Implementing Real-Time MIDI Music Synthesis Algorithms, ABS/OLA, and SMS for the TMS320C32 DSP

APPLICATION REPORT: SPRA355

Authors: Susan Yim (yim@hc.ti.com)
Yinong Ding (ding@hc.ti.com)
E.Bryan George (george@hc.ti.com)
DSP Research and Development Center
P.O. BOX 655474, MS 446
Dallas, TX 75265

Digital Signal Processing Solutions
January 1998

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Implementing Real-Time MIDI Music Synthesis Algorithms, ABS/OLA, and SMS for the TMS320C32 DSP

Abstract

This application report describes a real-time MIDI music synthesis system using a low cost digital signal processor (DSP) such as the Texas Instruments (TI™) TMS320C32 in a PC environment. The system consists of a MIDI device with a MIDI interface, an IBM compatible personal computer, and a TMS320C32 development board where the core of the music synthesis engine resides.

The MIDI device generates music synthesis commands. The host computer handles communications between the MIDI device and the DSP, where music samples are synthesized using sinusoidal modeling-based music synthesis techniques. Two sinusoidal model-based music synthesis algorithms are discussed: the Spectral Modeling System (SMS) and the Analysis-By-Synthesis and OverLap-Add (ABS/OLA).

This report details the interoperability of the three processes, the control and handshake for data flow, the input file structure, and the mechanism used for synthesis.

Keywords: DSPRDC, MIDI, Music Synthesis, Digital Audio Signal Processing, TMS320C32, DSP

Product Support on the World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Introduction

Music synthesis plays an important role in multimedia applications, modern entertainment, and professional music systems. Popular music synthesis techniques used in the market include

- Sampling/wavetable synthesis
- Frequency modulation (FM)
- Physical modeling

Most PC sound cards in the market use a combination of FM and sampling technology where only limited control capabilities are provided.

In the REALMS (Real-time Environment and ALgorithms for Music Synthesis) project, we aim to contribute to this rapidly growing market by combining TI's proprietary music synthesis algorithms with TI's state-of-the-art DSP technologies. We designed this MIDI-driven real time music synthesis system to demonstrate our music synthesis algorithms using a TI TMS320C3X digital signal processor as a music synthesis engine. This system is intended to provide a foundation and a reference design for future development of a music synthesis card or single music synthesis IC, as well as for low cost fixed-point implementation of music synthesis engines for the moderate quality music synthesis market.

One challenge in implementing the MIDI-driven real time music synthesis system is to integrate MIDI signal with the music synthesis engine and the necessary Input/Output control. The other challenge is to synthesize music samples using our music synthesis algorithms in real time on the TI TMS320C3X digital signal processor.

The MIDI-driven real-time music synthesis system consists of three major components:

- MIDI device that generates the MIDI data stream
- TMS320C3X development card
- IBM compatible PC (a sound generator that receives MIDI data stream)

The communications or handshake between the MIDI device and DSP are controlled by the host computer running under the Windows 3.1/95 operating environment. Hence, the MIDI device is the initiator, the host computer is the recipient of the MIDI data which interprets the received MIDI messages and drive the DSP-based synthesis engine accordingly.

The Music Instrument Digital Interface (MIDI) is simply a data communications protocol for music equipment, computers and software to exchange information and control signals to synthesize music [1]. It symbolically describes instructions for synthesizers to generate sounds. The MIDI data stream is a unidirectional asynchronous bit stream at 31.25 kbits/sec with 10 bits transmitted per byte (a start bit, *8* data bits, and one stop bit). A MIDI message is made up of an eight-bit status byte which is generally followed by one or two data bytes. For example, a MIDI message can include the type of instruments, the notes, the loudness or effects and many other types of control information that the user has selected/issued. The quality of the actual sound is dependent on the algorithms that the synthesizer adopts. Some of the popular methods that are currently used in the market are the sampling/wavetable synthesis, FM synthesis, physical modeling using waveguides. Another promising technology is the sinusoidal model-based music analysis and synthesis. We have been investigating and evaluating two of the approaches, the Spectral Modeling System (SMS) and the Analysis-By-Synthesis and OverLap-Add (ABS/OLA). Currently, we have implemented a signal-note synthesizer in real-time using either the ABS/OLA or SMS synthesis algorithm.

The paper is organized as follows:

- *System Components* describes the real-time MIDI driven music synthesis system.
- *MIDI Interface to Host Computer* introduces the MIDI keyboard and Opcode Note1/++ external MIDI interface.
- *Description of the Microsoft Windows MIDI Application Software* describes the MIDI application software.
- *Music Synthesis Engine* describes how music synthesis is performed on the TMS320C32 DSP.
- *Music Synthesis Algorithms* describes the music synthesis algorithms using sinusoidal modeling.
- *Discussion and Summary* includes results and discussion.

