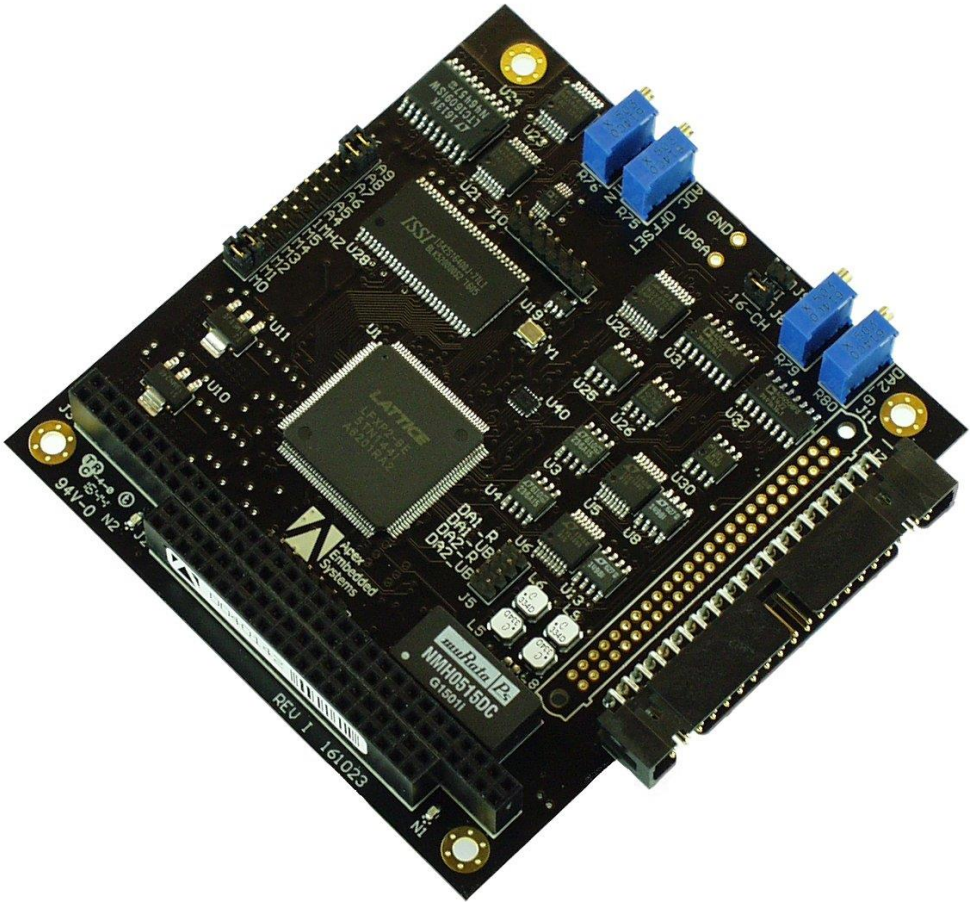
Current Monitoring Using STX104 for Software Power Debugging

By: Peter S Procek

Contents



1. Introduction

The capabilities of the STX104 provide the perfect solution for quick power analysis of various systems. The high 200K samples/sec sampling rate and low noise offers high resolution in acquired data and the on-board one million sample FIFO allows for easy data management without worry of overflows. When developing for power optimized embedded systems, it is useful to observe the power overhead contributed by user written pieces of software or firmware. One can more quickly root out causes of power bottlenecks and optimize their system through physical real-time current and voltage monitoring.

This application note discusses the usage of the STX104 in a typical PC/104 stack as a power monitor for software running on our target, a DIGI CCi.MX6UL Starter board. In our case, we were interested in optimizing I/O heavy workloads that read from and wrote to the microSD on the target. Our experiment involved many on-the-fly changes causing us to iteratively collect multiple power traces for comparison.

2. Hardware Setup

To monitor instantaneous power draw, we need to know voltage and current draw of the component we wish to analyze. In our case, we are monitoring V_{sys} of the CCi.MX6UL. Because the device is low power and we are only utilizing a few hardware components on the target, we expect the current draw fluctuation resolution to be relatively small. We analytically verify this by using an oscilloscope and monitoring V_{sys} directly. We therefor utilize a current sense amplifier to give our signal an appropriate gain to achieve a desirable resolution.

a. STX104 Configuration

We assume the STX104 has already been calibrated. We are going to collect data single-ended in a +/-5v range. We install a jumper across J8 for single-ended mode, and we leave J9 uninstalled for the +/-5v range. We leave J6 with its default configurations. The configuration is shown in figure 1 below.

