TEXAS INSTRUMENTS

Real-Time Wavelet-Based Video Compression System Using Multiple TMS320C40s

Application Report

1997 Digital Signal Processing Solutions

Printed in U.S.A., June 1997

SPRA165

Real-Time Wavelet-Based Video Compression System Using Multiple TMS320C40s

Dr. Nariman Farvardin

Hamid Jafarkhani, Jerome Johnson, and Ruplu Bhattacharya
University of Maryland, Department of Electrical Engineering

Literature Number: SPRA165
June 1997

Contents

1 Introduction **2**

2 Description of the Algorithm **3**

- 2.1 Encoder 5
  - 2.1.1 2-D Discrete-Wavelet Transform (2-D DWT) 6
  - 2.1.2 Mean and Variance Estimation 8
  - 2.1.3 Bit Allocation 8
  - 2.1.4 Normalization 9
  - 2.1.5 Quantization 9
  - 2.1.6 Huffman Encoding 10
  - 2.1.7 Multiplexing 10
- 2.2 Decoder 11
  - 2.2.1 Demultiplexing 11
  - 2.2.2 Huffman Decoding 11
  - 2.2.3 Dequantization 11
  - 2.2.4 Denormalization 11
  - 2.2.5 2-D Inverse DWT 11

3 Implementation on the TMS320C40 DSP **13**

- 3.1 Video Capture and Display 13
- 3.2 Discrete-Wavelet Transform 14
  - 3.2.1 Use of the DMA 14
  - 3.2.2 FIR Assembly Routine 14
- 3.3 Performance Data 14
- 3.4 Multiple Processor Implementation on the PPDS 15
  - 3.4.1 Parallel Organization 15
  - 3.4.2 Data Path 17
  - 3.4.3 Timing/Synchronization of Data Transfers 17

4 Summary **19**

References **20**

Appendix A Code Listings **A-1**

- A.1 send.c A-2
- A.2 receive.c A-6
- A.3 single.c A-10
- A.4 decomp.c A-15
- A.5 recomp.c A-18
- A.6 cvnld.asm A-23
- A.7 cnvlr.asm A-26
- A.8 bitalloc.c A-28
- A.9 utsqen.c A-29
- A.10 utsqde.c A-36
- A.11 bstrm.c A-40
- A.12 grab.c A-45
- A.13 unpackf.asm A-46

A.14 disp.c A-47
A.15 packf.asm A-49
A.16 adma.c A-51
A.17 bdma.c A-52
A.18 cdma.c A-54
A.19 ddma.c A-55
A.20 sdma.c A-56
A.21 options.h A-58
A.22 waveletc.h A-58
A.23 utsq.h A-59
A.24 mean.h A-60
A.25 bstrm.h A-61
A.26 grab.h A-62
A.27 ports.h A-62
A.28 sects.h A-63
A.29 average.h A-64
A.30 meml.cmd A-64
A.31 sendc.cmd A-64
A.32 sendd.cmd A-65
A.33 recea.cmd A-66
A.34 receb.cmd A-67
A.35 single.cmd A-68
A.36 comptbls.cmd A-69
A.37 comp4.cmd A-69
A.38 comp1.cmd A-69

List of Figures

1 Block Diagram of the Encoder 4
2 Block Diagram of the Decoder 5
3 Discrete-Wavelet Transform 7
4 Inverse Discrete-Wavelet Transform 12
5 Hardware Diagram 16

Real-Time Wavelet-Based Video Compression System Using Multiple TMS320C40s

ABSTRACT

The objective of this application report is the implementation of a video coding system of quarter common intermediate format (QCIF) (180x144 pixels) on a Texas Instruments (TI™)[1] TMS320C4x parallel processing development system (PPDS). The target bit rates are 64 to 384K bits per second (Kbps). The system uses the discrete-wavelet transform to decompose video frames into 10 subbands. Each of the subbands is then encoded separately using uniform-threshold scalar quantizers, followed by Huffman coding. The required rate for quantizing the coefficients of each subband is determined by an optimal bit-allocation procedure. The video frames are reconstructed by performing the inverse-filtering operation on the encoded coefficients. The goal is to perform the decomposition, quantization, and reconstruction in real time on the PPDS. The decomposition process is performed on two digital signal processors (DSPs); the other two DSPs are used for reconstructing the frames simultaneously.

1. TI is a trademark of Texas Instruments Incorporated.

1 Introduction

This report describes the design and implementation of a video teleconferencing system on a Texas Instruments TMS320C40 PPDS consisting of four TMS320C40 DSPs. The compression algorithm is based on the discrete-wavelet transform (DWT) coding scheme.

The project implements a video-compression/-decompression scheme successfully in real time. The video sequence is a full-motion (15 frames per second) of QCIF size, where each frame consists of 176 x 144 pixels monochrome. The system is very flexible and many of the parameters can be changed, depending on the application. The encoding rate is fully flexible.

The typical operating rates are 64 to 384 Kbps, providing a broad range of rate-quality tradeoffs. In the implementation, two DSP chips are used on the encoding side (DWT, mean and variance estimation, normalization, bit allocation, quantization, and Huffman coding), and the other two DSP chips are on the decoding side (Huffman decoding, dequantization, denormalization, and inverse DWT)

Due to the use of the DWT, the encoder is easily scaleable in space. Also, since no motion compensation is used, scalability in time is provided automatically. Such scalability features are highly desirable in heterogeneous networks where different link bandwidths and user resolutions coexist in the network.

To the authors’ knowledge, this is the first real-time implementation of a wavelet-based video-compression algorithm on a DSP board with the spatial and temporal resolution used here. The success of this project can pave the way for further work on DSP-based implementation of video-compression systems and can have a major impact on commercialization of video-communication products in the future.

2 Description of the Algorithm

Figure 1 shows the block diagram of the implemented encoder. Figure 2 shows the decoder. This section briefly describes each component of the encoder and decoder.

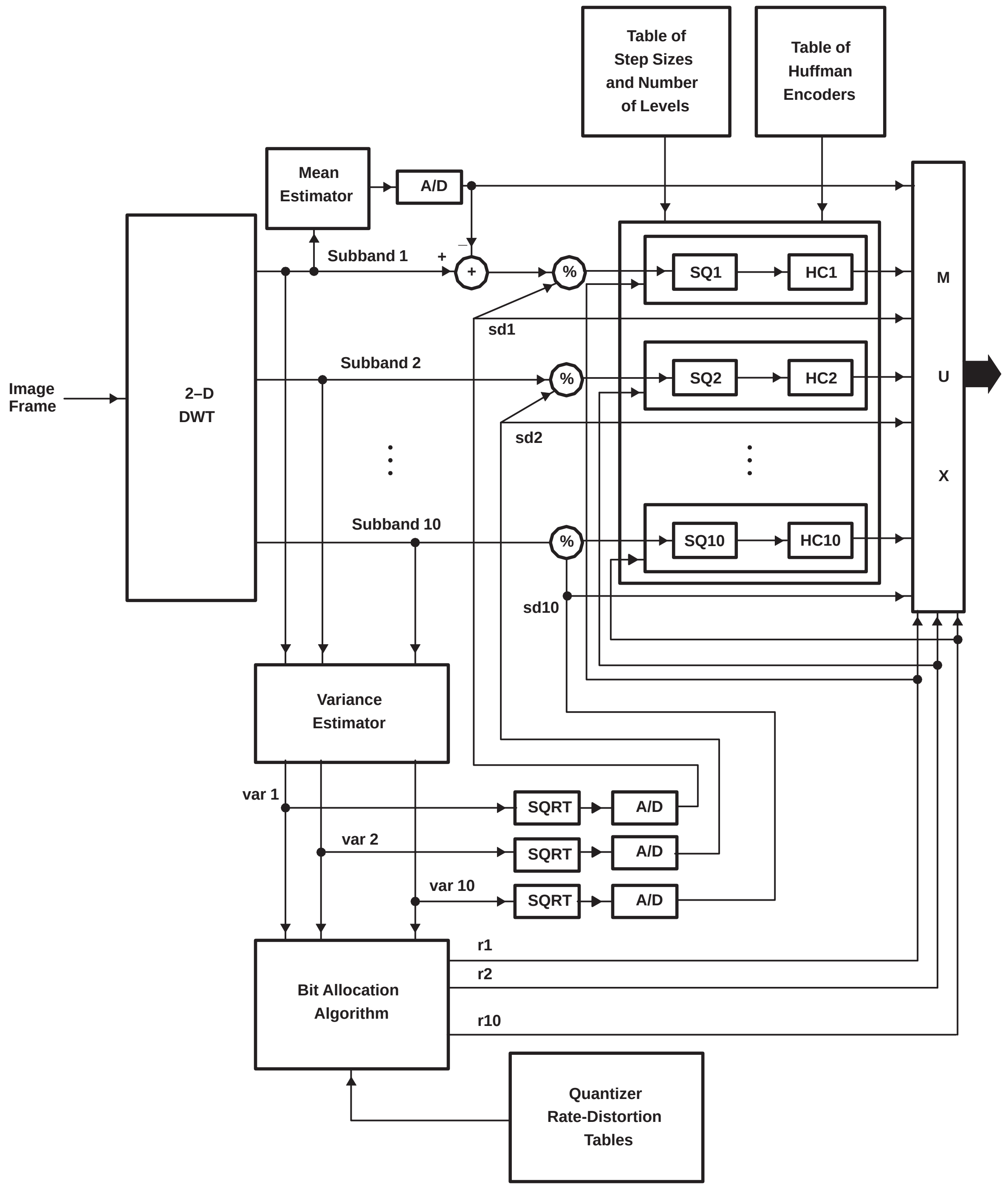


Figure 1. Block Diagram of the Encoder

2.1 Encoder

At the encoder, using the DWT, each video frame is decomposed into 10 frequency subbands. Then, each of the resulting subbands is encoded by means of an optimally designed uniform-threshold scalar quantizer followed by an optimally designed Huffman encoder. A bit-allocation algorithm is used to change the allocation of bits to the quantizers of different subbands dynamically. A variance-estimation procedure provides an estimate of the variances of the subbands to the bit-allocation algorithm. A mean-estimation algorithm is used to estimate the mean of the lowest frequency subband. The output of the encoder is a bit stream consisting of the output of the Huffman encoders and the overhead information for the variances of the subbands, the mean value of the lowest frequency subband, and the bit maps determined by the bit-allocation algorithm.

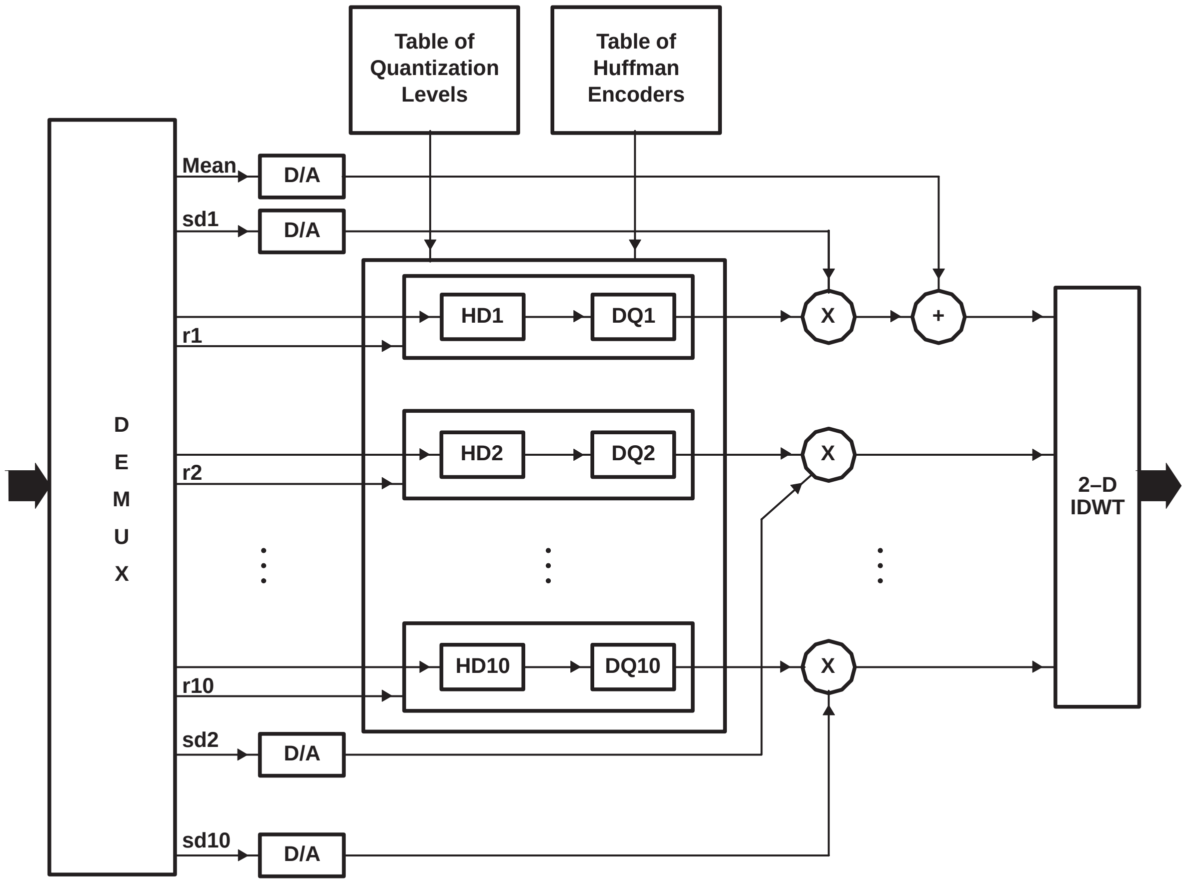


Figure 2. Block Diagram of the Decoder

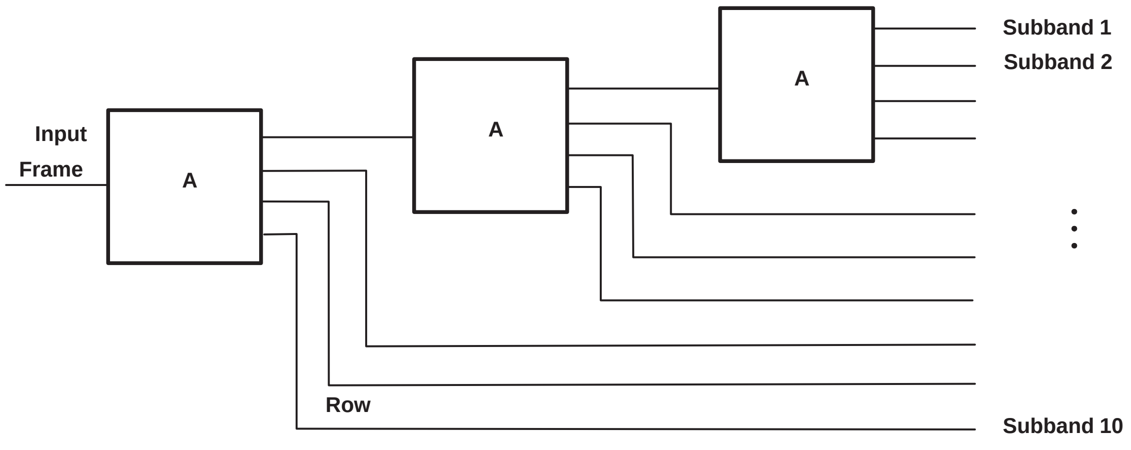
2.1.1 2-D Discrete-Wavelet Transform (2-D DWT)

Figure 3 shows the 2-D DWT block of the encoder. The 2-D DWT block consists of three levels of decomposition as illustrated in Figure 3(a). Clearly, the specific decomposition used here results in 10 subbands. Each level of decomposition, represented by the operation A in Figure 3(a), is described further in terms of simpler operations in Figure 3(b). Specifically, A consists of low-pass and high-pass filtering (H and G) in the row direction and subsampling by a factor of two, followed by the same procedure on each of the resulting outputs in the column direction, resulting in four subbands. The H and G filters (“Image Coding Using Wavelet Transform”[1]) are finite-impulse-response (FIR) digital filters. The specific input-output relationship for one level of DWT decomposition of a 1-D sequence X(n) can be represented as

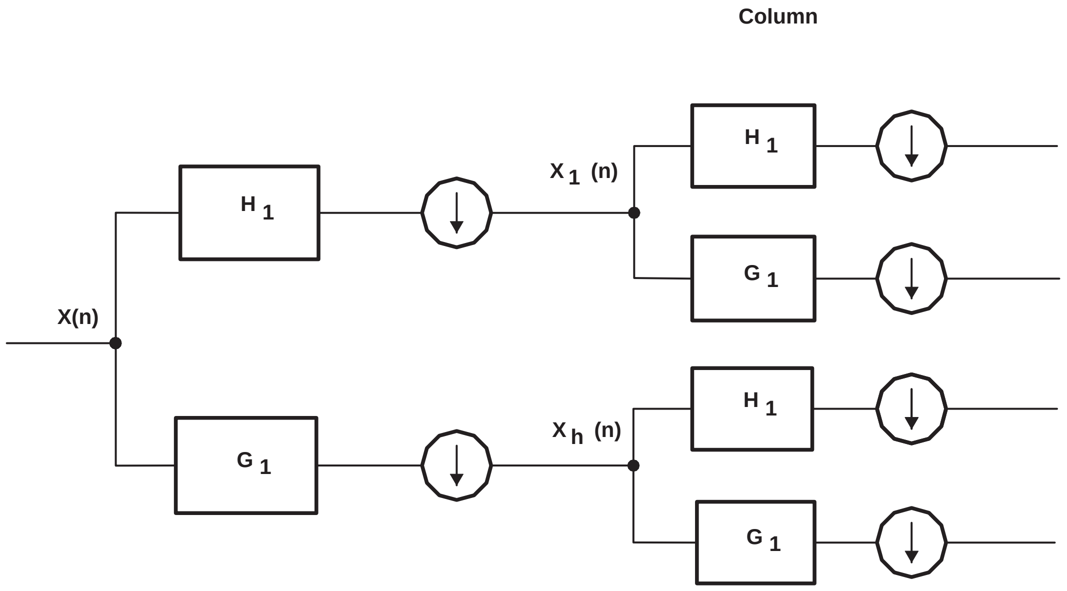
$$\begin{aligned} X_l(n) &= \sum_k h_1(2n-k)\ X(k) \\ X_h(n) &= \sum_k g_1(2n-k)\ X(k) \end{aligned} \tag{1}$$

in which $X_l(n)$ and $X_h(n)$ represent, respectively, the outputs of the low-pass and high-pass filters.

The resulting 2-D subbands after the 2-D DWT operation are labeled *subband1* through *subband10*. Due to the specific form of the decomposition, for a QCIF input sequence of size 176 $\times$ 144, subband1 through subband4 are of 22 $\times$ 18 pixel size, subband5 through subband7 are of 44 $\times$ 36 pixel size and subband8 through subband10 are of 88 $\times$ 72 pixel size.



