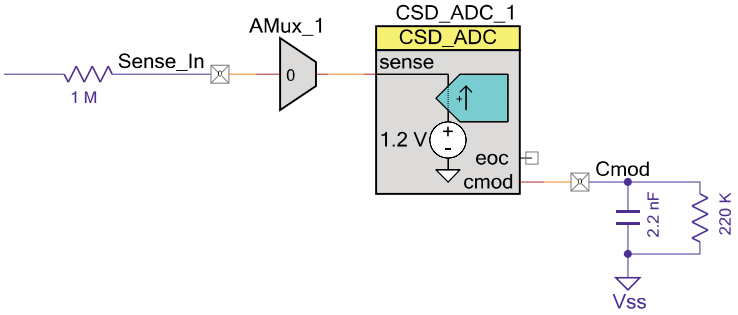
PSoC 4 Voltage ADC using CSD Hardware (CSD ADC)

1.0



Features

- 0 to 5 volt input range [1]
- Results provided in mV
- End of Conversion (EOC) terminal provided for an optional interrupt

General Description

The CSD Hardware ADC component repurposes the CapSense CSD hardware to perform voltage measurements. At its core, the CSD hardware operates as a current sensing circuit. A 1 MΩ resistor placed in series with the input converts an input voltage signal into a small current which is measured by the CSD hardware. The input voltage is calculated from this measured current and reported through an Application Programming Interface (API).

After each device reset, a simple open circuit calibration must be performed before any measurements are taken. This calibration requires starting the ADC, disconnecting all inputs using the analog mux, and making a single API call. The open circuit calibration can be called again at any time to ensure that the result is as accurate as possible. The ADC supports multiplexing input channels and two schematic macros are provided in the library to demonstrate this ability.

This component is only available when not using the the CapSense CSD block (capsense cannot be placed in the design).

Note The CSD_ADC requires that the 7-bit IDAC component be placed and enabled in the design to prevent signals on AmuxBusB from corrupting the ADC result. The output of the 7-bit IDAC is always connected to AmuxBusB in the silicon, and the output value can be anything from zero (essentially off) to full scale. It is only important that the IDAC is enabled during any ADC conversion. The CalibrateNoInput(), StartConvert(), and Read_mVolts() APIs return an error code to indicate when the IDAC is not enabled. If the IDAC is enabled, the conversion will start successfully and return a success code. If the IDAC is not enabled, the conversion will not start and the API will return an error code. You may use the IDAC for any purpose as long as it is

---

1 If the measured input is connected directly to the input resistor, the input range is not dependent on Vdd (or Vddio if available). However, one of the methods for multiplexing inputs to the ADC requires the input voltages connect directly to a pin. In this configuration, the input voltage must not exceed Vddd (or Vddio if available). See the Schematic Macro section for more information.

enabled during ADC conversions. An IDAC output setting of zero turns the output off while still keeping the IDAC enabled.

When to Use an ADC

The CSD_ADC can be used in any application to monitor external voltages from batteries, sensors, and transducers. This component is best suited for low-frequency (< 50 Hz) applications due to its low sampling rate.

Note: The CSD_ADC component uses resources from the PSoC 4 CapSense CSD block. Therefore, the CSD_ADC cannot be used in applications that also require capacitive sensing.

Input/Output Connections

This section describes the various input and output connections for the CSD_ADC.