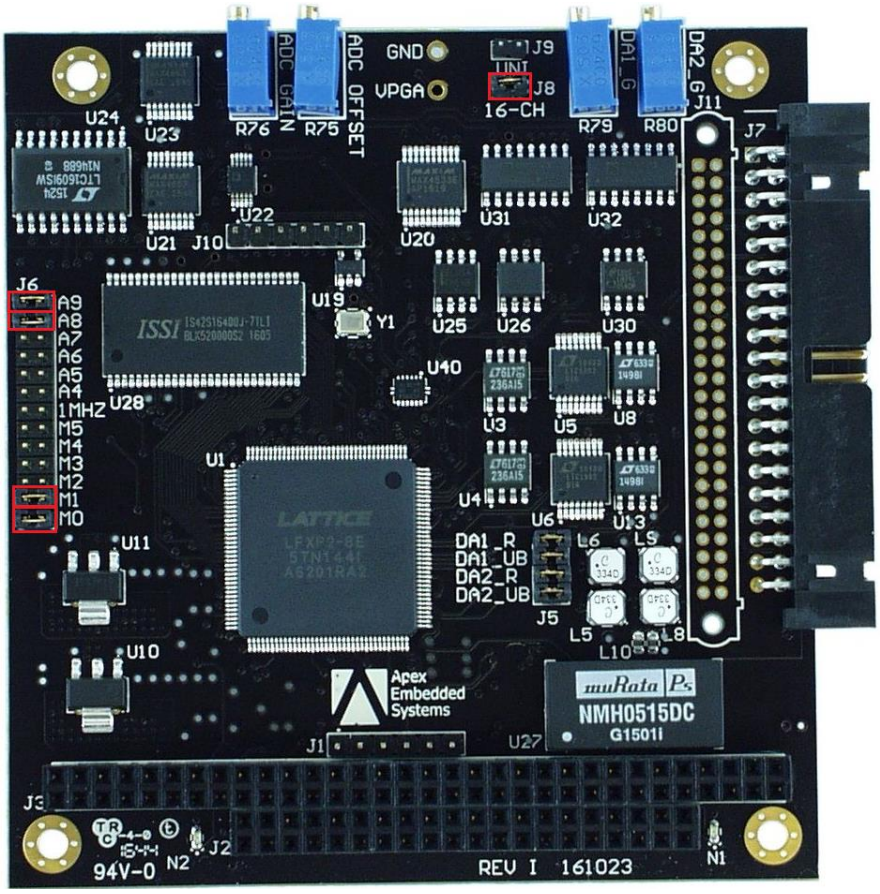


Figure 1. The STX104 configured as single ended, +/-5v in DAS1602 mode with base address 0x300

b. Target Configuration

We connect the jumper that exposes V_{sys} on our target board to an LT6105 current sense amplifier. We present the amplifier as a black box whose output is what the STX104 will be monitoring. We use a sense resistor across the jumper small enough to get a valid output that provides us with reasonable resolution while avoiding dropping too much voltage to where the board could not power on. We use a sense resistor valued at .025Ω, with the circuit configured to provide a gain of about 20. The output of this amplifier feeds into an input channel of the STX104. We also route a spare GPIO configured as an output from the target to the TRIG signal of the STX104 for software triggered collections. The configuration is noted in figure 2 below.

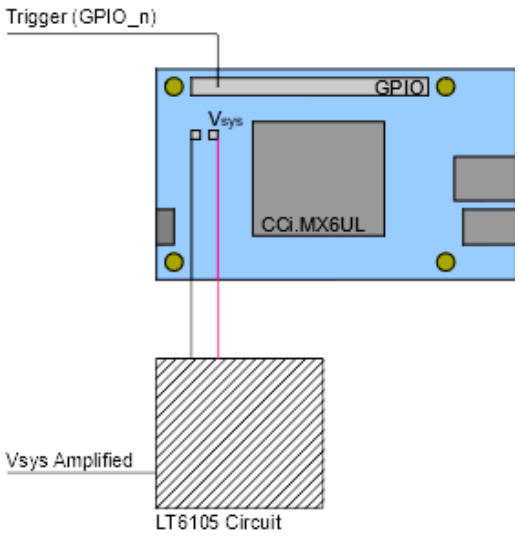


Figure 2. A block level diagram of the target configuration

***NOTE: Our CCi.MX6UL board runs a custom Linux build built with the Yocto build system. Because we wish to monitor the power overhead of the execution of specific workloads, we want to ensure that we capture the entire execution from beginning to end. Because of the abstractions in the target's Operating System and hardware level stack, we are not guaranteed real-time triggering. Thus, for any target running Linux, it is wise to route the Trigger signal back into the target itself via some other GPIO configured as an input. This can help provide relative execution points, which is elaborated more on below.*

c. Experiment Setup

We connect our target to the STX104 via a cable break-out connector pictured in figure 3. We connect our target's trigger to pin 12 (*TRIG*). We connect the output of our current sense amplifier to an input channel on the STX104, which in our case will be pin 36 (*CH0*). Lastly, we ground the target to the STX104 via pin 37 (*AGND*). The STX104 connect summary is pictured below in figure 4 for reference.