System Components

Our MIDI-driven real-time music synthesis system consists of three major components:

- MIDI device

  We use the Yamaha W7 MIDI keyboard with an external MIDI interface "*Note/1++"* from Opcode.

- TMS320C3X development board

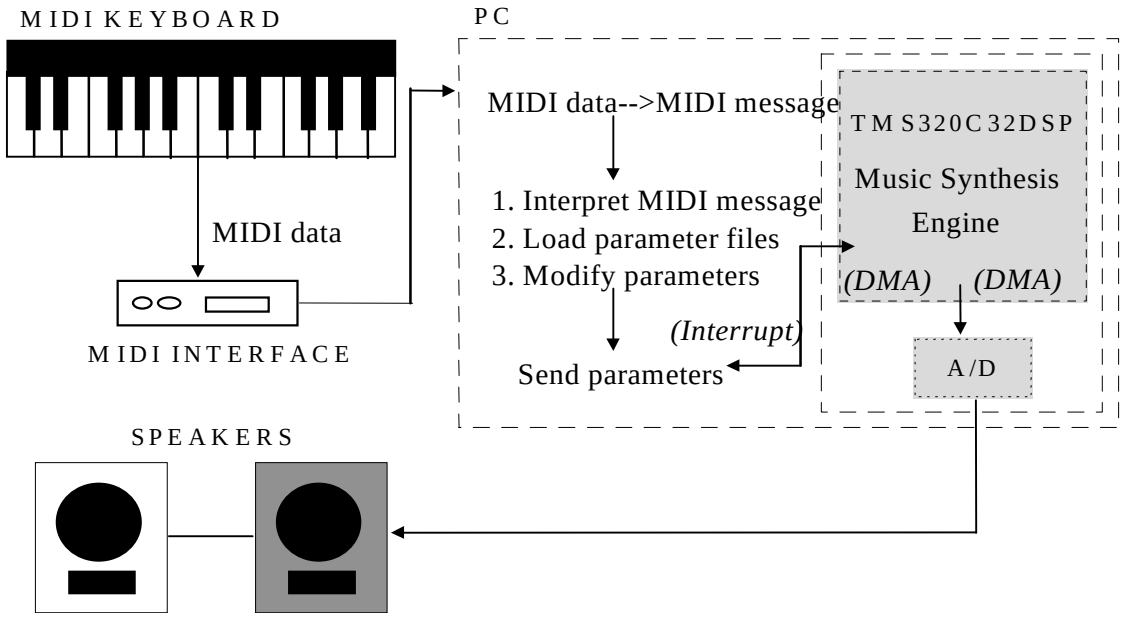
  Tiger32EVM board with the TMS320C32 low cost DSP operating as the music synthesis engine

- Host machine

  IBM compatible PC (Pentium)

The MIDI device initiates the music synthesis process by generating MIDI data. The host computer receives MIDI data, interprets the received MIDI messages, and drives the DSP-based synthesis engine accordingly. That is, the MIDI device communicates with the DSP via the host computer under the Windows 3.1/95 operating environments. Figure 1 shows the system setup.

Figure 1. MIDI Music Synthesis System



MIDI Interface to Host Computer

In this system, the MIDI keyboard generates a stream of serial MIDI data. The fundamental function of a MIDI interface is to convert the serial MIDI data to parallel data bytes, perform the necessary buffering (along with the required start and stop bits) and then transmit the data stream to the host computer.

The *Opcode Note1/++* is an external MIDI interface which communicates to the host computer via the printer port.

Description of the Microsoft Windows MIDI Application Software

A MIDI application must handle two levels of communications:

- Signal path from the MIDI interface to the host computer
- Handshake between the host computer and the DSP-based music synthesizer

