Application Report
SPRA634 - April 2000

TMS320C6000™ μ-Law and A-Law Companding with Software or the McBSP

Mark A. Castellano
Todd Hiers
Rebecca Ma

Digital Signal Processing Solutions

ABSTRACT

This document describes how to perform data companding with the TMS320C6000™ digital signal processors (DSP). Companding refers to the *com*pression and ex*pan*sion of transfer data before and after transmission, respectively.

The multichannel buffered serial port (McBSP) in the TMS320C6000 supports two companding formats: μ-Law and A-Law. Both companding formats are specified in the CCITT G.711 recommendation [1]. This application report discusses how to use the McBSP to perform data companding in both the μ-Law and A-Law formats. This document also discusses how to perform companding of data not coming into the device but instead located in a buffer in processor memory. Sample McBSP setup codes are provided for reference.

In addition, this application report discusses μ-Law and A-Law companding in software. The appendix provides software companding codes and a brief discussion on the companding theory.

Contents

List of Figures

List of Tables

1 Design Problem

How can the TMS320C6000 digital signal processor be used for μ-Law or A-Law companding?

2 Overview

Companding (compression and expansion of transfer data) can be found in many different applications, especially in digital telephone systems. A digital telephone system converts an analog speech signal to a digital signal. This digital signal is referred to as linear—meaning without compression. Instead of transmitting this linear digital signal across the telephone network, this digital signal is usually first compressed before being transmitted to reduce the transmission bandwidth. The receiver needs to expand this non-linear, compressed signal back to a linear digital signal. Companding refers to this combined process of compression and expansion. This application report discusses two μ-Law and A-Law companding implementations: McBSP and software.

3 Companding With the McBSP

The McBSP supports two companding formats—μ-Law and A-Law. This application report discusses how the McBSP hardware handles both the μ-Law and A-Law companding. Figure 1 is an overview of the McBSP companding hardware operation. On the receive side, the McBSP receives the compressed and nonlinear data and expands it to linear data. On the transmit side, the McBSP compresses the linear data using either the μ-Law or A-Law formats before its transmission. This document describes the McBSP configuration necessary to perform μ-Law and A-Law companding, and internal data companding.

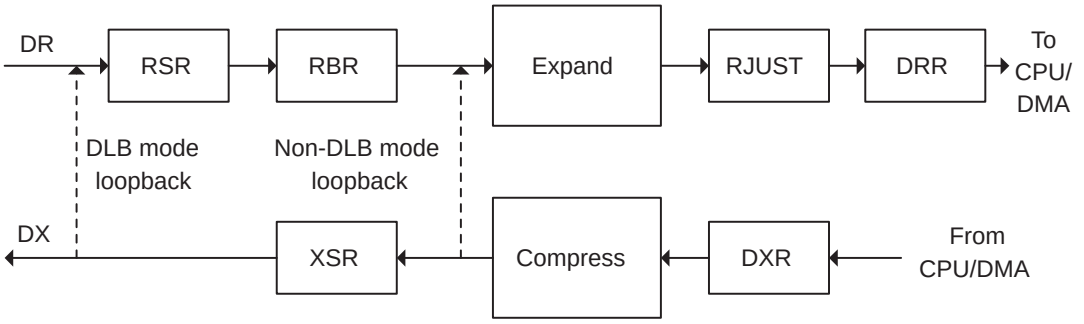


Figure 1. Block Diagram of Compand Flow

Compressed data is transmitted on the data lines in 8-bit words. However, the data once expanded is 13-bit (A-Law) or 14-bits (μ-Law) in length. Thus, the DMA or CPU should perform 16-bit read and write operations to service the McBSP. The expanded data is placed in the most significant bits in a LAW16 block (as shown in Figure 2).

3.1 μ-Law Companding

Companded data is always eight bits wide, therefore (R/X)WDLEN(1/2) in the receive/transmit control register (RCR/XCR) must be set to 0. If the data is not eight bits long, companding continues as if the element length were eight bits. **Therefore, it is very important that the data is eight bits wide to avoid data loss or corruption.** Incoming data will expand to 14 bits, left justified in a LAW16 block. The alignment of that LAW16 block in the 32-bit DRR is selectable by the RJUST field in the serial port control register (SPCR). Table 1 summarizes the format options available. Unlike received data, data to be transmitted with companding does not have a selectable alignment option. It must be left-justified in a LAW16 block that is in the lower half of DXR. **NOTE** that only the most significant 14 bits in the LAW16 block will be encoded.

Other than the word length register field, the only other field that needs to be set to enable companding is (R/X)COMPAND in RCR/XCR. This register field should be set to 10b for μ-Law companding.

Table 1. Justification of Expanded Data in DRR

	DRR Bits	
RJUST	31 16	15 0
00	0	LAW16
01	Sign	LAW16
10	LAW16	0
11	Reserved	Reserved

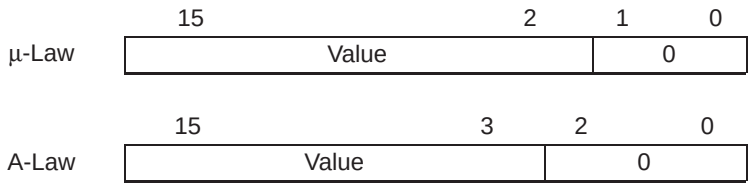


Figure 2. LAW16 Formats

3.1.1 McBSP Register Configuration for μ-Law Companding

The setup of bit fields in the control registers to enable μ-Law companding is listed in Table 2. The registers are configured to implement μ-Law companding (RCOMPAND = 10b).

Table 2. Bit-Field Values for McBSP Registers (μ-Law)

Register [Bit-Field #]	Bit-Field Name	Value (in Binary) Slave (Receiver)	Function
RCR[7:5]	RWDLEN1	000	8 bits Receive Word Length
RCR[20:19]	RCOMPAND	10	μ-Law Companding
XCR[7:5]	XWDLEN1	000	8 bits Transmit Word Length
XCR[20:19]	XCOMPAND	10	μ-Law Companding

NOTE: The bit fields and registers not listed in Table 2 assume their default values. The user is responsible to set some of the register fields and other parameters.

3.2 A-Law Companding

A-Law companding is similar to μ-Law companding, except that expanded values are 13 bits instead of 14. Again, this 13-bit value must be left aligned in a LAW16 block which is in the lower half of DXR. When receiving A-Law data, the 32-bit alignment in DRR is configurable via the RJUST field in SPCR, just like μ-Law (see Table 1). A-Law companding is enabled by setting (R/X)COMPAND to 11b.

3.2.1 McBSP Register Configuration for A-Law Companding

The setup of bit fields in the control registers to enable A-Law companding is listed in Table 3. The registers are set up to implement A-Law companding (RCOMPAND = 11b). The element size must be set to eight bits.

Table 3. Bit-Field Values for McBSP Registers (A-Law)

Register [Bit-Field #]	Bit-Field Name	Value (in Binary) Slave (Receiver)	Function
RCR[7:5]	RWDLEN1	000	8 bits Receive Word Length (Phase 1)
RCR[20–19]	RCOMPAND	11	A-Law Companding
XCR[7:5]	XWDLEN1	000	8 bits Transmit Word Length (Phase 1)
XCR[20–19]	XCOMPAND	11	A-Law Companding

NOTE: The bit fields and registers not listed in Table 3 assume their default values. The user is responsible to set some of the register fields and other parameters.

3.3 Companding Internal Data

If the McBSP is not otherwise being used in an application, it can be configured to compand internal data. It can do this using digital loop back (DLB) mode or non-DLB mode. shows the McBSP internal connection for the different digital loop back modes. DLB mode mimics the behavior of a serial transmitter/receiver pair by internally connecting DR to DX. The synchronization events (REVT and XEVT) from DLB mode make it possible to use the DMA split mode to service the McBSP. In addition, in DLB mode, you can use the sample rate generator to set the DMA bandwidth used. Non-DLB mode bypasses some of the serial circuitry and internally connects DRR to DXR through the companding logic. The advantage of the non-DLB mode is its speed and simplicity. Non-DLB companding is done with a simple software code. You do not need to set up any interrupt service routine or to configure the DMA to perform companding.

3.3.1 Non-DLB Mode

When both the transmitter and receiver are reset (/RRST = /XRST = 0), DRR and DXR are internally connected through the companding logic. Values written into DXR are compressed according to XCOMPAND, are then expanded according to RCOMPAND, and are available in DRR four CPU clocks after being written in DXR. This method is faster than the DLB mode, but RRDY and XRDY are not set, so there is no synchronization available to the CPU or DMA controller to control the flow of data.

There are six useful functions to perform on internal data. The functions and register fields needed to configure the functions are listed in Table 4.

Table 4. (R/X)COMPAND Values for Internal Data Compresssion

Function	XCOMPAND	RCOMPAND
Compress to μ-Law format	10b	00b
Expand from μ-Law format	00b	10b
Compress to A-Law format	11b	00b
Expand from A-Law format	00b	11b
Observe μ-Law quantization effects	10b	10b
Observe A-Law quantization effects	11b	11b

To use non-DLB mode, keep the receiver and transmitter in reset by setting /RRST and /XRST in SPCR to 0, then set (R/X)COMPAND and RJUST as desired to perform the desired operation. Data may then be written to DXR and the converted data read from DRR four CPU cycles later.

