

Model-Based Design of Fixed-Point Filters for Embedded Systems

Mark Corless
Arvind Ananthan
 The MathWorks, Inc.

Copyright © 2009 SAE International

ABSTRACT

Digital filters are used in many automotive applications ranging from identification and conditioning of signals in an engine controller to digital radio receivers. Often these filters are implemented in fixed point for reasons such as throughput or cost. Selecting a fixed-point implementation requires trading off behavioral performance for available resources on the embedded processor. In addition, many of these filters must support calibration at processor startup or run time. Hence the selected implementation must meet the behavioral requirements for a set of digital filters. A common example application is audio processing to optimize cabin acoustics. In this case, the embedded audio processor queries the vehicle identification over the vehicle network then selects a bank of digital filters which are calibrated to provide optimal acoustics to the passengers.

This paper describes a workflow that applies Model-Based Design to develop an algorithm with selectable banks of fixed-point digital filters. In this workflow, the algorithm specification begins in floating point. Simulation test benches are then created to explore and verify the behavior. The algorithm is then converted to fixed-point in stages. The test benches are reused to verify correct behavior is maintained throughout the elaboration of the algorithm specification to fixed point. Automatic code generation is then applied to implement the algorithm in C code which takes advantage of processor specific intrinsic functions for fixed-point mathematics on an Analog Devices' Blackfin processor. The example workflow is described in the context of an acoustic tone controls application, but the approach can be applied to many applications requiring digital filters

which will be deployed to a fixed-point embedded processor.

INTRODUCTION

When developing applications containing fixed-point digital filters whose coefficients can vary over time, designers must carefully select filter structures and quantization scaling that ensures stable and correct system behavior. The negative impact of poorly designed filters ranges from unexpected sounds generated by speakers which can create dangerous driving scenarios by distracting or startling drivers, to incorrect detection of spark knock which can lead to destruction of engine components.

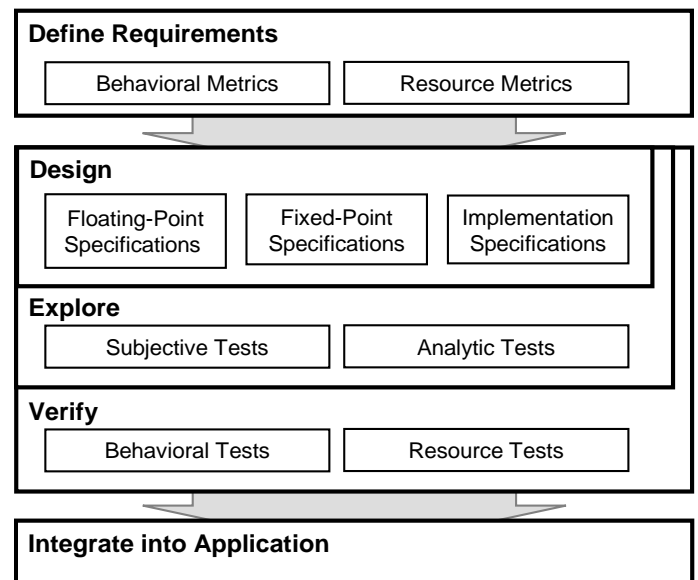


Figure 1. Typical algorithm development tasks.

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. This process requires a minimum of three (3) reviews by industry experts.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

ISSN 0148-7191

Positions and opinions advanced in this paper are those of the author(s) and not necessarily those of SAE. The author is solely responsible for the content of the paper.

SAE Customer Service: Tel: 877-606-7323 (inside USA and Canada)
 Tel: 724-776-4970 (outside USA)
 Fax: 724-776-0790
 Email: CustomerService@sae.org

SAE Web Address: <http://www.sae.org>

Printed in USA

This introduction will provide an overview of some typical development tasks and challenges required to prevent these types of failures in the field, followed by an overview of the tone controls algorithm which will be used to demonstrate development of a fixed-point audio algorithm using Model-Based Design.

AUDIO DSP DEVELOPMENT TASKS — The traditional DSP development process includes floating-point algorithm design with simulation, elaboration to fixed point, and implementation and evaluation in real time [1]. Additional development tasks are illustrated in **Figure 1**. Developing a DSP application often requires a combination of scripting, modeling, and code generation tools [1]. Designers are then challenged to apply these tools in a manner that minimizes effort and maximizes reuse throughout the development process.

Many automotive, aerospace, and communication companies apply Model-Based Design to identify issues early in the development process and reduce the time-to-market for embedded systems development. Some aspects of Model-Based Design include creating simulation models to specify and explore functional behavior, implementing these specifications through C code generation, and continuously testing and verifying the design against requirements. Some example applications of Model-Based Design include development of engine knock reduction systems [2], automotive audio playback mechanisms [3], and cochlear implants to improve hearing [4]. Applying Model-Based Design to develop audio applications enables designers to address common development challenges such as analysis of fixed-point tradeoffs, exploration of system behavioral performance, and rapid prototyping innovations.

Analyze Fixed-Point Tradeoffs — Designing digital filters requires balancing dynamic range and signal-to-noise ratio (SNR). In an audio tone controls application, this often requires that designers sacrifice SNR to provide headroom, particularly in the case where some spectral portion of the audio band is being boosted [5]. In applications where digital filter coefficients vary over time, digital filter morphing techniques must be applied to reduce audible transient distortions, which can produce disturbing acoustic effects. [6,7,8]. Designers select filter structures, precisions, and scale ordering to improve behavioral performance and reduce these effects [9]. DSP designers often begin their algorithm development in MATLAB®, a high-level language and interactive environment, and extend upon this environment to verify their designs [10].

Explore System Behavioral Performance — The final fixed-point design will be dependent on the overall system performance. For example, the behavioral performance of an automotive hands-free phone application will be highly dependant upon its ability to quickly adapt to changes in the acoustic environment.

Algorithm and environment modeling are often specified in Simulink®, a platform for multidomain simulation of dynamic systems. Simulink provides an interactive graphical environment, a customizable set of block libraries, and can be extended for specialized applications such as signal processing. Designers of hands-free applications often model and simulate the performance of echo and noise cancellation in Simulink to explore a wide range of design choices and reduce the time and expense of the design process [11].