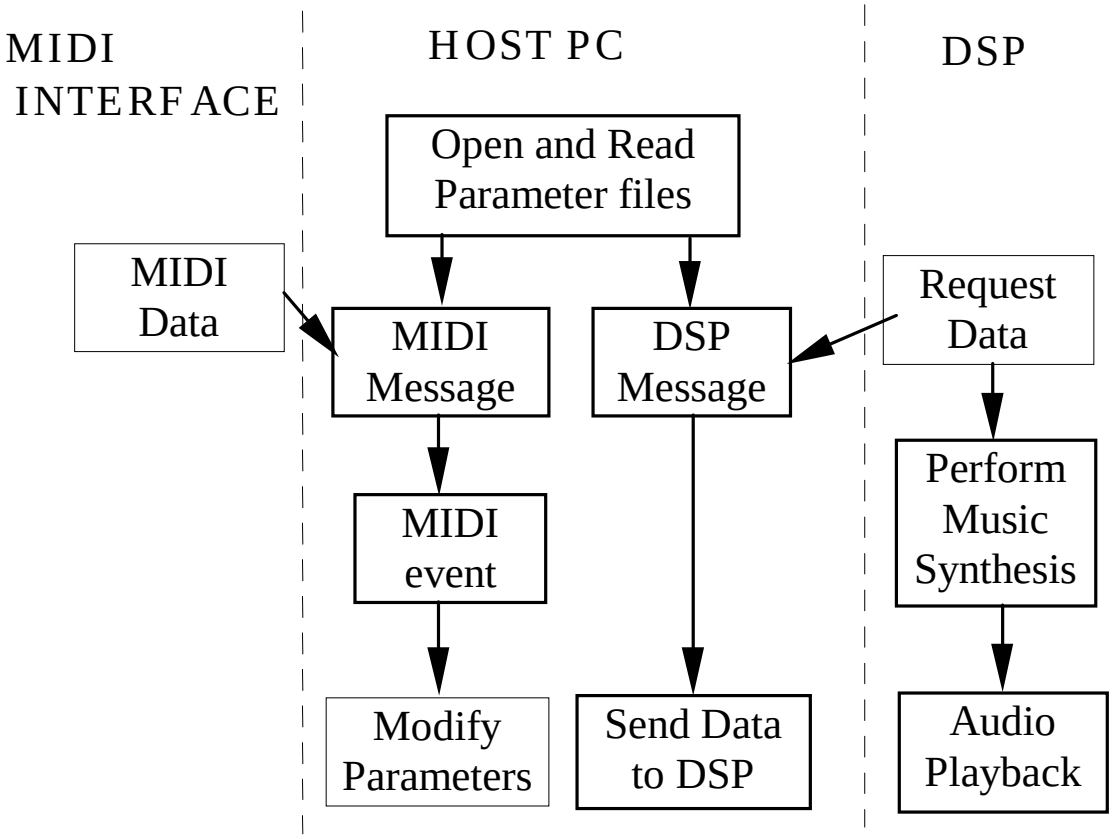
We have developed MIDI application software under Windows 3.1 (this can be upgraded to Windows 95 if necessary). Microsoft Windows applications address hardware devices (such as MIDI interfaces and synthesizers) by means of device drivers. Device drivers provide application software with a common interface through which hardware can be accessed. This simplifies the hardware compatibility issue and increases the flexibility of the application software. That is, the application software can be used with any MIDI device that has a standard Windows device driver.

The MIDI interface comes with a multi-client Windows device driver that provides low level control over the received MIDI data stream and decodes and exports incoming MIDI messages to the Windows operating system. The application software is responsible for system initialization, such as:

- Loading parameter files
- Receiving MIDI events (for example, musical instrument type, note on/off, key pressure, etc.)
- Invoking appropriate responses by modifying synthesis parameters
- Delivering synthesis parameters to the DSP at the request of the DSP

Figure 2 shows the software organization.

Figure 2. Interface between the MIDI Device, Host, and DSP



Parameter Update Problem (Handshake between PC and DSP)

There are two problems concerning the data communications or parameter update.

- The fixed rate parameter update required at a certain sample period (since we cannot load the entire parameter file into on-board RAM due to limited memory available on the current Tiger32EVM board)
- The burst of information generated by the MIDI events. The rate of data flow between the PC and DSP is greatly limited by the I/O bandwidth available on the hardware. There is only one 16-bit data transfer register (TXREG) on the Tiger32EVM board [2] for this purpose.

