The DSP6202 Processor Board Software

S.Simion. *CERN, 1 December 2000*

1 The DSP Software

1.1 What you should know

From the *TMS320C6000 Peripherals Reference Guide* you should read: Chapter 3 (C6202 Program and Data Memory), Chapter 5 (DMA), and Chapter 13 (Interrupts). From the *TMS320C6000 CPU and Instruction Set Reference Guide* you should read: Chapter 2 (CPU Data Paths and Control), Chapter 3 (C6202 Instruction Set), and Chapter 8 (Interrupts).

1.2 Data buffering and DMA scheduling logic

In the actual implementation, the data are transferred through four distinct buffers:

- The dual-port memory, accessible by the DSP as read-only external memory starting at address `0x01400000`, is organized as a circular buffer of 128 events.
- A double buffer for the input data is implemented in the DSP data memory. The events are brought into `data_in1` (or `data_in2`) by the DMA.
- Currently `NBUFOUT=4` buffers for the output data are implemented in the DSP data memory. The DSP optimal filtering code *loop26* reads directly from `data_in2` (or `data_in1`) and writes the output data to `data_out[n]` in a round-robin fashion.
- The output FIFO can hold up to 128 events, depending on the event size. The data are copied from `data_out` to the output FIFO by DMA, with write synchronization enabled in order not to write to an almost-full FIFO.

The main program manages the input and output data flow by scheduling the appropriate DMA transfers, calling the event processing routines, and raising the BUSY when necessary. The program acts as a state machine depending on:

- The number of events waiting in the dual-port memory.
- The states of the two input buffers, which can be:
  - **FREE**: able to receive a new event from the dual-port memory;
  - **BUSY**: an event is being copied from the dual-port memory to the buffer;
  - **READY**: the buffer contains a complete event ready to be processed by the DSP.
- The states of the output buffers, which are determined indirectly from the number of events which have been transferred to the output FIFO.

The simplified equations describing this state machine are shown in Figure 1. A more detailed description of the actual implementation is shown in Figure 2.

Figure 1: Simplified conditions for the DMA scheduling.

```
Nevt(DPRAM) > 0 && IN == Free      → Trigger Input DMA
                                   → IN = Busy

Input DMAC                         → Nevt(DPRAM) -= 1
                                   → IN = Ready

IN == Ready && OUT == Free         → Begin event processing
                                   → OUT = Busy

End event processing               → Trigger Output DMA
                                   → IN = Free

Output DMAC                        → OUT = Free
                                   → Nevt(Out) += 1
```

Figure 2: Implementation of the DMA scheduling logic in the main event loop.

```
loop:

/* --- Go ahead with the next output DMA if needed AND if DMA2 available */
if (Output_DMAC && nevt_out_next < nevt_out_rdy){
       schedule output DMA;
       nevt_out_next++;
}

/* --- Polling on the dual-port RAM counter */
nevt_input = RRAM[COUNTERS] & 0xffff;

/* --- Busy logic */
if (nevt_input - nevt_input_next > NEVT_BUSY_ON)
       LVL2_BUSY_ON;

else if (nevt_input - nevt_input_next == NEVT_BUSY_OFF)
       LVL2_BUSY_OFF;

if (nevt_input - nevt_input_next != 0 && IN1 == FREE){
       Schedule Input DMA to Buffer1;
       IN1 = BUSY;
       nevt_input_next++;
}
else if (IN1 == BUSY && Input_DMAC)
       IN1 = READY;

if (nevt_input - nevt_input_next != 0 && IN2 == FREE){
       schedule input DMA to Buffer2;
       IN2 = BUSY;
       nevt_input_next++;
}

/* --- Ready input AND there is a free output buffer.
       Note that the last output buffer is never used directly,
       since the last DMA transfer may still be in progress */

if (IN1 == READY && nevt_out_rdy - nevt_out_next < NBUFOUT-1){

       /* Process event */
       process_event(IN1, OUT(nevt_out_rdy));

       /* One more event waiting to be serviced by output DMA */
       nevt_out_rdy++;

       /* Input buffer becomes free */
       IN1 = FREE;

       Swap buffers 1 and 2;

}

goto loop;
```

Note that the LVL2 BUSY is only controlled by the number of events waiting in the dual-port memory. However, if for some reason the output FIFO becomes almost full (e.g. due to the ROB not ready to receive), then the output DMA cannot proceed, which may result in all four DSP output buffers to be busy, which in turn will prevent the DSP to process any events, and so the LVL2 BUSY will be turned on at some point, to avoid the dual-port RAM buffer to fill up.

