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]

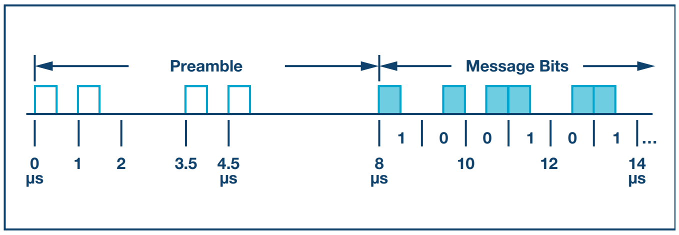


Figure 1. Structure of a Mode S message.

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

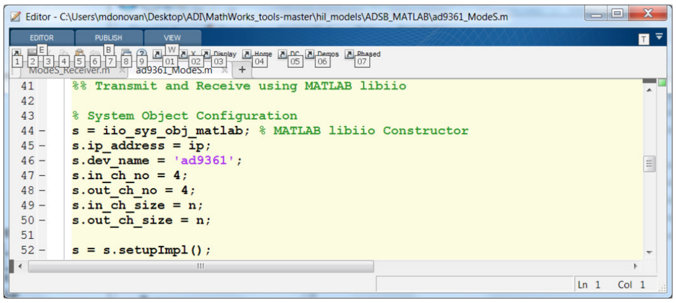
1. The receive environment typically contains very short messages interspersed with long idle periods, and the received signals can be very weak when the transmitting aircraft is a long distance from the receiver. Legacy waveforms are also transmitted at 1090 MHz. The receiver needs to use the preamble 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

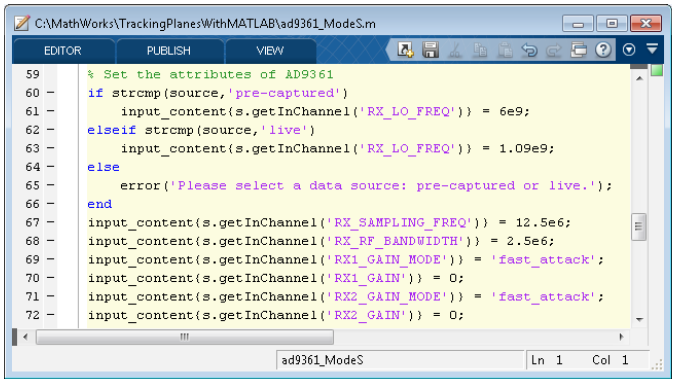
Readers who are interested in following along with the MATLAB source code can find the files on the Analog Devices GitHub repository. The entry level function is ad9361_ModeS.m, and the files called by this function are also provided.

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

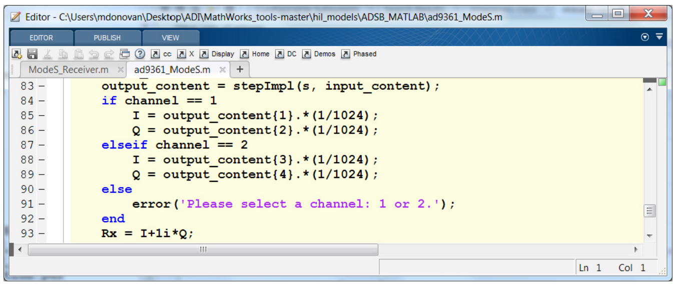
```
%% Transmit and Receive using MATLAB libiio

% 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.


C:\MathWorks\TrackingPlanesWithMATLAB\ad9361_ModeS.m