Prototype Innovations — Typically acoustic engineers and software engineers come together to create market-differentiating features. Commonly, acoustic engineers prefer a graphical interface to explore innovations. These ideas must then be transferred to the software development team to prototype an implementation. The MATLAB and Simulink environment can be extended to rapidly design and prototype audio applications [12]. To reduce the time to implementation, acoustic engineers can use Simulink to specify the flow of the algorithm, and then apply Real-Time Workshop Embedded Coder™ C code generation technology to quickly implement the algorithm in real time. For example, acoustic engineers can graphically specify the flow of a surround-sound audio application and automate the generation and integration of C code into a real-time prototyping environment. After the best algorithm solution is identified, is it implemented on a DSP and committed to production hardware [13].

APPLY MODEL-BASED DESIGN TO AN AUDIO TONE CONTROLS ALGORITHM — This paper describes how scripting, modeling, and code generation are combined to develop a tone controls algorithm. A focus is placed on effectively elaborating design specifications from floating point to fixed point, creating reusable exploration and verification test benches, and automating analysis and collection of resource metrics.

The tone controls algorithm is similar to what can be found in a typical automotive radio as a three-band equalizer. The user can adjust the amount of lower-frequency content (bass), mid-range frequency content (mid), and higher-frequency content (treble). The user can select from a set of filter responses to increase the frequency content (boost), decrease the frequency content (cut), or maintain unity gain across the band (detent).

This algorithm was selected because it is functionally straightforward to understand, yet requires the designer to make typical fixed-point filtering tradeoffs. The requirement that the response for each filter bank be user-selectable at run time introduces additional fixed-point dynamics whose affects can be assessed through simulation. Although first- and second-order curves are commonly used in tone control applications, we chose fourth-order curves for this example to highlight

quantization and scaling techniques that can be applied to higher order filters.

Model-Based Design was applied to develop this algorithm. A floating-point behaviorally correct model was specified, explored, and verified based on a set of requirements. The design was elaborated to include fixed-point specifications, and then implementation specifications before being integrated into a standalone application. The development tasks addressed in this example are shown in **Figure 1**. A final set of design specifications would typically be achieved through iterations of these development tasks. For readability, these development tasks are discussed sequentially.

DEFINE REQUIREMENTS

Algorithm development often begins with both behavioral and resource utilization requirements. These requirements guide the development process and provide measures to evaluate performance.

DEFINE BEHAVIORAL METRICS — Magnitude response, phase response, Signal-to-Noise Ratio (SNR), and Spurious Free Dynamic Range (SFDR) are common metrics to evaluate the behavioral performance of applications with digital filtering components. For this tone control example, the design engineer is given a set of 15 bass, mid, and treble curves that users can select. The desired curves for this example are shown in **Figure 2**.

Along with these reference magnitude response curves, there is typically a requirement metric that the measured response be within some delta from the reference. For this example, the final fixed-point magnitude response must not deviate from the reference response by more than 0.2 dB. There is also a subjective behavioral requirement that the levels of boost and cut can be varied at run time without audible clicks or pops.

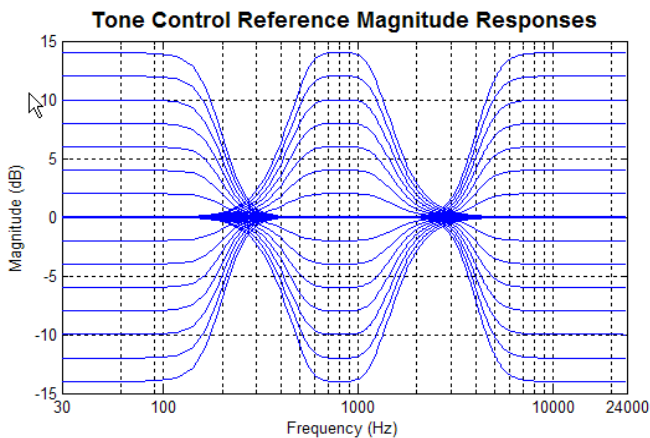


Figure 2. Tone control magnitude responses.

DEFINE RESOURCE METRICS — Resource requirements ensure that the algorithm implemented as a software component can be integrated into the larger application. Execution cycles, RAM/ROM usage, and stack usage are common metrics to evaluate resource usage on an embedded processor. In this example, the tone control algorithm is to be integrated into an application deployed on an Analog Devices' Blackfin® processor.

The level of detail for behavioral and resource requirements will vary based on existing application knowledge. In many cases, requirements are incomplete, conflicting, or unfeasible. Some of the following sections will show how Model-Based Design can identify holes in requirements early in the development process.

DESIGN BEHAVIORAL FLOATING-POINT ALGORITHM

New design tasks often begin in a floating-point environment because it allows the designer to focus on the behavior of the algorithm separate from fixed-point quantization effects. Model-Based Design enables a designer to create an executable specification for the algorithm behavior in floating point. This executable specification will be used to assess the behavioral requirements as well as to provide a basis for further elaboration to a fixed-point design. The specification for the tone control algorithm consists of model and parameterized data elements as shown in **Figure 3**.

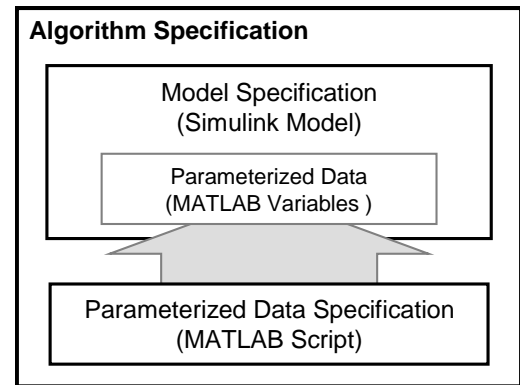


Figure 3. Algorithm specification.

SPECIFY ALGORITHM MODEL — To facilitate communication among different design teams, structural specifications are often expressed graphically while finer specifications may be expressed graphically or textually. Simulink® provides a graphical hierarchical environment to specify and simulate the design using models. Within Simulink, the designer can also specify models using state machines (Stateflow® software [14]) or textually (Embedded MATLAB™ subset [15]). In this example, the algorithm is graphically specified in Simulink.

A screen shot of the tone controls high-level model specification is shown in **Figure 4**. A frame-based stereo signal is passed through bass, mid, and treble tone controls. Filter selection signals are grouped as a multiplexed input to the model.

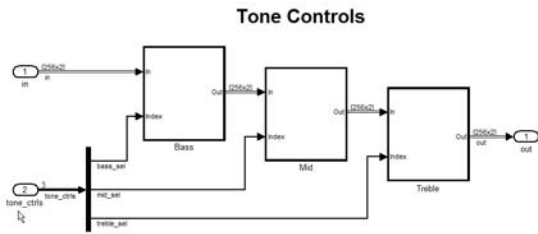


Figure 4. Tone controls, floating-point model.

