Using the DE0-Nano ADC Controller

For Quartus II 14.0

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 Altera DE0-Nano board. 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 II and Qsys software.

Contents:

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 DE0-Nano board contains an ADC128S002 Analog-to-Digital Converter. This chip provides up to eight channels of analog input and converts them into a 12-bit digital signal.

2.1 ADC Signals

The ADC is connected to both the FPGA and the 2x13 GPIO header by several wires, as shown in Figure 1. The

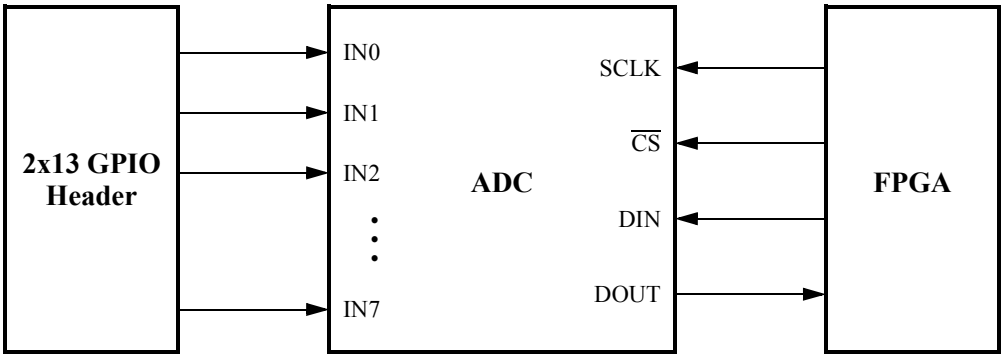

Figure 1. Signals to and from the ADC

ADC receives analog signals via the eight analog input pins *IN0* through *IN7*. When performing a conversion, the ADC reads the signal on one of these eight input channels and converts it to a digital output. On the DE0-Nano board, these eight pins are part of the 26-pin connector on the underside of the board, which is referred to as the 2x13 GPIO header.

In addition to the eight analog input wires, the ADC also has four wires connected to the FPGA. These 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

The ADC128S002 operates on a 16-cycle operational frame, as shown in Figure 2. 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.

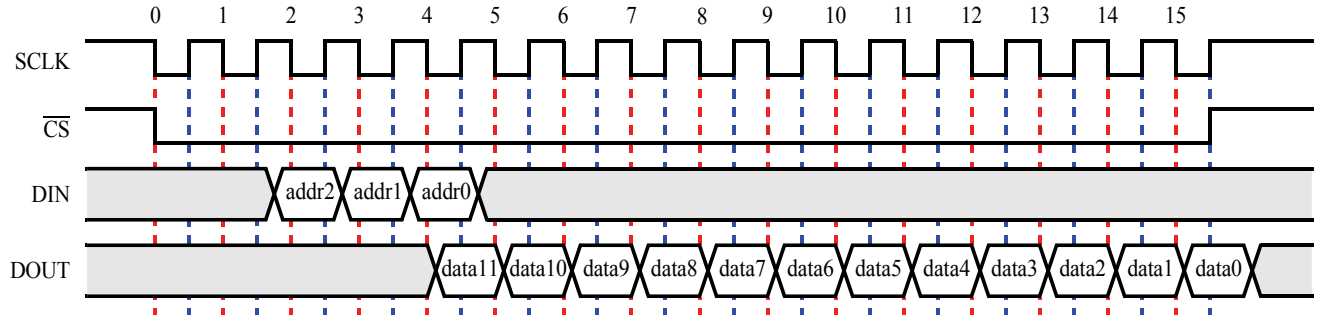

Figure 2. Timing requirements for the ADC

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. The *SCLK* frequency is limited to a range of 0.8 to 3.2 MHz in which the ADC will function correctly.

