Using the DE-Series ADC Controller

For Quartus® Prime 18.1

1 Introduction

This tutorial describes how to use the University Program IP core to operate the built-in Analog-to-Digital Converter (ADC) component on the Intel® DE-series boards. It demonstrates the basic signal and timing requirements of the ADC, and how to use the core in hardware- or software-based projects.
The tutorial is based on the assumption that the reader has basic knowledge of both the C and Verilog languages, and is familiar with the Quartus® Prime and Platform Designer softwares.

Contents:

- Background
- The DE-Series ADC Controller
- Implementing the ADC Controller with Platform Designer and Nios® II
- Using the ADC Controller with HAL
- Implementing the ADC Controller with IP Catalog

2 Background

Analog-to-Digital Converters are used to connect analog devices (such as a microphones) to a digital system. The ADC performs the function of converting a continuous-valued analog signal into a discrete-valued digital one.

The DE-series FPGA boards that contain analog-to-digital converters, are shown in Table 1. ADC devices can have different attributes such as the range of clock frequencies at which it can be driven, the range of voltages that it can sample, the number of channels, and resolution. The attributes for the ADC devices found on DE-series boards are summarized in Table 1.

Board	ADC Chip	ADC Clock Freq.	Voltage Range	Channels	Resolution
DE0-Nano	ADC128S022	0.8 - 3.2 MHz	0 - 3.3 V	8	12 bit
DE0-Nano-SoC	LTC2308	0.01 - 20 MHz	0 - 5 V	8	12 bit
DE1-SoC (rev. A-E)	AD7928	0.01 - 20 MHz	0 - 5 V	8	12 bit
DE1-SoC (rev. F+)	LTC2308	0.01 - 20 MHz	0 - 5 V	8	12 bit
DE10-Standard	LTC2308	0.01 - 20 MHz	0 - 5 V	8	12 bit
DE10-Nano	LTC2308	0.01 - 20 MHz	0 - 5 V	8	12 bit
DE10-Lite	MAX® 10 Internal ADC	10 MHz	0 - 5 V	6	12 bit

Table 1. DE-series boards with analog-to-digital converters

2.1 ADC Signals

The connections between the ADC connectors, the ADC device, and the FPGA vary depending on the DE-series board. The connections are shown in Figures 1, 2, and 3.

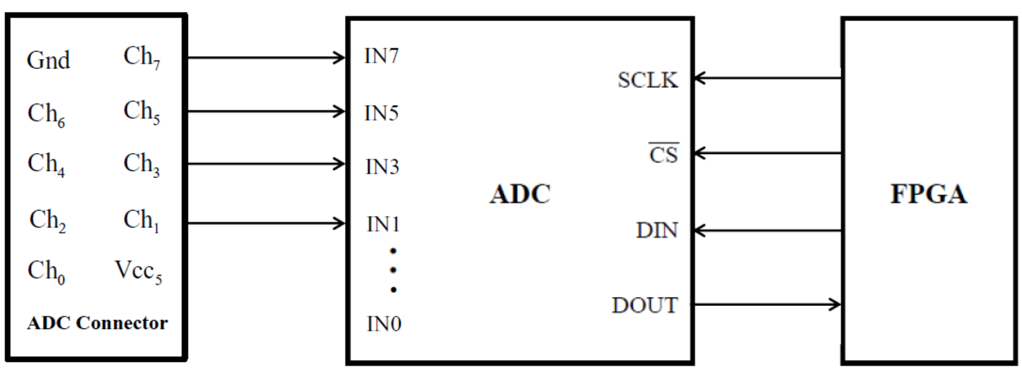


Figure 1. Signals to and from the ADC on the **DE10-Standard**, **DE10-Nano**, **DE1-SoC**, and **DE0-Nano-SoC**

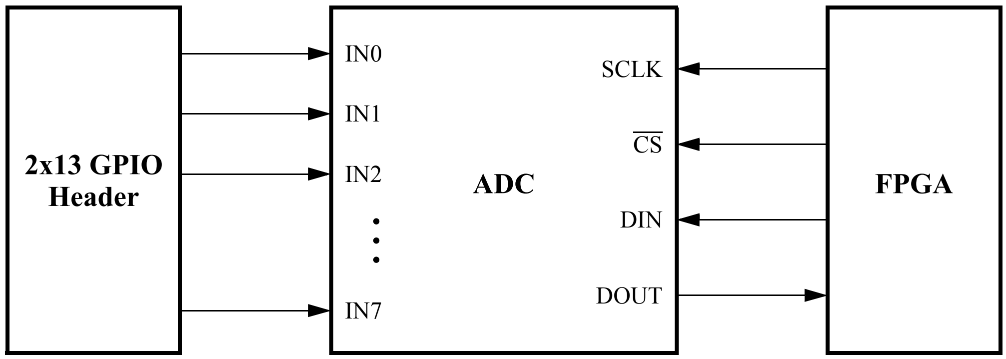


Figure 2. Signals to and from the ADC on the **DE0-Nano**

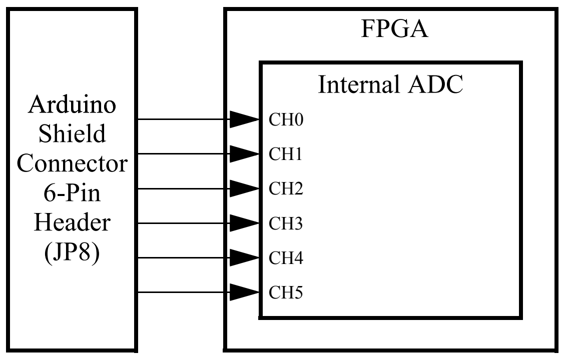


Figure 3. Signals to and from the ADC on the **DE10-Lite**

The ADC receives analog signals via its input pins (each corresponding to a channel). When performing a conversion, the ADC reads the signal on one of these input channels and converts it to a digital output. On the DE10-Standard, DE10-Nano, DE0-Nano-SoC and DE1-SoC boards, these eight pins are connected to the dedicated 10-pin ADC header. On the DE0-Nano board, these eight pins are connected to the 2x13 GPIO header on the underside of the board. On the DE10-Lite board, the six input pins are connected to the 6-Pin header of the Arduino* Shield Connectors.