The subsystems for each tone control contain further specifications for the flow of dynamically selected coefficients. The model specification of the Bass Tone Control is shown in **Figure 5**. The banks of filter coefficients, designed in the technical computing language MATLAB®, are stored in a matrix. The input selection signal is used to select the appropriate filter coefficients from the matrix. To address the requirement that there be no clicks or pops when switching between filters, the selected coefficient array is passed through a *Discrete Filter* block configured as a leaky integrator to smooth the transition between filter selections. The coefficients are then split into numerator and denominator coefficients and fed into a *Biquad Filter* block. The Mid and Treble subsystems are specified in a similar manner.

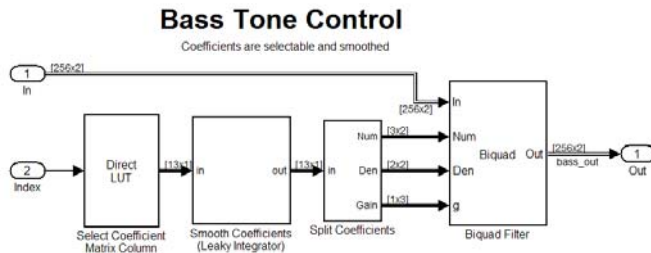


Figure 5. Bass Tone Control model.

SPECIFY ALGORITHM PARAMETERS — Algorithm parameters, such as filter coefficients are often designed using a technical computing language. A technical computing language enables designers to document the design flow of parameters in the algorithm. These parameters are then accessed by the graphical modeling and simulation environment.

In the tone controls example, the parameter design flow is specified in a MATLAB script which is automatically run at the start of the corresponding Simulink simulation. The filter banks are designed using the *parameq* function from Filter Design Toolbox™ [16]. In this example, the magnitude response curves are achieved using a family of fourth-order low pass shelving filters for

bass, fourth-order peaking filters for mid, and fourth-order high-pass shelving filters for treble. Each family of curves is packed into a matrix that is the parameter referenced in the Simulink structural specification.

EXPLORE FLOATING-POINT BEHAVIOR IN SIMULATION

Exploration test benches provide insight into the behavior of the design specifications. In this tone controls example, different test benches are used to explore the behavior analytically and subjectively.

ANALYTICALLY TEST BEHAVIOR — Analytical exploration test benches calculate metrics and provide graphical insight into the behavior of the algorithm design specification. In this tone controls example, the magnitude response is estimated during simulation. The structure of the magnitude response exploration test is shown in **Figure 6**.

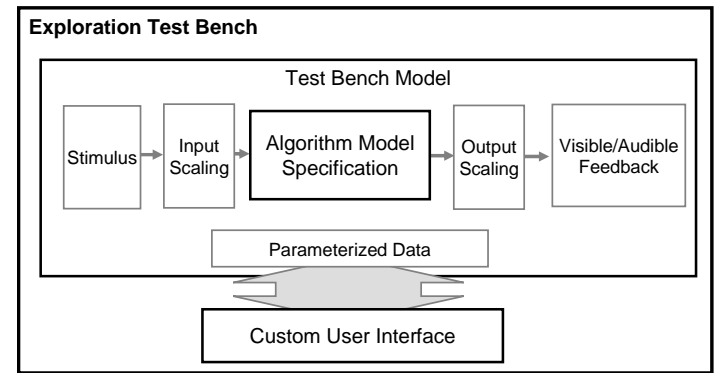


Figure 6. Exploration test bench.

As the algorithm design specification evolves, the test bench can be reused to test changes in the specification. The algorithm is specified in a separate model that is referenced by the test bench model – this provides for the separation of algorithm and test bench, which streamlines the design elaboration by making the test bench reusable throughout the design process. Interface subsystems connect the algorithm specification model to the stimulus and visualizations. These interface subsystems contain *Data Type Conversion* blocks that automate converting the test signals to the data type specified in the algorithm. This enables the same test bench to be used for both floating- and fixed-point algorithm specifications.

A custom user interface is programmed in MATLAB to interactively explore the behavior of the model during simulation [17]. The user interface provides pull down menus for each of the possible levels of boost or cut. A screen shot of the final magnitude response test bench is shown in **Figure 7**.

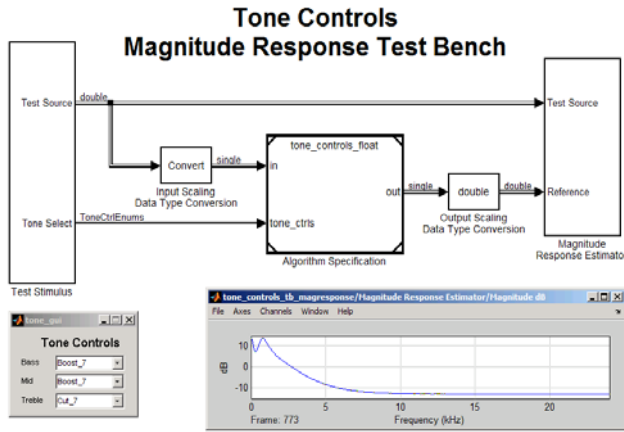


Figure 7. Magnitude response exploration test.

Analytical exploration provides insight into design requirements that can easily be overlooked in the rush to market and difficult to identify, debug, or change when deployed into a real-time application. For example, in this tone controls example, steady-state magnitude response estimations were correct, but the startup transients were unexpected. Further investigation showed that the states of the leaky integrators used to smooth coefficients were initialized to zero instead of the startup detent coefficients.

SUBJECTIVELY TEST BEHAVIOR — Subjective exploration of test benches enables designers to interactively explore system dynamics where analytical test benches do not already exist. For example, additional insight can be gathered by creating a test bench that connects the tone controls algorithm specification to live audio streaming through a sound card. This listening test bench is shown in **Figure 8**.

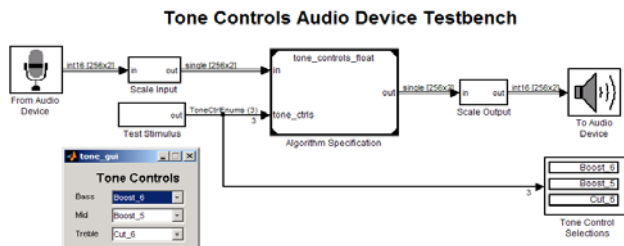


Figure 8. Audio device exploration test.