3.3.2 DLB Mode

When DLB in SPCR is set, DR and DX are internally connected. Values written to DXR are transmitted through all of the serial transmit circuitry and received through all of the receiver circuitry, ending up in DRR. (R/X)COMPAND controls the function performed (see Table 4). Using this method, CPU or DMA controller synchronization is possible, through transmit interrupts (RINT and XINT) or synchronization events (REVT and XEVT), respectively. Since the full serial circuitry is involved, the conversion time is dependent on the serial bit rate selected.

To use this mode, configure the McBSP for use like a serial port interfaced to the external world (see *TMS320C6000 McBSP Initialization Application Report, literature number SPRA488).* Then set the DLB bit in SPCR and set (R/X)COMPAND to perform the desired function. Data transfer may then be started by the CPU or DMA controller.

3.4 Sample C Functions

The C functions in Appendix C set up the McBSP and DMA to perform data transfer. The code can be tested in DLB mode by uncommenting the indicated line, or used as is for serial communication with an external device.

McBSP 0 is set up to do A-Law companding using the CPU clock to derive the serial clock. The McBSP can be configured for μ-Law companding by changing the appropriate two constants in the serial setup function. DMA channels 0 and 1 are set up to service the McBSP with 256 16-bit data elements. DMA channel 0 transfers data from memory to the McBSP, and DMA channel 1 transfers data from the McBSP back to memory. A single DMA channel could be split to perform the same task.

4 Companding With Software

It is sometimes desirable to transmit one or more 8-bit channels as uncompressed data. This data may reflect signaling or other transmission information, or merely represent raw data. Such a channel is often referred to as a “clear channel.” To accommodate a mixture of compressed and raw data would require selectivity on a per channel basis within the McBSP. This can be done in software.

If all McBSPs within a DSP are being utilized, multichannel companding can still be implemented in software, without the use of peripherals. The transmission would still consist of 8-bit channels. The received compressed channels would be passed into an expansion routine, while the “clear channels” would be processed as-is. Similarly, linear voice channels would be passed to a compression routine prior to being transmitted by the McBSP.

A detailed description of implementing G.711 companding in software can be found in Appendix A at the end of this document. Sample assembly routines for both A-Law and μ-Law companding can be found in Appendix B. All routines are C-callable and hand-assembled to optimize speed and utilize only the resources of the TMS320C6000 DSP core. While a detailed description of G.711 companding theory is beyond the scope of this document, comparable information may be found in the references mentioned at the end of this document.

5 Conclusion

The TMS320C6000 device can perform companding in both μ-law and A-law configurations. The companding modes in the McBSP provide the simplest and most efficient way to perform companding. Alternatively, if an application requires a mixture of raw and compressed data on a per-channel basis, the fast software implementation detailed in Appendix A and Appendix B can be used.

6 References

1. *TMS320C6201 Digital Signal Processor Data Sheet*, SPRS051.
2. *TMS320C6201/C6701 Peripherals Reference Guide*, SPRU190.
3. *TMS320C6x Peripheral Support Library Programmer's Reference*, SPRU273.
4. Bellamy, J., *Digital Telephony, 2nd Edition*, John Wiley & Sons, Inc., New York, 1991.
5. Brokish, C., Lewis, M., *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, SPRA163.
6. Pagnucco, L., Erskine C., *Companding Routines for the TMS32010/TMS32020*, *DSP Applications with the TMS320 Family, Vol. 1.*

Appendix A Companding Discussion and Implementation

A system that compresses information, transmits it through a channel, then expands it upon reception performs companding. Companding may be achieved in hardware by a codec, or in software by either direct calculation or a look-up table approach. Companding is used to limit a signal's amplitude or bandwidth. Any intermediate processing should be performed on the original uncompressed data.

There are two international companding standards: μ-law and A-law. Each standard retains up to five bits of precision by compressing data into eight bits. The United States and Japan both support μ-law, while A-law is the accepted European standard. Both standards are discussed in the following sections.

A.1 μ-Law Companding

The United States and Japan support μ-law companding. Limiting sample values to 13 magnitude bits, μ-law compression can be defined mathematically by the following continuous equation:

$$F(x) = \mathrm{sgn}(x)\ \ln(1 + \mu\,|x|) / \ln(1 + \mu) \qquad \textbf{Equation (1)}$$
$$-1 \le x \le 1$$

where μ is the compression parameter ($\mu=255_{10}$ for the U.S. and Japan), and x is the normalized integer to be compressed. Figure A–1 illustrates a piece-wise linear approximation to this compression equation.

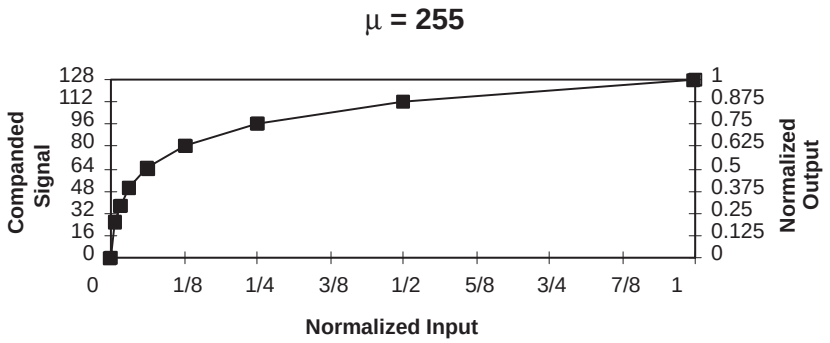


Figure A–1. μ-Law Companding Curve

The least significant bits of large amplitude values are discarded during compression. The number of deleted bits is encoded into a field of the encoded word, called the segment. Each segment of this piece-wise linear approximation is equally divided into quantization levels. The segment size between adjacent codewords is doubled for each succeeding segment. Moreover, the most significant bit of the codeword contains the sign of the original integer. An 8-bit μ-255 codeword is comprised of one sign bit, concatenated with a 3-bit segment, concatenated with a 4-bit quantization value. Prior to transmission, all the bits are inverted so a positive value will have a sign bit of "1" (one).

Prior to segment determination, the sign of the original integer is set aside and a bias of 33_{10} is added to the absolute value (magnitude) of the integer. The bias limits the maximum allowable input to 8159_{10}, and reduces the minimum step size to $2/8159_{10}$. The bias simplifies the calculation by making the endpoints of each segment powers of two. Locating the segment is determined by detecting the most significant "1" of the biased magnitude, while the quantization value is comprised of the four bits following it.

The translation from linear to μ-law compression is illustrated in Table A–1. Of the compressed codeword, bits 0–3 represent the *quantization* and bits 4–6 represent the *segment*. The sign of the compressed codeword is left out for simplicity.

Table A–1. μ-Law Binary Encoding Table

Biased Input Values														Compressed Code Word							
														Chord				Step			
Bit:	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Bit:**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	0	0	0	0	0	0	1	a	b	c	d	x		0	0	0	a	b	c	d
	0	0	0	0	0	0	1	a	b	c	d	x	x		0	0	1	a	b	c	d
	0	0	0	0	0	1	a	b	c	d	x	x	x		0	1	0	a	b	c	d
	0	0	0	0	1	a	b	c	d	x	x	x	x		0	1	1	a	b	c	d
	0	0	0	1	a	b	c	d	x	x	x	x	x		1	0	0	a	b	c	d
	0	0	1	a	b	c	d	x	x	x	x	x	x		1	0	1	a	b	c	d
	0	1	a	b	c	d	x	x	x	x	x	x	x		1	1	0	a	b	c	d
	1	a	b	c	d	x	x	x	x	x	x	x	x		1	1	1	a	b	c	d

The entire μ-law codeword is inverted prior to transmission. The inversion is performed because low amplitude signals occur more frequently than large amplitude signals. Consequently, inverting the bits increases the positive pulse density on the transmission line, which improves system performance. μ-law expansion can be defined mathematically by the following continuous equation:

$$F^{-1}(y) = \mathrm{sgn}(y)\ (1 / \mu)\ [(1+ \mu)^{|y|} - 1] \qquad \textbf{Equation (2)}$$
$$-1 \le y \le 1$$

Prior to expansion, the μ-law codeword is inverted again During expansion, the least significant bits are discarded but are approximated by the median interval, to reduce the loss in accuracy. For example, if five of the least significant bits of the original integer were discarded during compression, 10000_2 will approximate them during expansion. The translation from μ-law to linear expansion is illustrated in Table A–2. Again, the sign bits are left out for simplicity. After decoding the μ-law codeword, the bias is removed and the sign bit is applied to obtain the final linear value.

Table A–2. μ-Law Binary Decoding Table

Compressed Code Word								Biased Input Values													
Chord				Step																	
Bit:	6	5	4	3	2	1	0	Bit:	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	a	b	c	d		0	0	0	0	0	0	0	1	a	b	c	d	1
	0	0	1	a	b	c	d		0	0	0	0	0	0	1	a	b	c	d	1	0
	0	1	0	a	b	c	d		0	0	0	0	0	1	a	b	c	d	1	0	0
	0	1	1	a	b	c	d		0	0	0	0	1	a	b	c	d	1	0	0	0
	1	0	0	a	b	c	d		0	0	0	1	a	b	c	d	1	0	0	0	0
	1	0	1	a	b	c	d		0	0	1	a	b	c	d	1	0	0	0	0	0
	1	1	0	a	b	c	d		0	1	a	b	c	d	1	0	0	0	0	0	0
	1	1	1	a	b	c	d		1	a	b	c	d	1	0	0	0	0	0	0	0

Dynamic range can be defined as the ratio between the lowest amplitude occupying the entire range of the first segment, and the highest amplitude that can occur. Hence, the dynamic range for μ-law companding can be calculated by the following equation:

$$DR = 20 \log_{10} (8159_{10}/31_{10}) = 48.4_{10}dB \quad \textbf{Equation (3)}$$

where 8159_{10} is the largest amplitude possible, and 31_{10} is the lowest amplitude spanning the first segment.

