# Development and Debugging of Katmai 24-Bit ADC

**Daniel Siegel**
danieledisonsiegel@gmail.com

**Chase Martin**
chase.h.martin@gmail.com

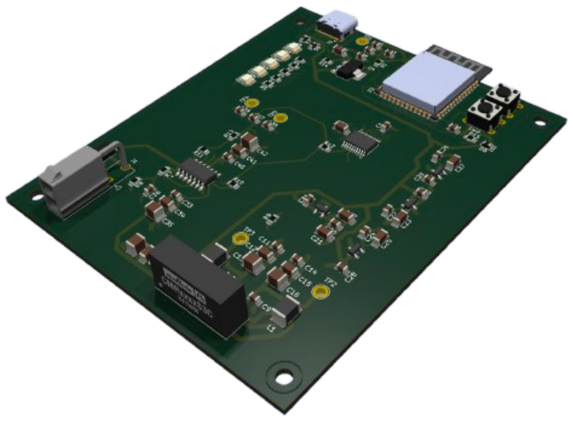


**Figure 1. ADC KiCad Rendering**

**ABSTRACT**

This document outlines the process of establishing communication with a 24-bit ADC prototype using an ESP32-S3 microcontroller. Initial efforts involved testing hardware connections with tools such as a signal analyzer, oscilloscope, and multimeter to verify ESP32 functionality, including USB CDC communication over D+/D-. Development steps included the use of the TinyUSB stack, Espressif's 'idf.py' for firmware uploads, and custom hardware abstraction layer code for SPI configuration. These methods faced compatibility issues that limited their effectiveness. The final implementation used modified Arduino libraries and the Arduino core for the ESP32, which enabled reliable SPI communication with the ADS131M02 ADC from Texas Instruments. This paper describes the challenges encountered, the steps taken to resolve them, and the final system configuration, concluding with recommendations for further debugging and refinement.

## Definitions

This section provides definitions for common acronyms and standard terms used throughout the document to ensure clarity and consistency.

- **ADC:** Analog-to-Digital Converter. A device that converts continuous analog signals into discrete digital values.
- **CDC:** Communication Device Class. A standard USB class that facilitates communication between devices, typically used for serial communication.
- **D+/D-:** Differential USB signal lines used for data communication in USB protocols.
- **ESP32:** A series of low-cost, low-power system-on-chip microcontrollers with integrated Wi-Fi and Bluetooth.
- **FTDI:** Future Technology Devices International. A company that designs USB-to-serial communication devices and drivers.
- **GPIO:** General-Purpose Input/Output. Pins on a microcontroller that can be configured as digital inputs or outputs.
- **HAL:** Hardware Abstraction Layer. A software layer that abstracts hardware-specific functions, enabling easier development and portability.
- **LED:** Light-Emitting Diode. A semiconductor device that emits light when current flows through it.
- **SPI:** Serial Peripheral Interface. A communication protocol commonly used for short-distance communication between a microcontroller and peripheral devices.
- **TinyUSB:** An open-source USB Host/Device stack designed for embedded systems.
- **TX/RX:** Transmit and Receive. Commonly used to describe data flow in serial communication.
- **USB:** Universal Serial Bus. A standard interface for communication between devices and a host controller.
- **VBUS:** Voltage Bus. The power line in a USB connection that supplies power to connected devices.

## INTRODUCTION

The main purpose of this documentation is to outline the process undertaken to design and implement the Katmai 24-bit ADC prototype, focusing on three main components: the analog system, USB CDC initialization and communication, and SPI communication. Each component presented unique challenges that required a systematic approach to troubleshooting and development. The analog system was designed to ensure both frontend and backend confirmation of analog signal integrity. The ADS131M02 ADC required an input voltage of a diffrential 2.5V, which necessitated careful design of the analog system to provide accurate voltage regulation and stable references. This was critical for ensuring the greatest signal integrity and reliable ADC performance. Special attention was given to voltage regulation levels for both the ADC and the analog system, which were tested and confirmed for consistent operation. The second component was USB CDC initialization and communication. This required setting up D+/D- pins and implementing the TinyUSB stack for communication. Enumeration of the USB device proved to be a significant challenge. Troubleshooting was conducted in two ways: analyzing packets using Wireshark and examining signals directly on the D+/D- pins with a logic analyzer. It was determined that the USB connector on the PCB was faulty, so a modified USB cable was used to connect directly to the ESP32 pins for GND, 5V (VBUS), and D+/D- lines. This approach successfully established a connection, and data transfer was enabled using 'tusb.h', 'tusb cdc', and 'tusb console' libraries to display data in the computer console. The third component involved SPI communication between the ADC and the ESP32-S3. Implementing this communication required configuring the ADS131M02 over SPI, which was achieved using modified Arduino libraries. These libraries enabled the Arduino core to facilitate seamless data transfer and control between the ESP32 and the ADC. Challenges such as data synchronization and clock signal stability were addressed during the implementation phase. This documentation outlines the steps taken to address these challenges and provides a guide for replicating the process in future designs.

- The Analog System: The design of the analog system posed several challenges, particularly in achieving the required a diffrential 2.5V input for the ADS131M02 while maintaining signal integrity. A single OPAMP IC was utilized to perform gain, unity, and inverting stages, each requiring distinct voltage references for stability. Cost-effective voltage regulation was a priority, leading to the selection of inexpensive yet accurate regulators. Trace length considerations, decoupling capacitors, and spatial arrangement on the PCB minimized noise, especially around the OPAMPs. For power, a Murata Power Solutions CMR0512S3C DC-DC converter provided a diffrential 12V with decoupling capacitors to ensure stability. Additional testing with a signal generator will be necessary to validate the differential filter design, as exact-specification capacitors were unavailable during initial development.

