Xspress API Manual

V2.2.0 **2019**

TABLE OF CONTENTS

INTRODUCTION

This document is intended to aid those writing software to control data acquisition using Quantum Detectors Xspress family digital pulse processors. It describes the top level functions of the proprietary library.

SOFTWARE OVERVIEW

The software API is provided by the proprietary library, libxspress3.a. Libxspress3.a is written in C and supplied as a binary file which is only compatible with 64bit Linux operating systems. The library provides calling functions with names of the form xsp3_*.

The library will work will all Xspress variants. From time to time Quantum Detectors will update the library as a result of maintenance, bug fixes, or in response to changes in the hardware. We work to ensure backwards compatibility with older hardware each time an update is made.

All Xspress systems require a Linux PC to drive them. Quantum Detectors only supports Enterprise Linux distributions (RHEL, CentOS), though binaries are available for Debian based operating systems on request.

Setup procedure

By default, communication with Xspress 3 is via ethernet using control port 30123. For Xspress 3 Mini and Xspress 4 the port is 30124. The software tries ports 30123 and then 30124 then reads the revision register to determine which variant it is controlling. The library provides a common API to all three generations where possible.

The library supports running 1 or more Xspress 3 systems. To connect to a system of 1 or more boards use the command xsp3_config. This returns a handle which should be used to identify the system to all subsequent calls. Next setup the clock source. For Xspress 3 normal operation the ADC and data processing clock is set to use the crystal oscillator module on the ADC board using the following:

```
xsp3_clocks_setup(path, -1, XSP3_CLK_SRC_XTAL, XSP3_CLK_FLAGS_MASTER | XSP3_CLK_FLAGS_NO_DITHER,
0)
```

For Xspress 3 Mini there are two choices of ADC board clocks: CDCM61004 or LMK61E2. These are set using either:

```
xsp3_clocks_setup(path, -1, XSP3M_CLK_SRC_CDCM61004, XSP3_CLK_FLAGS_NO_DITHER, 0)
```

or

```
xsp3_clocks_setup(path, -1, XSP3M_CLK_SRC_LMK61E2, XSP3_CLK_FLAGS_NO_DITHER, 0)
```

For all generations to allow digital only testing it is possible to generate the processing clock from the FPGA on board clocks using:

```
xsp3_clocks_setup(path, -1, XSP3_CLK_SRC_INT, XSP3_CLK_FLAGS_NO_DITHER, 0)
```

The processing register settings and BRAM contents can then be loaded using xsp3_restore_settings. To hide the complexity of the clock setup, the library will save the clock configuration in the settings directory. The settings and clock setup can be restored using xsp3_restore_settings_and_clock().

The overall run flags should then be set to specify whether scalers should be recorded. See xsp3_restore_settings

The timing control register allows the acquisition enable/disable and time frame number to come from either software control, the internal time frame generator or from external hardware - for accurately timed multi-frame experiments. See xsp3_set_glob_timeA and xsp3_set_glob_timeFixed

User Settings

Two in window scalers are provided per channel, typically used to measure the counts in the main characteristic peak of 1 or 2 elements See **xsp3_set_window.** The histogram data is usually set to collect the full spectrum in 4096 energy bins. Spectra size is set using **xsp3_format_run**.

Collection

If the internal time frame generator is being used, this is setup using **xsp3_itfg_setup()**. Once setup the histogramming memory is cleared using **xsp3_histogram_clear**. Then the system is started using **xsp3_histogram_start**. If the timing control has been set to a fixed frame from software control, counting then starts. Wait for the desired time and then counting is disabled using **xsp3_histogram_stop**. It is possible to get the system to an armed start using **xsp3_histogram_arm** and then start counting with **xsp3_histogram_continue**. With software timing it is possible to stop the current frame and then start counting into the next frame using **xsp3_histogram_pause()** and **xsp3_histogram_continue()**.

If the internal time frame generator is used the system starts and the ITFG controls counting. It will start immediately unless the trigger mode **XSP3_ITFG_TRIG_MODE** has been set to make it wait for an internal or external trigger. The ITFG will take a burst of frames, waiting for software or hardware triggers between frames if specified by the trigger mode.

If the timing control has been set to use external timing inputs, counting is armed, but will occur only when the timing veto input is asserted. If the external trigger is negated and then asserted again then acquisition moves onto the next frame. Progress of the current input time frame can be monitored using **xsp3_get_glob_time_statA()**. However, acquisition into memory will lag this slightly. If the internal time frame generator is being used, the progress is monitored using **xsp3_get_glob_time_statA()** and the status of the ITFG can be extracted from the upper bits using **XSP3_GLOBAL_TIME_STATUS_A**. If timing if being controlled by hardware, the external timing generator must determine when the exposure is finished. After this call **xsp3_histogram_stop** to flush remaining data.

Data Access

For Xspress 3 and Xspress 4 the histogram data is stored in /dev/shm shared memory on the Linux server. It can be read at any time, accepting that data read will be slightly stale while histogramming is running. For current Xspress 3 and Xspress 4 versions, the scalars are accumulated into /dev/shm shared memory

or calculated from the spectra as necessary. In Xspress 3 Mini the normal spectra are collected into BRAM and transferred to DRAM by DMA at the end of the frame. The scalars are counted in VHDL scalars and transferred to DRAM by DMA. To provide a common API to these implementations, user code should call **xsp3_scaler_check_progress_details()** or its simplified interface **xsp3_scaler_check_progress()**. This will report the number of completed frames and it is safe to read data up to that point.

Note that with the original Xspress 3 version the intention was that while the experiment was running, realtime display data would be read out, accepting that it may be slightly stale. The intention was that at the end of the experiment **xsp3_histogram_stop** would be called to start flushing of the UDP data. After calling **xsp3_histogram_stop**, **xsp3_histogram_is_busy** or for a compatible interface for a multi-box system **xsp3_histogram_is_any_busy()** was to be polled until two consecutive calls returned not busy to check that the histogramming threads had flushed any network stack buffers. These functions are still supported for Xspress 3 and Xspress 4, but have no meaning for Xspress 3 Mini. They are largely superseded by **xsp3_scaler_check_progress_details()**.

However, to save time at the end of a many frame experiment, the EPICS and TANGO/LIMA implementations start reading early frames once the experiment has finished them. For current Xspress 3/Xspress 4 and Xspress 3 Mini **xsp3_scaler_check_progress_details()** provides the information as to how far the system has got. Completed frames can safely be read. For the original Xspress 3 (scalar in FEM DRAM), **xsp3_scaler_check_progress_details()** measures progress by seeing how many time frames of scalars have been written to FEM-1 DRAM. The scalars can safely be read out up to this frame. However, they may be some lag in the processing of the UDP data. Currently there is no way to verify when this has finished.

The histogram data can then read using any of **xsp3_histogram_read4d**, **xsp3_histogram_read3d** or **xsp3_histogram_read_chan.** The raw scaler data is then read using **xsp3_scaler_read.**

Circular Buffer Mode

Normally the maximum number of time frames in an experiment is set when calling **xsp3_config()** With a typical 16384 frames used in conventional 4096 bin spectra mode, this requires 256 MB of DRAM per channel. In Xspress 3, this can be increased significantly by installing more memory in the server and increasing the value passed to to **xsp3_config()**. There are limits in the library at 1 MB frames, 16 GB per frame.

To allow for experiments requiring very large numbers of time frames, the system can be operated in circular buffer mode. The user software has to read the data keeping up with incoming frame rate. It can lag, but only to the size of the number of frames in the circular buffer, typically 16384, and has to save the data to (networked) disk. This is enabled by calling **xsp3_set_run_flags()** including the flag XSP3_RUN_FLAGS_CIRCULAR_BUFFER

The receive threads clear each time frame before they histograms into them. The receive threads also software extends the 24 bit firmware time frame to currently 64 bits. The bottom 32 bits of the extended time frame are used to address the histogram and scalar read functions. They are wrapped to read from the circular buffer.

There is also a time frames status buffer. This is an area in /dev/shm. It stores 4 off 32 bit unsigned integers per channel per circular buffer time frame. One row per channel stores a used count which is incremented whenever a receive thread starts histogramming into a given frame. If when the receive thread wraps round to use that frame again, the used count has not been cleared by a call to **xsp3_histogram_circ_ack()**, the system detects wrap round. The new frame is collected and the old frame is over written. If this happens, it is recorded in various error status bits, which can be seen by **xsp3_scaler_check_progress_details()** setting Xsp3ErrFlag_CircBuffOverRun in the error flags.

The extended time frame stored in each circular buffer frame is stored in 2 rows of the time frame status module. The full status information for a frame/channel can be read using **xsp3_histogram_get_tf_status()**. For use in particularly long raster scan experiments, there is an option to use TTL_IN(2..3) to give 2 markers in the data. These are saved in the 3rd row of the time frame status module and can be read back by **xsp3_histogram_get_tf_markers()**. This feature also exists in normal (single pass) buffer mode.

The circular buffer mode is also available in Xspress 3 Mini, but the implementation is rather different. The circular buffer is within the DRAM, with the frames being written to the DRAM via AXIS DMA engines. These are programmed to stop with an error rather overwrite the data. The API is preserved except that now calling **xsp3_histogram_circ_ack()** is mandatory. The API for **xsp3_histogram_get_tf_status** still exists, though it is now strongly encouraged to use **xsp3_histogram_get_tf_status_block()** as this is more efficient.

