Analog Dialogue

A forum for the exchange of circuits, systems, and software for real-world signal processing • Volume 49, Number 4, 2015

Analog Dialogue Since 1967

analog.com/analogdialogue

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

Editor's Notes

IN THIS ISSUE

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 2—Mode S Detection and Decoding Using MATLAB and Simulink

The series on model-based software-defined radio system design continues toward the ultimate objective of building a platform that will receive and decode the automatic dependent surveillance broadcast (ADS-B) transmissions from commercial aircraft. This month our designers analyze the Mode S extended squitter format used in ADS-B signal transmissions and demonstrate how to capture these Mode S signals with a receiver platform based on the AD9361 RF Agile Transceiver™ IC. (Page 3)

Using ESD Diodes as Voltage Clamps

When external overvoltage conditions are applied to an amplifier's input, ESD diodes are the last line of defense between the amplifier and the catastrophic damage that can result from electrical overstress. In this article our expert shows us how ESD cells are implemented in an amplifier device, discusses their characteristics, and explains how they can be used to improve the robustness of a design. (Page 8)

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 3—Mode S Signals Decoding Algorithm Validation Using Hardware in the Loop

Part 3 of our ongoing series on software-defined radio, covering validation of the system algorithm using live data as input and some powerful software and system development tools provided by ADI, MathWorks, and Avnet. If you've read Parts 1 and 2, you've seen this SDR system steadily progress from initial prototyping toward a production design. (Page 11)

Versatile, Precision Single-Ended-to-Differential Signal Conversion Circuit with Adjustable Output Common Mode Boosts System Dynamic Range

This article revisits an earlier design of a versatile, low power single-ended-to-differential converter circuit. This time around, this useful converter circuit is made even more versatile. The authors have addressed using the circuit in applications that require greater output dynamic range, such as in signal conditioning of temperature and pressure sensor outputs. (Page 16)

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 4—Rapid Prototyping Using the Zynq SDR Kit and Simulink Code Generation Workflow

This article concludes the four-part series on software-defined radio design. The authors bring the algorithm and hardware together and take their radio for a real-world test drive. This is the culmination of a journey that has taken us from simulation to prototyping to production-worthy design, and now we see it in action. (Page 19)

New Complete, High Resolution, and Multifunctional Bipolar DACs: an Easy to Use, Universal Solution

This article focuses on the critical role that precision bipolar DACs serve in calibration and control functions in a multitude of applications from motor control to industrial automation. Several system block diagrams are explored with a focus on the design considerations for the DAC functions involved therein. (Page 26)

Jim Surber [jim.surber@analog.com]

Product Introductions: Volume 49, Number 4

Data sheets for all ADI products can be found by entering the part number in the search box at analog.com.

October

Rail-to-rail input/output low noise amplifier	**ADA4807-4**
Low power, multimode, rail-to-rail amplifier for driving SAR ADCs	**ADA4806-1**
Multifunction video interface IC with (MIPI®/DSI) input port and (HDMI®) output	**ADV7535**
3.75 kV rms quad digital isolator family:	
Quad-channel isolator with input disable and 0 reverse channels	**ADuM140D**
Quad-channel isolator with input disable and 1 reverse channel	**ADuM141D**
Quad-channel isolator with input disable and 2 reverse channels	**ADuM142D**
Ultralow power step-down buck regulator	**ADP5300**
Complete, 1.25 GHz, dual integrated dcl with PPMU	**ADATE320**
Blackfin+ DSP family combines dual 16-bit MAC, 32-bit MAC, and 16-bit complex MAC with L2 SRAM and DDR2/LPDDR interface:	
400 MHz, 1 MB	**ADSP-BF707**
400 MHz, 512 kB	**ADSP-BF705**
400 MHz, 256 kB	**ADSP-BF703**
200 MHz, 128 kB	**ADSP-BF701**
Triple-channel OOK (on-off-keying) isolated coupler family:	
Input disable, 3/0 channel directionality	**ADuM230D**
Output enable, 3/0 channel directionality	**ADuM230E**
Output enable, 2/1 channel directionality	**ADuM231E**
Input disable, 2/1 channel directionality	**ADuM231D**
Input disable, 4/0 channel directionality	**ADuM240D**
Output enable, 4/0 channel directionality	**ADuM240E**
Input disable, 3/1 channel directionality	**ADuM241D**
Output enable, 3/1 channel directionality	**ADuM241E**
Input disable, 2/2 channel directionality	**ADuM242D**
Output enable, 4/0 channel directionality	**ADuM242E**
Ultralow power step-down buck regulator	**ADP5303**
50 mA/500 mA, ultralow power step-down regulator	**ADP5302**
Synchronous, pulse-width modulation (PWM) controller	**ADP1974**
2.5 GHz to 7 GHz, wideband, double balanced passive mixer	**HMC557A**
6 GHz to 26.5 GHz, GaS, MMIC fundamental mixer	**HMC773A**
JESD204B-compatible high performance dual-loop integer N jitter attenuator	**HMC7044**

November

Operational amplifier with a femtoampere level input bias current	**ADA4530-1**
High isolation, nonreflective, 0.1 GHz to 6 GHz silicon SPDT switch	**HMC8038**
Low power, complete 3-axis accelerometer with signal conditioned voltage outputs	**ADXL316**
Ultralow noise, low power current amplifier	**ADPD2210**
High quality, low power, single-input HDMI to LVDS display bridge	**ADV7613**
Highly efficient, ultralow quiescent current step-down regulator	**ADP5304**
High performance, dc-to-dc inverting regulator generates regulated negative rails	**ADP5073**
1.2 A dc-to-dc switching inverting regulator	**ADP5074**
High isolation, silicon SPDT, nonreflective switch, covers 9 GHz to 13.0 GHz range	**HMC1118**
Triple SPDT switch with user-defined fault protection and detection	**ADG5243F**

December

Zero-drift amplifier with wide operating voltage and temperature ranges	**ADA4522-4**
Single-channel digital isolator	**ADuM110N**
MMIC VCO with half frequency output 11.07 GHz to 11.62 GHz	**HMC1165**
GaAs, pHEMT, MMIC power amplifier provides 19 dB of gain	**HMC1144**
6-bit digital phase shifter that is rated from 4 GHz to 7 GHz	**HMC1133**

Analog Dialogue

Analog Dialogue, www.analog.com/analogdialogue, the technical magazine of Analog Devices, discusses products, applications, technology, and techniques for analog, digital, and mixed-signal processing. Published continuously for 49 years—starting in 1967—it is available in two versions. Monthly editions offer technical articles; timely information including recent application notes, circuit notes, new-product briefs, webinars, and published articles; and a universe of links to important and relevant information on the Analog Devices website, www.analog.com. Printable quarterly issues and ebook versions feature collections of monthly articles. For history buffs, the *Analog Dialogue* archive, www.analog.com/library/analogdialogue/archives.html, includes all regular editions, starting with Volume 1, Number 1 (1967), and three special anniversary issues. To subscribe, please go to www.analog.com/library/analogdialogue/subscribe.html. Your comments are always welcome: Facebook: www.facebook.com/analogdialogue; EngineerZone: ez.analog.com/blogs/analogdialogue; Email: dialogue.editor@analog.com or *Jim Surber*, Editor [jim.surber@analog.com].

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 2—Mode S Detection and Decoding Using MATLAB and Simulink

By Mike Donovan, Andrei Cozma, and Di Pu

Automatic Dependent Surveillance Broadcast Waveforms

Wireless signals that can be detected and decoded are everywhere, and they are easily accessible with today's software-defined radio (SDR) hardware like the Analog Devices AD9361/AD9364 integrated RF Agile Transceivers.™[1,2] The automatic dependent surveillance broadcast (ADS-B) transmissions from commercial aircraft provide a readily available wireless signal that can be used to demonstrate a rapid prototyping flow based on the AD9361 connected to a Xilinx® Zynq®-7000 All Programmable SoC. Commercial aircraft use ADS-B transmitters to report their position, velocity, altitude, and aircraft ID to air traffic controllers.[3] The flight data format is defined in the International Civil Aviation Organization's (ICAO) Mode S Extended Squitter specification.[4] ADS-B is being introduced throughout the world to modernize air traffic control and collision avoidance systems. It has already been adopted in Europe and is being gradually introduced in the United States.

The Mode S Extended Squitter standard provides details of the RF transmission format and encoded data fields. The transponder transmission has the following properties:

- Transmit frequency: 1090 MHz
- Modulation: pulse position modulation (PPM)
- Data rate: 1 Mbps
- Message length: 56 µs or 112 µs
- 24-bit CRC checksum

The tuning frequency and bandwidth are well within the capabilities of the AD9361 RF transceiver, and the received I/Q samples can be detected and decoded with a variety of software or embedded platform options.

In this article we will discuss how to capture these Mode S signals with a receiver platform based on the AD9361, and then use MATLAB and Simulink® to develop an algorithm that can decode the messages. The algorithm will be developed with the ultimate goal of deploying the solution onto a Zynq SoC platform, such as Avnet's PicoZed™ SDR System on Module (SOM).

Receiver Design Challenges

Mode S messages are either short (56 µs) or long (112 µs). Short messages contain the message type, aircraft identification number, and a cyclic redundancy check (CRC) checksum. Long messages also contain the altitude, position, velocity, and flight status. In either case, the Mode S transmission begins with an 8 µs preamble. This preamble pattern is used by receivers to establish that a valid message is being transmitted and helps the receivers determine when the message bits start. See Figure 1 for details.[5]

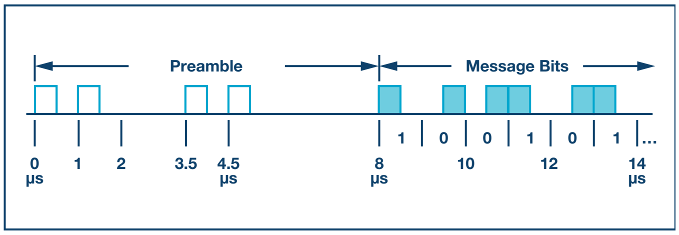


Figure 1. Structure of a Mode S message.

The Mode S waveform is fairly simple, but there are still several challenges involved in successfully receiving and decoding the transmitted messages.

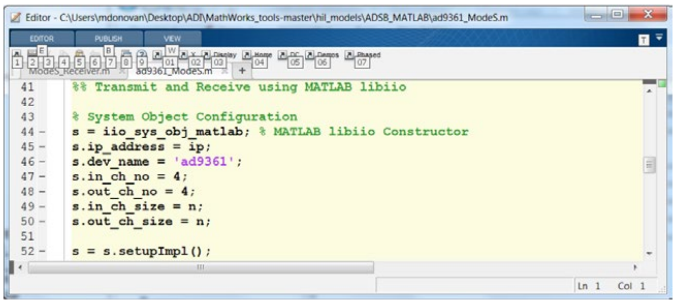
1. The receive environment typically contains very short messages interspersed with long idle periods, and the received signals can be very weak when the transmitting aircraft is a long distance from the receiver. Legacy waveforms are also transmitted at 1090 MHz. The receiver needs to use the preamble to identify both high and low amplitude Mode S transmissions in a congested frequency band.
2. Bits have one of two possible patterns within the 1 µs bit interval. A Logic 1 is ON for the first ½ µs and OFF for the second ½ µs. A Logic 0 is OFF for the first ½ µs and ON for the second ½ µs. Since the bit decisions are made based on time-based patterns, the receiver needs to use the preamble to accurately find the I/Q sample where the message bits start.
3. The Mode S message is composed of 88 information bits and 24 checksum bits. The receiver needs to be able to clear registers, make bit decisions, compute the checksum, and read the checksum registers at the correct times. Timing control is required for the receiver to function properly.
4. For an embedded design, the decoding process has to work on a sample by sample basis. Storing large amounts of data for batch processing is not a realistic receiver design for an embedded system.

The combination of a powerful RF front end like the AD9361 and a technical computing language like MATLAB® greatly simplifies the problems associated with detecting and decoding these transmissions. Functions from MATLAB and Signal Processing Toolbox can be used to identify the sync pattern, calculate the noise floor, make bit decisions, and calculate the checksum. The conditional and execution control functions in MATLAB simplify the control logic. Accessing test data is easy, both from binary or text files, or streamed directly into MATLAB using the AD9361 SDR platforms. Finally, the interpreted nature of MATLAB makes it easy to interact with data, try different approaches, and interactively develop a solution.

Modeling and Verifying Mode S Receiver Algorithms in MATLAB

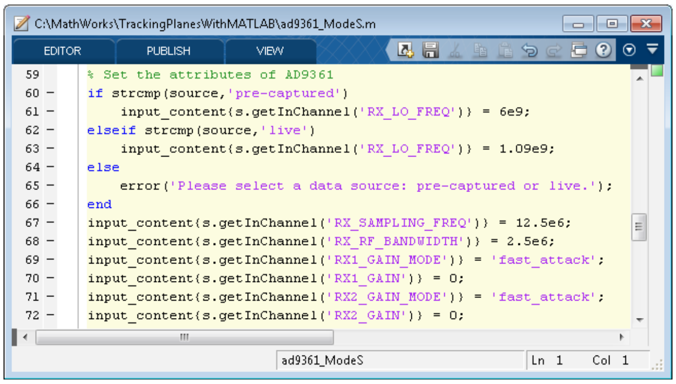
Readers who are interested in following along with the MATLAB source code can find the files on the Analog Devices GitHub repository. The entry level function is ad9361_ModeS.m, and the files called by this function are also provided.

The first step in designing a receiver algorithm is to access some source data. Since many aircraft are now equipped with Mode S transponders it's possible to just tune a receiver to the broadcast frequency of 1090 MHz and capture local transmissions. In our case we can use the Zynq SDR Rapid Prototyping Platform. Analog Devices provides a MATLAB System Object™ that is capable of receiving data from the FMCOMMS platform over Ethernet.[6] The System Object allows a user to select a tuning frequency and sampling rate, collect receive samples using the radio hardware, and bring the receive samples directly into the MATLAB workspace as a MATLAB variable. The required code is very short; a few lines of code to set up the MATLAB System Object, a few more to set up the FMCOMMS3, and a few lines of code to capture I/Q samples and write them to a MATLAB variable. A sample of the code is shown in Figure 2, Figure 3, and Figure 4.


Editor - C:\Users\mdonovan\Desktop\ADI\MathWorks_tools-master\hil_models\ADSB_MATLAB\ad9361_ModeS.m
EDITOR
PUBLISH
VIEW
ModeS_Receiver.m
ad9361_ModeS.m
Ln 1 Col 1

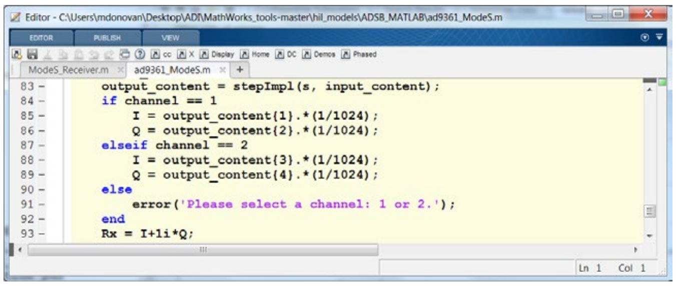
```
%% Transmit and Receive using MATLAB libiio

% System Object Configuration
s = iio_sys_obj_matlab; % MATLAB libiio Constructor
s.ip_address = ip;
s.dev_name = 'ad9361';
s.in_ch_no = 4;
s.out_ch_no = 4;
s.in_ch_size = n;
s.out_ch_size = n;

s = s.setupImpl();
```

Figure 2. Sample MATLAB code to set up MATLAB System object.



```
% Set the attributes of AD9361
if strcmp(source,'pre-captured')
    input_content(s.getInChannel('RX_LO_FREQ')) = 6e9;
elseif strcmp(source,'live')
    input_content(s.getInChannel('RX_LO_FREQ')) = 1.09e9;
else
    error('Please select a data source: pre-captured or live.');
end
input_content(s.getInChannel('RX_SAMPLING_FREQ')) = 12.5e6;
input_content(s.getInChannel('RX_RF_BANDWIDTH')) = 2.5e6;
input_content(s.getInChannel('RX1_GAIN_MODE')) = 'fast_attack';
input_content(s.getInChannel('RX1_GAIN')) = 0;
input_content(s.getInChannel('RX2_GAIN_MODE')) = 'fast_attack';
input_content(s.getInChannel('RX2_GAIN')) = 0;
```

Figure 3. Sample MATLAB code to configure FMCOMMS3 board.


Editor - C:\Users\mdonovan\Desktop\ADI\MathWorks_tools-master\hil_models\ADSB_MATLAB\ad9361_ModeS.m
EDITOR
PUBLISH
VIEW
ModeS_Receiver.m
ad9361_ModeS.m
Ln 1 Col 1

```
    output_content = stepImpl(s, input_content);
    if channel == 1
        I = output_content{1}.*(1/1024);
        Q = output_content{2}.*(1/1024);
    elseif channel == 2
        I = output_content{3}.*(1/1024);
        Q = output_content{4}.*(1/1024);
    else
        error('Please select a channel: 1 or 2.');
    end
    Rx = I+1i*Q;
```

Figure 4. Sample MATLAB code to capture I/Q samples and write them to the Rx variable.

We used some code based on these commands to capture several data sets at a sample rate of 12.5 MHz. The 12.5 MHz rate was chosen to provide enough samples to fine tune the alignment of the preamble to the first message bit and average out some of the noise in the samples used to make bit decisions. The results of a one million sample capture are shown in Figure 5.

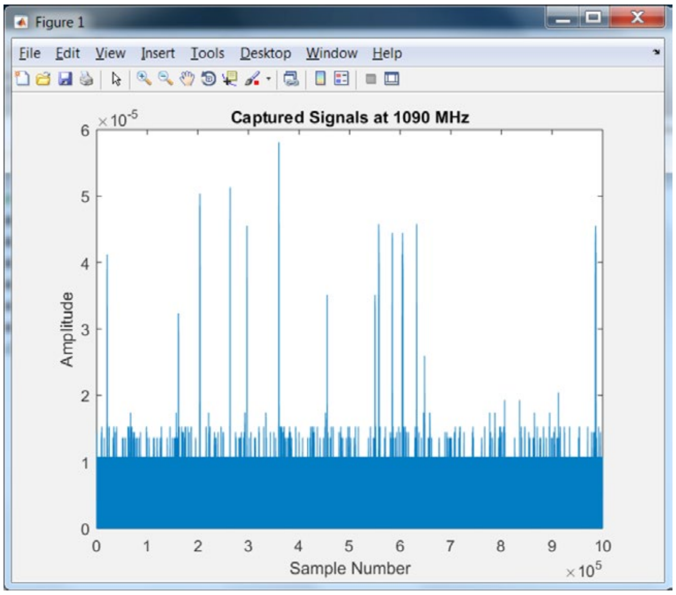


Figure 5. Sample data capture at 1090 MHz.

In this short data set there are 14 signals that stand out above the noise floor. Of those 14 signals, two are Mode S messages. The rest are legacy or spurious signals that should be rejected. Zooming in to the region near sample number 604000 shows one of the valid messages (see Figure 6).

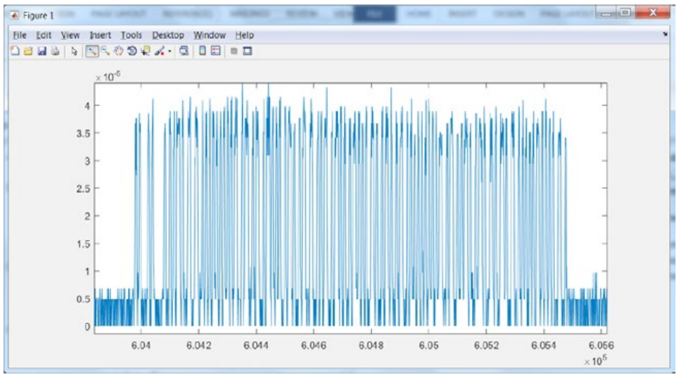


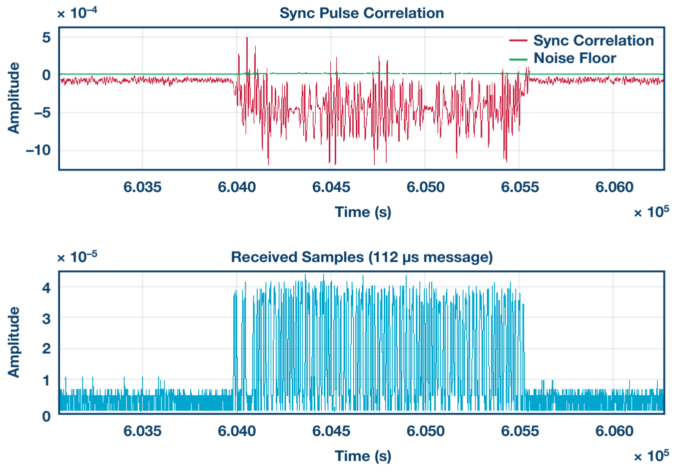
Figure 6. Single Mode S message.