- USB CDC Initialization and Communication: USB device enumeration presented significant obstacles. While the VBUS and GND connections were intact, poor soldering practices caused scratches under the USB connector, leading to shorts between D+ and D-. This issue was identified after the device failed to enumerate, even with FTDI drivers installed on macOS. A custom USB cable modification was implemented to bypass the faulty connector, connecting the differential transfer lines directly to the board. This solution successfully enabled communication and device enumeration.

- SPI Communication: Configuring the SPI communication with the ADS131M02 was the most challenging aspect of the project. Initial attempts involved custom libraries and hardware abstraction layer code to work with ESP32's native SPI protocols but resulted in inconsistent device ID detection and unreliable outputs. The ADC's 24-bit signed integer data was difficult to interpret, even with reference to the datasheet. Ultimately, a switch to the Arduino core for ESP32 provided a simpler and more reliable SPI interface, enabling stable communication with the ADC.

## USB CDC INITIALIZATION AND COMMUNICATION

Establishing USB CDC communication is essential for enabling data transfer between the ESP32-S3 and a host computer. In this section, we will analyze the steps necessary to replicate this process by explaining the installation of drivers and addressing the challenges encountered during device enumeration. To begin, installing the appropriate FTDI drivers on macOS ensures that the host system can recognize USB devices. Despite proper driver installation, device enumeration may still fail due to hardware issues. In this project, scratches beneath the USB connector caused shorts between D+ and D-, resulting in communication failure. To resolve this, a custom USB cable was constructed to bypass the faulty connector. Differential transfer lines (D+ and D-) were wired directly to the ESP32 board, alongside GND and VBUS for stable power delivery. After this modification, the device successfully enumerated, and communication was established. This process highlights the importance of both software and hardware debugging to achieve reliable USB CDC communication.

### USB CDC Initialization and Troubleshooting

Establishing USB CDC communication with the ESP32-S3 WROOM2 required a systematic approach to troubleshooting due to several challenges during development. This section provides a detailed explanation of the steps taken to identify and resolve these issues, enabling reliable communication between the microcontroller and the host computer.

The first step was to use the macOS System Information utility to determine if the device was being enumerated correctly. When this failed, Wireshark was employed to monitor USB packet transfers. Configuring Wireshark to detect USB inputs on macOS required disabling certain security features by entering safe boot mode. Detailed instructions for this process can be found online by searching for methods to monitor USB/Serial inputs using Wireshark. This tool proved essential for diagnosing USB communication, especially when no console output was available.

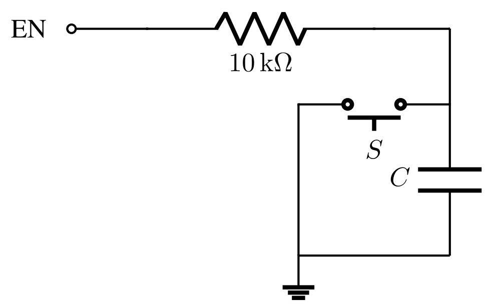


**Figure 2. Original Circuit**

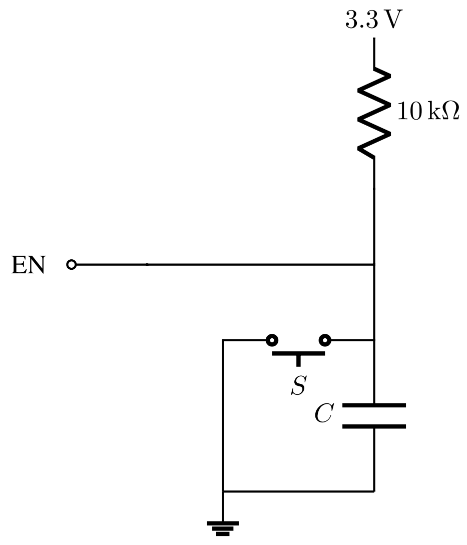


**Figure 3. Modified Circuit**

The next step involved circuit analysis. Using a multimeter, all ESP32 circuit pins were tested to ensure there were no shorts. Upon consulting the ESP32-S3 datasheet, it was discovered that the enable pin (EN) required a pull-up resistor to 3.3V (Figures 2 and 3). A standard resistor was soldered from a 3.3V test point to the EN pin, and the existing capacitor and resistor pair for the EN circuit was removed. This modification allowed the ESP32 to reset correctly and enter bootloader/download mode.

Further validation was conducted using an oscilloscope to measure voltage curves on the reset and boot pins (GPIO0 and EN). These measurements confirmed that the ESP32 was turning on properly. However, data inconsistencies persisted on the tx/rx pins when analyzed using a Saleae Logic Pro 8 logic analyzer. These inconsistencies verified that the ESP32 was operational, as Espressif's signature and device information were successfully transmitted.

The focus then shifted to the D+/D- pins, where data was analyzed using Saleae's logic software. By performing resets and monitoring USB CDC data transfers, it was confirmed that "KEEP ALIVE" packets were being sent. This eliminated the ESP32 as the source of the problem and identified the USB connector on the PCB as the culprit. Scratches on the PCB near the connector caused shorts between D+, D-, and ground, rendering the connector unreliable. A temporary fix involved insulating the connector area, but connection issues persisted.