The structure of the listening test bench is similar to that of the magnitude response test bench, except the stimulus and visualization subsystems are replaced with *To Audio Device* and *From Audio Device* blocks from Signal Processing Blockset. These blocks access the DirectSound drivers in a Windows platform and OSS drivers on a Linux platform. These drivers can be extended to support additional driver formats such as DK, ASIO, and ALSA [18].

Subjective testing may identify missing design requirements or specifications as well as promote the creation of additional analytical tests. For example,

subjective testing of the tone controls algorithm through listening identified clipping at the output audio driver caused due to the high level of the input to the algorithm. This resulted in a modification to the algorithm specification to include both input and output level adjustment parameters. This is a simple example, but demonstrates how subjective interactive tests often catch dynamic phenomenon which is easily overlooked in aggressive development cycles. Design engineers can then assess if additional formal requirements and analytical tests should be created.

These exploration test benches will later be reused to verify correct behavior as the algorithm specification is evolved to fixed point. This enables the designer to explore, identify, and correct issues that may arise as fixed-point effects are introduced.

VERIFY FLOATING-POINT BEHAVIOR

Once the designer has identified a candidate algorithm specification through use of exploration test benches, the designer often creates a reusable verification test harness and performs a rigorous assessment of behavior against requirements. The exploration test benches enable the designer to explore design elaborations and evolve them through the design process, while the verification tests often calculate metrics that helps in qualifying the design as pass or fail when compared to desired specifications. The verification test harness helps with continuous verification of the design, thus identifying errors that could lead to significant deviation from the desired reference.

In this tone controls example, the magnitude response exploration test bench was modified to create a formal verification test. A MATLAB script is executed to run the test, estimate the transfer function, compare the estimation against the reference response, and report maximum, average, and standard deviation error statistics. This verification test is used to verify the floating-point design is correct and will continue to be applied throughout the conversion to fixed-point to ensure that design elaborations continue to meet the original design specification.

ELABORATE ALGORITHM WITH FIXED-POINT SPECIFICATIONS

Fixed-point designs are often preferable to floating-point designs for products that require reduced cost and/or reduced power consumption. Elaborating an algorithm from floating point to fixed point often requires modifications to the original design as well as the selection of quantization attributes. This elaboration process can require significant engineering effort. For example, if several banks of filter coefficients are transitioning at the same time, it can be difficult to mathematically express the impact of transients in the

system. Simulation of fixed-point effects can reduce engineering effort by providing insight into system dynamics early in the development process.

Designers subscribe to different fixed-point conversion best-practices that suit their needs and styles. A common approach is to independently convert portions of the algorithm to fixed point while keeping the rest of the design in floating point. These portions are then merged to specify and test the entire fixed-point system. In this tone controls example, the following tasks are performed to incrementally convert to fixed point:

- Specify fixed-point I/O and add dynamic headroom scaling
- Convert coefficients to fixed point
- Convert coefficient slewing to fixed point
- Convert filter internals to fixed point

Typically designers iterate through these tasks to identify the least amount of precision and mathematical operations that achieve the behavioral requirements. The following sections describe a single iteration through these tasks. Note that intermediate stages in the conversion process are a hybrid of floating- and fixed-point specifications. After each conversion, the verification test (described in the previous section) is applied to continuously verify that the result of the each conversion stage meets within the behavioral requirements. The results of these verification tests are shown in **Table 1**.

Stage of Fixed-Point Conversion	Magnitude Response Error		
	Avg (dB)	STD (dB)	Max Abs (dB)
Floating-Point Reference	0.00001	0.0018	0.0441
Fixed-Point Headroom Scaling	-0.0003	0.0038	0.0542
Fixed-Point Coefficients	-0.0012	0.0112	0.1206
Fixed-Point Coefficient slew	-0.0060	0.0139	0.0895
Fixed-Point Filter Internals	-0.0018	0.0141	0.0933

Table 1. Fixed-Point Conversion Verification Results

SPECIFY FIXED-POINT INPUT AND OUTPUT AND ADD DYNAMIC HEADROOM SCALING — The most common tradeoff when transitioning from floating- to fixed-point is balancing dynamic range for SNR. This example assumes that the software interface is defined as fixed point with 16-bit word length and 15-bit fractional length, in order to integrate with other software components in the system. The maximum boost level for each tone control bank is 14 dB. To provide enough headroom for the boost, the input could be attenuated by the maximum gain, but this approach adversely affects the SNR as the bits are unnecessarily thrown away if the user doesn't choose the maximum boost settings, such as when the tone control settings are set to detent or cut. The approach used in this example is to dynamically adjust the input and output levels based on the settings of the tone controls by adding pre- and postscaling as shown in **Figure 9**.

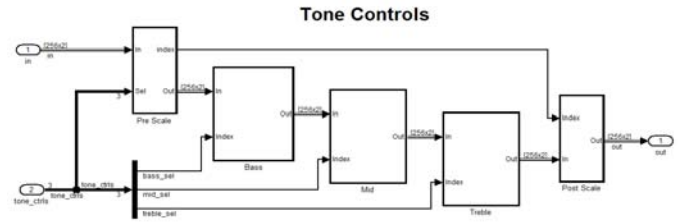


Figure 9. Tone controls fixed-point model.

The model of the Pre Scale subsystem is shown in **Figure 10**. A lookup table is used to select a scale value based on the maximum gain corresponding to any tone control request as chosen by the user. (Note that the maximum gain corresponds to the minimum filter index for this example.) This scale value is then passed through a *Discrete Filter Block* configured as a leaky integrator. This filtering will smooth gain transitions between different filter selections.

The attenuation applied in the Pre Scale subsystem is compensated for in the Post Scale subsystem in a similar dynamic manner.

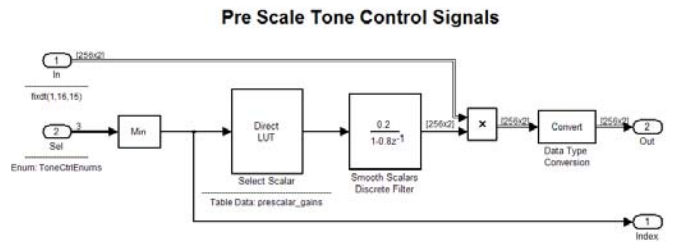


Figure 10. Tone controls Pre Scale model

Automatic scaling of the pre and post scalar look up table values is accomplished using *fi* object constructor from Fixed-Point Toolbox [19]. **Figure 11** shows an excerpt of MATLAB code used to create and quantize the pre- and postscalars.

```
>> prescalar_dB = -(14:-2:0)';
>> prescalar_fract = 10.^(prescalar_dB/20);
>> prescalar_gain = fi(prescalar_fract,true,16,15);
>> postscalar_dB = -prescalar_dB;
>> postscalar_fract = 10.^(postscalar_dB/20);
>> postscalar_gain = fi(postscalar_fract,true,16)

postscalar_gain =

    5.0120
    3.9810
    3.1624
    2.5120
    1.9954
    1.5850
    1.2590
    1.0000

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 12
```

Figure 11. MATLAB code for fixed-point scalars.