In this plot the preamble can clearly be seen, and the bit transitions due to the PPM modulation are apparent. Even with a clean signal like this, decoding the bits by inspection would require good eyesight and a lot of patience. Clearly an automated program is required to decode these messages. MATLAB is a good solution for developing this program.

The MATLAB code that can receive and decode Mode S messages can be summarized as follows:

1. Calculate the noise floor and preamble correlation with the filter() function over a short time window. In our solution we use 75 samples, which is equivalent to 6 μs.

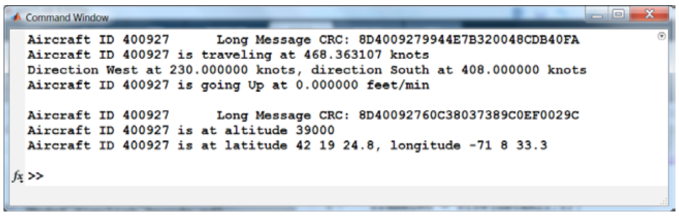
2. When the preamble correlation exceeds the noise floor by a significant factor, launch logic to find the first message bit sample.
   a. The choice of this threshold is subjective. It should be small enough to detect weak signals but large enough to prevent a lot of false positives. We chose a value of 10× above the noise floor as a reasonable threshold that captures most decodable messages.
   b. The preamble pattern produces several peaks. Since the best match is over the first 6 μs, store the first peak, start the search for the first message bit, and see if another larger peak occurs in the next 3 μs. If it does occur, store the new peak and reset the search for the start of the first message bit.
   c. When the max peak occurs, start the message bit decoding 2 μs later.
   d. Figure 7 shows the noise floor in green and the result of correlating an ideal preamble to the incoming data. There are several peaks above the noise floor, but the one of interest is the one with the maximum amplitude. The sample for the first message bit occurs 2 μs after that peak.



© 1984–2015 The MathWorks, Inc.

Figure 7. Calculation of noise floor and preamble correlation.

3. For each individual bit, sum the amplitude of the samples for the first ½ μs and second ½ μs. Whichever sum is larger determines whether the bit is Logic 1 or Logic 0.
4. Compute the checksum as the bit decisions are made. This requires some control logic for resetting the CRC registers when the first bit arrives, calculating the checksum for 88 bits, and then emptying the CRC registers for the final 24 bits. The ADS-B message is valid when the receive bits match the checksum.
5. Parse the message bits according to the Mode S standard (see Figure 8).



```
Aircraft ID 400927      Long Message CRC: 8D4009279944E7B320048CDB40FA
Aircraft ID 400927 is traveling at 468.363107 knots
Direction West at 230.000000 knots, direction South at 408.000000 knots
Aircraft ID 400927 is going Up at 0.000000 feet/min

Aircraft ID 400927      Long Message CRC: 8D40092760C38037389C0EF0029C
Aircraft ID 400927 is at altitude 39000
Aircraft ID 400927 is at latitude 42 19 24.8, longitude -71 8 33.3

fx >>
```

© 1984–2015 The MathWorks, Inc.

Figure 8. Decoded Mode S messages.

The above figure from the MATLAB command window shows the two messages that were successfully decoded from the one million sample data set. The hex characters that make up the 88-bit message and 24-bit checksum are displayed, and the results of the decoding process show the aircraft ID, message type, and aircraft velocity, altitude, and position.

MATLAB provides a powerful mathematical and signal processing language to make it possible to solve this problem relatively easily. The MATLAB code needed to process the data samples and ultimately decode the messages is short—only 200 lines of MATLAB code. In addition, the interpreted nature of MATLAB makes it easy to interactively try out design ideas and quickly settle on a viable solution. Several timing mechanisms, thresholds, and noise levels were tested on various data sets to produce a satisfactory program.

This MATLAB code has been tested on signals from aircraft flying in the local airspace and the decoded messages have been checked against sources like airframes.org and flightaware.com. The hardware and the code perform very well; we've been able to decode transmissions from planes at a distance of 50 miles.

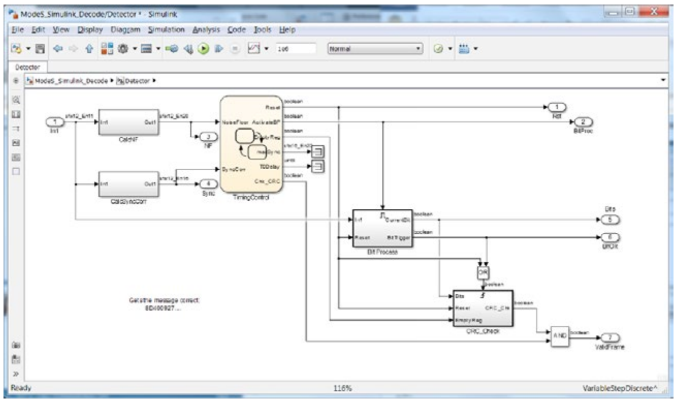
Path to Implementation

Readers who are interested in following along with the Simulink model can find the files on the Analog Devices GitHub repository:

https://github.com/analogdevicesinc/MathWorks_tools/tree/master/hil_models/ADSB_Simulink

MATLAB is a great environment for testing design ideas and running algorithms on a PC, but if the ultimate goal is to produce software or HDL to be used on an embedded platform, particularly one like a Zynq SoC, then Simulink is a good solution. Simulink is well suited to modeling the hardware specific elaborations needed to target the programmable device. A good workflow is to use MATLAB to develop and verify an algorithm, and then translate the design into Simulink and continue down the development path to a final hardware implementation.

Fortunately, the MATLAB code for this algorithm processes data on a sample by sample basis, so the conversion to Simulink is fairly straightforward. In contrast to the 200 lines of MATLAB code, the Simulink model is simple to display and describe (see Figure 9).



ModeS_Simulink_Decode/Detector • Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
Normal
Detector
TimingControl
BitProcess
CRC_Check
Ready
116%
VariableStepDiscrete

© 1984–2015 The MathWorks, Inc.

Figure 9. Simulink model of Mode S detection and decoding algorithm.

In Figure 9, you can see the first step in the decoding is to calculate the noise floor and the correlation to the preamble. Digital filter blocks are used for these calculations. The timing control block is implemented using Stateflow,® which is a state machine tool that is used to generate the timing, reset, and control signals for the rest of the decoding algorithm. Stateflow is very useful for models where you want to separate the control logic from the data flow. Once the timing and triggers are activated, the block named BitProcess takes the input I/Q samples and calculates the data bits, and the CRC_Check block computes the checksum. The message parsing still takes place in a MATLAB script driven by this Simulink model.

Digging deeper into the model you can see a few features that make Simulink suitable for embedded development, especially for partitioning the design into functions targeted at a Zynq SoC and for generating HDL code and C code.

1. Simulink has excellent fixed-point support, so you can build and test a bit-true version of your design. The individual blocks allow you to set the word length and fractional length for the mathematical operations in your model. The digital filter block that is used to calculate the preamble correlation is a good example (Figure 10). You can set the rounding mode and the overflow behavior for the calculations (Floor and Wrap are the simplest choices for math being done in HDL). In addition, you can specify different word lengths and fractional precisions for the product and accumulator operations for the filter (Figure 11). You can use word length choices that map to the receiver ADC and take advantage of hardware multipliers like the 18 bit × 25-bit multipliers within the DSP48 slices of the Zynq SoC.

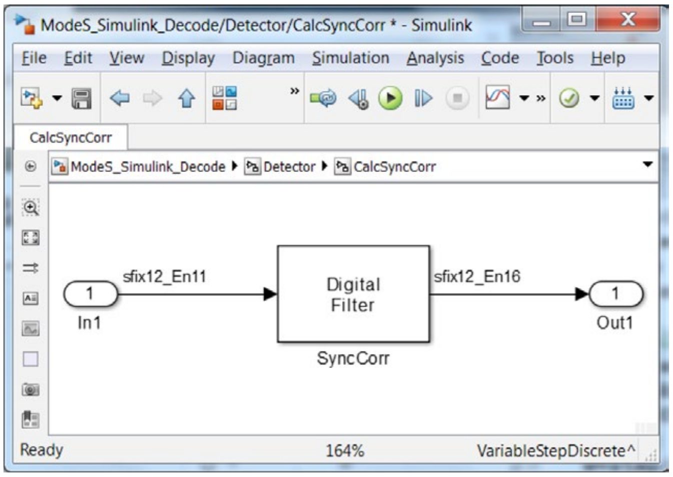
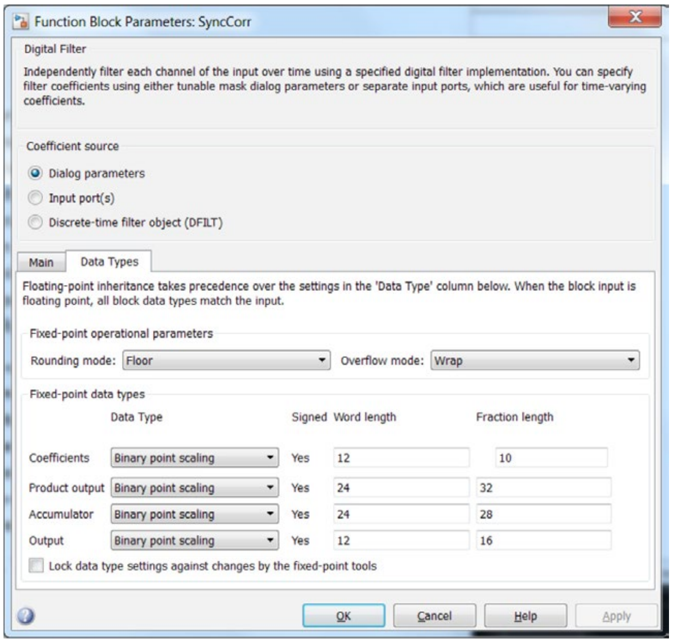

ModeS_Simulink_Decode/Detector/CalcSyncCorr * - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
CalcSyncCorr
ModeS_Simulink_Decode ▸ Detector ▸ CalcSyncCorr
sfix12_En11
1
In1
Digital Filter
SyncCorr
sfix12_En16
1
Out1
Ready
164%
VariableStepDiscrete

Figure 10. Simulink digital filter block used for preamble correlation, 12-bit data types.


Function Block Parameters: SyncCorr

Digital Filter

Independently filter each channel of the input over time using a specified digital filter implementation. You can specify filter coefficients using either tunable mask dialog parameters or separate input ports, which are useful for time-varying coefficients.

Coefficient source

● Dialog parameters
○ Input port(s)
○ Discrete-time filter object (DFILT)

Main | Data Types

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Fixed-point data types

	Data Type	Signed	Word length	Fraction length
Coefficients	Binary point scaling	Yes	12	10
Product output	Binary point scaling	Yes	24	32
Accumulator	Binary point scaling	Yes	24	28
Output	Binary point scaling	Yes	12	16

☐ Lock data type settings against changes by the fixed-point tools

Figure 11. Fixed-point data type settings.

2. Embedded designs often have many of modes of operation and conditionally executed algorithms. Stateflow is particularly good at managing these control signals. Stateflow gives you a visual representation of the control logic needed to detect and decode the Mode S messages. In Figure 12 below, you can see the states in the logic are:
   a. SyncSearch: look for the preamble in the captured samples
   b. WaitForT0: look for the start of the first message bit
   c. BitProcess: enable the bit processing
   d. EmptyReg: empty the checksum register and compare the bits to the output of the bit processing

As the detection and decoding algorithm progresses through the different states, the Stateflow block generates the signals that enable the bit processing, reset the bit decision counters and checksum registers, and read out the checksum bits at the end of the Mode S messages.

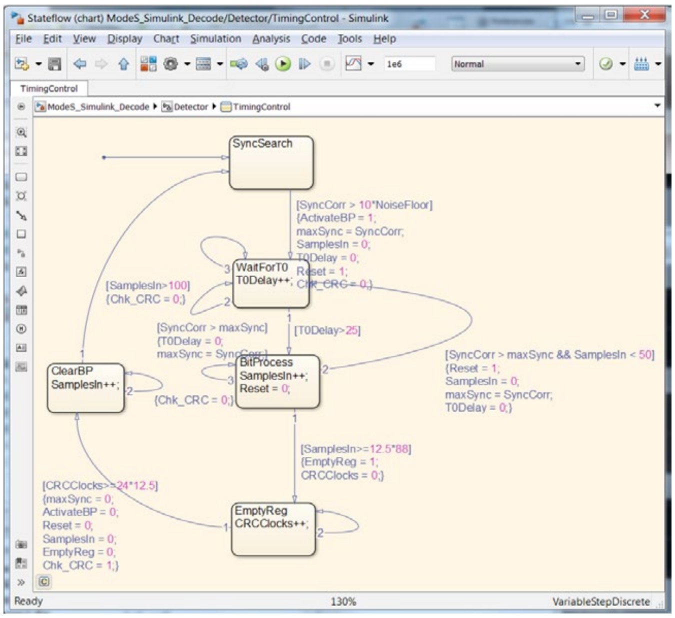

Stateflow (chart) ModeS_Simulink_Decode/Detector/TimingControl - Simulink
File Edit View Display Chart Simulation Analysis Code Tools Help
1e6
Normal
TimingControl
ModeS_Simulink_Decode ▸ Detector ▸ TimingControl
SyncSearch
[SyncCorr > 10*NoiseFloor] {ActivateBP = 1; maxSync = SyncCorr; SamplesIn = 0; T0Delay = 0; Reset = 1; Chk_CRC = 0;}
WaitForT0 T0Delay++;
[SamplesIn>100] {Chk_CRC = 0;}
[SyncCorr > maxSync] {T0Delay = 0; maxSync = SyncCorr;}
[T0Delay>25]
[SyncCorr > maxSync && SamplesIn < 50] {Reset = 1; SamplesIn = 0; maxSync = SyncCorr; T0Delay = 0;}
ClearBP SamplesIn++;
BitProcess SamplesIn++; Reset = 0;
{Chk_CRC = 0;}
[SamplesIn>=12.5*88] {EmptyReg = 1; CRCClocks = 0;}
[CRCClocks>=24*12.5] {maxSync = 0; ActivateBP = 0; Reset = 0; SamplesIn = 0; EmptyReg = 0; Chk_CRC = 1;}
EmptyReg CRCClocks++;
Ready
130%
VariableStepDiscrete

Figure 12. Stateflow chart for decoding Mode S messages.

3. The Simulink block libraries give engineers options to work at a very high level or at a very fine level of detail. Simulink has high level blocks like Digital Filter, FFT, and Numerically Controlled Oscillator to make it easy to build signal processing designs. If more precise control of the design is required, possibly for speed or area optimizations, engineers can use low level blocks like Unit Delays, Logic Operators (XOR for example), and Switches. The 24-bit checksum in this model is a feedback shift register built using those low level blocks (Figure 13).

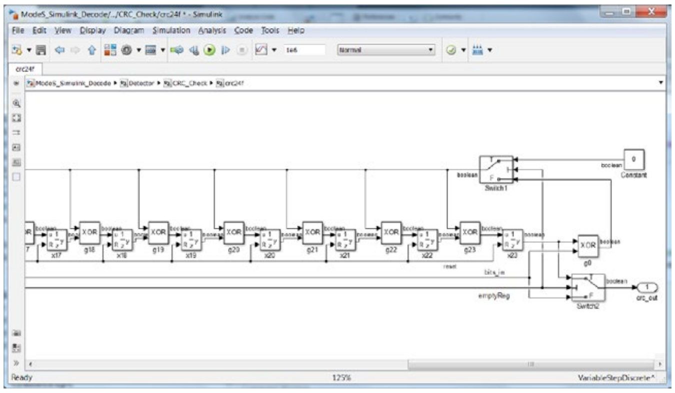

ModeS_Simulink_Decode/.../CRC_Check/crc24f * - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
crc24f
ModeS_Simulink_Decode ▸ Detector ▸ CRC_Check ▸ crc24f
XOR
Switch1
Constant
Switch2
crc_out
Ready
125%
VariableStepDiscrete

Figure 13. Feedback shift register for Mode S checksum computation.

This Simulink model is a hardware specific version of the MATLAB algorithm that detects and decodes Mode S messages. Simulink is a useful tool for bridging the gap between a behavioral algorithm written in MATLAB and implementation code for embedded hardware. You can introduce hardware specific elaborations into the Simulink model, run the model, and verify that the changes you've made don't break the decoding algorithm.

Conclusion

The combination of a Zynq SDR Rapid Prototyping Platform and MathWorks software gives communication engineers a new and flexible way to quickly prototype design ideas for wireless receivers. The high degree of programmability and performance provided by the AD9361/AD9364 agile wideband RF transceiver and the simple connectivity between the hardware and MATLAB environment makes a wide variety of interesting wireless signals available to the engineer. Engineers who use MATLAB can quickly try a multitude of design ideas and settle on a promising solution. If the ultimate target of the design is an embedded processor, Simulink is a tool that engineers can use to refine the design with hardware specific ideas and ultimately produce the code used to program the processor. This workflow reduces the number of skills needed to design a wireless receiver and shortens the development cycle from concept to working prototype.

In the next article in this series, we will show how to use hardware in the loop (HIL) to validate a receiver design, capturing signals with the target transceiver while executing a model of the signal processing system on the host in Simulink for verification.

References

[1] AD9361. Analog Devices.

[2] AD9364. Analog Devices.

[3] 960-1164 MHz. National Telecommunications and Information Administration.

[4] Technical Provisions for Mode S Services and Extended Squitter. International Civil Aviation Organization.

[5] Surveillance and Collision Avoidance Systems. *Aeronautical Telcommunications, Volume IV.* International Civil Aviation Organization.

[6] Di Pu, Andrei Cozma, and Tom Hill. "Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio." *Analog Dialogue*, Volume 49.

Links to Source Code and Models

MATLAB Mode S Decoding Algorithm: https://github.com/analogdevicesinc/MathWorks_tools/blob/master/hil_models/ADSB_MATLAB/

Simulink Mode S Decoding Models: https://github.com/analogdevicesinc/MathWorks_tools/tree/master/hil_models/ADSB_Simulink

Acknowledgements

The authors would like to thank Mike Mulligan from MathWorks who contributed some of the MATLAB code used in this example.

Mike Donovan [mike.donovan@mathworks.com] is a manager in the Application Engineering Group at MathWorks. He has a B.S.E.E. from Bucknell University and an M.S.E.E. from the University of Connecticut. Prior to joining MathWorks, Mike worked on radar and satellite communications systems and in the broadband telecommunications industry.

Mike Donovan

Andrei Cozma [andrei.cozma@analog.com] is an engineering manager for ADI, supporting the design and development of system level reference designs. He holds a B.S. degree in industrial automation and informatics and a Ph.D. in electronics and telecommunications. He has been involved in the design and development of projects from different industry fields such as motor control, industrial automation, software-defined radio, and telecommunications.

Andrei Cozma

Also by this Author:

FPGA-Based Systems Increase Motor-Control Performance

Volume 49, Number 1

Di Pu [di.pu@analog.com] is a system modeling applications engineer for ADI, supporting the design and development of software-defined radio platforms and systems. She has been working closely with MathWorks to solve mutual end customer challenges. Prior to joining ADI, she received her B.S. degree from Najing University of Science and Technology (NJUST), Nanjing, China, in 2007 and her M.S. and Ph.D. degrees from Worcester Polytechnic Institute (WPI), Worcester, MA, U.S.A., in 2009 and 2013—all in electrical engineering. She is a winner of the 2013 Sigma Xi Research Award for Doctoral Dissertation at WPI.