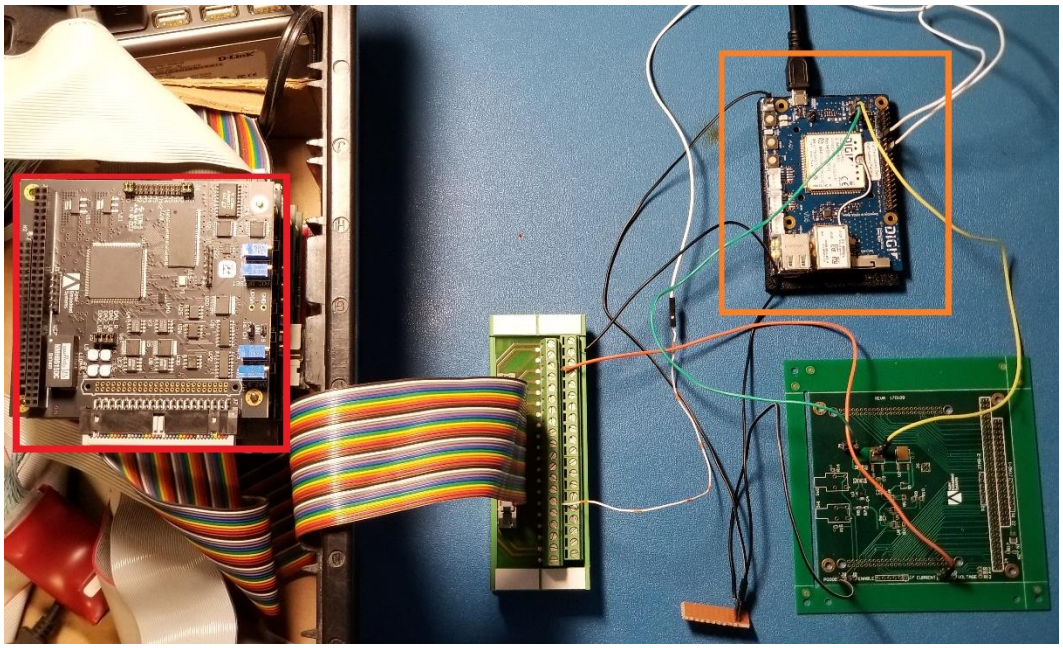


Figure 3. The target (orange) routes its trigger to the STX104 via screw terminal breakout. It also routes to the current sense circuit whose output is also routed to the screw terminal.

***NOTE: In our experiment, we only monitor current and assume a constant input voltage for simplicity as our target is powered via USB. For higher accuracy, route an additional wire from the input side of the voltage rail to an additional input channel on the STX104 and monitor input voltage. Because the STX104 is not a simultaneous sample card, additional post-processing and integration will be required to get an accurate power estimate.*

STX104 I/O Connector J7

Output	+5V POWER	1	■ ●	2	CT_OUT2	Output
Output	CT_OUT0	3	● ●	4	CT_CLK0	Input
Output	DOUT3	5	● ●	6	DOUT2	Output
Output	DOUT1	7	● ●	8	DOUT0	Output
Input	DIN3	9	● ●	10	CT_GATE0 / DIN2	Input
Input	DIN1	11	● ●	12	TRIG / DIN0	Input
Input	DGND	13	● ●	14	SSH	Output
Output	-5V	15	● ●	16	DAC_OUT_2	Output
Output	DAC_OUT_1	17	● ●	18	AGND	InOut
	No Connection (NC)	19	● ●	20	AGND	InOut
Input	CH7 LOW / CH15	21	● ●	22	CH7 / CH7 HIGH	Input
Input	CH6 LOW / CH14	23	● ●	24	CH6 / CH6 HIGH	Input
Input	CH5 LOW / CH13	25	● ●	26	CH5 / CH5 HIGH	Input
Input	CH4 LOW / CH12	27	● ●	28	CH4 / CH4 HIGH	Input
Input	CH3 LOW / CH11	29	● ●	30	CH3 / CH3 HIGH	Input
Input	CH2 LOW / CH10	31	● ●	32	CH2 / CH2 HIGH	Input
Input	CH1 LOW / CH9	33	● ●	34	CH1 / CH1 HIGH	Input
Input	CH0 LOW / CH8	35	● ●	36	CH0 / CH0 HIGH	Input
InOut	AGND	37	● ●	38	No Connection (NC)	
	No Connection (NC)	39	● ●	40	No Connection (NC)	

Figure 4. STX104 Connector Summary Reference

3. Software Setup

a. STX104 Software

In our experiment, we must initialize the STX104 properly in software to sample only *CH0* at 10Khz with burst collection mode and +/-10v gain. We want to begin our collections when triggered by *TRIG/DIN0*. This code is set-up to poll when there is enough data in a FIFO and dump that data into a file when available. After an event causes collection to stop, post processing occurs dumping either raw values or real converted voltage values into a csv file. The code is malleable and can be set-up for interrupts – interrupt setup routine examples are provided. Example code is provided in Appendix A.

b. Target Software

Our target software runs various workloads to stress the I/O behavior of the device. On the target, we simply need to set the GPIO pin routed to the STX104

to go high prior to the workload, and then set the GPIO to go low once the workload is complete.