ERROR CODES

The following error codes maybe returned;

```
#define XSP3_OK                 0
#define XSP3_ERROR              -1
#define XSP3_INVALID_PATH       -2
#define XSP3_ILLEGAL_CARD       -3
#define XSP3_ILLEGAL_SUBPATH    -4
#define XSP3_INVALID_DMA_STREAM -5
#define XSP3_RANGE_CHECK        -6
#define XSP3_INVALID_SCOPE_MOD  -7
#define XSP3_OUT_OF_MEMORY      -8
#define XSP3_ERR_DEV_NOT_FOUND  -9
#define XSP3_CANNOT_OPEN_FILE   -10
#define XSP3_FILE_READ_FAILED   -11
#define XSP3_FILE_WRITE_FAILED  -12
#define XSP3_FILE_RENAME_FAILED -13
#define XSP3_LOG_FILE_MISSING   -14
#define XSP3_WOULD_BLOCK        -20
```

EXAMPLE PROGRAMS

A demonstration/example program has been written which uses **xsp3_scaler_check_progress_details()** to monitor progress and follow the experiment read data as completed frames become available. It can work with an external timing generator or the internal time frame generator (ITFG).

```
 * Program to demonstrate how to setup and readout scaler data for Xspress3.
 *
 * Matthew Pearson, Nov 2012.
 */

#include "stdio.h"

#include "xspress3.h"

#define XSP3_MAXFRAMES 16384
#define XSP3_MAXSPECTRA 4096
#define XSP3_CONFIGPATH "/etc/xspress3/calibration/initial/settings/"
#define MAX_CHECKDESC_POLLS 1000

int main(int argc, char *argv[])
{
    unsigned int poll = 0;
    unsigned int chan = 0;
    unsigned int sca = 0;
    unsigned int energy = 0;
    unsigned int frame = 0;
    int xsp3_handle = 0;
    int xsp3_status = 0;
    unsigned int num_frames = 0;
    unsigned int num_frames_to_read = 0;
    unsigned int last_num_frames = 0;
    Xsp3ErrFlag error_flags;

    u_int32_t *pRAW = NULL;
    u_int32_t *pRAW_OFFSET = NULL;
    u_int64_t cur_frame;

    double *pSCA = NULL;
    double *pSCA_OFFSET = NULL;
    double *pDUMP = NULL;
    unsigned int dump_offset = 0;
    unsigned int dump_offset_mca = 0;
    double *pMCA = NULL, *pMCA_OFFSET;
    u_int32_t *pMCA_RAW = NULL;
    u_int32_t *pMCA_RAW_OFFSET;
    int i;
    int first_card=0, num_cards=1;
    int debug=1;
    int num_chan;
    int use_dtc = 0;
    int use_itfg=0;
    unsigned int nframes = 0;
    double frame_len = 1.0;
    u_int32_t tstat;
    int running, counting, paused, finished=0;
    int prev_frame = -1;
    int prev_paused = -1;
    int prev_finished = 0;
    int show_progress=0;

    for (i=1; i<argc; i++)
    {
        if (strncmp(argv[i],"-card=", 6) == 0)
            first_card = strtol(argv[i]+6, NULL, 0);
        else if (strncmp(argv[i],"-num-cards=", 11) == 0)
            num_cards = strtol(argv[i]+11, NULL, 0);
```

```
        else if (strcmp(argv[i],"-quiet") == 0)
            debug = 0;
        else if (strcmp(argv[i],"-debug") == 0)
            debug = 1;
        else if (strcmp(argv[i],"-verbose") == 0)
            debug = 2;
        else if (strcmp(argv[i],"-dtc") == 0)
            use_dtc = 1;
        else if (strcmp(argv[i],"-progress") == 0)
            show_progress = 1;
        else if (strncmp(argv[i],"-nframes=", 9) == 0)
        {
            use_itfg = 1;
            nframes = strtol(argv[i]+9, NULL, 0);
        }
        else if (strncmp(argv[i],"-len=", 5) == 0)
        {
            use_itfg = 1;
            frame_len = strtod(argv[i]+5, NULL);
        }
        else
        {
            printf("Unknown option %s\n", argv[i]);
            printf("USAGE: %s [-card=<first_card> -num-cards=<num cards> -quiet -debug -verbose -
nframes=<nn> -len=<seconds> -progress]\n");
            exit(1);
        }
    }


    printf("Connect to the Xspress3...\n");

    xsp3_handle = xsp3_config(num_cards, XSP3_MAXFRAMES, NULL, -1, NULL, -1, 1, NULL, 1,
first_card);
    if (xsp3_handle < 0) {
        printf("ERROR calling xsp3_config. Return code: %d\n", xsp3_handle);
        return EXIT_FAILURE;
    }
    num_chan = xsp3_get_num_chan(xsp3_handle);

    pRAW = (u_int32_t*)(calloc(XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan, sizeof(u_int32_t)));
    pSCA = (double*)(calloc(XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan, sizeof(double)));
    pMCA = (double*)(calloc(XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA, sizeof(double)));
    pMCA_RAW = (u_int32_t*)(calloc(XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA, sizeof(u_int32_t)));

    /*
    printf("Set up clocks register to use ADC clock...\n");
    xsp3_status = xsp3_clocks_setup(xsp3_handle, 0, XSP3_CLK_SRC_XTAL, XSP3_CLK_FLAGS_MASTER, 0);
    if (xsp3_status != XSP3_OK) {
    printf("ERROR calling xsp3_clocks_setup. Return code: %d\n", xsp3_status);
    return EXIT_FAILURE;
    }
    */

    printf("Restoring the settings from the configuration files...\n");
    xsp3_status = xsp3_restore_settings_and_clock(xsp3_handle, XSP3_CONFIGPATH, 0);
    if (xsp3_status != XSP3_OK) {
        printf("ERROR calling xsp3_restore_settings. Return code: %d\n", xsp3_status);
        return EXIT_FAILURE;
    }

    printf("Calling xsp3_format_run...\n");
    for (chan=0; chan<num_chan; ++chan) {
        xsp3_status = xsp3_format_run(xsp3_handle, chan, 0, 0, 0, 0, 0, 12);
        if (xsp3_status < XSP3_OK) {
            printf("ERROR calling xsp3_restore_settings. channel: %d, Return code: %d\n", chan,
xsp3_status);
            return EXIT_FAILURE;
        }
        else {
            printf("  Channel: %d, Number of time frames configured: %d\n", chan, xsp3_status);
```

```
        }
    }

    printf("Set up playback data output...\n");
    xsp3_status = xsp3_set_run_flags(xsp3_handle, XSP3_RUN_FLAGS_PLAYBACK |
XSP3_RUN_FLAGS_SCALERS | XSP3_RUN_FLAGS_HIST);
    if (xsp3_status < XSP3_OK) {
        printf("ERROR calling xsp3_set_run_flags. Return code: %d\n", xsp3_status);
        return EXIT_FAILURE;
    }
    if (use_itfg == 0)
    {
        printf("Set up trigger source to use external veto input...\n");
        xsp3_status = xsp3_set_glob_timeA(xsp3_handle, 0,
XSP3_GLOB_TIMA_TF_SRC(XSP3_GTIMA_SRC_TTL_VETO_ONLY));
        if (xsp3_status != XSP3_OK) {
            printf("ERROR calling xsp3_set_glob_timeA. Return code: %d\n", xsp3_status);
            return EXIT_FAILURE;
        }
    }
    else
    {
        printf("Set up trigger source to use Internal TFG for %d frames of %g s...\n", nframes,
frame_len);
        xsp3_status = xsp3_set_glob_timeA(xsp3_handle, 0,
XSP3_GLOB_TIMA_TF_SRC(XSP3_GTIMA_SRC_INTERNAL));
        if (xsp3_status != XSP3_OK) {
            printf("ERROR calling xsp3_set_glob_timeA. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
            return EXIT_FAILURE;
        }
        xsp3_status = xsp3_itfg_setup(xsp3_handle, 0, nframes, (u_int32_t)(80000000*frame_len),
XSP3_ITFG_TRIG_MODE_BURST, XSP3_ITFG_GAP_MODE_25NS);
        if (xsp3_status != XSP3_OK)
        {
            printf("ERROR calling xsp3_itfg_setup. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
            return EXIT_FAILURE;
        }

    }

    printf("Starting histogramming...\n");
    xsp3_status = xsp3_histogram_start(xsp3_handle, -1);        // Start all cards.
    if (xsp3_status != XSP3_OK) {
        printf("ERROR calling xsp3_histogram_start. Return code: %d\n", xsp3_status);
        return EXIT_FAILURE;
    }

    printf("Now polling xsp3_scaler_check_progress_details and printing out any scaler
data...\n");

    for (poll=0; poll<MAX_CHECKDESC_POLLS ; )
    {
        if ((use_itfg && nframes != 0) || show_progress)
        {
            if ((xsp3_status = xsp3_get_glob_time_statA(xsp3_handle, 0, &tstat)) < 0)
            {
                printf("ERROR: Cannot read timing status: %s\n", xsp3_get_error_message());
                return xsp3_status;
            }

            frame = XSP3_GLOB_TSTAT_A_FRAME(tstat);
            running = XSP3_GLOB_TSTAT_A_ITFG_RUNNING(tstat);
            counting = XSP3_GLOB_TSTAT_A_ITFG_COUNTING(tstat);
            paused = XSP3_GLOB_TSTAT_A_ITFG_PAUSED(tstat);
            finished = XSP3_GLOB_TSTAT_A_ITFG_FINISHED(tstat);

            if (show_progress)
            {
                if (frame != prev_frame || paused != prev_paused || prev_finished != finished)
```

```
                    printf("ITFG Frame = %d, running=%d, counting=%d, paused=%d, finished=%d\n",
frame, running, counting, paused, finished);
                prev_frame  = frame;
                prev_paused = paused;
                prev_finished = finished;
            }
        }
        cur_frame = xsp3_scaler_check_progress_details(xsp3_handle,&error_flags, 0, NULL);
        if (cur_frame < 0) {
            printf("ERROR calling xsp3_scaler_check_progress_details. poll number: %d, Return
Message: %s, code: %d\n", poll, xsp3_get_error_message(), xsp3_status);
            return EXIT_FAILURE;
        }
       printf("  Check_desc poll: %d, Current frame=%d\r", poll, cur_frame);
        fflush(stdout);

        if (cur_frame > last_num_frames) {

            num_frames = cur_frame;
            num_frames_to_read = num_frames - last_num_frames;
            printf("\n.... last_num_frames: %d, reading out %d\n", last_num_frames,
num_frames_to_read);
            pRAW_OFFSET = pRAW+last_num_frames*(XSP3_SW_NUM_SCALERS * num_chan);
            pSCA_OFFSET = pSCA+last_num_frames*(XSP3_SW_NUM_SCALERS * num_chan);
            pMCA_OFFSET = pMCA+last_num_frames*(XSP3_MAXSPECTRA * num_chan);
            pMCA_RAW_OFFSET = pMCA_RAW+last_num_frames*(XSP3_MAXSPECTRA * num_chan);
            if (use_dtc)
            {
                xsp3_status = xsp3_hist_dtc_read4d(xsp3_handle, pMCA_OFFSET, NULL, 0, 0, 0,
last_num_frames, XSP3_MAXSPECTRA, 1, num_chan, num_frames_to_read);
//              xsp3_status = xsp3_scaler_dtc_read(xsp3_handle, pSCA_OFFSET, 0, 0,
last_num_frames, XSP3_SW_NUM_SCALERS, num_chan, num_frames_to_read);
            }
            else
            {
               xsp3_status = xsp3_scaler_read(xsp3_handle, pRAW_OFFSET, 0, 0, last_num_frames,
XSP3_SW_NUM_SCALERS, num_chan, num_frames_to_read);
                xsp3_status = xsp3_histogram_read4d(xsp3_handle, pMCA_RAW_OFFSET, 0, 0, 0,
last_num_frames, XSP3_MAXSPECTRA, 1, num_chan, num_frames_to_read);
            }
            if (xsp3_status < XSP3_OK) {
                printf("ERROR calling xsp3_scaler_read. Return code: %d\n", xsp3_status);
                return EXIT_FAILURE;
            }

            /*printf("  Printing %d frames of scaler data...\n", num_frames_to_read);
            pDUMP = pSCA;
            for (frame=last_num_frames; frame<(num_frames_to_read+last_num_frames); frame++) {
            for (chan=0; chan<num_chan; chan++) {
              for (sca=0; sca<XSP3_SW_NUM_SCALERS; sca++) {
                //printf("  frame: %d, chan: %d, sca: %d, data[%d]: %d\n",
                //   frame, chan, sca, dump_offset, *(pSCA_DUMP+dump_offset));
                printf("  frame: %d, chan: %d, sca: %d, data[%d]: %f\n",
                   frame, chan, sca, dump_offset, *(pDUMP+dump_offset));
                dump_offset++;
              }
            }
            }*/

            /*pDUMP = pMCA;
            printf("  Printing elements of spectra data...\n");
            for (energy=0; energy<XSP3_MAXSPECTRA; energy++) {
            printf("  energy[%d]: %f\n", energy, *(pDUMP+dump_offset_mca));
            dump_offset_mca++;
            }*/

            last_num_frames = num_frames;
        }
        if (use_itfg && nframes != 0 && finished)
            poll++; /* Count polls once teh Itfg has finished */
```

```
    } /*end of poll loop*/

    printf("\nDone.\n");

    return EXIT_SUCCESS;

}

/**
```