The above code demonstrates how designers can explicitly specify fixed-point attributes or request automatic selection of fraction length. The following arguments are passed to the *fi* constructor for the `prescalar_gain`: values, signed, word length, and fraction length. In this case the word length is set to 16 bits and the fraction length is explicitly set to 15 bits to represent a dynamic range of -1 to almost 1. For the `postscalar_gain`, the fraction length is omitted from the constructor argument list and the fraction length is automatically selected as 12 bits to ensure that all values in the array can be represented.

The `prescalar_gain` and `postscalar_gain` fixed-point variables are referenced by a lookup table in the Simulink model configured to inherit these values and data types.

The same analytic and subjective exploration and verification tests used previously were reused to explore the effects of fixed-point pre- and postscaling. The results of the verification test are included in **Table 1**.

CONVERT COEFFICIENTS TO FIXED-POINT — The designer must select coefficient precisions and filter structures that maintain the desired behavioral requirements. The negative impact of poor coefficient quantization can range from an unexpected frequency response to an unstable filter.

To reduce the number of system tests during this exploration, it is useful to narrow the exploration space by assessing the performance of several fixed-point coefficient specifications. The results of this exploration can then be used as a starting point to model the effects of fixed-point filter coefficients and assess the behavioral impact through simulation.

Explore fixed-point coefficients and filter structures — For floating-point audio filters the selected biquad filter structure often has negligible behavioral performance effects. However, for the fixed-point filters, the selected structure can significantly impact the behavior. The designer often starts with a common filter structure or a filter structure whose performance is known on the desired target processor. For this tone controls example, the filter structures, coefficient bits, scale value constraints, and p-norms explored are shown in **Table 2**.

Fixed-Point Attribute	Values Explored
Coefficient Bits	32, 16
Filter Structures	Direct Form 1, Direct Form 2, Direct Form 2 Transposed
Scale Value Constraint	None, Unit, All Ones
P-Norm	Linf, linf

Table 2. Exploring Fixed-Point Filter Attributes

Filter Design Toolbox was used to specify and assess the fixed-point filter attributes shown in **Table 2** [20]. Note that the scale value constraints of none and unit are standard options supported by Filter Design Toolbox.

For this example, a customized scaling technique of 'All Ones' was added to explore designing filters with only numerator and denominator coefficients, but no scale terms. Example MATLAB code demonstrating how to design and create a fourth-order fixed-point parametric equalizer digital filter with a 10 dB boost at 1 kHz, and 5 dB boost corresponding to a bandwidth of 200 Hz. is shown in **Figure 12**.

```
>> d = fdesign.parmeq(...
    ['N','F0','BW','Gref','G0','GBW'],...
    4, 1000, 400, 0, 10, 5, 48000);
>> opts = fdopts.sosscaling(...
    'ScaleValueConstraint','none');
>> hfilt = design(d,'butter',...
    'FilterStructure', 'df2sos', ...
    'SOSScaleNorm', 'Linf',...
    'SOSScaleOpts', opts);
>> set(hfilt,...
    'Arithmetic', 'fixed',...
    'CoeffAutoScale', true,...
    'CoeffWordLength', 16);
```

Figure 12. Design fixed-point filter.

The `measure` method of the digital filter object `hfilt` tests the filter to determine the actual values of the filter specification such as the quantized gains and center frequency. **Figure 13** shows an example script demonstrating how to call the `measure` command with an excerpt of the results returned.

```
>> m = measure(hfilt,'Gpass',5,'Gstop',5)
m =
Passband Bandwidth : 499.496 Hz
Flow              : 780.603 Hz
Fhigh             : 1.2806 kHz
Reference Gain    : -0.00041828 dB
Center Frequency Gain : 10.0041 dB
```

Figure 13. Measure fixed-point filter.

In the above code, `measure` is configured to return the measured lower and upper passband frequencies corresponding to 5 dB gain along with a default set of measurements. Since the filter is configured with a center frequency of 1000 Hz and bandwidth of 400 Hz, the expected values are 800 Hz and 1200 Hz. The measurement shows there is slight deviation attributed to coefficient quantization.

For the tone controls example, these techniques were extended to design and measure banks of filters. A filter bank class was created using the object-oriented programming capabilities from MATLAB. Object methods were created to leverage the fixed-point filter design and measurement capabilities of Filter Design Toolbox to operate on groups of filters. Coefficient precisions were selected to represent the dynamic range of all of the filters included in the bank.

A MATLAB script was created to automate the design of filter banks for all combinations of fixed-point attributes shown in **Table 2**. The script also measured the gain and bandwidth for each filter within the filter bank. Finally, the script created an HTML report that

summarized the fixed-point design specifications and corresponding measurements.

This report was used to assess which fixed-point filter banks passed or failed the magnitude response requirement of 0.2 dB deviation, and to assess which filters should be used as the starting point in the fixed-point design. An example table from the treble filter bank coefficient comparison report is shown in **Table 3**.

Coeff Bits	Structure	Scale Value Constraint	PNorm	G0 Err Max (dB)	Gref Err Max (dB)	BW Err Max (Hz)
32	df1sos	none	Linf	+0.000000	+0.000000	+23.448955
32	df1sos	unit	Linf	+0.000000	+0.000000	+23.448955
32	df1sos	allones	Linf	+0.000000	+0.000000	+23.448955
16	df1sos	none	Linf	+0.000941	+0.007773	+24.457020
16	df1sos	unit	Linf	+0.000941	+0.007773	+24.457020
16	df1sos	allones	Linf	+0.002020	+0.010051	+24.067540
16	df2sos	none	Linf	+0.014013	+0.015475	+31.605114
16	df2sos	unit	Linf	+0.013620	+0.014603	+33.575422
16	df2sos	allones	Linf	+0.001982	+0.008889	+24.915231
16	df2tsos	none	Linf	+0.000941	+0.007773	+24.457020
16	df2tsos	unit	Linf	+0.000941	+0.007773	+24.457020
16	df2tsos	allones	Linf	+0.002020	+0.010051	+24.067540

Table 3. Excerpt from Treble Coefficient Report

The treble filter bank report identified that both 32-bit and 16-bit coefficients were sufficient to produce the required magnitude response. Similarly, the bass report identified that 32 bits were required for that filter bank.