The configuration of the DMA transfers is explained in the DSP6202 board *hardware reference*.

In addition, the completion of the output DMA (referred to as Output_DMAC) is detected by testing the DSP interrupt flag register as described in Section 8.3 of the *TMS320C6000 CPU and Instruction Set Reference Guide*.

The output DMA is performed by DMA channel 2, which has been configured to interrupt the CPU at the end of each event transfer. The interrupt is disabled: it does not trigger an interrupt processing, has no associated interrupt service packet. The interrupt is processed manually by polling the IFR register. Note that the DMA channel 2 interrupt, DMA_INT2, by default maps to CPU INT11 as explained in Chapter 13 of the *Peripherals Reference*. In turn, CPU INT11 is tested using Bit 11 of the IFR. INT11 is cleared (acknowledged) before starting a new DMA transfer, by writing to the ICR.

The output DMA source address and word count are configured for each event. The destination address is set to the XCE3 address `0x70000000` at initialization time. The output DMA secondary control register configuration is shown in Table 1, and the primary control register configuration is shown in Table 2. Please refer to Section 5.2 of the *Peripherals Reference*.

Table 1: Output DMA Secondary Control Register Field description. All other fields are set to 0.

Field	Description
RSPOL = 1	Polarity of the EXT_INT6.
FSIG = 1	Select level-triggered frame synchronization.
DMAC EN = 100b	DMAC reflects FRAME COND.
FRAME IE = 1	DMA channel interrupt enabled at the end of the event transfer.

Table 2: Output DMA Primary Control Register Field Description. All other fields are set to 0.

Field	Description
DST RELOAD = 00b	No need to reload the destination register for FIFO writing.
SRC RELOAD = 00b	The source address is loaded manually.
FS = 1	Frame synchronization is enabled.
TCINT = 1	Transfer controller interrupt is enabled.
WSYNC = 00000b	WSYNC not applicable when FS = 1.

Table 2: Output DMA Primary Control Register Field Description. All other fields are set to 0.

Field	Description
RSYNC = 00110b	Frame synchronization on EXT_INT6.
DST DIR = 00b	The destination address is fixed for FIFO writing.
SRC DIR = 01b	The source address is incremented.
START = 01b	Start the transfer without autoinitialization.

Currently the completion of the input DMA is detected by checking the STATUS field of the corresponding DMA primary control register.

1.3 Initialization

On the DSP6202 board, upon reset, the DSP will transfer 64 kBytes of data from the dual-port RAM to its internal data memory, then starts executing the code at address 0.

In the current implementation, the code at address 0 is a branch into the `c_int00` routine, which is the run-time initialization of the C environment. Then the user `main()` program is called. The `main()` program calls `periph_init()` to configure the DSP peripherals (EMIF, serial ports, XBUS) to conform to the actual hardware. The actual settings are shown in Table 3.

Table 3: Peripherals configuration in periph_init()

Register	Value	Description
EMIF CE1 Space Control	0xfff1 0220	Asynchronous memory timings Read Setup=1, Strobe=2, Hold=0
Expansion Bus Global Control (XBGC)	0x0000 6000	Clock = CPU / 4
McBSP0 Receive Control (RCR) McBSP0 Transmit Control (XCR)	0x0001 00A0	
McBSP0 Sample Rate Generator (SRGR)	0x2020 0008	
McBSP0 Pin Control (PCR)	0x0000 0A02	
McBSP0 Serial Port Control (SPCR)	0x00C1 0001	
McBSP1 Pin Control (PCR)	0x3800	
Interrupt Multiplexer High	0x0820 2d4d	RINT0 replaces SD_INT as INT10 for handling serial data from VME.

The main() program then calls `memcpy()` to transfer the calibration constants from the second half of the dual-port memory, to the data memory pointed to by `cc_table`.

Finally, the event loop is entered.

testSysCall Subroutine

Purpose

To intercept and process messages sent to the DSP serial port.

Syntax

```
void testSysCall()
```

Description

In between event processing, the DSP main program checks for any serial data sent via VME by calling the **testSysCall** subroutine. Currently the following values are recognized:

0	Write the contents of the energy histograms to the FPGA histogram FIFO.
1	Write the contents of the time histograms to the FPGA histogram FIFO.
2	Enter PD3 power-down mode.

If no serial data is present, the **testSysCall** subroutine returns after 8 cycles.

