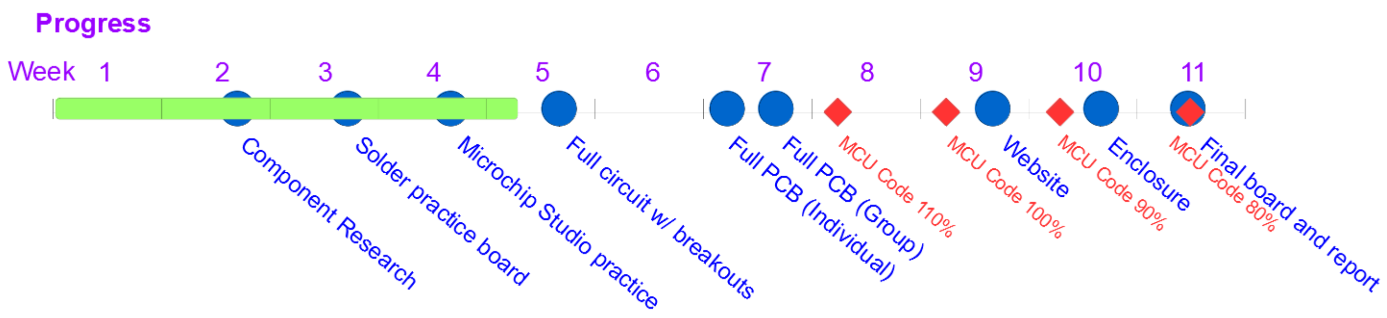
EE 326: Electronic System Design 1

Firmware Design

Due:	Feb 25, 2025	(110%)
	Mar 04, 2025	(100%)
	Mar 11, 2025	(90%)
	Mar 19, 2025	(80%)



Please complete this assignment with your group. To get credit for your work, please complete the following 4 parts.
1) Upload your zipped project folder to Canvas. The project folder should contain the .atsln file, and a folder containing all the source and debug files.
2) Please record and narrate a video of Tera Term (or equivalent) showing (a) initialization after reset, (b) the server starting, (c) a client connecting, and (d) image transfer.
3) Please record and narrate a video of the logic analyzer capturing a full cycle of MCU-to-WiFi image transfer, i.e. MCU initializing image transfer over UART, WiFi module setting its comm pin low, image transfer over SPI, WiFi module setting its comm pin high. Please zoom in on each part and explain what you are seeing. Please include at least the UART lines (with a UART analyzer), the SPI lines (with a SPI analyzer), and the comm pin. See Appendix C for an example (and an important note).
4) Please record a video of your website streaming images from your camera.

For this assignment, you will program your MCU and configure the WiFi module. The end goal is to have your system take continuous photos and send them to the ESP32 to display on a website.
Note: you do not have to implement every one of these functions by hand. Many of them will be directly copy-and-pasted from sample projects. Others will require only slight modifications. Only a few have to written from start to finish. I recommend using the following sample projects: common ioport service example, ov7740 imagesensor capture example, usart serial example, and spi example.

1 Programming the MCU (SAM4S8B)

The files you should modify (or create, shown in **bold purple**) are:

- main.c
- **wifi.c**
- **wifi.h**
- **camera.c**
- **camera.h**
- conf_board.h
- conf_clock.h
- init.c

Below are the descriptions of what you should implement. File names are shown in **orange**, function declarations are shown in **green**, and variables are shown in **red**.

- **wifi.h**: WiFi control pin definitions, WiFi UART parameters and pin definitions, WiFi SPI parameters and pin definitions, WiFi function and variable declarations.
- **wifi.c**
  - WiFi variable initializations.
  - **void wifi_usart_handler(void)**: Handler for incoming data from the WiFi. Should call **process_incoming_byte_wifi** when a new byte arrives.
  - **void process_incoming_byte_wifi(uint8_t in_byte)**: Stores every incoming byte (**in_byte**) from the ESP32 in a buffer.
  - **void wifi_command_response_handler(uint32_t ul_id, uint32_t ul_mask)**: Handler for "command complete" rising-edge interrupt from ESP32. When this is triggered, it is time to process the response of the ESP32.
  - **void process_data_wifi (void)**: Processes the response of the ESP32, which should be stored in the buffer filled by **process_incoming_byte_wifi**. This processing should be looking for certain responses that the ESP32 should give, such as "SUCCESS" when "test" is sent to it.
  - **void wifi_provision_handler(uint32_t ul_id, uint32_t ul_mask)**: Handler for button to initiate provisioning mode of the ESP32. Should set a flag indicating a request to initiate provisioning mode.
  - **void wifi_spi_handler(void)**: Handler for peripheral mode interrupts on SPI bus. When the ESP32 SPI controller requests data, this interrupt should send one byte of the image at a time.
  - **void configure_usart_wifi(void)**: Configuration of USART port used to communicate with the ESP32.
  - **void configure_wifi_comm_pin(void)**: Configuration of "command complete" rising-edge interrupt.
  - **void configure_wifi_provision_pin(void)**: Configuration of button interrupt to initiate provisioning mode.
  - **void configure_spi(void)**: Configuration of SPI port and interrupts used to send images to the ESP32.
  - **void spi_peripheral_initialize(void)**: Initialize the SPI port as a peripheral (slave) device. *Note: The embedded industry is trying to phase out the "master/slave" terminology that is widespread. In the SPI example project, this function is called spi_slave_initialize, just so you know where to look. For more details about this topic in general, you can start here and here.*
  - **void prepare_spi_transfer(void)**: Set necessary parameters to prepare for SPI transfer. *Note: Same as above. In the SPI example project, this function corresponds to the spi_slave_transfer function.*
  - **void write_wifi_command(char* comm, uint8_t cnt)**: Writes a command (**comm**) to the ESP32, and waits either for an acknowledgment (via the "command complete" pin) or a timeout. The timeout can be created by setting the global variable **counts** to zero, which will automatically increment every second, and waiting while **counts** < **cnt**.
  - **void write_image_to_web(void)**: Writes an image from the SAM4S8B to the ESP32. If the length of the image is zero (i.e. the image is not valid), return. Otherwise, follow this protocol (illustrated in Appendix C):
    1. Configure the SPI interface to be ready for a transfer by setting its parameters appropriately.
    2. Issue the command "image_transfer xxxx", where xxxx is replaced by the length of the image you want to transfer.
    3. The ESP32 will then set the "command complete" pin low and begin transferring the image over SPI.