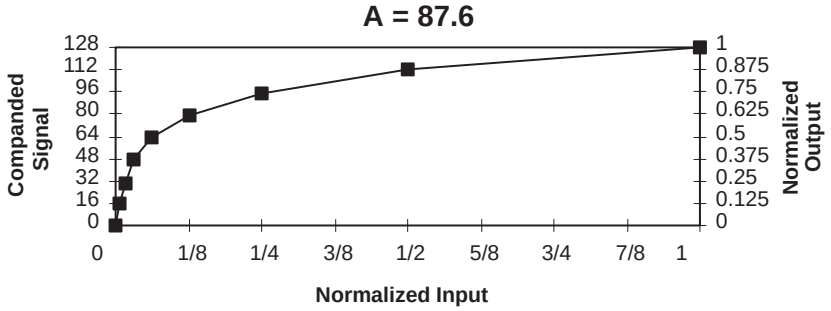
A.2 A-Law Companding

The European standard for companding is A-law. Limiting sample values to 12 magnitude bits, A-law compression can be defined mathematically by the following continuous equation:

$$F(x) = \begin{cases} \text{sgn}(x)\, A\,|x| / (1 + \ln A) & 0 \le |x| < 1/A \\ \text{sgn}(x)\,(1+\ln A|x|) /(1 + \ln A) & 1/A \le |x| \le 1 \end{cases} \quad \textbf{Equation (4)}$$

where A is the compression parameter ($A=87.6_{10}$ in Europe), and x is the normalized integer to be compressed. Figure A–2 illustrates a piece-wise linear approximation to this compression equation.

Figure A–2. A-Law Companding Curve



It is interesting to note here a few comparisons between the two types of companding. For example, A-law companding has similar features and implementation advantages as μ-law companding. A-law companding is approximated by linear pieces, with the first segment defined to be exactly linear. A zero-level output value is undefined for the first quantization level. Biasing of the input integer is not required, but the maximum allowable input is reduced to 4096_{10}. This results in a larger minimum step size of $2/4096_{10}$, leading to a higher quantization error. Subsequently, A-law companding produces small amplitude signals of lower quality than μ-law companding. However, A-law companding yields a slightly higher dynamic range, as illustrated by the following equation:

$$DR = 20 \log_{10} (4096_{10}/15_{10}) = 48.7_{10} \text{ dB} \quad \textbf{Equation (5)}$$

where 15_{10} is the lowest amplitude spanning the first segment.

The two companding standards may also be compared with respect to precision of their binary integer representations. For both standards, up to five bits of precision are retained: the 4-bit quantization value, and the leading “1” (with the exception of values within segment zero). Thus, for μ-law companding, up to eight bits of precision are lost, while a maximum of seven bits of precision are lost for A-law companding.

While a single procedure may be devised for μ-law compression, an additional one may be needed for A-law compression. For input values of magnitude greater than $0x1F_{16}$, a procedure to determine segment and quantization values similar to μ -law compression may be implemented. For input values of magnitude less than or equal to $0x1F_{16}$, the segment is equal to 000, and the quantization is equal to the resulting four least significant bits after dividing the integer magnitude by two.

The translation from linear to A-law compression is illustrated in Table A–3. Of the compressed codeword, bits 0–3 represent the *quantization* and bits 4–6 represent the *segment*. The sign of the compressed codeword is left out for simplicity.

Table A–3. A-Law Binary Encoding Table

Input Values													Compressed Code Word						
													Chord			Step			
Bit:	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Bit:**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	0	0	0	0	0	0	a	b	c	d	x		0	0	0	a	b	c	d
	0	0	0	0	0	0	1	a	b	c	d	x		0	0	1	a	b	c	d
	0	0	0	0	0	1	a	b	c	d	x	x		0	1	0	a	b	c	d
	0	0	0	0	1	a	b	c	d	x	x	x		0	1	1	a	b	c	d
	0	0	0	1	a	b	c	d	x	x	x	x		1	0	0	a	b	c	d
	0	0	1	a	b	c	d	x	x	x	x	x		1	0	1	a	b	c	d
	0	1	a	b	c	d	x	x	x	x	x	x		1	1	0	a	b	c	d
	1	a	b	c	d	x	x	x	x	x	x	x		1	1	1	a	b	c	d

Prior to transmission, an inversion pattern applied to the codeword to improve performance. For A-law companding, the pattern is every other bit starting with the least significant bit zero, as illustrated in Table A–3.

A-law expansion can be defined mathematically by the following continuous equation:

$$F^{-1}(y) = \begin{cases} \text{sgn}(y)\,|y|\,[1+\ln(A)]/A, & 0 \le |y| \le 1/(1+\ln(A)) \\ \text{sgn}(y)\,e^{(|y|[1+\ln(A)]-1)}/[A+A\ln(A)], & 1/(1+\ln(A)) \le |y| \le 1 \end{cases} \quad \textbf{Equation (6)}$$

Prior to expansion, the inversion pattern is reapplied to the A-law codeword. As in μ-law expansion, the least significant bits are discarded but are approximated by the median interval, to reduce the loss in accuracy. For example, if five of the least significant bits of the original integer were discarded during compression, 10000_2 will approximate them during expansion. The translation from A-law to linear expansion is illustrated in Table A–4. Again, the sign bits are left out for simplicity. After decoding the A-law codeword, the sign bit is applied to obtain the final linear value.

Table A–4. A-Law Binary Decoding Table

Compressed Code Word								Biased Output Values												
Chord				Step																
Bit:	6	5	4	3	2	1	0	Bit:	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	a	b	c	d		0	0	0	0	0	0	0	a	b	c	d	1
	0	0	1	a	b	c	d		0	0	0	0	0	0	1	a	b	c	d	1
	0	1	0	a	b	c	d		0	0	0	0	0	1	a	b	c	d	1	0
	0	1	1	a	b	c	d		0	0	0	0	1	a	b	c	d	1	0	0
	1	0	0	a	b	c	d		0	0	0	1	a	b	c	d	1	0	0	0
	1	0	1	a	b	c	d		0	0	1	a	b	c	d	1	0	0	0	0
	1	1	0	a	b	c	d		0	1	a	b	c	d	1	0	0	0	0	0
	1	1	1	a	b	c	d		1	a	b	c	d	1	0	0	0	0	0	0

A.3 Implementation Using the TMS320C6000 DSP

The input value is passed into the subroutine by register A4. For compression, the input is assumed to be right justified and is sign-extended to 16-bits. The returned value is an unsigned, 8-bit, right-justified value. For expansion, the input is assumed to be an 8-bit, right-justified value that is zero-extended to 16 bits. The subroutines employ single assignment, so interrupts need not be disabled prior to execution. The compression and expansion routines perform the necessary calculations in seven and six execute packets respectively. Performing the calculations within the registers of the TMS320C6000 core conserves data memory. The results are returned via register A4. Finally, the subroutine is exited by branching to the contents of register B3, which contains the return address. The TMS320C6000 implementations make extensive use of the LMBD and EXTU instructions. The former detects where a left-most “1” first appears in a biased word during compression. The latter is used for extracting the sign, segment, and quantization bits during expansion. **Together, these instructions save many shift and test operations, as well as look-up table memory.** The routines also make extensive use of the parallelism capabilities of the TMS320C6000 DSP. **Up to eight instructions may be executed simultaneously to form an execute packet, resulting in fewer machine cycles.** For example, the μ-law and A-law compression routines are implemented in 20 and 21 instructions respectively, but only require seven machine cycles to execute. Similarly, the μ-law and A-law expansion routines are implemented in 11 and 14 instructions respectively, but only require six machine cycles to execute.

Further discussion of system requirements with respect to TMS320C6000 coding schemes is presented in section A.4. The actual coding schemes implemented for μ-law and A-law companding are provided in Appendix B.

A.4 System Requirements vs. Coding Scheme

The major considerations in implementing the companding routines are program overhead, memory usage, and speed of operation. Overhead consists mainly of context saves and restores, and depends largely on how the routines are invoked, and which registers are in use by other routines in the system. If certain registers are not used by other routines, they may be initialized once, sparing the context save and restore overhead. These routines were designed to be C-callable, whereby registers A0–A9 and B0–B9 are automatically saved by the calling program. Registers A10–A15 or B10–B15 are not used, so the routines need not perform any context saves or restores. In the worst case, registers A0–A4 and B0–B5 are used, and the majority of the routines use considerably less saves and restores. Hence, only up to eleven registers would need to be saved and restored if a routine is to be called from assembly language. Memory usage is confined solely to program instructions, ranging from 11 to 21 words. The overall design goal is to minimize potential overhead and memory utilization, while maximizing execution speed.