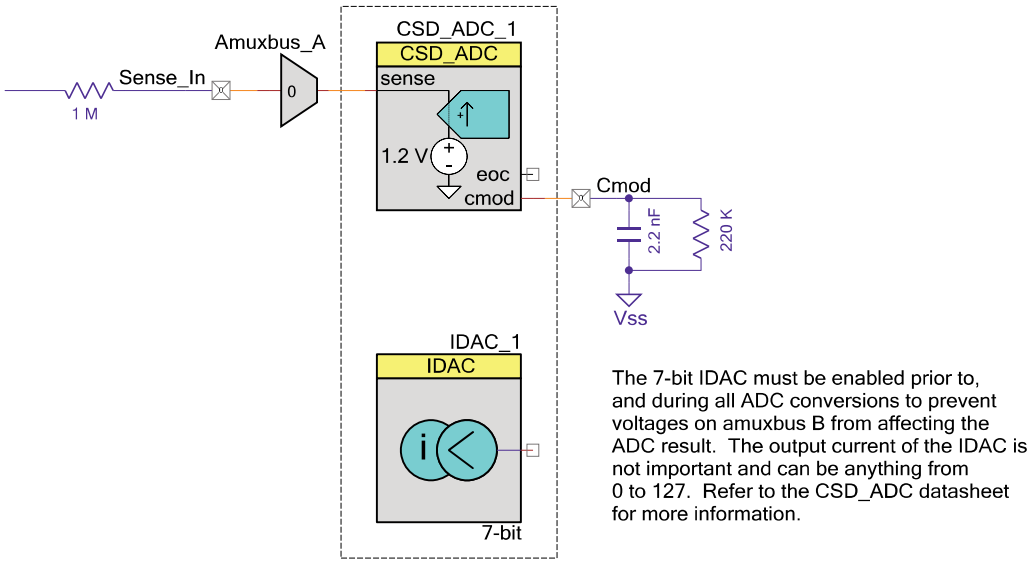
Terminal Name	I/O Type	Description
sense	Analog Input	Sense is the current sensing analog input of the ADC. The sense input measures a very small current consumed by or sourced from the node. The voltage at this terminal will be nominally 1.2 volts when the ADC is enabled. For the ADC to function properly, an external 1 MΩ resistor must be placed in series with this terminal. This resistor converts the applied voltage into a current, which is measured by the CSD_ADC. The API then determines the input voltage using the measured current, known input resistance (1 MΩ), and known sense node voltage (1.2 volts). For better accuracy, use resistors with a lower tolerance (1% or better).
eoc	Digital Output	End of Conversion interrupt terminal. This terminal can only be connected to an interrupt. The EOC interrupt can be used to execute code when an ADC conversion has completed.
cmod	Analog	The cmod pin must be connected to an external grounded 2.2 nF capacitor with a 220 KΩ resistor in parallel to Cmod. This node is part of the ADC's modulator circuit and should not be connected to anything else. This terminal will be nominally 1.2 volts when the ADC is enabled. The resistor must be 220 KΩ. For better accuracy, use resistors with a lower tolerance (1% or better).

Schematic Macro Information

This section contains pertinent information for the CSD_ADC schematic macros.

CSD_ADC_SingleInput

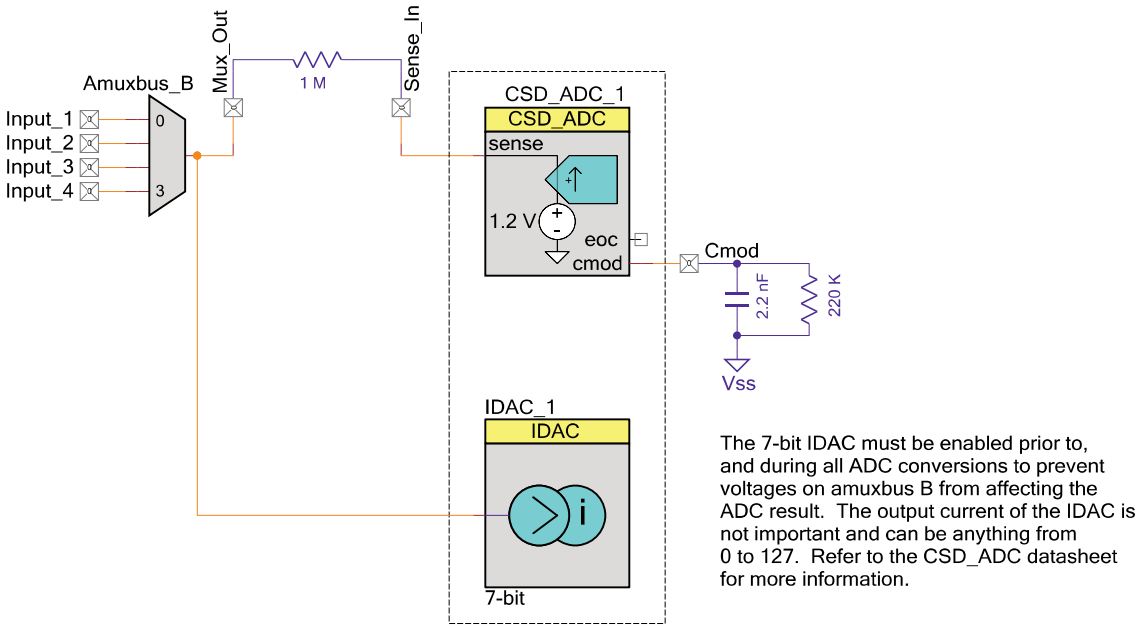
This schematic macro shows the internal and external configuration required for a single ADC channel. It includes the required external passive components, as well as the analog mux (Amuxbus_B) required to disconnect the input during the CSD_ADC_CalibrateNoInput() API call.