(a)



(b)

Figure 3. Discrete-Wavelet Transform

2.1.2 Mean and Variance Estimation

The mean estimator estimates the mean of the lowest frequency subband. This is done by adding all the samples in this subband and dividing by the total number of samples (in this case, $22 \times 18 = 396$). The mean values of all other subbands are assumed to be zero (as supported by empirical evidence) and, therefore, are not estimated.

The variance estimators operate on each of the 10 subbands and estimate their variances once per frame. The variance estimators compute the sum of the squares of the samples of each subband (in the case of the lowest frequency subband after subtracting the mean) and divide by the number of samples in that band.

The estimated mean of the lowest frequency subband and the estimated standard deviation (square root of the variance) of each of the subbands are then represented by a finite number of bits (11 bits for the mean and 16 bits for each of the 10 standard deviations) using a simple uniform quantizer. The quantized mean and the quantized standard deviations are sent to the decoder as overhead information, once per frame.

2.1.3 Bit Allocation

The overall mean squared error per sample can be written as

$$D = \sum_{i=1} w_i \delta_i^2 \; d\left(r_i\right) \tag{2}$$

in which w_1 = 1/64, i –1,2,3,4, w_1 = 1/16, i = 5,6,7 and w_1 = 1/4, i = 8,9,10,δ_i^2 is the variance of subband i and d(ri) is the distortion-rate performance of the quantizer operating on the ith subband at the rate of r_i bits per sample. The encoding rate in this system is given by

$$R = \sum_{i=1} w_i r_i \tag{3}$$

For a given average number of bits per pixel, R, the objective is to determine the optimal r_i, i = 1,...,10, such that D is minimized.

In this system, the d(r) performance results were precomputed and stored in tables, for a finite number of values of *r*, starting at 0.25 bits per sample, up to 6.00 bits per sample in increments of 0.25 bits per sample. This is done based on an assumed distribution of the subband coefficients obtained from extensive simulations in the laboratory.

The bit-allocation algorithm used is based on the steepest descent integer programming procedure where, at each step of the process, 0.25 bits are allocated to the quantizer for the subband which results in the largest distortion reduction. Starting from an all zero bit allocation, this process is repeated until the total average bit rate is achieved. The details of this algorithm can be found in “Efficient Bit Allocation for an Arbitrary Set of Quantizers”[2].

The bit-allocation block, therefore, reads in the variances from the variance estimation block and, for the specified average bit rate R, generates the optimal bit rates for the quantizers to be used for encoding the subbands, r_1,..., r_{10}.

Notice that r_1,..., r_{10} rare integer multiples of 0.25 bits per sample and do not exceed 6.00 bits per sample. There are a total of 25 different possibilities for the bit rate of each quantizer. Therefore, it suffices to use 5 bits to specify the allocated rate for each subband. These bits, amounting to 50 bits per frame, are also transmitted as overhead to the decoder.

2.1.4 Normalization

Since all the quantizers are designed for operation on zero-mean and unit-variance inputs, before quantizing the subbands, they should be normalized to have zero mean and unit variance. This is done by first subtracting the quantized mean from the lowest frequency subband. Then, the mean-subtracted lowest frequency subband as well as all other subbands are divided by their corresponding estimated and quantized standard deviation.

Then each normalized subband is quantized by a quantizer with a bit rate specified by the bit-allocation algorithm.

2.1.5 Quantization

The quantizers used here are symmetric uniform-threshold quantizers as described in “Optimum Quantizer Performance for a Class of Non-Gaussian Memoryless Sources”[3]. The encoding intervals of these quantizers are of equal size and therefore can be characterized by just one stepsize parameter. The quantization levels are optimized for an assumed subband distribution.

Two distributions were used for the design of the quantizers: (a) a Gaussian distribution and (b) a generalized Gaussian distribution with parameter alpha = 0.6 (see “Subband Image Coding Using Entropy-Coded Quantization Over Noisy Channels”[4] for details). The Gaussian assumption (a) is most appropriate for encoding the lowest frequency subband while the generalized Gaussian distribution with parameter alpha = 0.6 and (b) is appropriate for all other subbands. These two sets of quantizers are designed once for each of the prescribed encoding rates of 0.25, 0.50,..., 6.00 bits per sample. For each rate r and each input distribution (a or b) (assuming zero mean and unit variance), the number of quantization levels Nr, the optimum stepsize Δr and the optimum quantization levels $Q_1, Q_2, ..., Q_{Nr}$ are computed and stored in memory. The specifics of the quantizer design is shown in “Optimum Quantizer Performance for a Class of Non-Gaussian Memoryless Sources”[3].

For each normalized subband, the actual quantization operation consists of determining the index of the interval to which the input sample belongs. Since the quantizer is a symmetric uniform-threshold quantizer, this process is very simple and the corresponding index can be determined by considering the value of the input sample divided by the quantizer stepsize. These indices are then represented by an appropriate code word using the Huffman encoding operation.

2.1.6 Huffman Encoding

As mentioned before, for each of the two different assumed distributions, 25 quantizers at different rates were designed. In addition, for each of these quantizers an optimal Huffman code is designed. For each subband, the Huffman encoder takes the index at the output of the quantizer and produces a variable-length binary code word to represent that index. The Huffman codes are designed to minimize the average code-word length for each of the designed quantizers.

The output of the Huffman encoders constitute the bulk of the encoded data—the information determining the quantized values of the subband data.

2.1.7 Multiplexing

The output of the Huffman encoders of the 10 subbands and the overhead information (quantized mean of the lowest frequency subbands, the quantized standard deviation of all subbands, and the bit-allocation maps) are multiplexed and transmitted to the decoder.

2.2 Decoder

At the decoder, the received bit stream (consisting of the encoded data and the overhead bits) is used to decode and dequantize the subbands. Then, the inverse DWT (IDWT) is used to reconstruct each video frame.

2.2.1 Demultiplexing

The demultiplexer separates the received bitstream into the overhead information and the Huffman encoder output bits.

2.2.2 Huffman Decoding

For each subband, using the specified number of bits used (sent to the decoder as overhead), the Huffman decoders use the appropriate Huffman table to decode each code word into its corresponding quantizer index.

2.2.3 Dequantization

For each subband, the quantizer index and the corresponding table of quantization levels are used to produce the dequantized version of the normalized subband samples.

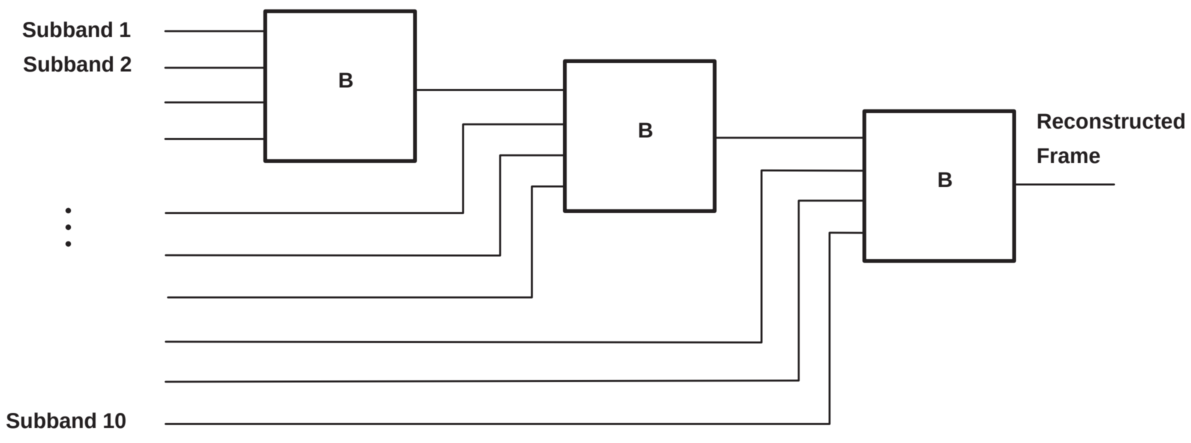
2.2.4 Denormalization

For each subband, the dequantized version of the normalized subband samples are multiplied by the corresponding quantized standard deviation (sent to the decoder as overhead) to produce the denormalized version of the samples. For the lowest frequency subband, the quantized mean is also added to the samples.

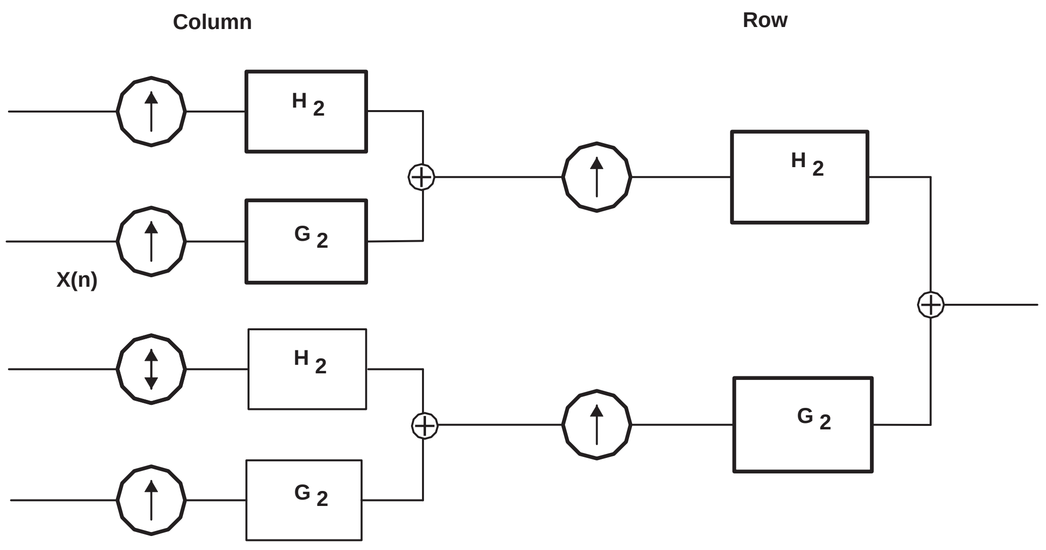
2.2.5 2-D Inverse DWT

To reconstruct a replica of the image frame, the dequantized and denormalized subbands are then fed into the 2-D IDWT block. Figure 4 shows the details of the 2-D IDWT operation. The 2-D IDWT block consists of three levels of reconstruction as illustrated in Figure 4(a). Each level of reconstruction, represented by the operation B in Figure 4, is described in terms of simpler operations in Figure 4(b). Specifically, B consists of up-sampling by a factor of two and low-pass and high-pass filtering in the column direction followed by the same procedure on the outputs of this process in the row direction, integrating four subbands into one wider band. The filters used for reconstruction (“Image Coding Using Wavelet Transform”[1]) are FIR digital filters. The specific input-output relationship for the reconstruction of the sequence X(n) is represented by

$$X(n) = \sum_{k} h_2(2k-n)\ X_l(k) + g_2(2n-k)\ X_h(k) \tag{4}$$



(a)



(b)

Figure 4. Inverse Discrete-Wavelet Transform

3 Implementation on the TMS320C40 DSP

This section describes the implementation on the TMS320C40 DSP, including video capture and display, use of the direct memory access (DMA), the FIR assembly routine, and performance data.

3.1 Video Capture and Display

The video capture and display routines depend on the system setup. Some cases require rewriting and modification of the following functions: init_dma, grab, unpack, pack, and send_frame. The given functions assume that the input video (8 bits per pixel monochrome) is supplied to commport 5 with four pixels in each 32-bit word. The video is line-by-line (from left to right) starting from the upper left corner and ending in the lower right. The pixels are packed into words with the first pixel in the least significant byte and the last pixel in the most significant byte. For example, in the first word of the video stream, bits 0–7 contain pixel 0; bits 8–15 contain pixel 1; bit 16–23 contain pixel 2; and bits 24–31 contain pixel 3. Video is displayed by sending it to commport 2 in the same format. The SIZE_X and SIZE_Y macros defined in options.h determine the dimensions of the input video; currently there is no way to change this at run time.

The init_dma routine sets up the auto-initialization tables for receiving and sending video. For receiving video, the DMA is read-synchronized on commport 5, and for sending video, it is write-synchronized on commport 2. The init_dma function also initializes the DMA channels control registers and link pointers so that the grab and send_frame functions only need to start their respective channels. If these channels are not used for any other transfers it is only necessary to call init_dma once; the control registers and link pointers remain properly set for grab and send_frame to begin the frame transfers. Otherwise, the DMA control registers and link pointers must be reinitialized each time before grab or send_frame are called. See auto-initialization method 2a in Chapter 9 of the *TMS320C4x User's Guide* (literature number SPRU063) for more details. To save memory, the destination for input video is the last quarter of the image array (image[SIZE_Y*3/4][0] to image[SIZE_Y-1][SIZE_X-1]). The unpack routine then unpacks each pixel from the 32 bit words and casts them to floating point so that each element in the image array is the floating-point value for a single pixel. Of course this routine overwrites the original input video; however, no word is over written before it is unpacked. The pack routine, used to generate the output video, works in a similar manner. The floating-point elements of the image array are cast to integers and their range is checked to ensure that they are between 0 to 255. These are then packed into the first quarter of the image array (image[0][0] to image[SIZE_Y/4-1][SIZE_X-1]). It is very important that the pixels of the input and output video are unpacked and packed in the correct order; otherwise, the pixels in each group of four might be in reverse order during the compression resulting in bad performance. This may require modifying unpack and pack to change the order. These routines are written in c-callable assembly language since they are time consuming. They only support the register passing model.

3.2 Discrete-Wavelet Transform

The DWT and IDWT are the most computationally intensive and time critical portions of the algorithm. The DWT uses 7-tap and 9-tap FIR filters. For a QCIF size image and three levels of decomposition, this requires approximately 532 000 multiplication operations and 467 000 addition operations for a total of 1 000 000 arithmetic operations. These numbers do not include any of the overhead calculations such as indexing the filters.

3.2.1 Use of the DMA

Since it is necessary to make a copy of each row or column before convoluting it and to reduce the time consumed by memory accesses, the DMA is used to asynchronously copy the next row/column of the image into one of the internal RAM blocks while the CPU is processing the current row/column in the other internal RAM block. The CPU has higher priority. The filter coefficients, stack, and variables are also stored in the internal RAM blocks. Because of the limited space in the internal RAM, the maximum row/column length was restricted to 256 pixels.

3.2.2 FIR Assembly Routine

The FIR filters are implemented in assembly using the parallel multiply add instruction (MPYF||ADDF) with circular addressing to cycle through the filter coefficients. The multiply add instruction effectively reduces the number of clock cycles needed to perform the 1,000,000 arithmetic operations by performing many of them in parallel; and the circular addressing reduces the overhead of cycling through the filters. The functions *convolutiond* and *convolutionr* perform the decomposition and reconstruction, respectively, on a single row or column. For maximum speed, these routines are specifically written for the filters designed by the authors so it is necessary to modify the routines for use with different filters.

3.3 Performance Data

The following benchmarks where collected using the single CPU version running on the PPDS. The clock speed is 32 MHz.

3.4 Multiple Processor Implementation on the PPDS

The implementation is based on the parallel processing development system, which is a standalone board consisting of four TMS320C40s operating at 32 MHz. Each ‘C40 has 246K bytes of local memory and there are 512K bytes of shared memory. All video compression and decompression is done on these four ‘C40s. For video capture and display, a third-party video card, containing a single ‘C40, plugged into the PC was used. Video captured from the camera is transmitted by one of this ‘C40 commports to CPU D on the PPDS. This ‘C40 also receives the compressed/decompressed video from CPU B by another commport for display.

3.4.1 Parallel Organization

To achieve 15 frames per second video it is necessary to compress and send each captured frame in less than 1/15 of a second, or about 67 milliseconds. For two-way video communication, decompression of received video also must be done during the same 67 ms. Because of this time constraint and the computationally intense nature of the wavelet transform, it is important to use the four ‘C40s in an efficient parallel manner. One obvious possibility would be to divide up the image between multiple processors for the DWT and IDWT. The disadvantage of this scheme is that it would require the use of slower shared memory or the significant overhead of data transfers across the commports. Instead, two ‘C40s (CPUs C and D) for compression and the other two for decompression were chosen. As shown in Figure 5, CPUs C and D each do the entire compression (DWT, quantization, and Huffman Coding) on alternate frames. CPU C works on odd video frames and CPU D works on even video frames. Similarly, CPUs A and B are each responsible for the decompression (Huffman decoding, dequantization, and IDWT) for alternate frames. In this scheme, each CPU has 2/15 = 133 ms to perform its task for each frame.

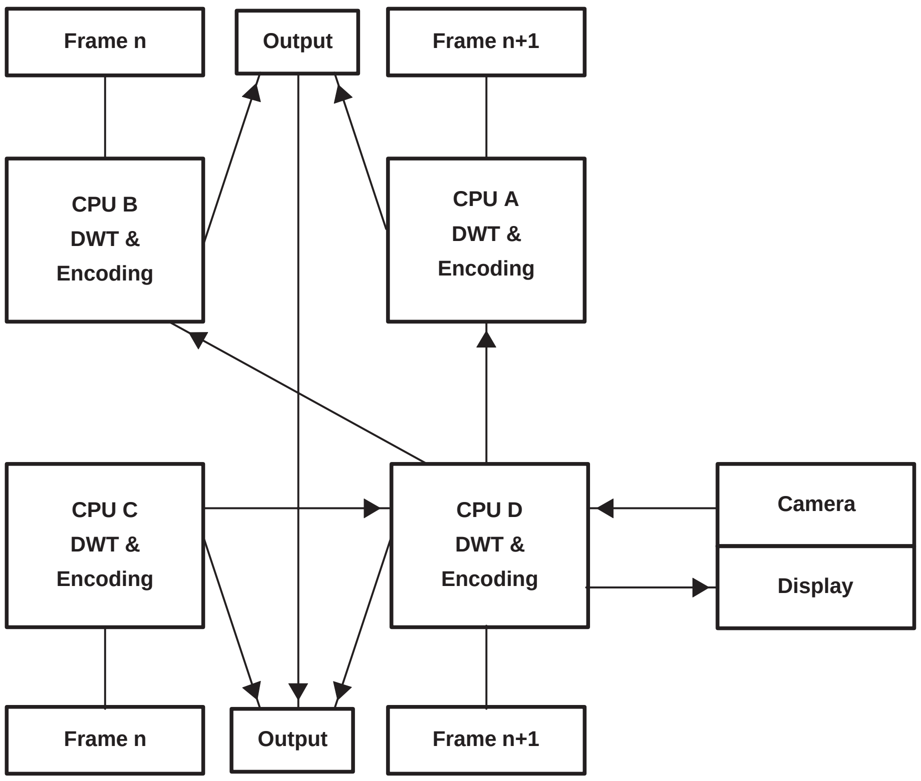


Figure 5. Hardware Diagram

The key advantage of this system is simplicity. The problem of performing both compression and decompression on four chips in under 67 ms has been reduced to two much simpler problems: a single ‘C40 must be able to compress a frame in under 133 ms; and a single ‘C40 must be able to decompress a frame in under 133 ms. Another advantage is that this does not require communication overhead or shared memory since each CPU operates independently; however, there are some disadvantages. First, it requires more memory since each processor needs to have space for its own video frame, program, and tables. Second, the delay from the time a frame is captured to the time the compressed/decompressed frame is displayed is given by the total number of CPUs, divided by the frame rate. In this case, that is 4/15 = 267 ms, which is very noticeable. Using only two chips at a higher clock rate would reduce the delay accordingly.