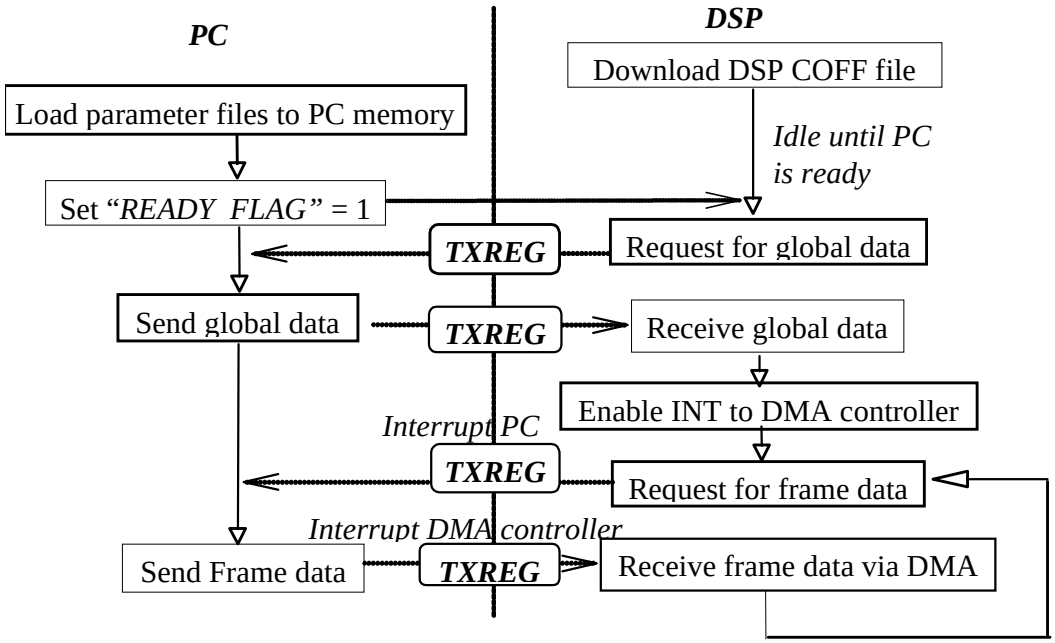
We chose the most basic update mechanism by updating all parameters at the beginning of the sample computation cycle. We incorporated synthesis commands and changes to existing parameter values in the new information.

One problem with this approach is that the duration of the update period is fixed. Therefore, only a certain number of updates can take effect on any one cycle. If the rate of the updating process is higher than the rate of the incoming MIDI events, there is no audible erratic or delayed musical response. Based on our experiments, we found that this update delay might be acceptable within the range of 2-5 ms for some applications.

The sequence of operations involved in the handshaking between the host PC and the DSP is illustrated in Figure 3. In the initialization phase, the user selected input parameter files are loaded into PC memory. A signal is sent to the DSP to indicate that the PC is ready for operation. Two types of parameter, global data and frame data have to be transmitted. Global data is transmitted once during initialization, whereas frame data is updated in every frame when the DSP requests for data by interrupting the host PC. In the interrupt service routine, a "*DSP DATA REQUEST*" message is posted to the window operating system.

Interrupt to the PC occurs when the DSP writes to *TXREG* and conversely, the host PC can interrupt the DSP (external hardware interrupt 0) when it writes data to *TXREG*. Data transfer is triggered by interrupting the host (since the host cannot dedicate its processing power to poll for only one event) and using DMA on the remote DSP (reserve CPU power for synthesis). In our data transfer scheme, the DSP requests data by writing to *TXREG*, which interrupts the PC. When the application receives a "*DSP DATA REQUEST*" message, it will update the new frame data by writing data to *TXREG*. This generates an interrupt to the DMA controller on the DSP, which in turn causes a 16-bit DMA data transfer from *TXREG* to another specified memory location [3].

Figure 3. Handshake between PC and DSP



PC Interrupt

Three steps are involved in using interrupts:

- Generating interrupts
- Enabling interrupts
- Servicing interrupts

Interrupts are generated when the DSP writes to *TXREG*. We can select the *IRQ* line (*IRQ10-12,14,15*) on the host PC by setting a jumper on the Tiger32EVM board [2]. There are two programmable interrupt controllers (*PIC*) on most PCs. *IRQ0-IRQ7* are enabled by masking the corresponding bit of *PIC1* and *IRQ8-IRQ15* uses *PIC2*. Disabling *IRQ2* disables *IRQ8-IRQ15* [4][5].

A "*DSP DATA REQUEST*" message is posted to the window operating system during the interrupt service routine indicating that DSP is ready for new parameter update. When the application receives a "*DSP DATA REQUEST*" message, it sends frame data to the DSP.

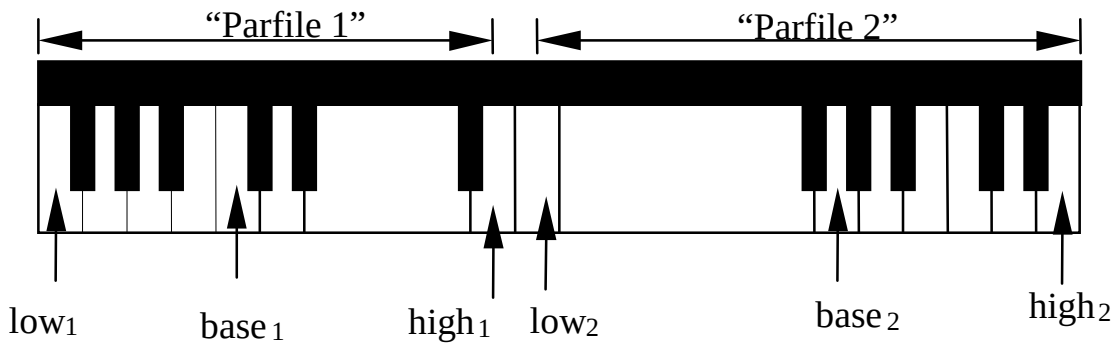
Floating-Point Format Conversion (UNIX to PC IEEE, IEEE to DSP)