If the energy or time histograms are requested, their addresses are taken from the global symbols `_ehist` and `_thist` respectively; the size is hard-coded to 4096 words. The DMA transfer is scheduled only if the DMA3 channel is not busy with a previous request; otherwise the subroutine has no effect. In either case, the subroutine returns after 14 cycles.

If the power-down mode is entered, the subroutine never returns. A complete reset is then needed to reboot the DSP.

Developers are encouraged to extend the functionality of this service subroutine, according to their needs.

Implementation details

The **testSysCall** subroutine checks for serial data by polling Bit 10 of the IFR. This interrupt was mapped to RINT0 and is associated with one 32-bit serial word available for reading in the McBSP0 receive register. Once detected, the interrupt flag is cleared manually by writing to the ICR. The scheduled assembler code is shown in Table 4.

Table 4: testSysCall.asm scheduled assembler flow

.S2	.S1	
MVC IFR, B0	MVKL DRR_0, A3	
EXTU B0, 21, 31, B0	MVKH DRR_0, A3	
[!B0] B B3	MVKL DMA_3, A3	[B0] LDW *A3, B1
[B0] MVK 0x400, B0	MVKH DMA_3, A3	
		[B0] LDW *A3, A1
MVKL _ehist, B4	MVKL 0x00180008, A4	
MVKH _ehist, B4	MVKH 0x00180008, A4	
[B0] MVC B0, ICR		CMPGT B1,1, A2[a]
[!A2] B B3[b]		
[B1] MVKL _thist, B4	[!A2] EXTU A1, 28, 30, A2[c]	
[B1] MVKH _thist, B4	MVK 256, A5	[!A2] STW A4, *+A3[2][d]
MVKL 0x04015011, B5	MVKLH 16, A5	[!A2] STW B4,*+A3[4][e]
MVKH 0x04015011, B5		[!A2] STW A5,*+A3[8][f]
		[!A2] STW B5, *A3[g]
MVK 0x7000, B4		
MVC B4, CSR[h]		
IDLE		

a. Here A2=1 means **not** a DMA request.
b. Final return, after a DMA request.
c. Check if DMA3 is busy.
d. Write secondary control register
e. Write the source address
f. Write frame and word count
g. Trigger the transfer
h. Power-down.

process_event Subroutine (evtproc26.asm implementation)

Purpose

To compute the energy, time, and pulse quality for all channels, by using the optimal filtering.

Syntax

```
void process_event(data_in, data_out)
const void *data_in;
void *data_out;
```

Description

The **process_event** routine is called by the main event loop whenever there is a full input buffer `data_in` and a free output buffer `data_out`. The current implementation is based upon the *loop26* optimal filtering code. In addition to its arguments, this routine also uses the `_div_table` and `_cc_table` global symbols, which refer to the division table and respectively the calibration constants table, in the format needed by *loop26*.

In particular, the *loop26* program imposes the following constraints concerning the memory organization. The calibration constants shall be aligned on a 0x4000-word boundary. For performance reasons (access to the data memory), the input data, output data, and the division table shall reside in memory block 1. The calibration constants shall reside in memory block 0.

These constraints are fulfilled by the main program in conjunction with the linker, by using the following directives and declarations:

```
#pragma DATA_SECTION(".febdata")
unsigned int data_in1[512];

#pragma DATA_SECTION(".febdata")
unsigned int data_in2[512];

#pragma DATA_SECTION(".febdata")
unsigned int data_out[NBUFOUT][300];

#pragma DATA_ALIGN(0x4000)
unsigned int cc_table[4096];
```

process_event Subroutine (evtproc26_mask.asm implementation)

Purpose

To compute the energy, time, and pulse quality for all channels, by using the optimal filtering; to flag the channels above a given energy threshold.

Syntax

```
double process_event(data_in, data_out, threshold)
const void *data_in;
void *data_out;
float threshold;
```

Description

This version of the **process_event** routine combines the evtproc26.asm and emask.asm functionality. It is more efficient than the two parts used separately, because it eliminates one subroutine call and allows better pipelining. Please refer to the evtproc26.asm description and to the emask subroutine for more information.

emask Subroutine (emask.asm implementation)

Purpose

To flag the channels above some energy threshold.