If your workload is running on top of an Operating System, it is wise to guarantee that the trigger has been set before starting the workload as noted in section b. We route the trigger pin back into another GPIO configured as an input that will trigger an interrupt on an active-high signal. Prior to starting a workload, we set the trigger high, and then we poll on the input GPIO of the target to which the trigger is also routed. We start the workload once the trigger has been sensed as active-high. This avoids situations where the trigger may be set, but the execution and scheduling paths may cause the software under test to start executing before the signal is set high.

4. Conclusion

Using the STX104, we can collect current traces of our target of interest. Using this acquired data, we can post process to calculate the power graph and overall energy consumption of the system. The STX104 provides an easy and intuitive way to easily collect fine grain, high resolution power data for power debugging. However, because it does not support simultaneous channel sampling, if one wishes to collect both current and voltage at the point of interest, there is a little bit more post processing overhead to integrate the results and properly determine power draw.

Appendix A

```
/*************************************************************************
Apex Embedded Systems
STX104 ADC Triggered Collection Example

COPYRIGHT
Copyright(c) 1999-2007 by Apex Embedded Systems.
All rights reserved.  This document is the
confidential property of Apex Embedded Systems
Any reproduction or dissemination is prohibited
unless specifically authorized in writing.

Customers free to use this or portions of code below
to develop their applications.
*************************************************************************/

#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

/************************************************************************
/                                          C OR C++ INTERRUPT ARGUMENTS
*/
#ifdef __cplusplus
#define __CPPARGS ...
#else
#define __CPPARGS
#endif
/************************************************************************
/                                                              8259 PIC
*/

/* These are the port addresses of the 8259 Programmable
Interrupt Controller (PIC).
*/
#define IMR             0x21    /* Interrupt Mask Register port */
#define ICR             0x20    /* Interrupt Control Port       */
/* An end of interrupt needs to be sent to the Control Port of
the 8259 when a hardware interrupt ends. */
#define NSEOI           0x20    /* End Of Interrupt */
```

```
/* RUNTIME CONSTANTS*/
#define MAXIMUM_BUFFER_SAMPLE_SIZE  16384
#define STX104_BLOCK_SAMPLE_SIZE      256

/*****************************************************************************
/                                                                   PROTOTYPES
*/
void interrupt(*old_isr_handler)(__CPPARGS);

/*****************************************************************************
/                                                                   VARIABLES
*/
long max_sample_blocks;

char out_file[16];
unsigned int data_buffer[MAXIMUM_BUFFER_SAMPLE_SIZE];
long isr_count;
int first_channel;
int last_channel;
int interrupt_number;
unsigned int base_address;
unsigned int ff_status;
int ff_fbr;
unsigned char ff_ff, ff_fe, ff_int;

bool individual_files;
FILE *out_file_handle;
long minimum_burst_interval;

int  analog_input_gain;
bool convert_to_voltage;

bool databus_8bits;
char databus_path;
unsigned long total_samples;
int  blocks_to_readout;

int  restricted_read_delay;

/*****************************************************************************
/
```

```
double adc_gain[8] = { 20.0 / 65536.0, 10.0 / 65536.0, 5.0 / 65536.0, 2.5 /
65536.0, 10.0 / 65536.0,  5.0 / 65536.0, 2.5 / 65536.0, 1.25 / 65536.0 };

double adc_offset[8] = { -10.0, -5.0, -2.5, -1.25, 0.0, 0.0, 0.0, 0.0 };

/*************************************************************************
/                                                        CONVERT TO VOLTAGE
*/
double Convert_To_Voltage(long value)
{
        int index;
        float voltage;
        index = inportb(base_address + 0xB) & 0x07;
        voltage = ((double)value) * adc_gain[index] + adc_offset[index];
        return voltage;
}

/*************************************************************************
/                                                                      ISR
*/
void interrupt STX104_Isr(__CPPARGS)
{
        disable();
        /* acquired at least two 256-sample FIFO blocks */
        isr_count++;

        /* updata fifo status here */
        Get_Fifo_Status();

        /* clear interrupt at STX104 */
        outportb(base_address + 8, 0x00);
        /* acknowledge at PIC */
        outportb(ICR, NSEOI);
        enable();
}

/*************************************************************************
/                                                            INSTALL ISR
*/
void Install_Isr(int int_number)
{
        unsigned char mask;
```

```
        if (interrupt_number > 0)
        {
                old_isr_handler = getvect(int_number + 8);
                setvect(int_number + 8, STX104_Isr);
                mask = 0x01 << int_number;
                outportb(IMR, inportb(IMR) & ~mask);  /* turn on PIC */
        }
}
/*************************************************************************