CIRCULAR BUFFER EXAMPLE PROGRAM

An extensive test and demonstration program has been provided for circular buffer readout mode.

```
 * Program to demonstrate how to setup and readout MCA and scalar data for Xspress3 when running
with the circular buffer enabled
 *
 * Derived from test_check_desc by  Matthew Pearson, Nov 2012.
 *                                  William Helsby   Nov 2016.
 */

#include <stdio.h>
#include <math.h>

#include "xspress3.h"

#include "test_circ_buff.h"

#define XSP3_MAXFRAMES 16384
#define XSP3_MAXSPECTRA 4096
#define XSP3_CONFIGPATH "/etc/xspress3/calibration/initial/settings/"

#define MAX_CHECKDESC_POLLS 1000
#define MAX_EXT_FRAMES  (16*1024*1024*4)
#define MAX_NUM_CHAN 12
#define TF_STATUS_BLOCK 100

int print_errors=1;
int max_prints = 20;
#define CHECK_MATCH 0   // For varying rate data, just check for agreement between scalars and
MCA
#define CHECK_AVE   1   // For fixed rate playback data, check counts against the average.
#define CHECK_RATE  2   // Check varying rate matches expected for generated data.

int check_type=CHECK_AVE;
int good_thres=100;
int win_low=540;
int win_high=700;
int marker_modulus = 80;
int marker_frame = 20;
int fc_del_frame = 10;
int fc_del_modulus= 30;
int itfg_cont=0;
int alternate_readout=0;
XspressGeneration generation;
int num_chan=-1;
int just_time = 0;

typedef struct _pass_info
{
    int errors;
    unsigned nframes;
    int64_t  frames_read;
    unsigned ave_block, max_block;
} PassInfo;

int ave_first(MOD_IMAGE3D *mod, int num_t, int chan, int *min, int *max)
{
```

```
    int sum = 0;
    int i;
    u_int32_t *p;
    double ave, sd;

    p = xsp3_mod_get_ptr(mod, 0, 0, chan);

    for (i=0; i<num_t; i++)
        sum += *p++;
    ave = (double) sum/ (double)num_t;
    sd = sqrt(ave);
    if (min != NULL)
        *min= (int)(ave-4*sd-0.06*ave);
    if (max != NULL)
        *max = (int)(ave+4*sd+0.06*ave);

    return (int) ave;
}

#define USE_ITFG_OFF            0
#define USE_ITFG_BURST          1
#define USE_ITFG_CONTINUOUS     2
#define USE_ITFG_SW_PAUSE       3
#define USE_ITFG_SW_FRAME_CAP   4
#define USE_ITFG_FW_MARKER      5
#define USE_ITFG_FW_FRAME_CAP   6

u_int32_t frame_len_cycles;
#define FIRST_ERROR_A if (first_error) {first_error = 0; printf("%sT=%d=(%d,%d), Ch=%d: ",
(msg_level==1&&first_error_this_block)?"\n":"", frame, frame%RES_MOD_X, frame/RES_MOD_X, chan);
first_error_this_block=0;}
#define FIRST_ERROR_B if (first_error) {first_error = 0; printf("T=%d=(%d,%d), Ch=%d: ",
frame+last_num_frames, frame%RES_MOD_X, frame/RES_MOD_X, chan);}
#define FIRST_ERROR_C if (first_error) {first_error = 0; printf("%sT=%d=(%d,%d), Ch=%d: ",
(msg_level==1&&first_error_this_block)?"\n":"", frame+j, (frame+j)%RES_MOD_X,
(frame+j_/RES_MOD_X, chan); first_error_this_block=0;}

MOD_IMAGE3D *mca_all_mod, *mca_good_mod, *scal_all_mod, *scal_good_mod, *mca_win_mod,
*scal_win_mod, *time_sum_mod, *time_save_mod, *marker_mod;
mh_com *mca_all_head, *mca_good_head, *scal_all_head, *scal_good_head, *mca_win_head,
*scal_win_head, *time_sum_head, *time_save_head, *marker_head;


int main(int argc, char *argv[])
{
    unsigned int poll = 0;
    unsigned int chan = 0;
    unsigned int sca = 0;
    unsigned int energy = 0;
    unsigned int frame = 0;
    int xsp3_handle = 0;
    int xsp3_status = 0;
    unsigned int num_frames = 0;
    unsigned int num_frames_to_read = 0;
    unsigned int last_num_frames = 0;
    Xsp3ErrFlag prev_error_flags=0, error_flags;
    unsigned max_num_frames;

    u_int32_t *pRAW = NULL;
    u_int32_t *pRAW_OFFSET = NULL;
    int64_t cur_frame, prev_cur_frame=-1, furthest_frame;
    int64_t checked_frames = 0;
    int check_itfg_frame = 0;

    double *pSCA = NULL;
    double *pSCA_OFFSET = NULL;
    double *pDUMP = NULL;
    unsigned int dump_offset = 0;
    unsigned int dump_offset_mca = 0;
    double *pMCA = NULL, *pMCA_OFFSET;
    u_int32_t *pMCA_RAW = NULL;
```

```
    u_int32_t *pMCA_RAW_OFFSET;
    int i, j;
    int first_card=0, num_cards=1;
    int debug=1;
    int use_dtc = 0;
    int use_itfg=0;
    unsigned int nframes = 0, itfg_nframes;
    double frame_len = 1.0;
    u_int32_t tstat;
    int running, counting, paused, finished=0;
    int prev_frame = -1;
    int prev_paused = -1;
    int prev_finished = 0;
    int show_progress=0;
    u_int32_t *scalar, *mca;
    u_int64_t errors = 0, total_errors = 0;
    u_int64_t errors_per_chan[MAX_NUM_CHAN];
    int mca_all_ave[MAX_NUM_CHAN], good_ave[MAX_NUM_CHAN], scal_all_ave[MAX_NUM_CHAN],
win_ave[MAX_NUM_CHAN], time_ave[MAX_NUM_CHAN];
    int mca_all_min[MAX_NUM_CHAN], good_min[MAX_NUM_CHAN], scal_all_min[MAX_NUM_CHAN],
win_min[MAX_NUM_CHAN], time_min[MAX_NUM_CHAN];
    int mca_all_max[MAX_NUM_CHAN], good_max[MAX_NUM_CHAN], scal_all_max[MAX_NUM_CHAN],
win_max[MAX_NUM_CHAN], time_max[MAX_NUM_CHAN];
    int msg_level=1;
    int64_t num_overrun, first_overrun;
    int first_to_process;
    int num_prints;
    int pass, num_pass=1;
    FILE *ofp=NULL;
    int scan_rate=0;
    int64_t first_error_frame, last_error_frame;
    int skipped_read;
    u_int32_t time_a, time_b;
    int itfg_frame;
    int first_error_b = 1;
    int rc;
    XspressGeneration generation;
    PassInfo *pass_info=NULL;
    unsigned ave_block, max_block, num_blocks;
    enum    {GenNone, GenFixed, GenRampRate} gen_type = GenNone;
    int playback_num_t=1;
    Xsp3GenDataType gd_type;
    int read_alt=0;
    int64_t frame_start_ts[MAX_NUM_CHAN];
    int chans_card0;
    int frame_gap=80;
    int lib_debug=1;

    gd_type.period_type = Xsp3GDPTriangleRate;
    gd_type.height_type = Xsp3GDHKAlphaBeta;
    gd_type.ave_hgt= 100.0;
    gd_type.min_sep = 25;
    gd_type.max_sep = 1000;
    gd_type.stream_sep = 10;

    for (i=1; i<argc; i++)
    {
        if (strncmp(argv[i],"-card=", 6) == 0)
            first_card = strtol(argv[i]+6, NULL, 0);
        else if (strncmp(argv[i],"-num-cards=", 11) == 0)
            num_cards = strtol(argv[i]+11, NULL, 0);
        else if (strcmp(argv[i],"-quiet") == 0)
            debug = 0;
        else if (strcmp(argv[i],"-debug") == 0)
            debug = 1;
        else if (strcmp(argv[i],"-verbose") == 0)
            debug = 2;
        else if (strcmp(argv[i],"-dtc") == 0)
            use_dtc = 1;
        else if (strcmp(argv[i],"-progress") == 0)
            show_progress = 1;
```

```
        else if (strcmp(argv[i],"-alt") == 0)
            alternate_readout = 1;
        else if (strncmp(argv[i],"-alt=", 5) == 0)
            read_alt = strtol(argv[i]+5, NULL, 0);
        else if (strncmp(argv[i],"-num-chan=", 10) == 0)
            num_chan = strtol(argv[i]+10, NULL, 0);
        else if (strcmp(argv[i],"-gen-rate") == 0)
        {
            check_type = CHECK_RATE;
            gen_type = GenRampRate;
        }
        else if (strcmp(argv[i],"-check-rate") == 0)
        {
            check_type = CHECK_RATE;
        }
        else if (strncmp(argv[i],"-nframes=", 9) == 0)
        {
            use_itfg = 1;
            nframes = strtol(argv[i]+9, NULL, 0);
        }
        else if (strncmp(argv[i],"-len=", 5) == 0)
        {
            use_itfg = 1;
            frame_len = strtod(argv[i]+5, NULL);
        }
        else if (strncmp(argv[i],"-msg=", 5) == 0)
        {
            msg_level = strtol(argv[i]+5, NULL, 0);
        }
        else if (strcmp(argv[i], "-scan-rate") == 0)
        {
            use_itfg = 1;
            scan_rate = 1;
        }
        else if (strcmp(argv[i], "-pause") == 0)
        {
            use_itfg = USE_ITFG_SW_PAUSE;
        }
        else if (strcmp(argv[i], "-sw-fc") == 0)
        {
            use_itfg = USE_ITFG_SW_FRAME_CAP;
        }
        else if (strcmp(argv[i], "-mark") == 0)
        {
            use_itfg = USE_ITFG_FW_MARKER;
        }
        else if (strcmp(argv[i], "-fc") == 0)
        {
            use_itfg = USE_ITFG_FW_FRAME_CAP;
        }
        else if (strcmp(argv[i], "-just-time") == 0)
        {
            just_time = 1;      // Just measure timing, set input to read ADC even if playback is
present
        }
        else
        {
            printf("Unknown option %s\n", argv[i]);
            printf("USAGE: %s [-card=<first_card> -num-cards=<num cards> -num-chan=<num-chan> -
quiet -debug -verbose -nframes=<nn> -len=<seconds> -progress -msg=<message level>] -sw-fc -fc -
mark -alt -gen-rate -check-rate -just-time\n", argv[0]);
            exit(1);
        }
    }


    printf("Connect to the Xspress3...\n");

    xsp3_handle = xsp3_config(num_cards, XSP3_MAXFRAMES, NULL, -1, NULL, num_chan, 1, NULL,
lib_debug, first_card);
    if (xsp3_handle < 0) {
```

```
        printf("ERROR calling xsp3_config. Return code: %d\n", xsp3_handle);
        return EXIT_FAILURE;
    }
    num_chan = xsp3_get_num_chan(xsp3_handle);
    generation = xsp3_get_generation(xsp3_handle, 0);
    if (num_chan < 0 || generation == XspressGenError)
    {
        printf("Error rading number of channels = %d or generation= %d: %s\n", num_chan,
generation, xsp3_get_error_message());
        return EXIT_FAILURE;
    }
    chans_card0 = xsp3_get_num_chan_used(xsp3_handle, 0);

    /* These buffers are used to read up to the XSP3_MAXFRAMES from the circular buffer */
    pRAW = (u_int32_t*)(calloc(XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan, sizeof(u_int32_t)));
    pSCA = (double*)(calloc(XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan, sizeof(double)));
    pMCA = (double*)(calloc(XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA, sizeof(double)));
    pMCA_RAW = (u_int32_t*)(calloc(XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA, sizeof(u_int32_t)));

    /* The data is reduced and stored in a much longer data module to check for dropped frames */

    mca_all_mod   = xsp3_mkmod3d ("mca_all",   RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &mca_all_head);
    mca_good_mod  = xsp3_mkmod3d ("mca_good",  RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &mca_good_head);
    scal_all_mod  = xsp3_mkmod3d ("scal_all",  RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &scal_all_head);
    scal_good_mod = xsp3_mkmod3d ("scal_good", RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &scal_good_head);
    mca_win_mod   = xsp3_mkmod3d ("mca_win",   RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &mca_win_head);
    scal_win_mod  = xsp3_mkmod3d ("scal_win",  RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &scal_win_head);
    time_sum_mod  = xsp3_mkmod3d ("time_sum",  RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &time_sum_head);
    time_save_mod = xsp3_mkmod3d ("time_save", RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF
Ext (low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &time_save_head);
    marker_mod = xsp3_mkmod3d ("markers", RES_MOD_X, MAX_EXT_FRAMES/RES_MOD_X, num_chan, "TF Ext