The selected structures and scaling for all three-filter banks are shown in **Table 4**. Note that the selected fixed-point filters are intentionally kept heterogeneous to demonstrate the ability to work with different filter types in the same design.

	Bass	Mid	Treble
Coefficient word length	32	16	16
Numerator fraction length	29	14	14
Denominator fraction length	30	14	14
Scale value fraction length	NA	11	12
Filter structure	DF2T	DF2	DF1
Scale value constraint	All Ones	None	Unit
Number of scale values	0	3	1
P-norm	Linf	linf	Linf

Table 4. Tone Control Coefficient Quantization

Based on the system behavior or the resources used on the processor, the designer may go back to this stage to select another set of fixed-point coefficient specifications that improve behavior or use fewer resources. In such cases, the MATLAB scripts already created to iterate through earlier design explorations could be extended to check for resource constrains and other optimization criteria. For simplicity, these coefficient specifications will be used throughout the rest of this paper.

Verify Fixed-Point Coefficient Behavior — It is preferable to assess the fixed-point effects of coefficients independent of fixed-point effects introduced within filter calculations. In this example, the Simulink model is

modified to integrate the fixed-point coefficients while using floating point for the filter internal calculations.

The original floating-point specification model combined all the coefficients and gain scalars into a single matrix. Because the data types of a matrix must be the same for all elements, the designer must create separate variables for coefficients and scalars of different data types or choose another mechanism of packing data. For the tone controls example, the word length of all the coefficients and scalars is the same, so the values can be stored as a matrix of integers with the same number of bits as the word length. When unpacking the matrix in numerator, denominator, and gains, the integer data is re-interpreted as fixed-point data types as shown in **Figure 14**.

During the initial steps of the fixed-point conversion process, *Data Type Conversion* blocks are used to isolate the coefficient slewing from the filter calculations (specified in floating point). This enables independent evaluation of the effects of any fixed-point conversions incrementally throughout the process. After each step, the behavioral verification test harness created in the floating-point behavioral design stage is reused to verify that the fixed-point parameter specification continues to meet the behavioral requirements.

One example of this approach happens during the fixed-point conversion process. A separate fixed-point conversion test bench is used to explore different fixed-point designs as the tone controls system is evolved through the design. After a candidate design is obtained, before it is locked down as the next step, it is verified with the behavioral test harness to ensure it meets the original design specification.

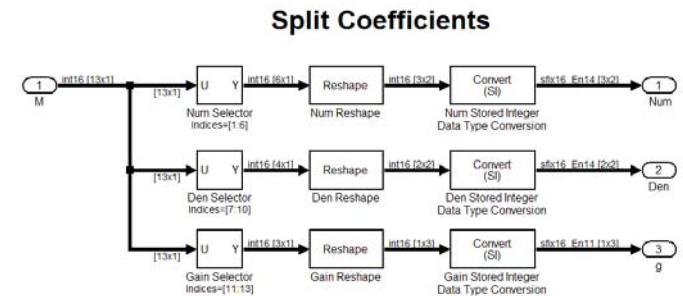


Figure 14. Split integer coefficients to fixed point.

CONVERT COEFFICIENT SLEWING TO FIXED-POINT — The mathematics to slew the filter coefficients can introduce additional quantization noise that may require further careful analysis and verification before conversion to fixed point. In this design, we settled with a simple *Discrete Filter* block that functions as a leaky integrator. The fixed-point settings for the slewing filter were directly entered into the *Discrete Filter* dialog. The exploration and verification test benches were used to verify that adding fixed-point mathematics for slewing the

coefficients did not introduce any behavioral errors. The results of the verification test are shown in **Table 1**.

If a higher order or more complicated slewing mechanism was chosen, direct entry of fixed-point settings may be challenging. Tools exist to help the designer specify fixed-point settings. The application of such tools will be demonstrated in the next section to help convert the filter internals of the fourth-order tone control filters.

CONVERT FILTER INTERNALS TO FIXED-POINT —

The designer must select fixed-point settings for filter internals such as states, product, and accumulator sizes. Similar to the coefficient investigation, fewer bits often require fewer resources. Automated scaling tools, such as the Fixed-Point Tool, help the designer select fractional lengths that minimize quantization effects [21]. The Fixed-Point Tool allows the designer to independently override subsystems to use floating-point mathematics, log minimums, maximums, and overflows, then use this logged data to recommend fractional lengths. The designer can then choose to accept these proposed scaling values or directly specify the scaling based on engineering knowledge.

In the tone controls example, *Data Type Conversion* blocks were added to the model to enable easy switching of the *Biquad Filter* block between floating-point and fixed-point mathematics during the conversion process. Specification of the fractional lengths can be directly specified in the dialog of the *Biquad Filter* block as shown in **Figure 15**. However, optimal selection of fraction length can be challenging because the optimal value will be based on system inputs. For this example, a fixed-point test was created to cycle through typical edge cases including pass through at maximum amplitude and maximum boost of all filter banks at attenuated amplitude.

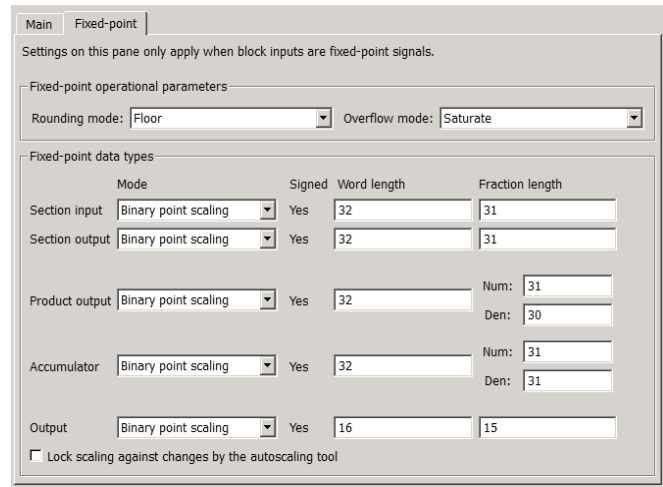


Figure 15. Fixed-point settings for biquad filter.