CSD_ADC_MultipleInput_1

This schematic macro shows one of two internal and external configurations required for multiple ADC input channels. It includes the required external passive components, as well as the analog mux (Amuxbus_B) used for multiplexing the input and for disconnecting all inputs during the CSD_ADC_CalibrateNoInput() API call.

This option requires fewer external passive components but consumes both analog mux busses (amuxbus A and amuxbus B), the 7 bit IDAC (the 7-bit IDAC is always connected to amuxbus B in the silicon), and prevents inputs above Vdd on the input channels.

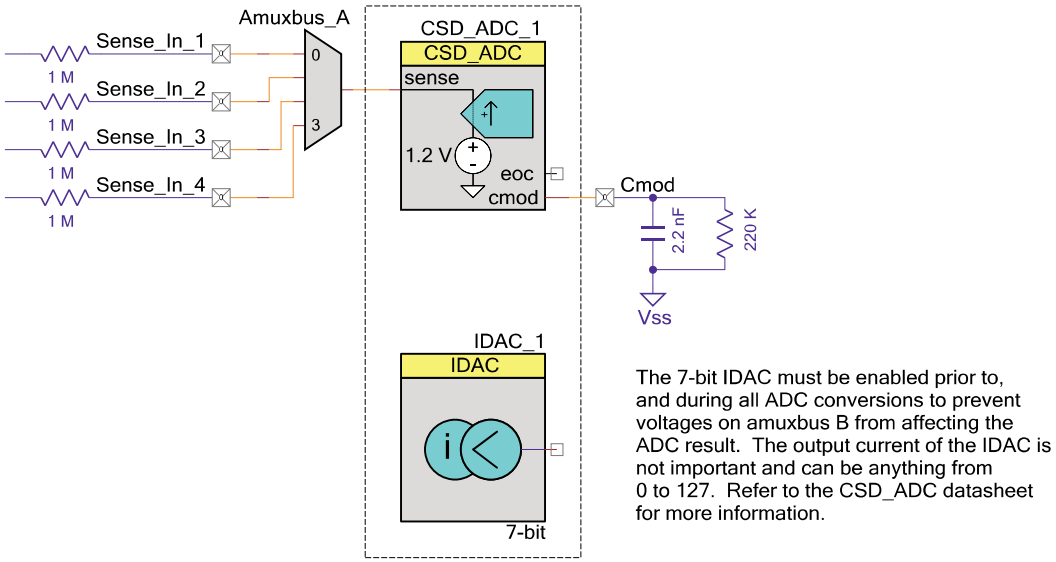


- Pros:
  - Requires fewer external passive components for multiple channels
- Cons:
  - Requires more pins than option # 2
  - Consumes both analog muxes in the device
  - Cannot be used with the 7-bit IDAC due to sharing amuxbus B with the IDAC output
  - Input range is limited to Vdd (or Vddio if available)

CSD_ADC_MultipleInput_2

This schematic macro shows the second of two internal and external configuration required for multiple ADC channels. It includes the required external passive components, as well as the Amux used for multiple channels and for disconnecting all inputs during the CSD_ADC_CalibrateNoInput() API call.

This option enables the use of another AMux or the 7-bit IDAC for other purposes, and allows inputs above Vdd.



- Pros:
  - One of the analog muxes is free for other uses (amuxbus B)
  - Can be used with the 7-bit IDAC
  - Input range on each channel can be higher than Vddd since the inputs are connected to resistors and not directly to the device pins
- Cons:
  - Requires more passive components than option #1

Component Parameters

There are no customizable parameters for the CSD_ADC.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "CSD_ADC_1" to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is "CSD_ADC."