/                                                              RESTORE ISR
*/
void Restore_Isr(int int_number)
{
        unsigned char mask;

        if (interrupt_number > 0)
        {
                mask = 0x01 << int_number;
                outportb(IMR, inportb(IMR) | mask);  /* turn off PIC */
                setvect(int_number + 8, old_isr_handler);
        }
}

/*************************************************************************
/                                                                    INSW
*/
void Insw(unsigned int port, unsigned int *buf, int count)
{
        _ES = FP_SEG(buf);    /* Segment of buf */
        _DI = FP_OFF(buf);    /* Offset of buf  */
        _CX = count;          /* Number to read */
        _DX = port;           /* Port           */
        asm   REP INSW;
}


/*************************************************************************
/                                                              FIFO STATUS
/ Revision History:
/ 15JAN07 - read FIFO status twice to remove the possibility
/           of reading an incorrect value due to FIFO status
/           changing during a 25nSec interval.
*/
```

```
#define STX104_FIFO_STATUS_READ_COUNT_MAX   1
void Get_Fifo_Status(void)
{
        union { unsigned int value; unsigned char octet[2]; }
ff_stat[STX104_FIFO_STATUS_READ_COUNT_MAX + 1];
        unsigned int fbr_sample;
        unsigned int fbr_minimum_value;
        unsigned char i;

        fbr_minimum_value = 0x0FFF;

        for (i = 0; i <= STX104_FIFO_STATUS_READ_COUNT_MAX; i++)
        {
                ff_stat[i].octet[0] = inportb(base_address + 0x0F);  /* low
byte  */
                ff_stat[i].octet[1] = inportb(base_address + 0x0A);  /* high
byte */

                fbr_sample = ff_stat[i].value & 0x0FFF;
                if (fbr_sample < fbr_minimum_value)
                {
                        fbr_minimum_value = fbr_sample;
                }
        }
        ff_fbr = (int)fbr_minimum_value;

        ff_status = (ff_stat[STX104_FIFO_STATUS_READ_COUNT_MAX].value &
0xE000) | ff_fbr;
        if ((ff_stat[STX104_FIFO_STATUS_READ_COUNT_MAX].value & 0x1000) !=
0x0000) ff_fe = 1;
        else ff_fe = 0;
        if ((ff_stat[STX104_FIFO_STATUS_READ_COUNT_MAX].value & 0x2000) !=
0x0000) ff_ff = 1;
        else ff_ff = 0;
        if ((ff_stat[STX104_FIFO_STATUS_READ_COUNT_MAX].value & 0x8000) !=
0x0000) ff_int = 1;
        else ff_int = 0;
}

/*************************************************************************
/                                                    WORD READ INTO BUFFER
/
*/
```

```
void Fill_Word_Buffer_Read_Word_Wide(unsigned int port, unsigned int *buf,
int count)
{
        int x;
        int i;

        for (i = 0; i<count; i++)
        {
                if ((restricted_read_delay) && (databus_path == 16))
                {
                        for (x = 0; x<restricted_read_delay; x++)
                        {
                                asm{ nop };
                        }
                }
                buf[i] = inpw(port);
        }
}
/*************************************************************************
/                                                    BYTE READ INTO BUFFER
/
*/
void Fill_Word_Buffer_Read_Byte_Wide(unsigned int port, unsigned int *buf,
int count)
{
        int x;
        int i;
        union { unsigned int word; unsigned char byte[2]; } sample;

        for (i = 0; i<count; i++)
        {  /* x86 low byte first */
                if ((restricted_read_delay) && (databus_path == 8))
                {
                        for (x = 0; x<restricted_read_delay; x++)
                        {
                                asm{ nop };
                        }
                }
                sample.byte[0] = inportb(port + 0);
                sample.byte[1] = inportb(port + 1);
                buf[i] = sample.word;
        }
}
```

```
/*************************************************************************
/                                                                READ BLOCKS
/ Cannot have more than 2^15 / 2^8 = 2^7 blocks or 128 blocks
*/
void Read_Blocks_To_Buffer(int num_blocks)
{
        int quantity;
        int i;

        if (num_blocks == 0) return;

        if (databus_8bits == false)
        {
                quantity = num_blocks << 8;
                /* read in the data into the buffer */
                if (databus_path == 16)
                {
                        Fill_Word_Buffer_Read_Word_Wide(base_address,
&data_buffer[0], quantity);
                }
                else
                {
                        Insw(base_address, &data_buffer[0], quantity);
                }
        }
        else
        {
                quantity = num_blocks << 8;
                /* read in the data into the buffer */
                Fill_Word_Buffer_Read_Byte_Wide(base_address, &data_buffer[0],
quantity);
        }
}

/*************************************************************************
/                                                            PROCESS BLOCKS
*/
void Process_Blocks(int num_blocks)
{
        int quantity;
        //int actual;
```

```
        if (num_blocks == 0) return;

        quantity = num_blocks << 8;

        /* write to output file */
        fwrite(&data_buffer[0], 2, quantity, out_file_handle);
}

/*************************************************************************