3.4.2 Data Path

As previously mentioned and illustrated in Figure 5, the input video is received from the video board by CPU D. Odd frames are passed through to CPU C by a commport-to-commport DMA transfer. CPU C stores the odd frames in its local memory. CPU D stores even frames to its local memory. In a two-way system connected by a transmission line, the compressed bit streams of CPUs C and D would be multiplexed together and transmitted. Compressed video from the other end would be demultiplexed and distributed between CPUs A and B. In this system, the bit stream from CPU C is sent to CPU A, and the stream from CPU D is sent to CPU B by DMA transfers over the commports. After decompression, the odd frames are sent from CPU A to CPU B. These odd frames are passed through to the video board by a commport-to-commport transfer. Even frames are also sent from CPU B to the video board.

3.4.3 Timing/Synchronization of Data Transfers

The synchronization of the data transfers between the CPUs on the PPDS and between the PPDS and video board is accomplished almost entirely by the DMA commport-synchronization mechanism. When any CPU is ready for or needs data it simply starts a read-synchronized DMA transfer from a commport. When a CPU is ready to transmit its data, the CPU starts a write-synchronized DMA transfer to a commport. If necessary, the CPU waits for these transfers to complete so that it does not attempt to use data that has not arrived yet or to overwrite data before it has been sent.

The exceptions are the demultiplexing of input video by CPU D and the multiplexing of output video by CPU B. These both require commport-to-commport transfers. Since each DMA can synchronize only with its associated commport, only one side of the commport-to-commport transfer can be synchronized. For example, to read data from commport 5 and write it to commport 0, you have to decide between read-synchronization on commport 5 or write-synchronization on commport 0. You cannot do both. CPU D is read-synchronized on commport 5 for receiving input video and not write-synchronized on the commport to CPU C. This means that when the video board is sending an odd frame to CPU D and CPU D is attempting to pass it through to CPU C, if CPU C is not ready the DMA on CPU D is halted until CPU C becomes ready. This is undesirable but does not appear to cause any major problems. If it is necessary to fix this, a similar scheme to the one explained in the following paragraphs for the multiplexing of the output video can be used.

When CPU B is ready to pass an odd frame from CPU A through to the video board it sets up a single word dummy transfer which is read-synchronized on a level-triggered external interrupt. When CPU A is ready to send a frame, it first sends the external interrupt to B by using the local control synchronization register (LCSR). CPU A then begins to send the frame. After receiving the interrupt, the dummy transfer on CPU B is completed. Then the DMA on B autoinitializes and begins the pass-through transfer from CPU A to the video board. This transfer is write-synchronized on the commport to the video board. When the transfer is complete, CPU A resets its LCSR to discontinue the ready interrupt to CPU B. This mechanism prevents CPU B from beginning the pass-through transfer before A is ready.

If it is possible, it is recommended that each CPU have direct access for receiving input video and displaying output video, instead of using the pass-through transfers described earlier. These pass-through transfers are prone to gridlock when rearranging code. Ideally, the processors should not be on a different board from the video capture and display equipment so that input video can be captured to shared memory and output video written to shared memory for display.

4 Summary

This project has resulted in a successful real-time implementation of a wavelet-based video compression/decompression system using the Texas Instruments parallel processing development board. The demonstration of the working project consists of a video camera connected to a video frame grabber whose output is fed into the TMS320C40 board. The entire encoding and decoding operation takes place in real time on the TMS320C40 board. The decoder output is displayed subsequently in real time on a monitor to assess the quality of the reconstructed video. This is the first successful real-time implementation of a video-coding system on a DSP board.

This work can be pursued in several directions: (i) further complexity reduction to allow for real-time implementation of higher resolution sequences such as CIF, (ii) inclusion of motion estimation and compensation schemes for improved rate-distortion performance, and (iii) improved quantization and encoding algorithms.

References

1. M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, “Image Coding Using Wavelet Transform”, *IEEE Trans. On Image Proc.*, Vol. IP-1, pp. 205 220, Apr. 1992.
2. Y. Shoham and A. Gersho, “Efficient Bit Allocation for an Arbitrary Set of Quantizers”, *IEEE Trans. Acoust. Speech and Signal Proc.*, Vol. ASSP-36, pp. 1445 1453, Sep. 1988.
3. N. Farvardin and J. W. Modestino, “Optimum Quantizer Performance for a Class of Non-Gaussian Memoryless Sources”, *IEEE Trans. Inform. Theory*, Vol. IT-30, pp. 485 497, May 1984.
4. N. Tanabe and N. Farvardin, “Subband Image Coding Using Entropy-Coded Quantization Over Noisy Channels”, *IEEE J. Select. Areas in Com.*, Vol. 10, pp. 926 943, June 1992.

Appendix A Code Listings

This appendix contains listings of all of the program, header, and linker command files necessary to run either a single CPU or four CPU version of the video coder on the PPDS. For practical purposes the Huffman tables are not included here.

FILE	DESCRIPTION	PAGE
`send.c`	4 CPU encoder main program	A-2
`receive.c`	4 CPU decoder main program	A-6
`single.c`	1 CPU encoder and decoder main program	A-10
`decomp.c`	DWT function	A-15
`recomp.c`	IDWT function	A-18
`cnvld.asm`	DWT Convolution function	A-23
`cnvlr.asm`	IDWT Convolution function	A-26
`bitalloc.c`	Bit allocation function	A-28
`utsqen.c`	Quantizer functions	A-29
`utsqde.c`	Dequantizer functions	A-36
`bstrm.c`	Huffman bit stream functions	A-40
`grab.c`	Video capture functions	A-45
`unpackf.asm`	Video unpacking function	A-46
`disp.c`	Video display functions	A-47
`packf.asm`	Video packing function	A-49
`adma.c`	DMA autoinitialization function for CPU A	A-51
`bdma.c`	DMA autoinitialization function for CPU B	A-52
`cdma.c`	DMA autoinitialization function for CPU C	A-54
`ddma.c`	DMA autoinitialization function for CPU D	A-55
`sdma.c`	1 CPU DMA autoinitialization function	A-56
`options.h`	Header defining image size, target rate, etc.	A-58
`waveletc.h`	Header for wavelet module	A-58
`utsq.h`	utsq.h	A-59
`mean.h`	Header for quantizer/dequantizer module	A-60
`bstrm.h`	bstrm.h	A-61
`grab.h`	Header for video capture and display module	A-62
`ports.h`	Comport and LCSR macros	A-62
`sects.h`	USECT macro	A-63
`average.h`	Running average macros	A-64
`meml.cmd`	Linker command file	A-64

FILE	DESCRIPTION	PAGE
`sendc.cmd`	Linker command file for CPU C	A-64
`sendd.cmd`	Linker command file for CPU D	A-65
`recea.cmd`	Linker command file for CPU A	A-66
`receb.cmd`	Linker command file for CPU B	A-67
`single.cmd`	1 CPU linker command file	A-68
`comptbls.cmd`	Batch file to compile tables	A-69
`comp4.cmd`	Batch file to compile 4 CPU version	A-69
`comp1.cmd`	Batch file to compile 1 CPU version	A-69

A.1 send.c

```
/*  Copyright (c) 1996, University of Maryland at College Park. */
/*                  All Rights Reserved.                        */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: send.c                                         */
/* by: Jerome Johnson, Hamid Jafarkhani                 */
/* description: multiple CPU video encoder main program */
/* cl30 –v40 –g –as –dCPUC –mn –mr –mf –o2 send.c       */
/* cl30 –v40 –g –as –dCPUD –mn –mr –mf –o2 send.c       */
#ifdef _TMS320C40
#include <compt40.h>
#include <intpt40.h>
#include <dma40.h>
#include ”sects.h”
#endif
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include ”ports.h”
#include ”options.h”
#include ”mean.h”
#include ”bstrm.h”
#include ”waveletc.h”
#include ”utsq.h”
#include ”grab.h”
#ifdef _TMS320C40
```

```
#ifdef CPUC
#include ”cdma.c”
#define D_WAIT_FOR_CDMA()
#define D_START_CDMA()
#else
#include ”ddma.c”
#define D_WAIT_FOR_CDMA() \
  while( !chk_iif_flag( GRABPORT+25 ) );\
  reset_iif_flag( GRABPORT+25 );\
  padin()
#define D_START_CDMA() padin(); DMA_RESTART( GRABPORT )
#endif /* #ifdef CPUC */
#else  /* _TMS320C40 not defined */
#define D_WAIT_FOR_CDMA()
#define D_START_CDMA()
#define GRABPORT 0
#endif /* #ifdef _TMS320C40 */
extern float HA[];         /* access wavelet filters */
float *H = HA;
extern codeword_t codebook_base_addr[];  /* access codebooks */
codeword_t *cbook[MAX_R_D];
#ifdef _TMS320C40
USECT( IMAGE, ”IMAGE$”, SIZE_X * SIZE_Y );  /* create a section
for image */
#endif
extern float IMAGE[SIZE_Y][SIZE_X];    /* declare image defined by
USECT */
float (*image)[SIZE_X] = IMAGE;
#ifdef _TMS320C40
USECT( ostream_base_addr, ”STREAM”, MAXSTREAM ); /* create sect
for ostream */
#endif
extern unsigned int ostream_base_addr[ MAXSTREAM ];
bit_stream_t ostream;
volatile float target_rate = TARGET_RATE;
int main(void) {
  unsigned long *inframe;
  volatile float deconst, wait, send, encode, grab, grab0,
unpack_time;
  volatile float start_time;
  /************* variables for encode and send *****************/
  int size_x,size_y;
  float mean_LFS;
  float var[NO_SUB], sd[NO_SUB];
  float rate_sub[MAX_R_D]={0.,0.579456,0.697600,0.853440,1.336192,

1.479616,1.635776,1.848640,2.034048,2.278656,2.530688,2.788544,

3.071744,3.320128,3.511040,3.743104,3.965547,4.244139,4.465762,
    4.728298,5.049574,5.242066,5.464252,5.727122,6.048888};
  float dist_sub[MAX_R_D]={1.,0.520418,0.348135,0.243303,0.171546,

0.121190,0.088637,0.061269,0.046040,0.032251,0.022458,0.016133,

0.011466,0.008245,0.006134,0.004321,0.003582,0.002392,0.001795,
    0.001275,0.000847,0.000607,0.000451,0.000322,0.000212};
  /************** Changed *********/
  short N[MAX_R_D] =
{0,49,81,169,17,21,25,31,35,43,53,63,77,91,105,127,35,
    43,49,59,73,85,99,117,147};  /* sizes of huff codebooks */
  /************** Changed *********/
  short N1[4] = {0,7,9,13};    /* number of quantization levels
for HC2 */
  float delta[MAX_R_D] =
{0.,5.000000,3.375000,2.500000,1.937500,1.531250,\

1.250000,1.000000,0.843750,0.687500,0.562500,0.468750,0.390625,0.3
28125,\

0.281250,0.234375,0.265625,0.218750,0.187500,0.156250,0.125000,0.1
09375,\
    0.093750,0.078125,0.062500};
  /************** Changed *********/
  float rate_weight[NO_SUB] =
{0.015625,0.015625,0.015625,0.015625,0.0625,\
    0.0625,0.0625};    /* ,0.25,0.25,0.25}; */
  float rate;
  int ii, jj;
  unsigned short alloc_rates[NO_SUB];
  unsigned short sd_quan[NO_SUB];
  float quan_mean_LFS;
  unsigned short mean_index;
  /**************** end variables for encode and send ***********/
```

```
#ifdef _TMS320C40
  INT_DISABLE(); /* disable all cpu interupts GIE = 0 */
  load_die( 0 );
  load_iie( 0 );
  load_iif( 0 );
  asm( "     OR 1800h, st" ); /* cache enable */
#endif
  /* initialization of encode and send */
  bs_init( &ostream, ostream_base_addr );
  hc2_init_cb( cbook, N );
  init_dma();
  /* inframe = last 1/4 of image space for reading in packed frame
*/
  inframe = (unsigned long *)image + (3*SIZE_X*SIZE_Y>>2);
  size_x = SIZE_X >> DECOMPOSITION_LEVEL;
  size_y = SIZE_Y >> DECOMPOSITION_LEVEL;
  while(1) {
    time_start(0);    start_time = time_read(0);
    D_WAIT_FOR_CDMA();
    grab_frame( GRABPORT, inframe, 1 );
    while( !chk_iif_flag( GRABPORT+25 ) ); /* wait for grab to
complete */
    reset_iif_flag( GRABPORT+25 );
    D_START_CDMA();
    grab = time_read(0);
    unpackf( inframe, image, (SIZE_X*SIZE_Y>>2) );
    unpack_time = time_read(0) – grab;
    /* decompose image */
    wavelet_dec( image, SIZE_X, SIZE_Y, DECOMPOSITION_LEVEL, 2 );
    deconst = time_read(0) – grab;
    /* encode  */
    /************** Changed *********/
    var_7( image, size_x, size_y, var, &mean_LFS );
    bit_alloc( rate_sub, dist_sub, rate_weight, var, MAX_R_D,
target_rate,
          &rate, NO_SUB, alloc_rates );
    var_to_sd( var, sd, sd_quan );
    quan_mean( mean_LFS, &quan_mean_LFS, &mean_index );
    /*********** Change *************/
    for( ii = 0; ii < NO_SUB; ii++ ) {
      if( sd_quan[ii] == 0 )
   alloc_rates[ii] = 0;
    }
    /********** end change *********/
    bs_reset_stream( &ostream );
    write_header( &ostream, sd_quan, mean_index, alloc_rates );
    write_out_stream( &ostream, image, size_x, size_y,
alloc_rates,
            N, N1, delta, quan_mean_LFS, sd );
    encode = time_read(0) – deconst – grab;
    /* send header and stream */
    bs_send_stream( ostream );
    send = time_stop(0) – encode – deconst – grab;
  }
  return 0;
}
#ifdef _TMS320C40
/* temporary ”fix” for a cable problem */
void padin( void ) {
  int x = 1;
  while( x != 0 ) {
    while( !cp_in_level( GRABPORT ) );
    x = in_word( GRABPORT );
  }
  while( x == 0 ) {
    while( !cp_in_level( GRABPORT ) );
    x = in_word( GRABPORT );
  }
}
#endif
```

A.2 receive.c

```
/*      Copyright (c) 1996, University of Maryland at College
Park.   */
/*                           All Rights Reserved.
   */
/*              (developed by  Jerome Johnson,  Hamid Jafarkhani
   */
/*                 Ruplu Bhattacharya and Nariman Farvardin)
   */
```

```
/* file: receive.c                                          */
/* by: Jerome Johnson, Hamid Jafarkhani                     */
/* description: multiple CPU video decoder main program */
/* cl30 –v40 –g –as –dCPUA –mn –mr –mf –o2 send.c       */
/* cl30 –v40 –g –as –dCPUB –mn –mr –mf –o2 send.c       */
#include <stdlib.h>
#include <math.h>
#ifdef _TMS320C40
#include <compt40.h>
#include <intpt40.h>
#include <dma40.h>
#include ”sects.h”
#endif
#include ”ports.h”
#include ”options.h”
#include ”mean.h”
#include ”bstrm.h”
#include ”waveletc.h”
#include ”utsq.h”
#include ”grab.h”
#ifdef _TMS320C40
#ifdef CPUA
#include ”adma.c”
#define B_WAIT_FOR_ADMA()
#else   /* CPUA not defined */
#include ”bdma.c”
#define B_WAIT_FOR_ADMA()\
  if( (DMA_ADDR( DISPPORT )–>_gctrl._bitval.start == 3) )\
    while( !chk_iif_flag( 25+DISPPORT ) );\
  reset_iif_flag( 25+DISPPORT );
#endif  /* #ifdef CPUA */
#else   /* _TMS320C40 not defined */
#define B_WAIT_FOR_ADMA()
#define DISPPORT 0
#endif  /* #ifdef _TMS320C40 */
extern float HA[];      /* access filters */
float *H = HA;
extern mynode_t hufftree_base_addr[];  /* access huff trees
*/
```

```
mynode_t *huff_tree[ MAX_R_D ];
extern int huffsize_base_addr[];      /* access sizes of trees
*/
int *huffsize = huffsize_base_addr;
extern float centroid_base_addr[];    /* access centroid tables
*/
float *centroid[ MAX_R_D ];
#ifdef _TMS320C40
USECT( IMAGE, ”IMAGE$”, SIZE_X * SIZE_Y );
#endif
extern float IMAGE[SIZE_Y][SIZE_X];   /* declare image defined by
USECT  */
float (*image)[SIZE_X] = IMAGE;
#ifdef _TMS320C40
USECT( istream_base_addr, ”STREAM”, MAXSTREAM ); /* create sect
for ostream */
#endif
extern unsigned int istream_base_addr[ MAXSTREAM ];
bit_stream_t istream;
int main(void) {
  unsigned long *outframe;
  volatile float reconst, receive, decode, sendframe;
  /*********** variables for receive and decode ****************/
  int i,j;
  int size_x,size_y;
  float sd[NO_SUB];
  short N[MAX_R_D] =
{0,49,81,169,17,21,25,31,35,43,53,63,77,91,105,127,35,
    43,49,59,73,85,99,117,147};  /* sizes of huff codebooks */
  short N1[4] = { 0, 7, 9, 13 };
  unsigned short alloc_rates[NO_SUB];
  unsigned short sd_quan[NO_SUB];
  float quan_mean_LFS;
  unsigned short mean_index;
  /********* end variables for receive and decode ***************/
#ifdef _TMS320C40
  INT_DISABLE(); /* disable all cpu interupts GIE = 0 */
  load_die( 0 );
  load_iie( 0 );
  load_iif( 0 );
  asm( ”     OR 1800h, st” ); /* cache enable */
#endif
  /* initialization for decode */
  bs_init( &istream, istream_base_addr );
  init_hufftree( huff_tree, huffsize );
  init_centroid( centroid, N, N1 );
  size_x = SIZE_X >> DECOMPOSITION_LEVEL;
  size_y = SIZE_Y >> DECOMPOSITION_LEVEL;
  init_dma();
  /* pack frames into 1st 1/4 image space */
  outframe = (unsigned long *)image;
  while(1) {
    time_start(1);
    /* receive header and stream */
    bs_reset_stream( &istream );
    bs_receive_stream( istream );
    for( i = 0; i < SIZE_Y; i++ ) /* reset image to 0’s */
      for( j = 0; j < SIZE_X; j++ )
   image[i][j] = 0.0;
    receive = time_read(1);
    /* decode stream */
    read_header( &istream, sd_quan, &mean_index, alloc_rates,
       sd, &quan_mean_LFS );
    read_in_stream( &istream, image, size_x, size_y, alloc_rates,
          quan_mean_LFS, sd, N1 );
    decode = time_read(1) – receive;
    /* reconstruct image */
    wavelet_rec( image, SIZE_X, SIZE_Y, DECOMPOSITION_LEVEL, 5 );
    reconst = time_read(1) – decode – receive;
    /* write out image */
    packf( image, outframe, SIZE_X * SIZE_Y );
    B_WAIT_FOR_ADMA();
    LCSR_SET_INT( 2 );
    send_frame( DISPPORT, outframe, 1 );
    while( !chk_iif_flag( 25+DISPPORT ) ); /* wait for send frame
to finish */
    reset_iif_flag( 25+DISPPORT );
    LCSR_RESET_INT( 2 );
    sendframe = time_stop(1) – decode – receive – reconst;
  }
  return 0;
}
```