Companding is usually performed by using either a look-up table method, or by direct implementation. Each method has is advantages and disadvantages, with respect to overhead, memory usage, and speed issues. The look-up table method is simplest, but is the most memory intensive. In addition to program memory, two 256-word tables are necessary to perform companding. Coding would consist of loading the table’s base address into a register, adding the data sample as an offset, then retrieving the appropriate codeword. Additional overhead would be required for placing the tables into memory.

Companding may be implemented directly by using mathematical equations (often called direct encoding), or by creating simplified algorithms. Either direct method limits memory requirements to program memory, and overhead consists of register initializations. The enclosed routines are based upon simplified algorithms, and perform companding using 11 to 21 instruction words requiring 6 to 7 machine cycles. Maximum performance is achieved by placing the routines within internal program memory. For compression, it is assumed that the input data is sign-extended. For all routines, it is assumed that the input data is right-justified. Integrating the assembly routines with other system functions can further reduce overhead and increase performance.

Sections A.5, A.6, A.7, and A.8 contain detailed discussions of each routine. These sections focus on the execute packet level, highlighting details for each instruction within a given execute packet. The TMS320C6000 assembly language source code files are listed in Appendix B.

A.5 μ-Law Compression (Seven Execute Packets)

The **first execute packet** contains four operations. The input data is assumed to be right-justified and sign-extended. The first instruction takes the absolute value of the input data, and stores the results in register A0. A test is made to see if the input data is actually less than zero. Registers are initialized for later use. Register A1 contains the value 26_{10}, which is later used for determining the quantization value. Register B2 contains the maximum input value, referred to as a saturation rail. In the case of μ-law, this would be 0x1FFF minus the bias value 33_{10} used in the encoding process.

The **second execute packet** contains three operations. The bias value 33_{10} is added to the absolute value of the input. The branch to the return address is initiated, which takes six machine cycles to complete. The absolute value of the input is also compared to the saturation rail. If it is greater than the rail, a “1” is stored in register B1, to be used later.

The **third execute packet** contains three operations. The biased input is right-shifted by one bit, and the results are stored in register A0. It may be further right-shifted in future execute packets until the proper quantization bits are shifted into position. Register B2 is initialized with 25_{10} for later use. A left-most bit detect (LMBD) is performed on the biased input, and the results are stored in register A2. This will determine where a “1” first occurs, starting from the left side of the data. All user registers are 32 bits for the TMS320C6000. Therefore, a LMBD of “00001xxxxxxxxxxxxxxxxxxxxxxxxxxx” would return four when detecting a “1” (one).

The **fourth execute packet** contains three operations. The LMBD instruction is useful for determining the segment and quantization values, but does not do so directly. For example, to obtain a segment value of seven, a biased input value of “1abcdxxxxxxxx” would be required. This would produce a LMBD value of 19_{10}. To obtain the quantization bits “abcd”, the biased input value must be right-shifted by seven more bits (recall that it was right-shifted once in the previous execute packet). This value is determined in the first instruction by subtracting the LMBD results from 26_{10} (stored in register A1). However, right-shifting the biased input produces a “1” in the lower-most segment bit. Therefore, adding a left-shifted six would produce the correct segment value. The six is obtained in the next instruction by subtracting the LMBD results from 25_{10} (stored in register B2). The last instruction initializes register A3 with 0x7F. This value is used later for switching the polarity of the results, or for saturated output.

The **fifth execute packet** contains three operations. If the biased input is not saturated, it is right-shifted until the quantization bits occupy the least four bits of register A0. Also, if the biased input is not saturated, the segment is left-shifted by four bits. If the biased input is saturated, a value of 0x7F (from register A3) is moved into the return register A4.

The **sixth execute packet** contains two operations. If the biased input is not saturated, the quantization value is added to the segment value, and the results are stored in return register A4. Register B2 is initialized with 0xFF, for positive polarity switching.

The **seventh execute packet** contains two instructions. If the original unbiased input was negative, the result is exclusive–ORd with 0x7F (stored in register A3). Otherwise, it is exclusive–ORd with 0xFF (stored in register B2). By the end of this instruction, the program branches to the return address, previously stored in register B3 by the calling program.

The μ-law compression routine requires 20 instruction words and no data memory. Since execute packets cannot cross fetch packet boundaries, 24 words are actually occupied in the program memory of the TMS320C6000. The additional words are padded with NOP instructions. Registers A0–A4 and B0–B4 are used. If the routine is to be called from C, these registers need not be preserved on the stack. The routine executes in seven machine cycles. For a 200-MHz device, this results in a 35-ns execution time. Assuming an 8-KHz sampling rate, this routine requires only 0.056 MIPS.

A.6 μ-Law Expansion (Six Execute Packets)

The **first execute packet** contains three operations. The input data is assumed to be right-justified and unsigned. The one's complement of the input is determined and stored in register A0. The EXTU instruction extracts the sign bit from the input, and stores the results in register A1. The branch to the return address is initiated, which takes six machine cycles to complete.

The **second execute packet** contains two instructions. Register A2 is initialized with 33_{10}, to be used later. The modified input is left-shifted by one bit, but the results are stored in register B0.

The **third execute packet** contains two instructions. The first EXTU instruction extracts the segment from bits four through seven of the modified input, and stores the results in register A0. The second EXTU instruction extracts the quantization from bits one through four of register B0, and stores the results in register B0.

The **fourth execute packet** contains one instruction. A bias of 33_{10} (from register A2) is added to the quantization bits "abcd" to form "1abcd1". The results are stored in register B0.

The **fifth execute packet** contains one instruction. The biased quantization value "1abcd1" (from register B0) is left-shifted by the segment value stored in register A0. For example, a segment value of seven produces "1abcd10000000". The results are stored in return register A4.

The **sixth execute packet** contains two instructions. If the input was positive (based on the value of register A1), the bias of 33_{10} is removed. If the input was negative, the expanded value is subtracted from 33_{10}. This not only removes the bias, but negates the result. By the end of this instruction, the program branches to the return address, previously stored in register B3 by the calling program.

The μ-law expansion routine requires 11 instruction words and no data memory. Since execute packets cannot cross fetch packet boundaries, 16 words are actually occupied in the program memory of the TMS320C6000. The additional words are padded with NOP instructions. Registers A0–A2, A4 and B0, B3 are used. If the routine is to be called from C, these registers need not be preserved on the stack. The routine executes in six machine cycles. For a 200-MHz device, this results in a 30-ns execution time. Assuming an 8-KHz sampling rate, this routine requires only 0.048 MIPS.

A.7 A-Law Compression (Seven Execute Packets)

The **first execute packet** contains four operations. The input data is assumed to be right-justified and sign-extended. The first instruction takes the absolute value of the input data, and stores the results in register A4. A test is made to see if the input data is actually less than zero. Registers are initialized for later use. Register A0 contains the value 26_{10}, which is later used for determining the quantization value. Register B2 contains the maximum input value, referred to as a saturation rail. In the case of A-law, this would be 0xFFF. Unlike μ-law, A-law compression does not require an input bias.

The **second execute packet** contains four operations. A left-most bit detect (LMBD) is performed on the absolute value of the input, and the results are stored in register A1. This will determine where a “1” first occurs, starting from the left side of the data. All user registers are 32 bits for the TMS320C6000. Therefore, a LMBD of “00001xxxxxxxxxxxxxxxxxxxxxxxxxxx” would return four when detecting a “1” (one). The absolute value of the input is also compared to the saturation rail. If it is greater than the rail, a “1” is stored in register B1, to be used later. Register A3 is initialized with 0xD5, which is used later for a positive polarity switch. The branch to the return address is initiated, which takes six machine cycles to complete.

The **third execute packet** contains four operations. The first is a test for the case where the absolute value of the input is “000abcdx”. This is done by comparing the results of the LMBD to the value 26_{10} (stored in register A0), and storing the results in register A2. The LMBD is useful for determining the segment and quantization values, but does not do so directly. For example, to obtain a segment value of seven, an input value of “1abcdxxxxxxx” would be required. This would produce a LMBD value of 20_{10}. To obtain the quantization bits “abcd”, the modified input value must be right-shifted by seven bits. This shift value is partially determined by subtracting the LMBD results from 26_{10} (stored in register A0), producing a value of six. This value is still useful, because it can be used towards the segment. Right-shifting the modified input produces a “1” in the lower-most segment bit. Therefore, adding a left-shifted six would produce the correct segment value. (If a value of 27_{10} were previously stored, the shift value could be completely determined. A value of 26_{10} is used instead, because it is needed to test for “000abcdx”.) Register B2 is initialized with 0x55, which is used later for a negative polarity switch.

The **fourth execute packet** contains four operations. If the absolute value of the input is “000abcdx”, register A1 is cleared (for segment=0) and register A0 is initialized with a “1”. Otherwise, a “1” is added to register A1 to complete the segment, which is stored in register A0. If the absolute value of the input is greater than the rail, register B5 is initialized with 0x7F, which is used later as a saturation value.

The **fifth execute packet** contains three operations. The absolute value of the input is right-shifted until the quantization bits are set, and the results are stored in register A4. The segment value is left-shifted by four bits, and the results are stored in register B4. However, if the modified input is greater than the rail, these operations do not occur. Instead, the saturation value is moved into return register A4.