    4. After the image is done sending, the ESP32 will set the “command complete” pin high. The MCU should sense this and then move on.

- **camera.h**: Camera pin definitions, camera TWI parameters, camera function and variable declarations.
- **camera.c**
  - Camera variable initializations.
  - **void vsync_handler(uint32_t ul_id, uint32_t ul_mask)**: Handler for rising-edge of VSYNC signal. Should set a flag indicating a rising edge of VSYNC.
  - **void init_vsync_interrupts(void)**: Configuration of VSYNC interrupt.
  - **void configure_twi(void)**: Configuration of TWI (two wire interface).
  - **void pio_capture_init(Pio *p_pio, uint32_t ul_id)**: Configuration and initialization of parallel capture.
  - **uint8_t pio_capture_to_buffer(Pio *p_pio, uint8_t *uc_buf, uint32_t ul_size)**: Uses parallel capture and PDC to store image in buffer.
  - **void init_camera(void)**: Configuration of camera pins, camera clock (XCLK), and calling the **configure_twi** function.
  - **void configure_camera(void)**: Configuration of OV2640 registers for desired operation. To properly initialize the camera for JPEG (at 320 × 240 resolution), use the following commands, instead of the corresponding one in the sample project):

    ```
    ov_configure(BOARD_TWI, JPEG_INIT);
    ov_configure(BOARD_TWI, YUV422);
    ov_configure(BOARD_TWI, JPEG);
    ov_configure(BOARD_TWI, JPEG_320x240);
    ```

  - **uint8_t start_capture(void)**: Captures an image after a rising edge of VSYNC, and gets image length. Returns 1 on success (i.e. a nonzero image length), 0 on error.
  - **uint8_t find_image_len(void)**: Finds image length based on JPEG protocol. Returns 1 on success (i.e. able to find “end of image” (EOI) and “start of image” (SOI) markers, with SOI preceding EOI), 0 on error.
- **conf_board.h**: Pin definitions for general board.
- **conf_clock.h**: Clock definitions for general board.
- **init.c**: Pin initializations for general board.
- **main.c**: General operation. A flow chart of the whole program is shown in Appendix A, along with a data flow diagram in Appendix B. First, run all initializations. Then, loop through a set of commands. More specifically:
  - Initialization
    - ∗ Initialize clock and board definitions.
    - ∗ Configure and start the Timer. (Look in the “timer_interface” functions.)
    - ∗ Configure the WiFi USART and SPI, as well as the “command complete” and “provision” pins (interrupts).
    - ∗ Configure the indicators and the “command complete”, ”network”, and ”clients” GPIOs through the UART interface of the ESP32 (detailed in Section 2).
    - ∗ Initialize and configure the camera.
    - ∗ Reset the WiFi and wait for it to connect to a network. While waiting, make sure to listen for the “provision” pin.
    - ∗ Send “test” to the WiFi module and wait for a response of “SUCCESS”. If you do not receive it, wait 10 seconds, reset the WiFi module, and try again.
  - Loop

    - ∗ Check for provisioning request and act accordingly.
    - ∗ If network is available and clients are connected, take picture.
    - ∗ If picture taken successfully, transfer it to the ESP32.

In your file structure, make sure to also include the files **ov2640.c**, **ov2640.h**, **ov2640_table_registers.c**, **timer_interface.c**, and **timer_interface.h**.

Notes:

- Don't forget to include <asf.h> in every .h file.
- Don't forget to include .h files in .c files.
- Don't forget to declare variables as "volatile" if they are changed in interrupt service routines.
- It may be helpful to turn off compiler optimization for debugging.

2 Configuring the WiFi Module (ESP32)

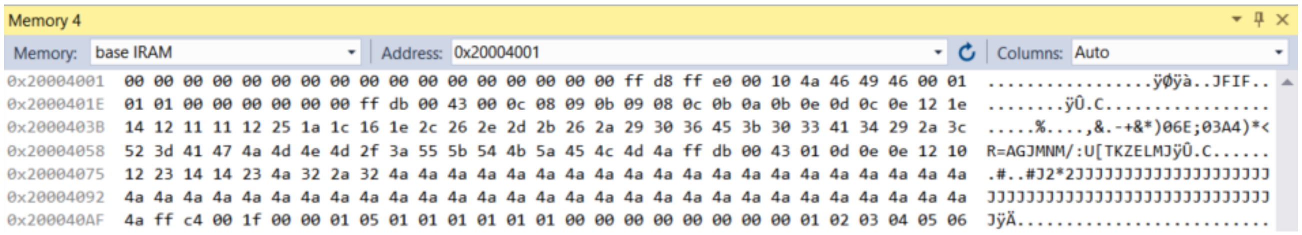
1. Make sure the ESP32 has the proper firmware and file system loaded, as detailed in Task 4.
2. Configure the LED indicators:
   (a) Configure the top LED as the wlan output (indicating that the ESP32 is connected to a network).
   (b) Configure the middle LED as the websockets output (indicating that clients are viewing the video stream).
   (c) Configure the bottom LED as the AP output (indicating whether provisioning mode is active).
3. Configure the "command complete", "network", and "clients" GPIOs.