(low)", "TF Ext (high)", "Chan", NULL, DATA_LONG, &marker_head);

    if (mca_all_mod == NULL || mca_good_mod == NULL || scal_all_mod == NULL || scal_good_mod ==
NULL ||  mca_win_mod== NULL || scal_win_mod == NULL ||
        time_sum_mod == NULL || time_save_mod == NULL || marker_mod == NULL)
    {
        fprintf(stderr, "ERROR creating modules: %s\n", xsp3_get_error_message());
        exit(1);
    }

    if (!just_time)
    {
        printf("Restoring the settings from the configuration files...\n");
        xsp3_status = xsp3_restore_settings_and_clock(xsp3_handle, XSP3_CONFIGPATH, 1);
        if (xsp3_status != XSP3_OK)
        {
            printf("ERROR calling xsp3_restore_settings. Return '%s, code: %d\n",
xsp3_get_error_message(), xsp3_status);
            return EXIT_FAILURE;
        }
    }
    else
        printf("In just-time mode have all zero control registers\n");

/*
        This is all done by libxspress3 xsp4_system_start_count_enb now.
if (generation == XspressGen3Mini)
    {
        if (num_cards > 1)
            xsp3_status = xsp4_rmw_glob_reg(xsp3_handle, -1, XSP4_GLOB_SCOPE_CONT, 0xFFFFFFFF,
XSP4_GSCOPE_PLAYBACK_COUNT_ENB, NULL);
        else
```

```
            xsp3_status = xsp4_rmw_glob_reg(xsp3_handle, -1, XSP4_GLOB_SCOPE_CONT,
~XSP4_GSCOPE_PLAYBACK_COUNT_ENB, 0,  NULL);
        if (xsp3_status != 0)
        {
            printf("Error setting scope mode playback triggering: %s\n",
xsp3_get_error_message());
            return EXIT_FAILURE;
        }
    }
    else if (generation == XspressGen4)
    {
        if (num_cards > 1)
            xsp3_status = xsp4_rmw_glob_reg(xsp3_handle, -1, XSP4_GLOB_SCOPE_CONT, 0xFFFFFFFF,
XSP4_GSCOPE_PLAYBACK_RADIAL, NULL);
        else
            xsp3_status = xsp4_rmw_glob_reg(xsp3_handle, -1, XSP4_GLOB_SCOPE_CONT,
~XSP4_GSCOPE_PLAYBACK_RADIAL, 0,  NULL);
        if (xsp3_status != 0)
        {
            printf("Error setting scope mode playback triggering: %s\n",
xsp3_get_error_message());
            return EXIT_FAILURE;
        }
    }
*/
    if (just_time)
    {
        xsp3_set_chan_cont(xsp3_handle, -1, 0);
    }
    else if (gen_type == GenRampRate || check_type == CHECK_RATE)
    {
        for (chan=0; chan<num_chan; chan++)
        {
            u_int32_t chan_cont;
            xsp3_get_chan_cont(xsp3_handle, chan, &chan_cont);
            chan_cont &= ~XSP3_CC_SEL_DATA(0xFF);
            if (generation == XspressGen3)
            {
                if (chan & 1)
                    chan_cont |= XSP3_CC_SEL_DATA(XSP3_CC_SEL_DATA_EXT1);
                else
                    chan_cont |= XSP3_CC_SEL_DATA(XSP3_CC_SEL_DATA_EXT0);
            }
            else
                chan_cont |= XSP3_CC_SEL_DATA(XSP3_CC_SEL_DATA_PB_CHAN);
            chan_cont &= ~XSP3_CC_SEL_ENERGY(0xFF);
            chan_cont |= XSP3_CC_SEL_ENERGY(2); // Energy scaling is now 4
            xsp3_set_chan_cont(xsp3_handle, chan, chan_cont);
        }
        xsp3_bram_init(xsp3_handle, -1, XSP3_REGION_RAM_QUOTIENT, -1.0);    // No additonal
scaling in quotient RAM
        win_low  = 0.95*4*gd_type.ave_hgt;
        win_high = 1.05*4*gd_type.ave_hgt;
    }
    if (gen_type != GenNone)
    {
        playback_num_t = xsp3_playback_generate(xsp3_handle, -1, &gd_type);
    }
    else if (check_type == CHECK_RATE)
        playback_num_t = xsp3_playback_get_num_t(xsp3_handle, 0);

    printf("Calling xsp3_format_run...\n");
    for (chan=0; chan<num_chan; ++chan)
    {
        max_num_frames = xsp3_format_run(xsp3_handle, chan, 0, 0, 0, 0, 0, 12);
        if (max_num_frames < XSP3_OK) {
            printf("ERROR calling xsp3_restore_settings. channel: %d, Return code: %d\n", chan,
max_num_frames);
            return EXIT_FAILURE;
        }
        else {
```

```
            printf("  Channel: %d, Number of time frames configured: %d\n", chan,
max_num_frames);
        }
    }
    xsp3_set_good_thres(xsp3_handle, -1, good_thres);
    xsp3_set_window(xsp3_handle, -1, 0, win_low, win_high);

    printf("Set up playback data output...\n");
    xsp3_status = xsp3_set_run_flags(xsp3_handle, XSP3_RUN_FLAGS_PLAYBACK |
XSP3_RUN_FLAGS_SCALERS | XSP3_RUN_FLAGS_HIST | XSP3_RUN_FLAGS_CIRCULAR_BUFFER);
    if (xsp3_status < XSP3_OK) {
        printf("ERROR calling xsp3_set_run_flags. Return code: %d\n", xsp3_status);
        return EXIT_FAILURE;
    }

    if (scan_rate)
    {
        num_pass = 11;
        print_errors = 0;

    }
    if (num_pass  < 4  && alternate_readout)
        num_pass = 3;

    if (num_pass > 1)
    {
        ofp = fopen("results.txt", "w");
        if (ofp == NULL)
        {
            fprintf(stderr, "Cannot open output file results.txt\n");
            exit(1);
        }
        fprintf(ofp, "frame_len\tGood Frames\tnum_overrun\terrors\tave_block\tmax_block\n");
    }
    if ((pass_info = (PassInfo *)calloc(num_pass, sizeof(PassInfo))) == NULL)
    {
        printf("Out of memory\n");
        exit(1);
    }
    for (pass=0; pass<num_pass; pass++)
    {
        errors = 0;
        for (i=0; i<MAX_NUM_CHAN; i++)
        {
            errors_per_chan[i] = 0;
            frame_start_ts[i] = 0;
        }
        last_num_frames = 0;
        prev_error_flags=0;
        prev_cur_frame=-1;
        checked_frames = 0;
        prev_frame = -1;
        prev_paused = -1;
        prev_finished = 0;
        itfg_frame = 0;
        check_itfg_frame = 0;
        ave_block = 0;
        max_block=0;
        num_blocks=0;
        id_clear_mod(mca_all_mod);
        id_clear_mod(mca_good_mod);
        id_clear_mod(scal_all_mod);
        id_clear_mod(scal_good_mod);
        id_clear_mod(mca_win_mod);
        id_clear_mod(scal_win_mod);
        id_clear_mod(time_sum_mod);
        id_clear_mod(time_save_mod);
        id_clear_mod(marker_mod);

        memset(pRAW, 0,  XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan*sizeof(u_int32_t));
        memset(pSCA, 0, XSP3_SW_NUM_SCALERS*XSP3_MAXFRAMES*num_chan*sizeof(double));
```

```
        memset(pMCA, 0, XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA*sizeof(double));
        memset(pMCA_RAW, 0, XSP3_MAXFRAMES*num_chan*XSP3_MAXSPECTRA*sizeof(u_int32_t));

        if (alternate_readout)
            read_alt = pass % 4;

        if (scan_rate)
        {
            if (generation == XspressGen3Mini)
                frame_len = (1000-pass*80)*1E-6*num_cards;
            else
                frame_len = (250-pass*10)*1E-6;
        }
        if (use_itfg == 0)
        {
            printf("Set up trigger source to use external veto input...\n");
            xsp3_status = xsp3_set_glob_timeA(xsp3_handle, -1,
XSP3_GLOB_TIMA_TF_SRC(XSP3_GTIMA_SRC_TTL_VETO_ONLY));
            if (xsp3_status != XSP3_OK) {
                printf("ERROR calling xsp3_set_glob_timeA. Return code: %d\n", xsp3_status);
                return EXIT_FAILURE;
            }
        }
        else
        {
            int trig_mode = XSP3_ITFG_TRIG_MODE_BURST;
            int m_period=0, m_frame=0;
            Xsp3Timing timing;

            memset(&timing, 0, sizeof(Xsp3Timing));
            printf("Set up trigger source to use Internal TFG for %d frames of %g s...\n",
nframes, frame_len);
            // First set all other cards to use LEMO cables with some debounce.
#if 0
            xsp3_status = xsp3_set_glob_timeA(xsp3_handle, -1,
XSP3_GLOB_TIMA_TF_SRC(XSP3_GTIMA_SRC_TTL_VETO_ONLY)|XSP3_GLOB_TIMA_DEBOUNCE(5));
            xsp3_status = xsp3_set_glob_timeFixed(xsp3_handle, -1, 0);
            time_a = XSP3_GLOB_TIMA_TF_SRC(XSP3_GTIMA_SRC_INTERNAL);
            time_b = 0;
            if (use_itfg == USE_ITFG_SW_PAUSE)
            {
                time_b |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_SW);
                trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE;
            }
            else if (use_itfg == USE_ITFG_FW_MARKER)
            {
                time_b |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_ITFG);
            }
            else if (use_itfg == USE_ITFG_SW_FRAME_CAP)
            {
                time_a |= XSP3_GLOB_TIMA_FRAME_CAPTURE;
                time_b |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_SW);
                trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE;
                if (fc_del_frame != 0)
                    time_a |= XSP3_GLOB_TIMA_SW_MARKERS(1); // Exercise Software bit 0 as Capture
Bit
            }
            else if (use_itfg == USE_ITFG_FW_FRAME_CAP)
            {
                time_a |= XSP3_GLOB_TIMA_FRAME_CAPTURE;
                time_b |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_ITFG);
            }

            xsp3_status = xsp3_set_glob_timeA(xsp3_handle, 0, time_a);
            if (xsp3_status != XSP3_OK) {
                printf("ERROR calling xsp3_set_glob_timeA. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
            xsp3_status = xsp3_set_glob_timeFixed(xsp3_handle, 0, time_b);
            if (xsp3_status != XSP3_OK) {
```

```
                printf("ERROR calling xsp3_set_glob_timeFixed. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
#else
            timing.t_src = XSP3_GTIMA_SRC_INTERNAL;
            if (use_itfg == USE_ITFG_SW_PAUSE)
            {
                timing.fixed |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_SW);
                trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE;
            }
            else if (use_itfg == USE_ITFG_FW_MARKER)
            {
                // Markers testing not extended to multi card systems yet.
                timing.fixed |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_ITFG);
            }
            else if (use_itfg == USE_ITFG_SW_FRAME_CAP)
            {
                timing.card0_a_extra |= XSP3_GLOB_TIMA_FRAME_CAPTURE;
                timing.cardn_a_extra |= XSP3_GLOB_TIMA_FRAME_CAPTURE;
                timing.fixed |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_SW);
                trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE;
                if (fc_del_frame != 0)
                {
                    timing.card0_a_extra |= XSP3_GLOB_TIMA_SW_MARKERS(1);   // Exercise Software
bit 0 as Capture Bit
                    timing.cardn_a_extra |= XSP3_GLOB_TIMA_SW_MARKERS(1);   // Exercise Software
bit 0 as Capture Bit
                }
            }
            else if (use_itfg == USE_ITFG_FW_FRAME_CAP)
            {
                // Frame capture test not extended to muti-card systems yet
                timing.card0_a_extra |= XSP3_GLOB_TIMA_FRAME_CAPTURE;
                timing.fixed |= XSP3_GLOB_TIMB_ITFG_MARK(XSP3_GTIMB_IMRK_ITFG);
            }

            xsp3_status = xsp3_set_timing(xsp3_handle, &timing);
            if (xsp3_status != XSP3_OK) {