The **sixth execute packet** contains one operation. If the absolute value of the input is less than the rail, the segment and quantization values are added, and the results are stored in register A4.

The **seventh execute packet** contains two operations. If the original input was negative, the result is exclusive–ORd with 0x55 (stored in register B2). Otherwise, it is exclusive–ORd with 0xD5 (stored in register A3). By the end of this instruction, the program branches to the return address, previously stored in register B3 by the calling program.

The A-law compression routine requires 21 instruction words and no data memory. Since execute packets cannot cross fetch packet boundaries, 24 words are actually occupied in the program memory of the TMS320C6000. The additional words are padded with NOP instructions. Registers A0–A4 and B0–B5 are used. If the routine is to be called from C, these registers need not be preserved on the stack. The routine executes in seven machine cycles. For a 200-MHz device, this results in a 35-ns execution time. Assuming an 8-KHz sampling rate, this routine only requires 0.056 MIPS.

A.8 A-Law Expansion (Six Execute Packets)

The **first execute packet** contains three operations. Since only 5-bit signed constants may be embedded in TMS320C6000 instructions, the odd bits must be uninverted in stages. The input is exclusive–ORd with 0x05, which uninverts the odd quantization bits. The results are stored in register A3. Register A0 is initialized with 0x50, which will be used to uninvert the odd segment bits. The branch to the return address is initiated, which takes six machine cycles to complete.

The **second execute packet** contains three operations. The odd segment bits of register A4 are exclusive-ORd with 0x50 (stored in register A0), and the results are stored in register A2. Register A0 is initialized with 33_{10}, to be used later. The modified quantization bits are left-shifted by one bit, and the results are stored in register B0.

The **third execute packet** contains two operations. The first EXTU instruction extracts the segment from bits four through seven of register A2, and stores the results in register A2. The second EXTU instruction extracts the quantization from bits one through four of register B0, and stores the results in register B0.

The **fourth execute packet** contains four operations. The sign bit (seven) of the original input is extracted and stored in register A1. For the case of “000abcd”, a bias of “1” is added to the quantization bits stored in register B0. Otherwise, a bias of 33_{10} (stored in register A0) is added to the quantization bits in register B0. Additionally, the segment is decremented by one for shifting later.

The **fifth execute packet** contains one operation. The biased quantization bits in register B0 are left-shifted by the contents of register A2.

The **sixth execute packet** contains one operation. If the polarity, indicated by register A1 was positive, negate the results. The branch to the return address is initiated, which takes six machine cycles to complete.

The A-law expansion routine requires 14 instruction words and no data memory. Since execute packets cannot cross fetch packet boundaries, 16 words are actually occupied in the program memory of the TMS320C6000. The additional words are padded with NOP instructions. Registers A0–A4 and B0, B3 are used. If the routine is to be called from C, these registers need not be preserved on the stack. The routine executes in six machine cycles. For a 200 MHz device, this results in a 30 nanosecond execution time. Assuming an 8 KHz sampling rate, this routine only requires 0.048 MIPS.

A.9 Summary

This application note presented companding routines for the TMS320C6000 DSP. It included descriptions of μ-law and A-law companding, as well as discussions of each routine. Table A–5 provides a summary of the memory and MIPS requirements. The values in the “Overhead” category refer to the additional instruction words incurred (padded with NOP instructions) so that execute packets do not cross fetch packet boundaries.

Table A–5. Companding Algorithms Summary

	Memory Requirements				MIPS
	Total	Data	Program	Overhead	
μ-law compression	24	0	20	4	.056
μ-law expansion	16	0	11	5	.048
A-law compression	24	0	21	3	.056
A-law expansion	16	0	14	2	.048

Appendix B Companding Sample Source Code

B.1 μ-Law Compression: int2ulaw.asm

```
*=========================================================================
*
*     TEXAS INSTRUMENTS, INC.
*
*     NAME INT2ULAW
*
*     Revision Date: 10/6/99
*
*     USAGE
*           This routine is C–callable and can be called as:
*
*           unsigned char int2ulaw(short linear);
*
*           linear = 14–bit, right–justified PCM, sign–extended to 16–bits
*                    an 8–bit unsigned right–justified value is returned
*
*           If the routine is not to be used as a C–callable function,
*           then you need to initialize all passed parameters since these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler. (See the C compiler reference
*           guide.)
*
*     C CODE
*           This is the C equivalent of the assembly code without
*           restrictions.  Note that the assembly code is hand optimized.
*
*           unsigned char int2ulaw(short linear){
*              unsigned char i, sign, segment, quant;
*              unsigned short output, absol, temp;
*
*                 absol=abs(linear)+33;
*                 temp=absol;
*                 sign=(linear >= 0)?1:0;
*                 for(i=0;i<16;i++){
*                    output=temp&0x8000;
*                    if(output)break;
*                    temp<<=1;
*                 }
*                 segment=11–i;
*                 quant=(absol>>segment)&0x0F;
*                 segment––;
*                 segment<<=4;
*                 output=segment+quant;
*                 if(absol>8191) output=0x7F;
*                 if(sign)
*                    return output^=0xFF;
*                 else
*                    return output^=0x7F;
*           }
*
```

```
*     DESCRIPTION
*           u–law binary encoding table
*           Biased Linear abs(Input)    Segment Quantization
*           0 0 0 0 0 0 0 1 a b c d x   0 0 0   a b c d
*           0 0 0 0 0 0 1 a b c d x x   0 0 1   a b c d
*           0 0 0 0 0 1 a b c d x x x   0 1 0   a b c d
*           0 0 0 0 1 a b c d x x x x   0 1 1   a b c d
*           0 0 0 1 a b c d x x x x x   1 0 0   a b c d
*           0 0 1 a b c d x x x x x x   1 0 1   a b c d
*           0 1 a b c d x x x x x x x   1 1 0   a b c d
*           1 a b c d x x x x x x x x   1 1 1   a b c d
*
*           Input:  14–bit linear PCM value in register A4
*           Output: 8–bit logarithmic u–law value in register A4
*           Return: return address in register B3
*           Regs:   code utilizes registers A0–A4 and B0–B4
*
*     TECHNIQUES
*           This code yields the smallest cycle–count for one channel by
*           using as many resources in parallel as possible.  It can be
*           modified to process several channels whereby the aggregate
*           cycle–count will be greater, but the effective cycle–count per
*           channel will be less.  This code only utilizes fixed–point
*           instructions, and can be used by any member of the TMS320C6000
*           DSP family.
*
*     ASSUMPTIONS
*           The input is assumed to be a 14–bit, right–justified value that
*           is sign–extended to 16 bits.  The returned value is unsigned,
*           8 bits, and right–justified.  This code does not employ
*           software pipelining, so interrupts need not be disabled.
*
*     MEMORY NOTE
*           This code does not utilize data memory, but does occupy 24
*           words of program memory.  All computation is performed within
*           CPU registers A0–A4 and B0–B4.
*
*     CYCLES
*           7 cycles, 35.0 nanoseconds at 200MHz, regardless of input
*
*===============================================================================
        .global _int2ulaw
        .text
*** BEGIN Benchmark Timing ***
_int2ulaw:
        ABS    .L1   A4,         A0 ;take absolute value of input
||      CMPLT  .L2x  A4, 0,      B0 ;is input<0?
||      MVK    .S1   26,         A1 ;this is used for determining segment
||      MVK    .S2   0x1FFF–33,  B2 ;set saturation rail

        ADDK   .S1   33,         A0 ;add bias to abs(input)
||      B      .S2   B3             ;begin to exit subroutine
||      CMPGTU .L2x  A0,   B2,   B1 ;is abs(input)>rail?
```

```
         SHR    .S1   A0,   1,    A0 ;start determining quantization
||       MVK    .S2   25,         B2 ;need this for shift
||       LMBD   .L1   1,    A0,   A2 ;figure where "1"s start for abs(input)

         SUB    .L1   A1,   A2,   A2 ;need this to determine quantize value
||       SUB    .L2x  B2,   A2,   B2 ;need this for segment value
||       MVK    .S1   0x7F,       A3 ;use for polarity switch or saturation

  [!B1] SHR    .S1   A0,   A2,   A0 ;set quantization value of abs(input)
||[!B1] SHL    .S2   B2,   4,    B4 ;position place segment value
||[B1]  MV     .L1   A3,         A4 ;if abs(input)>rail, use saturation

  [!B1] ADD    .L1x  B4,   A0,   A4 ;add quantization value to segment
||       MVK    .S2   0xFF,       B2 ;need this for polarity switch at end

  [B0]  XOR    .L1   A4,   A3,   A4 ;reverse polarity for negative input
||[!B0] XOR    .S1x  A4,   B2,   A4 ;reverse polarity for positive input
*** END Benchmark Timing ***
```