Syntax

```
double emask(data, threshold)
const float *data;
float threshold;
```

Description

The *loop26* implementation of the optimal filtering computes the time and pulse shape quality for all channels. The **emask** subroutine is designed to flag the channels above the specified energy `threshold`. The `data` parameter points to the beginning of the energy data block, which must contain (at least) 64 floating-point data words. The **emask** subroutine returns a 64-bit channel mask in big-endian order: the MSB refers to channel 0, while the LSB refers to channel 63.

The bit ordering is chosen to allow further processing by using the *bit search forward* (left to right) opcodes available on many processors: LMBD for the ‘C6000 architecture, CNTLZ on PowerPC, BSR on Intel. See also the LEADZ Fortran90 intrinsic.

Implementation details

The emask.asm implementation needs 32 cycles to process 64 channels. Two loads are performed every cycle, so the corresponding energy words must reside in different memory banks. This is done by processing channels 31→0 in parallel with channels 32→63. The emask.asm does not modify the A10..A15 or B10..B15 registers.

filterHisto Subroutine (filterHisto.asm implementation)

Purpose

To remove the time/chi2 information for specified channels; to fill the energy and time histograms; to flag channels for monitoring purposes.

Syntax

```
int filterHisto(data, mask, ecut2, ecut3, chi2cut, Offset)
unsigned int *data;
double mask;
float ecut2;
float ecut3;
int chi2cut;
float Offset;
```

Description

The data parameter points to the beginning of the energy data block, immediately followed by the time/chi2 data block. The **filterHisto** subroutine modifies the time/chi2 data block by preserving only the time/chi2 information for those channels which are flagged by `mask`. The subroutine returns the number of channels with time/chi2 information.

In parallel, for the channels which are flagged, the **filterHisto** subroutine fills the energy and time histograms. The base address of the energy histograms is taken from the `_ehist` global symbol, and the base address of the time histograms is taken from `_thist`.

For every channel, there are three energy histograms and three time histograms, one for each gain. All bins are 32-bits, with saturation at 2^{32}–1 in case of overflow. All histograms have overflow and underflow bins. The number of bins must be a power of 2, and may be defined separately for the energy and for the time histograms. Currently both types of histograms have 16 bins.

The energy histograms are logarithmic. The `Offset` parameter contains the biasing exponent used to fill the energy histograms, and must be a power of 2, i.e. its fraction (bits 0 to 22 of the IEEE floating-point format) must be zero. The quantity histogrammed is $\log_2$(E/`Offset`).

The time histograms are linear, centered around zero. They are filled only for those channels flagged in the input mask, **and** satisfying E > `ecut2`. The range of the time histograms is hard-coded in filterHisto.asm. Please refer to Table 5 in order to modify this parameter.

In addition, the **filterHisto** subroutine fills a new channel mask for those channels satisfying E > `ecut3` or χ^2 > `chi2cut`. In the current C linkage, this second mask is not returned to the caller.

Implementation details

The time histograms shall reside in memory block 1. The energy histograms shall reside in memory block 0. The filterHisto.asm implementation needs 6 CPU cycles for each channel flagged in the input mask, plus a small overhead.