                printf("ERROR calling xsp3_set_timing. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
            // Get glob_timeA so software can continue to manipulate this in tests.
            xsp3_status = xsp3_get_glob_timeA(xsp3_handle, 0, &time_a);
            if (xsp3_status != XSP3_OK) {
                printf("ERROR calling xsp3_get_glob_timeA. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
#endif
            if (use_itfg == USE_ITFG_FW_MARKER)
            {
                m_period = marker_modulus;
                m_frame = marker_frame;
            }
            else if (use_itfg == USE_ITFG_FW_FRAME_CAP)
            {
                m_period = fc_del_modulus;
                m_frame = fc_del_frame;
            }
            if (nframes >= 16*1014*1024-1)
            {
                itfg_cont = 1;
                itfg_nframes = 0;
            }
            else
            {
                itfg_nframes = nframes;
                itfg_cont = 0;
            }
```

```
            printf("Marker period=%d, marker_frame=%d\n", m_period, m_frame);
            frame_len_cycles = (u_int32_t)(80000000*frame_len);
            xsp3_status = xsp3_itfg_setup2(xsp3_handle, 0, itfg_nframes, frame_len_cycles,
trig_mode, XSP3_ITFG_GAP_MODE_1US, 0, m_period, m_frame);
            if (xsp3_status != XSP3_OK)
            {
                printf("ERROR calling xsp3_itfg_setup. Return error %s: code: %d\n",
xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
        }

        printf("Starting histogramming...\n");
        xsp3_status = xsp3_histogram_start(xsp3_handle, -1);        // Start all cards.
        if (xsp3_status != XSP3_OK) {
            printf("ERROR calling xsp3_histogram_start. Return code: %s: %d\n",
xsp3_get_error_message(), xsp3_status);
            return EXIT_FAILURE;
        }
        itfg_frame = 1;
        printf("Now polling xsp3_scaler_check_progress_details and printing out any scaler
data...\n");

        if (msg_level >= 1)
            show_progress = 1;

        for (poll=0; poll<MAX_CHECKDESC_POLLS ; )
        {
            int itfg_print=0, check_prog_print = 0;
            if ((use_itfg && nframes != 0) || show_progress)
            {
                if ((xsp3_status = xsp3_get_glob_time_statA(xsp3_handle, 0, &tstat)) < 0)
                {
                    printf("ERROR: Cannot read timing status: %s\n", xsp3_get_error_message());
                    return xsp3_status;
                }

                frame = XSP3_GLOB_TSTAT_A_FRAME(tstat);
                running = XSP3_GLOB_TSTAT_A_ITFG_RUNNING(tstat);
                counting = XSP3_GLOB_TSTAT_A_ITFG_COUNTING(tstat);
                paused = XSP3_GLOB_TSTAT_A_ITFG_PAUSED(tstat);
                finished = XSP3_GLOB_TSTAT_A_ITFG_FINISHED(tstat);

                if (show_progress)
                {
                    if (frame != prev_frame || paused != prev_paused || prev_finished !=
finished)
                        itfg_print = 1;
                    prev_frame  = frame;
                    prev_paused = paused;
                    prev_finished = finished;
                }
                if (paused)
                {
                    if (use_itfg == USE_ITFG_SW_PAUSE || use_itfg == USE_ITFG_SW_FRAME_CAP)
                    {
                        xsp3_get_glob_timeA(xsp3_handle, 0, &time_a);   // Exercise Software bit
1 as a status bit
                        time_a &= ~XSP3_GLOB_TIMA_COUNT_ENB;        // Negate CountEnb so
continue can assert it.
                        time_a &= ~XSP3_GLOB_TIMA_SW_MARKERS(3); // Negate both markers
                        if (itfg_frame % marker_modulus == marker_frame)
                            time_a |= XSP3_GLOB_TIMA_SW_MARKERS(2); // Exercise Software bit 1 as
a status bit
                        if (use_itfg == USE_ITFG_SW_FRAME_CAP && itfg_frame % fc_del_modulus !=
fc_del_frame)
                            time_a |= XSP3_GLOB_TIMA_SW_MARKERS(1); // Exercise Software bit 0 as
Capture Bit
                        xsp3_set_glob_timeA(xsp3_handle, 0, time_a );   // Exercise Software bit
1 as a status bit
```

```
                        xsp3_set_glob_timeA(xsp3_handle, 0, time_a | XSP3_GLOB_TIMA_COUNT_ENB );
// Set CountEnb to Continue, only OK on single card system
                        itfg_frame++;
                    }
                    else
                    {
                        printf("Unexpected pause state in ITFG\n");
                        exit (1);
                    }
                }
            }
            cur_frame = xsp3_scaler_check_progress_details(xsp3_handle, &error_flags, 1, NULL);
            if (cur_frame < 0) {
                printf("ERROR calling xsp3_scaler_check_progress_details. poll number: %d, Return
Message: %s, code: %d\n", poll, xsp3_get_error_message(), xsp3_status);
                return EXIT_FAILURE;
            }
            if (error_flags != prev_error_flags)
                cur_frame = xsp3_scaler_check_progress_details(xsp3_handle, &error_flags, 0,
NULL);     // Print the error report once
            prev_error_flags = error_flags;

            if (msg_level > 0 && (itfg_print || prev_cur_frame != cur_frame))
            {
                printf("Firmware Frame = %d, running=%d, counting=%d, paused=%d, finished=%d.
Hist Cur Frame=%ld, Flags=%08X", frame, running, counting, paused, finished,  cur_frame,
error_flags);
                if (cur_frame > last_num_frames)
                    printf(".... last_num_frames: %d, reading out %d     ", last_num_frames,
cur_frame - last_num_frames );
                else
                    printf("                                                    ");
                if (msg_level >= 2)
                    printf("\n");
                else
                {
                    printf("\r");
                    fflush(stdout);
                }
            }
            prev_cur_frame = cur_frame;
            if ((use_itfg == USE_ITFG_SW_FRAME_CAP || use_itfg ==  USE_ITFG_FW_FRAME_CAP) &&
cur_frame > 0)
                cur_frame-- ;   // In Frame Capture mode with SW pauses we will get an End Frame
marker marking the current frame as complete before getting the first packet of the next frame
                                // With the delete bit set which re-clears the frame and starts
again.
            if (cur_frame > last_num_frames)
            {
                num_frames = cur_frame;
                num_frames_to_read = num_frames - last_num_frames;
                num_blocks++;
                if (num_frames_to_read > max_block)
                    max_block = num_frames_to_read;
                ave_block += num_frames_to_read;

                if (num_frames_to_read > max_num_frames)
                {
                    last_num_frames = num_frames-max_num_frames;
                    printf("\nERROR: Overrun as num_frames_to_read=%d > %d. Skipping %d to read
from %d\n", num_frames_to_read, XSP3_MAXFRAMES, num_frames_to_read - XSP3_MAXFRAMES,
last_num_frames);
                    errors++;
                    num_frames_to_read = max_num_frames;
                    checked_frames = last_num_frames;
                    skipped_read = 1;
                }
                else
                    skipped_read = 0;
                if (use_dtc)
                {
```

```
                    xsp3_status = xsp3_hist_dtc_read4d(xsp3_handle, pMCA, NULL, 0, 0, 0,
last_num_frames, XSP3_MAXSPECTRA, 1, num_chan, num_frames_to_read);
    //              xsp3_status = xsp3_scaler_dtc_read(xsp3_handle, pSCA_OFFSET, 0, 0,
last_num_frames, XSP3_SW_NUM_SCALERS, num_chan, num_frames_to_read);
                }
                else
                {
                    switch (read_alt)
                    {
                    case 1:
                        read_by_chan(xsp3_handle, last_num_frames,  num_frames_to_read);
                        break;
                    case 2:
                        read_by_card(xsp3_handle, last_num_frames,  num_frames_to_read);
                        break;
                    case 3:
                        read_by_parts(xsp3_handle, last_num_frames,  num_frames_to_read);
                        break;

                    default:
                        xsp3_status = xsp3_scaler_read(xsp3_handle, pRAW, 0, 0, last_num_frames,
XSP3_SW_NUM_SCALERS, num_chan, num_frames_to_read);
                        if (!just_time)
                            xsp3_status = xsp3_histogram_read4d(xsp3_handle, pMCA_RAW, 0, 0, 0,
last_num_frames, XSP3_MAXSPECTRA, 1, num_chan, num_frames_to_read);
                        break;
                    }
                }
                if (use_itfg == USE_ITFG_SW_PAUSE || use_itfg == USE_ITFG_SW_FRAME_CAP ||
use_itfg == USE_ITFG_FW_MARKER || use_itfg == USE_ITFG_FW_FRAME_CAP)
                {
                    if (alternate_readout && (pass & 1))
                    {
                        frame = last_num_frames;
                        for (i=0; i<num_frames_to_read; i+=TF_STATUS_BLOCK)
                        {
                            unsigned this_num_frames = cur_frame-frame;
                            if (this_num_frames > TF_STATUS_BLOCK)
                                this_num_frames = TF_STATUS_BLOCK;
                            for (chan=0; chan<num_chan; chan++)
                            {
                                int first_error=1;
                                Xsp3TFStatus tf_status[TF_STATUS_BLOCK];
                                u_int32_t *dp;
                                int mod_x = frame % RES_MOD_X;
                                int mod_y = frame / RES_MOD_X;
                                xsp3_histogram_get_tf_status_block(xsp3_handle, chan, frame,
this_num_frames, tf_status);
                                dp = xsp3_mod_get_ptr(marker_mod, mod_x, mod_y, chan);
                                for (j=0; j<this_num_frames; j++)
                                {
                                    if (tf_status[j].state != 1)
                                    {
                                        if (++num_prints < max_prints && print_errors)
                                        {
                                            printf("TF Status: Frame not completed at T=%ld\n",
frame+j);
                                        }
                                        errors++;
                                    }
                                    if (tf_status[j].time_frame != (int64_t)frame+j)
                                    {
                                        if (++num_prints < max_prints && print_errors)
                                        {
                                            printf("TF Status: Time frame mismatch at T=%ld,
found %ld\n", frame+j, tf_status[j].time_frame);