For all boards except the DE10-Lite (whose ADC is built into the FPGA), the ADC also has four wires connected to the FPGA. The wires are used to control the ADC and to allow communication between it and the FPGA. The *SCLK* and $\overline{CS}$ signals are used to control the ADC, and are generated by circuitry in the FPGA. The *SCLK* signal serves as a device clock for the ADC, while the $\overline{CS}$ signal serves as an active-low chip select for the ADC chip. The *DIN* and *DOUT* wires are used for transferring addresses and data between the two chips. The FPGA uses the *DIN* connection, which is mapped to the *ADC_SADDR* pin on the FPGA, to provide the address of the next channel requested for conversion. The address is 3 bits in length, and is sent to the ADC serially at a rate of 1 bit per *SCLK* cycle. The *DOUT* connection is mapped to the *ADC_SDAT* pin on the FPGA, and is used by the ADC to send the digital value of the converted signal to the FPGA. This value is 12 bits long, and is sent to the FPGA in a serial manner at a rate of 1 bit per *SCLK* cycle.

2.2 Timing and Signal Requirements

Each boards ADC controller operates on a 16-cycle operational frame, as shown in Figure 4. The user is required to provide the *SCLK*, $\overline{CS}$, and *DIN* signals to the ADC, and to capture the *DOUT* signal as it is transmitted.

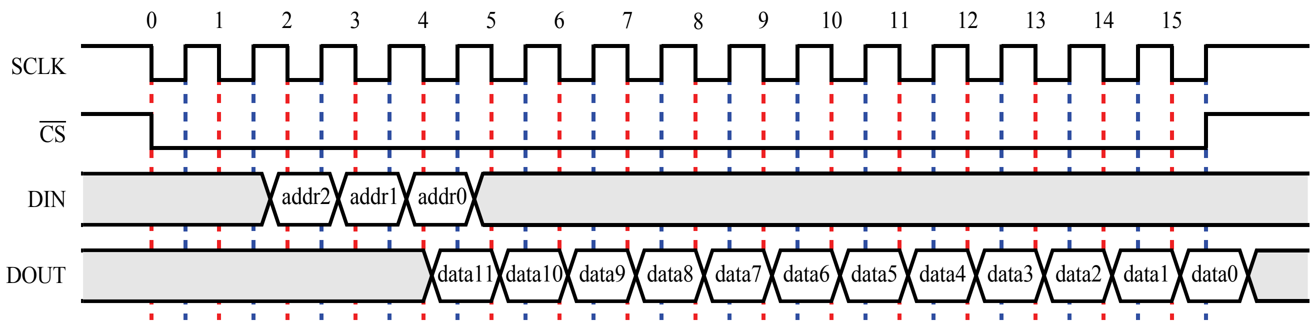


Figure 4. Timing requirements for the ADC

The *DOUT* signal provides the 12-bit converted value of the selected channel. On power-up, channel 0 is selected by default, while subsequent reads will use the address provided in the previous operational frame. The data bits are transmitted in descending order, such that the highest-order bit is delivered first. It is captured by the user on the rising edge of *SCLK*.

The *DIN* signal is used to select the channel to be converted in the following frame. It is delivered in descending order, and is captured by the ADC on the positive edges of *SCLK*. In order to avoid potential race conditions, the user should generate *DIN* on the negative edges of *SCLK*.

$\overline{CS}$ should be lowered on the first falling edge of *SCLK*, and raised on the last rising edge of an operational frame. See Table 1 for the timing requirements for each board.

2.3 Analog Circuit Requirements

All analog inputs are referenced against a Vdd signal hardwired to the ADC. Therefore, to avoid damaging the boards, any voltages provided to the ADC should not exceed this maximum voltage pins should not exceed the maximum voltages listed in Table 1. If the analog circuitry is powered by a supply voltage greater than the maximum voltage, voltage dividers should be used to limit the maximum output voltage to the maximum. Example analog circuits for measuring a variety of stimuli are shown in Figure 5. The resistance values given are approximate; all analog signals should be measured before being connected to the ADC.

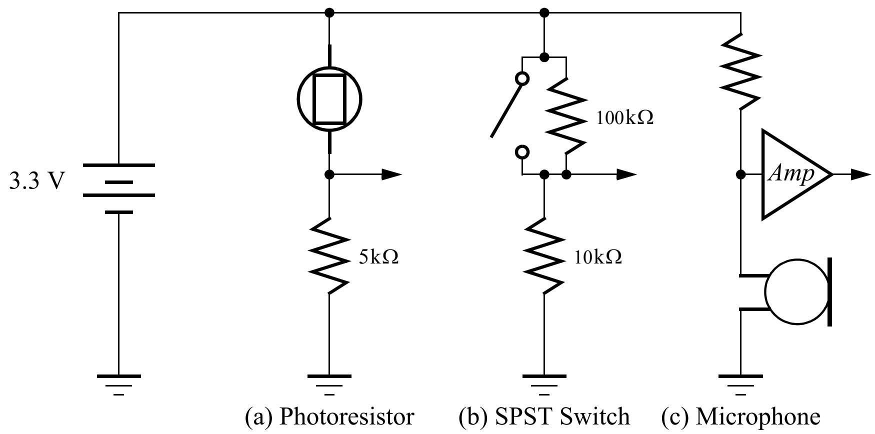


Figure 5. Examples of analog circuits using a variety of sensors.

Figure 5a includes a photoresistor. A photoresistor can be used to detect sources of light by changing resistance based upon the amount of light that strikes its surface. In this configuration, a high output voltage represents a bright signal, and low output represents a dark one. This can be reversed by switching the Vdd and GND connections.

Figure 5b shows the usage of a simple switch. The output voltage is low when the switch is open, and high when the switch is closed. As with the photoresistor, this can be changed by swapping the Vdd and GND connections.