Table 5: filterHisto.asm scheduled assembler flow

	.D2 (I/O, E)	.S2 (E)	.L2 (E)	.D1 (t)	.S1 (t)	.L1 (t, mask')	.M
	MV B4, B0 (m0)	*MV A4, B4 (eIn)*	*MV B5, B1 (m1)*		*ADDK 256, A4*	*MV SF, A0*	
Load 0	*PUSH A13*	*MV A4, B5 (xIn)*	LMBD 1, m0, n	*PUSH B3*	[!m0] B **epilog0**		
	LDW *++xIn[n], x	SHL m0, n, bit	ZERO N	*PUSH B14*			
	LDW *++eIn[n], e	CLR bit, 31, 31, m0		*PUSH B13*	*MVK 48, C48*		[m0] MPY 4, n, N
	PUSH A12	*MVK _ehist, hpe*		*PUSH B12*	*MVKLH 48, C48*		
	PUSH A11	*MVKH _ehist, hpe*	XOR bit, m0, bit	*PUSH B11*	*MVK _thist, hpt*	ZERO hptWR	
	PUSH A10	SHRU bit, n, bit	ZERO hpeWR	*PUSH B10*	*MVKH _thist, hpt*	ZERO mask	
Load i Save i–1 Load histo i–1 Save histo j–2	**STW** x, *xOut++	[m0] B **here**	LMBD 1, m0, n		SHL x, 18, g:t0	CMPGT x, xcut, k	MPY C48, N, jmpt
	LDW *++xIn[n], x	SHL m0, n, m0	SUB e, Offset, e0		SSHL t0, 10, t0	AND 3, g, g	MPYHL C48, N, jmpe
	LDW *++eIn[n], e	SSHL e0, 5, e0	MV g, g'	ADD hpt, jmpt, hptRD	SHRU g:t0, 28, g:t0	[!k] CMPGT e, thr4, k	
	[m0] ADDAW N, n, N	SHRU g':e0, 28, g':e0	ADD hpe, jmpe, hpeRD	**LDW** *++hptRD[t0], binT		SADD 1, binT, binT	
	LDW *++hpeRD[e0], binE	CLR m0, 31, 31, m0	SADD 1, binE, binE	[hptWR] **STW** binT, *hptWR		CMPGT e, thr_th, hptWR	
	[hpeWR] **STW** binE, *hpeWR	SHRU bit, n, bit	MV hpeRD, hpeWR	[hptWR] MV hptRD, hptWR		[k] ADD bit, mask, mask	
epilog0: Save last histo Load 32	SUB xIn, N, xIn	SUB eIn, N, eIn	LMBD 1, m1, n		[!m1] B **epilog1**		
	LDW *++xIn[n], x	SHL m1, n, bit					
	LDW *++eIn[n], e	MVK 128, N				SADD 1, binT, binT	
	[m1] ADDAW N, n, N	CLR bit, 31, 31, m1	SADD 1, binE, binE	[hptWR] **STW** binT, *hptWR			
	[hpeWR] **STW** binE, *hpeWR	MV mask, m0	XOR bit, m1, bit			ZERO hptWR	
		SHRU bit, n, bit	ZERO hpeWR			ZERO mask	
Load i Save i–1 Load histo i–1 Save histo j–2	**STW** x, *xOut++	[m1] B **here**	LMBD 1, m1, n		SHL x, 18, g:t0	CMPGT x, xcut, k	MPY C48, N, jmpt
	LDW *++xIn[n], x	SHL m1, n, m1	SUB e, Offset, e0		SSHL t0, 10, t0	AND 3, g, g	MPYHL C48, N, jmpe
	LDW *++eIn[n], e	SSHL e0, 5, e0	MV g, g'	ADD hpt, jmpt, hptRD	SHRU g:t0, 28, g:t0	[!k] CMPGT e, thr4, k	
	[m1] ADDAW N, n, N	SHRU g':e0, 28, g':e0	ADD hpe, jmpe, hpeRD	**LDW** *++hptRD[t0], binT		SADD 1, binT, binT	
	LDW *++hpeRD[e0], binE	CLR m1, 31, 31, m1	SADD 1, binE, binE	[hptWR] **STW** binT, *hptWR		CMPGT e, thr_th, hptWR	
	[hpeWR] **STW** binE, *hpeWR	SHRU bit, n, bit	MV hpeRD, hpeWR	[hptWR] MV hptRD, hptWR		[k] ADD bit, mask, mask	
epilog1: Save last histo Restore registers Return to caller	*Restore B3*						
	Restore B14					*MV SF, A0*	
	Restore B13			*Restore A13*			
	Restore B12		SADD 1, binE, binE	*Restore A12*		SADD 1, binT, binT	
	[hpeWR] **STW** binE, *hpeWR			[hptWR] **STW** binT, *hptWR			
	Restore B11	*B B3*		*Restore A11*			
	Restore B10			*Restore A10*	*MV mask, A5*	*MV m0, A4*	
		NOP 4					

Table 6: filterHisto.asm register allocation

Reg	A side	B side
0	e = energy loaded	m0 = input channel mask for channels 0..31 **(from 2nd input argument)**
1	x = time/chi2 loaded, k = condition to dump samples for this channel	m1 = input channel mask for channels 32..63 **(from 2nd input argument)**
2	hptWR = condition to fill time histogram and write pointer to histogram contents	hpeWR = write pointer to energy histogram (or 0)
3	mask = output mask for dumping samples	bit = running bitmask to build the output mask, initially 0x8000 0000
4	xOut = pointer to filtered output time/chi2 (initially xIn)	eIN = read pointer to energy block **(from 1st input argument)**
5	hpt = base pointer for time histograms (initialized from global symbol)	xIN = read pointer to time/chi2 block (initially eIN + 64 words)
6	thr_th = energy threshold to fill time histogram (**input**, **3rd argument**, floating-point constant)	thr4 (ecut4) = energy cut to dump samples (**input**, **4th argument**, floating-point constant)
7	Nbins (C48) = number of bins per channel in energy histogram (16..31) and in time histogram (0..15)	n = number of channels to skip according to the input mask
8	xcut (chi2cut) = chi2 cut to dump samples (**input**, **5th argument**, integer constant)	Offset = **constant** containing the biasing exponent to be subtracted **(6th argument?)**
9	jmpe, jmpt = offset to e/t histograms	N = current channel being processed, or Sum(n)
10	binT = time histogram channel content	binE = energy histogram channel
11	hptRD = read pointer to time histogram	hpeRD = read pointer to energy histogram
12	t0 = time extracted from time/chi2	e0 = log2(energy) extracted from e
13	g2 = gain extracted from time/chi2	g1 = copy of g2
14		hpe = base pointer for energy histograms (initialized from global symbol)
15		SP

