ADC121S051 INTERFACE

Information

The ADC121S051 is a low-power, single-channel CMOS 12-bit analog-to-digital converter with a high-speed serial interface.

Sample rate range of 200 ksps to 500 ksps

SPI™/QSPI™/MICROWIRE/DSP Compatible

Supply Voltage Range (VA) - −0.3V to 6.5V

Input Signal Voltage Range - −0.3V to (VA +0.3V)

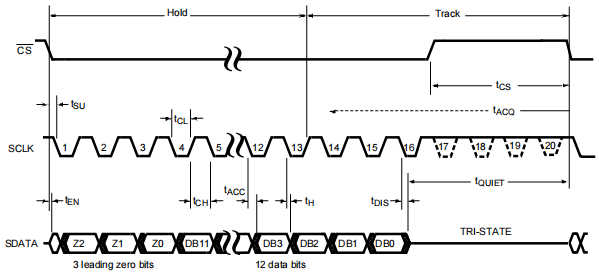
fSCLK (Clock Frequency) - Min - 4 MHz to 10 MHz

$$CalculatedVoltage = (ResolutionValue * MultiplyFactor) * 2$$

Multiply Factor = Supply Voltage / 4096 == 1 LSB

and transition from 0000 to 0001 is 1/2 of LSB. so Multiply by 2.

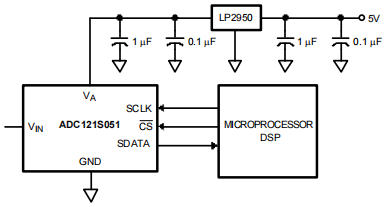
Read 16bit From Adc in that from LSB (DB0 to DB11) are Data bits (12bit). Remaining 3 bits up to MSB i.e. DB15 are 0.



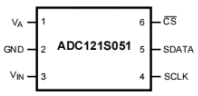
Important

For Persice output, use accurate 5v so the output is Presice.

Power is provided in this example by the National Semiconductor LP2950 low-dropout voltage regulator, available in various fixed and adjustable output voltages. The power supply pin is bypassed with a capacitor network located close to the ADC. Because the reference for the ADC is the supply voltage, any noise on the supply will degrade the device noise performance. To keep noise off the supply, use a dedicated linear regulator for this device, or provide sufficient decoupling from other circuitry to keep noise off the ADC supply pin. Because of the ADC's low power requirements, it is also possible to use a precision reference as a power supply to maximize performance.



Pinout Diagram



6-Lead SOT-23 or WSON

Pin Description

Pin No.	Symbol	Description
ANALOG I/O		
3	V_{IN}	Analog input. This signal can range from 0V to V_A
DIGITAL I/O		
4	SCLK	Digital clock input. This clock directly controls the conversion and readout processes.
5	SDATA	Digital data output. This clock directly controls the conversion and readout processes.
6	$\overline{CS}$	Chip select. On the falling edge of CS, a conversion process begins.
POWER SUPPLY		
1	V_A	Positive supply pin. This pin should be connected to a quiet +2.7V to +5.25V source and bypassed to GND with a 1 µF capacitor and a 0.1 µF monolithic capacitor located within 1 cm of the power pin.
2	GND	The ground return for the supply and signals.
PAD	GND	For package suffix CISD(X) only, it is recommended that the center pad should be connected to ground.

ADC121S051 with Beaglebone Black in C

You need to enable the pins with uBoot overlays:

You need to edit /boot/uEnv.txt:

Change these lines: (Change as per SPI that you are used)
#uboot_overlay_addr5=<file5>.dtbo
to
uboot_overlay_addr5=/lib/firmware/BB-SPIDEV1-00A0.dtbo

hen reboot.