A.3 single.c

```
/*      Copyright (c) 1996, University of Maryland at College
Park.   */
/*                     All Rights Reserved.                      */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*          Ruplu Bhattacharya and Nariman Farvardin)           */
/* file: single.c                                                */
/* by: Jerome Johnson, Hamid Jafarkhani
*/
/* description: single CPU video encoder/decoder main program   */
/* cl30 –v40 –g –as –mn –mr –mf –o2 single.c                    */
#ifdef _TMS320C40
#include <compt40.h>
#include <intpt40.h>
#include <dma40.h>
#include ”sects.h”
#endif
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include ”ports.h”
#include ”options.h”
#include ”mean.h”
#include ”bstrm.h”
#include ”waveletc.h”
#include ”utsq.h”
#include ”grab.h”
#ifdef _TMS320C40
#include ”sdma.c”
#else
#define GRABPORT 0
#define DISPPORT 0
#endif
void padin( void );
```

```
#ifdef TEST_HC
unsigned long *test1base = (unsigned long *)(0x80000000);
unsigned long *test2base = (unsigned long *)(0x80010000);
unsigned long *test1;
unsigned long *test2;
#endif
extern float HA[];         /* access filters */
float *H = HA;
/* send.c */
extern codeword_t codebook_base_addr[];  /* access codebooks */
codeword_t *cbook[MAX_R_D];
#ifdef _TMS320C40
USECT( IMAGE, "IMAGE$", SIZE_X * SIZE_Y );  /* create a section
for image */
#endif
extern float IMAGE[SIZE_Y][SIZE_X];    /* declare image defined by
USECT */
float (*image)[SIZE_X] = IMAGE;
#ifdef _TMS320C40
USECT( stream_base_addr, "STREAM", MAXSTREAM ); /* create sect for
ostream */
#endif
extern unsigned int stream_base_addr[ MAXSTREAM ];
bit_stream_t ostrm;
bit_stream_t istrm;
/* end send.c */
/* receive.c */
extern mynode_t hufftree_base_addr[];  /* access huff trees
*/
mynode_t *huff_tree[ MAX_R_D ];
extern int huffsize_base_addr[];       /* access sizes of trees
*/
int *huffsize = huffsize_base_addr;
extern float centroid_base_addr[];     /* access centroid tables
*/
float *centroid[ MAX_R_D ];
/* end receive.c */
int main(void) {
  unsigned long *inframe, *outframe;
  /*************** variables for encode and send ****************/
  int size_x,size_y;
  float mean_LFS;
  float var[NO_SUB], sd[NO_SUB];
  float rate_sub[MAX_R_D]={0.,0.579456,0.697600,0.853440,1.336192,

1.479616,1.635776,1.848640,2.034048,2.278656,2.530688,2.788544,

3.071744,3.320128,3.511040,3.743104,3.965547,4.244139,4.465762,
    4.728298,5.049574,5.242066,5.464252,5.727122,6.048888};
  float dist_sub[MAX_R_D]={1.,0.520418,0.348135,0.243303,0.171546,

0.121190,0.088637,0.061269,0.046040,0.032251,0.022458,0.016133,

0.011466,0.008245,0.006134,0.004321,0.003582,0.002392,0.001795,
    0.001275,0.000847,0.000607,0.000451,0.000322,0.000212};
  /************** Changed *********/
  short N[MAX_R_D] =
{0,49,81,169,17,21,25,31,35,43,53,63,77,91,105,127,35,
    43,49,59,73,85,99,117,147};  /* sizes of huff codebooks */
  /************** Changed *********/
  short N1[4] = {0,7,9,13};    /* number of quantization levels
for HC2 */
  float delta[MAX_R_D] =
{0.0,5.000000,3.375000,2.500000,1.937500,1.531250,\

1.250000,1.000000,0.843750,0.687500,0.562500,0.468750,0.390625,0.3
28125,\

0.281250,0.234375,0.265625,0.218750,0.187500,0.156250,0.125000,0.1
09375,\
    0.093750,0.078125,0.062500};
  /************** Changed *********/
  float rate_weight[NO_SUB] =
{0.015625,0.015625,0.015625,0.015625,0.0625,\
    0.0625,0.0625};    /* ,0.25,0.25,0.25}; */
  float rate;
  volatile float target_rate = TARGET_RATE;
  int ii, jj, i, j;
  unsigned short alloc_rates[NO_SUB];
  unsigned short sd_quan[NO_SUB];
  float quan_mean_LFS;
  unsigned short mean_index;
  /********** end variables for encode and send ****************/
```

```
#ifdef _TMS320C40
  INT_DISABLE(); /* disable all cpu interupts GIE = 0 */
  load_die( 0 );
  load_iie( 0 );
  load_iif( 0 );
  asm( ”     OR 1800h, st” ); /* cache enable */
#endif
  /* initialization of encode and send */
  bs_init( &ostrm, stream_base_addr );
  hc2_init_cb( cbook, N );
  init_dma();
  /* inframe = last 1/4 of image space for reading in packed frame
*/
  inframe = (unsigned long *)image + (3*SIZE_X*SIZE_Y>>2);
  /* initialization for decode */
  bs_init( &istrm, stream_base_addr );
  init_hufftree( huff_tree, huffsize );
  init_centroid( centroid, N, N1 );
  /* pack frames into first 1/4 image space */
  outframe = (unsigned long *)image;
  size_x = SIZE_X >> DECOMPOSITION_LEVEL;
  size_y = SIZE_Y >> DECOMPOSITION_LEVEL;
/*
  for( i = 0; i < 16; i++ )
    H[ 16+i ] *= 2;
  for( i = 0; i < 16; i++ )
    H[ 48+i ] *= 2;
*/
  while(1) {
    /* grab frame */
    grab_frame( GRABPORT, inframe, 1 );
    while( !chk_iif_flag( GRABPORT+25 ) ); /* wait for grab to
complete */
    reset_iif_flag( GRABPORT+25 );
    padin();
    unpackf( inframe, image, (SIZE_X*SIZE_Y>>2) );
    /* decompose image */
    wavelet_dec( image, SIZE_X, SIZE_Y, DECOMPOSITION_LEVEL, 0 );
    /* encode  */
    /************** Changed *********/
    var_7( image, size_x, size_y, var, &mean_LFS );
    bit_alloc( rate_sub, dist_sub, rate_weight, var, MAX_R_D,
target_rate,
         &rate, NO_SUB, alloc_rates );
    var_to_sd( var, sd, sd_quan );
    quan_mean( mean_LFS, &quan_mean_LFS, &mean_index );
    /*********** Change *************/
    for( ii = 0; ii < NO_SUB; ii++ ) {
      if( sd_quan[ii] == 0 )
   alloc_rates[ii] = 0;
    }
    /********** end change *********/
    bs_reset_stream( &ostrm );
    write_header( &ostrm, sd_quan, mean_index, alloc_rates );
    write_out_stream( &ostrm, image, size_x, size_y, alloc_rates,
N, N1,
           delta, quan_mean_LFS, sd );
    /* send stream */
    /* bs_send_stream( ostream ); */
    /* receive stream */
    /* receive_stream(); */
    /* decode stream */
    bs_reset_stream( &istrm );
    for( i = 0; i < SIZE_Y; i++ ) /* reset image to 0's */
      for( j = 0; j < SIZE_X; j++ )
   image[i][j] = 0.0;
    read_header( &istrm, sd_quan, &mean_index, alloc_rates,
       sd, &quan_mean_LFS );
    read_in_stream( &istrm, image, size_x, size_y, alloc_rates,
          quan_mean_LFS, sd, N1 );
    /* reconstruct image */
    wavelet_rec( image, SIZE_X, SIZE_Y, DECOMPOSITION_LEVEL, 0 );
    /* write out image */
    packf( image, outframe, SIZE_X * SIZE_Y );
    send_frame( DISPPORT, outframe, 1 );
    while( !chk_iif_flag( 25+DISPPORT ) ); /* wait for send frame
to finish */
    reset_iif_flag( 25+DISPPORT );
```

```
  }
  return 0;
}
#ifdef _TMS320C40
void padin( void ) {
  int x = 1;
  while( x != 0 ) {
    while( !cp_in_level( GRABPORT ) );
    x = in_word( GRABPORT );
  }
  while( x == 0 ) {
    while( !cp_in_level( GRABPORT ) );
    x = in_word( GRABPORT );
  }
}
#endif
```

A.4 decomp.c

```
/*  Copyright (c) 1996, University of Maryland at College Park. */
/*                    All Rights Reserved.                      */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: decomp.c                                       */
/* by: Hamid Jafarkhani, Jerome Johnson                 */
/* description: wavelet decomposition function          */
/* cl30 –v40 –g –as –mn –mr –mf –o2 decomp.c            */
#include ”waveletc.h”
#include <stdlib.h>
#include <dma40.h>
#include ”sects.h”
USECT( XA, ”XA$”, MAXRC );
USECT( XB, ”XB$”, MAXRC );
extern float XA[MAXRC], XB[MAXRC];
float *X0 = XA;
float *X1 = XB;
float *Z0, *Z1, *Ztemp;
void wavelet_dec( float (*image)[SIZE_X], int SIZE_x, int SIZE_y,\
                  int decomposition_level, int ch_no ) {
```

```
  extern float *H;
  DMA_REG *dmax = DMA_ADDR( ch_no );
  DMA_REG dma_table[2][3];
  int SIZE_x_temp, SIZE_y_temp;
  int j, i;
  int n, index, f, g;
  int next_frame_x, next_frame_y;
  SIZE_x_temp = SIZE_x;
  SIZE_y_temp = SIZE_y;
  /* set up dma autoinit sequence tables */
  dma_table[0][0]._gctrl._intval   = 0x80c00009;
  dma_table[0][0].dma_regs.dst     = &X0[MARGE];
  dma_table[0][0].dma_regs.dst_idx = 1;
  dma_table[0][0].dma_link         = (unsigned long
*)&dma_table[0][1];
  dma_table[1][0]._gctrl._intval   = 0x80c00009;
  dma_table[1][0].dma_regs.dst     = &X1[MARGE];
  dma_table[1][0].dma_regs.dst_idx = 1;
  dma_table[1][0].dma_link         = (unsigned long
*)&dma_table[1][1];
  /* boundary reflection */
  dma_table[0][1]._gctrl._intval   = 0x80c00009;
  dma_table[0][1].dma_regs.src     = &X0[9];
  dma_table[0][1].dma_regs.src_idx = –1;
  dma_table[0][1].dma_regs.count   = 4;
  dma_table[0][1].dma_regs.dst     = &X0[1];
  dma_table[0][1].dma_regs.dst_idx = 1;
  dma_table[0][1].dma_link         = (unsigned long
*)&dma_table[0][2];
  dma_table[1][1]._gctrl._intval   = 0x80c00009;
  dma_table[1][1].dma_regs.src     = &X1[9];
  dma_table[1][1].dma_regs.src_idx = –1;
  dma_table[1][1].dma_regs.count   = 4;
  dma_table[1][1].dma_regs.dst     = &X1[1];
  dma_table[1][1].dma_regs.dst_idx = 1;
  dma_table[1][1].dma_link         = (unsigned long
*)&dma_table[1][2];
  dma_table[0][2]._gctrl._intval   = 0x80c0000d;
  dma_table[0][2].dma_regs.src_idx = –1;
  dma_table[0][2].dma_regs.count   = 4;
  dma_table[0][2].dma_regs.dst_idx = 1;
  dma_table[0][2].dma_link         = (unsigned long
*)&dma_table[1][0];
  dma_table[1][2]._gctrl._intval   = 0x80c0000d;
  dma_table[1][2].dma_regs.src_idx = –1;
  dma_table[1][2].dma_regs.count   = 4;
  dma_table[1][2].dma_regs.dst_idx = 1;
  dma_table[1][2].dma_link         = (unsigned long
*)&dma_table[0][0];
  /* set up DMA channel for autoinit sequence */
  dmax–>_gctrl._intval   = 0x8000000d;
  dmax–>dma_regs.count   = 0;
  dmax–>dma_link         = (unsigned long *)&dma_table[0];
  for( i = 0; i < decomposition_level; i++ ) {
    next_frame_x = SIZE_x_temp >> 1;
    next_frame_y = SIZE_y_temp >> 1;
    dma_table[0][0].dma_regs.src_idx = 1;
    dma_table[0][0].dma_regs.count   = SIZE_x_temp;
    dma_table[0][2].dma_regs.src = &X0[SIZE_x_temp+3];
    dma_table[0][2].dma_regs.dst = &X0[SIZE_x_temp+5];
    dma_table[1][0].dma_regs.src_idx = 1;
    dma_table[1][0].dma_regs.count   = SIZE_x_temp;
    dma_table[1][2].dma_regs.src = &X1[SIZE_x_temp+3];
    dma_table[1][2].dma_regs.dst = &X1[SIZE_x_temp+5];
    dma_table[0][0].dma_regs.src = &image[0][0];      /* load row
0 into X0 */
    DMA_RESTART( ch_no );
    Z0 = X0; Z1 = X1;
    for( f = 1; f < SIZE_y_temp; f++ ) {
      while( chk_dma( ch_no ) );
      dma_table[f&1][0].dma_regs.src = &image[f][0];
      DMA_RESTART( ch_no );
      convolutiond( Z0, image[f–1], H, next_frame_x, 1 );
      Ztemp = Z0;
      Z0 = Z1;
      Z1 = Ztemp;
    }
    dma_table[0][0].dma_regs.src_idx = SIZE_x;
    dma_table[0][0].dma_regs.count   = SIZE_y_temp;
```

```
        dma_table[0][2].dma_regs.src = &X0[SIZE_y_temp+3];
        dma_table[0][2].dma_regs.dst = &X0[SIZE_y_temp+5];
        dma_table[1][0].dma_regs.src_idx = SIZE_x;
        dma_table[1][0].dma_regs.count   = SIZE_y_temp;
        dma_table[1][2].dma_regs.src = &X1[SIZE_y_temp+3];
        dma_table[1][2].dma_regs.dst = &X1[SIZE_y_temp+5];
        while( chk_dma( ch_no ) );
        dma_table[0][0].dma_regs.src = &image[0][0];  /* load col 0
    int X0 */
        DMA_RESTART( ch_no );
        /* convolution on last row */
        convolutiond( Z0, image[f–1], H, next_frame_x, 1 );
        Z0 = X0; Z1 = X1;
        for( g = 1; g < SIZE_x_temp; g++ ) {
          while( chk_dma( ch_no ) );
          dma_table[g&1][0].dma_regs.src = &image[0][g];
          DMA_RESTART( ch_no );
          convolutiond( Z0, (image[0]+g–1), H, next_frame_y, SIZE_X );
          Ztemp = Z0;
          Z0 = Z1;
          Z1 = Ztemp;
        }
        while( chk_dma( ch_no ) );
        /* convolution on last col */
        convolutiond( Z0, (image[0]+g–1), H, next_frame_y, SIZE_X );
        SIZE_x_temp = next_frame_x;
        SIZE_y_temp = next_frame_y;
      }
      return;
    }
```