                                        }
                                        errors++;
                                    }
                                    *dp = tf_status[j].markers;
                                    mod_x++;
```

```
                                    dp++;
                                    if (mod_x == RES_MOD_X)
                                    {
                                        mod_x = 0;
                                        mod_y++;
                                        dp = xsp3_mod_get_ptr(marker_mod, mod_x, mod_y, chan);
                                    }
                                }
                            }
                            frame += TF_STATUS_BLOCK;
                        }
                    }
                    else
                    {
                        int mod_x = last_num_frames % RES_MOD_X;
                        int mod_y = last_num_frames / RES_MOD_X;
                        u_int32_t *dp;
                        for (chan=0; chan<num_chan; chan++)
                        {
                            if(msg_level > 2)
                                printf("Reading markers into module Chan %d at (%d, %d) from %u
for %u\n", chan, mod_x, mod_y, last_num_frames, num_frames_to_read);
                            dp = xsp3_mod_get_ptr(marker_mod, mod_x, mod_y, chan);
                            if ((rc=xsp3_histogram_get_tf_markers(xsp3_handle, chan,
last_num_frames, num_frames_to_read, (int *)dp)) < 0)
                                printf("\nERROR reading tf markers:%s:%d\n",
xsp3_get_error_message(), rc);
                        }
                    }
                }
                if (xsp3_status < XSP3_OK) {
                    printf("\nERROR calling xsp3_scaler_read. Return code: %s:%d\n",
xsp3_get_error_message(), xsp3_status);
                    return EXIT_FAILURE;
                }
                cur_frame = xsp3_scaler_check_progress_details(xsp3_handle, &error_flags, 1,
&furthest_frame);
                if (skipped_read)
                {
                    printf("... after skipped read furthest_frame-max_num_frames=%ld -
last_num_frames=%d = %ld\n", furthest_frame-max_num_frames, last_num_frames, furthest_frame-
max_num_frames-last_num_frames);
                }
                first_to_process = 0;
                if (furthest_frame-max_num_frames >= (int64_t)last_num_frames)
                {
                    first_to_process = 10+furthest_frame-max_num_frames - last_num_frames;  //
cur_frame is the LOWEST frame, some may be further on.
                    checked_frames = last_num_frames+first_to_process;
                    printf("%s... After read current frame=%ld, further_frame=%ld has overrun
buffer, last_num_frame=%d, omitting %d checking from %d", msg_level==1?"\n":"",
                        cur_frame, furthest_frame, last_num_frames, first_to_process,
checked_frames);
                }
                if ((rc=xsp3_histogram_circ_ack(xsp3_handle, 0, last_num_frames, num_chan,
num_frames_to_read)) <0)
                {
                    printf("\nERROR acknowledging read: %s : %d\n", xsp3_get_error_message(),
rc);
                }
                if (read_alt == 0)
                {
                    scalar = pRAW+first_to_process*num_chan*XSP3_SW_NUM_SCALERS;
                    first_error_b=1;
                    for (frame=first_to_process; frame<num_frames_to_read; frame++)
                    {
                        for (chan=0;chan<num_chan; chan++)
                        {
                            u_int32_t acc_time =
scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_LIVE_TICKS];
```

```
                            u_int32_t total_time =
scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_TOTAL_TICKS];
                            if (use_itfg)
                            {
                                if (acc_time < 2 || acc_time < frame_len_cycles-2 || acc_time >
frame_len_cycles+2 || total_time < 2 || acc_time < total_time -2 || acc_time > total_time+2)
                                {
                                    if (++num_prints < max_prints && print_errors)
                                    {
                                        if (first_error_b)
                                        {
                                            printf("\n");
                                            first_error_b = 0;
                                        }
                                        printf("T=%d=(%d,%d), Ch=%d: ", (frame+last_num_frames),
(frame+last_num_frames)%RES_MOD_X, (frame+last_num_frames)/RES_MOD_X, chan);
                                        printf("ERROR in live time: read accumulated=%d, read
total=%d, Expected=%d\n", acc_time, total_time, frame_len_cycles);
                                    }
                                    errors++;
                                    errors_per_chan[chan]++;
                                }
                            }
                            else
                            {
                                if (acc_time < 2 || total_time < 2 || acc_time < total_time-2 ||
acc_time > total_time+2)
                                {
                                    if (++num_prints < max_prints && print_errors)
                                    {
                                        if (first_error_b)
                                        {
                                            printf("\n");
                                            first_error_b = 0;
                                        }
                                        printf("T=%d=(%d,%d), Ch=%d: ", (frame+last_num_frames),
(frame+last_num_frames)%RES_MOD_X, (frame+last_num_frames)/RES_MOD_X, chan);
                                        printf("ERROR in live time: read accumulated=%d, read
total=%d\n", acc_time, total_time);
                                    }
                                    errors++;
                                    errors_per_chan[chan]++;
                                }
                            }
                        }
                        scalar += num_chan*XSP3_SW_NUM_SCALERS;
                    }
                }
                if (read_alt == 0)
                {
                    scalar = pRAW+first_to_process*num_chan*XSP3_SW_NUM_SCALERS;
                    for (frame=first_to_process; frame<num_frames_to_read; frame++)
                    {
                        int mod_x = (last_num_frames+frame) % RES_MOD_X;
                        int mod_y = (last_num_frames+frame) / RES_MOD_X;
                        for (chan=0;chan<num_chan; chan++)
                        {
                            u_int32_t *datap;
                            u_int32_t total;
                            int eng;
                            mca = pMCA_RAW + XSP3_MAXSPECTRA*num_chan*frame +
chan*XSP3_MAXSPECTRA;

                            total = 0;
                            for (eng=0; eng<XSP3_MAXSPECTRA; eng++)
                                total += mca[eng];
                            datap = xsp3_mod_get_ptr(mca_all_mod, mod_x, mod_y, chan);
                            *datap = total;

                            total = 0;
                            for (eng=good_thres; eng<XSP3_MAXSPECTRA; eng++)
```

```
                            total += mca[eng];
                        datap = xsp3_mod_get_ptr(mca_good_mod, mod_x, mod_y, chan);
                        *datap = total;

                        total = 0;
                        for (eng=win_low; eng<=win_high; eng++)
                            total += mca[eng];
                        datap = xsp3_mod_get_ptr(mca_win_mod, mod_x, mod_y, chan);
                        *datap = total;

                        datap = xsp3_mod_get_ptr(scal_all_mod, mod_x, mod_y, chan);
                        *datap = scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_ALL_EVENT];
                        datap = xsp3_mod_get_ptr(scal_good_mod, mod_x, mod_y, chan);
                        *datap = scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_ALL_GOOD];
                        datap = xsp3_mod_get_ptr(scal_win_mod, mod_x, mod_y, chan);
                        *datap = scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_IN_WINDOW0];
                        datap = xsp3_mod_get_ptr(time_sum_mod, mod_x, mod_y, chan);
                        *datap = scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_LIVE_TICKS];
                        datap = xsp3_mod_get_ptr(time_save_mod, mod_x, mod_y, chan);
                        *datap = scalar[chan*XSP3_SW_NUM_SCALERS+XSP_SW_SCALER_TOTAL_TICKS];
                    }
                    scalar += num_chan*XSP3_SW_NUM_SCALERS;
                }
            }
            if (num_frames > 20)
            {
                if (checked_frames == 0)
                {
                    for (chan=0; chan<num_chan; chan++)
                    {
                        time_ave[chan] = ave_first(time_save_mod, num_frames, chan, NULL,
NULL);
                        if (chan < chans_card0)
                        {
                            time_min[chan] = time_ave[chan]-1;
                            time_max[chan] = time_ave[chan]+1;
                        }
                        else
                        {
                            time_min[chan] = time_ave[chan]-2; // On other cards, due to
clock drift and sampling count enb across differetn clock domains, allow slightly wider tolerance
                            time_max[chan] = time_ave[chan]+2;
                        }
                    }
                }
                if (check_type == CHECK_AVE)
                {
                    if (checked_frames == 0)
                    {
                        for (chan=0; chan<num_chan; chan++)
                        {
                            mca_all_ave[chan]  = ave_first(mca_all_mod, num_frames, chan,
mca_all_min+chan, mca_all_max+chan);
                            scal_all_ave[chan] = ave_first(scal_all_mod, num_frames, chan,
scal_all_min+chan, scal_all_max+chan);
                            good_ave[chan]     = ave_first(mca_good_mod, num_frames, chan,
good_min+chan, good_max+chan);
                            win_ave[chan]      = ave_first(mca_win_mod, num_frames, chan,
win_min+chan, win_max+chan);
                        }
                    }
                }
                num_prints = 0;
                first_error_frame = -1;
                for (frame=checked_frames; frame<num_frames; frame++)
                {
                    int first_error_this_block = 1;
                    int mod_x = frame % RES_MOD_X;
                    int mod_y = frame / RES_MOD_X;