The parameter files required by the music synthesis algorithms are obtained by analyzing the music signal under the UNIX environment. Two conversions are required when reading a floating-point value from a binary file produced from UNIX to the DSP. Byte swapping and then word swapping are required for a 32-bit floating-point value in converting from big-endian format to little-endian format. Then, another conversion from IEEE floating-point format to TI TMS320C3x floating-point format has to be performed as well [3].

Key Mapper

It is not practical to store analysis parameters for each note across the entire keyboard due to large memory requirements. Typically, one set of parameters generated from the analysis of one note is stored and used for a range of notes by applying pitch shifting. An example of this strategy using parameters from two notes is given in Figure 4. The original note is mapped to the base key covering a range from the low key to the high key.

Figure 4. Key Mapping for Music Synthesis



MIDI Message Processor

The process begins with initializing and opening the specified device for MIDI input. The incoming MIDI input received from the Windows drivers is handled by internal buffers embedded in the API (application programming interface). The window handle for the application will only receive two different types of messages. One is the MIDI event related message and the other is the MIDI device closing message.

When the application software receives a MIDI message, it translates the 3-bytes of data into a meaningful MIDI event according to the MIDI protocol. The list of MIDI event data bytes can be found in reference [1]. The MIDI event is then decoded into synthesis commands and transmitted to the DSP along with the appropriate modified synthesis parameters.

Music Synthesis Engine

Frame-based music synthesis is performed on the DSP. At the beginning of each frame, the DSP requests a set of synthesis parameters from the host PC by sending an interrupt message to the operating system. It transfers the synthesis parameters to the DSP memory using the DMA mode so that the DSP processing power can be reserved for music synthesis. The synthesized music samples are sent to the digital-to-analog converter (DAC) via the serial port using the DMA mode as well.

Music synthesis is carried out by the music synthesis engine processing on the TI TMS320C32 digital signal processor. Synthesis is driven by the synthesis commands and the parameters received from the host computer. At the end of each computation cycle, audio samples are generated and playback through the audio codec. Audio interface uses DMA to shift samples to the audio buffer.

Music Synthesis Algorithms

At present, two music synthesis algorithms are implemented on this system

- SMS (Spectral Modeling System) algorithm
- ABS/OLA (Analysis-By-Synthesis/OverLap-Add) algorithm

The SMS algorithm employs a sinusoidal plus stochastic model. Synthesis of the sinusoidal components is accomplished using an oscillator bank. Synthesis of the stochastic part is achieved by using inverse FFT or driving a “formant filter” with white Gaussian noise [6].

The ABS/OLA algorithm uses an analysis-by-synthesis approach based on a pure sinusoidal model. Synthesis is performed by inverse FFT [7].

The real-time implementation of the two sinusoidal model-based music synthesis approaches, ABS/OLA and SMS, has been demonstrated for the REALMS project. The heart of the music synthesis engine is the TMS320C32 DSP. Currently, we chose this low cost floating-point DSP, the TMS320C32, as our platform since the music synthesis algorithms are sensitive to numerical precision.

Both algorithms were initially simulated on an SGI workstation in C and later ported to the TMS320C32 DSP (Tiger32EVM board incorporates a TMS320C32 yielding a throughput of *25 MIPS* and two memory banks of *128* kbytes of zero wait state static RAM). It is essential to use the most effective algorithm in terms of speed and memory in order to meet the constraints imposed by the real time system. We have achieved single-note music synthesis driven by MIDI-device at 44.1 kHz sampling rate on this system. The implementation of system level issues can be referred to reference [8].

In this section, we discuss the issues involved in real-time implementation of the two music synthesis algorithms, ABS/OLA and SMS, which include code and memory optimization.

Descriptions of Music Synthesis Algorithms

Music signals can be analyzed using either ABS/OLA or SMS algorithm. Both approaches are based on sinusoidal models. Information regarding the number of sinusoids, amplitude, frequency and phase of each sinusoid is extracted from the music signal during analysis phase and used in the synthesis stage.