Functions

Function	Description
CSD_ADC_Start()	Initializes and then enables the component.
CSD_ADC_Stop()	Disables power to the component.
CSD_ADC_Init()	Performs the required initialization of the component. Should be called before enabling the component.
CSD_ADC_Enable()	Enables power to the component.
CSD_ADC_CalibrateNoInput()	After starting the CSD_ADC (or initializing / enabling), the user is required to disconnect all inputs using the input mux and call this API to calibrate the CSD_ADC. Calling the CSD_ADC_CalibrateNoInput() API is only required once after a reset, but can be called again before any conversion to improve the accuracy of the ADC.
CSD_ADC_Read_mVolts()	Starts a conversion, waits for the conversion to complete, then returns the result in millivolts. Due to the unique construction of this ADC, there is no API to return counts since the returned value would not be meaningful.
CSD_ADC_StartConvert()	Begins a single ADC conversion. After calling this API, use the CSD_ADC_IsEndConversion() API to determine if the sample is complete.
CSD_ADC_IsEndConversion()	Either waits for the current conversion to complete or returns a status of the current conversion. Call this API after initiating a single conversion by calling the CSD_ADC_StartConvert() API.
CSD_ADC_GetResult_mVolts()	After a conversion has completed, calling this API returns the result of the conversion in millivolts.
CSD_ADC_ClearInterrupt()	Use this API in an ISR connected to the EOC terminal of the ADC to clear the interrupt request flag.
CSD_ADC_SetInterrupt()	Triggers the ISR connected to the EOC terminal of the ADC from firmware.
CSD_ADC_Sleep()	Prepares the component for deep sleep.
CSD_ADC_Wakeup()	Restores the component from deep sleep.

void CSD_ADC_Start (void)

Description: Initializes and enables the CSD_ADC component. Calls the CSD_ADC_Init() and CSD_ADC_Enable() API.

Note If the CSD_ADC is being started for the first time since a device reset, ensure that you disconnect all inputs to the CSD_ADC and call the CSD_ADC_CalibrateNoInput() API. The CSD ADC only requires calibration after reset.

After calling the CSD_ADC_Start() API and calibrating, the ADC is ready to take samples by calling the CSD_ADC_StartConvert() API.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_Stop(void)

Description: Disables the CSD_ADC component. All configuration and calibration information is retained when the component is stopped.

If CSD_ADC_Stop() is called during a conversion, the conversion that is currently in progress is aborted and any result returned by the CSD_ADC_GetResult_mVolts() API should be discarded. ADC_CSD_Stop() or ADC_CSD_Sleep() should be called before putting the device to sleep.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_Init (void)

Description: Initializes the CSD hardware by setting all the appropriate registers for the component. Initialization is only required once after reset.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_Enable(void)

Description: Enables the CSD_ADC component. This API should only be called after the component has been initialized. Enabling the component before initialization may result in unexpected behavior.

Parameters: None

Return Value: None

Side Effects: None

int32 CSD_ADC_CalibrateNoInput(void)

Description: This API must be called once after the component has been started before any measurements are taken. After calling CSD_ADC_Start(), it is recommended to delay at least 5 ms before calling the calibrate function to ensure the cmod has had sufficient time to charge up and stabilize. Calibration is only required after a device reset. It is not required to calibrate again after waking up from sleep or deepsleep but is recommended.

This API performs a calibration to determine the value of the internal modulator current source. The calibration algorithm uses the known voltage across the 220 K resistor connected to cmod to measure a known current and calibrate the internal current source.

Note All inputs must be disconnected from the sense terminal of the ADC to prevent external signals from invalidating the calibration. All of the CSD_ADC macros include an analog mux in front of the sense input to facilitate disconnecting all inputs from the CSD_ADC for calibration. It is the user's responsibility to disconnect the inputs using this analog mux.

Note The 7-bit IDAC must be started to perform the calibration. See the CSD_ADC_StartConvert() API description for more information.

The API takes a sample with the knowledge that nothing is connected to the input and calculates the value of the internal current source. This calculation is critical for accurate operation of the CSD_ADC. For the best accuracy, call this API before any conversion. Recalibrating the ADC will remove the effect of temperature drift on the IDAC improving the accuracy of the result.