Di Pu

Using ESD Diodes as Voltage Clamps

By Paul Blanchard and Brian Pelletier

Abstract

When external overvoltage conditions are applied to an amplifier, ESD diodes are the last line of defense between your amplifier and electrical over stress. With proper understanding of how an ESD cell is implemented in a device, a designer can greatly extend the survival range of an amplifier with the appropriate circuit design. This article aims to introduce readers to the various types of ESD implementations, discuss the characteristics of each implementation, and provide guidance on how to utilize these cells to improve the robustness of a design.

Introduction

In many applications where the input is not under system control but rather connects to the outside world, such as test equipment, instrumentation, and some sensing equipment, it is possible for input voltages to exceed the maximum rated voltage of a front-end amplifier. In these applications, protection schemes must be implemented to preserve the survival range and robustness of the design. The front-end amplifier's internal ESD diodes are sometimes used for clamping overvoltage conditions, but many factors need to be considered to ensure these clamps will provide sufficient and robust protection. Understanding the various ESD diode architectures that are inside of front-end amplifiers, along with understanding the thermal and electromigration implications of a given protection circuit, can help a designer avoid problems with their protection circuits and improve the longevity of their applications in the field.

ESD Diode Configurations

It is important to understand that not all ESD diodes are simple diode clamps to the power supplies and ground. There are many possible implementations that can be used, such as multiple diodes in series, diodes and resistors, and back to back diodes. Some of the more common implementations are detailed below.

Diodes Connected to the Power Supply

Figure 1 shows an example of an amplifier with diodes connected between the input pins and the supplies. The diodes are reverse-biased under normal operating conditions, but become forward-biased as the inputs rise above the positive supply voltage or below the negative power supply voltage. As the diodes become forward-biased, current flows through the amplifier's inputs to the respective supply.

In the case of the circuit in Figure 1, the input current is not inherently limited by the amplifier itself when the overvoltage goes above $+V_s$, and will require external current limiting in the form of a series resistor. When the voltage goes below $-V_s$ the 400 Ω resistor provides some current limiting, which should be factored into any design considerations.

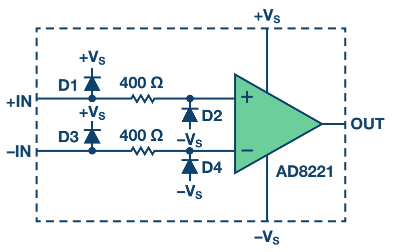


Figure 1. Input ESD topology of AD8221.

Figure 2 shows an amplifier with a similar diode configuration, but in this case the current is limited by the internal 2.2 kΩ series resistor. This differs from the circuit shown in Figure 1 by not only the value of the limiting R but also the 2.2 kΩ protects from voltages above $+V_s$. This is an example of the intricacies that must be fully understood to optimize protection when using ESD diodes.

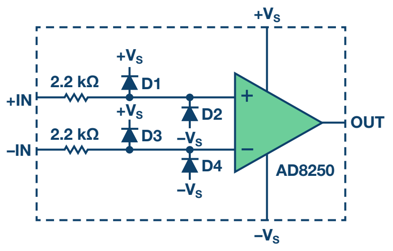


Figure 2. Input ESD topology of AD8250.

Current-Limiting JFETs

In contrast to the implementation in Figure 1 and Figure 2, current-limiting JFETs may be used in IC designs as an alternative to diode clamps. Figure 3 shows an example where JFETs are used to protect a device when the input voltages exceed the specified operating range of the device. This device is inherently protected up to 40 V from the opposite rail by the JFET inputs. Because the JFET will limit the current into the input pins, the ESD cells cannot be used as additional overvoltage protection.

Where voltage protection up to 40 V is required, this device's JFET protection offers a well controlled, reliable, fully specified option for protection. This is often in contrast to using ESD diodes for protection, where information on diode current limits are often specified as typical, or possibly not specified at all.

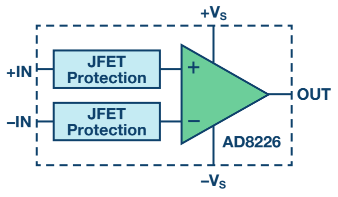


Figure 3. Input protection scheme of AD8226.

Diode Stacks

In applications where the input voltage is allowed to exceed the power supply voltage or ground, a stack of diodes may be used to protect the input from ESD events. Figure 4 shows an amplifier that implements a stacked diode protection scheme. In this configuration, the diode string is used to protect from negative transients. The string of diodes are used to limit the leakage current in a usable input range, but provide protection when the negative common-mode range is exceeded. Keep in mind the only current limiting will be the equivalent series resistance of the diode string. An external series resistance can be used to decrease the input current for a given voltage level.

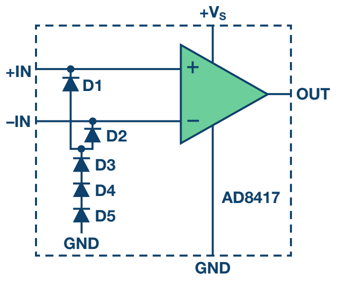


Figure 4. Low-side input protection scheme of AD8417.

Back to Back Diodes

Back to back diodes are also used when the input voltage range is allowed to exceed the power supply. Figure 4 shows an amplifier that implements back to back diodes to provide ESD protection on a device that allows voltages up to 70 V using a 3.3 V supply. D4 and D5 are high voltage diodes used to standoff the high voltages that could be present on the input pins and D1 and D2 are used to prevent leakage currents while the input voltages are within the normal operating range. In this configuration, using these ESD cells for overvoltage protection would not be recommended because exceeding the maximum reverse bias of the high voltage diode can easily lead to situations that cause permanent damage.

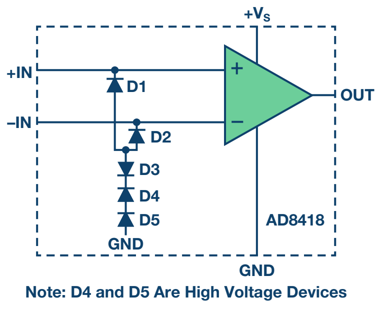


Figure 5. High-side input protection scheme of AD8418.

No ESD Clamps

Some devices do not include ESD devices on the front end. While it is obvious that a designer cannot use ESD diodes for clamping if they are not there, this architecture is mentioned as a situation to look out for when investigating overvoltage protection (OVP) options. Figure 6 shows a device that uses only large value resistors to protect the amplifier.

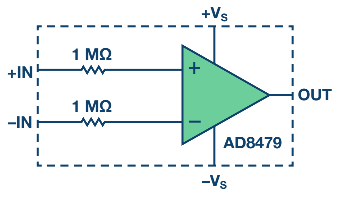


Figure 6. Input protection scheme of AD8479.

ESD Cells as Clamps

In addition to understanding how the ESD cells are implemented, it is important to understand how to utilize the structures for protection. In a typical application, a series resistor is used to limit the current over a specified voltage range.

When amplifiers are configured as shown in Figure 7 or where the inputs are protected by a diode to the supply, input current is limited using the equation in the following formula.

$$I_{DIODE} = \frac{V_{STRESS} - (V_{SUPPLY} + 0.7\ \text{V})}{R_{PROTECTION}} \qquad (1)$$

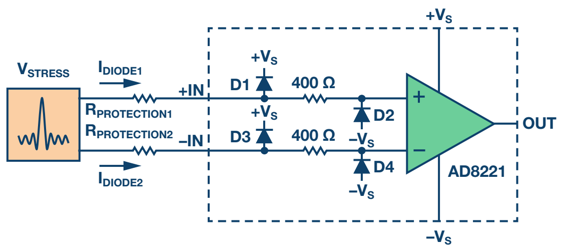


Figure 7. Using ESD cells as clamps.

An assumption used for Equation 1 is that $V_{STRESS} > V_{SUPPLY}$. If this is not the case, a more precise diode voltage should be measured and used for the calculation instead of the 0.7 V approximation.

An example calculation follows for protecting an amplifier using ±15 V supplies, from input stresses up to ±120 V, while limiting the input current to 1 mA. Using Equation 1, we can use these inputs to calculate the following.

$$I_{DIODE} = \frac{V_{STRESS} - (V_{SUPPLY} + 0.7\ \text{V})}{R_{PROTECTION}} \qquad (1)$$

$$1\ \text{mA} = \frac{120\ \text{V} - (15\ \text{V} + 0.7\ \text{V})}{R_{PROTECTION}} \qquad (2)$$

$$R_{PROTECTION} = 104{,}300\ \Omega \qquad (3)$$

Given these requirements, an $R_{PROTECTION} > 105$ kΩ would limit the diode current to <1 mA.

Understanding the Current Limitations

Maximum values for I_{DIODE} will vary from part to part, and also be dependent on the particular application scenarios in which the stress is applied. The maximum current will be different for a one-time event lasting milliseconds vs. if the current was constantly applied over the entire 20 or more year mission profile life of the application. Guidance on the particular values may be found in amplifier data sheets in the absolute max section or application notes and are usually in the range of 1 mA to 10 mA.

Failure Modes

The maximum current rating for a given protection scheme will ultimately be limited by two factors: the thermal implication of the power dissipated in the diode and the maximum current rating for the current path. The power dissipation should be kept below a threshold that maintains the operating temperature in a valid range and the current should be chosen to be within the specified maximum to avoid reliability issues due to electromigration.

Thermal Implications

When current flows into the ESD diodes, there will be a temperature increase due to the power dissipated in the diodes. Most amplifier data sheets specify a thermal resistance (usually specified as Θ_{JA}) that will indicate how junction temperatures will increase as a function of power dissipation. Considering the worst-case application temperature, along with the worst-case temperature increase due to power dissipation, will give an indication of the viability of a protection circuit.

Electromigration

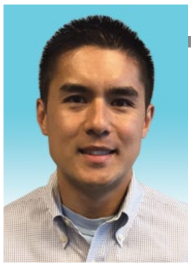
Even when the current does not cause thermal problems, the diode current could still create a reliability problem. There is a maximum lifetime current rating for any electrical signal path due to electromigration. The electromigration current limit for the diode current path is typically limited by thickness of the internal traces in series with the diodes. This information is not always published for amplifiers, but needs to be considered if the diodes are active for long portions of time, as opposed to transient events.

An example where electromigration can be a problem is when an amplifier is monitoring, and therefore connected to, a voltage rail that is independent of its own supply rail. When there are multiple power domains, there can be instances where power supply sequencing can cause voltages to temporarily exceed absolute maximum conditions. By considering the worst-case current path, the duration over life that this current could be active, and understanding the maximum allowable current for electromigration, reliability issues due to electromigration can be avoided.

Conclusion

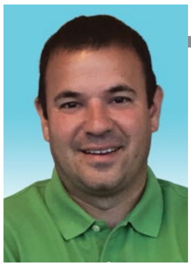
Understanding how an amplifier's internal ESD diodes are activated during electrical overstress events can enable simple improvements to the robustness of a design. Examining both the thermal and electromigration implications of a protection circuit can highlight potential problems and indicate where additional protection may be warranted. Considering the conditions outlined here enables designers to make smart choices and avoid potential robustness issues in the field.

Paul Blanchard



Paul Blanchard [paul.blanchard@analog.com] is an applications engineer at Analog Devices in the Instrumentation, Aerospace, and Defense business unit, located in Wilmington, MA. Paul started with ADI in 2002 in the Advanced Linear Products (ALP) Group covering instrumentation amplifiers and variable gain amplifiers. In 2009, as part of the Linear Products Group (LPG), he was primarily responsible for automotive radar, current sensing, and AMR related applications. Currently, as part of the Linear and Precision Technology (LPT) Group, he is working on precision input signal conditioning (PISC) signal chain technologies. Paul earned his bachelor's and master's degree in electrical engineering from Worcester Polytechnic Institute.

Brian Pelletier



Brian Pelletier [brian.pelletier@analog.com] is a product development engineer in the Linear Product Technologies division of Analog Devices. He joined ADI in 2003 after obtaining a bachelor's degree in electrical engineering from the University of Massachusetts. Brian specializes in precision amplifiers, including instrumentation amplifiers and current sense amplifiers.

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 3—Mode S Signals Decoding Algorithm Validation Using Hardware in the Loop

By Di Pu and Andrei Cozma

Introduction

After implementing any signal processing algorithm in MATLAB or Simulink, the next natural step is to verify the algorithm's functionality using real data acquired from the actual SDR hardware system that it is going to run on. As a first step, the verification of the algorithm is done using different sets of input data captured from the system. This helps validate the algorithm's functionality, but does not guarantee that the algorithm will perform as expected in environmental conditions other than the ones used to make the data captures, or what the behavior and performance will be for different settings of the analog front end and digital blocks of the SDR system. In order to verify all of these aspects, it is very beneficial if the algorithm can be run online to receive live data as input and to tune the settings of the SDR system for optimal performance. This part of the article series[1] discusses the software tools provided by Analog Devices to allow direct interaction between MATLAB and Simulink models with the FMCOMMSx SDR platforms and shows how these tools can be used to verify the ADS-B models presented in Part 2 of the article series.[2]

MATLAB and Simulink IIO System Object

Analog Devices provides a complete software infrastructure that enables MATLAB and Simulink models to interact in real time with FMCOMMSx SDR platforms that are connected to FPGA/SoC systems running Linux. This is possible due to an IIO System Object[3] that is designed to exchange data over TCP/IP with the hardware system in order to stream data to and from a target, control the settings of a target and monitor different target parameters such as the RSSI. Figure 1 presents the high level architecture of the software infrastructure and the data flow between the components in the system.

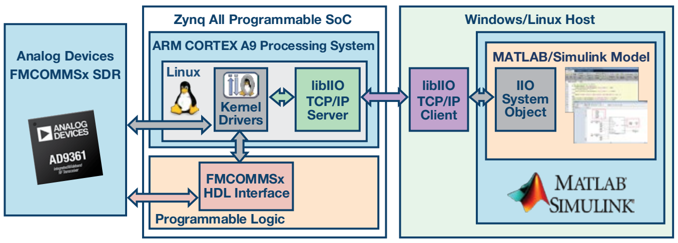


Figure 1. Software infrastructure block diagram.

The IIO System Object is based on the MathWorks System Objects specification[4] and exposes data and control interfaces through which the MATLAB/Simulink models communicate to IIO-based platforms. These interfaces are specified in a configuration file that links the System Object interface to IIO data channels or to IIO attributes. This makes the implementation of the IIO System Object generic, allowing it to work with any IIO platform just by modifying the configuration file. Some of the platforms for which configuration files and examples are available on the ADI GitHub repository[5] include the AD-FMCOMMS2-EBZ/AD-FMCOMMS3-EBZ/AD-FMCOMMS4-EBZ/AD-FMCOMMS5-EBZ SDR boards and the high speed data acquisition AD-FMCDAQ2-EBZ board. The communication between the IIO System Object and the target is accomplished through the libiio server/client infrastructure. The server runs on an embedded target under Linux and manages real-time data exchange between the target and both local and remote clients. The libiio library abstracts the low level details of the hardware and provides a simple yet complete programming interface that can be used for advanced projects with a variety of language bindings (C, C++, C#, Python).

The next sections of the article provide real life examples on how the IIO System Object can be used for validating the ADS-B MATLAB and Simulink models. An AD-FMCOMMS3-EBZ SDR platform[6] connected to a ZedBoard[7] running the Analog Devices Linux distribution were used as the SDR hardware system for verifying the operation of the ADS-B signals detection and decoding algorithm, as shown in Figure 2.

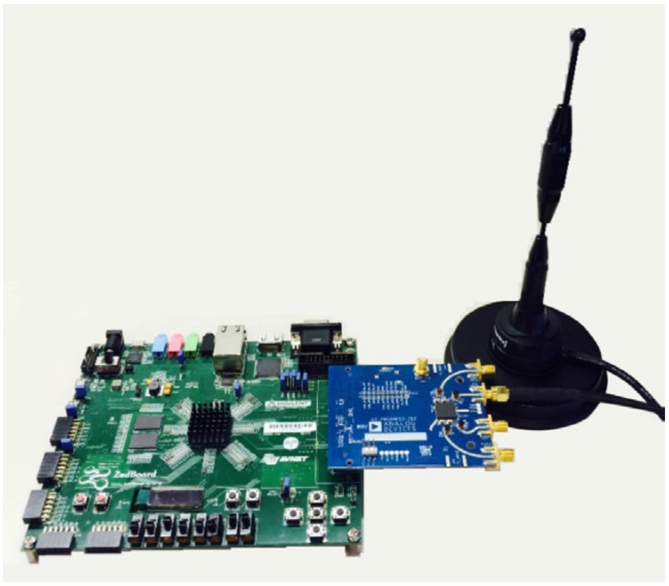


Figure 2. Hardware setup for ADS-B algorithm validation.

MATLAB ADS-B Algorithm Validation Using the IIO System Object

To validate the MATLAB ADS-B decoding algorithm operation with real-time data acquired from the AD-FMCOMMS3-EBZ SDR platform, a MATLAB script has been developed to perform the following operations:

- Calculate the earth zone according to user input
- Create and configure the IIO System Object
- Configure the AD-FMCOMMS3-EBZ analog front end and digital blocks through the IIO System Object
- Receive data frames from the SDR platform using the IIO System Object
- Detect and decode the ADS-B data
- Display the decoded ADS-B information

After an IIO System Object is constructed it must be configured with the IP address of the SDR system, the target device name and input/output channels sizes and numbers. Figure 3 presents an example on how to create and configure the MATLAB IIO System Object.

```
% System Object Configuration
s = iio_sys_obj_matlab; % MATLAB libiio Constructor
s.ip_address = ip;
s.dev_name = 'ad9361';
s.in_ch_no = 4;
s.out_ch_no = 4;
s.in_ch_size = n;
s.out_ch_size = n;

s = s.setupImpl();
```

Figure 3. MATLAB IIO System Object creation and configuration.

The IIO System Object is then used to set the attributes of AD9361 and to receive the ADS-B signals. The attributes of AD9361 is set up based upon the following considerations:

```
% Set the attributes of AD9361
if strcmp(source,'pre-captured')
    input_content(s.getInChannel('RX_LO_FREQ')) = 6e9;
elseif strcmp(source,'live')
    input_content(s.getInChannel('RX_LO_FREQ')) = 1.09e9;
else
    error('Please select a data source: pre-captured or live.');
end
input_content(s.getInChannel('RX_SAMPLING_FREQ')) = 12.5e6;
input_content(s.getInChannel('RX_RF_BANDWIDTH')) = 4e6;
input_content(s.getInChannel('RX1_GAIN_MODE')) = 'fast_attack';
input_content(s.getInChannel('RX1_GAIN')) = 0;
input_content(s.getInChannel('RX2_GAIN_MODE')) = 'fast_attack';
input_content(s.getInChannel('RX2_GAIN')) = 0;
input_content(s.getInChannel('TX_LO_FREQ')) = 6e9;
input_content(s.getInChannel('TX_SAMPLING_FREQ')) = 12.5e6;
input_content(s.getInChannel('TX_RF_BANDWIDTH')) = 4e6;
```

Figure 4. MATLAB libiio sets the attributes of AD9361.

The sampling rate is quite straightforward with the AD9361-based platforms. The transmit data rate normally equals the RX data rate, and ultimately depends on the baseband algorithm. In this example, since the decoding algorithm is designed to work with the sampling rate of 12.5 MSPS, the data rate of AD9361 is set accordingly. By doing this, the received samples can be applied directly to the decoding algorithm, without any additional decimation or interpolation operations.