/                                                              READ SAMPLES
*/
int Read_Samples_To_Buffer()
{
        int x;
        int i;
        union { unsigned int word; unsigned char byte[2]; } sample;

        Get_Fifo_Status();
        i = 0;

        if (databus_8bits == false)
        {
                while (ff_fe == 0)
                {
                        if ((restricted_read_delay) && (databus_path == 16))
                        {
                                for (x = 0; x<restricted_read_delay; x++)
                                {
                                        asm{ nop };
                                }
                        }

                        data_buffer[i] = inpw(base_address);  /* 16-bit read */

                        i++;
                        Get_Fifo_Status();
                }
        }
        else
        {
                while (ff_fe == 0)
                {  /* x86 low byte first */
                        if ((restricted_read_delay) && (databus_path == 8))
```

```
                        {
                                for (x = 0; x<restricted_read_delay; x++)
                                {
                                        asm{ nop };
                                }
                        }
                        sample.byte[0] = inportb(base_address + 0);
                        sample.byte[1] = inportb(base_address + 1);

                        /* copy to buffer */
                        data_buffer[i] = sample.word;

                        i++;
                        Get_Fifo_Status();
                }
        }
        return i;  /* number of samples read */
}

/*************************************************************************
/                                                          PROCESS SAMPLES
*/
void Process_Samples(int samples)
{
        if (samples == 0) return;

        /* write to output file */
        fwrite(&data_buffer[0], 2, samples, out_file_handle);
}
/*************************************************************************
/                                                                  PROCESS
*/
void Process()
{
        int trigger_stop, scratch;
        int read_blocks, samples_read;
        long blocks_read;
        char ch;
        float rate, rate_error;

        total_samples = 0;
        trigger_stop = 0;
        blocks_read = 0;
```

```
        samples_read = 0;
        Get_Fifo_Status();

        while (trigger_stop == 0)
        {
                /* quit if keyboard hit  */
                if (kbhit())
                {
                        trigger_stop = 1;
                }

                /* trigger_stop if maximum fifo blocks reached */
                if (blocks_read >= max_sample_blocks) trigger_stop = 2;

                /* quit if fifo is full */
                if (ff_ff != 0) trigger_stop = 3;

                /* Check if TRIG went low */
                if (((inportb(base_address + 3) & 0x01) == 0x00)) trigger_stop
= 4;

                /* stop triggering if we are to quit */
                if (trigger_stop != 0)
                {
                        /* disable interrupts and triggering */
                        scratch = inportb(base_address + 9) & 0x7C;
                        outportb(base_address + 9, scratch);
                }

                Get_Fifo_Status();

                if (ff_fbr >= blocks_to_readout)
                {
                        Read_Blocks_To_Buffer(blocks_to_readout);
                        blocks_read = blocks_read + ((long)blocks_to_readout);
                        Process_Blocks(blocks_to_readout);
                        Get_Fifo_Status();
                }
        }

        /* read out the rest of the data */
        Get_Fifo_Status();
        while (ff_fbr > 0)
```

```
        {
                if (ff_fbr > blocks_to_readout) read_blocks =
blocks_to_readout;
                else read_blocks = ff_fbr;
                Read_Blocks_To_Buffer(read_blocks);
                blocks_read = blocks_read + ((long)read_blocks);
                Process_Blocks(read_blocks);
                Get_Fifo_Status();
        }
        /*
        Read out any remaining samples.  Less than 256 samples
        may be left in FIFO.
        */
        samples_read = Read_Samples_To_Buffer();
        Process_Samples(samples_read);

        /* print what caused sampling to stop */
        printf("Trigger Stop: ");
        switch (trigger_stop)
        {
                case 1: printf("keyboard\n");  break;
                case 2: printf("normal\n");           break;
                case 3: printf("fifo full\n"); break;
                case 4: printf("DIN0 inactive\n"); break;
        }

        /* print statistics */
        total_samples = ((unsigned long)samples_read) + ((unsigned
long)(blocks_read * 256));
        //printf("Total Samples Expected  = %8ld samples\n",
MAX_ISR_COUNT*256*2 );
        printf("\n\n");
        printf("Total Samples Expected  = %8ld samples\n", max_sample_blocks *
256);
        printf("Total Samples Readout   = %8ld samples\n", total_samples);
        printf("Total Blocks Read       = %8ld blocks\n", blocks_read);
        printf("Total ISR Counts        = %8ld\n", isr_count);
        printf("\n");
}

/*****************************************************************************
/                                                                  TERMINATION
*/
```