For further information, consult the esp32_firmware_api document.

3 Testing the Webcam

While we have not yet made a custom website to display our image, we can still check to see that everything is working. The easiest check is to open a browser to the IP address of your ESP32. It should immediately start streaming images. If that does not work, we have to break it down into steps.

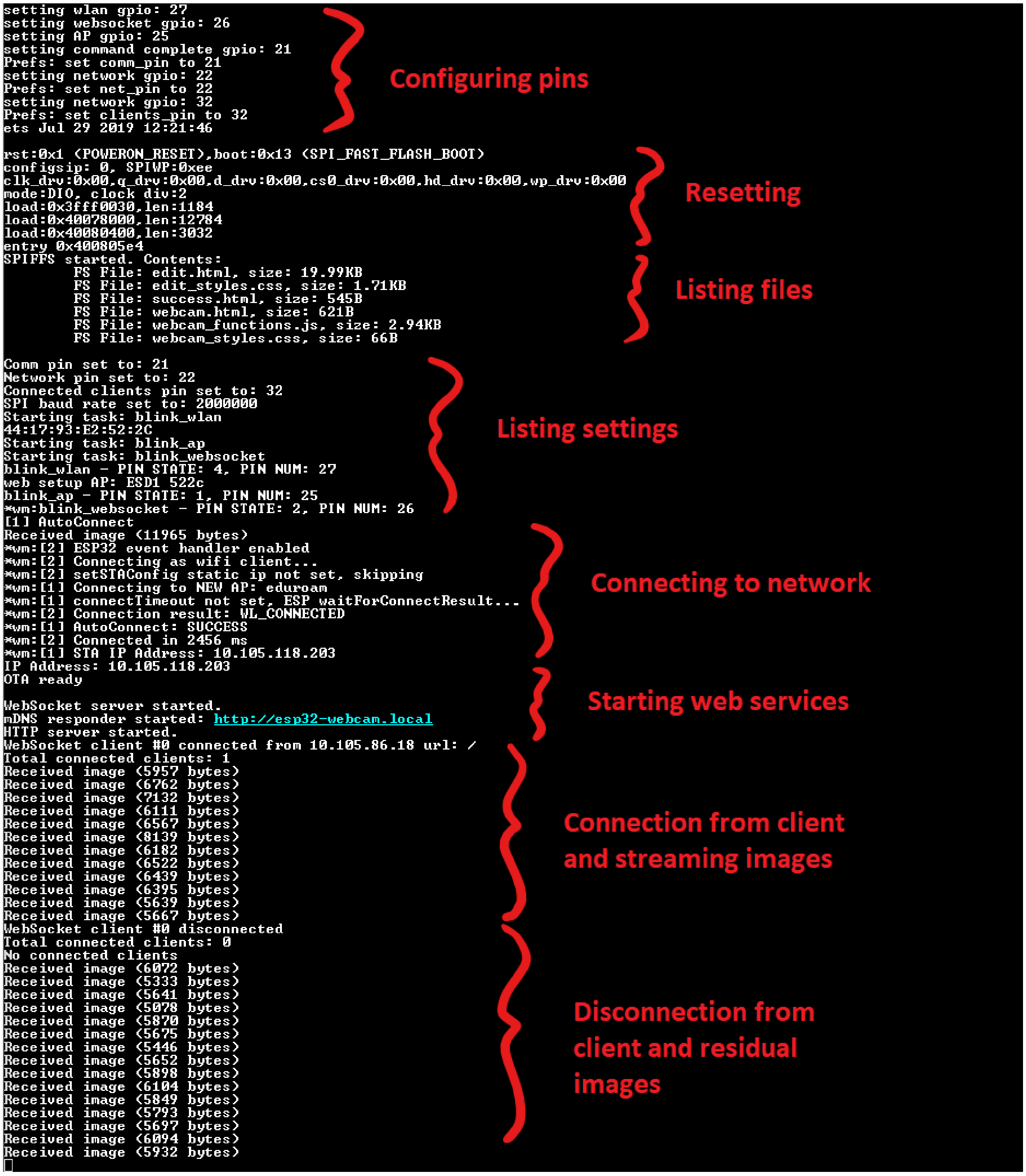
Assuming your circuit passed the tests of Task 4, most of your connections should be OK (with the possible exception of the connections that were not fully specified and therefore not tested, such as "free GPIOs"). Try placing a breakpoint right after your image is captured. When execution stops there, open the "memory" interface in the debugger and go to "Base IRAM" (internal RAM). If you scroll through it a little, you should be able to find your image. Based on the JPEG protocol, you should see a structure similar to the one below, though probably at a different address than mine (depending on your program).



```
0x20004001  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff d8 ff e0 00 10 4a 46 49 46 00 01  .................ÿØÿà..JFIF..
0x2000401E  01 01 00 00 00 00 00 00 ff db 00 43 00 0c 08 09 0b 09 08 0c 0b 0a 0b 0e 0d 0c 0e 12 1e  ........ÿÛ.C.................
0x2000403B  14 12 11 11 12 25 1a 1c 16 1e 2c 26 2e 2d 2b 26 2a 29 30 36 45 3b 30 33 41 34 29 2a 3c  .....%....,&.-+&*)06E;03A4)*<
0x20004058  52 3d 41 47 4a 4d 4e 4d 2f 3a 55 5b 54 4b 5a 45 4c 4d 4a ff db 00 43 01 0d 0e 0e 12 10  R=AGJMNM/:U[TKZELMJÿÛ.C......
0x20004075  12 23 14 14 23 4a 32 2a 32 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a  .#..#J2*2JJJJJJJJJJJJJJJJJJJJ
0x20004092  4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a 4a  JJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
0x200040AF  4a ff c4 00 1f 00 00 01 05 01 01 01 01 01 01 00 00 00 00 00 00 00 00 01 02 03 04 05 06  JÿÄ..........................
```