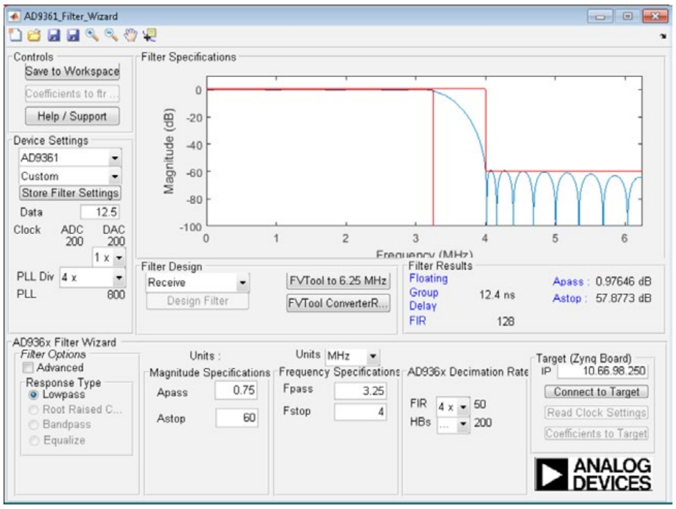
The RF bandwidth control sets the AD9361's RX analog baseband low-pass filter's bandwidth to provide antialiasing and out-of-band signal rejection. In order to successfully demodulate the received signals, the system must maximize the signal-to-noise ratio (SNR). In order to do this, the RF bandwidth needs to be set as narrow as possible while meeting flatness and the out-of-band rejection specification to minimize in-band noise and spurious signal levels. If the RF bandwidth is set wider than it needs to be, the ADC's linear dynamic range will be reduced due to the extra noise. Similarly, ADC's spurious-free dynamic range will be reduced due to the lower out-of-band signal rejection resulting in overall receiver dynamic range reduction. Therefore, setting the RF bandwidth at an optimal value is critical to receive desired in-band signals and reject out-of-band signals. By observing the spectrum of received signals, we find 4 MHz is a proper value for the RF bandwidth.

Besides setting up the analog filters of AD9361 via RF bandwidth attribute, we can also improve the decoding performance by enabling the digital FIR filters on AD9361 via the IIO System Object, as shown in Figure 5. According to the spectrum characteristics of the ADS-B signal, we design an FIR filter with data rate of 12.5 MSPS, pass band frequency of 3.25 MHz and stop band frequency of 4 MHz. In this way, we can further focus on the bandwidth of interest.

```
s.writeFirData('adsb.ftr');
```

Figure 5. Enable the proper FIR filter on AD9361 via libiio.

Adsb.ftr is a file containing the coefficients of an FIR filter designed using the Analog Devices AD9361 Filter Wizard MATLAB application.[8] This tool provides not only a general-purpose low-pass filter design, but it also provides magnitude and phase equalization for other stages in the signal path.



Device Settings: AD9361, Custom
Data 12.5
Clock ADC 200 DAC 200, 1 x
PLL Div 4 x
PLL 800
Filter Design: Receive
Filter Results: Floating Group Delay 12.4 ns; FIR 128; Apass : 0.97646 dB; Astop : 57.8773 dB
Filter Options: Advanced
Response Type: [✓ Lowpass] Root Raised C... Bandpass Equalize
Units :
Magnitude Specifications: Apass 0.75; Astop 60
Units MHz
Frequency Specifications: Fpass 3.25; Fstop 4
AD936x Decimation Rate: FIR 4 x 50; HBs ... 200
Target (Zynq Board): IP 10.66.98.250

Figure 6. FIR filter designed for ADS-B signals using the MATLAB AD9361 Filter Wizard.

The versatile and highly configurable AD9361 transceiver has several gain control modes that enable its use in a variety of applications. The Gain Mode parameter of the IIO System Object selects one of the available modes: manual, slow_attack, hybrid, and fast_attack. The most frequently used modes are manual, slow_attack, and fast_attack. Manual gain control mode allows the baseband processor (BBP) to control the gain. Slow_attack mode is intended for slowly changing signals, while fast_attack mode is intended for waveforms that "burst" on and off. Gain mode highly depends on the strength of received signals. If the signal is too strong or too weak, it is suggested to use manual mode or slow_attack. Otherwise, fast_attack is a good option. In the case of ADS-B the fast_attack gain mode provides the best results due to the bursty nature of these signals. Fast_attack mode is a requirement for this waveform since there is preamble, and the AGC needs to react fast enough so that the first bit is captured. There is a difference between attack time—the time it takes to ramp down gain—and decay time—how long it takes to increase gain—in the absence of a signal. The goal is to quickly turn down the gain, so that a valid "1" can be seen on the first bit, but not increase the gain between bit times.

In the end, depending on how you set up the TX_LO_FREQ and RX_LO_FREQ, there are two ways of using this model: using precaptured data (RF loopback) and using live data off the air.

Precaptured Data

In this case, we are transmitting and receiving some precaptured ADS-B signals using AD-FMCOMMS3-EBZ . These signals are saved in a variable called "newModeS."

```
input_content{1} = (2^13).*newModeS./sqrt(2);
input_content{2} = (2^13).*newModeS./sqrt(2);
input_content{3} = (2^13).*newModeS./sqrt(2);
input_content{4} = (2^13).*newModeS./sqrt(2);
```

Figure 7. Define input using precaptured ADS-B signals.

The requirement for this case is to make TX_LO_FREQ = RX_LO_FREQ, and it can be any LO frequency value that AD-FMCOMMS3-EBZ supports. Due to the nature of precaptured data, there is plenty of ADS-B valid data in there, so it is a good way to verify whether the hardware setup is appropriate.

Live Data

In this case, we are receiving the real-time ADS-B signals over the air, instead of the signals transmitted by AD-FMCOMMS3-EBZ. According to ADS-B specification, it is transmitted at the center frequency of 1090 MHz, so the requirements for this case are:

- RX_LO_FREQ=1090 MHz, TX_LO_FREQ far away from 1090 MHz in order to avoid interference.
- Use a proper antenna on the receiver side, which is capable of covering the 1090 MHz band, such as an ADS-B Double 1/2 Wave Mobile Antenna[9]; using a poorly tuned or poorly made antenna will result in a lack of range for your air radar.

With everything set up properly, in order to run the MATLAB model, simply use the following command:

```
[rssi1,rssi2]=ad9361_ModeS('ip','data source',channel);
```

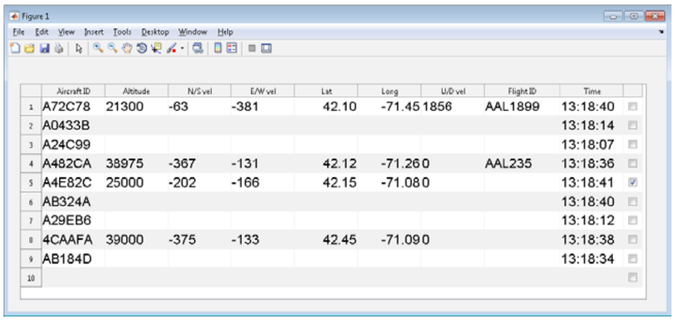
where ***ip*** is the IP address of the FPGA board, and ***data source*** specifies the data source of the received signal. Currently, this model supports data sources of "precaptured" and "live."

Channel specifies whether signals are received using Channel 1 or Channel 2 of the AD-FMCOMMS3-EBZ.

For example, the following command receives the precaptured data on Channel 2:

```
[rssi1,rssi2]=ad9361_ModeS('192.168.10.2','pre-captured',2);
```

At the end of the simulation, you will get the RSSI values on both channels, as well as the result tables shown below:



	Aircraft ID	Altitude	N/S vel	E/W vel	Lat	Long	U/D vel	Flight ID	Time	
1	A72C78	21300	-63	-381	42.10	-71.45	1856	AAL1899	13:18:40	☐
2	A0433B								13:18:14	☐
3	A24C99								13:18:07	☐
4	A482CA	38975	-367	-131	42.12	-71.26	0	AAL235	13:18:36	☐
5	A4E82C	25000	-202	-166	42.15	-71.08	0		13:18:41	☑
6	AB324A								13:18:40	☐
7	A29EB6								13:18:12	☐
8	4CAAFA	39000	-375	-133	42.45	-71.09	0		13:18:38	☐
9	AB184D								13:18:34	☐
10										☐

Figure 8. Result table shown at the end of the simulation.

This result table shows the information of aircrafts appearing during the simulation. With a proper antenna, this model is able to capture and decode the aircraft signals in an 80 mile range with AD-FMCOMMS3-EBZ. Since there are two types of Mode S messages (56 µs or 112 µs), some messages contain more information than the other.

When trying out this model with the real-world ADS-B signals, the signal strength is very important for successful decoding, so make sure to put the antenna in a good line of sight location with the aircraft. The received signal strength can be seen by looking at the RSSI values on both channels. For example, if receiving the signals on Channel 2, the RSSI of Channel 2 should be significantly higher than that of Channel 1. You can tell whether there is any useful data by looking at the spectrum analyzer.

RF Signal Quality

For any RF signal, there needs to be a quality metric. For example, for signals like QPSK, we have error vector magnitude (EVM). For ADS-B signals, it isn't enough to look at the output of a slicer for correct messages, as shown in Figure 8. We need a metric to define the quality of ADS-B/pulse position modulation, so that we can tell whether one setting is better than the other.

In ModeS_BitDecode4.m function, there is a variable *diffVals*, which can be used as such a metric. This variable is a 112 × 1 vector. It shows for each decoded bit in one Mode S message, how far is it away from the threshold. In other words, how much margin each decoded bit has with respect to a correct decision. It is obvious the more margin a bit has, the more confident the decoded result is. On the other hand, if the margin is low, it means the decision is in the border area, so it is very likely that the decoded bit is wrong.

The following two figures compare the *diffVals* values obtained from the ADS-B receivers with and without the FIR filter. By looking at the y-axis, we find with the FIR filter, *diffVals* is larger regardless of whether it is at the highest point, lowest point, or average. However, when there is no FIR filter, the *diffVals* of several bits are very close to 0, which means the decoded results could be wrong. Therefore, we are able to verify that using a proper FIR filter improves the signal quality for decoding.

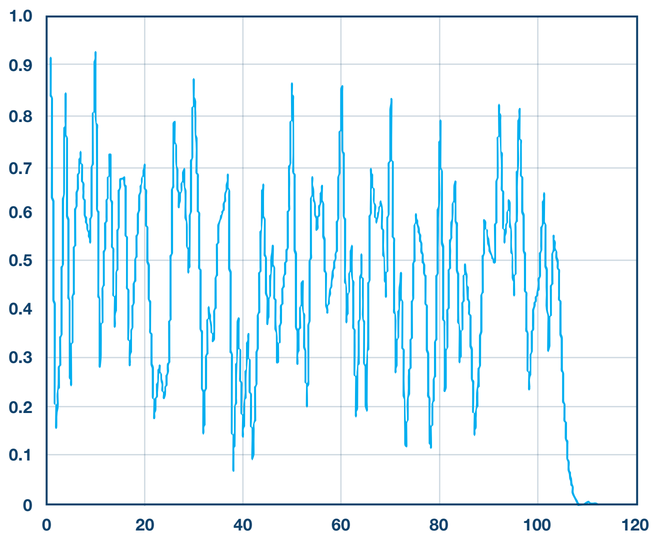


Figure 9. diffVals values obtained from the ADS-B receiver with an FIR filter.

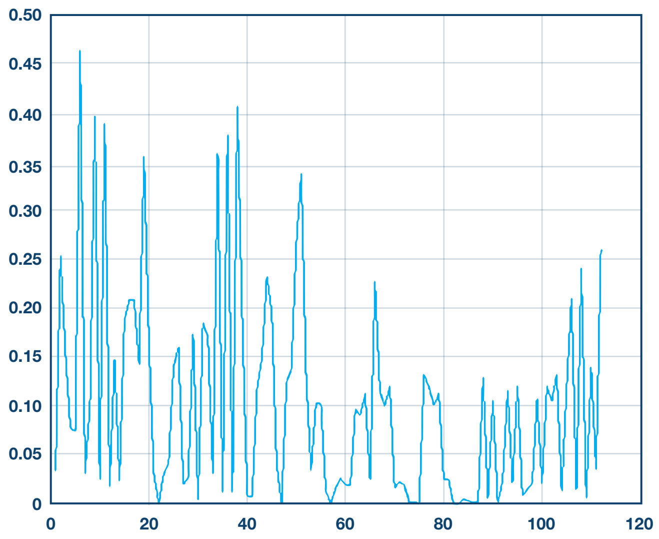


Figure 10. diffVals values obtained from the ADS-B receiver without FIR filter.

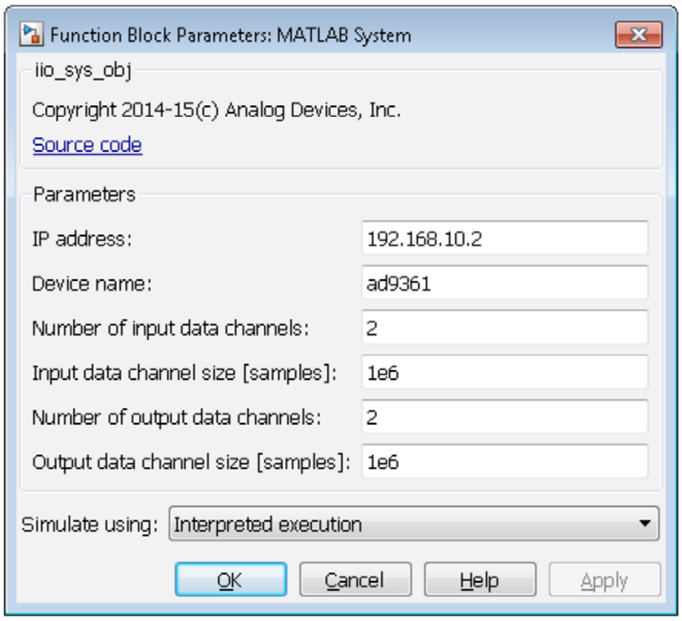
The MATLAB ADS-B algorithm using the IIO System Object can be downloaded from the ADI GitHub repository.[10]

Simulink ADS-B Algorithm Validation Using the IIO System Object

The Simulink model is based upon the model introduced in Part 2 of the article series.[2] The detector and decoding piece comes directly from that model, and we add the Simulink IIO System Object to conduct the signal reception and hardware in the loop simulation.

The original model works with sample time = 1 and frame size = 1. However, the Simulink IIO System Object works in a buffer mode—it accumulates a number of samples and then processes them. In order to make the original model work with the System Object, we added two blocks between them: unbuffer to make frame size = 1 and rate transition to make sample time = 1. By doing this, we can keep the original model intact.

The Simulink IIO System Object is set up as following. Similar to the MATLAB one, it creates a System Object, and then defines the IP address, device name, and input/output channels number and sizes related to this System Object.



Function Block Parameters: MATLAB System

iio_sys_obj

Copyright 2014-15(c) Analog Devices, Inc.

Source code

Parameters

IP address:	192.168.10.2
Device name:	ad9361
Number of input data channels:	2
Input data channel size [samples]:	1e6
Number of output data channels:	2
Output data channel size [samples]:	1e6

Simulate using: Interpreted execution

Figure 12. Simulink IIO System Object.

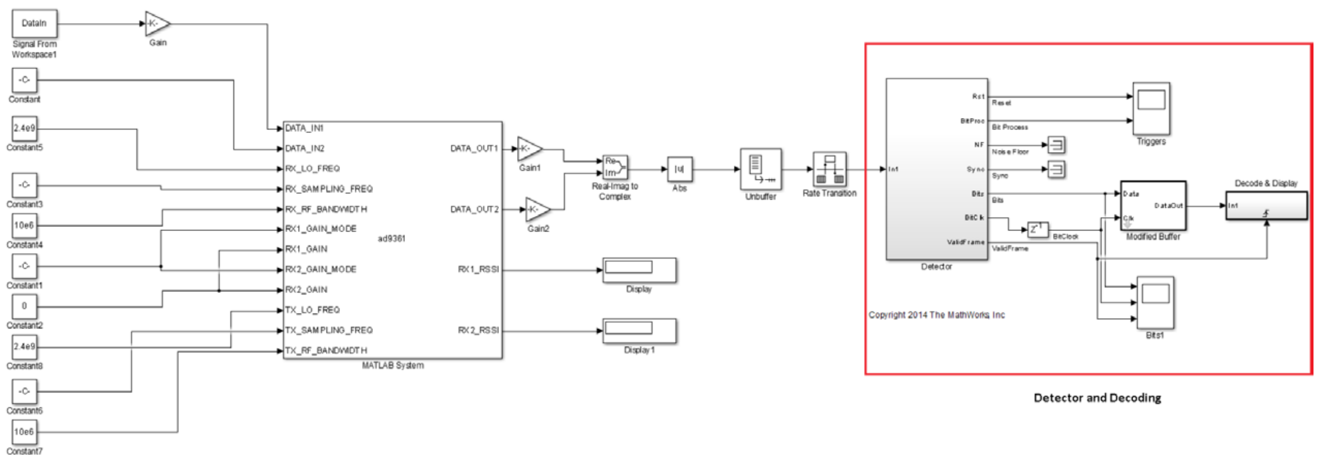


Figure 11. Simulink model to capture and decode ADS-B signals.

The input and output ports of this Simulink block corresponding to an IIO System Object are defined through the properties dialog of the object's block as well as through a configuration file that is specific to the targeted ADI SDR platform. The input and output ports are categorized as data and control ports. The data ports are used to receive/transmit buffers of continuous data from/to the target system in a frame-based processing mode, while the control ports are used to configure and monitor different target system parameters. The number and size of the data ports are configured from the block's configuration dialog while the control ports are defined in the configuration file. The attributes of AD9361 are set up according to the same factors as introduced in MATLAB model. All the theories and methods employed in the MATLAB model can be applied here.

Depending on how you set up the TX_LO_FREQ and RX_LO_FREQ, this Simulink model can be run in two modes: using precaptured data "DataIn" and using live data. Taking the precaptured data, for example, at the end of the simulation, we can see the following results in command window.

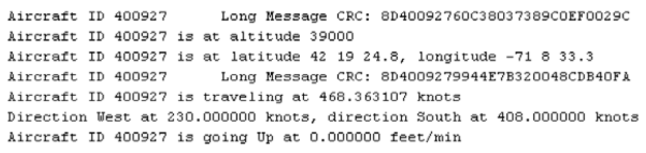

```
Aircraft ID 400927      Long Message CRC: 8D40092760C38037389C0EF0029C
Aircraft ID 400927 is at altitude 39000
Aircraft ID 400927 is at latitude 42 19 24.8, longitude -71 8 33.3
Aircraft ID 400927      Long Message CRC: 8D4009279944E7B320048CDB40FA
Aircraft ID 400927 is traveling at 468.363107 knots
Direction West at 230.000000 knots, direction South at 408.000000 knots
Aircraft ID 400927 is going Up at 0.000000 feet/min
```

Figure 13. Results in command window at the end of simulation using precaptured data.

Instead of the result table shown in the MATLAB model, the results here are displayed in the text format.

The Simulink ADS-B model using the IIO System Object can be downloaded from the ADI GitHub repository.[11]

Conclusion

This article talked about hardware in the loop simulation using the libiio infrastructure provided by Analog Devices. Using this infrastructure, the MATLAB and Simulink algorithms for ADS-B signals detection and decoding can be validated with the real-world signals and real hardware. Since the attribute setting is very application and waveform dependent, what works for one waveform will not work for a different one. This is a critical step to ensure that the analog front end and the digital blocks of the SDR system are properly tuned for the algorithm and waveform of interest and that the algorithm is robust enough and works as expected with real life data acquired in varying environmental conditions. Having a verified algorithm, it is now time to move to the next step, which consists of translating the algorithm to HDL and C code using the automatic code generation tools from MathWorks and integrating this code into the programmable logic and software of the actual SDR system. The next part of the article series will show how to generate code and deploy it in the production hardware and will talk about the results obtained by operating the platform with real-world ADS-B signals at an airport. This will complete the steps required to take an SDR system from prototyping all the way to production.

References

1 Andrei Cozma, Di Pu, and Tom Hill. "Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio—Part 1." *Analog Dialogue*, Volume 49, Number 3, 2015.

2 Mike Donovan, Andrei Cozma, and Di Pu. "Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio—Part 2." *Analog Dialogue*, Volume 49, Number 4, 2015.