2 The DSP6202 Host Software

A set of C functions are available for communicating with the DSP6202 board from a VME CPU. These functions allow to configure the input FPGA; download the DSP code and boot the DSP; send and receive data by using the DSP serial port etc.

Currently these functions are grouped in the libti.a library. Communication with VME is done via 5 well-defined low-level functions, which may be modified to comply with the actual CPU and VME environment (Figure 3).

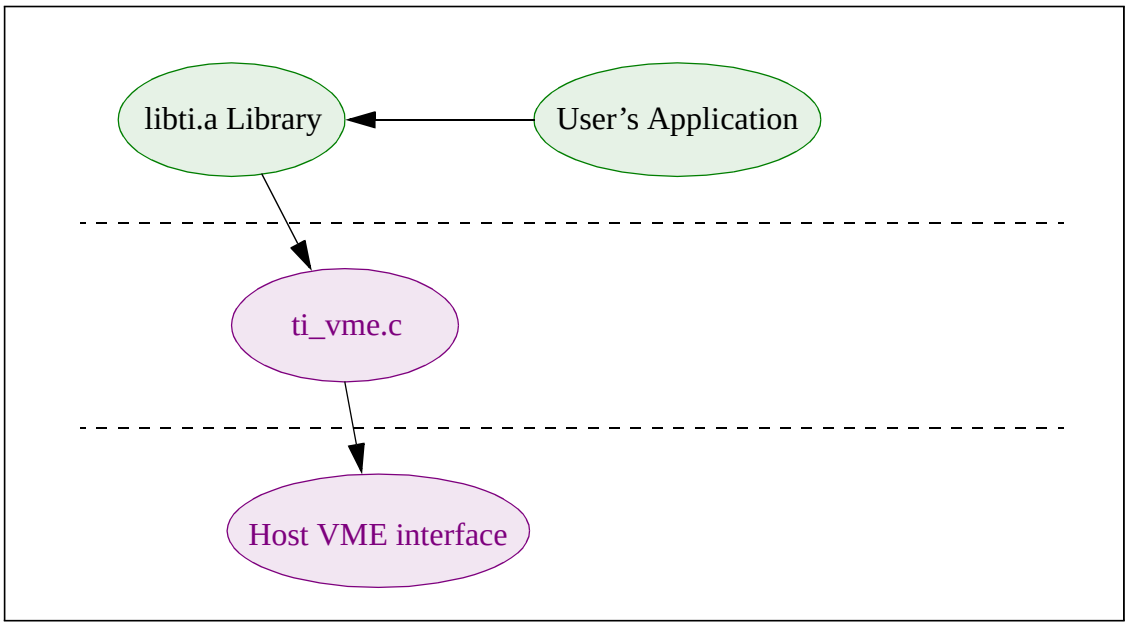

Figure 3: Structure of the DSP6202 Host Software

ti_boot Subroutine

Purpose

Full initialization of the DSP6202 board.

Syntax

```
#include “ti_dsp.h”
ti_boot(dsp, file1, file2)
ti_dsp_p dsp;
const char *file1, *file2;
```

Description

The **ti_boot** subroutine configures the DSP6202 board input FPGA, then downloads the contents of the two files to the dual-port memory.

The `dsp` input argument is a pointer to a `ti_dsp_t` data structure, with the `rodid` and `puid` fields properly initiaized. The `puid` field is the slot number of the PU to configure.

The binary file ALTERA.dat is used to configure the input FPGA. This file can be obtained directly from an ALTERA **.ttf** file by using the **ttf2bin** program.