A.5 recomp.c

```
    /*  Copyright (c) 1996, University of Maryland at College Park. */
    /*                   All Rights Reserved.                        */
    /*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
    /*           Ruplu Bhattacharya and Nariman Farvardin)           */
    /* file: recomp.c                                        */
    /* by: Hamid Jafarkhani, Jerome Johnson                  */
    /* description: wavelet reconstruction function         */
```

```
/* cl30 –v40 –g –as –mn –mr –mf –o2 recomp.c            */
#include ”waveletc.h”
#include <stdlib.h>
#include <dma40.h>
#include ”sects.h”
/* the following are also done in decomp.c so it is necessary
   to compile recomp.c with the –dSINGLE option to prevent
   multiple definitions when using a single CPU configuration */
#ifndef SINGLE
USECT( XA, ”XA$”, MAXRC );
USECT( XB, ”XB$”, MAXRC );
extern float XA[MAXRC], XB[MAXRC];
float *X0 = XA;
float *X1 = XB;
#else
extern float XA[MAXRC], XB[MAXRC];
extern float *X0;  /* = XA; */
extern float *X1;  /* = XB; */
#endif
void wavelet_rec( float (*image)[SIZE_X], int SIZE_x, int SIZE_y,\
         int decomposition_level, int ch_no ) {
  extern float *H;
  float *Z0, *Z1, *Ztemp;
  DMA_REG *dmax = DMA_ADDR( ch_no );
  DMA_REG dma_table[2][3];
  int SIZE_x_temp, SIZE_y_temp;
  int p, s, c;
  int i, j, n;
  int dec_index, m;
  int SIZE_xx, SIZE_yy;
  /* set up dma autoinit sequence tables */
  /* low–pass source */
  dma_table[0][0]._gctrl._intval   = 0x80c00009;
  dma_table[0][0].dma_regs.dst     = &X0[MARGE];
  dma_table[0][0].dma_regs.dst_idx = 1;
  dma_table[0][0].dma_link         = (unsigned long
*)&dma_table[0][1];
  dma_table[1][0]._gctrl._intval   = 0x80c00009;
  dma_table[1][0].dma_regs.dst     = &X1[MARGE];
  dma_table[1][0].dma_regs.dst_idx = 1;
  dma_table[1][0].dma_link         = (unsigned long
*)&dma_table[1][1];
  /* high–pass source */
  dma_table[0][1]._gctrl._intval   = 0x80c0000d;
  dma_table[0][1].dma_regs.dst     = &X0[MARGE + MAXRC/2];
  dma_table[0][1].dma_regs.dst_idx = 1;
  dma_table[0][1].dma_link         = (unsigned long
*)&dma_table[1][0];
  dma_table[1][1]._gctrl._intval   = 0x80c0000d;
  dma_table[1][1].dma_regs.dst     = &X1[MARGE + MAXRC/2];
  dma_table[1][1].dma_regs.dst_idx = 1;
  dma_table[1][1].dma_link         = (unsigned long
*)&dma_table[0][0];
  /* set up DMA channel for autoinit sequence */
  dmax–>_gctrl._intval   = 0x8000000d;
  dmax–>dma_regs.count   = 0;
  dmax–>dma_link         = (unsigned long *)&dma_table[0];
  for( dec_index = decomposition_level; dec_index >= 1;
dec_index–– ) {
    SIZE_xx = SIZE_x >> dec_index;
    SIZE_yy = SIZE_y >> dec_index;
    SIZE_x_temp = (SIZE_xx << 1);
    SIZE_y_temp = (SIZE_yy << 1);
    dma_table[0][0].dma_regs.src_idx = SIZE_x;
    dma_table[0][0].dma_regs.count   = SIZE_yy;
    dma_table[0][1].dma_regs.src_idx = SIZE_x;
    dma_table[0][1].dma_regs.count   = SIZE_yy;
    dma_table[1][0].dma_regs.src_idx = SIZE_x;
    dma_table[1][0].dma_regs.count   = SIZE_yy;
    dma_table[1][1].dma_regs.src_idx = SIZE_x;
    dma_table[1][1].dma_regs.count   = SIZE_yy;
    dma_table[0][0].dma_regs.src = &image[0][0];      /* load col
0 into X0 */
    dma_table[0][1].dma_regs.src = &image[SIZE_yy][0];
    DMA_RESTART( ch_no );
    Z0 = X0; Z1 = X1;
    for(j = 1; j < SIZE_x_temp; j++ ) {
      dma_table[j&1][0].dma_regs.src = &image[0][j];
      dma_table[j&1][1].dma_regs.src = &image[SIZE_yy][j];
      while( chk_dma( ch_no ) );
      DMA_RESTART( ch_no );
      /* boundary reflection */
      Z0[4] = Z0[6];
      Z0[3] = Z0[7];
      Z0[2] = Z0[8];
      Z0[1] = Z0[9];
      Z0[SIZE_yy+5] = Z0[SIZE_yy+4];
      Z0[SIZE_yy+6] = Z0[SIZE_yy+3];
      Z0[SIZE_yy+7] = Z0[SIZE_yy+2];
      Z0[SIZE_yy+8] = Z0[SIZE_yy+1];
      Z0[4 + MAXRC/2] = Z0[5 + MAXRC/2];
      Z0[3 + MAXRC/2] = Z0[6 + MAXRC/2];
      Z0[2 + MAXRC/2] = Z0[7 + MAXRC/2];
      Z0[1 + MAXRC/2] = Z0[8 + MAXRC/2];
      Z0[SIZE_yy+5 + MAXRC/2] = Z0[SIZE_yy+3 + MAXRC/2];
      Z0[SIZE_yy+6 + MAXRC/2] = Z0[SIZE_yy+2 + MAXRC/2];
      Z0[SIZE_yy+7 + MAXRC/2] = Z0[SIZE_yy+1 + MAXRC/2];
      Z0[SIZE_yy+8 + MAXRC/2] = Z0[SIZE_yy   + MAXRC/2];
      convolutionr( Z0, (image[0] + j–1), H, SIZE_yy, SIZE_x );
      Ztemp = Z0;
      Z0 = Z1;
      Z1 = Ztemp;
    }

    dma_table[0][0].dma_regs.src_idx = 1;
    dma_table[0][0].dma_regs.count   = SIZE_xx;
    dma_table[0][1].dma_regs.src_idx = 1;
    dma_table[0][1].dma_regs.count   = SIZE_xx;
    dma_table[1][0].dma_regs.src_idx = 1;
    dma_table[1][0].dma_regs.count   = SIZE_xx;
    dma_table[1][1].dma_regs.src_idx = 1;
    dma_table[1][1].dma_regs.count   = SIZE_xx;
    dma_table[0][0].dma_regs.src = &image[0][0];      /* load row
0 into X0 */
    dma_table[0][1].dma_regs.src = &image[0][SIZE_xx];
    DMA_RESTART( ch_no );
```

```
      /* convolution on last col */
      convolutionr( Z0, (image[0] + j–1), H, SIZE_yy, SIZE_x );
      Z0 = X0; Z1 = X1;
      for( j = 1; j < SIZE_y_temp; j++ ) {
        dma_table[j&1][0].dma_regs.src = &image[j][0];
        dma_table[j&1][1].dma_regs.src = &image[j][SIZE_xx];
        while( chk_dma( ch_no ) );
        DMA_RESTART( ch_no );
        /* boundary reflection */
        Z0[4] = Z0[6];
        Z0[3] = Z0[7];
        Z0[2] = Z0[8];
        Z0[1] = Z0[9];
        Z0[SIZE_xx+5] = Z0[SIZE_xx+4];
        Z0[SIZE_xx+6] = Z0[SIZE_xx+3];
        Z0[SIZE_xx+7] = Z0[SIZE_xx+2];
        Z0[SIZE_xx+8] = Z0[SIZE_xx+1];
        Z0[4 + MAXRC/2] = Z0[5 + MAXRC/2];
        Z0[3 + MAXRC/2] = Z0[6 + MAXRC/2];
        Z0[2 + MAXRC/2] = Z0[7 + MAXRC/2];
        Z0[1 + MAXRC/2] = Z0[8 + MAXRC/2];
        Z0[SIZE_xx+5 + MAXRC/2] = Z0[SIZE_xx+3 + MAXRC/2];
        Z0[SIZE_xx+6 + MAXRC/2] = Z0[SIZE_xx+2 + MAXRC/2];
        Z0[SIZE_xx+7 + MAXRC/2] = Z0[SIZE_xx+1 + MAXRC/2];
        Z0[SIZE_xx+8 + MAXRC/2] = Z0[SIZE_xx   + MAXRC/2];
        convolutionr( Z0, image[j–1], H, SIZE_xx, 1 );
        Ztemp = Z0;
        Z0 = Z1;
        Z1 = Ztemp;
      }
      /* convolution on last row */
      convolutionr( Z0, image[j–1], H, SIZE_xx, 1 );
    }
  }
```

A.6 cvnld.asm

```
* file: cnvld.asm
* by: Jerome Johnson
* description: performs wavelet decomposition on a single row
*              or column of the image
* cl30 –v40 –g –as –mn –mr –mf cnvld.asm
FP      .set AR3
HMAX    .set 16
HLEN0   .set 9
        .global _convolutiond
        .align                  ; align on cache boundary
_convolutiond                   ; convolutiond( X, Y, &HA,
size/2, step )
        PUSH FP
        LDI SP, FP
        PUSH AR4
        PUSH AR5
; AR2 is source row/col         ; R2 is destination row/col
; R3 is address of filter       ; RC is size/2
; RS is step size of dest
        LDA R2, AR4             ; load 1st dest row/col address
into AR4
        MPYI3 RS, RC, R1
        ADDI3 R2, R1, AR5       ; load 2nd dest row/col address
into AR5
        SUBI  2, RC             ; RC =  size/2 – 2 for size/2 – 1
iterations
        LDA R3, AR0             ; load filter addresses
        ADDI3 R3, 2*HMAX, AR1
        LDI RS, IR1             ; load step size of destination
into IR1
        LDA HLEN0, BK           ; load block size for circ
addressing
; AR2 is source row or col      ; AR4, AR5 are dest row or col
; R0, R1 are products           ; R2, R3 are accumulators
; AR0, AR1 are addresses of filters 0, 2 respectively
; RC is sizex/2 – 2
        NOP *AR2++(HLEN0)         ; use ARAU to add 9 to AR2
        MPYF *AR0++%, *AR2––, R1   ; AR2 must start at X[9]
        MPYF *AR0++%, *AR2––, R3   ; H[0][1] X[8]
        MPYF *AR0++%, *AR2––, R1   ; H[0][2] X[7]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][3] X[6]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][4] X[5]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][5] X[4]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][6] X[3]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][7] X[2]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R1   ; H[0][8] X[1]
||      ADDF R1, R3
        ADDF R1, R3                ; accumulate last product
        STF R3, *AR4++(IR1)        ; store result
        RPTB EL1
SL1     NOP *AR2++(2+HLEN0)        ; add n + 9 to source index

        MPYF *AR0++%, *AR2, R1     ; H[0][0] X[n+9]
        MPYF *AR1++%, *AR2––, R0   ; H[2]
        MPYF *AR0++%, *AR2, R3     ; H[0][1] X[n+8]
        MPYF *AR1++%, *AR2––, R2   ; H[2]
        MPYF *AR0++%, *AR2, R1     ; H[0][2] X[n+7]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][3] X[n+6]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][4] X[n+5]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][5] X[n+4]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
```

```
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][6] X[n+3]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][7] X[n+2]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[0][8] X[n+1]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R0   ; H[2]
||      ADDF R0, R2
        ADDF R1, R3                ; accumulate last products
        ADDF R0, R2
        STF  R3, *AR4++(IR1)       ; store results
EL1     STF  R2, *AR5++(IR1)
        NOP *AR2++(2+HLEN0)        ; increment AR2
        MPYF *AR1++%, *AR2––, R1   ; AR2 must start at X[sizex+9]
        MPYF *AR1++%, *AR2––, R3   ; H[2][1] X[sizex+8]
        MPYF *AR1++%, *AR2––, R1   ; H[2][2] X[sizex+7]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][3] X[sizex+6]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][4] X[sizex+5]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][5] X[sizex+4]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][6] X[sizex+3]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][7] X[sizex+2]
||      ADDF R1, R3
        MPYF *AR1++%, *AR2––, R1   ; H[2][8] X[sizex+1]
||      ADDF R1, R3
        ADDF R1, R3                ; accumulate last product
        STF R3, *AR5               ; store result
        LDI *–FP(1), R1            ; load return address
        POP AR5
        BD R1                      ; delayed branch
        POP AR4
        POP FP
        SUBI 01H, SP               ; POP return address
;
```

A.7 cnvlr.asm

```
* file: cnvlr.asm
* by: Jerome Johnson
* description: performs wavelet reconstruction on a single row
*              or column of the image
* cl30 –v40 –g –as –mn –mr –mf cnvlr.asm
FP      .set AR3
HMAX    .set 16
HLEN    .set 9
MARGE2  .set 10
SRC2    .set 276/2
        .global _convolutionr
        .align                   ; align on cache boundary
_convolutionr                    ; convolutionr( X, Y, &HA,
size/2, index )
        PUSH FP
        LDI SP, FP
        PUSH AR4
; AR2 is source row/col          ; R2 is destination row/col
; R3 is address of filter        ; RC is size/2
; RS is index of dest
        LDA R2, AR4              ; load dest row/col address into
AR4
        LDI SRC2, IR0            ; load offset of 2nd source
        SUBI  1, RC              ; RC =  size/2 – 1 for size/2
iterations
        ADDI3 R3, 1*HMAX, AR0    ; load filter addresses
        ADDI3 R3, 3*HMAX, AR1
        LDI RS, IR1              ; load step size of destination
into IR1
        LDA HLEN, BK             ; load block size for circ
addressing
; AR2 is source row or col       ; AR4 is dest row or col
; R0, R1 are products            ; R2, R3 are accumulators
; AR0, AR1 are addresses of filters 1, 3 respectively
; RC is size/2 – 1               ; IR0 is offset of 2nd source
; IR1 is index of dest
        RPTBD EL1
        NOP *AR2++(2)             ; initialize source address
        NOP
        NOP
SL1     NOP *AR2++(5)             ; increment source address
        MPYF *AR1++%, *+AR2(IR0), R0  ; H[3][0]
        MPYF *AR0++%, *AR2––, R2      ; H[1][0]
        MPYF *AR1++%, *+AR2(IR0), R1  ; H[3][1]
        MPYF *AR0++%, *AR2, R3        ; H[1][1]
        MPYF *AR1++%, *+AR2(IR0), R0  ; H[3][2]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2––, R0      ; H[1][2]
||      ADDF R0, R2
*****************************************************
        MPYF *AR1++%, *+AR2(IR0), R1  ; H[3][3]
||      ADDF R1, R3
        MPYF *AR1++%, *+AR2(IR0), R0  ; H[3][4]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[1][3]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R0   ; H[1][4]
||      ADDF R0, R2
        MPYF *AR1++%, *+AR2(IR0), R1  ; H[3][5]
||      ADDF R1, R3
        MPYF *AR1++%, *+AR2(IR0), R0  ; H[3][6]
||      ADDF R0, R2
        MPYF *AR0++%, *AR2, R1     ; H[1][5]
||      ADDF R1, R3
        MPYF *AR0++%, *AR2––, R0   ; H[1][6]
||      ADDF R0, R2
        MPYF *AR1++%, *+AR2(IR0), R1   ; H[3][7]
||      ADDF R1, R3
        MPYF *AR1++%, *+AR2(IR0), R0   ; H[3][8]
||      ADDF R0, R2
        NOP *AR0++(2)%            ; advance filter 1
        ADDF R1, R3                 ; accumulate last products
        ADDF R0, R2
        ; MPYF 2, R3
        ; MPYF 2, R2
        STF R3, *AR4++(IR1)         ; store results
EL1     STF R2, *AR4++(IR1)
        LDI *–FP(1), R1             ; load return address
        BD R1                       ; delayed branch
        POP AR4
        POP FP
        SUBI 01H, SP                ; POP return address
;
```

A.8 bitalloc.c

```
/*  Copyright (c) 1996, University of Maryland at College Park. */
/*                  All Rights Reserved.                        */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: bitalloc.c                                     */
/* by: Hamid Jafarkhani                                 */
/* description: bit allocation function                 */
/* cl30 –v40 –g –as –dCPUD –mn –mr –mf bitalloc.c       */
#include <math.h>
#include <stdlib.h>
#include ”utsq.h”
#include ”mean.h”
void bit_alloc( float *rate_sub, float *dist_sub, float
*rate_weight,
                float *var,int max_R_D, float target_rate, float
*rate,
                int no_sub,unsigned short *alloc_rates ) {
  float slope[NO_SUB];
  unsigned short No_R_D[NO_SUB];
  int i,j;

  (*rate)=0.;
  for(i=0;i<no_sub;i++){
    (*rate)+=rate_sub[0]*rate_weight[i];
    slope[i]=var[i]*(dist_sub[0]–dist_sub[1]);
```

```
      slope[i]/=((rate_sub[1]–rate_sub[0])*rate_weight[i]);
      No_R_D[i]=1;
    }
    while((*rate)<target_rate){
      i=max_slope(slope,no_sub);

(*rate)+=(rate_sub[No_R_D[i]]–rate_sub[No_R_D[i]–1])*rate_weight[i
];
      if(No_R_D[i]+1<max_R_D){
        slope[i]=var[i]*(dist_sub[No_R_D[i]]–dist_sub[No_R_D[i]+1]);

slope[i]/=((rate_sub[No_R_D[i]+1]–rate_sub[No_R_D[i]])*rate_weight
[i]);
      }
      else{
        slope[i]=0.;
        for(j=0;j<no_sub–1 && slope[j]<Epsilon;j++);;
        if(j==no_sub–1 && slope[j]<Epsilon){
     target_rate=0.;
        }
      }
      No_R_D[i]++;
    }
    for(i=0;i<no_sub;i++){
      alloc_rates[i]=No_R_D[i]–1;
    }
  }
```

A.9 utsqen.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: utsqen.c                                           */
/* by:  Hamid Jafarkhani, Jerome Johnson                    */
/* description: uniform threshold scalar quantizer          */
/* cl30 –v40 –g –as –mn –mr –mf –o2 utsqen.c                */
#include <math.h>
#include <stdlib.h>
#include <string.h>
```

```
#ifdef _TMS320C40
#include <compt40.h>
#endif
#include ”ports.h”
#include ”utsq.h”
#include ”bstrm.h”
#include ”mean.h”
#include ”options.h”
#define SQR(x) ((x)*(x))
/*****************************************************************/
void hc2_init_cb( codeword_t *cbook[], short *N ) {
  int i, j, offset = 0;
  extern codeword_t codebook_base_addr[];  /* access codebooks */
  extern subfield_t codeword_base_addr[];  /* access codebooks */
  cbook[0] = NULL;
  for( i = 1; i < MAX_R_D; i++ ) {
    cbook[i] = codebook_base_addr + offset;
    offset += N[i];
  }
  offset = 0;
  for( i = 1; i < MAX_R_D; i++ ) {
    for( j = 0; j < N[i]; j++ ) {
      cbook[i][j].bits = codeword_base_addr + offset;
      offset += ( (cbook[i][j].len+31)>>5 );
    }
  }
  return;
}
/*****************************************************************/
void var_7( float image[][SIZE_X], int size_x, int size_y, float
*var,\
        float *mean_LFS ) {
  register int ii,jj;
  double  mean,variance;
  int xx, yy;
  int i,j,n,sub_index;
  int size_x2, size_y2;
  mean = 0.0; variance = 0.0;
  for( ii = 0; ii < size_y; ii++ ) {
    for(jj=0;jj<size_x;jj++){
      mean+=image[ii][jj];
      variance+=SQR(image[ii][jj]);
    }
  }
  mean /= (double)(size_y*size_x);
  *mean_LFS = mean;
  var[0] = variance/(size_y*size_x)–SQR(mean);
  sub_index = 0;
  /***** changed *************/
  /* for( n=1; n<=4; n*=2 ) */
  for( n=1; n<3; n++ ){
    size_x2=size_x*n;
    size_y2=size_y*n;
    for(i=0;i<2;i++){
      for(j=0;j<2;j++){
   if(0!=i+j){
     xx=size_y2*j;
     yy=size_x2*i;
     variance=0.0;
     for(ii=0;ii<size_y2;ii++){
       for(jj=0;jj<size_x2;jj++){
         variance+=SQR(image[xx+ii][yy+jj]);
       }
     }
     var[++sub_index]=variance/(size_y2*size_x2);
   }
      }
    }
  }
}
/*****************************************************************/
int max_slope( float *slope, int no_sub ) {
  register int i;
  float max;
  int max_slope;
  max=slope[0];
  max_slope=0;
  for(i=1;i<no_sub;i++){
    if(slope[i]>max){
      max=slope[i];
      max_slope=i;
    }
  }
  return(max_slope);
}
/*****************************************************************/
void var_to_sd( float *var, float *sd, unsigned short *sd_quan ) {
  int i;
  for(i=0;i<NO_SUB;i++){
    sd_quan[i]=(unsigned short)(sqrt((double)var[i])/SD_NORM+0.5);
    sd[i]=sd_quan[i]*SD_NORM;
  }
}
/*****************************************************************/
void quan_mean( float mean_LFS, float *quan_mean_LFS,\
           unsigned short *mean_index) {
  (*mean_index)=(unsigned short)(mean_LFS/MEAN_NORM+0.5);
  (*quan_mean_LFS)=(*mean_index) * MEAN_NORM;
}
/*****************************************************************/
void write_out_stream( bit_stream_t *ostrm, float image[][SIZE_X],
             int size_x, int size_y,
             unsigned short *alloc_rates, short *N, short *N1,
             float *delta, float quan_mean_LFS, float *sd ) {
  extern codeword_t * cbook[MAX_R_D];  /* array of codebooks */
  int i, j, n, xx, yy;
  float half_bound, delta_reciprocal, sd_recip;
  register int ii, jj;
  int size_x2, size_y2;
  int sub_index;
  int data_int, data_int1;
#ifdef TEST_HC
  extern unsigned long *test1, *test1base;
  test1 = test1base;
```