Analysis-By-Synthesis OverLap-Add (ABS/OLA)

The ABS/OLA algorithm uses an analysis-by-synthesis approach based on a pure sinusoidal model. Synthesis is performed by inverse FFT. In the synthesis stage, a frequency spectrum is constructed by summing the sinusoidal components extracted in the analysis stage along with the necessary time scale or frequency modifications. Since the frequency obtained from analysis does not necessarily lie exactly on the DFT bin, we use two frequency components at the DFT bins closest to the extracted frequency to form an approximation by solving a set of normal equations [7]. The time-domain signal can simply be synthesized by performing an inverse FFT on the reconstructed spectrum on a frame-by-frame basis with 50% overlap-add. The frame length is 220 samples at 44.1 kHz sampling rate which corresponds to approximately 5 ms. The computational loading for the ABS/OLA algorithm is dominated by the inverse FFT if we can minimize the overhead required by the approximation of sinusoidal components. We have attempted to reduce the complexity of constructing the spectrum by exploiting the psychoacoustic model. It is known that we do not perceive a shift in higher frequencies as well as lower frequencies. Therefore, it is possible to form the spectrum by approximating sinusoids as described above for lower frequency components and quantizing the sinusoidal component to the closest DFT bin in high frequency region. In our experiments, there is no audible distortion when a cut-off frequency of 4 kHz was used. One advantage of ABS/OLA is that only one inverse FFT has to be computed during synthesis for any number of sinusoids and any number of notes.

Spectral Modeling System (SMS)

The SMS algorithm employs a sinusoidal plus stochastic model to synthesize music on a frame-by-frame basis. The synthesis frame length in our current system is approximately 2.5 ms at 44.1 kHz sampling rate. Each sinusoidal component is constructed by an oscillator with given amplitude, frequency and phase extracted from the analysis and then summed together. The phase continuity at frame boundaries is conserved by using a cubic polynomial interpolation algorithm. The stochastic component can be synthesized by using inverse FFT or driving a “formant filter” with white Gaussian noise. However, we have omitted the stochastic component in our implementation. The oscillator bank is simple. However, if the number of sinusoids and the number of notes grow, the computational loading will increase linearly.

TMS320C32 Implementation and Optimization

The process required for implementing the music synthesis algorithms in real time involves porting the simulated C code from a UNIX environment to a DSP by compiling the code using the target DSP's C cross-compiler, TI TMS320C3x compiler in this case. We will discuss some of the effective ways that we applied to speed up our music synthesis process in the following sections.

Compiler Optimization

The compiler provides optimizations if we enable optimization in the compiling command line such as -o3 where the compiler will assign registers to variables whenever possible to minimize memory fetches according to some cost functions, utilize pipeline delay, repeat blocks with zero overhead, autoincrement addressing and conditional loading. We have also found some of the compiling options which are effective, such as

- -mc option

  Provides a one instruction float to integer conversion and vice versa

- -x2 option

  For function inlining to reduce function call overheads and allow the compiler to optimize in the context of the function body

Further details and options on the compiler are given in references [9][10].

Code Optimization

Code optimization includes reducing function calls inside critical loops and assigning reference pointer to loop invariant long pointer expression outside critical loop [10]. We also tried to minimize some of the more expensive DSP operations such as division, power and to calculate sine, cosine and sinc values using table look-up.

Table Look-Up for Sine, Cosine and Sinc Functions

The sine, cosine and sinc functions are pre-calculated and stored from *0 to 2π*. For example, let *M* be the size of the table *COSTAB* and *M/2π* be the scale where *M* is a power of *2*. Then, cos(*θ*) can be obtained from *COSTAB[(int)(θ×(2π/M))&(M-1)]* where the *&(M-1)* term gives modulo of *2π*. We have found that a sine-table of size *4096* provides enough resolution for both synthesis algorithms. The necessary accuracy can be traded off with memory consumption. The pre-storage of sinc table provides us even greater savings on computational time since normally a computation of sine followed by a division is required. The sinc function is used in the reconstruction of sinusoidal components in the frequency domain for the ABS/OLA algorithm. Due to the special property of the sinc function where more significant values lie in the main lobe, we have used a high resolution table for arguments within *±4* and a low resolution table for values outside this range.

In general, look-up tables can be used to reduce computational loading whenever incremental values have to be computed over and over again. However, there is a trade off between speed and memory.