The `file1` and `file2` input arguments specifiy two **.x0** files in Tektronix format. `file1` contains the DSP object file produced by the TI Compiler and Linker, followed by the TI Hex Conversion Utility (see Chapter 10 and Section 10.9.5 of the *TMS320 C6000 Assembly Language Tools User’s Guide*). `file2` contains the calibration constants for optimal filtering. Either `file1` or `file2` may be NULL pointers, in which case nothing is loaded.

Once the DSP object file has been downloaded, the **ti_boot** subroutine releases the DSP $\overline{\text{RESET}}$. The DSP should respond by sending a serial data word. The **ti_boot** subroutine completes the initialization by clearing the output FPGA buffer and event counter, and sending the INT7 interrupt to the DSP which then enters the main event loop.

ti_altera_cfg Subroutine

Purpose

To configure the DSP6202 input FPGA

Syntax

```
#include “ti_dsp.h”
ti_altera_cfg(dsp)
ti_dsp_p dsp;
```

Description

The **ti_altera_cfg** subroutine is used internally by **ti_boot** to configure the input FPGA. It uses the binary file ALTERA.dat, which can be obtained from an ALTERA **.ttf** file by using the **ttf2bin** program.

ti_dpcpy Subroutine

Purpose

Data transfer from the host CPU to the DSP6202 input FPGA or dual-port memory.

Syntax

```
#include “ti_dsp.h”
ti_dpcpy(dsp, s1, s2, wc)
ti_dsp_p dsp;
unsigned int s1;
unsigned int *s2;
int wc;
```

Description

The **ti_dpcpy** subroutine copies 32-bit words from memory area `s2` into the input FPGA starting at offset `s1` words, stopping after `wc` words have been copied. This routine is also used to send data to the dual-port RAM. The input FPGA DP_OFFLINE flag is set high by this routine. Clearing the DP_OFFLINE flag is caller’s responsibility.

The **ti_dpcpy** subroutine relies on the output FPGA to increment the target address and serialize each data and address words which are sent to the input FPGA using the DP_DMS, DP_DLS, DP_ADD, and DP_FRM signals. The input FPGA decodes the address and uses the data to either configure its own registers and/or memory, or write the data to the dual-port memory.

In the present configuration of the input FPGA, the memory map is shown in Table 7.

Table 7: Input FPGA memory map

Address Range (Hex Words)	Size Words	Description
0000–7FFF	32K	Dual-port RAM.
8000 (ADD_MODE in ti_dsp.h)	1	Input FPGA Mode Register.
9000 (ADD_NSAMP in ti_dsp.h)	1	Input FPGA Register (Number of samples).
A000 (ADD_NGAIN in ti_dsp.h)	1	Input FPGA Register (Number of gains).
B000–B1FF (ADD_MAP in ti.dsp.h)	512	Input FPGA LPM RAM. Used to rearrange the FEB data words.
C000 (ADD_SHFT in ti_dsp.h)	1	Input FPGA Register. Left shift of FEB ADC words.

ti_dpset Subroutine

Purpose

To initialize the DSP6202 input FPGA or dual-port memory

Syntax

```
#include “ti_dsp.h”
ti_dpset(dsp, s, c, wc)
ti_dsp_p dsp;
unsigned int s;
unsigned int c;
int wc;
```

Description

The **ti_dpset** subroutine sets the first `wc` words in the input FPGA area starting at offset `s` to the value of `c`. It may be used to initialize the input FPGA registers or the dual-port memory with a given value. The input FPGA DP_OFFLINE flag is set high by this routine. Clearing the DP_OFFLINE flag is caller’s responsibility.

The current input FPGA memory map is shown in Table 7.

ti_dpsend_xfile Subroutine

Purpose

To load a DSP object file in Tektronix format

Syntax

```
#include “ti_dsp.h”
ti_dpsend_xfile(dsp, file)
ti_dsp_p dsp;
const char *file;
```

Description

The `file` argument is the name of a **.x0** file in Tektronix format, to be loaded to the DSP via the dual-port memory. The input FPGA DP_OFFLINE flag is set high by this routine. Clearing the DP_OFFLINE flag is caller’s responsibility.

It is caller’s responsibility to maintain the DSP $\overline{\text{RESET}}$ active during the transfer. Usually the **ti_dpsend_xfile** subroutine is called by **ti_boot**.

ti_status and ti_pstatus Subroutines

Purpose

To read and print the DSP6202 output FPGA status register.