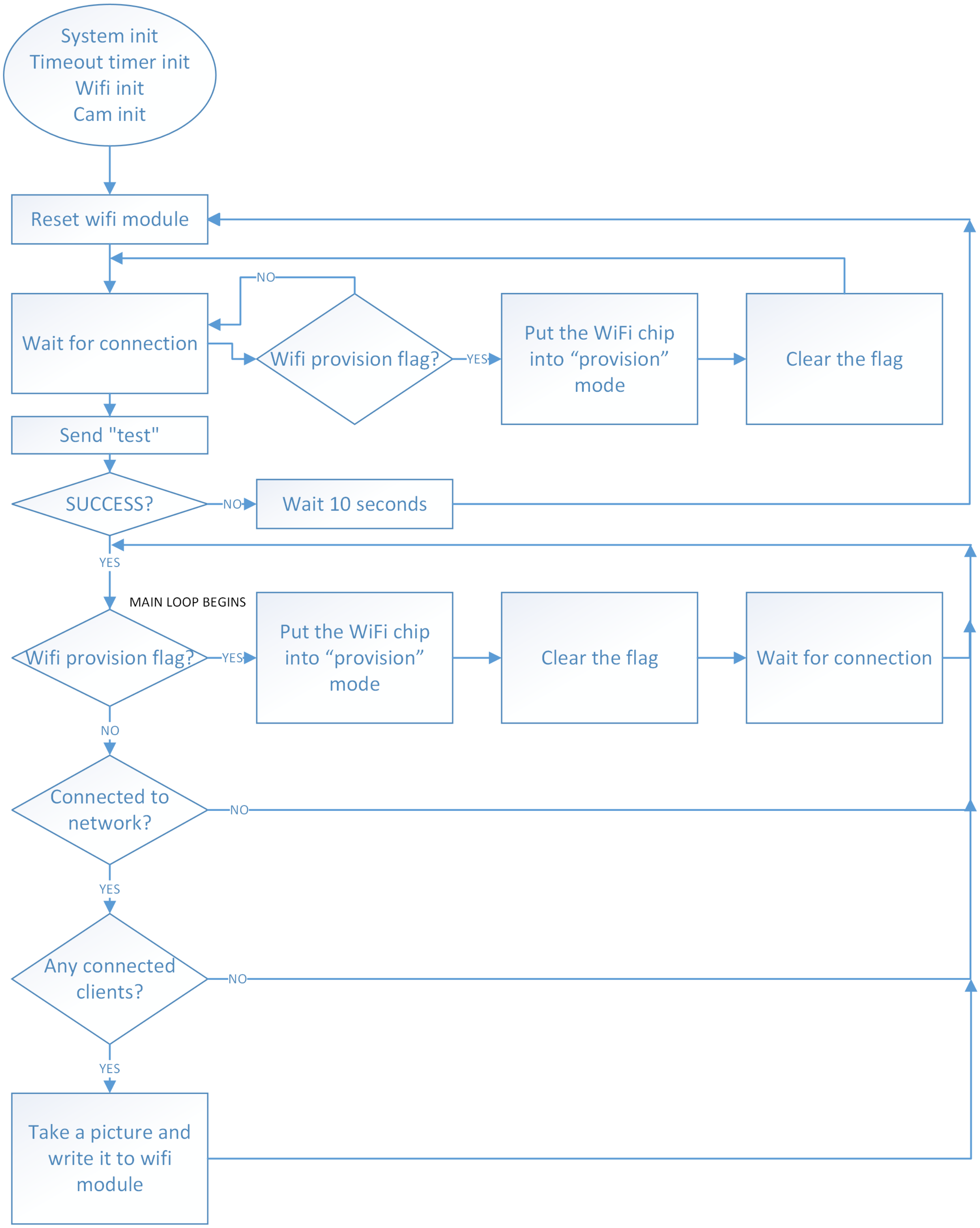
If you see this, then the image was probably captured successfully. Make sure your **find_image_len** function finds a non-zero image length.

To debug your MCU-to-WiFi interface, keep your terminal application open to be able to see the debugging responses of the WiFi module to the MCU commands. You should see acknowledgments of configuring the various functions of the WiFi module. You should also see a list of the files that are on the ESP32. When a client connects to the website, you should see that information, along with information about the images that are transferred from the MCU to the ESP32. A sample output is shown below. A more detailed view of the image transfer, using the Saleae logic analyzer, can be seen in Appendix C.