In Below Example we use SPI1 (Tested C Code in BBB) debian OS.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <sys/mman.h>

const char *spi_device = "/dev/spidev1.0";
uint8_t spi_mode = SPI_MODE_0;
uint8_t bits_per_word = 16;
uint32_t spi_speed_hz = 10000000; // 10 MHz

int spifd; //spi file
float Voltage1;
int value;

// Define base addresses for GPIO control registers
#define GPIO0_BASE  0x44E07000
#define GPIO1_BASE  0x4804C000
#define GPIO2_BASE  0x481AC000
#define GPIO3_BASE  0x481AE000

// GPIO register offsets
#define GPIO_OE           0x134   // Offset for GPIO output enable register
#define GPIO_SETDATAOUT   0x194   // Offset for setting GPIO data output register
#define GPIO_CLEARDATAOUT 0x190   // Offset for clearing GPIO data output register

int out = 1;  // 0 for input, 1 for output
int in = 0;   // 0 for input, 1 for output

#define SPICS 60  // P9_12 Change this to the GPIO pin you want to control

// Functions

int spi_init(const char *spi_device, uint8_t spi_mode, uint8_t bits_per_word, uint32_t spi_speed_hz);
int read_16bit_value(int spifd);
void cleanup();
void signal_handler(int signo);
void set_gpio_res_low(volatile unsigned int *gpio_base, int gpio_offset);
void set_gpio_res_high(volatile unsigned int *gpio_base, int gpio_offset);
void set_gpio_res_direction(volatile unsigned int *gpio_base, int gpio_offset, int direction);
```

```
volatile unsigned int *mmap_gpio(int gpio_base);
volatile unsigned int *map_gpio_by_pin(int pin);

int main() {
    // Register the signal handler
    if (signal(SIGINT, signal_handler) == SIG_ERR) {
        perror("Error registering signal handler");
        exit(EXIT_FAILURE);
    }

    // Register the cleanup function
    if (atexit(cleanup) != 0) {
        perror("Error registering cleanup function");
        exit(EXIT_FAILURE);
    }

    // Determine GPIO base address based on pin number
    volatile unsigned int *gpio_base;

    // Map GPIO memory based on pin numbers
    gpio_base = map_gpio_by_pin(SPICS); // Assuming GPIO1 for your example

    // Check if map_gpio_by_pin returned a valid pointer
    if (gpio_base == NULL) {
        printf("Failed to map GPIO memory.\n");
        return -1;
    }

    // Set pin direction
    set_gpio_res_direction(gpio_base, SPICS % 32, out);

    // Initialize SPI
    int spifd = spi_init(spi_device, spi_mode, bits_per_word, spi_speed_hz);
    if (spifd < 0) {
        printf("Error initializing SPI\n");
        return 1;
    }

    // Main loop to read value every 1 second
    while (1) {
        // Set GPIO low
        set_gpio_res_low(gpio_base, SPICS % 32);
        value = read_16bit_value(spifd);
        set_gpio_res_high(gpio_base, SPICS % 32);

        if (value < 0) {
            printf("Error reading value from Voltage1\n");
            close(spifd);
            return 1;
        }

        Voltage1 = (value * 0.00120) * 2;


        if (Voltage1 < 0)
        {
          Voltage1 = 0.00;
        }

        printf("Calculated Voltage ADC1 (U4): %f \n", Voltage1);

/*
        // Print value in binary, hexadecimal, and decimal formats
        printf("SPI Read Value: Binary: ");
        for (int i = 15; i >= 0; i--) {
            printf("%d", (value >> i) & 1);
        }
        printf(" | Hexadecimal: 0x%04X | Decimal: %d\n", value, value);
*/
        // Sleep for 1 second
        usleep(100000);
    }
```

```
    // Unmap GPIO memory before exiting
    munmap((void *)gpio_base, 0x1000);
    return 0;
}

// Function to set GPIO low and unexport the pin
void cleanup() {
    // Close SPI device
    close(spifd);
}

// Function to initialize SPI
int spi_init(const char *spi_device, uint8_t spi_mode, uint8_t bits_per_word, uint32_t spi_speed_hz) {
    // Open SPI device
    spifd = open(spi_device, O_RDWR);
    if (spifd < 0) {
        perror("Error opening SPI device");
        return -1;
    }