2.3 Analog Circuit Requirements

All analog inputs are referenced against a 3.3 V signal hardwired to the ADC. Therefore, to avoid damaging the DE0-Nano board, any voltages provided to the ADC via the 2x13 header pins should not exceed 3.3 V. If the analog circuitry is powered by a supply voltage greater than 3.3 V, voltage dividers should be used to limit the maximum output voltage to 3.3 V. Example analog circuits for measuring a variety of stimuli are shown in Figure 3. The resistance values given are approximate; all analog signals should be measured before being connected to the ADC.

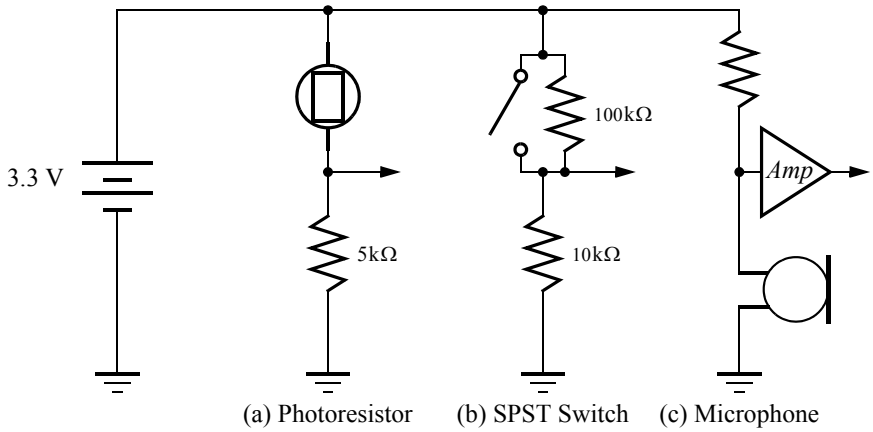

Figure 3. Examples of analog circuits using a variety of sensors.

Figure 3a 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 3.3 V and GND connections.

Figure 3b 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 3.3 V and GND connections.

Figure 3c 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 26) of the 2x13 GPIO header on the DE0-Nano board. This creates a common reference point for both the circuit and the board, so that voltages can be compared accurately. Figure 4 illustrates how an analog circuit should be connected to the board.

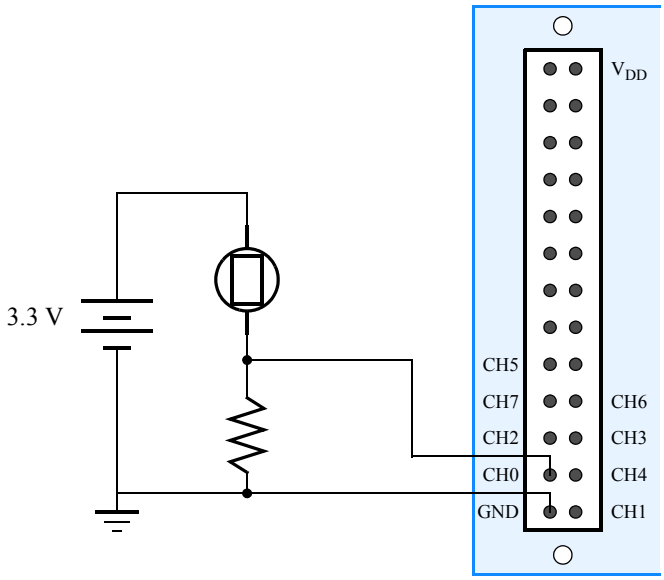

Figure 4. An analog circuit connected to the 2x13 GPIO header, shown from the underside of the DE0-Nano board.

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

3 The DE0-Nano ADC Controller

The DE0-Nano ADC Controller 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 of 0.8 to 3.2 MHz. 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 Qsys 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 Qsys component compatible with a Nios II processor. For information on designing systems with the Qsys tool

and Nios II, refer to the *Introduction to the Altera Qsys Integration Tool* and *Introduction to the Altera Nios II Soft Processor* tutorials.

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

Table 1. DE0-Nano 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 8 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.