```
% 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

```
    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.

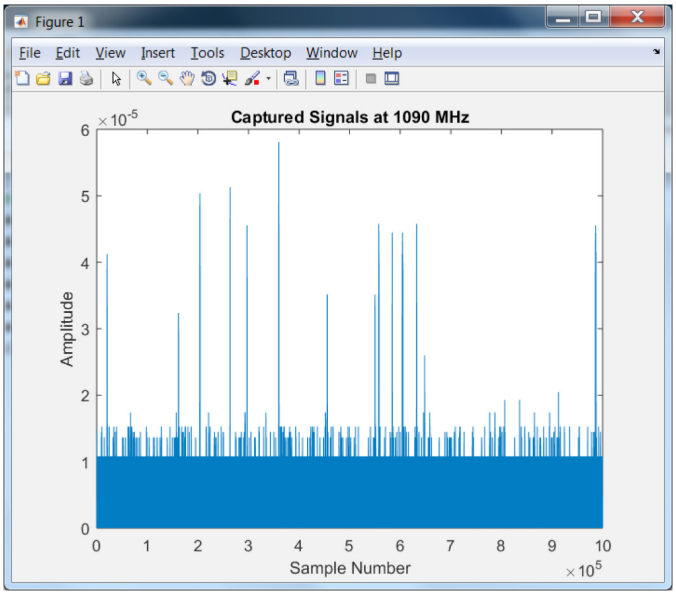


Figure 5. Sample data capture at 1090 MHz.

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).

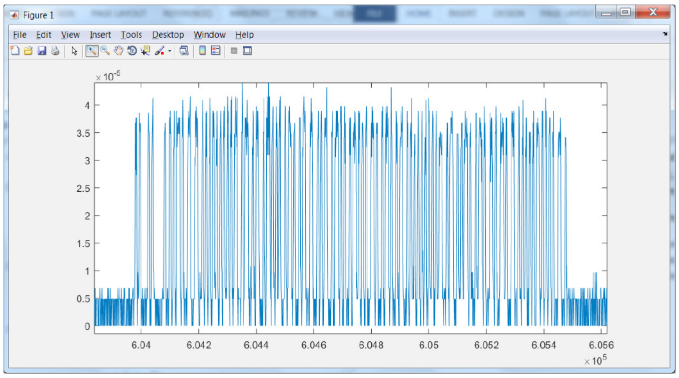


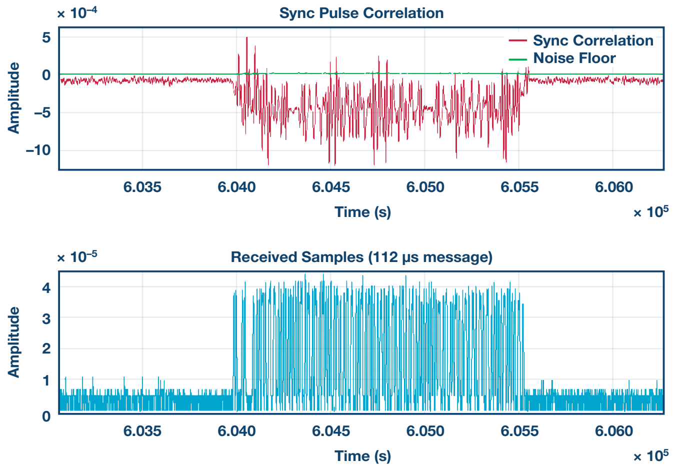
Figure 6. Single Mode S message.

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.

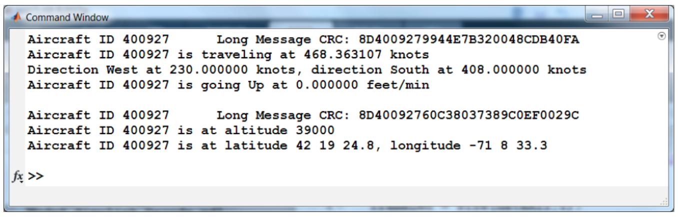
2. When the preamble correlation exceeds the noise floor by a significant factor, launch logic to find the first message bit sample.
   a. The choice of this threshold is subjective. It should be small enough to detect weak signals but large enough to prevent a lot of false positives. We chose a value 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.

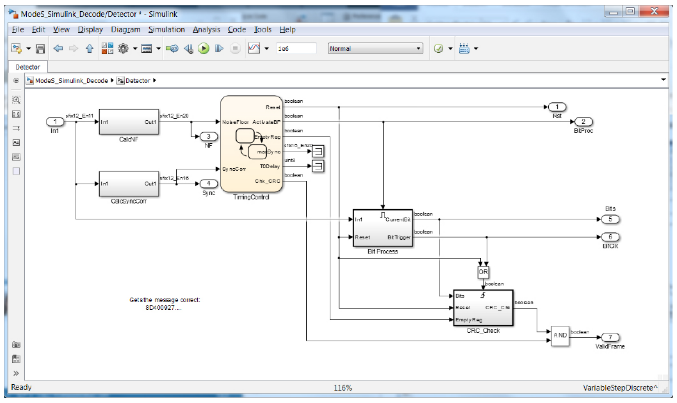
Path to Implementation

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).



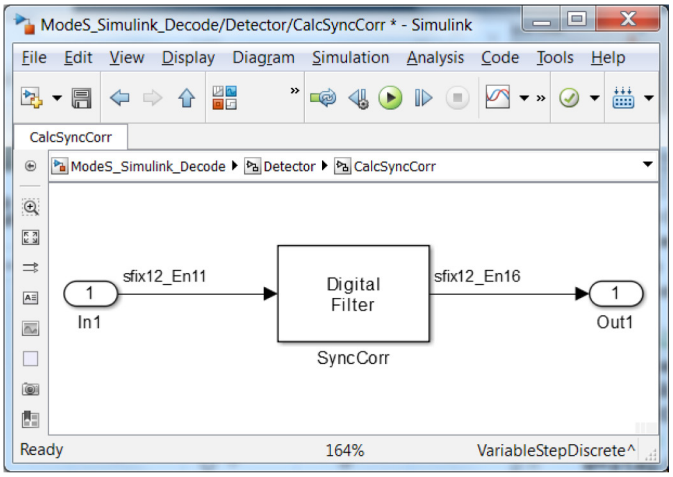
© 1984–2015 The MathWorks, Inc.

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.

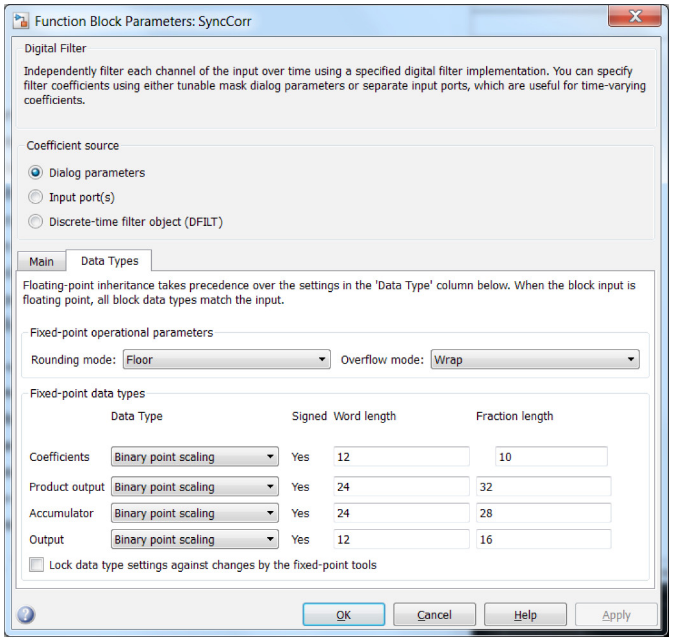
Digging deeper into the model you can see a few features that make Simulink suitable for embedded development, especially for partitioning the design into functions targeted at a Zynq SoC and for generating HDL code and C code.