B.2 μ-Law Expansion: ulaw2int.asm

```
*===========================================================================
*
*     TEXAS INSTRUMENTS, INC.
*
*     NAME ULAW2INT
*
*     Revision Date: 10/6/99
*
*     USAGE
*           This routine is C–callable and can be called as:
*
*           int ulaw2int(unsigned char log);
*
*           log = 8–bit unsigned, right–justified logarithmic value
*           a 14–bit linear PCM value, sign–extended to 32 bits is returned
*
*           If the routine is not to be used as a C–callable function,
*           then you need to initialize all passed parameters since these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler, (See the C compiler reference
*           guide.)
*
*     C CODE
*           This is the C equivalent of the assembly code without
*           restrictions.  Note that the assembly code is hand optimized.
*
*           int ulaw2int(unsigned char log){
*              unsigned char sign, segment;
*              unsigned short temp, quant;
*
*                 temp=log^0xFF;
*                 sign=(temp&0x80)>>7;
*                 segment=(temp&0x70)>>4;
```

```
*                quant=temp&0x0F;
*                quant<<=1;
*                quant+=33;
*                quant<<=segment;
*                if(sign)
*                return 33–quant;
*                        else
*                return quant–33;
*          }
*
*     DESCRIPTION
*          u–law binary decoding table
*          Segment Quant.    Biased Linear abs(Output)
*          0 0 0   a b c d   0 0 0 0 0 0 0 1 a b c d 1
*          0 0 1   a b c d   0 0 0 0 0 0 1 a b c d 1 0
*          0 1 0   a b c d   0 0 0 0 0 1 a b c d 1 0 0
*          0 1 1   a b c d   0 0 0 0 1 a b c d 1 0 0 0
*          1 0 0   a b c d   0 0 0 1 a b c d 1 0 0 0 0
*          1 0 1   a b c d   0 0 1 a b c d 1 0 0 0 0 0
*          1 1 0   a b c d   0 1 a b c d 1 0 0 0 0 0 0
*          1 1 1   a b c d   1 a b c d 1 0 0 0 0 0 0 0
*
*          Input: 8–bit logarithmic u–law value in register A4
*          Output:  14–bit linear PCM value in register A4 (becomes Q31)
*          Return: return address in register B3
*          Regs:  code utilizes registers A0–A2,A4 and B0,B3
*
*     TECHNIQUES
*          This code yields the smallest cycle–count for one channel by
*          using as many resources in parallel as possible.  It can be
*          modified to process several channels whereby the aggregate
*          cycle–count will be greater, but the effective cycle–count per
*          channel will be less.  This code only utilizes fixed–point
*          instructions, and can be used by any member of the TMS320C6000
*          DSP family.
*
*     ASSUMPTIONS
*          The input is assumed to be an 8–bit, right–justified value that
*          is zero–extended to 16 bits.  The returned value is 14 bits,
*          right–justified, and sign–extended to 32 bits.  This code does
*          not employ the use of software pipelining, so interrupts need
*          not be disabled.
*
*     MEMORY NOTE
*          This code does not utilize data memory, but does occupy 16
*          words of program memory.  All computation is performed within
*          CPU registers A0–A2,A4 and B0,B3.
*
*     CYCLES
*          6 cycles, 30.0 nanoseconds at 200MHz, regardless of input
*
*===============================================================================
```

```
        .global _ulaw2int
        .text
*** BEGIN Benchmark Timing ***

_ulaw2int:
        NOT    .L1   A4,   A0       ;reverse polarity of input bits
||      EXTU   .S1   A4,24,31,   A1 ;extract sign bit
||      B      .S2   B3             ;begin to exit subroutine


        MVK    .S1   33,   A2       ;create bias for expansion
||      SHL    .S2x  A0,   1,    B0 ;left-shift quantization bits by one

        EXTU   .S1   A0,25,29,   A0 ;extract segment bits
||      EXTU   .S2   B0,27,27,   B0 ;extract quantization bits

        ADD    .S2x  B0,   A2,   B0 ;add bias to quantization bits
        SHL    .S1x  B0,   A0,   A4 ;linearize

  [!A1] SUB    .L1   A2,   A4,   A4 ;input was negative, remove bias/negate
||[A1]  SUB    .S1   A4,   A2,   A4 ;input was positive, just remove bias
*** END Benchmark Timing ***
```

B.3 A-Law Compression: int2alaw.asm

```
*=============================================================================
*
*     TEXAS INSTRUMENTS, INC.
*
*     NAME INT2ALAW
*
*     Revision Date: 10/6/99
*
*     USAGE
*           This routine is C-callable and can be called as:
*
*           unsigned char int2alaw(short linear);
*
*           linear = 13-bit, right-justified PCM, sign-extended to 16-bits
*                    an 8-bit unsigned right-justified value is returned
*
*           If the routine is not to be used as a C-callable function,
*           then you need to initialize all passed parameters since these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler, (See the C compiler reference
*           guide.)
*
*     C CODE
*           This is the C equivalent of the assembly code without
*           restrictions.  Note that the assembly code is hand optimized.
*
*           unsigned char int2alaw(short linear){
*              char segment;
*              unsigned char i, sign,quant;
*              unsigned short output, absol, temp;
*
```

```
*                temp=absol=abs(linear);
*                sign=(linear >= 0)?1:0;
*                for(i=0;i<16;i++){
*                   output=temp&0x8000;
*                   if(output)break;
*                   temp<<=1;
*                }
*                segment=11–i;
*                if(segment<=0){
*                   segment=0;
*                   quant=(absol>>1)&0x0F;
*                   }
*                else
*                   quant=(absol>>segment)&0x0F;
*                segment<<=4;
*                output=segment+quant;
*                if(absol>4095) output=0x7F;
*                if(sign)
*                   return output^=0xD5;
*                else
*                   return output^=0x55;
*          }
*
*     DESCRIPTION
*          A–law binary encoding table
*          Linear abs(Input)         Segment Quantization
*          0 0 0 0 0 0 0 a b c d x   0 0 0   a b c d
*          0 0 0 0 0 0 1 a b c d x   0 0 1   a b c d
*          0 0 0 0 0 1 a b c d x x   0 1 0   a b c d
*          0 0 0 0 1 a b c d x x x   0 1 1   a b c d
*          0 0 0 1 a b c d x x x x   1 0 0   a b c d
*          0 0 1 a b c d x x x x x   1 0 1   a b c d
*          0 1 a b c d x x x x x x   1 1 0   a b c d
*          1 a b c d x x x x x x x   1 1 1   a b c d
*
*          Input:  13–bit linear PCM value in register A4
*          Output: 8–bit logarithmic A–law value in register A4
*          Return: return address in register B3
*          Regs:  code utilizes registers A0–A4 and B0–B5
*
*
*     TECHNIQUES
*          This code yields the smallest cycle–count for one channel by
*          using as many resources in parallel as possible.  It can be
*          modified to process several channels whereby the aggregate
*          cycle–count will be greater, but the effective cycle–count per
*          channel will be less.  This code only utilizes fixed–point
*          instructions, and can be used by any member of the TMS320C6000
*          DSP family.
*
```

```
*     ASSUMPTIONS
*           The input is assumed to be a 13–bit, right–justified value that
*           is sign–extended to 16 bits.  The returned value is unsigned,
*           8 bits, and right–justified.  This code does not employ the use
*           of software pipelining, so interrupts need not be disabled.
*
*     MEMORY NOTE
*           This code does not utilize data memory, but does occupy 24
*           words of program memory.  All computation is performed within
*           CPU registers A0–A4 and B0–B5.
*
*     CYCLES
*           7 cycles, 35.0 nanoseconds at 200MHz, regardless of input
*
*=============================================================================
        .global _int2alaw
        .text
*** BEGIN Benchmark Timing ***
_int2alaw:
        ABS    .L1   A4,         A4 ;take absolute value of input
||      CMPLT  .L2x  A4,   0,    B0 ;is input<0?
||      MVK    .S1   26,         A0 ;this is used for determining segment
||      MVK    .S2   0xFFF,      B2 ;set saturation rail

        LMBD   .L1   1,    A4,   A1 ;figure where ”1”s start for abs(input)
||      CMPGTU .L2x  A4,   B2,   B1 ;is abs(input)>rail?
||      MVK    .S1   0xD5,       A3 ;need this for positive polarity switch
||      B      .S2   B3             ;begin to exit subroutine

        CMPGTU .L1   A1,   A0,   A2 ;test for 000abcdx case
||      SUB    .S1   A0,   A1,   A1 ;need this to determine segment value
||      MVK    .S2   0x55,       B2 ;need this for negative polarity switch

  [A2]  ZERO   .D1   A1             ;segment=0 for 000abcdx case
||[!A2] ADD    .L1   A1,   1,    A0 ;add one for correct shift
||[A2]  MVK    .S1   1,          A0 ;use this for 000abcdx case
||[B1]  MVK    .S2   0x7F,       B5 ;set sat. value for abs(input)>rail

  [!B1] SHR    .S1   A4,   A0,   A4 ;set quantization value of abs(input)
||[!B1] SHL    .S2x  A1,   4,    B4 ;place segment value into position
||[B1]  MV     .L1x  B5,         A4 ;if abs(input)>rail, use saturation

  [!B1] ADD    .L1x  B4,   A4,   A4 ;add quantization value to segment

  [!B0] XOR    .L1   A4,   A3,   A4 ;invert odd bits for positive input
||[B0]  XOR    .S1x  A4,   B2,   A4 ;invert odd bits for negative input
*** END Benchmark Timing ***
```