Figure 5c utilizes a microphone. Since many basic microphones do not have a large enough signal amplitude to be detected by the ADC, the output may require amplification. The resistor should be matched to the impedance of the microphone.

When connecting analog circuits to the ADC, it it essential to connect the ground potential of the circuit to the GND pin. This creates a common reference point for both the circuit and the board, so that voltages can be compared accurately. You can find and use a GND pin on your board by consulting the board's User Manual. For example, you could use pin 10 of the 2x5 J15 ADC Controller header on the DE0-Nano-SoC and DE1- SoC boards, or pin 26 of the 2x13 GPIO header on the DE0-Nano board. Figures 6 and 7 illustrate how an analog circuit should be connected to the board.

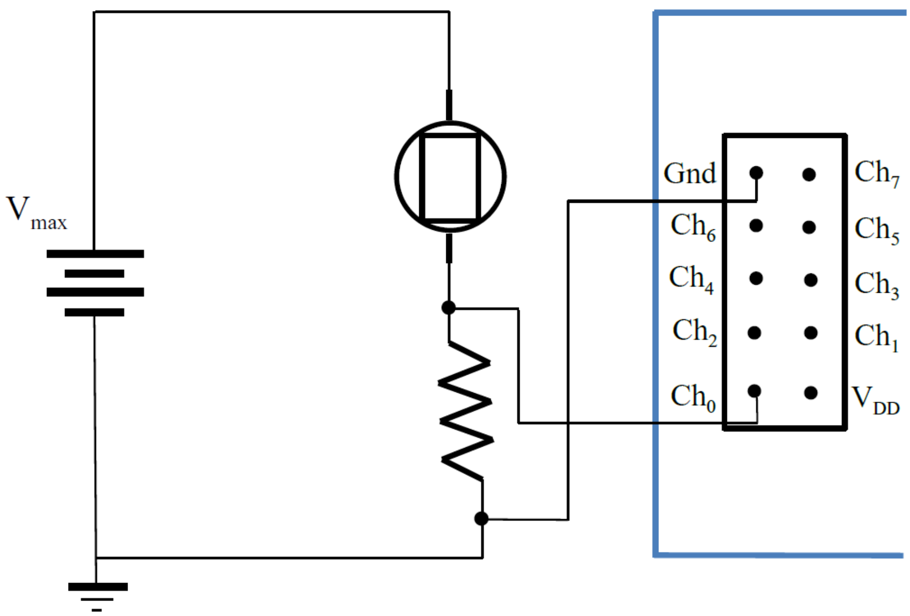


Figure 6. An analog circuit connected to the 2x5 ADC header on the **DE10-Standard, DE10-Nano, DE0-Nano-SoC or DE1-SoC**.

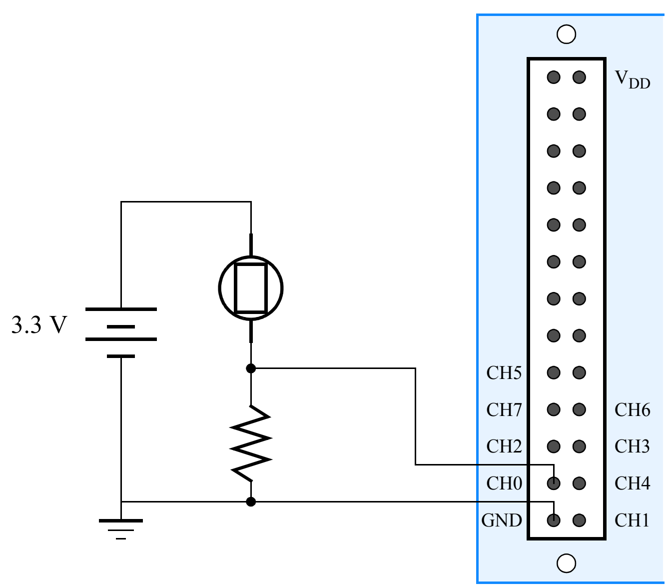


Figure 7. An analog circuit connected to the 2x13 GPIO header, shown from the underside of the **DE0-Nano board**.

The 2x13 GPIO should be on the right edge of the board.

3 The ADC Controller for DE-series Boards

The *ADC Controller for DE-series Boards* IP Core manages and controls the signals between the ADC and FPGA, and provides the user with the converted values. The core is usable in both hardware-only and software-controlled versions. It reads each of the input channels of the ADC in ascending order once per update cycle, storing the acquired values locally. Once the update cycle is complete, the new values are available for access. It also provides a number of customizations to the user to control its operation.

The ADC controller core defines the number of channels in use as a parameter, *NUM_CH*, which is set by the user when the core is instantiated. Since the core operates by sampling all used channels in series, reducing the number of used channels will reduce the total amount of time required to refresh the values.

The core also allows specification of the *SCLK* frequency. The user can enter a desired value in the allowed range (See Table 1). Exact matching of the desired *SCLK* value is not guaranteed, as *SCLK* is derived as an integer factor of the system clock. Typically, the mismatch will be less than a 5% difference between the desired and implemented value.

3.1 Implementing the ADC Controller with the Platform Designer Tool

3.1.1 The Software-Controlled ADC Core

For complex systems where a processor and software control is desired, the ADC controller can be included as a Platform Designer component compatible with a Nios® II or ARM* processor. For information on designing systems in Platform Designer that include Nios II and/or ARM, refer to the *Introduction to the Intel Platform Designer Integration Tool* and *Introduction to the Intel Nios II Soft Processor* or *Introduction to the ARM A9 Processor* tutorials.

The ADC Controller provides the processor with eight memory-mapped registers for reading and two registers for writing, as detailed in Table 1. The Controller is operated by reading from and writing to these registers.