To bypass the faulty connector, a custom USB Type A to Type C cable was constructed. The 5V (VBUS), ground (GND), and differential D+/D- wires were manually connected directly to the corresponding ESP32 pins. While challenging due to the small pin size, this approach successfully enabled the computer to detect the device. Once detected, custom USB CDC drivers were installed from **FTDI's website**, allowing data transmission to and from the ESP32.

Next, LEDs connected via resistors to the GPIO matrix provided visual debugging. Status indicator LEDs confirmed communication with the ESP32, and a "heartbeat LED" was added to verify operational status even without communication.

Using the TinyUSB libraries ('tusb.h', 'tusb_cdc', and 'tusb_console'), USB CDC debugging was configured. The ESP-IDF menuconfig was set to use Octal Flash and CDC communication instead of the default JTAG/Serial options. Adjustments to the 'sdkconfig' file included increasing the IRAM frequency from 100 Hz to 1000 Hz. Debugging output was then monitored through idf.py.

To integrate the ADS131M02 ADC library, modifications were made to the CMake configuration files to include the custom library from GitHub and adapt it for the ESP32 core. Additional steps included setting up managed components for the ESP32 core.

To replicate this process, follow these steps:

1. Install the necessary dependencies by navigating to the ESP-IDF folder in Terminal and running:
   ```
   . ./install.sh
   . ./export.sh
   ```
2. Navigate to the project directory (e.g., '/examples/peripherals/usb/device/-tusb_serial_device') and build the project using:
   ```
   idf.py build
   ```
3. Identify available serial ports with:
   ```
   ls /dev/cu.*
   ```
4. Flash the firmware to the ESP32 using:
   ```
   idf.py flash monitor -p /dev/cu.usbmodemX
   ```
   (Replace 'X' with the device identifier from the previous step.)
5. Monitor the console for debug output and ensure SPI communication is operational.

These steps outline a robust method for resolving USB CDC initialization issues and ensuring reliable operation for future projects.

## Analysis of Circuit Integrity Testing of PCB

The integrity of the PCB was tested to ensure all components were functioning as intended, with a particular focus on the enable and reset pins of the ESP32. The enable pin was tested alongside the reset pin (GPIO0) using an oscilloscope. Both channels of the oscilloscope were connected to a common ground, with CH1 connected to the enable pin and CH2 connected to the reset pin. The oscilloscope was configured with a voltage detection range of up to 5 volts and set to time-based mode to observe the voltage patterns over time. The analysis was supplemented by viewing key metrics such as $V_{\text{Max}}$ and $V_{\text{Avg}}$.

During the startup process, the voltage at both the enable and reset pins was monitored. It was observed that the voltages exhibited a stable RF nonlinear increase, indicative of the ESP32's typical voltage handling during startup. This behavior, influenced by the capacitor configuration, confirmed that both pins were functioning correctly.

Further testing focused on the 3.3V voltage supply to ensure the ESP32 received sufficient power. CH1 of the oscilloscope was attached to the 3.3V pin, and the voltage was analyzed while the device was powered via the USB-C connector. The results confirmed that the 3.3V voltage regulator was operating as expected, providing stable power to the ESP32.

To ensure further integrity of the circuit, specific debugging considerations were noted. One important step is to inspect all solder joints under a circuit microscope. This allows for verification that no ESP32 pins are shorting. Similarly, all SMD resistors should be checked using a multimeter to ensure they are not shorting. The use of 3.3V test points is crucial to confirm that the voltage supplied to the ESP32 is precisely 3.3V or within a 1% tolerance. Such testing ensures that the voltage regulator and associated components are functioning correctly.

To sum up, circuit integrity testing involves thorough inspection and analysis of key pins, components, and power delivery systems. The oscilloscope analysis and debugging steps outlined here are essential to validate proper operation and ensure a reliable design. The use of testing points provides an efficient and systematic approach to debugging and validation, ensuring all components perform as anticipated.

## Additional Details on Oscilloscope Testing

The oscilloscope played a critical role in verifying the behavior of the ESP32's key pins and ensuring proper power delivery during startup. This section provides a detailed explanation of the setup and measurement process, along with additional insights into how the oscilloscope was used.

*Setup and Measurement Configuration:*

The oscilloscope used in this testing was a four-channel model, configured to monitor voltage levels at key points on the PCB. CH1 was connected to the enable (EN) pin, while CH2 monitored the reset (GPIO0) pin. Both channels shared a common ground to ensure accurate differential voltage measurements. The voltage detection range was set to 5V, and the time base was configured to capture the full duration of the ESP32's startup sequence, including the capacitor charging behavior.