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 intially 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 responsibilty 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 Implementing the ADC Core

To demonstrate the use of the ADC Controller, we will implement a system using the Qsys tool. The system will be controlled by a processor and software, and the converted values from the ADC will be displayed on the DE0-Nano's LEDs.

To make a new system with the ADC Controller, create a new project in Quartus II named *adc_demo*. The top-level module should also be *adc_demo*. Specify the device as *Cyclone IV E* **EP4CE22F17C6**, and complete the project creation. Then, open the Qsys 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 5.

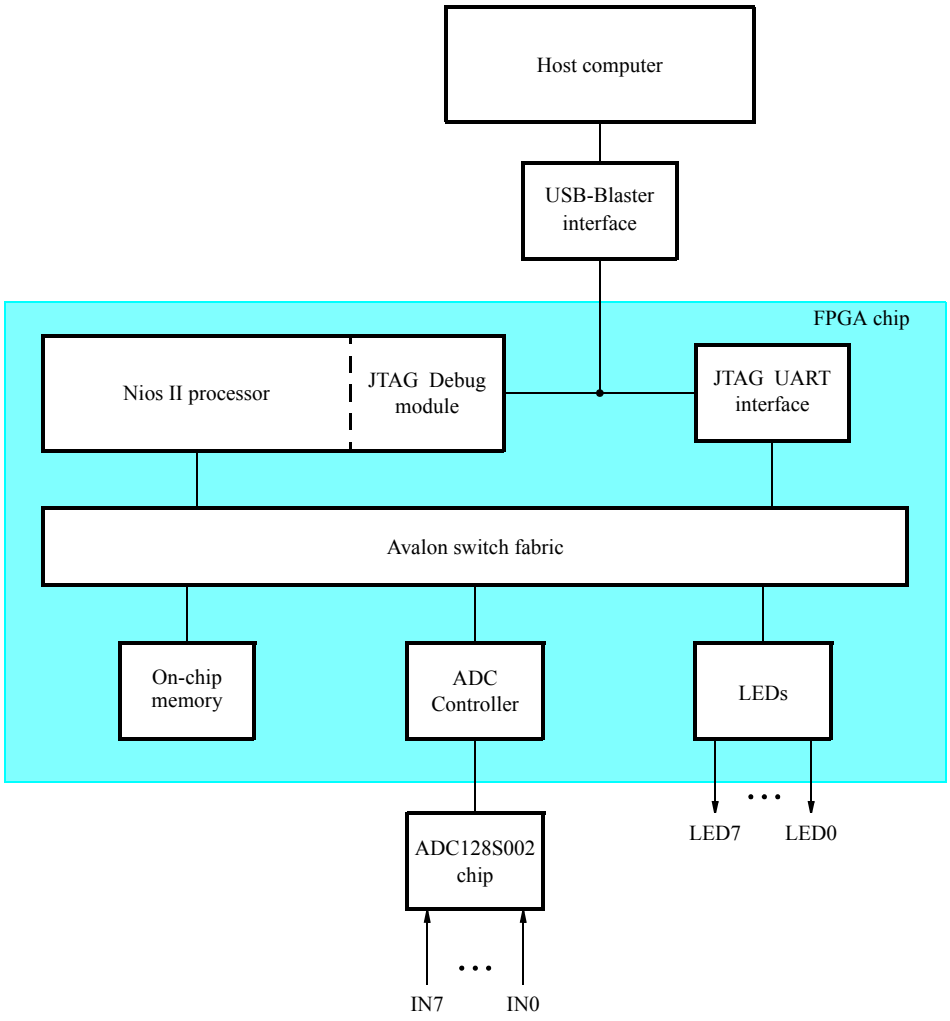

Figure 5. A simple Nios II system with the ADC Controller.

Firstly, rename the *clk_0* to *clk* by right-clicking on the *clk_0* and select rename.

From the Processors and Peripherals, select the Nios II 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 Parallel Port component to connect the system to the LEDs. Select University Program > Generic IO

and choose the Parallel Port component. Select DE0-Nano as the board type and LEDs as the I/O device. Rename the component to *LEDs*. Connect it to the clock and reset sources, the Nios II data_master, and export the conduit as *leds*.