32-bit and 16-bit word lengths (WL) for each section were investigated. The Fixed-Point Tool was used to recommend fraction lengths (FL) based on the test signals. The recommended fraction lengths were accepted and the results of this scaling are shown in **Table 5**. The fully specified fixed-point model was then tested against the original magnitude response test, and the results are shown in **Table 1**.

	Bass (DF2T)		Mid (DF2)		Treble (DF1)	
	WL	FL	WL	FL	WL	FL
Input	16	17	16	17	16	17
Section Input	32	33	32	33	16	17
Section Output	32	33	32	33	16	17
Product (Num)	32	32	32	28	32	32
Product (Den)	32	30	32	30	32	30
Accumulator (Num)	32	32	32	28	32	32
Accumulator (Den)	32	32	32	28	32	32
Output	16	17	16	17	16	17

Table 5. Fraction Lengths from Fixed-Point Tool

Based on insight gained through simulation and implementation testing, the design engineer may return to this task and assess the performance of other fixed-point tradeoffs. For example, selecting different fractional lengths for intermediate computations may require extra implementation cycles to shift and align data. If the implemented design requires too many cycle resources, the designer may go back to the simulation environment and assess the behavior of homogeneous fraction lengths.

ELABORATE ALGORITHM WITH IMPLEMENTATION SPECIFICATIONS

Once the desired simulation behavior is achieved, C code can be generated based on the algorithm specification model. Automatic code generation enables designers to quickly deploy their ideas to hardware to continue verifying performance on a real-time embedded system.

Real-Time Workshop Embedded Coder enables designers to generate C code from a Simulink model. Specifications can be added to the model to customize the generated code to ease the code review and integration process. Embedded IDE Link VS (for Analog Devices VisualDSP++) extends this capability and provides additional optimization and verification features [22].

In the tone controls example, the model was configured to generate code for the Analog Devices' Blackfin processor and leverage its fixed-point intrinsic functions. The model was also configured to generate an HTML report and enable bidirectional traceability between the model and the generated code. For example, **Figure 16** demonstrates that right clicking on the leaky integrator enables the designer to navigate to the corresponding code.

Select and Slew Coefficients

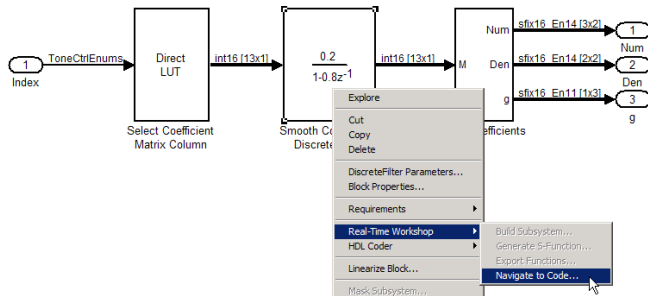


Figure 16. Navigate from filter block to code.

Excerpts from the generated code report are shown in **Figure 17** and **Figure 18**. Notice that the comments in the generated code include hyperlinks that can be used to navigate back to the corresponding block in the model. Also note that the function call `bf32x_sub_s32_s32_s32_sat` is used to implement the fixed-point saturated arithmetic. This function is inlined and mapped to the function `add_frlx32`, which is a processor optimized C callable routine provided by Analog Devices.

```
/* DiscreteFilter: '<S13>/Smooth Coefficients Discrete Filter' */
tone_controls_DWork.SmoothCoefficientsDiscreteFil_k[i_0] = (int16_T)
(bf53x_sub_s32_s32_s32_sat(rtb_SelectColumn[i_0] << 14,
bf53x_sh_s32_s32_s1_sat(-26214 *
tone_controls_DWork.SmoothCoefficientsDiscreteFilte[i_0], 2)) >> 17);
rtb_SmoothCoefficientsDiscreteF[i_0] = (int16_T) (bf53x_sh_s32_s32_s1_sat
(26214 * tone_controls_DWork.SmoothCoefficientsDiscreteFil_k[i_0], 3) >>
17);
```

Figure 17. Code for a filter with saturated arithmetic.

```
/*
 * operation:      s32 + s32
 * saturation:     Yes
 * rounding mode:  N/A
 */
inline int32_T bf53x_add_s32_s32_s32_sat(int32_T a, int32_T b)
{
    return add_frlx32(a, b);
}
```

Figure 18. Mapping to processor-specific intrinsic.

VERIFY BEHAVIOR AND RESOURCES ON TARGET

Designers commonly perform unit testing on the generated code to ensure that the compiled behavior executing on the target is consistent with simulation results, and to ensure that the code meets execution timing resource requirements. Once the behavior and resources have been verified, the code will be integrated with the other software components in the application.

TEST BEHAVIOR — Processor-in-the-loop (PIL) testing is a technique that ensures that the code compilation process has not introduced numerical inconsistencies to the simulation results. During PIL testing, the host simulation environment performs step-by-step co-

simulation with code executing on the target processor. In this example, Embedded IDE Link VS enables PIL testing between Simulink and the generated code running on the Blackfin DSP. A variant of the original simulation behavioral test was used to verify behavior of the generated code running in PIL mode.

TEST RESOURCES — Execution profiling, stack profiling, and RAM/ROM analysis are common techniques to verify that the generated code meets resource requirements on the target. RAM/ROM usage can be obtained from the memory map file generated during the build process. Embedded IDE Link VS enables automation for collecting and reporting execution time and stack usage.

IMPROVING PERFORMANCE — Based on analysis of the resource usage statistics, the designer will either deem the performance as acceptable or continue to elaborate on the design. Typically during this elaboration, the designer will trade off behavioral performance versus resource usage. For example, in this case, three different combinations of filter topologies and fixed-point attributes have been used to illustrate flexibility in specifying fixed-point filters. The designer may return to the model and make tradeoffs in the topology and fixed-point settings, or specify a custom implementation topology built from core Simulink or Embedded MATLAB blocks. In other cases, the designer may decide to integrate hand written C-callable assembly routines with Model-Based Design [23].

INTEGRATE INTO APPLICATION

Once the generated code has been verified to meet the behavioral and resource requirements, it can be integrated into a real-time application.

Commonly, designers want to integrate generated code into an existing application. This enables them to leverage their current software platform and easily integrate components developed using Model-Based Design. In some cases, designers will hand-integrate the generated files into existing projects [24]. In other cases, to ease the management of the compilation and the integration process, a designer will want to integrate a library into an existing project. Embedded IDE Link VS provides the automatic generation of a Visual DSP++ library, which can be easily be included into an existing project [25].