                    int error_this_frame=0;
```

```
                        if ((use_itfg == USE_ITFG_SW_FRAME_CAP || use_itfg ==
USE_ITFG_FW_FRAME_CAP) && check_itfg_frame % fc_del_modulus == fc_del_frame)
                            check_itfg_frame++;     // If ITFG frame was marked as Delete, this
should be from the next frame
                        for (chan=0; chan<num_chan; chan++)
                        {
                            int first_error=1;
                            int *mca_all, *mca_good, *scal_all, *scal_good;
                            int *mca_win, *scal_win;
                            int *time_sum, *time_save;
                            int *markerP;
                            mca_all = xsp3_mod_get_ptr(mca_all_mod, mod_x, mod_y, chan);
                            mca_good = xsp3_mod_get_ptr(mca_good_mod, mod_x, mod_y, chan);
                            scal_all = xsp3_mod_get_ptr(scal_all_mod, mod_x, mod_y, chan);
                            scal_good = xsp3_mod_get_ptr(scal_good_mod, mod_x, mod_y, chan);
                            mca_win = xsp3_mod_get_ptr(mca_win_mod, mod_x, mod_y, chan);
                            scal_win = xsp3_mod_get_ptr(scal_win_mod, mod_x, mod_y, chan);
                            time_sum  = xsp3_mod_get_ptr(time_sum_mod, mod_x, mod_y, chan);
                            time_save = xsp3_mod_get_ptr(time_save_mod, mod_x, mod_y, chan);

                            if (check_type == CHECK_RATE)
                            {
                                int64_t ts1, ts2;
//                              ts1 = ((int64_t)frame_len_cycles+80)*frame;
//                              ts2 = ((int64_t)frame_len_cycles+80)*frame+frame_len_cycles;

                                if (chan < chans_card0 || frame_start_ts[chan] < 10000)
                                    ts1 = frame_start_ts[chan] + (int64_t)frame_gap*frame;
                                else
                                    ts1 = frame_start_ts[chan] +
(int64_t)frame_gap*frame*((double)frame_start_ts[chan])/frame_start_ts[0];
                                if (use_itfg)
                                    ts2 = ts1+frame_len_cycles;
                                else
                                    ts2 = ts1+ *time_save;
                                if (chan < chans_card0 && use_itfg)
                                    frame_start_ts[chan] += frame_len_cycles;
                                else
                                    frame_start_ts[chan] += *time_save;

                                mca_all_max[chan] =
xsp3_playback_generate_est_counts(xsp3_handle, chan, frame, &gd_type, ts1, ts2, mca_all_min+chan,
NULL)+3;
                                scal_all_min[chan] = mca_all_min[chan];
                                scal_all_max[chan] = mca_all_max[chan];
                                good_min[chan] = mca_all_min[chan];
                                good_max[chan] = mca_all_max[chan];
                            }
                            if ((check_type== CHECK_AVE || check_type == CHECK_RATE) && (*mca_all
< mca_all_min[chan] || *mca_all > mca_all_max[chan] || *scal_all < scal_all_min[chan] ||
*scal_all > scal_all_max[chan]) || *mca_all < (0.98* *scal_all)-1 || *mca_all > *scal_all+2)
                            {
                                if (++num_prints < max_prints && print_errors)
                                {
                                    FIRST_ERROR_A
                                    printf("mca_all=%d, scal_all=%d, mca_min=%d, mca_max=%d",
*mca_all, *scal_all, mca_all_min[chan], mca_all_max[chan]);
                                }
                                errors++;
                                error_this_frame++;
                                errors_per_chan[chan]++;
                            }
                            if ((check_type== CHECK_AVE || check_type == CHECK_RATE) &&
(*mca_good < good_min[chan] || *mca_good > good_max[chan] || *scal_good < good_min[chan] ||
*scal_good > good_max[chan] ) || *mca_good < *scal_good || *mca_good > *scal_good)
                            {
                                if (++num_prints < max_prints && print_errors)
                                {
                                    FIRST_ERROR_A
                                    printf("mca_good=%d, scal_good=%d, good_min=%d, good_max=%d",
*mca_good, *scal_good, good_min[chan], good_max[chan]);
```

```
                                }
                                errors++;
                                error_this_frame++;
                                errors_per_chan[chan]++;
                            }
                            if (check_type== CHECK_AVE && (*mca_win < win_min[chan] || *mca_win >
win_max[chan] || *scal_win < win_min[chan] || *scal_win > good_max[chan]) || *mca_win < *scal_win
|| *mca_win > *scal_win)
                            {
                                if (++num_prints < max_prints && print_errors)
                                {
                                    FIRST_ERROR_A
                                    printf("mca_win=%d, scal_win=%d, win_min=%d, win_max=%d",
*mca_win, *scal_win, win_min[chan], win_max[chan]);
                                }
                                errors++;
                                error_this_frame++;
                                errors_per_chan[chan]++;
                            }
                            if (first_error == 0)
                                printf("\n");
                            first_error = 1;
                            if (*time_sum < time_min[chan] || *time_sum > time_max[chan] ||
*time_save < time_min[chan] || *time_save > time_max[chan] || *time_save < *time_sum || *time_sum
> *time_save)
                            {
                                if (++num_prints < max_prints && print_errors)
                                {
                                    FIRST_ERROR_B
                                    printf("time_sum=%d, time_save=%d, time_min=%d, time_max=%d",
*time_sum, *time_save, time_min[chan], time_max[chan]);
                                }
                                errors++;
                                error_this_frame++;
                                errors_per_chan[chan]++;
                            }
                            if (first_error == 0)
                                printf("\n");
                            if (use_itfg == USE_ITFG_SW_PAUSE || use_itfg ==
USE_ITFG_SW_FRAME_CAP || use_itfg == USE_ITFG_FW_MARKER)
                            {
                                if (generation == XspressGen4 || chan < chans_card0)    // Unless
markers are distributed to all cards, they are only available on channel on the fist card
                                {
                                    int expected_marker = (check_itfg_frame % marker_modulus) ==
marker_frame;
                                    markerP = xsp3_mod_get_ptr(marker_mod, mod_x, mod_y, chan);
                                    if (*markerP != expected_marker)
                                    {
                                        if (++num_prints < max_prints && print_errors)
                                        {
                                            FIRST_ERROR_B
                                            printf("Marker %d, Expected=%d", *markerP,
expected_marker);
                                        }
                                        errors++;
                                        error_this_frame++;
                                        errors_per_chan[chan]++;
                                    }
                                }
                            }
                        }
                        if (error_this_frame)
                        {
                            if (first_error_frame == -1)
                                last_error_frame = first_error_frame = frame;
                            else if (last_error_frame == frame-1)
                                last_error_frame++;
                            else
                            {
```

```
                                printf("\n Errors in Frame %ld to %ld\n", first_error_frame,
last_error_frame);
                                if (check_type == CHECK_RATE)
                                {
                                    printf("Frame Time stamps : ");
                                    for (i=0;i<num_chan; i++)
                                        printf("  %ld", frame_start_ts[i]);
                                    printf("\n");
                                }
                                first_error_frame = -1;
                            }
                        }
                        check_itfg_frame++;
                    }
                    if (first_error_frame != -1)
                        printf("\n Errors in Frame %ld to %ld\n", first_error_frame,
last_error_frame);
                    checked_frames = num_frames;
                }
                last_num_frames = num_frames;
                if (itfg_cont && num_frames >= nframes)
                {
                    xsp3_histogram_stop(xsp3_handle, -1);
                    break;
                }
            }
            if (use_itfg && nframes != 0 && finished)
                poll++; /* Count polls once the Itfg has finished */

        } /*end of poll loop*/
        if (num_blocks > 0)
            ave_block /= num_blocks;

        if (nframes > 0 && nframes != cur_frame)
        {
            printf("\nERROR: Lost frames: sent %d, received %d\n", nframes, cur_frame);
            errors++;
        }
        num_overrun = xsp3_histogram_get_circ_overrun(xsp3_handle, -1, &first_overrun);
        if (num_overrun > 0)
        {
            printf("\nDetected %ld over runs, first frame=%ld\n", num_overrun, first_overrun);
            errors += num_overrun;
        }
        printf("\nPass %d Done. with %ld errors\n", pass+1, errors);
        if (ofp)
            fprintf(ofp, "%g\t%ld\t%ld\t%ld\t%u\t%u\n", frame_len, cur_frame, num_overrun,
errors, ave_block, max_block);
        total_errors+= errors;
        pass_info[pass].errors  = errors;
        pass_info[pass].nframes = nframes;
        pass_info[pass].frames_read = cur_frame;
        pass_info[pass].ave_block   = ave_block;
        pass_info[pass].max_block   = max_block;
        if (errors > 0)
        {
            for (chan=0; chan<num_chan; chan++)
                printf("Chan %d : errors = %ld\n", chan, errors_per_chan[chan]);
        }
    }
    if (ofp)
        fclose(ofp);
    if (num_pass > 1)
        printf("All Done. with %ld errors\n", total_errors);
    return EXIT_SUCCESS;

}

/**
```