Parameters: None

Return Value: Reports if the conversion was started successfully. See CSD_ADC_StartConvert() for more information.

Constant	Description
CSD_ADC_START_CONV_OK	The 7-bit IDAC is enabled and the conversion was started
CSD_ADC_START_CONV_ERR	The 7-bit IDAC is not enabled and the conversion was not started

Side Effects: None

Side Effects: None

int32 CSD_ADC_Read_mVolts(void)

Description: Starts a conversion, waits for the conversion to complete, then returns the result in mV. This is identical to calling CSD_ADC_StartConvert(), CSD_ADC_IsEndConversion(CSD_ADC_WAIT_FOR_RESULT), and CSD_ADC_GetResult_mVolts() in order.

Due to the unique construction of this ADC, there is no raw count result. The CSD_ADC does not perform like a normal ADC and does not provide raw counts in line with a standard ADC. For example, a zero volt input would return a count of 15,000 and a full scale input would provide a result of 8,000. To extract meaningful information from these values, the result must be converted into a voltage and thus no API exists to return a raw count.

Parameters: None

Return Value: The measured voltage in millivolts as a signed 32 bit integer. Example: an input of 1.205 volts would be returned as 1205.

The API can also return CSD_ADC_START_CONV_ERR to indicate that the 7-bit IDAC is not enabled and the conversion was not started.

Constant	Description
CSD_ADC_START_CONV_ERR	The 7-bit IDAC is not enabled and the conversion was not started

Side Effects: None

int32 CSD_ADC_StartConvert(void)

Description: Initiates a single ADC conversion. This function will return as soon as the conversion has started to enable the user to perform other tasks while the conversion is progressing. To check on the status of the conversion, call the CSD_ADC_IsEndConversion() API.

When the conversion completes, the user must call the CSD_ADC_StartConvert() API again to initiate another conversion.

This API also checks the enabled state of the 7-bit IDAC. If the 7-bit IDAC is not enabled, the API will not start a conversion and will return an error code CSD_ADC_START_CONV_ERR. To resolve this error, make sure you have a 7-bit IDAC placed in your design (called IDAC_7bit in this example) and call the IDAC_7bit_Start() API. Refer to the PSoC 4 IDAC component datasheet for more information.

The 7-bit IDAC must be enabled during any ADC conversion to prevent signals that may be present on amuxbusb from affecting the measured voltage of the ADC.

If the IDAC is enabled, the conversion will start and the API will return a CSD_ADC_START_CONV_OK code.

Parameters: None

Return Value: Reports if the conversion was able to start successfully.

Constant	Description
CSD_ADC_START_CONV_OK	The 7-bit IDAC is enabled and the conversion was started
CSD_ADC_START_CONV_ERR	The 7-bit IDAC is not enabled and the conversion was not started

Side Effects: None

int32 CSD_ADC_IsEndConversion(int32 retMode)

Description: After starting a conversion by calling the CSD_ADC_StartConvert() API, this function will either:

- Wait for the conversion to finish
- Return the status of the current conversion

Parameters: The parameter passed to this API determines if the function will block and not return until the conversion has completed or if it will return the status of the current conversion.

Constant	Description
CSD_ADC_RETURN_STATUS	Return the status of the current conversion.
CSD_ADC_WAIT_FOR_RESULT	Do not return until the current conversion has completed.

Return Value: The status of the current conversion.

Constant	Description
CSD_ADC_CONV_IN_PROGRESS	The conversion is in progress
CSD_ADC_CONV_COMPLETE	The conversion has completed

Side Effects: None

int32 CSD_ADC_GetResult_mVolts(void)

Description: Call this API after CSD_ADC_IsEndConversion() indicates that the conversion has completed. Returns the result of the most recently completed conversion in millivolts. Calling this API during a conversion will yield the result of a partial conversion which will be meaningless. If called multiple times, this API will yield the same result until another conversion is started.

Due to the unique construction of this ADC, there is no raw count result. The CSD_ADC does not perform like a normal ADC and does not provide raw counts in line with a standard ADC.

For example, a zero volt input would return a count of 15,000 and a full scale input would provide a result of 8,000. To extract meaningful information from these values, the result must be converted into a voltage and thus no API exists to return a raw count.