```
#endif
  if( alloc_rates[0] != 0 ) {
    if( alloc_rates[0] <= 3 ) alloc_rates[0] = 4;
    half_bound = (.5) * ( delta[alloc_rates[0]] *
N[alloc_rates[0]] );
    delta_reciprocal = 1 / delta[alloc_rates[0]];
    sd_recip = 1 / sd[0];
    for( ii = 0; ii < size_y; ii++ ) {
      for( jj = 0; jj < size_x; jj++ ) {
   data_int = (int)( ((image[ii][jj]–quan_mean_LFS) * sd_recip\
            + half_bound) * delta_reciprocal );
   if( data_int >= N[alloc_rates[0]] )
     data_int = N[alloc_rates[0]] – 1;
   else if( data_int < 0 )
     data_int = 0;
   bs_write_subfield( ostrm, cbook[alloc_rates[0]][data_int] );
#ifdef TEST_HC
   *(test1++) = data_int;
#endif
      }
    }
  }
  sub_index = 1;
  /************** Changed *********/
  /*  for(n=1;n<=4;n*=2){*/
  for( n = 1; n < 3; n++ ) {
    size_x2 = size_x * n;
    size_y2 = size_y * n;
    for( i = 0; i < 2; i++ ) {
      for( j = 0; j < 2; j++ ) {
   if(0!=i+j && alloc_rates[sub_index]!=0){
     xx = size_y2 * j;
     yy = size_x2 * i;
     delta_reciprocal = 1 / delta[alloc_rates[sub_index]];
     sd_recip = 1 / sd[sub_index];
     /* 1ST ORDER HUFFMAN CODE */
     if( alloc_rates[sub_index] > 3 ) {
       half_bound = (.5) * ( delta[alloc_rates[sub_index]] * \
              N[alloc_rates[sub_index]] );
        for( ii = 0; ii < size_y2 ; ii++ ) {
          for( jj = 0; jj < size_x2; jj++ ) {
        data_int = (int)(( (image[xx+ii][yy+jj] * sd_recip) \
                 + half_bound ) * delta_reciprocal);
        if( data_int >= N[alloc_rates[sub_index]] )
          data_int = N[alloc_rates[sub_index]] – 1;
        else if(data_int < 0)
          data_int = 0;
        bs_write_subfield( ostrm,
                 cbook[alloc_rates[sub_index]][data_int] );
#ifdef TEST_HC
        *(test1++) = data_int;
#endif
          }   /* for jj */
        }   /* for ii */
      }   /* if( alloc_rates[sub_index] > 3 ) */
      /* 2ND ORDER HUFFMAN CODE */
      else  { /* alloc_rates[sub_index] <= 3 */
        half_bound = (.5) * ( delta[alloc_rates[sub_index]] * \
               N1[alloc_rates[sub_index]] );
        for( ii = 0; ii < size_y2 ; ii++ ) {
          for( jj = 0; jj < size_x2; jj+=2 ) {
        data_int = (int)(( (image[xx+ii][yy+jj] * sd_recip) \
                 + half_bound ) * delta_reciprocal);
        if( data_int >= N1[alloc_rates[sub_index]] )
          data_int = N1[alloc_rates[sub_index]] – 1;
        else if(data_int < 0)
          data_int = 0;
        data_int1 = (int)(( (image[xx+ii][yy+jj+1] * sd_recip) \
                  + half_bound ) * delta_reciprocal);
        if( data_int1 >= N1[alloc_rates[sub_index]] )
          data_int1 = N1[alloc_rates[sub_index]] – 1;
        else if(data_int1 < 0)
          data_int1 = 0;
        data_int = (data_int * N1[alloc_rates[sub_index]]
               + data_int1);
        bs_write_bitfield( ostrm,
                cbook[alloc_rates[sub_index]][data_int] );
#ifdef TEST_HC
       *(test1++) = data_int;
#endif
         }   /* for jj */
       }   /* for ii */
     }   /* else */
   } /* if(0!=i+j && alloc_rates[sub_index]!=0) */
   if( i+j != 0 )
     sub_index++;
      }   /* for j */
    }   /* for i */
  }   /* for( n = 1; n < 3; n++ ) */
  bs_flush_bitbuff( ostrm );
  return;
}
/****************************************************************/
void write_header( bit_stream_t *ostrm, unsigned short *sd_quan,
         unsigned short mean_index,
         unsigned short *alloc_rates ) {
  unsigned long temp;
  int i;
  for( i = 0; i < NO_SUB–1; i+=2 ) {
    bs_write_word( ostrm, (sd_quan[i] << 16) + sd_quan[i+1] );
  }
  temp = alloc_rates[0];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[1];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[2];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[3];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[4];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[5];
  bs_write_word( ostrm, temp );
  temp = sd_quan[NO_SUB–1];
  temp = (temp << ALLOC_RATE_SIZE) + alloc_rates[6];
  temp = (temp << MEAN_INDEX_SIZE) + mean_index;
  bs_write_word( ostrm, temp );
  return;
}
```

A.10 utsqde.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                    All Rights Reserved.                       */
/*         (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*            Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: utsqde.c                                        */
/* by:  Hamid Jafarkhani, Jerome Johnson                 */
/* description: uniform threshold scalar dequantizer    */
/* cl30 –v40 –g –as –mn –mr –mf –o2 utsqde.c            */
#include <math.h>
#include <stdlib.h>
#include <string.h>
#ifdef _TMS320C40
#include <compt40.h>
#endif
#include ”ports.h”
#include ”utsq.h”
#include ”bstrm.h”
#include ”mean.h”
#include ”options.h”
/*****************************************************************/
void init_centroid( float *centroid[], short *N, short *N1 ) {
  int i, offset = 0;
  extern float centroid_base_addr[];     /* access centroid tables
*/
  centroid[0] = 0;
  for( i = 1; i < 4; i++ ) {
    centroid[i] = centroid_base_addr + offset;
    offset += N1[i];
  }
  for( i = 4; i < MAX_R_D; i++ ) {
    centroid[i] = centroid_base_addr + offset;
    offset += N[i];
  }
  return;
}
/*****************************************************************/
void init_hufftree( mynode_t *huff_tree[], int *huffsize ) {
  int i, offset = 0;
  static int built = 0;
  /* extern int huffsize_base_addr[];       access sizes of trees
*/
  extern mynode_t hufftree_base_addr[];  /* access huff trees
*/
  huff_tree[0] = hufftree_base_addr;
  for( i = 1; i < MAX_R_D; i++ ) {
    huff_tree[i] = hufftree_base_addr + offset;
    offset += huffsize[i];
    if( !built )
      build_hufftree( huff_tree[i], huff_tree[i] );
  }
  built = 1;
  return;
}
/*****************************************************************/
void build_hufftree( mynode_t *stree, mynode_t *root ) {
  if( !stree ) return;
  if( (int)(stree–>left) > 0 )
    stree–>left = root + (int)(stree–>left);
  else
    stree–>left = NULL;
  if( (int)(stree–>right) > 0 )
    stree–>right = root + (int)(stree–>right);
  else
    stree–>right = NULL;
  build_hufftree( stree–>left, root );
  build_hufftree( stree–>right, root );
  return;
}
/*****************************************************************/
int huff_receive_code( bit_stream_t *istrm, mynode_t *root ) {
  while ( is_nonleaf(root) )
    root = ( bs_read_bit( istrm ) ? root–>right : root–>left );
  return (root–>code);
}
/*****************************************************************/
void read_header( bit_stream_t *istrm, unsigned short *sd_quan,
       unsigned short *mean_index,
       unsigned short *alloc_rates,
       float *sd, float *quan_mean_LFS ) {
  unsigned long temp;
  int i;
  for( i = 0; i < NO_SUB–1; i+=2 ) {
    temp = bs_read_word( istrm );
    sd_quan[i]   = temp >> 16;
    sd_quan[i+1] = temp & 0xffff;
  }
  temp = bs_read_word( istrm );
  alloc_rates[5] = (temp >> (ALLOC_RATE_SIZE*0)) & 31;
  alloc_rates[4] = (temp >> (ALLOC_RATE_SIZE*1)) & 31;
  alloc_rates[3] = (temp >> (ALLOC_RATE_SIZE*2)) & 31;
  alloc_rates[2] = (temp >> (ALLOC_RATE_SIZE*3)) & 31;
  alloc_rates[1] = (temp >> (ALLOC_RATE_SIZE*4)) & 31;
  alloc_rates[0] = (temp >> (ALLOC_RATE_SIZE*5)) & 31;
  temp = bs_read_word( istrm );
  *mean_index = temp & 2047;
  temp = temp >> MEAN_INDEX_SIZE;
  alloc_rates[6] = temp & 31;
  temp = temp >> ALLOC_RATE_SIZE;
  sd_quan[NO_SUB–1] = temp &0xffff;
  /* decode header */
  for( i = 0; i < NO_SUB; i++ )
    sd[i] = sd_quan[i] * SD_NORM;
  *quan_mean_LFS = *mean_index * MEAN_NORM;
  return;
}
/*****************************************************************/
void read_in_stream( bit_stream_t *istrm, float image[][SIZE_X],
          int size_x, int size_y,
          unsigned short *alloc_rates,
          float quan_mean_LFS, float *sd, short N1[] ) {
  extern mynode_t *huff_tree[ MAX_R_D ];
  extern float *centroid[ MAX_R_D ];
  int i, j, n, xx, yy;
  register int ii,jj;
  int size_x2, size_y2;
  int sub_index;
  int data_int;
#ifdef TEST_HC
  extern unsigned long *test2, *test2base;
  test2 = test2base;
#endif
  if( alloc_rates[0] != 0 ) {
    for( ii = 0; ii < size_y; ii++ ) {
      for( jj = 0; jj < size_x; jj++ ) {
   data_int = huff_receive_code( istrm, huff_tree[ alloc_rates[0]
] );
#ifdef TEST_HC
   *(test2++) = data_int;
#endif
   image[ii][jj] = sd[0] * centroid[ alloc_rates[0] ][ data_int ]\
     + quan_mean_LFS;
      }
    }
  }
  sub_index = 1;
  /************** Changed *********/
  /*  for(n=1;n<=4;n*=2){*/
  for( n = 1; n < 3; n++ ){
    size_x2 = size_x * n;
    size_y2 = size_y * n;
    for( i = 0; i < 2; i++ ) {
      for( j = 0; j < 2; j++ ) {
   if(0!=i+j && alloc_rates[sub_index] != 0 ) {
     xx = size_y2 * j;
     yy = size_x2 * i;
     if( alloc_rates[sub_index] > 3 ) {
       for( ii = 0; ii < size_y2; ii++ ) {
         for( jj = 0; jj < size_x2; jj++ ) {
      data_int = huff_receive_code( istrm,
        huff_tree[ alloc_rates[sub_index] ] );
#ifdef TEST_HC
      *(test2++) = data_int;
#endif
      image[xx+ii][yy+jj] = sd[sub_index] *
        centroid[ alloc_rates[sub_index] ][ data_int ];
         }
       }
     }   /* if( alloc_rates[sub_index] > 3 ) */
     else {
       for( ii = 0; ii < size_y2; ii++ ) {
         for( jj = 0; jj < size_x2; jj+=2 ) {
      data_int = huff_receive_code
        ( istrm, huff_tree[ alloc_rates[sub_index] ] );
#ifdef TEST_HC
      *(test2++) = data_int;
#endif
      image[xx+ii][yy+jj] = sd[sub_index] *\
        centroid[ alloc_rates[sub_index] ]
          [ data_int / N1[alloc_rates[sub_index]] ];
      image[xx+ii][yy+jj+1] = sd[sub_index] *\
        centroid[ alloc_rates[sub_index] ]
          [ data_int % N1[alloc_rates[sub_index]] ];
         }
       }
     }   /* else */
   }   /* if( (i+j != 0) && (alloc_rates[sub_index] != 0) ) */
   if( i+j != 0 )
     sub_index++;
      }
    }
  }
  return;
}
```

A.11 bstrm.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                  All Rights Reserved.                        */
/*         (developed by  Jerome Johnson,  Hamid Jafarkhani     */
/*            Ruplu Bhattacharya and Nariman Farvardin)         */
/* file: bstrm.c                                        */
/* by: Jerome Johnson                                  */
/* description: Huffman bitstream functions            */
/* cl30 –v40 –g –as –mn –mr –mf –o2 bstrm.c            */
#include ”average.h”
#include ”bstrm.h”
#include ”ports.h”
#ifdef _TMS320C40
#include <compt40.h>
#endif
void bs_init( bit_stream_t *stream, void *base ) {
  stream–>root    = base;
  stream–>curr    = stream–>root;
  stream–>buff    = 0;
  stream–>bufflen = 0;
  stream–>bit     = –1;
  return;
}
void bs_reset_stream( bit_stream_t *stream ) {
  stream–>curr    = stream–>root;
  stream–>buff    = 0;
  stream–>bufflen = 0;
  stream–>bit     = –1;
  return;
}
void bs_flush_bitbuff( bit_stream_t *ostream ) {
  if( ostream–>bufflen ) {
    *(ostream–>curr++) = ostream–>buff;
    ostream–>buff = 0;
    ostream–>bufflen = 0;
  }
  return;
}
void bs_write_bitfield( bit_stream_t *ostream, bitfield_t field )
{
  int i, len, shift;
  subfield_t tmp[5] = {0,0,0,0,0};
  len = ostream–>bufflen + field.len;
  shift = BS_SUBFIELD_LEN – ostream–>bufflen;
  for( i = 0; i < ((field.len+31)>>5); i++ )
    tmp[i] = (field.bits[i] >> ostream–>bufflen);
  tmp[0] |= ostream–>buff;
  for( i = 1; i < ((len+31)>>5); i++ ) {
    tmp[i] |= (ostream–>bufflen) ? (field.bits[i–1] << shift) : 0;
  }
  for( i = 0; i < (len>>5); i++ )
    *(ostream–>curr++) = tmp[i];
  ostream–>bufflen = len & BS_5LSB_MASK;
  ostream–>buff = (len & BS_5LSB_MASK) ?
    ( tmp[i] & (0xffffffff << (BS_SUBFIELD_LEN–ostream–>bufflen))
) : 0;
  return;
}
void bs_write_subfield( bit_stream_t *ostream, bitfield_t field )
{
  int  len, shift;
  len = ostream–>bufflen + field.len;
  shift = BS_SUBFIELD_LEN – ostream–>bufflen;
  ostream–>buff |= *field.bits >> ostream–>bufflen;
  if( len >= 32 ) {
    *(ostream–>curr++) = ostream–>buff;
    ostream–>buff = (ostream–>bufflen) ? (*field.bits<<shift) : 0;
  }
  ostream–>bufflen = len & BS_5LSB_MASK;
  ostream–>buff &= (ostream–>bufflen) ?
    ( 0xffffffff << (BS_SUBFIELD_LEN–ostream–>bufflen) ) : 0;

  return;
}
#ifdef _TMS320C40
void bs_send_stream( bit_stream_t ostream ) {
  static average_t ave_bitrate = { 0, 1 };
  static unsigned int  length;
  length = ostream.curr – ostream.root;
  send_msg( OSTREAM_PORT, ostream.root, length, 1 );
  UPDATE_AVE( &ave_bitrate, length );
  return;
}
```

```
void bs_receive_stream( bit_stream_t istream ) {
  receive_msg( ISTREAM_PORT, istream.root, 1 );
  while( chk_dma(ISTREAM_PORT) );
  return;
}
#endif
#ifndef _TMS320C40
void bs_print_stream( bit_stream_t ostream ) {
  int length, i;
  length = ostream.curr – ostream.root;
  printf( ”length: %i\n”, length );
  for( i = 0; i < length; i++ )
    printf( ”index: %i  value: %0#8x\n”, i, ostream.root[i] );
  printf( ”buff:    %0#10x\n”,  ostream.buff );
  printf( ”bufflen: %i\n”,  ostream.bufflen );
  return;
}
void main( void ) {
  subfield_t streambase[100], fieldbase[8] = {0,0,0,0,0,0,0,0};
  bit_stream_t ostream;
  bit_stream_t istream;
  bitfield_t bf;
  int i = 0, j = 0;
  average_t ave;
  RESET_AVE( &ave );
  while( i != –1 ) {
    printf( ”sample: ” );
    scanf( ”%i”, &i );
    UPDATE_AVE( &ave, i );
    printf( ”\n average = %f; count = %i\n”, ave.average,
ave.count );
  }
#if 0
  bf.bits = fieldbase;
  bs_init( &ostream, streambase );
  bs_init( &istream, streambase );
  bs_print_stream( ostream );
  for( i = 0; i < 10; i++ ){
    bs_write_word( &ostream, i );
    printf( ”%x\n”, bs_read_word( &istream ) );
  }
  for( j = 0; j < 5; j++ ) {
    printf( ”\nfieldlen: ” );
    scanf( ”%i”, &bf.len );
    for( i = 0; i < ((bf.len+31)>>5); i++ ) {
      printf( ”field: ” );
      scanf( ”%x”, &bf.bits[i] );
    }
    bs_write_subfield( &ostream, bf );
    bs_print_stream( ostream );
  }
  bs_reset_stream( &ostream );
  printf( ”\n” );
  bs_print_stream( ostream );
  printf( ”\n” );
  for( i = 0; i < 5*32; i++ )
    printf( ”%c”, bs_read_bit( &istream ) ? ’1’ : ’0’ );
  printf( ”\n” );
  printf( ”\n” );
  printf( ”\n” );
  for( j = 0; j < 5; j++ ) {
    printf( ”\nfieldlen: ” );
    scanf( ”%i”, &bf.len );
    for( i = 0; i < ((bf.len+31)>>5); i++ ) {
      printf( ”field: ” );
      scanf( ”%x”, &bf.bits[i] );
    }
    bs_write_subfield( &ostream, bf );
    bs_print_stream( ostream );
  }
  bs_flush_bitbuff( &ostream );
  bs_print_stream( ostream );
#endif
  return;
}
#endif
```

A.12 grab.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                        */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: grab.c                                         */
/* by: Jerome Johnson                                   */
/* description: video capture functions                */
/* cl30 –v40 –g –as –mn –mr –mf –o2 grab.c              */
#include <compt40.h>
#include <dma40.h>
#include ”grab.h”
#include ”options.h”
#include ”ports.h”
typedef unsigned long u_long;
void unpack_frame( unsigned long *src, float *dst ) {
  u_long *max;
#ifdef BYTE_REVERSED
  for( max = src + ((u_long)(SIZE_X * SIZE_Y) >> 2); src < max;
src++ ) {
    *(dst++) = (float)( (*src      ) & 0xff );
    *(dst++) = (float)( (*src >>  8) & 0xff );
    *(dst++) = (float)( (*src >> 16) & 0xff );
    *(dst++) = (float)( (*src >> 24) & 0xff );
  }
#else
  /* unpack and convert to float */
  for( max = src + ((u_long)(SIZE_X * SIZE_Y) >> 2); src < max;
src++ ) {
    *(dst++) = (float)( (*src >> 24) & 0xff );
    *(dst++) = (float)( (*src >> 16) & 0xff );
    *(dst++) = (float)( (*src >>  8) & 0xff );
    *(dst++) = (float)( (*src      ) & 0xff );
  }
#endif
  return;
}
void grab_frame( int ch_no, void *dst, int dst_idx ) {
  extern DMA_REG grab_table[];
  /* set up destination for async trans */
  grab_table[ 0 ].dma_regs.dst     = dst;
  grab_table[ 0 ].dma_regs.dst_idx = dst_idx;
  DMA_RESTART( ch_no );
  return;
}
```