CONCLUSION

Designing a fixed-point system requires trading off between behavior, resource utilization, and development effort. Better behavioral performance typically requires mitigating higher resource utilization, which in turn increases development time.

This paper describes how Model-Based Design can be applied to design applications with fixed-point digital filters whose coefficients can be changed at run time on a fixed-point embedded processor. We present an example workflow that includes specifying a floating-point reference design, exploring the behavioral design, creating a reusable verification test harness, elaborating to include fixed-point components, generating fixed-point code, and implementing this code on an Analog Devices' Blackfin processor, while continuously assessing performance throughout the design process.

This paper emphasizes creating reusable test benches to explore and continuously verify the algorithm specification through each elaboration stage of the design. Exploration test benches enable interactive experimentation intended to assess the response to transients and systems dynamics for different design specifications. At each design stage, from floating point through fixed point, we reuse a verification test to rigorously assess behavior of the candidate algorithm against requirements.

We applied a combination of scripting and model elaboration to evolve the algorithm specification from floating point to fixed point. This paper describes design tasks such as elaborating the model specification to include dynamic scaling, scripting in MATLAB to assess filter structures based on coefficient quantizations, and applying autoscaling tools to select filter internal fraction lengths. The final stage of the design involves generating code for the algorithm and deploying on an Analog Devices' Blackfin processor.

The approach described in this paper lays the foundation for a design process that will enable designers to quickly identify deficiencies, make changes, and automate performance assessment, thus decreasing design iteration time.

REFERENCES

- [1] "DSP Algorithm Development Tools," DSP & Multimedia Technology, November 1993:
www.bdti.com/articles/info_dspmt93algorithm.htm
- [2] "Hitachi Accelerates the Development of Engine Knock-Reduction Systems," The MathWorks, January 2005:
www.mathworks.com/company/user_stories/userstory8438.html
- [3] Schubert, Peter J., Vitkin, Lev, and Braun, David, "Model-Based Development for Event-Driven Applications using MATLAB: Audio Playback Case Study," Systems Engineering, January 2007:
<http://delphi.com/pdf/techpapers/2007-01-0783.pdf>
- [4] Wilber, Jack, "The Sound of Innovation at Cochlear Limited," The MathWorks News & Notes, October 2006:

www.mathworks.com/company/newsletters/news_notes/oct06/cochlear.html

- [5] Datta, Jayant and Jensen-Link, Leslie, "Study of Equalization Algorithms on a Fixed-Point DSP: Impact on Audible Quality, Headroom, and SNR," Audio Engineering Society 108th Convention, February 2000.
- [6] Mourjopoulos, J. N., Kyriakis-Bitaros, E. D., and Goutis, C. E., "Theory and Real-Time Implementation of Time-Varying Digital Audio Filters," Journal of Audio Engineering Society, Vol.38, No.7/8, July/August 1990.
- [7] Zoelzer, Udo, Redmer, Bernd, and Bucholtz, Jochen "Strategies for Switching Digital Audio Filters," Audio Engineering Society 95th Convention, October 1993.
- [8] Clark, Rob, Ifeachor, Emmanuel and Rogers, Glenn, "Filter Morphing – Topologies, Signals, and Sampling Rates," Audio Engineering Society 113th Convention, October 2002.
- [9] Zaucha, David, "Importance of Precision on Performance for Digital Audio Filters," Audio Engineering Society 112th Convention, May 2002.
- [10] Magee, David P., Ph.D., "MATLAB Extensions for the Development, Testing and Verification of Real-Time DSP Software," Texas Instruments, 2005 Design Automation Conference, June 2005:
http://videos.dac.com/42nd/papers/36_2.pdf
- [11] Eslinger, Greg and Dixon, John, "Dynamic Adjustment of System Parameters Improves Echo and Noise Cancellation," Texas Instruments, December 2006: <http://focus.ti.com/lit/wp/spry094/spry094.pdf>
- [12] Schillebeeckx, P., Paterson-Stephens, I. and Wiggins, B., "Using MATLAB/Simulink as an implementation tool for Multi-Channel Surround Sound," Proceedings of the 19th International AES conference on Surround Sound, Germany, pp. 366-372, June 2001.
- [13] Philips Consumer Lifestyle (Philips) Develops One-Piece Surround Sound System with MathWorks Tools, The MathWorks, April 2008:
www.mathworks.com/company/user_stories/userstory17418.html
- [14] Stateflow 7.2 Design and simulate state machines and control logic, The MathWorks, 2008:
www.mathworks.com/products/stateflow/
- [15] The Embedded MATLAB Language Subset, The MathWorks, 2008:
www.mathworks.com/products/featured/embeddedmatlab/
- [16] Filter Design Toolbox 4.4, Design and analyze fixed-point, adaptive, and multirate filters, The MathWorks, 2008: www.mathworks.com/products/filterdesign/
- [17] Fillyaw, Chris, Friedman, Jonathan, and Prabhu, Sameer M., "Creating Human Machine Interface (HMI) Based Tests within Model-Based Design," Society of Automotive Engineers 2007-01-0780, January 2007.

- [18] Rogers, Alec, "Customized Audio Driver Support (Using ASIO, ALSA, etc. with Signal Processing Blockset Audio Device Blocks)," February 2008:
www.mathworks.com/matlabcentral/fileexchange/18599
- [19] Fixed-Point Toolbox 2.3, Design and execute fixed-point algorithms and analyze fixed-point data, The MathWorks, 2008:
www.mathworks.com/products/fixed/
- [20] Losada, Ricardo, "Digital Filters with MATLAB," May 2008:
www.mathworks.com/matlabcentral/fileexchange/19880
- [21] Erkkinen, Tom, "Fixed-Point ECU Development with Model-Based Design," SAE 2008-01-0744, January 2008.
- [22] Embedded IDE Link VS 2.1 for Analog Devices VisualDSP++, The MathWorks, 2008:
www.mathworks.com/products/visualdsp/
- [23] Corless, Mark and Reddy, Vinod, "Integrating Your Processor-Specific Code with Model-Based Design of Embedded Systems," GSPx 4th International Signal Processing Conference, October 2006.
- [24] Donovan, Mike, "Deploying Simulink Designs on Your DSP: An Accelerated Approach to Custom Implementation," MATLAB Digest, January 2006.
- [25] Ananthan, Arvind, "Integrating Simulink Model with VisualDSP++ Project," November 2008:
www.mathworks.com/matlabcentral/fileexchange/22243

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.