    // Set SPI mode
    if (ioctl(spifd, SPI_IOC_WR_MODE, &spi_mode) < 0) {
        perror("Error setting SPI mode");
        close(spifd);
        return -1;
    }

    // Set SPI bits per word
    if (ioctl(spifd, SPI_IOC_WR_BITS_PER_WORD, &bits_per_word) < 0) {
        perror("Error setting SPI bits per word");
        close(spifd);
        return -1;
    }

    // Set SPI clock speed
    if (ioctl(spifd, SPI_IOC_WR_MAX_SPEED_HZ, &spi_speed_hz) < 0) {
        perror("Error setting SPI clock speed");
        close(spifd);
        return -1;
    }

    return spifd; // Return file descriptor for SPI device
}

// Function to read 16-bit value from SPI
int read_16bit_value(int spifd) {
    uint16_t data;

    // Perform SPI transfer
    struct spi_ioc_transfer transfer = {
        .tx_buf = 0, // no data to send
        .rx_buf = (unsigned long)&data,
        .len = sizeof(data),
        .delay_usecs = 0,
        .cs_change = 0,
    };

    if (ioctl(spifd, SPI_IOC_MESSAGE(1), &transfer) < 0) {
        perror("Error performing SPI transfer");
        return -1;
    }

    return data; // Return received data
}

// Signal handler for SIGINT (Ctrl+C)
void signal_handler(int signo) {
    if (signo == SIGINT) {
        printf("\nReceived SIGINT. Cleaning up...\n");
        cleanup();
        exit(EXIT_SUCCESS);
```

```
    }
}

// Function to mmap a specific GPIO bank
volatile unsigned int *mmap_gpio(int gpio_base) {
    int fd;
    void *gpio_addr;
    volatile unsigned int *gpio;

    fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd == -1) {
        perror("Error opening /dev/mem");
        return NULL;
    }

    gpio_addr = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, fd, gpio_base);
    close(fd);

    if (gpio_addr == MAP_FAILED) {
        perror("Error mapping GPIO memory");
        return NULL;
    }

    gpio = (volatile unsigned int *)gpio_addr;
    return gpio;
}

// Function to set GPIO pin direction (input or output)
void set_gpio_res_direction(volatile unsigned int *gpio_base, int gpio_offset, int direction) {
    if (direction == 0) {
        gpio_base[GPIO_OE / 4] |= (1 << gpio_offset);   // Set pin as input
    } else {
        gpio_base[GPIO_OE / 4] &= ~(1 << gpio_offset);  // Set pin as output
    }
}

// Function to set GPIO pin high
void set_gpio_res_high(volatile unsigned int *gpio_base, int gpio_offset) {
    gpio_base[GPIO_SETDATAOUT / 4] = 1 << gpio_offset;
}

// Function to set GPIO pin low
void set_gpio_res_low(volatile unsigned int *gpio_base, int gpio_offset) {
    gpio_base[GPIO_CLEARDATAOUT / 4] = 1 << gpio_offset;
}

// Function to map GPIO based on pin number
volatile unsigned int *map_gpio_by_pin(int pin) {
    volatile unsigned int *gpio_base;

    switch (pin / 32) {
        case 0:
            gpio_base = mmap_gpio(GPIO0_BASE);
            break;
        case 1:
            gpio_base = mmap_gpio(GPIO1_BASE);
            break;
        case 2:
            gpio_base = mmap_gpio(GPIO2_BASE);
            break;
        case 3:
            gpio_base = mmap_gpio(GPIO3_BASE);
            break;
        default:
            printf("Invalid GPIO bank.\n");
            return NULL;
    }

    return gpio_base;
}
```

Documentation

adc121s051.pdf

Links

https://github.com/DIGHEATUL/ADC121S51_INTERFACE.git