Parameters: None

Return Value: The measured voltage in millivolts as a signed 32 bit integer.

Example: an input of 1.205 volts would be returned as 1205.

Side Effects: None

void CSD_ADC_ClearInterrupt(void)

Description: Call this API to clear the pending interrupt of an ISR connected to the EOC terminal of the ADC. If you do not clear the pending interrupt, the ISR will execute repeatedly. It is recommended to call this API in the ISR.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_SetInterrupt(void)

Description: Call this API to trigger an ISR connected to the EOC terminal of the ADC via firmware.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_Sleep(void)

Description: Prepare the CSD_ADC for sleep. Call the CSD_ADC_Sleep() API before calling the CySysPmDeepSleep() API. The component's enabled state is checked, stored and then the component is stopped. When the CSD_ADC_Wakeup() API is used, the component is enabled based on the previously stored enabled state.

For example, if the component was enabled before going calling the CSD_ADC_Sleep() API, the component will be enabled when the CSD_ADC_Wakeup() API is called. If the component was not enabled before calling the CSD_ADC_Sleep() API, the component will not be enabled when the CSD_ADC_Wakeup() is called.

Parameters: None

Return Value: None

Side Effects: None

void CSD_ADC_Wakeup(void)

Description: Restores the CSD_ADC from sleep. If the component was enabled before the CSD_ADC_Sleep() API was called, the CSD_ADC_Wakeup() function will also re-enable the component.

It is not necessary to recalibrate the ADC after waking up from sleep.

Parameters: None

Return Value: None

Side Effects: None

Sample Firmware Source Code

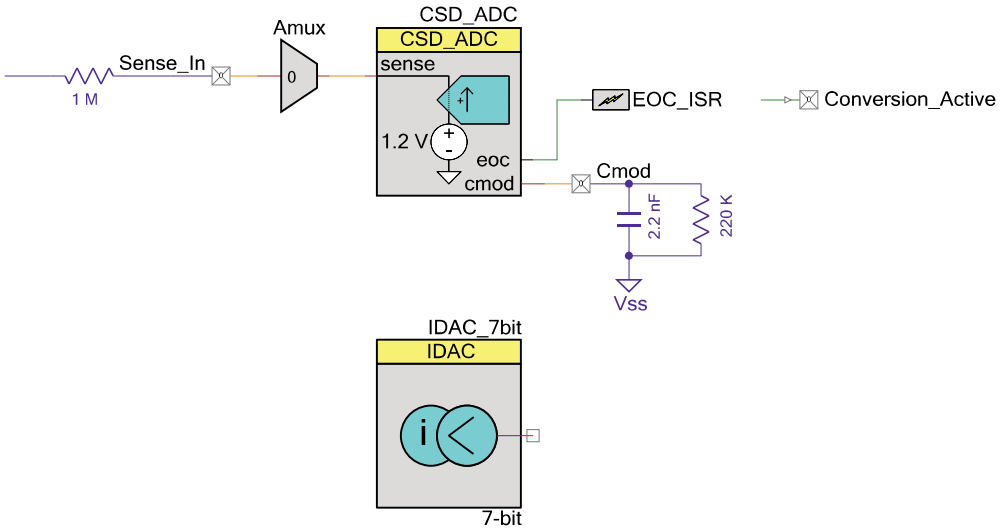
PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the "Find Example Project" topic in the PSoC Creator Help for more information.

The following example shows a basic setup for calibrating the ADC, taking a measurement and using the interrupt service routine. The project first starts the ADC, disconnects all inputs using the AMux, and performs the no input calibration step. The input is then connected to the pin and a conversion is started.

The code toggles a pin called “Conversion_Active” high to indicate that the conversion is started. An interrupt service routine (ISR) connected to the EOC terminal of the ADC toggles the “Conversion_Active” pin low when the conversion has completed.