3 Analog Devices. "IIO System Object."

4 MathWorks. "What Are System Objects?"

5 Analog Devices, "Mathworks_tools." GitHub repository.

6 Analog Devices. AD-FMCOMMS3-EBZ User Guide.

7 ZedBoard.

8 Analog Devices. MATLAB AD9361 Filter Design Wizard.

9 ADS-B Double 1/2 Wave Mobile Antenna.

10 MATLAB ADS-B Algorithm Using The IIO System Object Source Code.

11 Simulink ADS-B Model Using The IIO System Object Source Code.

Acknowledgements

The authors would like to thank Mike Donovan from MathWorks, who contributed to the development of the MATLAB and Simulink ADS-B signal detection and decoding algorithms used in this article.

Di Pu [di.pu@analog.com] is a system modeling applications engineer for ADI, supporting the design and development of software-defined radio platforms and systems. She has been working closely with MathWorks to solve mutual end customer challenges. Prior to joining ADI, she received her B.S. degree from Najing University of Science and Technology (NJUST), Nanjing, China, in 2007 and her M.S. and Ph.D. degrees from Worcester Polytechnic Institute (WPI), Worcester, MA, U.S.A., in 2009 and 2013—all in electrical engineering. She is a winner of the 2013 Sigma Xi Research Award for Doctoral Dissertation at WPI.

Di Pu

Andrei Cozma [andrei.cozma@analog.com] is an engineering manager for ADI, supporting the design and development of system level reference designs. He holds a B.S. degree in industrial automation and informatics and a Ph.D. in electronics and telecommunications. He has been involved in the design and development of projects from different industry fields such as motor control, industrial automation, software-defined radio, and telecommunications.

Andrei Cozma

Also by this Author:

FPGA-Based Systems Increase Motor-Control Performance

Volume 49, Number 1

Versatile, Precision Single-Ended-to-Differential Signal Conversion Circuit with Adjustable Output Common Mode Boosts System Dynamic Range

By Darwin Tolentino and Sandro Herrera

Differential signaling finds useful application in circuits where a large signal-to-noise ratio, high immunity to noise, and lower second harmonic distortion are desired, such as in driving high performance ADCs and high fidelity audio signal conditioning. A related previous article in *Analog Dialogue*, "Versatile, Low Power, Precision Single-Ended-to-Differential Converter,"[1] offers a greatly improved single-ended-to-differential circuit that has very high input impedance, 2 nA maximum input bias current, 60 μV maximum offset (RTI), and 0.7 μV/°C maximum offset drift. The improved performance is achieved by cascading an OP1177 in the feedback loop with the AD8476 that has a differential gain of 1.

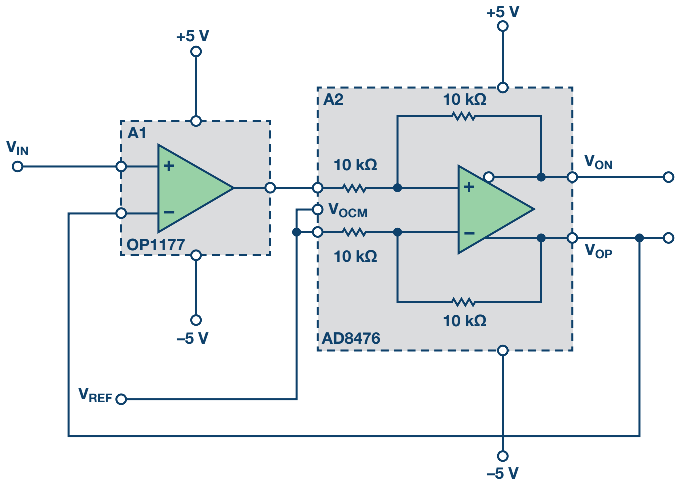


Figure 1. Improved single-ended-to-differential converter.

It is desirable, however, in many applications to have a greater output dynamic range, such as in signal conditioning of sensor outputs—for example, temperature and pressure. Being able to also adjust the common mode makes the circuit very convenient in interfacing to many ADCs where the reference determines full-scale range.

Configuring the differential amplifier inside the loop to a gain greater than 1 increases the output dynamic range of the circuit (Figure 2). The output is given by the following equation:

$$V_{OUT,\ DIFF} = V_{OP} - V_{ON} = 2\,(V_{IN}\,(1 + \frac{R_F}{R_G}) - V_{REF})$$

When R_G is left open the circuit has an overall gain of 2. The output of A1, OP1177, is given by the following:

$$V_{OUT,\ OP1177} = \frac{V_{OUT,\ DIFF}}{G_{DIFF,\ A2}} + V_{REF}$$

Notice that the V_{REF} is always added to the output of the OP1177 limiting its output headroom. In most applications the V_{REF} (the output common mode) is set at the center of the supplies for maximum output dynamic range. A differential amplifier inside the loop configured at a gain greater than 1, such as the ADA4940 in Figure 2 (gain of 2), reduces the output voltage of A1 by a factor of A2's differential gain and helps avoid saturating the output of A1. Because the OP1177 has a typical output swing of 4.1 V at ±5 V supplies, the differential output voltage swing of the circuit in Figure 2 is about ±8 V at V_{REF} set at 0. Configuring A2 to a gain of 3 further improves the output dynamic range and achieves the maximum output swing of the circuit. Another amplifier, the ADA4950 with available gains of 1, 2, and 3, may also be suitable for A2.

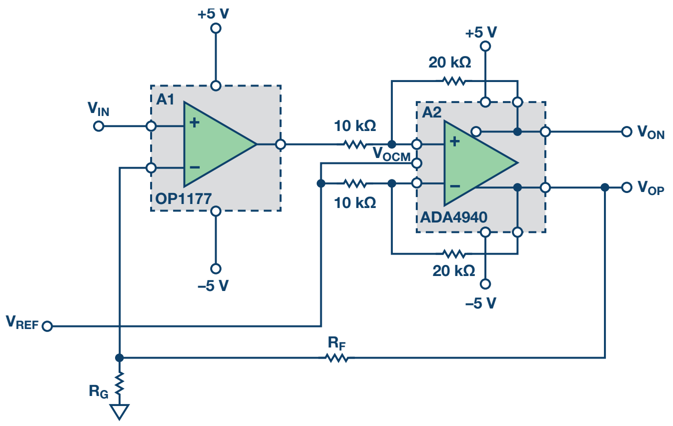
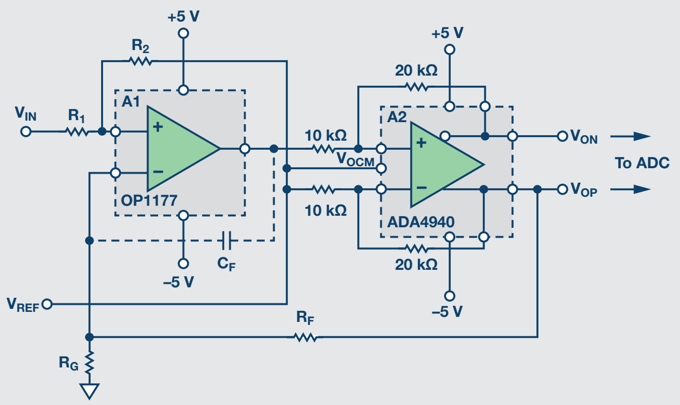
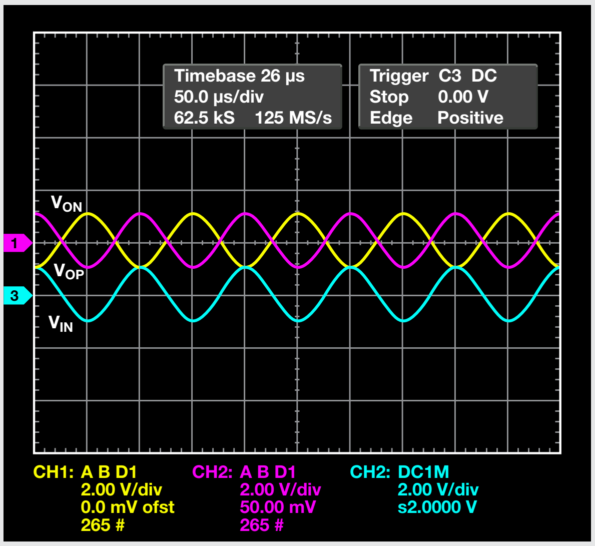


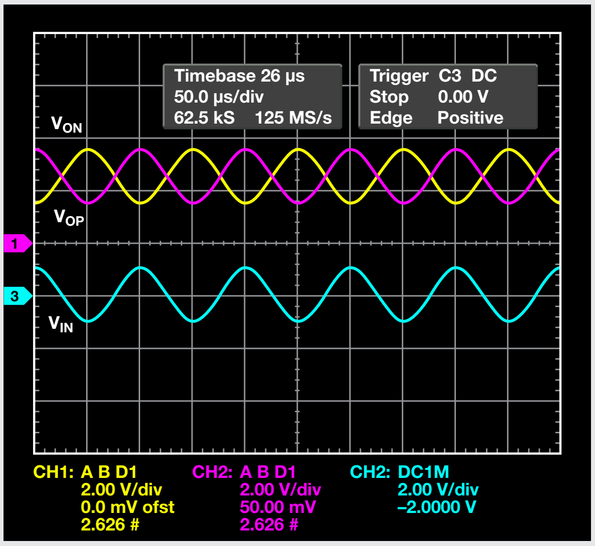
Figure 2. Single-ended-to-differential converter with improved dynamic range.



(a) Improved single-ended-to-differential converter with adjustable common mode.



(b) Input and output plots, V_{OP} in red, V_{ON} in yellow, and input in blue. Common mode is at 0 V.



(c) Input and output plots, V_{OP} in red, V_{ON} in yellow, and input in blue. Common mode is at 2.5 V.

Figure 3.

Adjustable Output Common Mode

The circuit can be modified to make the output common mode adjustable and independent of the common mode of the input signal. This adds great flexibility and convenience for single-supply applications where the input is referred to ground and is needed to be converted to a differential signal with an elevated common mode for ADC interfacing.

This can be accomplished by adding two resistors at the input R_1 and R_2, where R_2 is tied to V_{OCM}. If desired, using a dual version of the input amplifier A1, the OP2177, allows for the second amplifier to be used as a buffer to the input for very low input bias current.

In the circuit in Figure 1, the input is referred to V_{REF}. Referring to the circuit in Figure 3, the input is referred to ground taken directly and converted to differential output. The V_{OCM} can now be adjusted to shift the common-mode output while the input remains referenced to ground. The V_{OCM} can be tied to half of the reference or the midscale of the converter. The V_{OCM} basically acts as another input along with V_{IN}. The values of the resistors should be chosen such that $\frac{R_1}{R_G} = \frac{R_2}{R_F}$.

By superposition, when V_{IN} is 0, the output is forced at the same value as V_{OCM}. And since V_{OCM} is the value that sets the output common mode, the differential output is zero. If $R_1 = R_G$ and $R_2 = R_F$, the output voltages are given by:

$$V_{OP} = \left(\frac{R_F}{R_G}\right) V_{IN} + V_{OCM}$$

$$V_{ON} = -\left(\frac{R_F}{R_G}\right) V_{IN} + V_{OCM}$$

$$V_{OUT,\ DIFF} = 2\left(\frac{R_F}{R_G}\right) V_{IN}$$

Bandwidth and Stability

The two amplifiers form a composite differential output op amp in a servo-loop configuration. The OP1177/OP2177's open-loop gain and the differential gain of the ADA4940 combine for the total open-loop gain of the circuit that defines the overall bandwidth of the circuit. Their poles combine for additional phase shift in the loop. A higher gain for A2 reduces its bandwidth and may affect the stability of the overall circuit. The circuit designer must check the overall circuit frequency response and assess the need for compensation. A rule of thumb is that the combined open-loop gain over frequency must cross the unity gain at –20 dB/decade roll-off in order to ensure the stability of the feedback system. This is particularly more important in applications with the minimum gain (gain of 2) where the loop gain is at maximum and has the worst phase margin. A higher overall gain also improves stability by decreasing the bandwidth and increasing the phase margin of the feedback loop. Because the loop gain is decreased, it crosses the unity gain at a lower frequency. The loop gain is given by:

$$Loop\ Gain = (A_{OL,\ 1st\ Amp})(A_{Diff,\ 2nd\ Amp})\beta$$

$$\beta = \frac{1}{2}\left(\frac{R_G}{R_G + R_F}\right)$$

The feedback factor β has $\frac{1}{2}$ in the term because the output is differential and the feedback is taken only from one of the differential outputs. The ADA4940 has a bandwidth of 50 MHz at a gain of 2, while the OP1177 has a unity-gain bandwidth of about 4 MHz. The circuit in Figure 3 is stable with a bandwidth of about 1 MHz, limited by the OP1177 and the closed-loop gain. As pointed out in the previous article, when the stability condition cannot be met using different amplifiers, a bandwidth limiting capacitor can be used as shown in Figure 3(a). The capacitor forms an integrator with R_F inside the feedback loop and limits the bandwidth of the overall circuit to

$$\frac{1}{2} \times \frac{1}{2\pi R_F C_F}$$

The capacitor and feedback resistor can be chosen such that the overall bandwidth is limited by the equation above.

Reference:

[1] Herrera, Sandro and Moshe Gerstenhaber. "Versatile, Low Power, Precision Single-Ended-to-Differential Converter." *Analog Dialogue*, Volume 46, Number 4.

Darwin Tolentino [darwin.tolentino@analog.com] is a staff test development engineer in the Linear Precisions Technology Group at ADI Philippines. He has worked in the Product and Test Engineering Group and has developed test solutions for amplifiers and linear products, including converters. He joined ADI in 2000 and has 17 years of experience in the semiconductor industry. His interests include history and designing analog circuits.

Darwin Tolentino

Also by this Author:

Simple Circuit Provides Adjustable CAN-Level Differential-Output Signal

Volume 46, Number 2

Sandro Herrera [sandro.herrera@analog.com] is a circuit design engineer in the Integrated Amplifier Products (IAP) Group in Wilmington, MA. His design work currently focuses on fully differential amplifiers with either fixed, variable, or programmable gains. Sandro holds B.S.E.E. and M.S.E.E. degrees from the Massachusetts Institute of Technology. He joined Analog Devices in August 2005.

Sandro Herrera

Also by this Author:

Versatile, Low-Power, Precision Single-Ended-to-Differential Converter

Volume 46, Number 4

Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio

Part 4—Rapid Prototyping Using the Zynq SDR Kit and Simulink Code Generation Workflow

By Mike Donovan, Andrei Cozma, and Di Pu

Introduction

The previous parts of this article series introduced the Zynq SDR rapid prototyping platform,[1] presented the steps of using MATLAB and Simulink to develop an algorithm that can successfully process and decode ADS-B transmissions,[2] and showed how to verify the algorithm both in simulation and with live data acquired from the SDR platform.[3] The ultimate goal of all stages is to create a verified model that can be translated into C and HDL code and is ready to be integrated in the SDR platform's software and hardware infrastructure.

The Simulink model discussed in Part 2 of the series ("Mode S Detection and Decoding Using MATLAB and Simulink")[2] is a simulation model with enough hardware specific fidelity to verify that the design will successfully decode ADS-B messages. Using that model as a starting point, the final steps required to produce a working receiver design that runs on the Zynq SDR Rapid Prototyping Platform will be discussed. As in the previous articles in this series, the skills needed to develop this working design include: proficiency in MATLAB and Simulink, knowledge of the Zynq radio hardware, and software/hardware integration skills.

The steps to follow in this article include:

- Partition the Simulink model into functions that will target the FPGA fabric and the ARM® processing system on the Zynq SoC.
- Introduce design changes to the Simulink model to improve the performance of the generated HDL code.
- Generate the source HDL and C code for the ADS-B receiver algorithm.
- Integrate the generated source code in the Zynq radio platform design.
- Test the embedded design on the target hardware with live aircraft signals.

At the end of this process, a fully verified SDR system will be produced, running C and HDL code automatically generated from a Simulink ADS-B model and receiving and decoding live commercial aircraft signals in real time.

Partitioning a Model into Hardware and Software Components

The first step in the process of generating the implementation code is to partition the design into the functionality that will run on the programmable logic and the ARM processing system of the Zynq SoC.

Partitioning usually begins by identifying the processing requirements of the different components of the design and the required execution rates and times. Components (such as data modulation/demodulation algorithms) that are computationally intensive and need to run in real time at the sample rate are best suited to be implemented in the programmable logic. Less intensive processing tasks (such as data decoding and rendering, and system monitoring and diagnosis), are better suited for software implementation. Some other aspects to consider are: the data types and complexity of the operations and the precision of the input and output data. All the operations that target the programmable logic work on fixed-point, integer, or Boolean data types. In the case of more complex operations such as trigonometric functions or square roots, approximations are used to implement them efficiently using the available hardware resources. All these constraints result in precision loss that can adversely affect system functionality if not properly assessed and implemented. However, the components that target the processing system can work on floating-point numbers and implement operations of any complexity with the highest degree of fidelity, but usually at the expense of slower execution speed.

Using those constraints as a guideline, the partitioning of the ADS-B decoding algorithm is fairly obvious. The functionality in the Detector block in the ModeS_Simulink_Decode.slx model, which includes the front-end processing of the I/Q samples all the way through to the checksum computation, is well suited for implementation on the programmable logic of the Zynq SoC (Figure 1). The decoding of the message bits, which is implemented in the Modified Buffer and Decode and Display blocks, is easily implemented in the processing system.

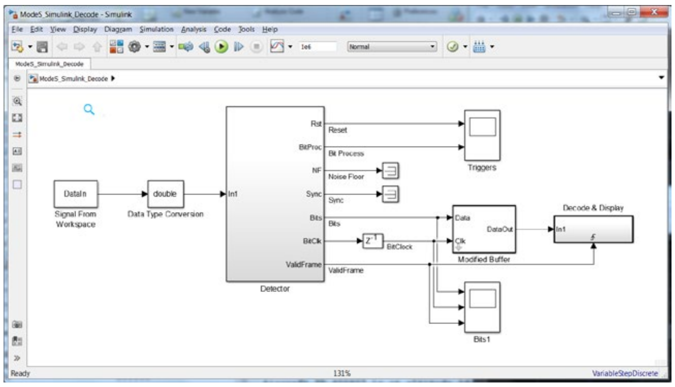


Figure 1. ModeS_Simulink_Decode.slx: FPGA and ARM processor partition.

Readers interested in following along with the Simulink model can find the files on the Analog Devices GitHub repository.[4]

Generating HDL Code from a Simulink Model

The Detector block in the Mode S Decoder model (Figure 2) is comprised of several subsystems: CalcSyncCorr, CalcNF, SyncAndControl, BitProcess, CalcCRC, and FameDetect. HDL Coder from MathWorks[5] is used to produce the source HDL code for this design.

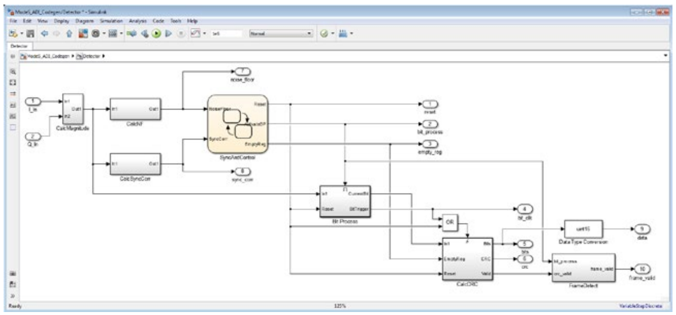


Figure 2. Detector block used for HDL code generation.