```
void Terminate()
{
        unsigned long data_length;

        Restore_Isr(interrupt_number);

        fclose(out_file_handle);

        if (individual_files == true) Channel_Files();

}

/*************************************************************************
/                                          PRODUCE INDIVIDUAL CHAN FILES
*/
void Channel_Files()
{
        int i;
        char file_name[16];
        FILE * cfh[16];
        char msg[256];
        char tstr[16];
        FILE * in_file;
        FILE * txt_file;
        int burst_size;
        int x;
        int lines_text;
        long value;
        double voltage;

        printf("\n");
        printf("Generating Individual Output files...");

        strcpy(file_name, out_file);

        /* produce individual channel files */
        if ((in_file = fopen(file_name, "rb")) == NULL)  /* read binary */
        {
                printf("Cannot open input file.\n");
                return;
        }

        strcpy(file_name, "out.csv");
```

```
        if ((txt_file = fopen(file_name, "wb")) == NULL)  /* read binary */
        {
                printf("Cannot open output file.\n");
                return;
        }

        /* create the output files */
        strcpy(msg, "");

        for (i = first_channel; i <= last_channel; i++)
        {
                strcpy(file_name, "CH");
                itoa(i, tstr, 10);
                strcat(file_name, tstr);
                strcat(file_name, ".BIN");
                strcat(msg, "CH");
                strcat(msg, tstr);
                if (i < last_channel) strcat(msg, ",");
                else strcat(msg, "\n");

                if ((cfh[i] = fopen(file_name, "wb")) == NULL)  /* write
binary */
                {
                        printf("Cannot open output file.\n");
                        return;
                }
        }

        /* top line to csv file */
        fputs(msg, txt_file);

        x = 0;

        burst_size = last_channel - first_channel + 1;
        while (fread(&data_buffer[0], 2, burst_size, in_file) == burst_size)
        {
                for (i = first_channel; i <= last_channel; i++)
                {
                        fwrite(&data_buffer[i], 2, 1, cfh[i]);

                        /* build text row, but only first 8192 lines */
                        value = (long)data_buffer[i];
                        if (convert_to_voltage == true)
```

```
                        {
                                voltage = Convert_To_Voltage(value);
                        }
                        else
                        {
                                voltage = (double)value;
                        }
                        gcvt(voltage, 5, tstr);
                        strcat(msg, tstr);
                        if (i < last_channel) strcat(msg, ",");
                        else strcat(msg, "\n");

                }
                /* write out line to file */
                fputs(msg, txt_file);
                strcpy(msg, "");
                lines_text++;
        }

        for (i = first_channel; i <= last_channel; i++)
        {
                fclose(cfh[i]);
        }

        fclose(txt_file);
        fclose(in_file);
}

/*************************************************************************
/                                                    SET BURST TRIGGER TIMER
*/
void Set_Burst_Trigger()
{
        long number_channels, trigger_time, high_count, low_count;
        unsigned int octet;

        number_channels = (long)((last_channel & 0x0F) - (first_channel &
0x0F) + 1);

        /* calculates triggering based on burst mode */

        /* burst mode with one trigger per burst */
        trigger_time = number_channels * 5L;  /*  5uS per channel minimum */
```

```
        if (trigger_time < minimum_burst_interval) trigger_time =
minimum_burst_interval;

        /* assumes 10MHz clock (i.e. no 1MHz jumper) */
        low_count = 10L; /* 1 microsecond intervals */

        high_count = trigger_time;
        while (high_count > 65536L)
        {
                high_count = high_count >> 1;
                low_count = low_count << 1;
        }
        while (high_count < 2L)
        {
                high_count = high_count << 1;
                low_count = low_count >> 1;
        }

        /* set counter/timer 2  */
        outportb(base_address + 15, 0xB4);
        octet = ((unsigned int)high_count) & 0x00FF;
        outportb(base_address + 14, octet);

        octet = ((unsigned int)high_count) >> 8;
        outportb(base_address + 14, octet);

        /* set counter/timer 1  */
        outportb(base_address + 15, 0x74);
        octet = ((unsigned int)low_count) & 0x00FF;
        outportb(base_address + 13, octet);

        octet = ((unsigned int)low_count) >> 8;
        outportb(base_address + 13, octet);

}

/************************************************************************
/                                                          INITIALIZATION
*/
int Initialize(void)
{
        unsigned char mask;
        unsigned int number_channels;
```

```
        unsigned long fps;
        double frame_period;            /* uSec */
        double frame_period_minimum;    /* uSec */

                                                                /* open file
for output */
        if ((out_file_handle = fopen(out_file, "wb")) == NULL)  /* write
binary */
        {