Approximation of Polynomial

The phase of a sinusoid in the SMS algorithm is represented by a cubic polynomial as:

$$\theta(t) = a + bt + ct^2 + dt^3 \tag{1}$$

It is possible to approximate $\theta(t)$ by accumulating its previous value $\theta(t-1)$ with a time varying term in order to reduce computations. However, the accumulated error also propagates along the time scale. Hence, we only use this approximation up to the quadratic term and an exact value is used for the cubic term. Equation(1) can be re-written as follows:

$$\theta(t) = p(t) + dt^3 \tag{2}$$

where *p(t)* is approximated as:

$$\begin{aligned} p(t) &= p(t-1) + Q(t), \\ Q(t) &= Q(t-1) + \Delta Q \end{aligned} \tag{3}$$

where the initial condition for equation (3) is:

$$\begin{aligned} p(0) &= a, \;\; Q(0) = b - c, \\ \Delta Q &= 2c \end{aligned} \tag{4}$$

Equation (1) requires 3-additions and 3-multiplications assuming table-look up is used for the quadratic and cubic term with two memory fetches. Now, by substituting equation (3) into equation (2), the number of instructions is reduced to 3-additions and 1-multiplications with one memory fetch. Memory fetches can be significant if external memory has to be accessed due to conflicts and contentions on the external bus. The look-up table most likely has to be stored in external memory due to the limited internal RAM available on TMS320C32.

In general, any polynomials can be approximated as described above where accuracy is compromised with speed.

Reduction on External Memory Fetches

There is usually a penalty on speed whenever an external memory fetch is involved. Hence, it is natural to store all the codes and data for a time critical loop in internal memory. However, the TMS320C32 DSP only provides two 256-words internal RAM which is impossible to fit all the critical codes and data. We try to maximize the performance first by enabling instruction caching to reduce repeat-loop instruction fetches and second by moving data into the internal RAM whenever operations are being performed on the data. However, we will introduce an overhead when copying data to and from the internal RAM. Therefore, we have to choose the best balance between the amount of time saved by operating data in internal RAM and the amount of overhead required for copying data.

In the ABS/OLA algorithm, the most expensive computation is the 1024-point IFFT. Since the largest continuous block of data that can be stored in internal RAM is 256 words, we split the IFFT algorithm into four 256-point IFFT. A 1024-point IFFT can be divided into 10-stages. Since our frequency spectrum is symmetric, an IFFT can be nicely computed using frequency decimation. There will be four 256-point IFFT remained after the second stage. In order to reduce the number of external memory fetches, we perform the first two stages of IFFT with data in the external memory. Then, the first 256-points is moved to the internal memory and an 256-point IFFT is performed. The results are copied back to the original external memory. This is followed by the same operations on the second, third and fourth set of 256 data points. Finally, bit-reversed addressing is performed on the 1024 data points. The overhead for moving data between internal and external memory is insignificant compared to the time spent on accessing external memory during the operation of an IFFT. We have further reduce the computational loading by storing the sine and cosine factors used by the IFFT in internal memory as well. These operations led us to a reduction of 30% in computational time for the inverse FFT.

In general, instructions and data involved in an interrupt service routine is also critical in most real time system. Hence, we have assigned the codes and data involved in the interrupt service routine to internal memory.

Hand Assembly Coding

Hand assembly coding can be used to take advantage of the special architecture of DSP hardware such as bit-reversed addressing, circular addressing and other parallel instructions which are not supported by C language. In the ABS/OLA algorithm, the inverse FFT routine is written in assembly language in order to fully exploit bit-reversing, zero-overhead loops, parallel multiply-add, parallel store and parallel load instructions.

Discussion and Summary

Real time systems are always constrained by the limited resources available to perform certain operations. Two major issues we addressed in this report are data transfer from the host computer to DSP and the complexity of synthesis algorithms.

The rate at which data can be transferred between the PC and the DSP is mainly limited by the I/O bandwidth of the hardware (or the communications between the development board and PC). The Tiger32 EVM has only one data transfer register (16 bits) for this purpose. It is time consuming to transmit a block of more than 300 floating-point (32 bits) values in each frame. Initially, we used polling on both the host and the remote processor, which exhausted more than 80% of the total CPU power. We alleviated this problem by using interrupt on the host machine and DMA on the remote processor. However, an ideal solution is to have a large block of PC mapped memory for high bandwidth data transfer, such as synthesis parameters, and a mapped register for rapid small bandwidth transfer, such as the burst of MIDI information or other control signal.

We attempted to reduce the computational time of the music synthesis algorithms by optimizing our code and hand assembly coding some of the critical functions, such as inverse FFT, by taking advantage of the special architecture of the DSP. We are also working on optimizing the complexity of the algorithms, such as exploring the use of split-radix FFT for lower computational loading; as well as developing more efficient high quality music synthesis.

Currently, we are using a floating-point DSP because music synthesis algorithms require parameters with very accurate precision. We have achieved single-note music synthesis in real-time at 44.1 kHz sampling rate for both ABS/OLA and SMS on our system [11].

At the present time, this system only supports single note synthesis. to demonstrate and test our synthesizer designs fully, we need to be able to implement polyphonic synthesis in the REALMS system. To accomplish this, we will investigate new algorithm implementations that perform similar functions more efficiently. The current system was intended to provide reference designs for future development of music system (cards or even a single music synthesis IC; we will eventually wish to study low-cost fixed-point DSP implementations as well). Our exercise has achieved this goal where most of the major system-level issues have been resolved.