Table 2. DE-Series ADC Controller register map

Offset in bytes	Register name	Read/Write	Purpose
0	`CH_0`	R	Converted value of channel 0
	`Update`	W	Update the converted values
4	`CH_1`	R	Converted value of channel 1
	`Auto-Update`	W	Enables or disables auto-updating
8	`CH_2`	R	Converted value of channel 2
12	`CH_3`	R	Converted value of channel 3
16	`CH_4`	R	Converted value of channel 4
20	`CH_5`	R	Converted value of channel 5
24	`CH_6`	R	Converted value of channel 6
28	`CH_7`	R	Converted value of channel 7

For reading, each of the eight registers corresponds to one of the eight input channels to the ADC. After the ADC converts the desired number of channels, the converted values will be available in these registers. If a channel is not in use, its corresponding register will contain zeroes. In each channel register, the low 12 bits (bits 11 to 0) are the value of the analog signal. Bit 15 in each register is a *refresh* bit, which is used in Auto-update mode. The bit is set to 1 when a corresponding channel is refreshed, and set to 0 when read. The remaining bits (31 to 16 and 14 to 12) are unused.

The *Update* register is used to initiate a conversion operation. Performing a write to this register will update all channels in use, with the new values becoming available once the entire conversion process has finished. If reads to the channel registers are attempted while a conversion is taking place, then the *wait_request* signal will be raised, causing the processor to stall until the update has finished.

The *Auto-Update* register is initially loaded with a zero value. The auto-update allows the system to automatically begin a second update cycle after the first finishes. As result, channel values can be accessed during an update cycle, and it is user's responsibility to ensure the values used are up-to-date. Writing a '1' to this register enables auto-update, while writing a '0' disables it.

3.1.2 Using the ADC Controller Core

To demonstrate the use of the ADC Controller, we will implement a system using the Platform Designer tool for the DE0-Nano-SoC board. The system will be controlled by a processor and software, and the converted values from the ADC will be displayed on the board's LEDs. Although we will use the DE0-Nano-SoC for demonstration purposes, similar steps can be used for other DE-series boards.

To make a new system with the ADC Controller, create a new project in Quartus Prime named *adc_demo*. The top-level module should also be *adc_demo*. Specify the device as the Cyclone® VE chip **5CSEMA4U23C6**, and complete the project creation. Then, open the Platform Designer tool.

The system will have four main components: the ADC Controller, a Nios II processor, on-chip memory, and LEDs to display the read values. The block diagram of the system is shown in Figure 8.

The system will use a 100 MHz system clock instead of CLOCK_50. First remove, the default clock source that is included in the new Platform Designer project. Next, from University Program, select the System and SDRAM Clocks for DE-series Boards from the Clock list. Change the Desired System clock to "100.0" MHz and the DE-Series Board to DE0-Nano-SoC, then select Finish. Export the clock input as clk and the reset input as reset. Use sys_clk as the system clock and reset_source as the system reset.

From the Processors and Peripherals, select the Nios II (Classic) Processor from the Embedded Processors list. Select Nios II/e as the processor type, and add it to the system. Rename this module to *cpu*, and connect it to the clock and reset signals.

Next add the on-chip memory by selecting On-Chip Memory (RAM or ROM) from the Basic Functions > On Chip Memory component list. Specify "8192" as the total memory size and select Finish to add it to the system. Connect the clock and reset signals to the memory, and connect the memory to the data_master and instruction_master sources of the Nios II processor. Rename the memory to *onchip_mem*. Edit the Nios II processor to specify *onchip_mem* as the reset and exception vector memories.

Use a PIO (Parallel I/O) component to connect the system to the LEDs. Select Processors and Peripherals > Peripherals and choose the PIO (Parallel I/O) component. Specify a width of 8, and select Output as the direction. Rename the component to *LEDs*. Connect it to the clock and reset sources, the Nios II data_master, and export the