                printf("Cannot open output file.\n");
                return 0;
        }

        /* Initialize STX104 */
        outportb(base_address + 9, 0x00); /* no interrupts, no DMA and s/w
trigger */
        outportb(base_address + 3, 0x00); /* DOUTs to zero
*/
        outportb(base_address + 8, 0x00); /* clear any pending interrupts
*/
        outportb(base_address + 10, 0x00); /* set pacer clock to free run
                  */
        outportb(base_address + 11, analog_input_gain); /* Gain set
          */
        outportb(base_address + 1030, 0x40); /* function enable
*/
        outportb(base_address + 1029, 0x40); /* burst enable
*/
        outportb(base_address + 1028, 0x40); /* disable triggers
*/
        outportb(base_address + 2, 0x00);  /* Reset Acquisition Controller
             */

                                                                /*
install interrupt */
        outportb(base_address + 8, 0x00); /* clear any pending interrupts
*/
        Install_Isr(interrupt_number);

        /* set counter/timer 1 & 2 to generate trigger every 10uS */
        Set_Burst_Trigger();

        /* gain */
```

```
        outportb(base_address + 11, analog_input_gain); /* gain   */

                        /* set burst count cont pacer clock control */
        mask = (last_channel & 0x0F) - (first_channel & 0x0F);
        mask = mask << 4;       /* burst count */

                                            /* DIN0 triggered */
        mask = mask | 0x01;  /* CT_SRC0=0, GCTRL=1 */

        outportb(base_address + 10, mask);

        mask = ((last_channel & 0x0F) << 4) | (first_channel & 0x0F);
        /* reset acquisition and fifo */
        outportb(base_address + 2, mask);
        isr_count = 0;  /* count indicates number of blocks gathered */

        /* the following CNV test has been added to test for STX104
        internal reset as a result of writing to Channel Register */
        /* wait for CNV to be false to make sure reset complete */
        while ((inportb(base_address + 8) & 0x80) == 0x80); /* wait */

        /* external DIN0 triggering*/
        mask = 0x00 | ((interrupt_number & 0x07) << 4) | 0x08 | 0x03;
        outportb(base_address + 9, mask);

        /* burst mode enabled */
        outportb(base_address + 1030, 0x40); /* function enable */
        outportb(base_address + 1029, 0x40); /* burst enable    */
        outportb(base_address + 1028, 0x00); /* conversion enable     */

        return 1;
}

/*************************************************************************
/                                                          TRIGGERED START
*/
void STX104_Ext_Trig_Start(void)
{
        bool not_done;
        char  ch;
        unsigned char ctrl_reg;

        not_done = true;
```

```
        /* wait for DIN0 to be high */
        do
        {
                if ((inportb(base_address + 3) & 0x01) == 0x01)
                {
                        not_done = false;
                        if (interrupt_number > 0)
                        {  /* enable interrupts */
                                ctrl_reg = inportb(base_address + 9);
                                ctrl_reg = ctrl_reg | 0x80;
                                outportb(base_address + 9, ctrl_reg);
                        }
                }
        } while (not_done);
        printf("DIN0 trigger\n");
        break;

        return status;
}

/*************************************************************************
/                                                                    MAIN
*/
int main(int argc, char *argv[])
{
        /* initialize variables */
        int blocks_to_readout_maximum; /* number of 256 blocks to collect */
        blocks_to_readout = 8;
        blocks_to_readout_maximum = (int)(MAXIMUM_BUFFER_SAMPLE_SIZE /
STX104_BLOCK_SAMPLE_SIZE);

        analog_input_gain = 0; /* +/-10v gain range*/
        max_sample_blocks = 100000; /* number of 256 sample blocks to get -
make arbitrarily large */

        restricted_read_delay = 0;
        base_address = 0x300; /* STX104 base address - based on installed
jumpers*/

        first_channel = 0; /* We want to only collect CH0 */
        last_channel = 0;  /* We want to only collect CH0 */
```

```
        minimum_burst_interval = 100; /* Sampling occurs at 100uS or 10Khz  */
        interrupt_number = 0;          /* IRQ disabled - we will just poll for
the data */

        strcpy(out_file, "out.bin");

        individual_files = false;
        convert_to_voltage = true; /* Output actual voltages instead of raw
values */

        databus_path = 16; /* 16 bit resolution */
        databus_8bits = false;

        /* cap the blocks to read out*/
        if (blocks_to_readout > blocks_to_readout_maximum)
        {
                blocks_to_readout = blocks_to_readout_maximum;
        }

        /* initialize STX104, variables and interrupts */
        if (Initialize() == 0)
        {
                printf("Initialization Failed\n");
        }
        else
        {
                printf("Initialization Complete\n");

                /* Wait for external trig to begin collections */
                STX104_Ext_Trig_Start()

                /* read in FIFO blocks and process on the fly   */
                Process();

                /* terminate STX104, variables and interrupts  */
                Terminate();
                printf("Termination Complete\n");

        }

        return 0;
}
```