Example Schematic



Example Code

```
#include <project.h>

/* Declare a prototype handler for our Interrupt Service Routine (ISR) */
CY_ISR_PROTO(MyISR);

int main()
{
    /* Declare a variable to hold the ADC result */
    int32 Voltage_mV;

    /* Enable Global Interrupts */
    CyGlobalIntEnable;

    /* Enable the 7-bit IDAC */
    IDAC_7bit_Start();

    /* Start our custom ISR handler */
    EOC_ISR_StartEx(MyISR);

    /* Start the ADC. This initializes and enables the ADC */
    CSD_ADC_Start();

    /* Ensure that nothing is connected to the input of the ADC */
    AMux_DisconnectAll();

    /* Allow time for the cmod charge up and stabilize */
    CyDelay(5);

    /* Perform the no input calibration */
    CSD_ADC_CalibrateNoInput();

    /* Connect and ADC channel to be measured */
    AMux_Connect(0);
```

```
    for(;;)
    {
        /* Initiate and ADC sample */
        CSD_ADC_StartConvert();

        /* Indicate that a sample is being taken by driving out
        Conversion_Active pin high */
        Conversion_Active_DR |= Conversion_Active_MASK;

        /* Wait for the conversion to complete */
        CSD_ADC_IsEndConversion(CSD_ADC_WAIT_FOR_RESULT);

        /* Read the result of the conversion from the ADC */
        Voltage_mV = CSD_ADC_GetResult_mVolts();
    }
}

/* Custom interrupt handler. This interrupt executes when the ADC
conversion has completed. This ISR will drive the Conversion_Active
pin low to indicate the conversion has completed and clear the interrupt
flag */
CY_ISR(MyISR)
{
    /* Drive the Conversion_Active pin low to indicate the conversion
    has completed */
    Conversion_Active_DR &= ~Conversion_Active_MASK;

    /* Clear the interrupt flag to prevent the ISR from executing again
    until the next End Of Conversion (EOC) signal is received */
    CSD_ADC_ClearInterrupt();
}
```

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- Project deviations – deviations that are applicable for all PSoC Creator components
- Specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the System Reference Guide along with information on the MISRA compliance verification environment.

The CSD_ADC component does not have any specific deviations.

API Memory Usage

The component memory usage varies significantly depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with an associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

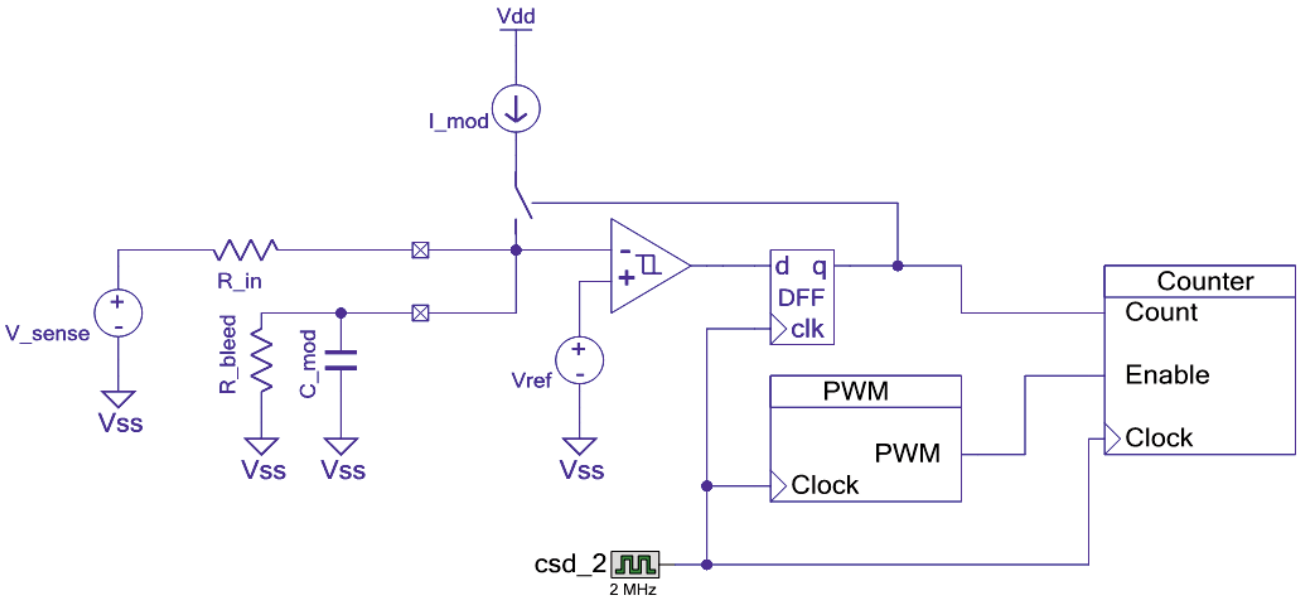
PSoC 4 (GCC)