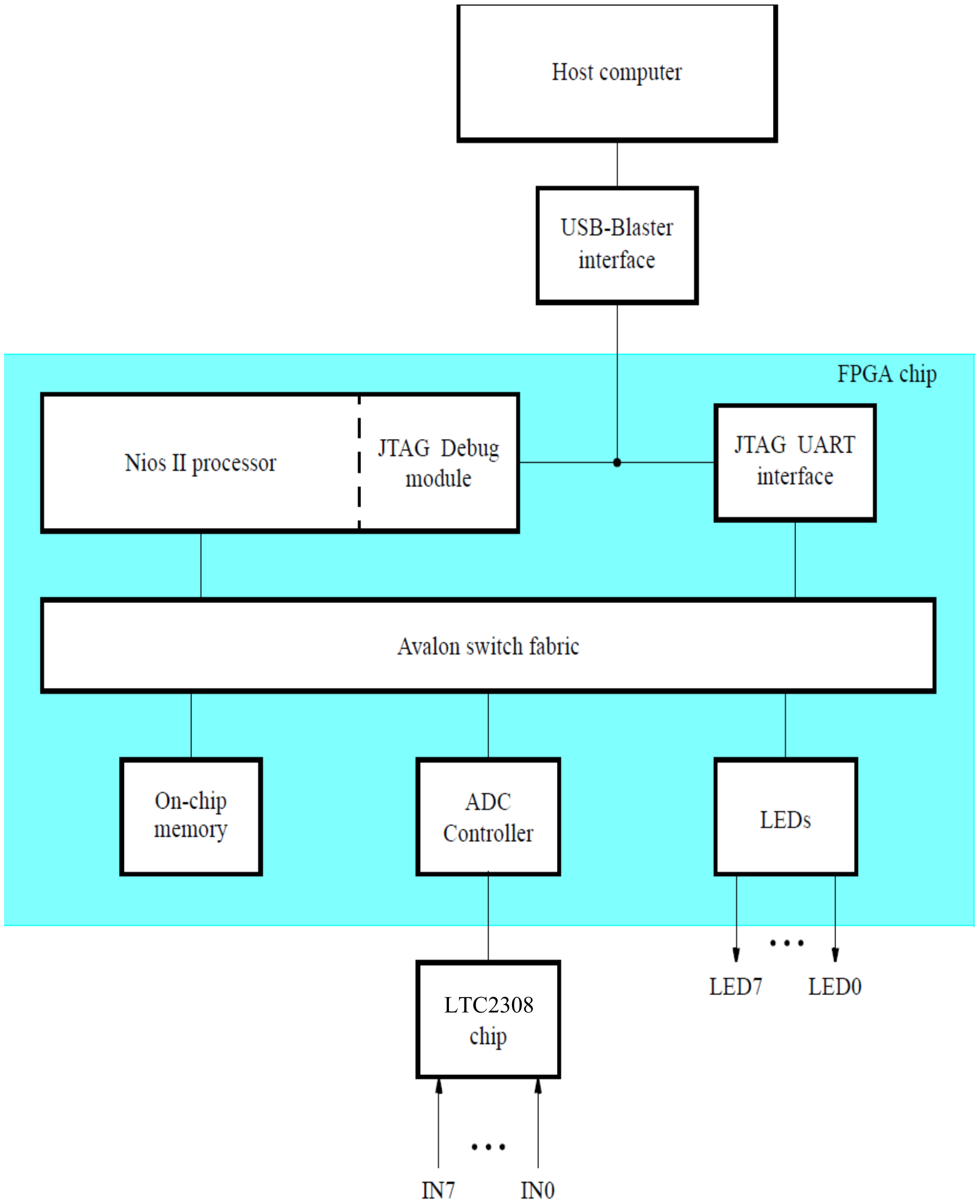


Figure 8. A simple Nios II system with the ADC Controller.

conduit as *leds*.

Lastly, include the ADC controller in the system by selecting University Program > Generic IO > ADC Controller for DE-Series Boards from the component list. Select the DE0-Nano-SoC board and specify "2" as the number of channels used, then select Finish to add it to the system. After adding the ADC controller to your system, rename the component to *ADC*. Connect the *clk*, *reset* and *adc_slave* signals to the clock source, clock reset and data_master

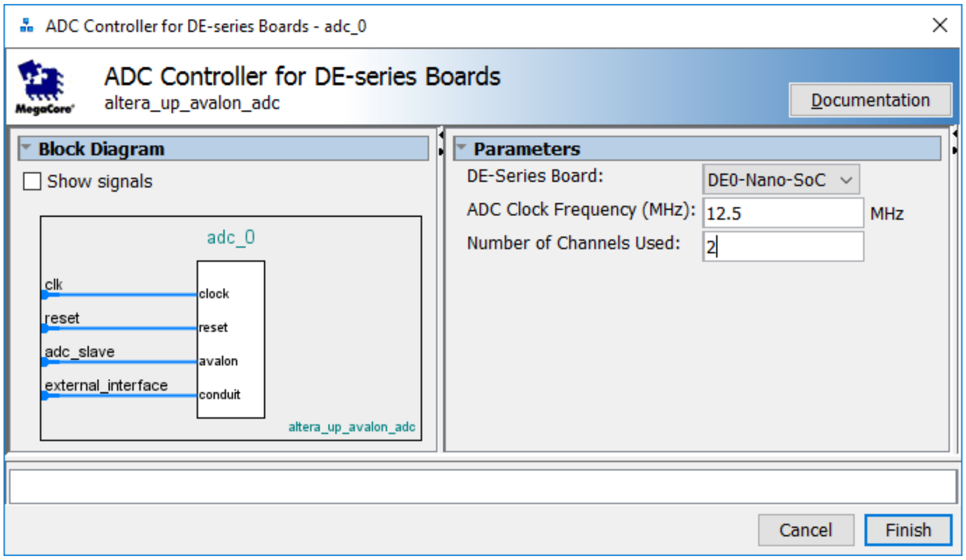


Figure 9. The ADC Controller component window.

sources, and export the *external_interface* signal as *adc*. Note: the ADC controller for the DE-10 Lite board does not contain an external_interface signal, as the connections are internal to the FPGA.

Assign the component addresses by selecting System > Assign Base Addresses. The system should match the one presented in Figure 10. Take note of the addresses assigned to the LEDs and the ADC controller, as these will be needed later. Save the system as *nios_system* and generate it using Generate > Generate HDL....

After generating the system, it is necessary to create a top-level module for the system. Create a new verilog file and copy the code from Figure 11 into it, or use the one provided in the "design files" subdirectory. Save this file as *adc_system.v*. Add the top level file just created and the *synthesis/nios_system.qip* file to project file list. Import the DE0-Nano-SoC Pin Assignments file and compile the project.

To use the ADC in a C program, declare a **volatile int *** for each peripheral, such as the ADC or LEDs. Assign this pointer the base address of the component as it was defined in Platform Designer. To read from or write to the component, use the dereference operator (*) to read or write values as appropriate. For peripherals with multiple registers, such as the ADC controller, treat the peripheral as an array of integer-sized values.

Example C code for operating the ADC is shown in Figure 12, and is available for use in the "design files" subdirectory. This code uses the first two channels of the ADC, alternating between them every 500,000 reads. The highest 8 bits for the channel will be displayed on the LEDs.

Use the Intel FPGA Monitor Program to download the system and C program to the FPGA chip. Users unfamiliar with the Intel FPGA Monitor Program should consult the *Intel FPGA Monitor Program Tutorial* for a detailed description of the program's features. To begin, create a new project using a custom system. Use the *.sopcinfo* generated by Platform Designer and the *.sof* file generated by Quartus to define the system. Next, choose C Program