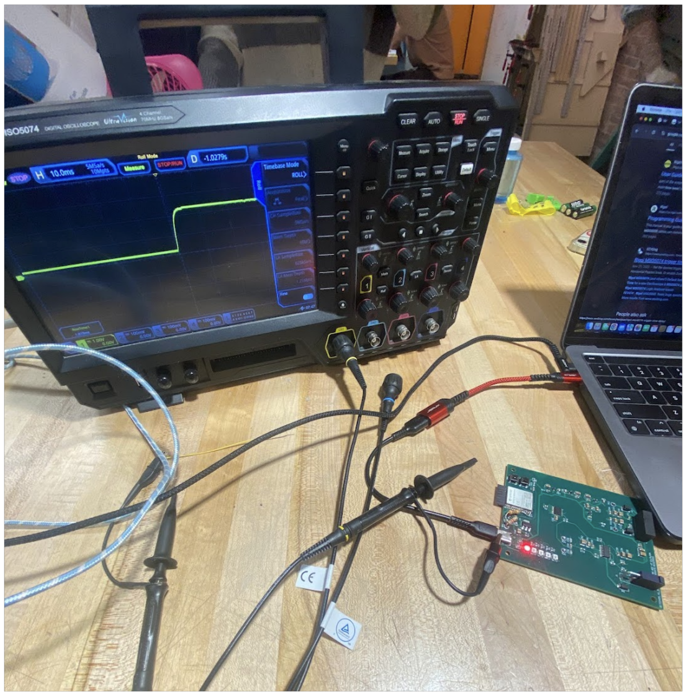


**Figure 4. Troubleshooting Boot/Enable Pins with Oscilloscope**

Figure 4 illustrates the setup of the oscilloscope probes on the PCB and the waveform outputs observed during testing. The measurement was performed while powering the ESP32 via USB-C to ensure that the power delivery system was functioning as intended.

*Observations:*

The oscilloscope captured a stable and consistent voltage ramp-up on both the EN and GPIO0 pins during startup. The waveform indicated the expected nonlinear increase, attributed to the charging of capacitors in the circuit. These results confirmed that the ESP32 was entering its boot sequence correctly, with no interruptions or irregularities in the power delivery. The oscilloscope was also used to monitor the 3.3V power supply pin during operation. The measurements verified that the voltage regulator consistently provided 3.3V within a 1% tolerance, even under load. This ensured that the ESP32 was supplied with adequate and stable power for proper functionality. The use of the oscilloscope allowed for precise diagnosis of startup and power delivery issues. By analyzing waveform patterns and voltage stability, potential faults, such as capacitor failures or irregular voltage regulation, were ruled out. This step was essential in validating the integrity of the ESP32 circuit and identifying any components requiring further inspection. The oscilloscope proved invaluable for capturing real-time data and ensuring that all electrical signals were within specification, providing confidence in the PCB's overall design and operation.

## CUSTOM LIBRARIES AND MODIFICATIONS

To adapt the open-source 'ADS131M0x.h' library for use with the ESP32 core, significant modifications were made to both the header and source files. The original library, designed for the Arduino environment, was rewritten to leverage the ESP-IDF core and ensure compatibility with ESP32 hardware. This section outlines the steps taken to modify the library and details the updated functionality.

### Header File Adjustments

The 'ADS131M0x.h' header file required several changes to integrate ESP32-specific dependencies and support its hardware capabilities:

- Replaced Arduino-specific includes with ESP-IDF libraries, such as `spi_master.h` and `gpio.h`.
- Introduced additional macros for ESP32 pin assignments and configuration constants, including definitions for clock frequency and GPIO pins.
- Revised function prototypes to align with the ESP-IDF core, including new initialization and configuration functions.

These adjustments allowed for precise control of SPI communication and GPIO behavior within the ESP32 environment. The modified header file introduced improved functionality and alignment with the ESP32's architecture.

### Source File Modifications

The source file, `ADS131M0x.cpp`, was extensively rewritten to replace Arduino SPI and GPIO functions with their ESP-IDF counterparts. The key changes included:

- Implemented the ESP32's LEDC peripheral to generate an 8.192 MHz clock signal, replacing Arduino-based timing functions.
- Replaced SPI communication functions with ESP-IDF's `spi_master` API to enable efficient and accurate data transfers.
- Enhanced error handling and logging using the ESP-IDF logging macros, such as `ESP_LOGE()` and `ESP_LOGI()`.
- Refactored GPIO handling to use the ESP-IDF `gpio_config()` function for configuring chip select (CS) and data ready (DRDY) pins.

#### *Generating the Clock Signal*

The ESP32's LEDC peripheral was used to generate the required 8.192 MHz clock signal for the ADS131M0x. This involved configuring the LEDC timer and channel:

```
void ADS131M0x::configureCLKIN() {
    ledc_timer_config_t ledc_timer = {
        .speed_mode = LEDC_LOW_SPEED_MODE,
        .duty_resolution = LEDC_TIMER_1_BIT,
        .timer_num = LEDC_TIMER_0,
        .freq_hz = 8192000,
        .clk_cfg = LEDC_AUTO_CLK
    };
```

```
    ledc_channel_config_t ledc_channel = {
        .gpio_num = CLKIN_GPIO_PIN,
        .speed_mode = LEDC_LOW_SPEED_MODE,
        .channel = LEDC_CHANNEL_0,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER_0,
        .duty = 1,
        .hpoint = 0
    };
    ledc_timer_config(&ledc_timer);
    ledc_channel_config(&ledc_channel);
}
```

#### *SPI Communication Adjustments*

The Arduino SPI functions were replaced with ESP-IDF's SPI API. This ensured compatibility with the ESP32's SPI peripheral and improved the reliability of data transfers. For example, the `sendCommand()` function was rewritten to use `hal_spi_transfer()`:

```
bool ADS131M0x::sendCommand(uint16_t command) {
    uint8_t tx_data[2] = { static_cast<uint8_t>(command >> 8), static_cast<
          uint8_t>(command & 0xFF) };
    hal_spi_toggle_cs(cs_pin, false); // Pull CS low
    bool result = hal_spi_transfer(spi_handle, tx_data, nullptr, sizeof(tx_data))
          ;
    hal_spi_toggle_cs(cs_pin, true);  // Pull CS high

    if (!result) {
        ESP_LOGE(TAG, "Failed␣to␣send␣SPI␣command:␣0x%04X", command);
        return false;
    }
    ESP_LOGI(TAG, "Command␣0x%04X␣sent␣successfully.", command);
    return true;
}
```