Syntax

```
#include “ti_dsp.h”
(void) ti_pstatus (ist)
int ti_status (dsp)
ti_dsp_p dsp;
unsigned int ist;
```

Description

The **ti_pstatus** subroutine produces a message on the standard error output, describing the contents of the DSP6202 status register which is taken from the `ist` argument.

The **ti_status** subroutine reads the DSP6202 status register via VME, and produces a message on the standard error output, describing the contents of the status register. The `dsp` argument is a pointer to a `ti_dsp_t` data structure properly initialized.

ti_interrupt Subroutine

Purpose

To send the INT7 external interrupt to the DSP.

Syntax

```
#include “ti_dsp.h”
ti_interrupt(dsp);
ti_dsp_t dsp;
```

Description

The **ti_interrupt** subroutine issues two VME write cycles to the DSP6202 control word, to assert and then deassert the DSP EXT_INT7 interrupt signal.

ti_syscall Subroutine

Syntax

```
#include “ti_dsp.h”
ti_syscall(dsp, c)
ti_dsp_t dsp;
unsigned int c;
```

Description

The 32-bit value `c` is sent to the DSP McBSP0 serial port.

fpga_map Subroutine

Purpose

Initialization of the input FPGA LPM RAM used for FEB data rearrangement.

Syntax

```
fpga_map_(nsamp, ngains, map)
int *nsamp;
int *ngains;
unsigned int map[512];
```

Description

The **fpga_map** user routine is called by **ti_boot** once, at initialization time. It must fill the `map` array with the address offsets, in the dual-port RAM event block, where the FEB data are to be written. The **ti_boot** subroutine then copies the contents of `map` to the input FPGA LPM RAM. The `*nsamp` and `*ngains` arguments are inputs to the **fpga_map** subroutine.

The current implementation of **fpga_map** is written in Fortran. The order of the FEB data building blocks after rearrangement, is specified using the following statement:

```
C=======================================
C     Put your code below
C=======================================
      WRITE(LUN,'(I6)')
     $     GAIN(0,0), ((ADC1(i,isamp),isamp=0,NSAMP-1),i=0,7),
     $     GAIN(1,0), ((ADC1(i,isamp),isamp=0,NSAMP-1),i=8,15),
     $     GAIN(2,0), ((ADC1(i,isamp),isamp=0,NSAMP-1),i=16,23),
     $     GAIN(3,0), ((ADC1(i,isamp),isamp=0,NSAMP-1),i=24,31),
     $     (ADDR(0,isamp),isamp=0,NSAMP-1),
     $     CTL1(0),CTL2(0),CTL3(0),ERR_FLAG
```

vme_dsp_write and vme_dsp_read Subroutines

Purpose

To interface the ti_ subroutines with the host VME

Syntax

```
#include “ti_dsp.h”
void vme_dsp_write(dsp, addr, data)
int vme_dsp_read(dsp, addr)
ti_dsp_t dsp;
ti_addr_t addr;
unsigned int data;
```

Description

The **vme_dsp_write** subroutine writes one 32-bit word specified by `data` to the PU specified by `dsp` at the PU address `addr`.

The **vme_dsp_read** subroutine reads one 32-bit word from the PU specified by `dsp` at the PU address `addr`.

These subroutines know about the PU address to VME address mapping as implemented by the motherboard.

vme_dsp_writeblk and vme_dsp_readblk Subroutines

Purpose

To interface the ti_ subroutines with the host VME

Syntax

```
#include “ti_dsp.h”
void vme_dsp_writeblk(dsp, addr, nw, data)
void vme_dsp_readblk (dsp, addr, nw, data)
ti_dsp_t dsp;
ti_addr_t addr;
int size;
unsigned int *data;
```

Description

The **vme_dsp_writeblk** subroutine copies `nw` 32-bit words from the `data` array, via VME, to the PU specified by `dsp` at the PU address specified by `addr`. The destination address is left unchanged (FIFO mode).

The **vme_dsp_readblk** subroutine copies `nw` 32-bit words from the PU specified by `dsp` and from the PU address specified by `addr`, via VME, to the `data` array. The source address is left unchanged (FIFO mode).

These subroutines know about the PU address to VME address mapping as implemented by the motherboard.

vme_check Subroutine

Purpose

To test if any errors have occured during the VME transfers

Syntax

```
#include “ti_dsp.h”
void vme_check()
```

Description

If an error occured during the VME transfers, the **vme_check** subroutine prints the details of the error to the standard error output, and then clears the error.