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

© 1984–2015 The MathWorks, Inc.

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

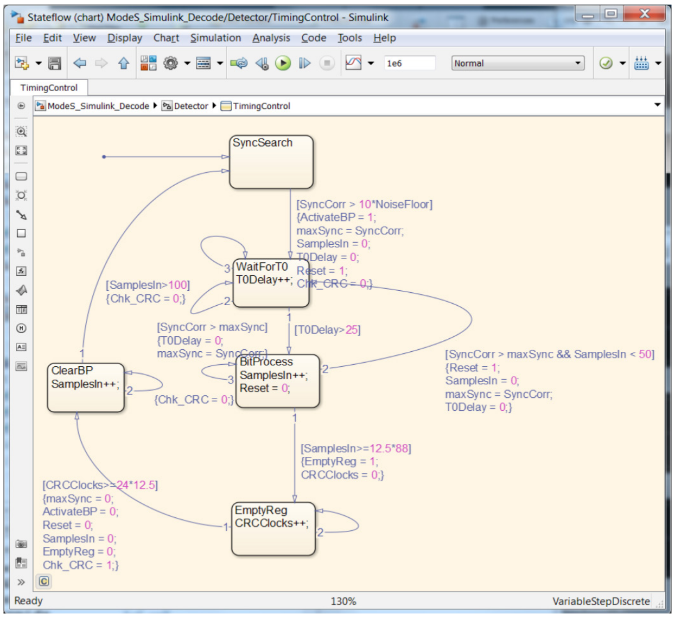
OK | Cancel | Help | Apply

© 1984–2015 The MathWorks, Inc.

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.

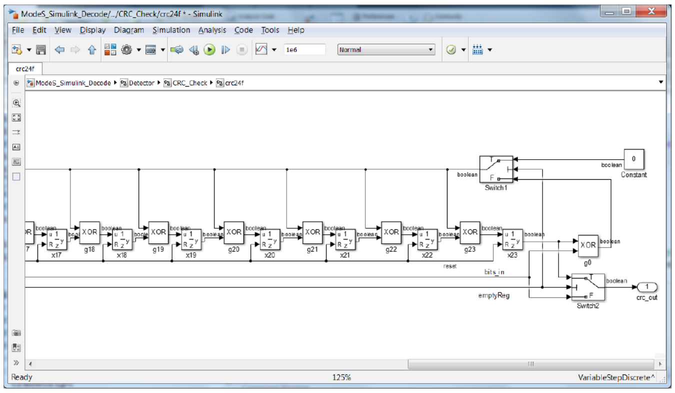

Stateflow (chart) ModeS_Simulink_Decode/Detector/TimingControl - Simulink
File Edit View Display Chart Simulation Analysis Code Tools Help
1e6
Normal
TimingControl
ModeS_Simulink_Decode ▸ Detector ▸ TimingControl
Ready
130%
VariableStepDiscrete

```
SyncSearch

WaitForT0
T0Delay++;

BitProcess
SamplesIn++;
Reset = 0;

ClearBP
SamplesIn++;

EmptyReg
CRCClocks++;

[SyncCorr > 10*NoiseFloor]
{ActivateBP = 1;
maxSync = SyncCorr;
SamplesIn = 0;
T0Delay = 0;
Reset = 1;
Chk_CRC = 0;}

[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;}

{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;}
```

© 1984–2015 The MathWorks, Inc.

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
emptyReg
crc_out
Ready
125%
VariableStepDiscrete

© 1984–2015 The MathWorks, Inc.

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] Pu, Di, 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