#### *Initialization and Configuration*

The `begin()` function was rewritten to configure SPI and GPIO pins using ESP-IDF functions. The initialization sequence included setting GPIO modes and attaching the SPI device to the host:

```
bool ADS131M0x::begin(spi_host_device_t host, gpio_num_t sck, gpio_num_t miso,
                      gpio_num_t mosi, gpio_num_t cs, gpio_num_t drdy,
                      gpio_num_t reset, gpio_num_t sync) {
    hal_gpio_config(cs, GPIO_MODE_OUTPUT, false, false);
    hal_gpio_config(drdy, GPIO_MODE_INPUT, true, false);
    if (!hal_spi_init(host, sck, miso, mosi, 1000000)) {
        ESP_LOGE(TAG, "Failed␣to␣initialize␣SPI.");
        return false;
    }
    if (!hal_spi_attach_device(host, cs, 1000000, &spi_handle)) {
        ESP_LOGE(TAG, "Failed␣to␣attach␣SPI␣device.");
        hal_spi_deinit(host);
        return false;
    }
    return true;
}
```

These modifications transformed the library into a tool optimized for the ESP32 platform, providing robust functionality for data acquisition and ADC control. The rewritten library demonstrated improved performance and compatibility in practical applications.

## TROUBLESHOOTING AND MODIFICATIONS

During the process of adapting the 'ADS131M0x' library to the ESP32 core, several issues were encountered, primarily related to the SPI communication protocol and unlock commands. This section outlines the troubleshooting steps, the core functionality of the associated 'hal.c' and 'hal.h' files, and how these issues were resolved.

### SPI Communication and Unlock Command Issue

The primary issue arose during SPI communication. While the device was recognized on the SPI bus, sending an unlock command consistently resulted in a device ID of `0x0000`. Verbose logging was enabled to trace all SPI communication, revealing discrepancies in how the custom Hardware Abstraction Layer (HAL) handled SPI transactions compared to the expectations of the ADS131M0x chip.

## HAL Overview and Modifications

The 'hal.c' and 'hal.h' files, originally adapted from TI's libraries for their custom chips, required modifications to accommodate the ESP32's SPI and GPIO handling. These changes aimed to align the HAL functionality with the ESP32 hardware and the ADS131M0x's communication requirements.

### *Core Functionality of* `hal.c`

The `hal.c` file provides implementations for GPIO configuration, SPI initialization, and data transfer. Key functions include:

- **GPIO Configuration:** Configures pins for input, output, or pull-up/down resistors using ESP-IDF's `gpio_config()`.

```
void hal_gpio_config(gpio_num_t pin, gpio_mode_t mode, bool pullup, bool
        pulldown) {
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << pin),
        .mode = mode,
        .pull_up_en = pullup ? GPIO_PULLUP_ENABLE : GPIO_PULLUP_DISABLE,
        .pull_down_en = pulldown ? GPIO_PULLDOWN_ENABLE :
                GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    esp_err_t err = gpio_config(&io_conf);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "Failed to configure GPIO %d: %s", pin, esp_err_to_name
                (err));
    }
}
```

- **SPI Initialization and Transfer:** Initializes the SPI bus using `spi_bus_initialize()` and performs data transfers with `spi_device_transmit()`.

```
bool hal_spi_transfer(spi_device_handle_t handle, const uint8_t *tx_data,
        uint8_t *rx_data, size_t len) {
    spi_transaction_t transaction = {
        .length = len * 8,
        .tx_buffer = tx_data,
        .rx_buffer = rx_data,
        .flags = 0
    };
    esp_err_t ret = spi_device_transmit(handle, &transaction);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "SPI_transfer_failed:_%s", esp_err_to_name(ret));
        return false;
    }
    return true;
}
```

- **CS Pin Handling:** Controls the chip select pin state for SPI transactions, ensuring proper timing.

```
void hal_spi_toggle_cs(gpio_num_t cs_pin, bool state) {
    gpio_set_level(cs_pin, state ? 1 : 0);
    delay_us(2);  // Ensure CS pin timing meets ADS131M0x requirements
}
```

### *Core Functionality of* `hal.h`

The header file defines constants, data structures, and function prototypes used across the HAL. Notable features include:

- **GPIO Configuration Table:** Predefines GPIO settings for SPI and control signals.

```
const gpio_config_entry_t gpio_config_table[] = {
    {DRDY_GPIO_PIN, GPIO_MODE_INPUT, true, false}, // DRDY: Input with pull-
            up
    {CS_GPIO_PIN, GPIO_MODE_OUTPUT, false, false}, // CS: Output
    {SYNC_RESET_GPIO_PIN, GPIO_MODE_OUTPUT, false, true}, // RESET: Output
            with pull-down
    {ADC_SPI_CLK, GPIO_MODE_OUTPUT, false, false}, // SPI CLK: Output
    {ADC_MISO, GPIO_MODE_INPUT, true, false}, // MISO: Input with pull-up
    {ADC_MOSI, GPIO_MODE_OUTPUT, false, false}, // MOSI: Output
    {GPIO_NUM_MAX, GPIO_MODE_DISABLE, false, false} // End of table marker
};
```

- **SPI Function Prototypes:** Declares functions for initializing and interacting with the SPI bus.