A Simulink model must satisfy several conditions to successfully generate HDL code using HDL Coder. A few of the most significant requirements are:

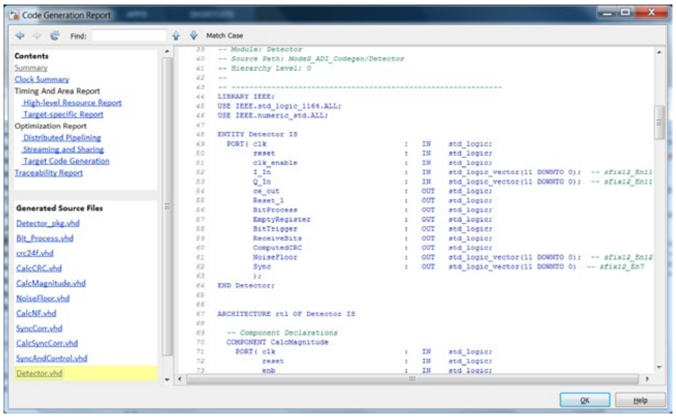
- Use blocks that support HDL code generation. HDL Coder supports code generation for approximately 200 Simulink blocks.[6] In the detector design, all the blocks, including the Stateflow diagram and the Digital Filter blocks, support HDL code generation.
- Use fixed-point data types. In the detector design, the signals use 12-bit, 24-bit, and Boolean data types. The 12-bit data type matches the bit width of the analog-to-digital converters on the Analog Devices AD9361 transceiver.
- Use scalar or vector signals. Vector signals can be used for multichannel signals or resource sharing.
- Avoid algebraic loops in the model. The HDL Coder software does not support HDL code generation for models in which algebraic loop conditions exist.

The ModeS_Simulink_Decode.slx model did not satisfy all these conditions, so the part of the CalcCRC block that compares the received bits to the computed checksum was moved outside the Detector block and ultimately implemented in C. The resulting model, ModeS_ADI_CodeGen.slx, was used to generate the HDL code. In contrast to a manual coding process, it only takes a couple minutes to generate several thousand lines of HDL code. The source code produced by HDL Coder is a bit true, cycle accurate version of the Simulink model. This is one of the major productivity gains in using model-based design; the generated code is an accurate translation of the Simulink model.

In addition, the code is designed to be readable and traceable so engineers can easily map the generated code to their design model. This is achieved in several ways (Figure 3):

- The hierarchy of the model is preserved in the HDL code files that get generated. In this example, the top level block is named Detector.vhd, and the subsystems at the next level of hierarchy are named CalcNF.vhd, Bit_Process.vhd, and so on.
- The block names, port names, signal names, data types, and complexity used in the model are preserved in the generated code.

Links between the model and source code allow a designer to click on a block in the Simulink model and automatically navigate to the generated HDL code. Similarly, there are hyperlinks in the generated code that will open the Simulink model and highlight the block associated with that segment of code.



```
-- Module: Detector
-- Source Path: ModeS_ADI_Codegen/Detector
-- Hierarchy Level: 0
--
-- -------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Detector IS
  PORT( clk                           :   IN    std_logic;
        reset                         :   IN    std_logic;
        clk_enable                    :   IN    std_logic;
        I_In                          :   IN    std_logic_vector(11 DOWNTO 0);  -- sfix12_En11
        Q_In                          :   IN    std_logic_vector(11 DOWNTO 0);  -- sfix12_En11
        ce_out                        :   OUT   std_logic;
        Reset_1                       :   OUT   std_logic;
        BitProcess                    :   OUT   std_logic;
        EmptyRegister                 :   OUT   std_logic;
        BitTrigger                    :   OUT   std_logic;
        ReceiveBits                   :   OUT   std_logic;
        ComputedCRC                   :   OUT   std_logic;
        NoiseFloor                    :   OUT   std_logic_vector(11 DOWNTO 0);  -- sfix12_En12
        Sync                          :   OUT   std_logic_vector(11 DOWNTO 0)  -- sfix12_En7
        );
END Detector;


ARCHITECTURE rtl OF Detector IS

  -- Component Declarations
  COMPONENT CalcMagnitude
    PORT( clk                         :   IN    std_logic;
          reset                       :   IN    std_logic;
          enb                         :   IN    std_logic;
```

Figure 3. Source HDL code for ModeS_ADI_CodeGen.slx.

Optimizing the ADS-B Model to Produce HDL Code with a Higher Clock Speed

Although the ModeS_ADI_CodeGen.slx model successfully generates HDL code, it is rare that a designer will not want to improve the initial results. Designers typically need to meet speed and area constraints, which usually involves optimizing the initial Simulink model to achieve the desired results. A major advantage of Simulink and code generation is that the designer can make those optimizations in the model, run a simulation to ensure the changes do not break the algorithm, and then re-generate the HDL code. This is usually much simpler and less error prone than making changes in the HDL source code and potentially breaking the algorithm.

In the case of this design, the HDL code generated by the model easily fit on the available FPGA fabric, but ran at a relatively low clock rate. This is common in many initial designs. A built-in analysis tool in HDL Coder shows that the critical path in the model extended from the I/Q sample input to the first register in the CalcCRC subsystem. Inserting pipeline registers in the design is one common method to increase the clock speed (Figure 4). Pipelining shortens the path between signal operations at the expense of adding delay to the overall processing. This trade-off is usually acceptable since a slight delay is typically a small price to pay for higher clock rates.

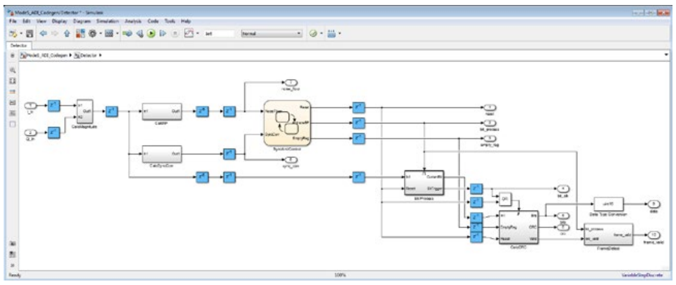
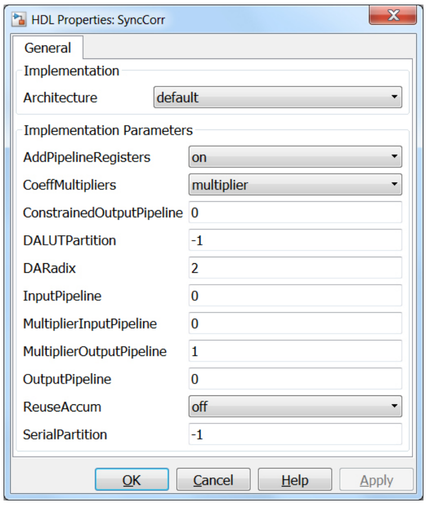


Figure 4. Pipeline registers inserted into detector design.

The pipeline registers in between the subsystems help improve the clock rate of the design, but better clock rates can be achieved by making favorable architecture choices for the Digital Filter blocks. Many of the Simulink blocks have architecture choices that enable a designer to optimize the design for speed or area. In the case of the digital filters used for the calculation of the noise floor and the preamble correlation (Figure 5), pipelining the output multipliers can shorten the critical path within the digital filter and improve the design clock rate.



Implementation

Architecture	default

Implementation Parameters

AddPipelineRegisters	on
CoeffMultipliers	multiplier
ConstrainedOutputPipeline	0
DALUTPartition	-1
DARadix	2
InputPipeline	0
MultiplierInputPipeline	0
MultiplierOutputPipeline	1
OutputPipeline	0
ReuseAccum	off
SerialPartition	-1

Figure 5. HDL block choices for the Digital Filter block.

After making these two simple pipeline changes, the clock rate of the generated HDL code exceeded 140 MHz. This is a useful lesson for engineers using code generation tools: applying a little knowledge of hardware design principles to the code generation models can have a significant impact on the results of the generated code. Further optimization of this design was possible, but deemed unnecessary, as the HDL code easily met the relatively simple timing and resource objectives for this design.

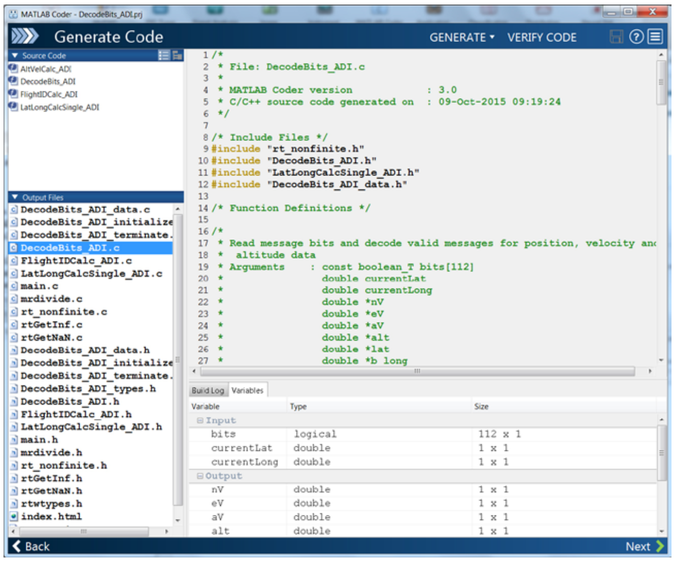
In a traditional radio design process, a large percentage of the development time is spent testing and debugging the HDL code. In the model-based design approach, used in this example, more time was spent on developing the simulation and code generation models. However, there was a significant savings in development time because the generated source code identically matched the validated behavior of the simulation; only a minimal amount of debugging had to be performed on the embedded hardware.

Generating C Code with MATLAB Coder[7]

Similar to HDL code generation, there are several conditions that must be satisfied in order to generate C code for the decoding functionality of this design. The two most important requirements are:

- Use functions supported by MATLAB Coder. MATLAB Coder supports most of the MATLAB language and a wide range of toolboxes,[8] but you may unknowingly use functions that are not supported for code generation. MATLAB Coder provides tools, such as the Code Readiness Tool,[9] to help find any unsupported functions.
- Ensure that once a MATLAB variable is declared, its size and type do not change. This is necessary to make sure that memory allocations are made correctly in the generated code.

The easiest way to generate C code from MATLAB is to open a new MATLAB Coder Project, which can be accessed from the Apps tab on the MATLAB Toolstrip. The final output of the MATLAB Coder Project can be seen in Figure 6.



```
/*
 * File: DecodeBits_ADI.c
 *
 * MATLAB Coder version            : 3.0
 * C/C++ source code generated on  : 09-Oct-2015 09:19:24
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "DecodeBits_ADI.h"
#include "LatLongCalcSingle_ADI.h"
#include "DecodeBits_ADI_data.h"

/* Function Definitions */

/*
 * Read message bits and decode valid messages for position, velocity an
 *  altitude data
 * Arguments    : const boolean_T bits[112]
 *                double currentLat
 *                double currentLong
 *                double *nV
 *                double *eV
 *                double *aV
 *                double *alt
 *                double *lat
 *                double *b long
```

Variable	Type	Size
⊟ Input		
bits	logical	112 x 1
currentLat	double	1 x 1
currentLong	double	1 x 1
⊟ Output		
nV	double	1 x 1
eV	double	1 x 1
aV	double	1 x 1
alt	double	1 x 1

Figure 6. MATLAB Coder project for DecodeBits_ADI.m.

In this project, the top level MATLAB function is DecodeBits_ADI.m. The user needs to specify the data types and sizes required by this function as input arguments. Figure 6 shows that the input arguments of this function are 112 Boolean data bits and two double precision values (to provide the user's current latitude and longitude). The output sizes and data types for DecodeBits_ADI.m (such as *nV for North Velocity, *eV for East Velocity, and *alt for altitude) are automatically determined by MATLAB Coder. MATLAB Coder finds all other functions called by the top level entry point file DecodeBits_ADI.m, including AltVelCalc_ADI.m and LatLongCalc_ADI.m, and then generates the source C code for the entire decoding algorithm.

The C code generated by MATLAB Coder is a fairly straightforward translation of the MATLAB functionality to the C language. As in the case of HDL code generation, the source code produced by MATLAB Coder is readable and traceable, so engineers can easily identify the relationship between the original MATLAB code and the generated C code. The C code from this example can be produced from the MATLAB command prompt and compiled by any ANSI C compiler.

HDL Code Platform Deployment

After partitioning the design into the functionalities that will run on the programmable logic and processing system of the Zynq, optimizing the design for HDL and C code generation, and verifying in simulation that the optimized design is functional and meets the performance criteria, it is now time to deploy the design on to the actual SDR hardware platform and verify the system's functionality under real-world conditions.

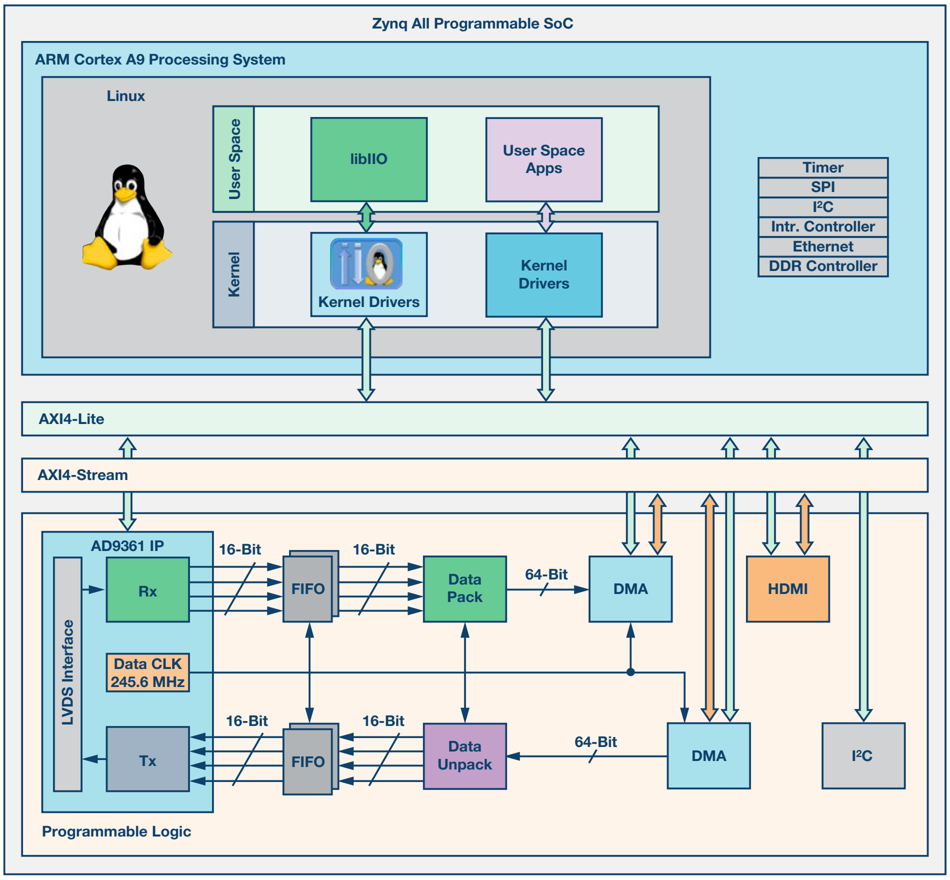


Figure 7. HDL reference design block diagram.

For this purpose, an Analog Devices AD-FMCOMMS3-EBZ SDR platform[10] connected to a Xilinx ZC706 board[11] running the Analog Devices Linux distribution is used.

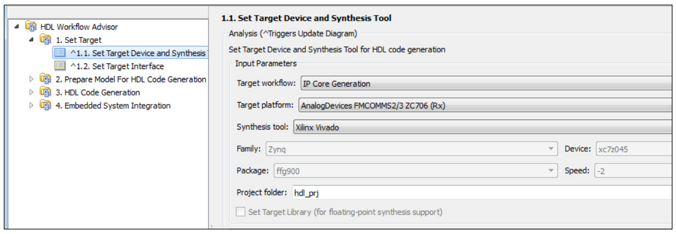
The AD-FMCOMMS3-EBZ board is accompanied by an open-source Vivado HDL reference design provided by Analog Devices.[12] This reference design contains all the IP blocks needed to configure and transfer data to and from the AD9361 transceiver on the AD-FMCOMMS3-EBZ board. Figure 7 presents a block diagram of the HDL reference design.

The AD9361 IP core implements the LVDS receive and transmit data interfaces between the AD9361 transceiver chip and the Zynq device, as well as the data interfaces to the rest of the design. DMA blocks are used for high speed data transfer between the AD9361 IP and the DDR memory. The data interface to the AD9361 IP block consists of four data lines for receive and four data lines for transmit, corresponding to the I&Q data for the two receive and two transmit channels of the AD9361. Each data line is 16 bits wide. To make the data transfers inside the system more efficient, the receive and transmit data is packed into 64-bit wide buses that are managed by the DMA blocks. Pack and unpack blocks are used to connect the 16-bit parallel data lines of the AD9361 IP to the DMAs.

Deploying the HDL code of the ADS-B model into the existing HDL infrastructure of the SDR platform requires creating an IP core that can be inserted into the data path; this is done to process the received data in real time and pass the processed data to the software layer. The deployment process can prove to be a difficult and time consuming task because it requires deep understanding of the HDL design's functionality and also adequate HDL programming skills. To simplify these steps, MathWorks includes a utility in HDL Coder called HDL Workflow Advisor, and Analog Devices provides a board support package (BSP) for the AD-FMCOMMS2-EBZ/AD-FMCOMMS3-EBZ SDR platform and Xilinx ZC706 board.[13]

The HDL Workflow Advisor guides the user through the steps needed to generate HDL code from a Simulink model. The user can choose from a selection of several different Target Workflows, including "ASIC/FPGA," "FPGA-in-the-Loop," and "IP Core Generation." Target Platform selections include Xilinx Evaluation Boards, Altera Evaluation Boards, or the FMCOMMS2/3 ZC706 SDR Platform. The rest of the code generation and target integration process can then be automated by the HDL Workflow Advisor.

The BSP provided by Analog Devices is a collection of board definitions and reference designs[14] that provide the HDL Workflow Advisor the required information and tools to generate an IP block compatible with the existing HDL reference design, and also insert the generated IP into the HDL reference design. Figure 8 shows how to configure the Workflow Advisor to generate the IP core for the ADS-B model. Please note that the IP Core Generation workflow must be selected, targeting the Analog Devices AD-FMCOMMS3-EBZ SDR platform and the Xilinx ZC706 board.


HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synthesis
^1.2. Set Target Interface
2. Prepare Model For HDL Code Generation
3. HDL Code Generation
4. Embedded System Integration

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: AnalogDevices FMCOMMS2/3 ZC706 (Rx)

Synthesis tool: Xilinx Vivado

Family: Zynq Device: xc7z045

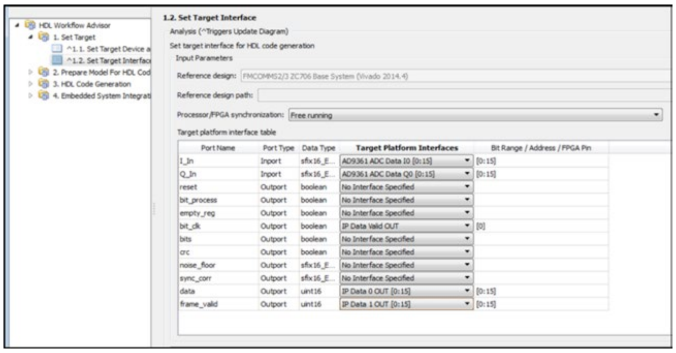
Package: ffg900 Speed: -2

Project folder: hdl_prj

☐ Set Target Library (for floating-point synthesis support)

Figure 8. Workflow Advisor configuration.

The next step is to configure the interfaces between the IP and the reference design. On the input side, the model accepts raw I&Q samples; this connects the model's input ports directly to the AD9361 receiver data ports. Of all the model's output signals, the only ones of interest at this stage are the data, frame_valid, and bit_clk signals. The data and frame_valid are 16 bits wide and are clocked by the bit_clk signal. These signals can be connected to the "DUT Data x Out" interfaces of the BSP, which means they will receive direct access to the DMA blocks; data can then be transferred into the DDR, which is accessible by the software layer. The bit_clk signal is connected to the "DUT Data Valid Out" BSP interface and controls the DMA sampling rate. Figure 9 shows how the HDL interface must be configured.



HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device a
^1.2. Set Target Interfac
2. Prepare Model For HDL Cod
3. HDL Code Generation
4. Embedded System Integrat

1.2. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Reference design: FMCOMMS2/3 ZC706 Base System (Vivado 2014.4)

Reference design path:

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
I_In	Inport	sfix16_E...	AD9361 ADC Data I0 [0:15]	[0:15]
Q_In	Inport	sfix16_E...	AD9361 ADC Data Q0 [0:15]	[0:15]
reset	Outport	boolean	No Interface Specified	
bit_process	Outport	boolean	No Interface Specified	
empty_reg	Outport	boolean	No Interface Specified	
bit_clk	Outport	boolean	IP Data Valid OUT	[0]
bits	Outport	boolean	No Interface Specified	
crc	Outport	boolean	No Interface Specified	
noise_floor	Outport	sfix16_E...	No Interface Specified	
sync_corr	Outport	sfix16_E...	No Interface Specified	
data	Outport	uint16	IP Data 0 OUT [0:15]	[0:15]
frame_valid	Outport	uint16	IP Data 1 OUT [0:15]	[0:15]

Figure 9. HDL interface configuration.

Once the target interface has been defined, Step 2 and Step 3 of the HDL Workflow Advisor can be left in their default state and the project generation process can be started by running Step 4.1 (Create Project). The result of this step is a Vivado project that has the ADS-B IP core integrated into the Analog Devices HDL reference design. Figure 10 depicts the connections between the ADS-B IP core and the rest of the blocks in the design.

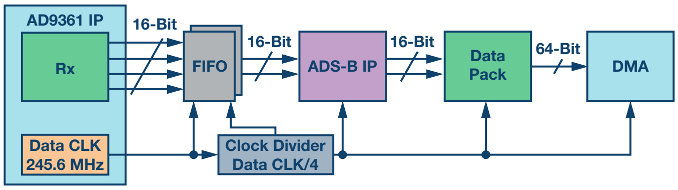


Figure 10. ADS-B IP connections in the HDL reference design.

Generating the bitstream from the Vivado project concludes the HDL integration process, but the final goal is to have Linux running on the system. For this purpose, after generating the bitstream, a Linux boot file can be created by following the standard Xilinx SDK first stage boot loader (fsbl) and Linux boot file creation process. The Linux device tree and image files corresponding to the newly created HDL design are distributed with the AD-FMCOMMS3-EBZ BSP. All files must be copied together with the Linux boot file on the boot partition of the SD card; this is used to store all files needed to run the Analog Devices Linux Distribution on the Xilinx ZC706 board.

C Code Platform Deployment

Now that the ADS-B HDL IP has been integrated into the SDR platform's HDL design, and the Linux SD card is created, it is time to implement the software application that decodes the ADS-B data. This application is based on the C code generated in Section 5 and performs the following tasks:

- Configures the AD9361 for ADS-B signals reception.
- Reads the data from the ADS-B IP core.
- Detects the valid ADS-B frames in the read data.
- Decodes and displays the ADS-B information.

The easiest way to implement Task 1 and Task 2 is to use the functionality provided by the libiio library.[15] This library provides interface functions that enable users to easily configure the AD9361 as well as receive and transmit data. The configuration sequence sets the following system parameters:

- LO frequency—1.09 GHz
- Sampling rate—12.5 MHz
- Analog bandwidth—4.0 MHz
- AGC—fast attack mode

Besides the parameters mentioned above, a digital FIR filter with data rate of 12.5 MSPS, a pass band frequency of 3.25 MHz, and a stop band frequency of 4 MHz is loaded into the AD9361 to ensure that the received data contains only the band of interest. The system parameters and the design methodology of this FIR filter are described in Part 3 of this article series.[3]

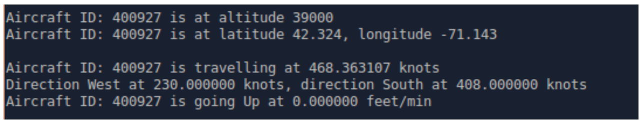
The output data of the ADS-B IP is transferred into the system's DDR memory by the DMA block. The libiio library provides the following functions: position the data acquired from the ADS-B IP into a memory buffer with a specified size; wait for the buffer to be filled; gain access the buffer through pointers. Once the buffer is filled, the ADS-B decoding algorithm can process the data. The ADS-B IP core has two output channels: one channel corresponding to the ADS-B bitstream, and the other channel indicating where a valid data frame ends in the bitstream. Both channels contain the same data rate and are synchronized with each other. A sample equal to "1" in the valid channel denotes the last bit of a valid frame in the data channel. By parsing both channels, the software can extract the valid ADS-B data frames from the bitstream and pass the data to the decoding function generated by MATLAB Coder. The decoding function uses the ADS-B data frame and the latitude and longitude of the current location as input when computing the aircraft's coordinates. The current latitude and longitude are specified as parameters of the application. The decoded ADS-B data is displayed similarly to the Simulink model.

The ADS-B data decoding application is built under Linux using a makefile. The source code of the application and the makefile can be found on the Analog Devices GitHub repository.[16]

This completes the platform deployment steps for both the HDL and C code generated from the ADS-B model using HDL Coder and MATLAB Coder from MathWorks. The next step is to verify the system's functionality and evaluate the results.

System Validation

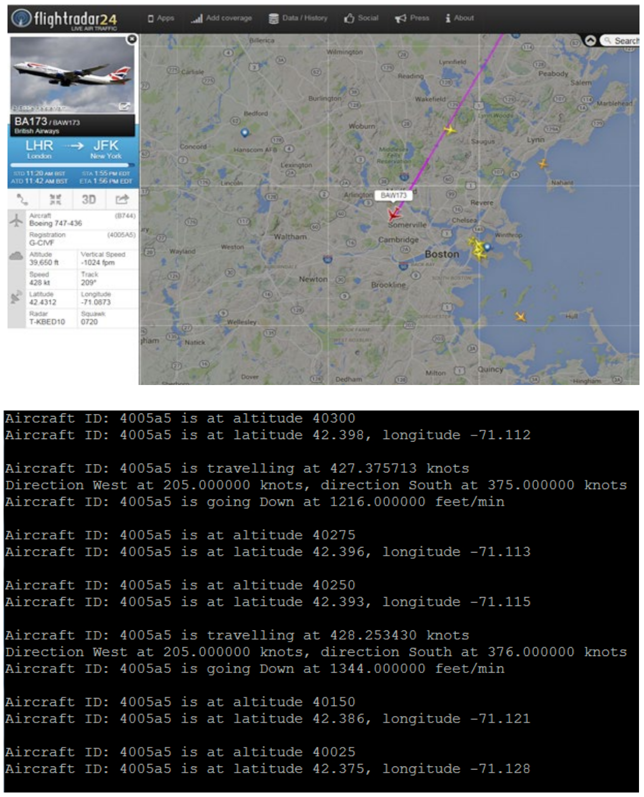
To validate the system's functionality, begin by creating a loopback connection between one receive and one transmit port of the AD-FMCOMMS3-EBZ board and transmit the same ADS-B signal that was used during simulation. By receiving and decoding this data, it can be verified that the output of the algorithm running on the SDR platform matches the simulation results. Figure 11 displays the output of the ADS-B data decoding application; the results are identical to those shown in Part 3 of the article series for HIL simulation using precaptured data. This provides confidence that the system is running as expected and is ready to be used with real-world data.



```
Aircraft ID: 400927 is at altitude 39000
Aircraft ID: 400927 is at latitude 42.324, longitude -71.143

Aircraft ID: 400927 is travelling at 468.363107 knots
Direction West at 230.000000 knots, direction South at 408.000000 knots
Aircraft ID: 400927 is going Up at 0.000000 feet/min
```

Figure 11. Loopback results.

For the actual field test, the SDR receiver was placed outside the MathWorks headquarters in Natick, MA, and compared against ADS-B information decoded by the system with the data provided by airplane live tracking websites (such as flightradar24.com). It was observed that the system was able to decode data received from the airplanes within the antenna's line of sight. Figure 12 shows a comparison between the aircraft information detected by the system and the online airplane tracking data; the decoding algorithm displays the correct aircraft ID, altitude, speed, and latitude/longitude coordinates.



```
Aircraft ID: 4005a5 is at altitude 40300
Aircraft ID: 4005a5 is at latitude 42.398, longitude -71.112

Aircraft ID: 4005a5 is travelling at 427.375713 knots
Direction West at 205.000000 knots, direction South at 375.000000 knots
Aircraft ID: 4005a5 is going Down at 1216.000000 feet/min

Aircraft ID: 4005a5 is at altitude 40275
Aircraft ID: 4005a5 is at latitude 42.396, longitude -71.113

Aircraft ID: 4005a5 is at altitude 40250
Aircraft ID: 4005a5 is at latitude 42.393, longitude -71.115

Aircraft ID: 4005a5 is travelling at 428.253430 knots
Direction West at 205.000000 knots, direction South at 376.000000 knots
Aircraft ID: 4005a5 is going Down at 1344.000000 feet/min

Aircraft ID: 4005a5 is at altitude 40150
Aircraft ID: 4005a5 is at latitude 42.386, longitude -71.121

Aircraft ID: 4005a5 is at altitude 40025
Aircraft ID: 4005a5 is at latitude 42.375, longitude -71.128
```

Figure. 12 Live data results.

Conclusion

This article concludes the four part article series demonstrating how model-based design can be used to take an SDR system all the way from simulation to production. The series addressed all the stages of developing a "hardware ready" ADS-B Simulink model. We designed a simulation model to prove we could decode recorded ADS-B messages, and then validated the model with live data acquired from the SDR hardware platform. This validated not only the model but also the SDR platform's settings for the analog front end and digital receiver chain; it also gave us confidence that the platform was properly tuned for receiving ADS-B signals. Afterward, we partitioned the model into the functionalities that run on the Zynq processing system and programmable logic, and optimized the model for automatic C and HDL code generation. Finally, we integrated the C and HDL code into the SDR design and validated the system's functionality with live commercial air traffic. The end result is a design process that uses modeling and code generation tools from MathWorks, together with the Zynq SDR platform, to create a fully functional SDR system.

This example system shows that the model-based design workflow in combination with the Analog Devices AD9361/AD9364 integrated RF Agile Transceiver programmable radio hardware can help design teams develop working radio prototypes more quickly and less expensively than using traditional design methodologies. This prototype was built by the authors in a relatively short time with minimal obstacles, drawing on the following resources:

- The ability to build a model of an ADS-B receiver in MATLAB and Simulink that can generate usable C and HDL source code.
- Functions within HDL Workflow Advisor to automate many of the hardware/software integration steps.
- Libraries (such as libiio) that assist in the remaining integration steps to deploy the SDR prototype.
- Product help and technical support that are available from MathWorks and Analog Devices.

ADS-B is a relatively simple standard and provides a good test case to demonstrate this approach to building an SDR prototype. Engineers who adopt model-based design and the Zynq SDR platform should be able to follow the workflow presented in this series of articles to develop much more complex and powerful QPSK-, QAM-, and LTE-based SDR systems.

References

[1] Di Pu, Andrei Cozma, and Tom Hill. "Four Quick Steps to Production: Using Model-Based Design for Software-Defined Radio. Part 1—the Analog Devices/Xilinx SDR Rapid Prototyping Platform: Its Capabilities, Benefits, and Tools." *Analog Dialogue*, Volume 49, Number 3.

[2] Mike Donovan, Andrei Cozma, and Di Pu. "Four Quick Steps to Production Using Model-Based Design for Software-Defined Radio. Part 2—Mode S Detection and Decoding Using MATLAB and Simulink." *Analog Dialogue*, Volume 49, Number 4.

[3] Di Pu, Andrei Cozma. "Four Quick Steps to Production Using Model-Based Design for Software-Defined Radio. Part 3—Mode S Signals Decoding Algorithm Validation Using Hardware in the Loop." *Analog Dialogue*, Volume 49, Number 4.

[4] Analog Devices GitHub repository.

[5] HDL Coder.

[6] HDL Coder Block Support.

[7] MATLAB Coder.

[8] MATLAB Toolboxes.

[9] MATLAB Code Generation Readiness Tool.

[10] AD-FMCOMMS3-EBZ User Guide.

[11] Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit.

[12] AD-FMCOMMS2-EBZ/AD-FMCOMMS3-EBZ/AD-FMCOMMS4-EBZ HDL/AD-FMCOMMS5-EBZ HDL Reference Design.

[13] Analog Devices BSP for MathWorks HDL Workflow Advisor.

[14] Board and Reference Design Registration System.

[15] What Is Libiio?

[16] MathWorks Targeting Models—ADSB.

Mike Donovan [mike.donovan@mathworks.com] is a manager in the Application Engineering Group at MathWorks. He has a B.S.E.E. from Bucknell University and an M.S.E.E. from the University of Connecticut. Prior to joining MathWorks, Mike worked on radar and satellite communications systems and in the broadband telecommunications industry.

Mike Donovan

Andrei Cozma [andrei.cozma@analog.com] is an engineering manager for ADI, supporting the design and development of system level reference designs. He holds a B.S. degree in industrial automation and informatics and a Ph.D. in electronics and telecommunications. He has been involved in the design and development of projects from different industry fields such as motor control, industrial automation, software-defined radio, and telecommunications.

Andrei Cozma

Also by this Author:

FPGA-Based Systems Increase Motor-Control Performance

Volume 49, Number 1

Di Pu [di.pu@analog.com] is a system modeling applications engineer for ADI, supporting the design and development of software-defined radio platforms and systems. She has been working closely with MathWorks to solve mutual end customer challenges. Prior to joining ADI, she received her B.S. degree from Najing University of Science and Technology (NJUST), Nanjing, China, in 2007 and her M.S. and Ph.D. degrees from Worcester Polytechnic Institute (WPI), Worcester, MA, U.S.A., in 2009 and 2013—all in electrical engineering. She is a winner of the 2013 Sigma Xi Research Award for Doctoral Dissertation at WPI.

Di Pu

New Complete, High Resolution, and Multifunctional Bipolar DACs: an Easy to Use, Universal Solution

By Estibaliz Sanz Obaldia and Junifer Frenila

With current market dynamics constantly driving toward shorter design cycles, enhanced system functionality, and more portable end systems, the need for new methodology to simplify these challenges without adding design complexity is a must. This article will address some key system challenges for control and measurement that are topical across a multitude of applications, including data acquisition systems, industrial automations, programmable logic controllers, and motor controls. It will explore the latest advances in bipolar digital-to-analog converter (DAC) architectures and how these topologies can address end system challenges, such as by adding even more functionality and intelligence within the same or reduced space. This article will explore discrete and more functionally complete solutions. Finally it will outline a number of alternatives to traditional design topology that support higher flexibility in design reuse and system modularity.

It should be noted that the following figures are not the actual schematics, but illustrations on how applications could be achieved with multifunctional DACs and other components. While it does not include aspects such as circuits for power supplies, bypassing, and other passive components, these diagrams illustrate how applications can be implemented in general.

Data Acquisition Systems

Data acquisition systems (DAQs) are used to measure an electrical or physical singularity such as voltage, current, or pressure with a microcontroller or microprocessor (MPU) for data processing capability. DAQs consist of sensors, amplifiers, data converters, and a controller with embedded software that controls the acquisition process.

In a process control application, it is critical that the sensor is sensitive enough to preserve the quality of the signal to be measured. But even if the sensor is sensitive enough, the signal chain errors such as gain and offset could still interfere with the signal quality. High performance applications employ DACs in automatic calibration of the conditioning circuits in data acquisition systems. Figure 1 shows the block diagram of a pressure sensing system. It illustrates how bipolar DACs such as AD5761R and its product family can be used in an automated gain and offset calibration scheme.

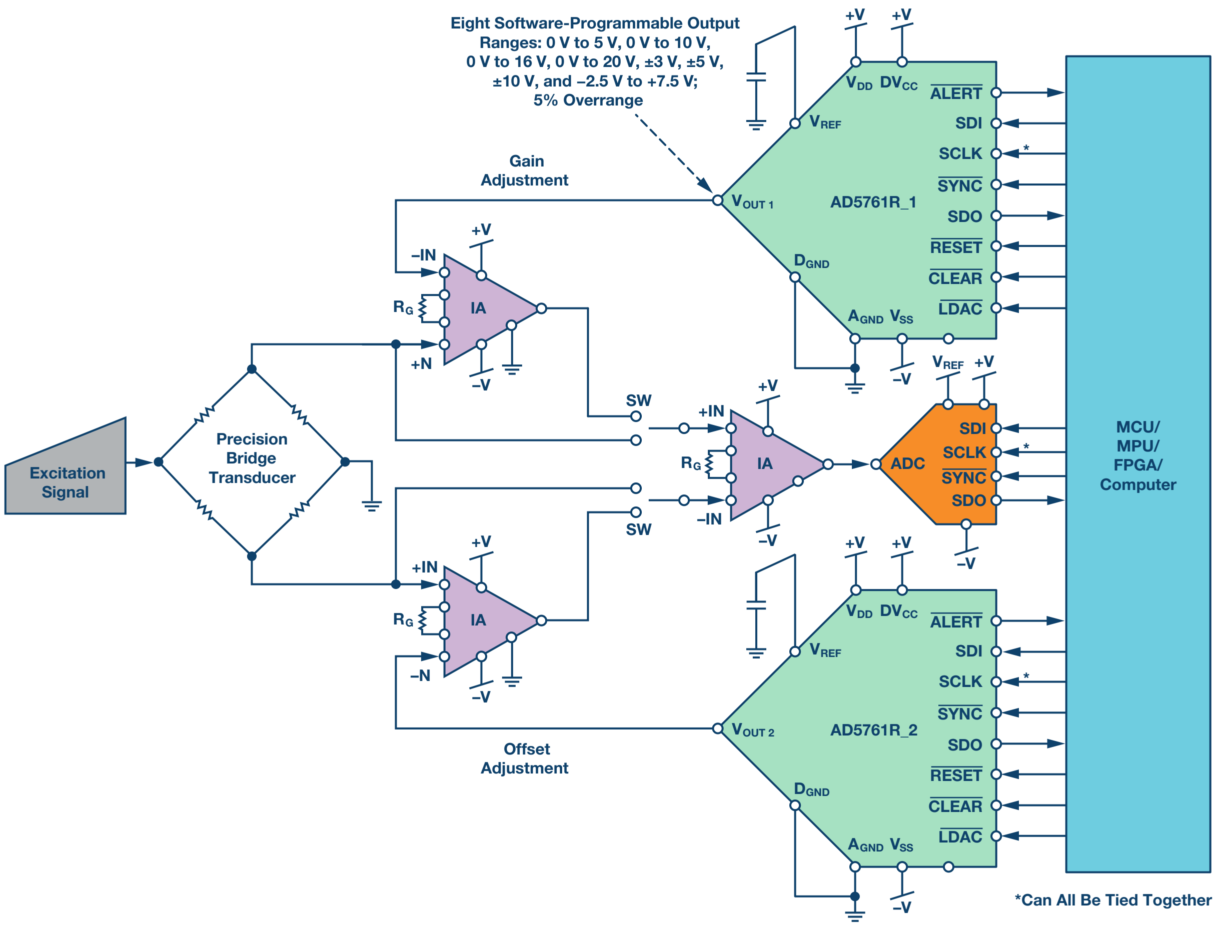

Eight Software-Programmable Output Ranges: 0 V to 5 V, 0 V to 10 V, 0 V to 16 V, 0 V to 20 V, ±3 V, ±5 V, ±10 V, and −2.5 V to +7.5 V; 5% Overrange
Gain Adjustment
Offset Adjustment
Excitation Signal
Precision Bridge Transducer
IA
RG
+IN
−IN
+V
−V
SW
ADC
AD5761R_1
AD5761R_2
VDD DVCC
VREF
VOUT 1
VOUT 2
DGND
AGND VSS
ALERT
SDI
SCLK
SYNC
SDO
RESET
CLEAR
LDAC
MCU/ MPU/ FPGA/ Computer
*Can All Be Tied Together

Figure 1. Automated calibration of a pressure sensing system.

The precision bridge transducer receives an excitation signal from a pressure sensor and produces an output voltage. Due to the low amplitude of the transducer's signal, an instrumentation amplifier is typically used as a signal multiplier. This low amplitude signal is susceptible to errors. Such errors are usually contributed by drift due to changes in temperature, parasitic errors across circuit boards, and tolerances of passive components.