B.4 A-Law Expansion: alaw2int.asm

```
*==============================================================================
*
*     TEXAS INSTRUMENTS, INC.
*
*     NAME ALAW2INT
*
*     Revision Date: 10/6/99
*
*     USAGE
*           This routine is C–callable and can be called as:
*
*           int alaw2int(unsigned char log);
*
*           log = 8–bit unsigned, right–justified logarithmic value
*           a 13–bit linear PCM value, sign–extended to 32 bits is returned
*
*           If the routine is not to be used as a C–callable function,
*           then you need to initialize all passed parameters since these
*           are assumed to be in registers as defined by the calling
*           convention of the compiler, (See the C compiler reference
*           guide.)
*
*     C CODE
*           This is the C equivalent of the assembly code without
*           restrictions.  Note that the assembly code is hand optimized.
*
*           int alaw2int(unsigned char log){
*              unsigned char sign, segment;
*              unsigned short temp, quant;
*
*                 temp=log^0xD5;
*                 sign=(temp&0x80)>>7;
*                 segment=(temp&0x70)>>4;
*                 quant=temp&0x0F;
*                 quant<<=1;
*                 if(!segment)
*                    quant+=1;
*                 else{
*                    quant+=33;
*                    quant<<=segment–1;
*                 }
*                 if(sign)
*                    return –quant;
*                 else
*                    return quant;
*           }
*
```

```
*     DESCRIPTION
*           A–law binary decoding table
*           Segment Quant.    Biased Linear abs(Output)
*           0 0 0   a b c d   0 0 0 0 0 0 0 a b c d 1
*           0 0 1   a b c d   0 0 0 0 0 0 1 a b c d 1
*           0 1 0   a b c d   0 0 0 0 0 1 a b c d 1 0
*           0 1 1   a b c d   0 0 0 0 1 a b c d 1 0 0
*           1 0 0   a b c d   0 0 0 1 a b c d 1 0 0 0
*           1 0 1   a b c d   0 0 1 a b c d 1 0 0 0 0
*           1 1 0   a b c d   0 1 a b c d 1 0 0 0 0 0
*           1 1 1   a b c d   1 a b c d 1 0 0 0 0 0 0
*
*           Input: 8–bit logarithmic A–law value in register A4
*           Output:  13–bit linear PCM value in register A4 (becomes Q31)
*           Return: return address in register B3
*           Regs:  code utilizes registers A0–A4 and B0,B3
*
*     TECHNIQUES
*           This code yields the smallest cycle–count for one channel by
*           using as many resources in parallel as possible.  It can be
*           modified to process several channels whereby the aggregate
*           cycle–count will be greater, but the effective cycle–count per
*           channel will be less.  This code only utilizes fixed–point
*           instructions, and can be used by any member of the TMS320C6000
*           DSP family.
*
*     ASSUMPTIONS
*           The input is assumed to be an 8–bit, right–justified value that
*           is zero–extended to 16 bits.  The returned value is 13 bits,
*           right–justified, and sign–extended to 32 bits.  This code does
*           not employ the use of software pipelining, so interrupts need
*           not be disabled.
*
*     MEMORY NOTE
*           This code does not utilize data memory, but does occupy 16
*           words of program memory.  All computation is performed within
*           CPU registers A0–A4 and B0,B3.
*
*     CYCLES
*           6 cycles, 30.0 nanoseconds at 200MHz, regardless of input
*
*=============================================================================
        .global _alaw2int
        .text
*** BEGIN Benchmark Timing ***

_alaw2int:
        MVK    .S1   0x50,       A0 ;use this to uninvert odd segment bits
||      B      .S2   B3             ;begin to exit subroutine
||      XOR    .L1   A4,   0x05, A3 ;uninvert odd quantization bits

        XOR    .L1   A4,   A0,   A2 ;uninvert odd segment bits
||      SHL    .S2x  A3,   1,    B0 ;left–shift quantization bits by one
||      MVK    .S1   33,         A0 ;create bias for expansion
```

```
        EXTU   .S1   A2,25,29,   A2 ;extract segment bits
||      EXTU   .S2   B0,27,27,   B0 ;extract quantization bits

        EXTU   .S1   A4,24,31,   A1 ;extract sign bit
|||[A2]  ADD    .S2x  B0,   A0,   B0 ;add bias to quantization bits
|||[A2]  SUB    .L1   A2,   1,    A2 ;adjust segment bits before shift
|||[!A2] ADD    .L2   B0,   1,    B0 ;add 1 for 000abcd case

        SHL    .S1x  B0,   A2,   A4 ;linearize

  [!A1] NEG    .S1   A4,         A4 ;input was positive, so negate
*** END Benchmark Timing ***
```