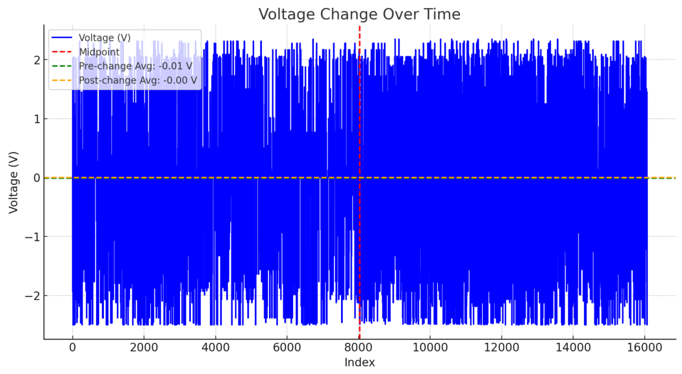


**Figure 5. SPI communication using the remade ADS131M0x library, showing trends under voltage application but unable to send phase shift commands.**

```
bool hal_spi_init(spi_host_device_t host, gpio_num_t sck, gpio_num_t miso,
        gpio_num_t mosi, uint32_t clk_speed_hz);
bool hal_spi_attach_device(spi_host_device_t host, gpio_num_t cs, uint32_t
        clk_speed, spi_device_handle_t *handle);
void hal_spi_detach_device(spi_device_handle_t handle);
bool hal_spi_transfer(spi_device_handle_t handle, const uint8_t *tx_data,
        uint8_t *rx_data, size_t len);
```

## Resolution

The issue was traced to a mismatch in SPI mode and timing between the TI-based HAL and the ESP32. Modifications were made to:

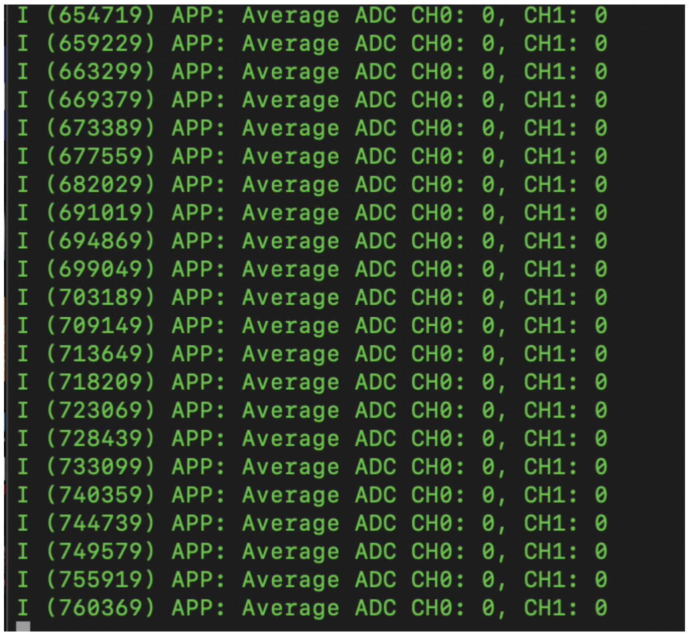
- Adjust SPI mode to `CPOL=0, CPHA=1` to match ADS131M0x requirements.
- Add precise delays using `esp_rom_delay_us()` to ensure compliance with the device's timing specifications.
- Improve verbose logging to track all SPI transactions and debug the unlock command sequence.

## Achieving SPI Communication

The process of establishing SPI communication involved several stages of development, debugging, and evaluation of potential solutions. Initially, the ESP32's hardware abstraction layer (HAL) was modified alongside custom libraries to support the ADS131M0x ADC. During testing with the remade ADS131M0x library for ESP32, some SPI communication was successfully established, as shown in Figure 5. The results displayed a trend when voltage was applied to the ADC input pins, indicating partial functionality. However, the system was unable to send the required phase shift command, rendering the implementation unusable for further tasks. Despite this limitation, the results confirmed that SPI communication could be established.

To address these limitations, the next steps involved modifying the HAL (hardware abstraction layer) files to refine how SPI commands were handled and transmitted by the ESP32. This approach aimed to ensure consistent communication and full functionality of the ADC over the SPI bus.

During initial testing with a modified `hal.h`, the device was partially recognized, and some SPI communication was established. However, the modifications to the custom libraries



```
I (654719) APP: Average ADC CH0: 0, CH1: 0
I (659229) APP: Average ADC CH0: 0, CH1: 0
I (663299) APP: Average ADC CH0: 0, CH1: 0
I (669379) APP: Average ADC CH0: 0, CH1: 0
I (673389) APP: Average ADC CH0: 0, CH1: 0
I (677559) APP: Average ADC CH0: 0, CH1: 0
I (682029) APP: Average ADC CH0: 0, CH1: 0
I (691019) APP: Average ADC CH0: 0, CH1: 0
I (694869) APP: Average ADC CH0: 0, CH1: 0
I (699049) APP: Average ADC CH0: 0, CH1: 0
I (703189) APP: Average ADC CH0: 0, CH1: 0
I (709149) APP: Average ADC CH0: 0, CH1: 0
I (713649) APP: Average ADC CH0: 0, CH1: 0
I (718209) APP: Average ADC CH0: 0, CH1: 0
I (723069) APP: Average ADC CH0: 0, CH1: 0
I (728439) APP: Average ADC CH0: 0, CH1: 0
I (733099) APP: Average ADC CH0: 0, CH1: 0
I (740359) APP: Average ADC CH0: 0, CH1: 0
I (744739) APP: Average ADC CH0: 0, CH1: 0
I (749579) APP: Average ADC CH0: 0, CH1: 0
I (755919) APP: Average ADC CH0: 0, CH1: 0
I (760369) APP: Average ADC CH0: 0, CH1: 0
```

**Figure 6. SPI communication after modifying `hal.h`, showing no data extracted due to device ID mismatch.**

for the ADC, which were tailored for ESP32 core compatibility, appeared to disrupt the data transfer. The only meaningful response obtained during this phase was a device ID of `0x0000`, indicating that communication with the ADC was not functioning correctly. When a voltage differential was applied to the ADC's input pins, no significant trend or meaningful data could be observed, and the output remained incoherent.