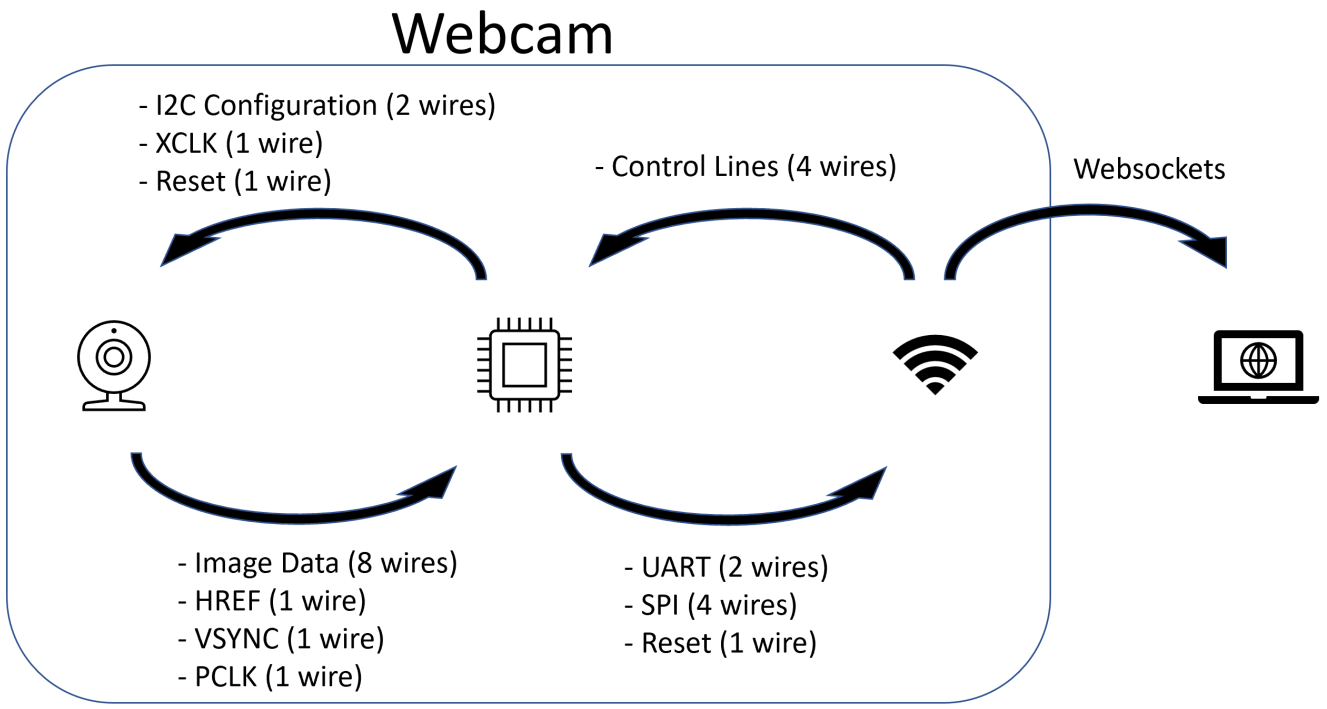


```
setting wlan gpio: 27
setting websocket gpio: 26
setting AP gpio: 25
setting command complete gpio: 21
Prefs: set comm_pin to 21
setting network gpio: 22
Prefs: set net_pin to 22
setting network gpio: 32
Prefs: set clients_pin to 32
ets Jul 29 2019 12:21:46

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1184
load:0x40078000,len:12784
load:0x40080400,len:3032
entry 0x400805e4
SPIFFS started. Contents:
        FS File: edit.html, size: 19.99KB
        FS File: edit_styles.css, size: 1.71KB
        FS File: success.html, size: 545B
        FS File: webcam.html, size: 621B
        FS File: webcam_functions.js, size: 2.94KB
        FS File: webcam_styles.css, size: 66B

Comm pin set to: 21
Network pin set to: 22
Connected clients pin set to: 32
SPI baud rate set to: 2000000
Starting task: blink_wlan
44:17:93:E2:52:2C
Starting task: blink_ap
Starting task: blink_websocket
blink_wlan - PIN STATE: 4, PIN NUM: 27
web setup AP: ESD1 522c
blink_ap - PIN STATE: 1, PIN NUM: 25
*wm:blink_websocket - PIN STATE: 2, PIN NUM: 26
[1] AutoConnect
Received image (11965 bytes)
*wm:[2] ESP32 event handler enabled
*wm:[2] Connecting as wifi client...
*wm:[2] setSTAConfig static ip not set, skipping
*wm:[1] Connecting to NEW AP: eduroam
*wm:[1] connectTimeout not set, ESP waitForConnectResult...
*wm:[2] Connection result: WL_CONNECTED
*wm:[1] AutoConnect: SUCCESS
*wm:[2] Connected in 2456 ms
*wm:[1] STA IP Address: 10.105.118.203
IP Address: 10.105.118.203
OTA ready

WebSocket server started.
mDNS responder started: http://esp32-webcam.local
HTTP server started.
WebSocket client #0 connected from 10.105.86.18 url: /
Total connected clients: 1
Received image (5957 bytes)
Received image (6762 bytes)
Received image (7132 bytes)
Received image (6111 bytes)
Received image (6567 bytes)
Received image (8139 bytes)
Received image (6182 bytes)
Received image (6522 bytes)
Received image (6439 bytes)
Received image (6395 bytes)
Received image (5639 bytes)
Received image (5667 bytes)
WebSocket client #0 disconnected
Total connected clients: 0
No connected clients
Received image (6072 bytes)
Received image (5333 bytes)
Received image (5641 bytes)
Received image (5078 bytes)
Received image (5870 bytes)
Received image (5675 bytes)
Received image (5446 bytes)
Received image (5652 bytes)
Received image (5898 bytes)
Received image (6104 bytes)
Received image (5849 bytes)
Received image (5793 bytes)
Received image (5697 bytes)
Received image (6094 bytes)
Received image (5932 bytes)
```