A.13 unpackf.asm

```
* file: unpackf.asm
* by: Jerome Johnson
* description: unpacks and converts video frame to floats
* cl30 –v40 –g –as –mn –mr –mf unpackf.asm
FP      .set AR3
        .global _unpackf
        .align                  ; align on cache boundary
        .text
; void unpackf( void *src, void *dst, int size_words );
_unpackf
        PUSH AR4
        LSH  –1, R3, RC     ; RC = size/2
        SUBI  1, RC         ; RC = size/2 – 1
        ADDI  1, AR2, AR1   ; AR1 = AR2 + 1
; AR2 is even words source  ; AR1 is odd words source
; AR0 is even bytes dest    ; AR4 is odd bytes dest
; R0 is even word           ; R3 is odd word
; R1, R2 are unpacked bytes –> floats
; IR0 is 2                  ; RC = size/2 – 1
        RPTBD UPF1
        LDA   2, IR0        ; IR0 is 2
        LDA   R2, AR0       ; AR0 = R2 is even bytes dest
        ADDI  1, R2, AR4    ; AR4 = R2 + 1
; BEGIN OF LOOP
        LDI *AR2++(IR0), R0
||      LDI *AR1++(IR0), R3
        LBU0  R0, R1
        FLOAT R1, R1
        LBU1  R0, R2
        FLOAT R2, R2
        STF R1, *AR0++(IR0)
||      STF R2, *AR4++(IR0)
        LBU2  R0, R1
        FLOAT R1, R1
        LBU3  R0, R2
        FLOAT R2, R2
        STF R1, *AR0++(IR0)
||      STF R2, *AR4++(IR0)
        LBU0  R3, R1
        FLOAT R1, R1
        LBU1  R3, R2
        FLOAT R2, R2
        STF R1, *AR0++(IR0)
||      STF R2, *AR4++(IR0)
        LBU2  R3, R1
        FLOAT R1, R1
        LBU3  R3, R2
        FLOAT R2, R2
UPF1    STF R1, *AR0++(IR0)
||      STF R2, *AR4++(IR0)
        POP AR4
        RETS
        .end
```

A.14 disp.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: disp.c                                         */
/* by: Jerome Johnson                                   */
/* description: video display functions                 */
/* cl30 –v40 –g –as –mn –mr –mf –o2 disp.c              */
#include <compt40.h>
#include <dma40.h>
#include ”grab.h”
#include ”options.h”
```

```
#include ”ports.h”
typedef unsigned long u_long;
void send_frame( int ch_no, void *src, int src_idx ) {
  extern DMA_REG disp_table[];
  /* set up source for async trans */
  disp_table[ 0 ].dma_regs.src     = src;
  disp_table[ 0 ].dma_regs.src_idx = src_idx;
  DMA_RESTART( ch_no );
  return;
}
void pack_frame( float *src, unsigned long *dst ) {
  float *max;
  u_long x;
  float y;
  int j;
  max = src + (SIZE_X*SIZE_Y);
  for( ; src < max; src+=4 ) {
    x = 0;
    for( j = 0; j < 4; j++ ) {
      y = src[j] + .5;
      if( y > 255 )
   x |= 255 << (j*8);
      else if( y < 0 )
   ;
      else
   x |= (u_long)(y) << (j*8);
    }
    *(dst++) = x;
  }
  /* temporary ”fix” for cable problem */
  /* pad transfer */
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0xffffffff;
  *(dst++) = 0x00000000;
  *(dst++) = 0x00000000;
  *(dst++) = 0x00000000;
  *(dst++) = 0x00000000;
  *(dst++) = 0x00000000;
  *(dst++) = 0x00000000;
  *(dst++) = 0xffffffff;
  return;
}
```

A.15 packf.asm

```
* file: packf.asm
* by: Jerome Johnson
* description: converts image of floats to unsigned chars and
packs
*              into video frame
* cl30 –v40 –g –as –mn –mr –mf unpackf.asm
FP      .set AR3
        .global _packf
        .align                   ; align on cache boundary
        .text
; void packf( void *src,  void *dst, int size_bytes )
_packf
        LSH  –2, R3, RC     ; RC = size/4
        SUBI  1, RC         ; RC = size/4 – 1
        LDA   R2, AR0       ; AR0 = R2 is packed word dest
; AR2 is float byte source  ; AR0 word dest
; R0  is packed word
; RC is size/4 –1 where size is the number of bytes
        RPTBD PF1
        POP AR1             ; POP return address for later branch
        NOP
        FIX *AR2++, R0      ; get and fix the first byte
; BEGIN OF LOOP
        LDIN 0, R0
        CMPI 255, R0
        LDIGE 255, R0
        FIX *AR2++, R1
        LDIN 0, R1
        CMPI 255, R1
        LDIGE 255, R1
        LWL1 R1, R0
        FIX *AR2++, R1
        LDIN 0, R1
        CMPI 255, R1
        LDIGE 255, R1
        LWL2 R1, R0
        FIX *AR2++, R1
        LDIN 0, R1
        CMPI 255, R1
        LDIGE 255, R1
        LWL3 R1, R0
PF1     STI R0, *AR0++     ; store the packed word R0
||      FIX *AR2++, R0     ; get and fix the next byte
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI –1, *AR0++
   STI  0, *AR0++
   STI  0, *AR0++
   STI  0, *AR0++
   STI  0, *AR0++
   BUD AR1           ; return
   STI  0, *AR0++
   STI  0, *AR0++
   STI –1, *AR0++
        .end
```

A.16 adma.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: adma.c                                           */
/* by: Jerome Johnson                                     */
/* description: DMA autoinitialization function for CPU A */
/*              video display                             */
/* #include in receive.c                                  */
#include <compt40.h>
#include <dma40.h>
#include ”grab.h”
#include ”options.h”
#include ”ports.h”
#define GRABPORT 0
#define DISPPORT      A_PORTTO_B
#define DISPLAY_ADDR  COMPORT_OUT_ADDR( A_PORTTO_B )
#define DISPLAY_STEP  0
DMA_REG disp_table[1];
void init_dma( void ) {
  extern DMA_REG disp_table[];
  /* set up dma auto init table for disp frame */
  disp_table[ 0 ]._gctrl._intval   = 0x00c4008d;
  disp_table[ 0 ].dma_regs.src     = 0;
  disp_table[ 0 ].dma_regs.src_idx = 0;
  disp_table[ 0 ].dma_regs.count   = (SIZE_X*SIZE_Y>>2) + 16;
  disp_table[ 0 ].dma_regs.dst     = DISPLAY_ADDR;
  disp_table[ 0 ].dma_regs.dst_idx = DISPLAY_STEP;
  disp_table[ 0 ].dma_link = (unsigned long *)&disp_table[ 0 ];
  /* initialize channel */
  DMA_ADDR( DISPPORT )–>_gctrl._intval = 0x0000008d;
  DMA_ADDR( DISPPORT )–>dma_regs.count = 0;
  DMA_ADDR( DISPPORT )–>dma_link       = (unsigned long
*)&disp_table[0];
  /* set DIE register it enable dma interrupts */
  load_iie( iie_value() & ~(0x3f<<25) ); /* DMA interrupts to cpu
disabled */
```

```
  dma_sync_set( DISPPORT, 1, 1 );  /* DIE = write sync on comport
*/
  /* set flag to signal 0th send frame complete */
  /* set_iif_flag( 25+DISPPORT ); */
  INT_ENABLE();
  return;
}
```

A.17 bdma.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: bdma.c                                           */
/* by: Jerome Johnson                                     */
/* description: DMA autoinitialization function for CPU B */
/*              video display                             */
/* #include in receive.c                                  */
#include <compt40.h>
#include <dma40.h>
#include ”grab.h”
#include ”options.h”
#include ”ports.h”
#define GRABPORT 0
#define DISPPORT      B_PORTTO_X
#define DISPLAY_ADDR  /*(long *)(0x80000000)*/ COMPORT_OUT_ADDR(
B_PORTTO_X )
#define DISPLAY_STEP  0
DMA_REG disp_table[3];
void init_dma( void ) {
  extern DMA_REG disp_table[];
  /* clean out comports */
  while( cp_in_level( B_PORTFROM_A ) )
    in_word( B_PORTFROM_A );
  /* set up dma auto init table for disp frame */
  disp_table[ 1 ]._gctrl._intval   = 0x00c00049;
  disp_table[ 1 ].dma_regs.src     = (long *)0x4000ffff;
  disp_table[ 1 ].dma_regs.src_idx = 0;
  disp_table[ 1 ].dma_regs.count   = 1;
  disp_table[ 1 ].dma_regs.dst     = (long *)0x4000ffff;
  disp_table[ 1 ].dma_regs.dst_idx = 0;
  disp_table[ 1 ].dma_link = (unsigned long *)&disp_table[ 2 ];
  disp_table[ 2 ]._gctrl._intval   = 0x00c4008d;
  disp_table[ 2 ].dma_regs.src     = COMPORT_IN_ADDR( B_PORTFROM_A
);
  disp_table[ 2 ].dma_regs.src_idx = 0;
  disp_table[ 2 ].dma_regs.count   = (SIZE_X*SIZE_Y>>2) + 16;
  disp_table[ 2 ].dma_regs.dst     = DISPLAY_ADDR;
  disp_table[ 2 ].dma_regs.dst_idx = DISPLAY_STEP;
  disp_table[ 2 ].dma_link = (unsigned long *)&disp_table[ 0 ];
  disp_table[ 0 ]._gctrl._intval   = 0x00c40089;
  disp_table[ 0 ].dma_regs.src     = 0;
  disp_table[ 0 ].dma_regs.src_idx = 0;
  disp_table[ 0 ].dma_regs.count   = (SIZE_X*SIZE_Y>>2) + 16;
  disp_table[ 0 ].dma_regs.dst     = DISPLAY_ADDR;
  disp_table[ 0 ].dma_regs.dst_idx = DISPLAY_STEP;
  disp_table[ 0 ].dma_link = (unsigned long *)&disp_table[ 1 ];
  /* initialize channel */
  DMA_ADDR( DISPPORT )–>_gctrl._intval = 0x0000008d;
  DMA_ADDR( DISPPORT )->dma_regs.count = 0;
  DMA_ADDR( DISPPORT )–>dma_link       = (unsigned long
*)&disp_table[1];
  DMA_RESTART( DISPPORT );
  /* set DIE register it enable dma interrupts */
  load_iie( iie_value() & ~(0x3f<<25) ); /* DMA interrupts to cpu
disabled */
  dma_sync_set( DISPPORT, 1, 1 );  /* DIE = write sync on comport
*/
  dma_sync_set( DISPPORT, 4, 0 );  /* DIE = read sync on IIOF2
from A */
  set_iiof( 2, 3 );                /* set IIOF2 as level trigger
int */
  /* set flag to signal 0th send frame complete */
  /* set_iif_flag( 25+DISPPORT ); */
  INT_ENABLE();
  return;
}
```

A.18 cdma.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: cdma.c                                           */
/* by: Jerome Johnson                                     */
/* description: DMA autoinitialization function for CPU C */
/*              video capture                             */
/* #include in send.c                                     */
#include <compt40.h>
#include <dma40.h>
#include <intpt40.h>
#include ”grab.h”
#include ”options.h”
#include ”ports.h”
#define DISPPORT 0
#define GRABPORT     C_PORTFROM_D
#define CAMERA_ADDR  COMPORT_IN_ADDR( C_PORTFROM_D )
#define CAMERA_STEP  0
DMA_REG grab_table[1];
void init_dma( void ) {
  extern DMA_REG grab_table[];
  /* clean out comports */
  while( cp_in_level( GRABPORT ) )
    in_word( GRABPORT );
  /* set up dma auto init table for frame grab */
  grab_table[ 0 ]._gctrl._intval   = 0x00c4004d;
  grab_table[ 0 ].dma_regs.src     = CAMERA_ADDR;
  grab_table[ 0 ].dma_regs.src_idx = CAMERA_STEP;
  grab_table[ 0 ].dma_regs.count   = SIZE_X*SIZE_Y>>2;
  grab_table[ 0 ].dma_regs.dst     = 0;
  grab_table[ 0 ].dma_regs.dst_idx = 0;
  grab_table[ 0 ].dma_link = (unsigned long *)&grab_table[ 0 ];
  /* initialize channel */
  DMA_ADDR( GRABPORT )–>_gctrl._intval = 0x0000004d;
  DMA_ADDR( GRABPORT )–>dma_regs.count = 0;
  DMA_ADDR( GRABPORT )–>dma_link       = (unsigned long
*)&grab_table[0];
```

```
  /* set DIE register it enable dma interrupts */
  load_iie( iie_value() & ~(0x3f<<25) ); /* DMA interrupts to cpu
disabled */
  dma_sync_set( GRABPORT, 1, 0 );       /* DIE = read sync on
comport */
  INT_ENABLE();
  return;
}
```

A.19 ddma.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*        (developed by  Jerome Johnson,  Hamid Jafarkhani      */
/*           Ruplu Bhattacharya and Nariman Farvardin)          */
/* file: ddma.c                                           */
/* by: Jerome Johnson                                     */
/* description: DMA autoinitialization function for CPU D */
/*              video capture                             */
/* #include in send.c                                     */
#include <compt40.h>
#include <dma40.h>
#include ”grab.h”
#include ”options.h”
#include ”ports.h”
#define DISPPORT 0
#define GRABPORT     D_PORTFROM_X
#define CAMERA_ADDR  COMPORT_IN_ADDR( D_PORTFROM_X )
#define CAMERA_STEP  0
DMA_REG grab_table[2];
void init_dma( void ) {
  extern DMA_REG grab_table[];
  /* clean out comports */
  while( cp_in_level( GRABPORT ) )
    in_word( GRABPORT );
  /* set up dma auto init table for grab frame grab */
  grab_table[ 1 ]._gctrl._intval   = 0x00c4004d;
  grab_table[ 1 ].dma_regs.src     = CAMERA_ADDR;
  grab_table[ 1 ].dma_regs.src_idx = CAMERA_STEP;
  grab_table[ 1 ].dma_regs.count   = SIZE_X*SIZE_Y>>2;
  grab_table[ 1 ].dma_regs.dst     = COMPORT_OUT_ADDR( D_PORTTO_C
);
  grab_table[ 1 ].dma_regs.dst_idx = 0;
  grab_table[ 1 ].dma_link = (unsigned long *)&grab_table[ 0 ];
  grab_table[ 0 ]._gctrl._intval   = 0x00c4004d;
  grab_table[ 0 ].dma_regs.src     = CAMERA_ADDR;
  grab_table[ 0 ].dma_regs.src_idx = CAMERA_STEP;
  grab_table[ 0 ].dma_regs.count   = SIZE_X*SIZE_Y>>2;
  grab_table[ 0 ].dma_regs.dst     = 0;
  grab_table[ 0 ].dma_regs.dst_idx = 0;
  grab_table[ 0 ].dma_link = (unsigned long *)&grab_table[ 1 ];
  /* initialize channel */
  DMA_ADDR( GRABPORT )–>_gctrl._intval = 0x0000004d;
  DMA_ADDR( GRABPORT )–>dma_regs.count = 0;
  DMA_ADDR( GRABPORT )–>dma_link       = (unsigned long
*)&grab_table[1];
  DMA_RESTART( GRABPORT );
  /* set IIE and DIE register it enable dma interrupts */
  load_iie( iie_value() & ~(0x3f<<25) ); /* DMA interrupts to cpu
disabled */
  dma_sync_set( GRABPORT, 1, 0 );       /* DIE = read sync on
comport */
  INT_ENABLE();
  return;
}
```

A.20 sdma.c

```
/* Copyright (c) 1996, University of Maryland at College Park.  */
/*                   All Rights Reserved.                       */
/*         (developed by  Jerome Johnson,  Hamid Jafarkhani     */
/*            Ruplu Bhattacharya and Nariman Farvardin)         */
/* file: sdma.c                                            */
/* by: Jerome Johnson                                      */
/* description: DMA autoinitialization function for single */
/*                CPU video capture and display            */
/* #include in single.c                                    */
#include <compt40.h>
#include <dma40.h>
#include <intpt40.h>
#include ”grab.h”
```