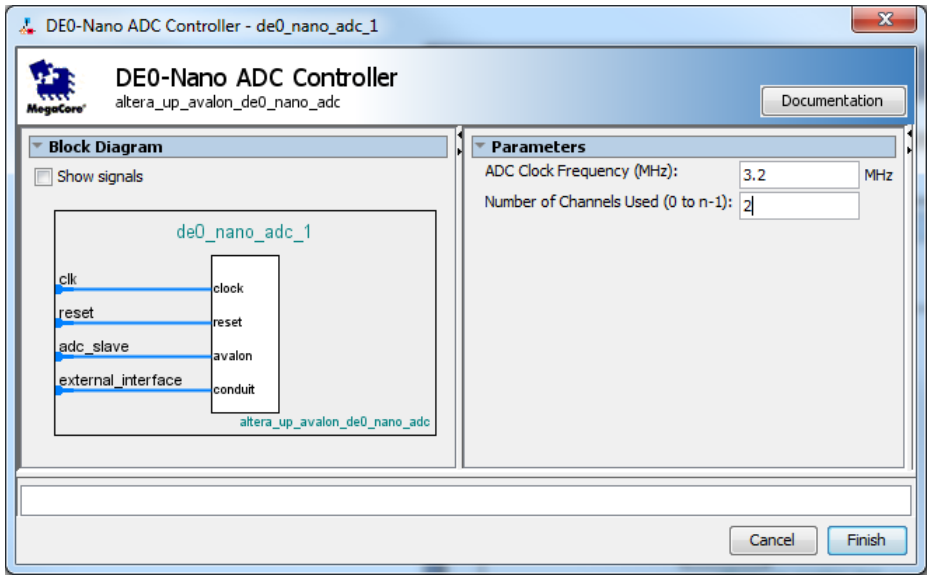

Figure 6. The ADC Controller component window.

Lastly, include the ADC Controller in the system by selecting *University Program* > *Generic IO* > *DE0-Nano ADC Controller* from the component list. Specify "2" as the number of channels used and 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 sources, and export the *external_interface* signal as *adc*.

Assign the component addresses by selecting System > Assign Base Addresses. The system should match the one presented in Figure 7. 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.

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 8 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 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 Qsys. 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 9, and is available for use in the "design files" sub-

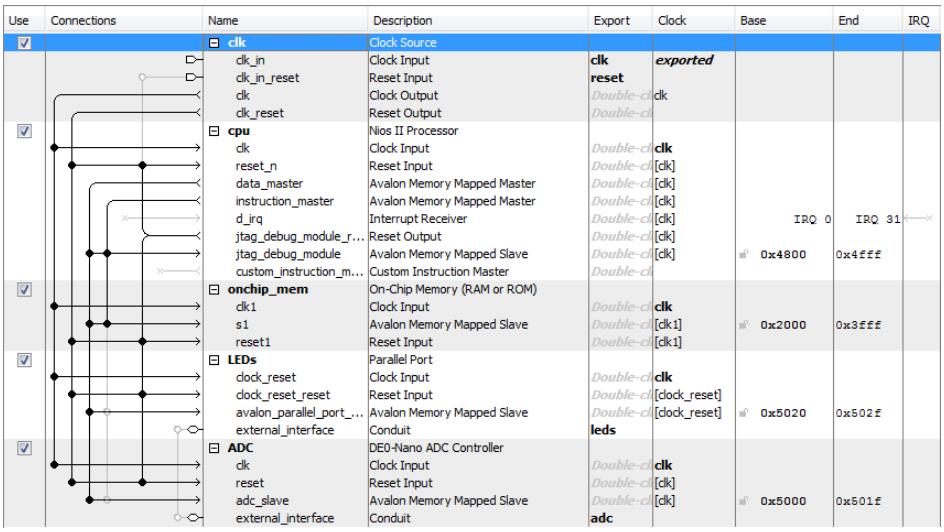

Figure 7. The system in Qsys with the ADC Controller.

directory. 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 displaced on the LEDs.