Flash Bytes	SRAM Bytes
730	5

Functional Description

At its core, the CSD_ADC operates as a current sensing circuit. A simplified version of the modulator is shown in Figure 1. The voltage on C_mod is monitored by the comparator, and when the voltage dips below Vref (1.2 volts), the I_mod current source is enabled until the voltage on C_mod goes above 1.2 volts.

Figure 1. Simplified modulator



The ratio of on time to off time is measured by the counter over a fixed period of time. The ratio of on time to off time relative to I_mod represents the average current pulled out of C_mod by Rbleed and the input.

Important Values

- **R_in –** The input resistance. For the purposes of the CSD_ADC, this value is fixed at 1 MΩ.
- **R_bleed –** The bleed resistance connected to C_mod, which sets I_bleed, the bleed current. For the purposes of the CSD_ADC, this value is fixed at 220 KΩ.
- **Resolution –** The period of the PWM that controls the counting window. For the purposes of the CSD_ADC, this value is fixed at 17,000.
- **I_mod –** The modulated current. This current is from the 8-bit IDAC. It can either be 1.2 µA per step or 2.4 µA per step, up to a maximum of 255 steps. For the purposes of the CSD_ADC, this value is fixed at 1.2 uA per step and a value of 16 (19.2 µA)
- **Vref –** The reference voltage. This is the voltage that is maintained across the Cmod capacitor and by extension, the input of the ADC. Nominally 1.2 volts.
- **ADC Result –** This is the raw count value from the ADC counter.

Important Equations

With the following equations, all the elements required to calculate the sensed-current and voltage are present.

The sensed-current sign convention is: current sourced from the sense node is considered positive.

$$I_{sense} = I_{mod} * \frac{Result}{Resolution} - I_{bleed}$$

$$V_{sense} = V_{ref} - I_{sense} * R_{in}$$

$$I_{bleed} = \frac{V_{ref}}{R_{bleed}}$$

Clock Selection

The CSD_ADC component has two internal clocks running at 1 MHz.

Resources

The CSD_ADC uses the following device resources:

- CSD hardware block.
- 8-bit IDAC
- Analog mux bus A
- 2 digital clocks

DC and AC Electrical Characteristics

All characterization done with 0.1% tolerance 1 MΩ input resistor (R_in), 0.1% tolerance 220 KΩ bleed resistor (R_bleed) and 2.2 nF C_mod capacitor.

DC Specifications DC Electrical Performance Requirements:

Symbol	Description	Conditions	Min	Typ	Max	Units
ADC_{Res}	ADC resolution	-	1	-	-	mV
ADC_{MONO}	ADC Monotonicity	Across PVT	-	-	-	Yes
ADC_{Error}	ADC gain error	For 0 to 5-V range, 0.1% accurate Rin / Rbleed ,1% accurate Internal Vref and Temp range of 0 to 70 C	-	-	1	%
ADC_{Offset}	ADC offset error		-	-	50	mV
ADC_{INMAX}	ADC input voltage range	-	0	-	5 / Vddio*	V

*Inputs applied directly to a pin must not exceed Vddio, but voltages applied to an input resistor can exceed Vddio.

AC Specifications AC Electrical Performance Requirements:

Symbol	Description	Conditions	Min	Typ	Max	Units
ADC_{INL}	ADC integral non-linearity	0 to 5 v input and 0 to 70 C	-	-	18	mV
ADC_{DNL}	ADC differential non-linearity	0 to 5 v input and 0 to 70 C	-	-	12	mV
ADC_{Samp}	ADC Sample rate	-	-	-	58	sps

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0.a	Removed component from Component Catalog for devices that don't support it. Added Cypress address to macros.	No impact.
	Updated datasheet.	Fixed some typos. Added PSoC 4 to the title.
1.0	First version of this component.	

© Cypress Semiconductor Corporation, 2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control, or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.