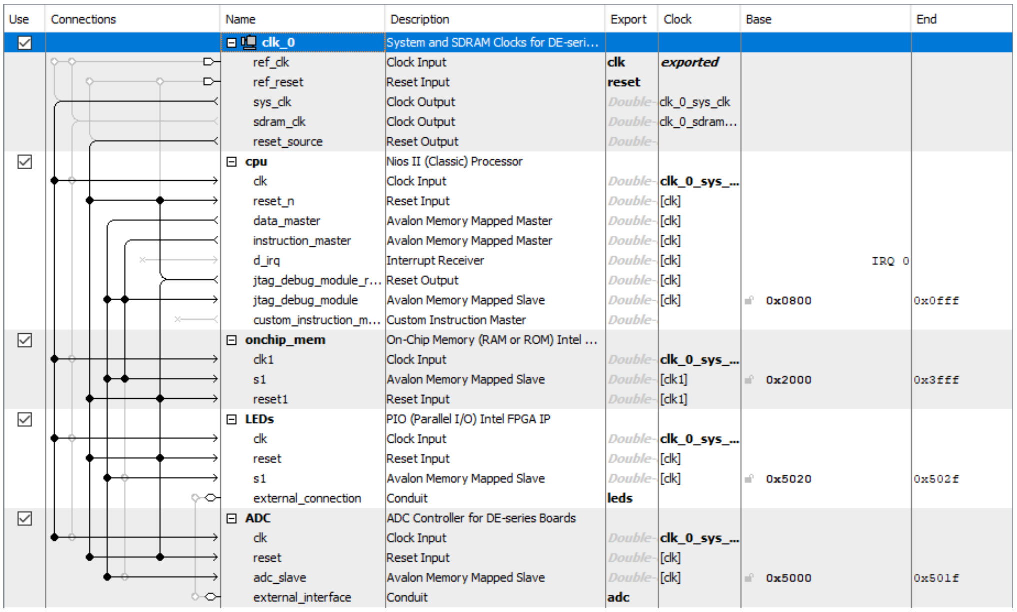


Figure 10. The system in Platform Designer with the ADC Controller.

```
module adc_demo (CLOCK_50, KEY, LED, ADC_SCLK,
            ADC_CONVST, ADC_SDO, ADC_SDI);
input CLOCK_50;
input [0:0] KEY;
output [7:0] LED;
input ADC_SDO;
output ADC_SCLK, ADC_CONVST, ADC_SDI;
nios_system NIOS (
    .clk_clk (CLOCK_50),
    .reset_reset (!KEY[0]),
    .leds_export (LED),
    .adc_sclk (ADC_SCLK),
    .adc_cs_n (ADC_CONVST),
    .adc_dout (ADC_SDO),
    .adc_din (ADC_SDI)
);
endmodule
```

Figure 11. Example top-level module for a project using the ADC Controller.

as the program type, and include the relevant C file. Leave all other settings unchanged, complete project creation and compile the program. After compilation is finished, load the program and select Actions > Continue to run it.

```c
/* Replace these addresses with the base addresses of the ADC and LEDs in
 * your Platform Designer project */
#define ADC_ADDR 0x00005000
#define LED_ADDR 0x00005020

int main (void){
    volatile int * adc = (int*)(ADC_ADDR);
    volatile int * led = (int*)(LED_ADDR);
    unsigned int data;
    int count;
    int channel;
    data = 0;
    count = 0;
    channel = 0;
    while (1){
        *(adc) = 0;             //Start the ADC read
        count += 1;
        data = *(adc+channel);  //Get the value of the selected channel
        data = data/16;         //Ignore the lowest 4 bits
        *(led) = data;          //Display the value on the LEDs
        if (count==500000){
            count = 0;
            channel = !channel;
        }
    }
    return 0;
}
```

Figure 12. C code to operate the ADC.

3.2 Using the ADC Controller with HAL

Alternatively, it is possible to use a processor and the various peripherals without creating a custom system. In this case, it is advantageous to use the *Hardware Abstraction Layer* or HAL. The HAL allows the use of task-specific function calls for accessing the peripheral, instead of accessing the peripheral directly. Additional details on the HAL can be found in the *Using HAL Device Drivers with the Intel FPGA Monitor Program* tutorial. The documentation for all University Program HAL devices can be found in the [Quartus Directory]/ip/University_Program directory.

The HAL Driver for the ADC offers five functions for accessing and controlling the ADC. To use these functions, the program must include the statement:

```
#include "altera_up_avalon_adc.h"
```

The first step when using the ADC with HAL is to create a device pointer to the ADC. HAL device drivers feature a different variable type for each device; for the ADC controller, the type "*alt_up_adc_dev*" is used. After creating the pointer, the value is assigned using the *alt_up_adc_open_dev (...)* function. This function takes in the name of the device and locates it within the system, and returns a pointer to the adc controller. If the default system is used, the string *"/dev/ADC"* should be used; otherwise, replace ADC with the name of the component as defined in the Platform Designer system. The result of this function should be assigned to the device pointer created for the ADC.

Once initialized, the other four functions can be used as desired. Definition prototypes and detailed descriptions for the HAL functions are shown in Figure 13. An alternative version of the C example presented above - now using the HAL - is shown in Figure 14, and in the "design files" subdirectory.

Having completed the code, load the program into the FPGA using the Intel FPGA Monitor Program. As in the previous section, a custom system can be used, though the use of HAL does allow the use of the *DE0-Nano-SoC Computer* instead. Additionally, instead of specifying the program type as C Program, choose Program with Device Driver Support. This option will include any relevant HAL drivers automatically during compilation, but the program will require significantly more memory. If the program is too large to fit in the on-chip memory, consider implementing an SDRAM module to provide additional memory for the system.

Compile and load the system and program to test the device.

alt_up_de0_nano_adc_open_dev

Prototype: `alt_up_adc_dev* alt_up_adc_open_dev( const char *name)`
Include: `<altera_up_avalon_adc.h>`
Parameters: `name` – the ADC Controller name. For example, if the ADC controller name in Platform Designer is "ADC", then *name* should be "/dev/ADC"
Returns: The corresponding device structure, or NULL if the device is not found.
Description: Open the ADC controller device specified by *name* .

alt_up_adc_read

Prototype: `unsigned int alt_up_adc_read ( alt_up_adc_dev *adc, unsigned channel)`
Include: `<altera_up_avalon_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
`channel` – the channel to be read, from 0 to 7.
Returns: data – The converted value from the desired channel.
Description: Read from a channel of the ADC.

alt_up__adc_update

Prototype: `void alt_up__adc_update( alt_up_adc_dev *adc)`
Include: `<altera_up_avalon_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Trigger the controller to convert all channels and store the values.

alt_up_adc_auto_enable

Prototype: `void alt_up_adc_auto_enable( alt_up_adc_dev *adc)`
Include: `<altera_up_avalon_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Enable automatic converting of channels.

alt_up_adc_auto_disable

Prototype: `void alt_up_adc_auto_disable( alt_up_adc_dev *adc)`
Include: `<altera_up_avalon_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Disable automatic converting of channels.