With the use of AD5761R, gain and offset calibrations can be implemented into the system to dynamically correct the errors as the system operates over time. Depending on the level of adjustment and the polarity required, a complete, high resolution, and multifunctional bipolar DAC can greatly simplify the calibration process. The AD5761R can be programmed through a high speed, 4-wire SPI interface with a serial data output (SDO) line available to facilitate daisy-chain and readback operation.

Industrial Automation

The applications for industrial automation are diverse. But regardless of what applications there may be, the functionality and performance of such automated systems lie in their signal acquisition and control units. On the acquisition side, the sensitivity of the sensors, adaptability of the conditioning circuits, and the speed of acquiring correct information from low level signals is very important. On the control side, the flexibility to adapt to the requirement of various actuators and drivers is vital.

Figure 2 shows an example of an industrial automated system. A thermocouple with cold-junction compensation is used to measure the temperature of industrial equipment such as a laser machine or heavy duty motor. The voltage is gained up, filtered, and sent to an integrated analog front-end (AFE) IC for conversion and the digital data is passed into the processor for analysis. Based on the processed data, the processor sends signal to a control DAC, which is also fully isolated, to drive an industrial fan, activate a cooling apparatus such as a Peltier, or open the valve of a water cooling system. Additionally, the user can input an override command via a control interface device.

The same system can be adopted for pressure and vibration measurement and control. A pressure sensor system can typically be used for oil and chemical tank monitoring, while a gyroscope system can typically be used for vibration monitoring of fast moving machine heads. These applications share the same AFE that is fully isolated from the external environment.

The AD5761R, a high voltage, high resolution, bipolar DAC with a low drift internal reference and software-selectable output range is a practical replacement for multiple DACs or a single multiplexed DAC. It provides unipolar and bipolar voltages while maintaining the same accuracy with an option of overrange output. This bipolar DAC supports the different needs that actuators require, including the adjustment of the control unit through software avoiding hardware modifications.

AD5761R and its product family come in small packages—a 3 mm × 3 mm, lead frame chip scale package (LFCSP) and 16-lead thin shrink outline package (TSSOP)—and support a wide operating temperature range of –55°C to +125°C. This new industrial control approach essentially helps to minimize board space and reduce cost.

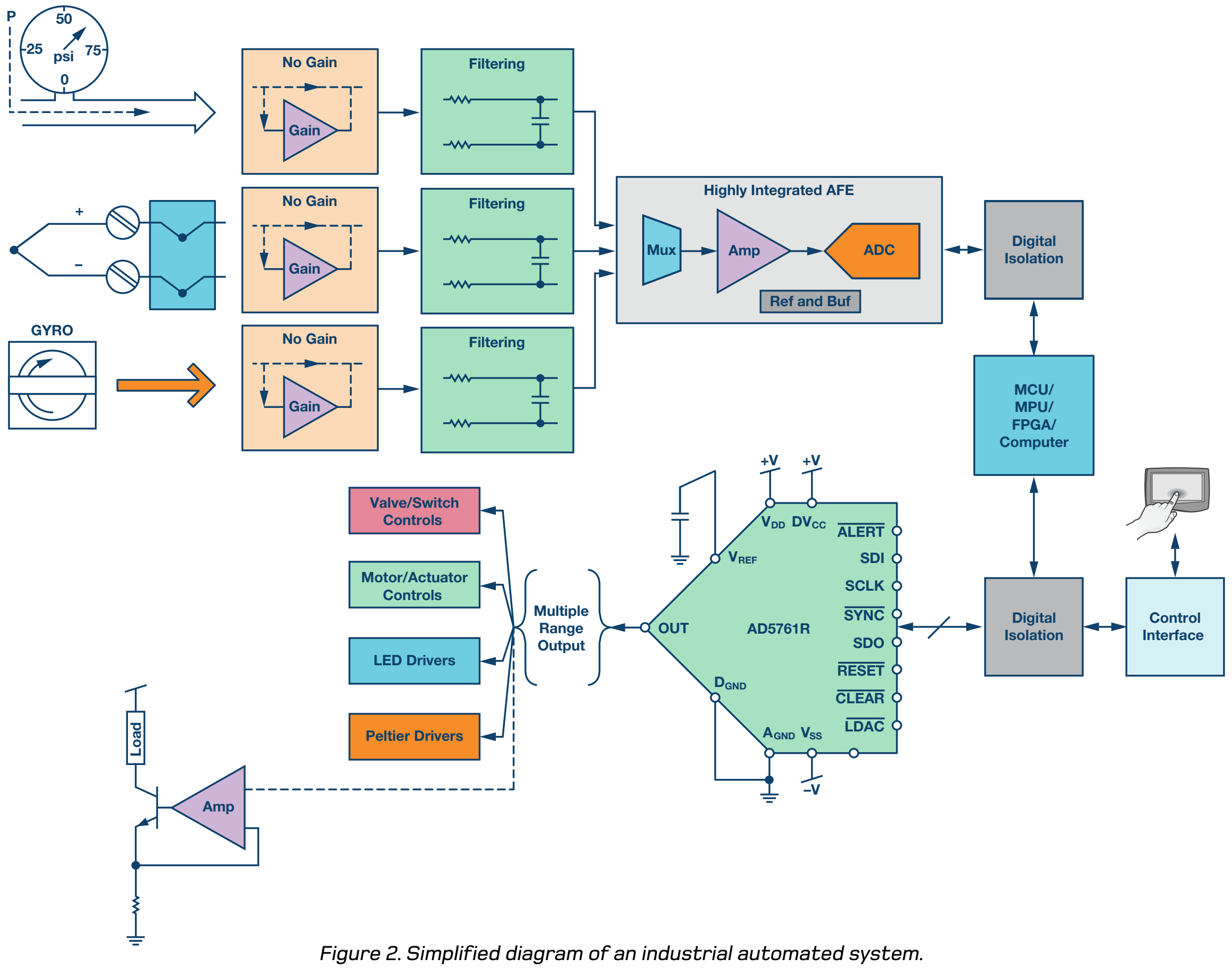


Figure 2. Simplified diagram of an industrial automated system.

Programmable Logic Controllers

Programmable logic controllers (PLCs) incorporate power supplies, central processing units, and several analog and digital I/O modules in order to control, actuate, and monitor complex machine variables. PLCs are widely used across industries and they offer extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. A fundamental process control system building block is shown in Figure 3. An input signal reporting on the status of a process variable is monitored via the input module and transferred to the MCU to be analyzed. Based on the results of this analysis, a response containing the necessary arrangements is managed by the output module to control the devices in the system.

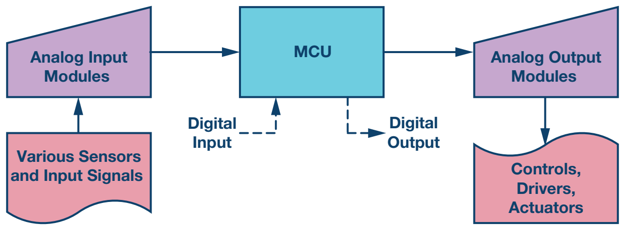


Figure 3. Process control system building block.

Figure 4 shows a more complete industrial PLC system including an embedded controller/processor as the main system controller interfacing to the fully isolated input and output modules. Excluding the power supply module, the system is divided into four subsystems that differentiate the analog input, analog output, digital input, and analog output modules. Several types of sensors are deployed to acquire analog signals of different amplitudes and frequencies. These signals need to be preprocessed and converted into digital form for further analysis. Programmable gain amplifiers condition the small input signals such that they can be accurately measured and converted into their digital representation by analog-to-digital converters (ADCs). Isolation is required to protect the controller or processor from possible unexpected overvoltage coming from the field, for which optical or integrated isolators are placed among the processor and the input and output modules.

The accuracy and resolution requirements for the input and output modules are considerably distinct. While the input modules are required to monitor highly precise and accurate data acquisitions from the process, the output modules essentially adjust the output with a 16-bit resolution and accuracy in high end applications. As a result of these conditions, Σ-Δ ADCs are commonly used for input modules in PLC systems from which a wide range of isolated, single-channel/multichannel, and simultaneous sampling ADCs are available in the market.

Output modules may offer precision voltage DACs, precision current DACs, or a combination of both. Several methods allow current and voltage levels to be generated for the PLC's analog output. The evolution of precision bipolar DACs such as the AD5761R, providing extra functionality and a high level of integration, significantly benefit PLC systems from reduction of system complexity, board size, and cost.

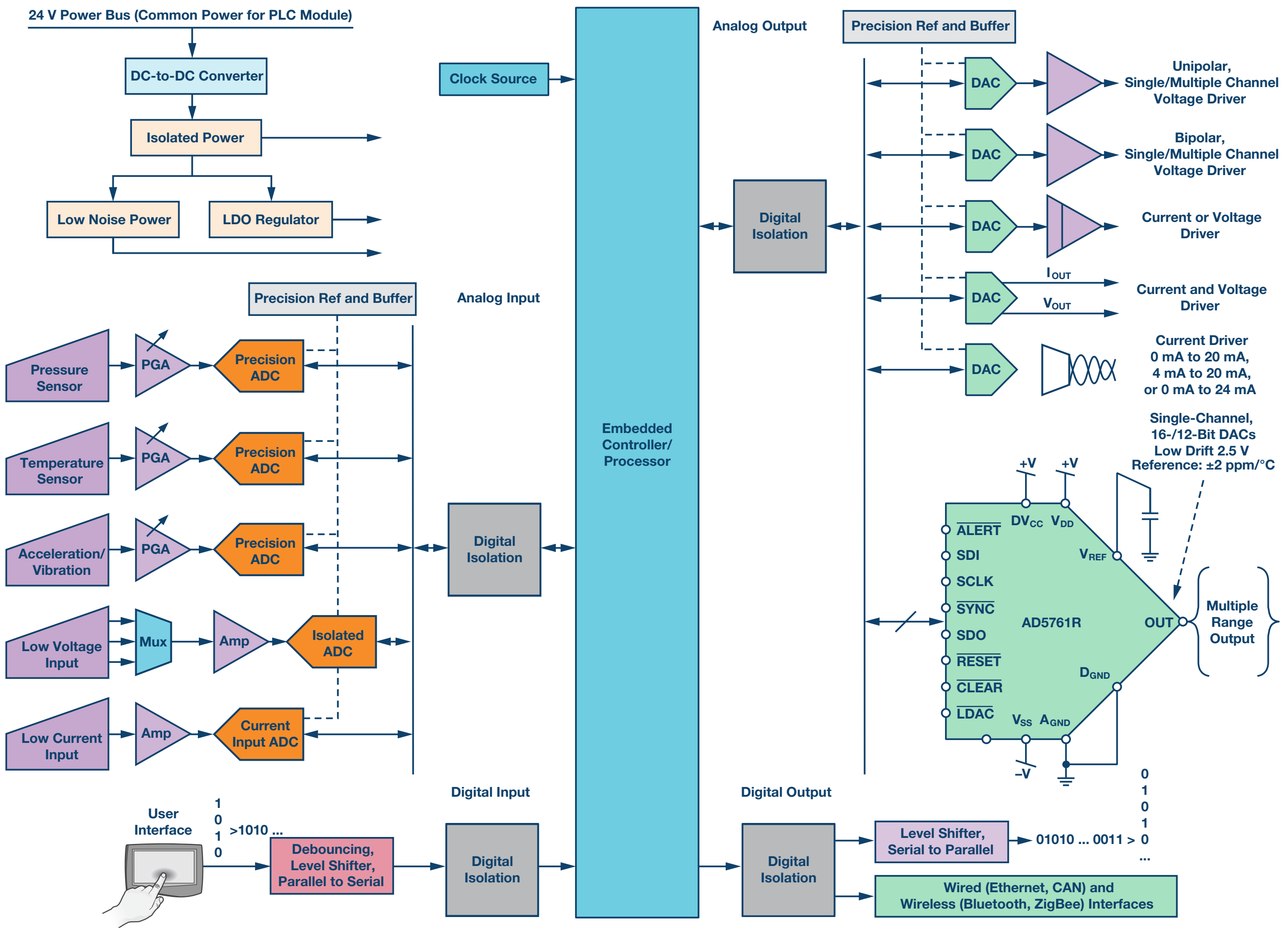


Figure 4. Block diagram of a complete PLC system.

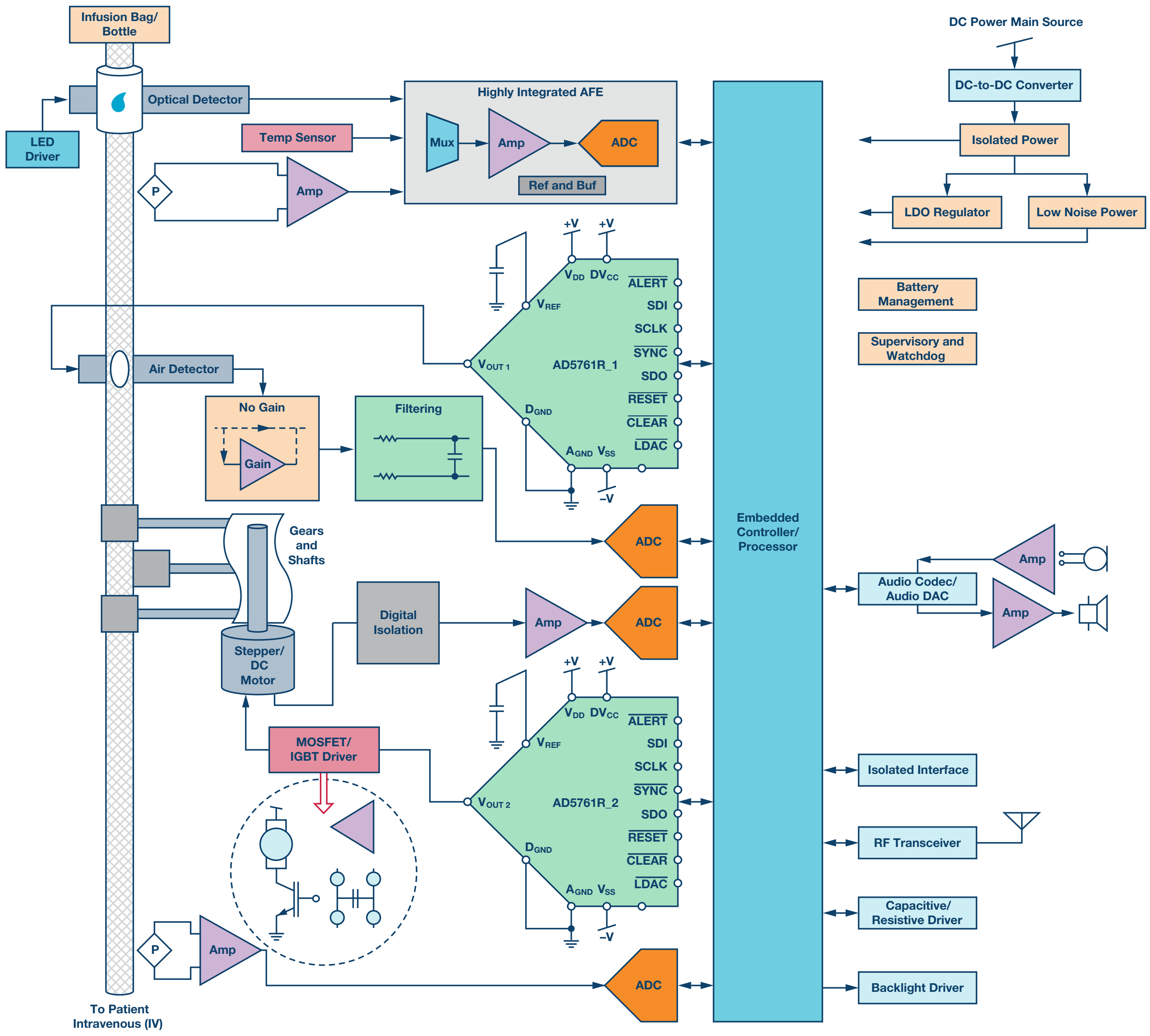


Figure 5. Large volume infusion pump system.

Motor Controls

DACs perform an integral function in motor control loops; for example, in infusion pump systems. Infusion pumps are widely used in human healthcare to provide medical treatment to patients of all ages. The role of an infusion pump is to deliver fluids, medication, or supplements to the patient's cardiovascular system in an intermittent or continuous procedure.

Although infusion pumps require a qualified user to program the specific parameters for the treatment, the implicated advantages over manual administration influence increasing user confidence. The capability of these instruments to accurately deliver tiny dosages at scheduled intervals in a self-operated mode negates the need for a nurse or doctor to manually control the flow of fluid to the patient. Doctors and medical administrators can depend on the safety of infusion pump systems to display real-time system information on dosage limitations for titration safety, to prevent overdose, as well as the physical delivery mechanism itself to be reliable and accurate.

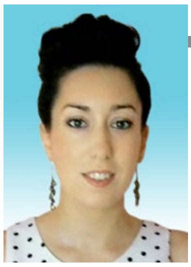
During operation, the microcontroller receives the monitored speed and direction signals from the dc motor, which are analyzed and adjusted (if required) to meet the setpoint. The DAC in the feedforward path provides the adjustments to the system while the ADC in the feedback path monitors the effect of each adjustment. The desired setpoint voltage set by the DAC is amplified through the driver network to provide the required drive current to the dc motor.

ADI offers high performance analog and mixed-signal processing solutions for detecting, measuring, and controlling sensors and actuators used in chemistry analyzers, flow cytometers, infusion pumps, dialysis equipment, ventilators, catheters, and many more medical instruments. In particular, the AD5761R, a high resolution, bipolar DAC with eight available software selectable output ranges while maintaining a common accuracy is an ideal part for motor control applications, supporting the different voltage swings needed by motors.

Conclusion

DACs play a key role in determining the performance and accuracy of many control systems and simple conversion circuits, as well as other complex applications. The AD5761R and its product family, which is a complete 16-bit resolution precision bipolar DAC with multiple programmable output ranges, are suitable for the above applications. The highly configurable ranges of the AD5761R family of DACs (0 V to 5 V, 0 V to 10 V, 0 V to 16 V, 0 V to 20 V, ±3 V, ±5 V, ±10 V, and −2.5 V to +7.5 V; 5% overrange), make this family of DACs a one size fits all solution for data acquisition systems, industrial automation, programmable logic controllers, and motor controllers. The integration offered within the AD5761R product family, including output buffer and a buffered 2 ppm/°C internal reference, significantly simplifies board design, reduces board size, and minimizes power consumption and cost.

Estibaliz Sanz Obaldia [Estibaliz.Sanz@analog.com] received her bachelor's degree in electronic engineering and automation from University of Deusto. Estibaliz joined ADI in 2010 and works as an applications engineer in the Precision Converter Group in Limerick, Ireland.



Estibaliz Sanz Obaldia

Junifer Frenila [Junifer.Frenila@analog.com] received his bachelor's degree in electronics and communications engineering from WVCST in 2005. He joined ADI in 2006 and works as a design evaluation engineer in the Precision Converter Group in ADI Philippines. Junifer is currently working on his doctoral degree in electronics engineering at Mapúa Institute of Technology.

Junifer Frenila

Notes

Analog Devices, Inc.
Worldwide Headquarters

Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
U.S.A.
Tel: 781.329.4700
(800.262.5643,
U.S.A. only)
Fax: 781.461.3113

Analog Devices, Inc.
Europe Headquarters

Analog Devices, Inc.
Wilhelm-Wagenfeld-Str. 6
80807 Munich
Germany
Tel: 49.89.76903.0
Fax: 49.89.76903.157

Analog Devices, Inc.
Japan Headquarters

Analog Devices, KK
New Pier Takeshiba
South Tower Building
1-16-1 Kaigan, Minato-ku,
Tokyo, 105-6891
Japan
Tel: 813.5402.8200
Fax: 813.5402.1064

Analog Devices, Inc.
Asia Pacific Headquarters

Analog Devices
5F, Sandhill Plaza
2290 Zuchongzhi Road
Zhangjiang Hi-Tech Park
Pudong New District
Shanghai, China 201203
Tel: 86.21.2320.8000
Fax: 86.21.2320.8222

analog.com/analogdialogue

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™