```
#include ”options.h”
#include ”ports.h”
#define GRABPORT      C_PORTFROM_X
#define CAMERA_ADDR   COMPORT_IN_ADDR( C_PORTFROM_X )
#define CAMERA_STEP   0
#define DISPPORT      A_PORTTO_X
#define DISPLAY_ADDR  COMPORT_OUT_ADDR( A_PORTTO_X )
#define DISPLAY_STEP  0
DMA_REG grab_table[1];
DMA_REG disp_table[1];
void init_dma( void ) {
  extern DMA_REG grab_table[];
  extern DMA_REG disp_table[];
  /* clean out input comport */
  while( cp_in_level( GRABPORT ) )
    in_word( GRABPORT );
  /* set up dma auto init table for frame grab */
  grab_table[ 0 ]._gctrl._intval   = 0x00c4004d;
  grab_table[ 0 ].dma_regs.src     = CAMERA_ADDR;
  grab_table[ 0 ].dma_regs.src_idx = CAMERA_STEP;
  grab_table[ 0 ].dma_regs.count   = SIZE_X*SIZE_Y>>2;
  grab_table[ 0 ].dma_regs.dst     = 0;
  grab_table[ 0 ].dma_regs.dst_idx = 0;
  grab_table[ 0 ].dma_link = (unsigned long *)&grab_table[ 0 ];
  /* initialize channel */
  DMA_ADDR( GRABPORT )–>_gctrl._intval = 0x0000004d;
  DMA_ADDR( GRABPORT )–>dma_regs.count = 0;
  DMA_ADDR( GRABPORT )–>dma_link       = (unsigned long
*)&grab_table[0];
  /* set DIE register to enable dma interrupts */
  dma_sync_set( GRABPORT, 1, 0 );       /* DIE = read sync on
comport */
  /* set up dma auto init table for disp frame */
  disp_table[ 0 ]._gctrl._intval   = 0x00c4008d;
  disp_table[ 0 ].dma_regs.src     = 0;
  disp_table[ 0 ].dma_regs.src_idx = 0;
  disp_table[ 0 ].dma_regs.count   = (SIZE_X*SIZE_Y>>2) + 16;
  disp_table[ 0 ].dma_regs.dst     = DISPLAY_ADDR;
  disp_table[ 0 ].dma_regs.dst_idx = DISPLAY_STEP;
  disp_table[ 0 ].dma_link = (unsigned long *)&disp_table[ 0 ];
  /* initialize channel */
  DMA_ADDR( DISPPORT )–>_gctrl._intval = 0x0000008d;
  DMA_ADDR( DISPPORT )–>dma_regs.count = 0;
  DMA_ADDR( DISPPORT )–>dma_link       = (unsigned long
*)&disp_table[0];
  /* set DIE register to enable dma interrupts */
  load_iie( iie_value() & ~(0x3f<<25) ); /* DMA interrupts to cpu
disabled */
  dma_sync_set( DISPPORT, 1, 1 );  /* DIE = write sync on comport
*/
  /* set flag to signal 0th send frame complete */
  /* set_iif_flag( 25+DISPPORT ); */
  INT_ENABLE();
  return;
}
```

A.21 options.h

```
#define SIZE_X 176
#define SIZE_Y 144
#define DECOMPOSITION_LEVEL 3
#define TARGET_RATE .25
#define MAXSTREAM 4096
#define CLOCK_PER_SEC 16000000.0
```

A.22 waveletc.h

```
#ifndef WAVELETC_H
#define WAVELETC_H
#ifdef _TMS320C40
#include ”sects.h”
#endif
#include ”options.h”
#define Hmax 16
#define TRUE 1
#define FALSE 0
#define MAX 255
#define QUATREU 4
#define SQR(x) ((x)*(x))
#define MAXRC 276
#define MARGE 5
```

```
#define MARGE2 10
/* function prototypes */
void convolutiond( void *X, void *Y, void *H, int size, int idx );
void convolutionr( void *X, void *Y, void *H, int size, int idx );
void wavelet_dec( float (*image)[SIZE_X], int SIZE_x, int SIZE_y,
        int decomposition_level, int ch_no );
void wavelet_rec( float (*image)[SIZE_X], int SIZE_x, int SIZE_y,
        int decomposition_level, int ch_no );
#endif
```

A.23 utsq.h

```
#ifndef UTSQ_H
#define UTSQ_H
#include ”bstrm.h”
#include ”options.h”
#ifndef NULL
#define NULL 0
#endif
#define BIT_BUFFER_LEN BS_SUBFIELD_LEN
typedef struct mynode {
  struct mynode *left, *right;
  int code;
} mynode_t;
typedef bitfield_t codeword_t;
#define is_leaf(root) ( ((root)–>left == NULL) && ((root)–>right
== NULL) )
#define is_nonleaf(root) ( (root)–>left || (root)–>right )
/* HUFFMAN FUNCTIONS */
int  huff_receive_code( bit_stream_t *istream, mynode_t *root );
/* void init_cb( codeword_t *cbook[], short *N ); */
void hc2_init_cb( codeword_t *cbook[], short *N );
void build_hufftree( mynode_t *stree, mynode_t *root );
void init_hufftree( mynode_t *huff_tree[], int huffsize[] );
void init_centroid( float *centroid[], short *N, short *n1 );
/* BIT ALLOCATION FUNCTIONS */
void var_10( float image[][SIZE_X], int size_x, int size_y, float
*var,\
        float *mean_LFS );
void var_7( float image[][SIZE_X], int size_x, int size_y, float
*var,\
       float *mean_LFS );
int max_slope( float *slope, int no_sub );
void bit_alloc( float *rate_sub, float *dist_sub,float
*rate_weight,\
                float *var,int max_R_D,float target_rate,float
*rate,\
                int no_sub,unsigned short *alloc_rates );
void var_to_sd( float *var, float *sd, unsigned short *sd_quan );
void quan_mean( float mean_LFS, float *quan_mean_LFS,\
           unsigned short *mean_index);
/* QUANTIZER FUNCTIONS */
void write_out_stream( bit_stream_t *ostream, float
image[][SIZE_X],
             int size_x, int size_y,
             unsigned short *alloc_rates, short *N, short *N1,
             float *delta, float quan_mean_LFS, float *sd );
void write_header( bit_stream_t *ostream, unsigned short *sd_quan,
         unsigned short mean_index,
         unsigned short *alloc_rates );
void read_header( bit_stream_t *istream, unsigned short *sd_quan,
        unsigned short *mean_index,
        unsigned short *alloc_rates,
        float *sd, float *quan_mean_LFS );
void read_in_stream( bit_stream_t *istrm, float image[][SIZE_X],
           int size_x, int size_y,
           unsigned short *alloc_rates,
           float quan_mean_LFS, float *sd, short N1[] );
#endif
```

A.24 mean.h

```
#define MEAN_NORM 0.125
#define SD_NORM 0.03125
#define MAX 255
#define DECOMPOSITION_LEVEL 3
#define NO_SUB 7
#define MAX_R_D 25
#define BYTE_SIZE 8
#define MEAN_SIZE 11
#define ALLOC_RATE_SIZE 5
```

```
#define MEAN_INDEX_SIZE 11
#define Epsilon 1e–35
```

A.25 bstrm.h

```
#ifndef BSTRM_H
#define BSTRM_H
#define BS_MAXSTREAM 4096
#define BS_WORDS 4
#define BS_WORDLEN (sizeof(unsigned int))
#define BS_SUBFIELD_LEN /* (BS_WORDLEN*8) */ 32
#define BS_BITLEN (BS_WORDBITLEN*BS_WORDS)
#define BS_5LSB_MASK 31
typedef unsigned int subfield_t;
typedef struct {
  subfield_t *bits;
  unsigned int len;
} bitfield_t;
typedef struct {
  subfield_t *root;
  subfield_t *curr;
  subfield_t buff;
  int bufflen;
  int bit;
} bit_stream_t;
void bs_init( bit_stream_t *stream, void *base );
void bs_reset_stream( bit_stream_t *stream );
void bs_send_stream( bit_stream_t ostream );
void bs_receive_stream( bit_stream_t istream );
void bs_flush_bitbuff( bit_stream_t *ostream );
void bs_write_bitfield( bit_stream_t *ostream, bitfield_t field );
void bs_write_subfield( bit_stream_t *ostream, bitfield_t field );
#define bs_read_bit( istrm ) (  ( ((istrm)–>bufflen) ?
(istrm)–>bufflen–– : \
  ( (istrm)–>bufflen=31, (istrm)–>buff=*((istrm)–>curr++) ) ), \
  ( (istrm)–>buff & ((unsigned)1<<(istrm)–>bufflen) )  )
#define bs_write_word( ostrm, word ) *((ostrm)–>curr++) = word
#define bs_read_word( istrm )        *((istrm)–>curr++)
void reset_istream( void );
void receive_stream( void );
```

```
#endif
```

A.26 grab.h

```
#ifndef GRAB_H
#define GRAB_H
void init_dma( void );
void grab_frame( int ch_no, void *dst, int dst_idx );
void send_frame( int ch_no, void *src, int src_idx );
void unpack_frame( unsigned long *src, float *dst );  /* c version
*/
void unpackf( void *src, void *dst, int size_words ); /* asm
version */
void pack_frame( float *src, unsigned long  *dst );   /* c version
*/
void packf( void *src,  void *dst, int size_bytes );  /* asm
version */
#endif
```

A.27 ports.h

```
#define READY     0x600d
#define NOTREADY  0xbad
/* local control synchronization register */
#define LCSR_REG ((unsigned int *)0x00400000)
#define LCSR_SET_INT( cpu ) ( *LCSR_REG |= (1 << cpu) )
#define LCSR_RESET_INT( cpu ) ( *LCSR_REG &= ~(1 << cpu) )
#define A_PORTTO_B   0
#define A_PORTTO_C   1
#define A_PORTTO_D   3
#define A_PORTTO_X   2
#define A_PORTFROM_B 0
#define A_PORTFROM_C 4
#define A_PORTFROM_D 3
#define A_PORTFROM_X 5
#define B_PORTTO_A   3
#define B_PORTTO_C   0
#define B_PORTTO_D   1
#define B_PORTTO_X   2
#define B_PORTFROM_A 3
#define B_PORTFROM_C 0
#define B_PORTFROM_D 4
```

```
#define B_PORTFROM_X 5
#define C_PORTTO_A   1
#define C_PORTTO_B   3
#define C_PORTTO_D   0
#define C_PORTTO_X   2
#define C_PORTFROM_A 4
#define C_PORTFROM_B 3
#define C_PORTFROM_D 0
#define C_PORTFROM_X 5
#define D_PORTTO_A   0
#define D_PORTTO_B   1
#define D_PORTTO_C   3
#define D_PORTTO_X   2
#define D_PORTFROM_A 0
#define D_PORTFROM_C 3
#define D_PORTFROM_B 4
#define D_PORTFROM_X 5
#define ISTREAM_PORT 4
#define OSTREAM_PORT 1
```

A.28 sects.h

```
/* file: sects.h */
/* by Jerome Johnson */
#ifndef SECTS_H
#define SECTS_H
#ifndef CAT
#define TOSTRING( x ) #x
#define TOSTR( x ) TOSTRING( x )
#define CONCAT( x, y) x ## y
#define CAT( x, y ) CONCAT( x, y )
#endif /* CAT */
/* define a macro for defineing external labels outside of .bss */
#ifndef USECT
#define USECT( LABEL, SNAME, SIZE ) \
    asm( ”                 .global _” TOSTR( LABEL )  ); \
    asm( ”_” TOSTR( LABEL ) ” .usect ” TOSTR( SNAME ) ”, ” TOSTR(
SIZE ) )
#endif /* USECT */
#endif /* SECTS_H */
```

A.29 average.h

```
#ifndef AVERAGE_H
#define AVERAGE_H
typedef struct {
  float average;
  unsigned int count;
} average_t;
#define UPDATE_AVE( ave, sample ) \
  (ave)–>average = ( ((float)++(ave)–>count – 1) / (ave)–>count )
\
  * (ave)–>average + (float)sample/(ave)–>count
#define RESET_AVE( ave ) (ave)–>average = 0; (ave)–>count = 0
#endif
```

A.30 meml.cmd

```
MEMORY
{
/*     INTROM(RX):     origin = , length =  */
/*     LEPROM(RX):     origin = , length =  */
     INTRAM0(RWXI):  origin = 0x002ff800,  length = 0x400
     INTRAM1(RWXI):  origin = 0x002ffc00,  length = 0x400
     LSRAM0(RWXI):    origin = 0x40000000,  length = 0x10000
     GSRAM0(RWXI):   origin = 0x80000000,  length = 0x10000
     GSRAM1(RWXI):   origin = 0x80010000,  length = 0x20000
}
```

A.31 sendc.cmd

```
/* OPTIONS */
–cr
–stack 0x200
send.obj
grab.obj
utsqen.obj
bitalloc.obj
bstrm.obj
decomp.obj
cnvld.obj
unpackf.obj
filters.obj
```

```
cdbk.obj
–m sendc.map
–o sendc.out
–l rts40r.lib
–l prts40r.lib
SECTIONS
{
  IMAGE$:    > LSRAM0, align 0x10000
  FILTER$:   > INTRAM0, align 0x10
  XA$:       > INTRAM0, align 0x10
  XB$:       > INTRAM1, align 0x10
  .sysmem:   > INTRAM1
  .stack:    > INTRAM0
  .bss:      > INTRAM1
  .text:     > LSRAM0
  .data:     > LSRAM0
  .const:    > LSRAM0
}
```

A.32 sendd.cmd

```
/* OPTIONS */
–cr
–stack 0x200
send.obj
grab.obj
utsqen.obj
bitalloc.obj
bstrm.obj
decomp.obj
cnvld.obj
unpackf.obj
filters.obj
cdbk.obj
–m sendd.map
–o sendd.out
–l rts40r.lib
–l prts40r.lib
SECTIONS
```

```
{
  IMAGE$:    > LSRAM0, align 0x10000
  FILTER$:   > INTRAM0, align 0x10
  XA$:       > INTRAM0, align 0x10
  XB$:       > INTRAM1, align 0x10
  .sysmem:   > INTRAM1
  .stack:    > INTRAM0
  .bss:      > INTRAM1
  .text:     > LSRAM0
  .data:     > LSRAM0
  .const:    > LSRAM0
}
```

A.33 recea.cmd

```
/* OPTIONS */
–cr
–stack 0x200
receive.obj
disp.obj
utsqde.obj
bstrm.obj
recomp.obj
cnvlr.obj
packf.obj
filters.obj
cntrd.obj
hctree.obj
hcsize.obj
–m receivea.map
–o receivea.out
–l prts40r.lib
–l rts40r.lib
SECTIONS
{
  IMAGE$:    > LSRAM0, align 0x10000
  FILTER$:   > INTRAM0, align 0x10
  XA$:       > INTRAM0, align 0x10
  XB$:       > INTRAM1, align 0x10
  .sysmem:   > INTRAM1
  .stack:    > INTRAM0
  .bss:      > INTRAM1
  .text:     > LSRAM0
  .data:     > LSRAM0
  .const:    > LSRAM0
}
```

A.34 receb.cmd

```
/* OPTIONS */
–cr
–stack 0x200
receive.obj
disp.obj
utsqde.obj
bstrm.obj
recomp.obj
cnvlr.obj
packf.obj
filters.obj
cntrd.obj
hctree.obj
hcsize.obj
–m receiveb.map
–o receiveb.out
–l prts40r.lib
–l rts40r.lib
SECTIONS
{
  IMAGE$:    > LSRAM0, align 0x10000
  FILTER$:   > INTRAM0, align 0x10
  XA$:       > INTRAM0, align 0x10
  XB$:       > INTRAM1, align 0x10
  .sysmem:   > INTRAM1
  .stack:    > INTRAM0
  .bss:      > INTRAM1
  .text:     > LSRAM0
  .data:     > LSRAM0
  .const:    > LSRAM0
}
```

A.35 single.cmd

```
/* OPTIONS */
–cr
–stack 0x200
single.obj
grab.obj
disp.obj
utsqen.obj
utsqde.obj
bitalloc.obj
bstrm.obj
decomp.obj
recomp.obj
cnvld.obj
cnvlr.obj
unpackf.obj
packf.obj
filters.obj
cdbk.obj
hctree.obj
hcsize.obj
cntrd.obj
–m single.map
–o single.out
–l rts40r.lib
–l prts40r.lib
SECTIONS
{
  IMAGE$:    > LSRAM0, align 0x10000
  FILTER$:   > INTRAM0, align 0x10
  XA$:       > INTRAM0, align 0x10
  XB$:       > INTRAM1, align 0x10
  .sysmem:   > INTRAM1
  .stack:    > INTRAM0
  .bss:      > INTRAM1
  .text:     > LSRAM0
  .data:     > LSRAM0
  .const:    > LSRAM0
}
```

A.36 comptbls.cmd

```
cl30 –v40 –g –as filters.asm
cl30 –v40 –g –as cdbk.asm
cl30 –v40 –g –as hctree.asm
cl30 –v40 –g –as hcsize.asm
cl30 –v40 –g –as cntrd.asm
```

A.37 comp4.cmd

```
cl30 –v40 –g –as –dCPUC –mn –mr –mf –o2 send.c
cl30 –v40 –g –as –dCPUA –mn –mr –mf –o2 receive.c
cl30 –v40 –g –as –mn –mr –mf bitalloc.c
cl30 –v40 –g –as –mn –mr –mf –o2 utsqen.c
cl30 –v40 –g –as –mn –mr –mf –o2 utsqde.c
cl30 –v40 –g –as –mn –mr –mf –o2 grab.c
cl30 –v40 –g –as –mn –mr –mf unpackf.asm
cl30 –v40 –g –as –mn –mr –mf –o2 disp.c
cl30 –v40 –g –as –mn –mr –mf packf.asm
cl30 –v40 –g –as –mn –mr –mf –o2 bstrm.c
cl30 –v40 –g –as –mn –mr –mf –o2 decomp.c
cl30 –v40 –g –as –mn –mr –mf –o2 recomp.c
cl30 –v40 –g –as –mn –mr –mf cnvld.asm
cl30 –v40 –g –as –mn –mr –mf cnvlr.asm
lnk30 meml.cmd sendc.cmd
lnk30 meml.cmd recea.cmd
cl30 –v40 –g –as –dCPUD –mn –mr –mf –o2 send.c
cl30 –v40 –g –as –dCPUB –mn –mr –mf –o2 receive.c
lnk30 meml.cmd sendd.cmd
lnk30 meml.cmd receb.cmd
```

A.38 comp1.cmd

```
cl30 –v40 –g –as –mn –mr –mf –o2 single.c
cl30 –v40 –g –as –mn –mr –mf bitalloc.c
cl30 –v40 –g –as –mn –mr –mf –o2 utsqen.c
cl30 –v40 –g –as –mn –mr –mf –o2 utsqde.c
```

```
cl30 –v40 –g –as –mn –mr –mf –o2 grab.c
cl30 –v40 –g –as –mn –mr –mf unpackf.asm
cl30 –v40 –g –as –mn –mr –mf –o2 disp.c
cl30 –v40 –g –as –mn –mr –mf packf.asm
cl30 –v40 –g –as –mn –mr –mf –o2 bstrm.c
cl30 –v40 –g –as –mn –mr –mf –o2 decomp.c
cl30 –v40 –g –as –mn –mr –mf –o2 –dSINGLE recomp.c
cl30 –v40 –g –as –mn –mr –mf cnvld.asm
cl30 –v40 –g –as –mn –mr –mf cnvlr.asm
lnk30 meml.cmd single.cmd
```