Figure 13. HAL functions for the ADC controller.

```c
#include "altera_up_avalon_parallel_port.h"
#include "altera_up_avalon_adc.h"
int main (void){
    alt_up_parallel_port_dev * led;
    alt_up_adc_dev * adc;
    unsigned int data;
    int count;
    int channel;
    data = 0;
    count = 0;
    channel = 0;

    led =alt_up_parallel_port_open_dev ("/dev/Green_LEDs");
    adc = alt_up_adc_open_dev ("/dev/ADC");

    while (led!=NULL&&adc!=NULL){
        alt_up_adc_update (adc);
        count += 1;
        data = alt_up_adc_read (adc, channel);
        data = data / 16;
        alt_up_parallel_port_write_data (led, data);
        if (count==500000){
            count = 0;
            channel = !channel;
        }
    }
    return 0;
}
```

Figure 14. C code using HAL to operate the ADC.

3.3 Using the ADC Controller with IP Catalog

To include the ADC controller in a hardware-based project, use the IP Catalog. Basic information on using the IP Catalog can be found in the *Using the Library of Parameterized Modules (LPM)* tutorial. The IP Catalog version of the controller allows access to between two and eight channels, with channel values updating automatically.

To instantiate the controller, open Tools > IP Catalog. Select University Program > Generic IO > ADC Controller for DE-series boards. The window in Figure 15 will appear. Set the values to match the figure and press OK. The window in Figure 16 will appear. Set the parameters of the ADC controller to those listed in the figure and then press Generate HDL... which will cause the window in Figure 17 to appear. Press Generate to generate the system. The window in Figure 18 may appear, if it does press Close after the system saves to continue the generation of HDL.

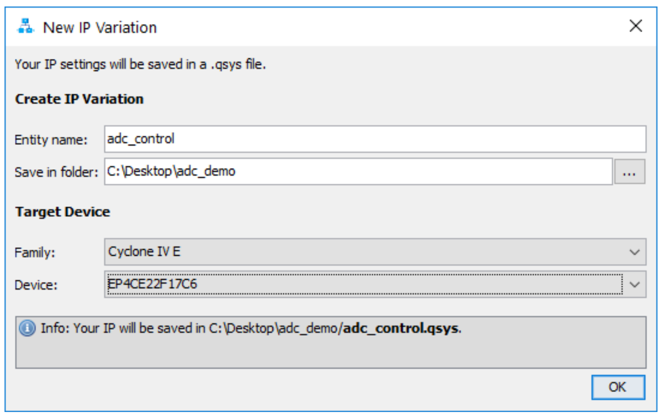


Figure 15. Creating a new IP variation of the ADC controller.

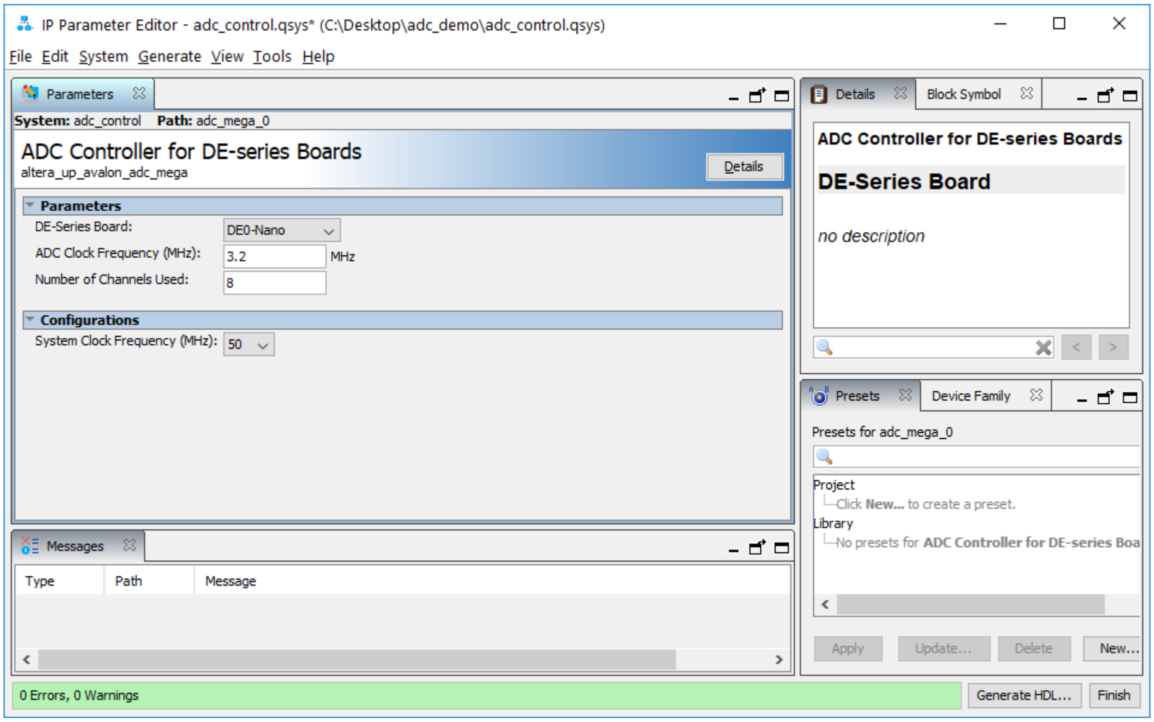


Figure 16. Configuring the ADC controller in the IP Catalog.