Appendix C Sample C Functions – McBSP and DMA Initialization

```
/*****************************************************************/
/*    mcbsp.c                                                    */
/*****************************************************************/

#include <mcbsp.h>
#include <dma.h>
#include <stdlib.h>

/* Definitions */
#define MEM_SRC        0x80000000
#define MEM_DST        0x80006000
#define EL_COUNT       256

/* Global variables */
int error       = 0;
int DMA_done[4] = {0, 0, 0, 0};

/* Prototypes */
extern void set_interrupts(void);
void cfg_sp_data(void);
void start_serial(void);
void wait_sp(void);
void start_sp_dma(void);


void
wait_sp(void)
{

      /* Wait until transfer completes */
      while (!DMA_done[1]);

      /* Disable McBSP transfers */
      MCBSP_TX_RESET(0);
      MCBSP_RX_RESET(0);
      MCBSP_TX_RESET(1);
      MCBSP_RX_RESET(1);

} /* end wait_sp */


void
cfg_sp_data(void)
{
unsigned short  *val;
unsigned int   i = 0;
```

```
        val = (unsigned short *)MEM_SRC;

        /* Set up transfer data */
        for (i = 0; i < EL_COUNT; i++){
              *val++ = ((i<<16) + i) <<3;
        } /* end for */

} /* end cfg_sp_data */


void
start_sp_dma(void)
{
unsigned int   dma_pri_ctrl   = 0;
unsigned int   dma_sec_ctrl   = 0;
unsigned int   dma_src_addr   = 0;
unsigned int   dma_dst_addr   = 0;
unsigned int   dma_tcnt       = 0;

        /* Clear completion flag */
        DMA_done[1] = 0;

        /* Reset DMA Control Registers */
        /* use DMA Ch0 to McBSP, Ch1 from McBSP */
        dma_reset();
        DMA_RSYNC_CLR(0);
        DMA_WSYNC_CLR(0);
        DMA_RSYNC_CLR(1);
        DMA_WSYNC_CLR(1);
        dma_reset();


        /* Set up DMA Channel to perform a block transfer of         */
        /* XFER_SIZE elements                                        */
        /*   from MEM_SRC to McBSP */
        /* Set up DMA Primary Control Register */
        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_NONE, DST_RELOAD,DST_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_NONE, SRC_RELOAD,SRC_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_DMA_PRI    , PRI       , 1           );
        LOAD_FIELD(&dma_pri_ctrl, SEN_XEVT0      , WSYNC     , WSYNC_SZ    );
        LOAD_FIELD(&dma_pri_ctrl, SEN_NONE       , RSYNC     , RSYNC_SZ    );
        LOAD_FIELD(&dma_pri_ctrl, DMA_SPLIT_DIS  , SPLIT     , SPLIT_SZ    );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ESIZE16     , ESIZE     , ESIZE_SZ   );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_NO_MOD, DST_DIR   , DST_DIR_SZ  );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_INC   , SRC_DIR   , SRC_DIR_SZ  );
        SET_BIT(&dma_pri_ctrl,EMOD);         /* Halt DMA with emu halt     */

        /* Set up DMA Secondary Control Register */
        LOAD_FIELD(&dma_sec_ctrl, DMAC_BLOCK_COND, DMAC_EN  , DMAC_EN_SZ   );
        SET_BIT(&dma_sec_ctrl, BLOCK_IE);

        /* Set up DMA Tranfer Count Register */
        LOAD_FIELD(&dma_tcnt, 0        , FRAME_COUNT  , FRAME_COUNT_SZ  );
        LOAD_FIELD(&dma_tcnt, EL_COUNT , ELEMENT_COUNT, ELEMENT_COUNT_SZ);
```

```
        /* Set up Source and Destination Address Registers */
        dma_src_addr = (unsigned int)MEM_SRC;
        dma_dst_addr = (unsigned int)MCBSP_DXR_ADDR(0);

        /* Store DMA Control registers */
        dma_init(0,
              dma_pri_ctrl,
              dma_sec_ctrl,
              dma_src_addr,
              dma_dst_addr,
              dma_tcnt);


        /*Modify DMA setup for second channel to service data transfer from*/
        /* McBSP to MEM_DST                                                */

        /* Set up DMA Primary Control Register */
        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_NONE, DST_RELOAD,DST_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_NONE, SRC_RELOAD,SRC_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_DMA_PRI    , PRI       , 1          );
        LOAD_FIELD(&dma_pri_ctrl, SEN_NONE       , WSYNC     , WSYNC_SZ   );
        LOAD_FIELD(&dma_pri_ctrl, SEN_REVT0      , RSYNC     , RSYNC_SZ   );
        LOAD_FIELD(&dma_pri_ctrl, DMA_SPLIT_DIS  , SPLIT     , SPLIT_SZ   );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ESIZE16     , ESIZE     , ESIZE_SZ  );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_INC, DST_DIR   , DST_DIR_SZ    );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_NO_MOD   , SRC_DIR, SRC_DIR_SZ );
        SET_BIT(&dma_pri_ctrl,EMOD);         /* Halt DMA with emu halt     */
        SET_BIT(&dma_pri_ctrl,TCINT);        /* Allow Ch to interrupt CPU  */

        /* Modify Source and Destination Address Registers */
        dma_src_addr = (unsigned int)MCBSP_DRR_ADDR(0);
        dma_dst_addr = (unsigned int)MEM_DST;

        /* Store DMA Control registers */
        dma_init(1,
              dma_pri_ctrl,
              dma_sec_ctrl,
              dma_src_addr,
              dma_dst_addr,
              dma_tcnt);

        /* Start DMA Transfer */
        DMA_START(0);
        DMA_RSYNC_CLR(0);
        DMA_WSYNC_CLR(0);
        DMA_START(1);
        DMA_RSYNC_CLR(1);
        DMA_WSYNC_CLR(1);

    } /* end start_sp_dma */
```

```
void
start_serial(void)
{
unsigned int spcr = 0;
unsigned int rcr  = 0;
unsigned int xcr  = 0;
unsigned int srgr = 0;
unsigned int mcr  = 0;
unsigned int rcer = 0;
unsigned int xcer = 0;
unsigned int pcr  = 0;


      /* Set up Pin Control Register */
      LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSXM , 1);
      LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSRM , 1);
      LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKXM, 1);
      LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKRM, 1);
      LOAD_FIELD(&pcr, FSYNC_POL_HIGH  , FSXP , 1);
      LOAD_FIELD(&pcr, FSYNC_POL_HIGH  , FSRP , 1);
      LOAD_FIELD(&pcr, CLKX_POL_RISING , CLKXP, 1);
      LOAD_FIELD(&pcr, CLKR_POL_FALLING, CLKRP, 1);

      /* Set up Receive Control Register */
      LOAD_FIELD(&rcr, SINGLE_PHASE    , RPHASE, 1);
      LOAD_FIELD(&rcr, FRAME_IGNORE    , RFIG , 1);
      LOAD_FIELD(&rcr, DATA_DELAY1     , RDATDLY, RDATDLY_SZ);
      LOAD_FIELD(&rcr, 0               , RFRLEN1, RFRLEN1_SZ);
      LOAD_FIELD(&rcr, WORD_LENGTH_8   , RWDLEN1, RWDLEN1_SZ);
      LOAD_FIELD(&rcr, COMPAND_ALAW    , RCOMPAND, RCOMPAND_SZ);

      /* Set up Transmit Control Register */
      LOAD_FIELD(&xcr, SINGLE_PHASE    , XPHASE, 1);
      LOAD_FIELD(&xcr, FRAME_IGNORE    , XFIG , 1);
      LOAD_FIELD(&xcr, DATA_DELAY1     , XDATDLY, XDATDLY_SZ);
      LOAD_FIELD(&xcr, 0               , XFRLEN1, XFRLEN1_SZ);
      LOAD_FIELD(&xcr, WORD_LENGTH_8   , XWDLEN1, XWDLEN1_SZ);
      LOAD_FIELD(&xcr, COMPAND_ALAW    , XCOMPAND, XCOMPAND_SZ);

      /* Set up Serial Port Control Register */
      LOAD_FIELD(&spcr, INTM_RDY       , XINTM, XINTM_SZ );
      LOAD_FIELD(&spcr, INTM_RDY       , RINTM, RINTM_SZ );
      /* uncomment the following line to test in DLB mode */
      //LOAD_FIELD(&spcr, DLB_ENABLE     , DLB, 1);

      /* Set up Sample Rate Generator Register */
      SET_BIT(&srgr, CLKSM);              /* CLKG derived from CPU clock */
      LOAD_FIELD(&srgr, FSX_DXR_TO_XSR, FSGM, 1);
      LOAD_FIELD(&srgr, 40, CLKGDV, CLKGDV_SZ);

      /* Store McBSP 0 registers */
      mcbsp_init(0, spcr, rcr, xcr, srgr, mcr, rcer, xcer, pcr);
```

```
      /* Bring McBSPs out of reset */

      MCBSP_SAMPLE_RATE_ENABLE(0);    /* Start Sample Rate Generator */
      MCBSP_FRAME_SYNC_ENABLE(0);     /* Enable Frame Sync pulse     */
      MCBSP_ENABLE(0, MCBSP_RX);      /* Bring Receive out of reset  */
      MCBSP_ENABLE(0, MCBSP_TX);      /* Bring Transmit out of reset */

} /* End start_serial */


/* McBSP verification test code.                                    */
void
main (void)
{

      set_interrupts();
      cfg_sp_data();           /* Configure data to be transferred */
      start_sp_dma();
      start_serial();
      wait_sp();

} /* end main */
/*********************************************************************/
/*    dma_int.c                                                      */
/*********************************************************************/

#include <intr.h>
#include <dma.h>

/* Global variables */
extern int DMA_done[4];

/* Prototypes */
interrupt void DMA_Ch0_ISR(void);
interrupt void DMA_Ch1_ISR(void);
interrupt void DMA_Ch2_ISR(void);
interrupt void DMA_Ch3_ISR(void);
void set_interrupts(void);

/* DMA Ch0 ISR used to clear block condition and flag when the     */
/* transfer has completed.                                         */
interrupt void
DMA_Ch0_ISR(void)
{
unsigned int sec_ctrl = 0x50000;

      sec_ctrl = REG_READ(DMA0_SECONDARY_CTRL_ADDR);
      RESET_BIT(&sec_ctrl, FRAME_COND);
      if (GET_BIT(&sec_ctrl, BLOCK_COND)){
            DMA_done[0] = 1;
            RESET_BIT(&sec_ctrl, BLOCK_COND);
      }
```

```
      else SET_BIT(&sec_ctrl, RSYNC_STAT);
      REG_WRITE(DMA0_SECONDARY_CTRL_ADDR, sec_ctrl);

} /* End DMA_Ch0_ISR */


/* DMA Ch1 ISR used to clear block condition and flag when the     */
/* transfer has completed.                                         */
interrupt void
DMA_Ch1_ISR(void)
{
unsigned int sec_ctrl = 0;

      sec_ctrl = REG_READ(DMA1_SECONDARY_CTRL_ADDR);
      RESET_BIT(&sec_ctrl, FRAME_COND);
      if (GET_BIT(&sec_ctrl, BLOCK_COND)){
            DMA_done[1] = 1;
            RESET_BIT(&sec_ctrl, BLOCK_COND);
      }
      else SET_BIT(&sec_ctrl, RSYNC_STAT);
      REG_WRITE(DMA1_SECONDARY_CTRL_ADDR, sec_ctrl);

} /* End DMA_Ch1_ISR */


/* DMA Ch2 ISR used to clear block condition and flag when the     */
/* transfer has completed.                                         */
interrupt void
DMA_Ch2_ISR(void)
{
unsigned int sec_ctrl = 0;

      sec_ctrl = REG_READ(DMA2_SECONDARY_CTRL_ADDR);
      RESET_BIT(&sec_ctrl, FRAME_COND);
      if (GET_BIT(&sec_ctrl, BLOCK_COND)){
            DMA_done[2] = 1;
            RESET_BIT(&sec_ctrl, BLOCK_COND);
      }
      else SET_BIT(&sec_ctrl, RSYNC_STAT);
      REG_WRITE(DMA2_SECONDARY_CTRL_ADDR, sec_ctrl);

} /* End DMA_Ch2_ISR */


/* DMA Ch3 ISR used to clear block condition and flag when the     */
/* transfer has completed.                                         */
interrupt void
DMA_Ch3_ISR(void)
{
unsigned int sec_ctrl = 0x50000;

      sec_ctrl = REG_READ(DMA3_SECONDARY_CTRL_ADDR);
      RESET_BIT(&sec_ctrl, FRAME_COND);
```

```
        if (GET_BIT(&sec_ctrl, BLOCK_COND)){
              DMA_done[3] = 1;
              RESET_BIT(&sec_ctrl, BLOCK_COND);
        }
        else SET_BIT(&sec_ctrl, RSYNC_STAT);
        REG_WRITE(DMA3_SECONDARY_CTRL_ADDR, sec_ctrl);

} /* End DMA_Ch3_ISR */


/* Routine to enable DMA and Timer interrupt service routines      */
void
set_interrupts(void)
{
        intr_init();
        intr_map(CPU_INT8, ISN_DMA_INT0);
        intr_hook(DMA_Ch0_ISR, CPU_INT8);
        intr_map(CPU_INT9, ISN_DMA_INT1);
        intr_hook(DMA_Ch1_ISR, CPU_INT9);
        intr_map(CPU_INT11, ISN_DMA_INT2);
        intr_hook(DMA_Ch2_ISR, CPU_INT11);
        intr_map(CPU_INT12, ISN_DMA_INT3);
        intr_hook(DMA_Ch3_ISR, CPU_INT12);
        INTR_GLOBAL_ENABLE();
        INTR_ENABLE(CPU_INT_NMI);
        INTR_ENABLE(CPU_INT8);
        INTR_ENABLE(CPU_INT9);
        INTR_ENABLE(CPU_INT11);
        INTR_ENABLE(CPU_INT12);

} /* End set_interrupts */
```

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products 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.

Customers are responsible for their applications using TI components.

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

TI assumes no liability for applications assistance or customer product design. TI does not 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. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.