Use the Altera Monitor Program to download the system and C program to the FPGA chip. Users unfamiliar with the Altera Monitor Program should consult the *Altera 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 Qsys and the *.sof* file generated by Quartus to define the system. Next, choose C Program 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.

```
module adc_demo (CLOCK_50, KEY, LED, ADC_SCLK,
                 ADC_CS_N, ADC_SDAT, ADC_SADDR);
    input CLOCK_50;
    input [0:0] KEY;
    output [7:0] LED;

    input ADC_SDAT;
    output ADC_SCLK, ADC_CS_N, ADC_SADDR;

        nios_system NIOS (
            .clk_clk (CLOCK_50),
            .reset_reset_n (KEY[0]),
            .leds_export (LED),
            .adc_sclk (ADC_SCLK),
            .adc_cs_n (ADC_CS_N),
            .adc_dout (ADC_SDAT),
            .adc_din (ADC_SADDR)
        );
endmodule
```

Figure 8. Example top-level module for a project using the ADC Controller.

```
#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 9. 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 on the DE0-Nano board 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 Altera 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_de0_nano_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_de0_nano_adc_dev*" is used. After creating the pointer, the value is assigned using the *alt_up_de0_nano_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 Qsys 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 10. An alternative version of the C example presented above - now using the HAL - is shown in Figure 11, and in the "design files" subdirectory.

alt_up_de0_nano_adc_open_dev

Prototype: `alt_up_de0_nano_adc_dev* alt_up_de0_nano_adc_open_dev( const char *name)`
Include: `<altera_up_avalon_de0_nano_adc.h>`
Parameters: `name` – the ADC Controller name. For example, if the ADC controller name in Qsys 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_de0_nano_adc_read

Prototype: `unsigned int alt_up_de0_nano_adc_read ( alt_up_de0_nano_adc_dev *adc, unsigned channel)`
Include: `<altera_up_avalon_de0_nano_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_de0_nano_adc_update

Prototype: `void alt_up_de0_nano_adc_update( alt_up_de0_nano_adc_dev *adc)`
Include: `<altera_up_avalon_de0_nano_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Trigger the controller to convert all channels and store the values.

alt_up_de0_nano_adc_auto_enable

Prototype: `void alt_up_de0_nano_adc_auto_enable( alt_up_de0_nano_adc_dev *adc)`
Include: `<altera_up_avalon_de0_nano_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Enable automatic converting of channels.

alt_up_de0_nano_adc_auto_disable

Prototype: `void alt_up_de0_nano_adc_auto_disable( alt_up_de0_nano_adc_dev *adc)`
Include: `<altera_up_avalon_de0_nano_adc.h>`
Parameters: `adc` – struct for the ADC controller device .
Description: Disable automatic converting of channels.

Figure 10. HAL functions for the ADC controller.

Having completed the code, load the program into the FPGA using the Altera 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 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.

```
#include "altera_up_avalon_parallel_port.h"
#include "altera_up_avalon_de0_nano_adc.h"

int main (void){
    alt_up_parallel_port_dev * led;
    alt_up_de0_nano_adc_dev * adc;
    unsigned int data;
    int count;
    int channel;
    data = 0;
    count = 0;
    channel = 0;

    led =alt_up_parallel_port_open_dev ("/dev/LEDs");
    adc = alt_up_de0_nano_adc_open_dev ("/dev/ADC");

    while (led!=NULL&&adc!=NULL){
        alt_up_de0_nano_adc_update (adc);
        count += 1;
        data = alt_up_de0_nano_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 11. C code using HAL to operate the ADC.