Appendix A: Program Flowchart



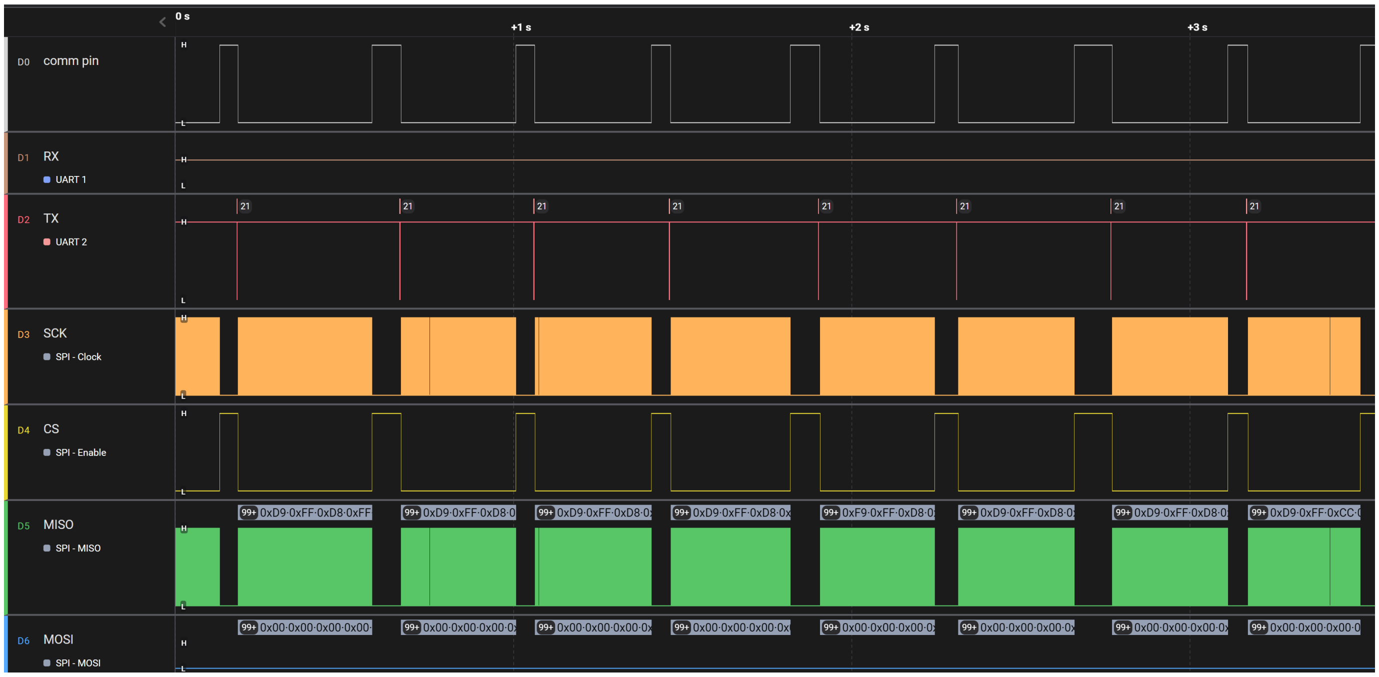
Appendix B: System Data Flow



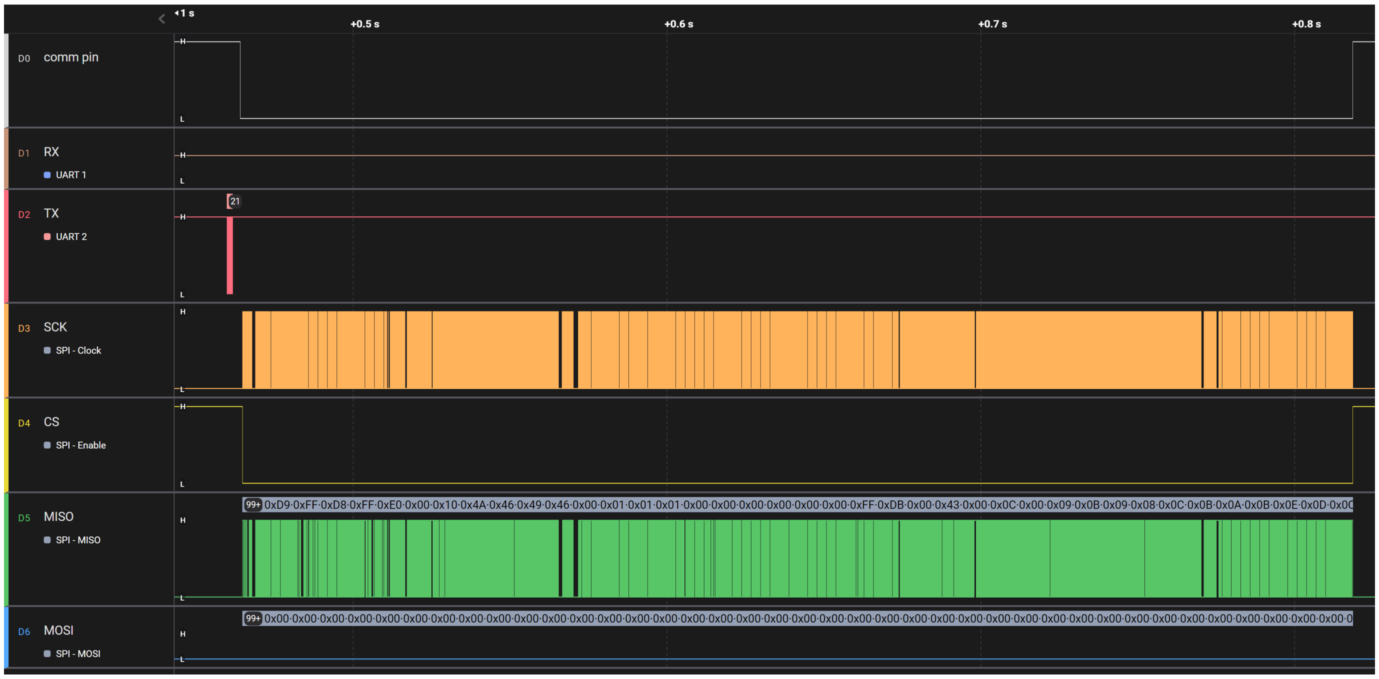
Appendix C: Communication with ESP32

The figures below show the MCU's "command complete", RX, TX, SCK, CS, MISO, and MOSI pins. NOTE!: Your camera may stop working when you connect the logic probe to MISO. To get the traces, disconnect MISO from the ESP32, and probe just the MCU MISO pin. While the images will not stream on the website, you should get the proper trace.

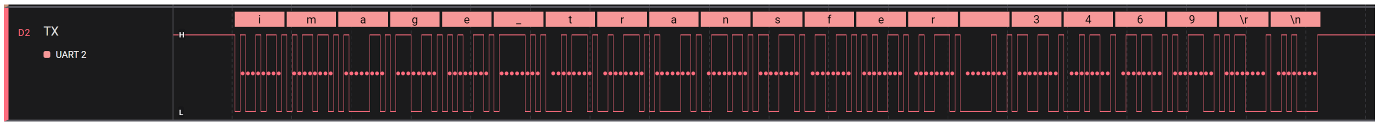