Further modifications to `hal.h` were attempted to improve the SPI communication. However, these changes resulted in a complete inability to extract data from the ADC. Despite the SPI bus being properly initialized and the ADC device being registered, a mismatch in the device ID prevented successful communication. Figure 6 shows the response from the ADC during this phase, where no meaningful data was captured, and attempts to unlock the ADC for communication failed.

Debugging logs provided further insight into the issue:

```
I (1946) APP: Logging raw ADC data:
I (1946) ADS131M0x: Raw ADC data: 00 00 00 00 00 00 00 00
    00 00 00 00
I (1947) APP: Logging ADC channel configuration:
I (1948) ADS131M0x: CH0_CFG: 0x0000, CH1_CFG: 0x0000
```

These logs indicated that the ADC initialization process within the modified hardware abstraction layer was failing to configure the device or establish communication properly. At this point, further efforts to modify the custom library were halted, and a simpler solution was evaluated.

### *Arduino Core Usage*

To streamline SPI bus initialization, we decided to use the Arduino core on the ESP32. This approach allowed for more straightforward implementation of SPI communication using existing libraries. However, integrating Arduino core required overcoming challenges with conflicting libraries, particularly related to USB communication.

Our solution involved utilizing TinyUSB within the Arduino IDE for USB communication. The ESP32 was initialized and enumerated using TinyUSB CDC and TinyUSB console libraries, with data sent over a custom serial syntax called `MySerial`. The custom syntax allowed for precise control of the communication process while resolving conflicts between libraries. Using this method, SPI communication was successfully established, providing a simpler yet effective solution compared to the modified `hal.h` approach.

## Code Interpretation and Implementation

This section interprets the code used for achieving SPI communication and USB CDC functionality using the Arduino core on the ESP32. The changes made from the earlier custom HAL-based implementation to the Arduino core are discussed, along with challenges encountered and solutions applied.

### *Library Imports and Initial Setup*

```
#include <Arduino.h>
#include "tusb.h"        // TinyUSB for USB CDC
    functionality
#include "cdcusb.h"      // USB CDC header for USB
    communication
#include "ADS131M0x.h" // Include ADS131M0x library
```

The code begins with importing necessary libraries. The `tusb.h` and `cdcusb.h` libraries provide TinyUSB support for USB CDC functionality, enabling USB communication between the ESP32 and the host system. The `ADS131M0x.h` library facilitates communication with the ADC over the SPI bus. Unlike the HAL-based approach, this implementation uses the Arduino core, which simplifies the integration of these functionalities.

Challenges encountered include non-compatibility between the ESP32 core and certain CDC libraries. To resolve this, the `esp32-usb-v2` library from GitHub was installed as a `.zip` file in the Arduino IDE, and conflicting libraries were downgraded.

### *Pin Definitions*

```
#define LED_RF          4
#define LED_USB         5
#define LED_STATUS      6
#define LED_FAULT       7
#define ADC_CS          18
#define ADC_MISO        19
#define ADC_SPI_CLK     20
#define ADC_MOSI        21
#define ADC_SYNC_RESET  15
#define ADC_DRDY        16
```

The pin definitions map GPIO pins on the ESP32 to the respective hardware components. These include LEDs for status indication and SPI pins for communication with the ADC. Compared to the earlier implementation, these mappings remain consistent, but their usage is simplified by the Arduino core.

### *GPIO Initialization*

```
void init_gpio() {
    pinMode(LED_RF, OUTPUT);
    pinMode(LED_USB, OUTPUT);
    pinMode(LED_STATUS, OUTPUT);
    pinMode(LED_FAULT, OUTPUT);
    digitalWrite(LED_FAULT, LOW);

    pinMode(ADC_SYNC_RESET, OUTPUT);
    pinMode(ADC_DRDY, INPUT_PULLUP);
}
```

The `init_gpio` function configures the GPIO pins for their respective roles. LEDs are set as outputs for status indication, while the ADC pins are configured for SPI communication. This is similar to the HAL-based approach but uses Arduino's `pinMode()` and `digitalWrite()` functions for simplicity.

### *USB CDC Initialization*

```
void init_usb() {
    MySerial.setCallbacks(new MyUSBCallbacks());
    if (!MySerial.begin()) {
        Serial.println("Failed to start USB CDC");
        return;
    }
    MySerial.println("TinyUSB CDC initialized.");
}
```

The `init_usb` function initializes USB CDC communication using the TinyUSB library. A custom `MyUSBCallbacks` class provides callback functions for handling USB events such as data reception and connection state changes. The use of TinyUSB simplifies USB communication compared to implementing it in a HAL-based environment.