3.3 Implementing 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.

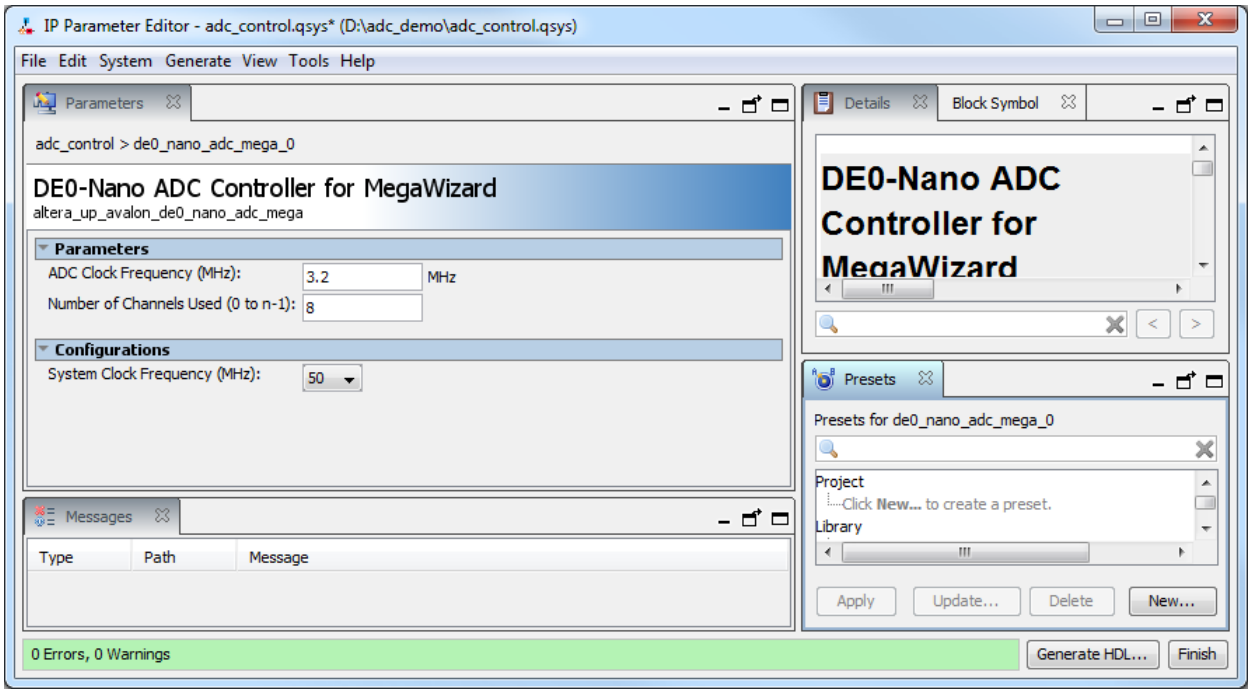

Figure 12. Configuring the ADC controller in the IP Catalog.

To instantiate the controller, open Tools > IP Catalog. Select University Program > Generic IO > DE0-Nano ADC Controller for MegaWizard from the list of plugins and specify the output file as *adc_control.qsys*. Select the desired number of channels, ADC clock frequency, and system clock frequency. The system clock frequency is required to create the requested ADC clock frequency, and must correspond to the frequency on the module's clock input. Set these values as required using Figure 12 as a reference, and click Generate HDL....

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 13, or use the one in the "design files" subdirectory. In this example, the Dip Switches on the DE0-Nano board are used to select the channel to display, from 0 to 7. The eight highest bits of the chosen channel will be displayed on the LEDs.

Compile the project and download it to the DE0-Nano board.

```
module adc_demo (CLOCK_50, KEY, SW, LED, ADC_SCLK,
                 ADC_CS_N, ADC_SDAT, ADC_SADDR);
    input CLOCK_50;
    input [0:0] KEY;
    input [2:0] SW;
    output [7:0] LED;

    input ADC_SDAT;
    output ADC_SCLK, ADC_CS_N, ADC_SADDR;

    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_CS_N),
            .ADC_SDAT (ADC_SDAT),
            .ADC_SADDR (ADC_SADDR),
            .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 13. Example top-level module for a project using the ADC Controller for MegaWizard.