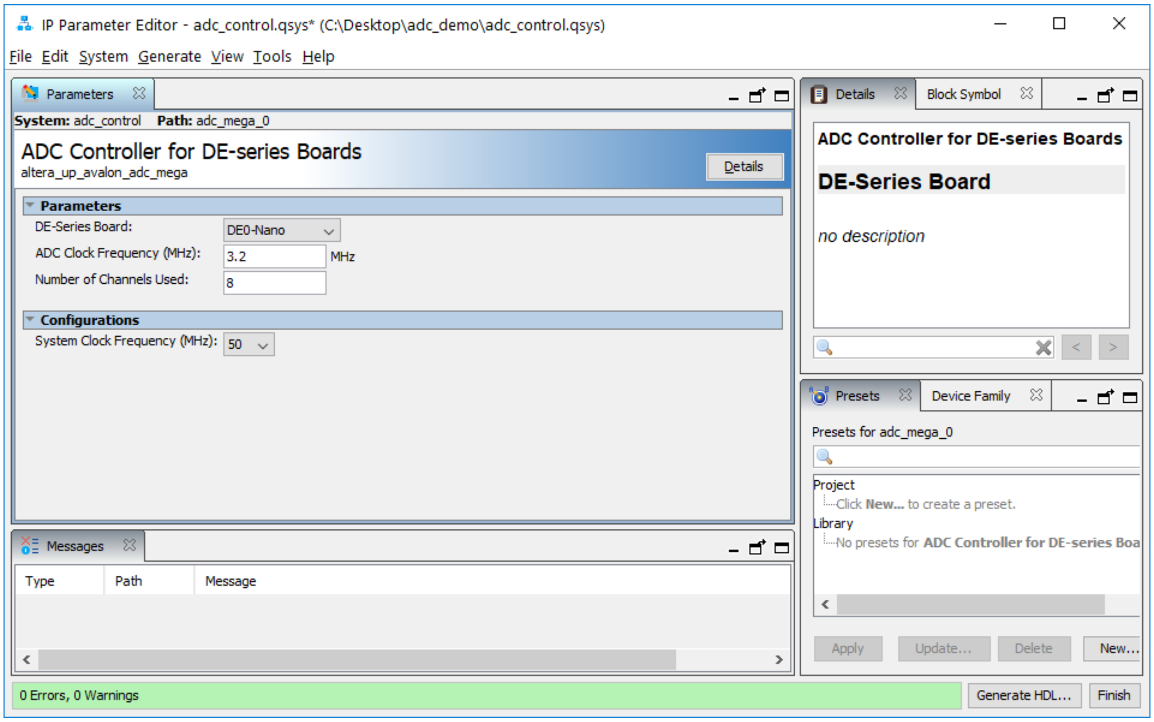


Figure 17. Generating the ADC controller.

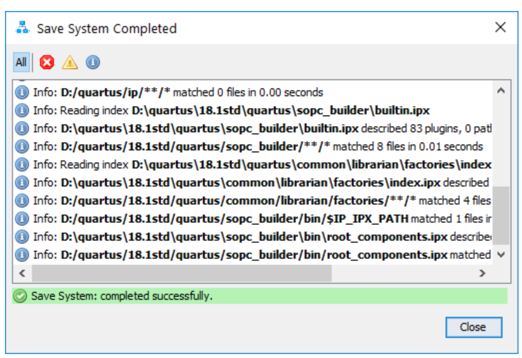


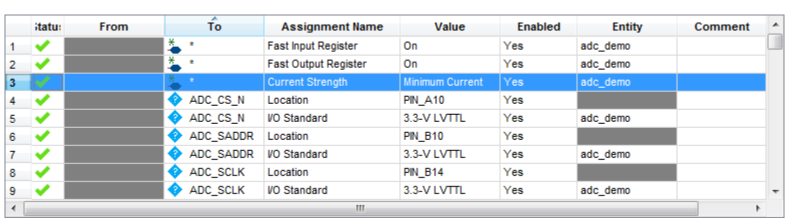
Figure 18. Saving the Platform Designer system for IP variation.

After finishing generating the system, add the IP core to the Quartus project by including the .qip file under <generation_directory>/synthesis folder. Once generation is complete, create a top-level file using the verilog code in Figure 19, or use the file *adc_demo_mega.v* in the "design files" directory. In this example, the Switches on the board are used to select the channel to display, from 0 to 7. The eight highest bits of the chosen channel are displayed on the LEDs.

```
module adc_demo_mega (CLOCK_50, KEY, SW, LED, ADC_SCLK,
            ADC_CONVST, ADC_SDO, ADC_SDI);
input CLOCK_50;
input [0:0] KEY;
input [2:0] SW;
output [7:0] LED;

input ADC_SDO;
output ADC_SCLK, ADC_CONVST, ADC_SDI;

wire [11:0] values [7:0];
assign LED = values [SW] [11:4];

adc_control ADC (
    .CLOCK (CLOCK_50),
    .RESET (!KEY[0]),
    .ADC_SCLK (ADC_SCLK),
    .ADC_CS_N (ADC_CONVST),
    .ADC_DOUT (ADC_SDO),
    .ADC_DIN (ADC_SDI),
    .CH0 (values[0]),
    .CH1 (values[1]),
    .CH2 (values[2]),
    .CH3 (values[3]),
    .CH4 (values[4]),
    .CH5 (values[5]),
    .CH6 (values[6]),
    .CH7 (values[7])
);
endmodule
```

Figure 19. Example top-level module for a project using the ADC Controller with IP Catalog.

Import the pin settings file corresponding to your board; it can be found on the Intel FPGA University program website. Open the assignment editor and delete the entry for Current Strength as shown in Figure 20; this must be done when compiling projects containing the ADC controller plugin. Compile the project and download it to the board.



	Statu:	From	To	Assignment Name	Value	Enabled	Entity	Comment
1			*	Fast Input Register	On	Yes	adc_demo	
2			*	Fast Output Register	On	Yes	adc_demo	
3			*	Current Strength	Minimum Current	Yes	adc_demo	
4			ADC_CS_N	Location	PIN_A10	Yes		
5			ADC_CS_N	I/O Standard	3.3-V LVTTL	Yes	adc_demo	
6			ADC_SADDR	Location	PIN_B10	Yes		
7			ADC_SADDR	I/O Standard	3.3-V LVTTL	Yes	adc_demo	
8			ADC_SCLK	Location	PIN_B14	Yes		
9			ADC_SCLK	I/O Standard	3.3-V LVTTL	Yes	adc_demo	

Figure 20. Modifying Assignment Editor Entry for Current Strength