### *ADC Initialization*

```
void init_adc() {
    MySerial.println("Initializing ADS131M0x...");

    adc.setClockSpeed(200000);  // Set SPI clock speed
    adc.reset(ADC_SYNC_RESET);  // Reset ADC
    adc.begin(&SpiADC, ADC_SPI_CLK, ADC_MISO, ADC_MOSI,
         ADC_CS, ADC_DRDY);

    // Configure both channels
    adc.setInputChannelSelection(0,
         INPUT_CHANNEL_MUX_AIN0P_AIN0N);
    adc.setInputChannelSelection(1,
         INPUT_CHANNEL_MUX_AIN0P_AIN0N);

    MySerial.println("ADS131M0x initialized.");
}
```

The `init_adc` function initializes the ADC. It sets the SPI clock speed, resets the ADC, and establishes SPI communication using the `ADS131M0x` library. This contrasts with the earlier HAL-based approach, where SPI communication required significant customization. Here, the Arduino core simplifies SPI initialization and communication.

### *ADC Task for Data Acquisition*

```
void adcTask() {
    if (adc.isDataReady()) {
        uint32_t rawCh0 = adc.readADC().ch0;
        uint32_t rawCh1 = adc.readADC().ch1;

        int32_t signedCh0 = (rawCh0 & 0x800000) ? (rawCh0
             | 0xFF000000) : rawCh0;
        int32_t signedCh1 = (rawCh1 & 0x800000) ? (rawCh1
             | 0xFF000000) : rawCh1;

        signedCh0 -= offsetCorrectionCh0;
        signedCh1 -= offsetCorrectionCh1;

        int32_t difference = signedCh0 - signedCh1;

        MySerial.printf("ADC-Ch0: %d\n", signedCh0);
        MySerial.printf("ADC-Ch1: %d\n", signedCh1);
        MySerial.printf("ADC-Diff: %d\n", difference);
    } else {
        MySerial.println("ADC data not ready. Retrying..."
            );
    }
}
```

The `adcTask` function reads data from the ADC channels and calculates the difference between them. Raw data is converted to signed integers, and offset corrections are applied. Results are sent to the USB CDC interface for debugging. This task leverages the Arduino core's SPI support, simplifying data handling compared to the HAL-based approach.

### *LED Task for Status Indication*

```
void led_task() {
    digitalWrite(LED_STATUS, HIGH);
    delay(250);
    digitalWrite(LED_STATUS, LOW);
    delay(250);

    if (data_received) {
        digitalWrite(LED_RF, HIGH);
        delay(83);
        digitalWrite(LED_RF, LOW);
        delay(83);
        data_received = false;
    }
}
```

The `led_task` function controls the status LEDs. It provides visual feedback for the system's operation, such as indicating data reception or system readiness.

### *Setup and Loop Functions*

```
void setup() {
    Serial.begin(115200);
    init_gpio();
    init_usb();
    init_adc();
    MySerial.println("Setup complete. Starting tasks...");
}

void loop() {
    adcTask();
    led_task();

    while (Serial.available()) {
        int len = Serial.available();
        char buf[len];
        Serial.read(buf, len);
        MySerial.write((uint8_t*)buf, len);
    }
}
```

The `setup` function initializes GPIOs, USB CDC, and ADC, while the `loop` function continuously executes the ADC and LED tasks. Additionally, it handles USB CDC communication, echoing data between the serial interface and the USB CDC interface.

—

### *Challenges and Solutions*

One major challenge was the non-compatibility between the ESP32 core and the CDC libraries, which required some libraries to be downgraded. The `esp32-usb-v2` library from GitHub was installed to address this issue. Furthermore, conflicts between SPI and USB CDC functionalities were resolved by leveraging the Arduino core's simplified initialization routines.

This implementation successfully established SPI communication and USB CDC functionality, highlighting the advantages of the Arduino core over the custom HAL-based approach.

### Conclusion and Next Steps

The debugging and troubleshooting process for the ESP32 microcontroller and ADS131M0x ADC highlighted the complexity of integrating hardware and software systems in embedded applications. By tackling USB communication, SPI communication, and system integration challenges, significant progress was made toward achieving a functional system. The use of TinyUSB for USB CDC communication and the Arduino core for SPI communication streamlined the development process, resolving many of the issues encountered with custom hardware abstraction layers. However, certain challenges remain unresolved, including proper decoding of 24-bit data and implementing phase shift commands.

The debugging process involved significant collaboration and effort:

- Two individuals were involved in debugging, one working remotely and the other onsite in a Makerspace.
- The process spanned six days and required over 60 hours of active work.

Key resources utilized during the debugging process included:

- Saleae Logic Pro 8 Logic Analyzer
- Rigol four-channel oscilloscope
- Soldering station for PCB modifications
- ChatGPT o1-mini for hardware abstraction layer code assistance

While the system demonstrates consistent ADC responses to voltage changes and successfully reads differential values from channels 0 and 1, the following next steps are crucial for completing the project:

1. **Figure out how to send the phase compensation commands over SPI** to ensure the ADC outputs accurate and reliable data.
2. Resolve compatibility issues with interpreting 24-bit signed integers in 32-bit operations, specifically addressing potential bit shift inaccuracies.
3. Decode the SPI data using the Texas Instruments library and the ADS131M0x documentation to validate and interpret ADC outputs correctly.

These steps are straightforward but will require thorough research and iterative testing to ensure the system functions as intended. Successfully addressing these challenges will finalize the integration process and establish a robust communication system for the ADC and ESP32.

### REFERENCES

1. FTDI Chip, *VCP Drivers*, available at: `https://ftdichip.com/drivers/vcp-drivers/`.
2. Niki Gagliardi, *ADS131M0x Library for Arduino*, available at: `https://github.com/nikiGagliardi/ADS131M0x/tree/master`.