then release the Rx first, then the Tx.

Now go to the IP of the Tx ESP (it will be printed out on bootup if you have a terminal connected), and verify the webpage comes up.

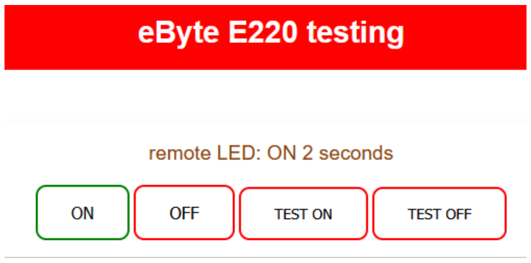


Now, poke the ON button, and observe the Tx blinks 1x, the Rx blinks 2x and sends an ACK to the Tx which blinks 3x. The remote Rx red LED turns on ~2 seconds, while the local Tx red LED stays on. ... it will stay this way until the OFF button is pushed.

To run continuously (e.g. test the setup for a few hours and gain confidence in the setup), select run Test in the browser. You'll see the On sequence happen (blue LEDs blinking), and things changing every 10 seconds. This will run until you stop it.

Results:
You should see the following printout:

```
/*  TRANSMITTER
    this is what should appear in the TRANSMITTER terminal

Connecting to WiFi .....192.168.1.170
SPIFFS mounted successfully


 setting up the webserver

starting transmitCommands task


 ... initializing eByte - write to 4 memory locations

starting continualTestingTxRx task

REG_MODULE_ADDR_H : now will start the server

finished setup
C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)OLED is setup

C1 6 1 1  = response. Expect (C1 6 1 1)unit now in WOR_Tx Mode 1 second cycle mode
```



```
(after poking ON button in browser >>>>)

C1 6 1 1  = response. Expect (C1 6 1 1)unit now in WOR_Tx Mode 1 second cycle mode

top of loop - sendWakeup = 1

This is a TRANSMITTER sending wakeup packet ... now will transmit 2 bytes
wakeup packet sent ... now will transmit the command
   ****  onCmd sent to Rx  ****

unit now in Normal Mode
A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?A?
incomingByteACK[0] = A
incomingByteACK[1] = C
incomingByteACK[2] = K
incomingByteACK[3] =
number of Tx cycles so far: 1

ACK = message recorded in Tx

found ACK: ACK received by Tx from Rx after received message)
C1 6 1 1  = response. Expect (C1 6 1 1)unit now in WOR_Tx Mode 1 second cycle mode
will wait 2 seconds before transmitting a packet again (= 1 sec LED blinking + 1 sec delay)

*/

/*   RECEIVER
   this is what should appear in the TRANSMITTER terminal

 doing fullInit for Rx - 1st time only; then go to sleep awaiting a wake-up packet and msg packet
 ... initializing eByte - write to 4 memory locations

REG_MODULE_ADDR_H : C1 0 1 1  = response. Expect (C1 0 1 1)
REG_MODULE_ADDR_L : C1 1 1 2  = response. Expect (C1 1 1 2)
REG0 : C1 2 1 64  = response. Expect (C1 2 1 64)
REG2_RF_CHAN : C1 4 1 28  = response. Expect (C1 4 1 28)
unit now in WOR_Rx 1 second cycle Modehello from firstTimeUp -- going to sleep now
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)

(after Tx browser selects ON button >>>>)


RX JUST WOKE UP by wakeup packet; now to wait for command

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

onCmd (received by Rx)

received onCmd

will now change to Normal mode and send Ack to Tx
unit now in Normal Mode
 ... ACK sent

unit now in WOR_Rx 1 second cycle Mode
RECEIVER going to sleep now ... will wait for eByte's AUX line to go LOW (i.e packet detected)

*/
```

NOTE --- when performing the Test function, it turns on/off every ~10 seconds. WHEN SWITCHING OFF the testing, it takes a while to stabalize (shutting it off is usually a random event in the testing cycle, and the Rx may end up waiting for a packet that the Tx doesn't send, so it may take a while to re-sync ... but it eventually does, and the test stops).

Visual Results:
You'll see this on the OLED
   -- present cycle number
   -- # no ACKs
   -- # corrupted ACKs (bad ACKs)
   -- cycle number for last error