This figure shows the complete data flow while streaming images. You can see a brief message on the TX line indicating the start of an image, followed by the transmission of the image over SPI.



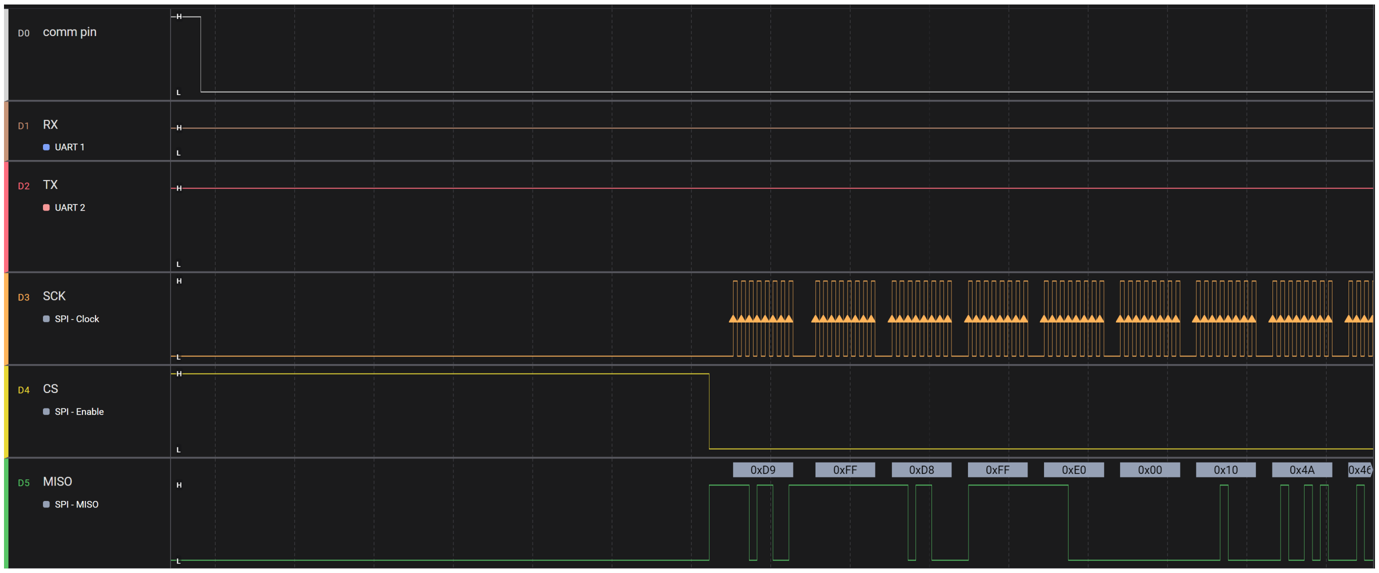
This figure shows the complete communication of one image. First, there is an "image_transfer" command on the TX line, followed by the start of the transfer on the SPI bus, as well as a falling edge on the "command complete" line. Then, the image is transferred. Afterwards, the ESP32 acknowledges the image transfer by setting the "command complete" line high.