We ported the two music synthesis algorithms to the TMS320C32 DSP. We achieved single note music synthesis using either the ABS/OLA or SMS approaches in real time on the TMS320C32 at 44.1 kHz sampling rate by applying the optimization discussed in the previous sections. Currently, we are using 20 sinusoids for ABS/OLA and 7 sinusoids for SMS. We plan to implement polyphonic music synthesis and with more sinusoids in the future, but on a more flexible platform. A summary of the usage on memory and computational loading to date for both ABS/OLA and SMS are provided in the following sections.

ABS/OLA Resources Usage

The computational loading for the ABS/OLA algorithm can be broadly divided into the approximation of frequency spectrum using the sinusoidal components obtained in the analysis stage and the computation of inverse FFT. The spectrum approximation requires a few sine, cosine and sinc table look-ups, divisions and solving a set of 2×2 normal equations. The processing power consumption and memory usage for 5 ms frame rate at 44.1 kHz sampling rate are tabulated in Table 1 and Table 2.

Table 1. Computational Loading for ABS/OLA (5 ms Frame Rate)

Routines	Timing in Instruction Cycles
Spectrum Approximation (20 sinusoids)	12078
Inverse FFT: initial 2 stages + 4 256-pt	46264
bit-reversed addressing	9195
memory update	4512
Others (system interface, audio & midi)	17951
Total	90000

Table 2. Memory Usage for ABS/OLA (5 ms Frame Rate)

Codes and Data	Number of Words in Decimal
Program: Spectrum Approximation	217
Inverse FFT	283
Others (system, audio & midi)	2701
Data: Sine Table	4096
Cosine Table	4096
Sinc Table	2048
Buffer for Spectrum and IFFT	1920
Buffer for Output Samples	660
Input Parameters	18224
Others	2441
Total	36686

SMS Resources Usage

The core of SMS includes phase interpolation using a cubic polynomial to maintain continuity at frame boundaries and the construction of oscillator bank by generating an oscillator from the information obtained in the analysis stage and summing the contributions from each sinusoidal components. The computational loading and memory usage for 2.5 ms frame rate at 44.1 kHz sampling rate is tabulated in Table 3 and Table 4. The core consumes approximately 35% of the processing power.

Table 3. Computational Loading for SMS (2.5 ms Frame Rate)

Routines	Timing in Instruction Cycles
Oscillator bank (7 sinusoids)	21876
Others (system interface, audio & midi)	8128
Total	30004

Table 4. Memory Usage for SMS (2.5 ms Frame Rate)

Codes and Data		Number of Words in Decimal
Program: Oscillator Bank		456
	Others (system, audio & midi)	2310
Data:	Cosine Table	4096
	Index Table (cubic term)	110
	Buffer for Output Samples	330
	Input Parameters	1428
Others		1567
Total		10297

Conclusion

We achieved real-time single note music synthesis driven by a MIDI-device at 44.1 kHz sampling rate on a TMS320C32 DSP by using the optimizations we discussed in this report. The computational loading and memory usage shown are the performance to date only. There is still room for improvement in speed and memory usage for both ABS/OLA and SMS algorithms.

References

[1] Joseph Rothstein, "*MIDI - A Comprehensive Introduction*", A-R Editions, 1992.

[2] "*Tiger32EVM User's Manual*"

[3] "TI TMS320C3x User's Guide"

[4] Al. Williams, "*DOS 5: A Developer's Guide-Advanced Programming Guide to DOS*",1991

[5] Thom Hogan, "The programmer's PC Source Book", 1991

[6] X. Serra and J. O. Smith, "Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plush stochastic decomposition," *Computer Music J.,* vol. 14, no. 4, pp. 12-24, Winter 1990.

[7] E.B.George and M.J.T.Smith, "Analysis-by-Synthesis/OverLap-Add sinusoidal modeling applied to the analysis and synthesis of musical tones*", J.Audio Eng. Soc.,* vol. 40, no. 6, pp. 497-516, June 1992.

[8] Susan Yim, "A MIDI-driven Real Time Music Synthesis System Using the TMS320C3x Digital Signal Processor Family", *TI TAR*, October 11, 1996.

[9] Mark E.Paley and David H.Bartley, "C-Language style optimizations for the TMS320C30", *TI TAR*, May 6, 1994.

[10] *TMS320 Floating-Point DSP Optimising C Compiler*

[11] S.Yim, Y.Ding & E.B.George, "Real-time implementation of Music synthesis algorithms, ABS/OLA and SMS on a TI TMS320C32", *TI TAR*, October, 1996.