+1 s
+0.5 s
+0.6 s
+0.7 s
+0.8 s
D0 comm pin
D1 RX
UART 1
D2 TX
UART 2
21
D3 SCK
SPI - Clock
D4 CS
SPI - Enable
D5 MISO
SPI - MISO
99+ 0xD9·0xFF·0xD8·0xFF·0xE0·0x00·0x10·0x4A·0x46·0x49·0x46·0x00·0x01·0x01·0x01·0x00·0x00·0x00·0x00·0x00·0x00·0xFF·0xDB·0x00·0x43·0x00·0x0C·0x00·0x09·0x0B·0x09·0x08·0x0C·0x0B·0x0A·0x0B·0x0E·0x0D·0x0C
D6 MOSI
SPI - MOSI
99+ 0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00·0x00
H
L

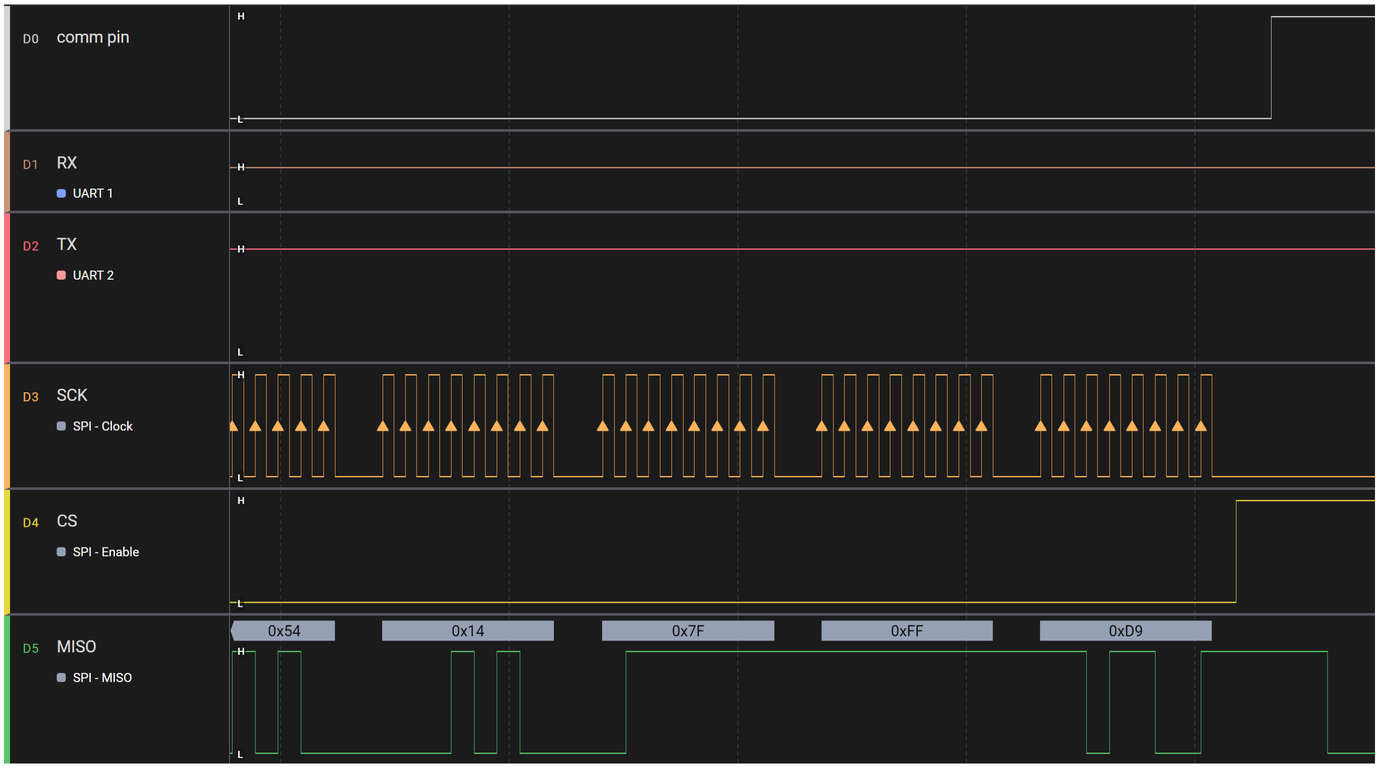
This figure shows a closeup of a communication that starts the image transfer.



This figure shows a closeup of the start of the image transfer, including the falling edge of the “command complete” line.



This figure shows a closeup of the end of the image transfer, including the rising edge of the “command complete” line.



FAQ

Note About Image Buffer

For your image buffer, use a preallocated array instead of a pointer (i.e. uint8_t im_buf[array_size], not uint8_t * im_buf).

Q. When it says “Reset the Wifi” in the main loop, is that a hard reset on the pin or just the command “reboot”?

A. That is a hard reset, since you have the WiFi pin connected to an MCU GPIO. I recommend pulling the pin low, waiting 100 ms, and then setting it high.

Q. What is meant by “configure camera pins” and “configure twi pins”. Is this different from the rest of the initialization functions we are calling? If we are supposed to use the gpio_configure_pin function, what flags should we use?

A. Configuring the pins means that you make the pins act as their appropriate peripheral instead of a general-purpose pin. For the flags, please refer to the OV7740 sample project. They have all of the proper flags.

Q. Microchip indicates that the connected device is not a SAM4S8B, but it is.

A. This means that the soldering is not complete somewhere, or that you may have connected a high voltage to the MCU’s 1.2V input. I saw this problem before when people accidentally shorted together the 3.0V and 1.2V input on the MCU.