MODULE DOCUMENTATION

TOP LEVEL FUNCTIONS

- int **xsp3_config** (int ncards, int num_tf, char *baseIPaddress, int basePort, char *baseMACaddress, int num_chan, int create_module, char *modname, int debug, int card_index)
  Configure and initialise the complete xspress3 system.
- int **xsp3_config_init** (int ncards, int num_tf, char *baseIPaddress, int basePort, char *baseMACaddress, int num_chan, int create_module, char *modname, int debug, int card_index, int do_init)
  Configure the complete xspress3 system with flag to specify whether the system in initialised.
- char * **xsp3_get_error_message** ()
  Get the last error message that was generated by the xspress3 system.
- int **xsp3_close** (int path)
  Close all open paths in the system.
- int **xsp3_get_revision** (int path)
  Get firmware revision.
- int **xsp3_get_num_chan** (int path)
  Get the number of channels currently configured in the system.
- int **xsp3_get_num_cards** (int path)
  Get the number of xspress3 cards currently configured in the system.
- int **xsp3_get_chans_per_card** (int path)
  Get the number of channels available per xspress3 card.
- int **xsp3_get_num_chan_used** (int path, int card)
  Get the number of channels used on a givem xspress3 card.
- int **xsp3_get_bins_per_mca** (int path)
  Get the number of bins per MCA configured in the xspress3 system.
- int **xsp3_get_max_num_chan** (int path)
  Get the maximum number of channels currently available in system, auto-sensed at configuration.
- int **xsp3_format_run** (int path, int chan, int aux1_mode, int aux1_thres, int aux2_cont, int flags, int aux2_mode, int nbits_eng)
  Set format of the data to be histogrammed.
- int **xsp3_format_sub_frames** (int path, int chan, int just_good, int just_good_thres, int flags, int nbits_eng, int num_sub_frames, int ts_divide)
  Set format of the data to be histogrammed when working with sub-time-frames calculated from event timestamping The output histogram format is a multi-dimensional space.
- int **xsp3_get_format** (int path, int chan, int *nbins_eng, int *nbins_aux1, int *nbins_aux2, int *nbins_tf)
  Get the format control parameters.

- int **xsp3_read_format** (int path, int chan, int diag_hist, int *num_used_chanP, int *nbins_eng, int *nbins_aux1, int *nbins_aux2, int *nbins_tf)
  Read the format control parameters from the hardware so this will see any changes set by another process.
- int **xsp3_set_window** (int path, int chan, int win, int low, int high)
  Set the scalar windows.
- int **xsp3_set_good_thres** (int path, int chan, u_int32_t good_thres)
  Set the threshold for the good event scaler.
- int **xsp3_get_window** (int path, int chan, int win, u_int32_t *low, u_int32_t *high)
  Get the scaler window settings.
- int **xsp3_set_glob_timeA** (int path, int card, u_int32_t time)
  Set-up the triggering or time framing control register.
- int **xsp3_set_glob_timeFixed** (int path, int card, u_int32_t time)
  Set the fixed time frame register - starting time frame.
- int **xsp3_get_glob_timeA** (int path, int card, u_int32_t *time)
  Get the time framing control register.
- int **xsp3_get_glob_timeFixed** (int path, int card, u_int32_t *time)
  Get fixed time frame register.
- int **xsp3_init_roi** (int path, int chan)
  Initialise the regions of interest.
- int **xsp3_set_roi** (int path, int chan, int num_roi, **XSP3Roi** *roi)
  Sets the regions of interest.
- int **xsp3_get_glob_time_statA** (int path, int card, u_int32_t *time)
  Get the timing status register A.
- int **xsp3_get_disable_threading** (int path)
  Get Disable multi threading flags.
- int **xsp3_set_disable_threading** (int path, int flags)
  Set Disable multi threading flags.
- int **xsp3_read_xadc** (int path, int card, int first, int num, u_int32_t *data)
  Read system monitor function from all generations of Xspress3, so far as supported.
- int **xsp3_set_alt_ttl_out** (int path, int card, int alt_ttl, int force)
  Specify how the TTL LEMOS are used, when not used for board to board and/or Global reset functions.
- int **xsp3_set_timing** (int path, **Xsp3Timing** *setup)
  Setup the time framing control for a whole system.
- int **xsp3_set_trig_in_term** (int path, int card, int flags)
  Set 50 Ohm termination on USER trigger inputs.
- int **xsp3_set_trig_out_term** (int path, int card, int flags)
  Set 50 Ohm series termination on USER trigger outputs.
- int **xsp3_get_trig_in_term** (int path, int card, int *flags)
  Get 50 Ohm termination on USER trigger inputs.

- int **xsp3_get_trig_out_term** (int path, int card, int *flags)
  Get 50 Ohm series termination on USER trigger outputs.
- int **xsp3_clocks_setup** (int path, int card, int clk_src, int flags, int tp_type)
  Set up ADC and data processing clock source.
- int **xsp3_dma_config_memory** (int path, int card, int layout)
  Configure the memory layout connected to the processing FPGA.
- int **xsp3_dma_get_memory_config** (int path, int card)
  Get last known memory configuration layout as set by xsp3_dma_config_memory.
- int **xsp3_set_run_flags** (int path, int flags)
  Set the run mode flags.
- int **xsp3_get_run_flags** (int path)
  Get the run mode flags.
- int **xsp3_scope_wait** (int path, int card)
  This function polls the DMA descriptors for the number of scope DMA engines in use to check whether all the data has been transferred.
- int **xsp3_histogram_start** (int path, int card)
  Start system and enable counting.
- int **xsp3_histogram_arm** (int path, int card)
  Start system ready for histogramming but do not software enable histogramming.
- int **xsp3_histogram_continue** (int path, int card)
  Continue counting from armed or paused when time framing is in software controlled mode.
- int **xsp3_histogram_pause** (int path, int card)
  Pause counting in series of software timed frames.
- int **xsp3_histogram_stop** (int path, int card)
  Stop histogramming.
- int **xsp3_histogram_clear** (int path, int first_chan, int num_chan, int first_frame, int num_frames)
  Clears a section of the histogramming data buffer ready to start histogramming the next experiment.
- int **xsp3_histogram_is_any_busy** (int path)
  Check if any histogarmming thread for this system is busy or waiting for data.
- int **xsp3_histogram_read4d** (int path, u_int32_t *buffer, unsigned eng, unsigned aux, unsigned chan, unsigned tf, unsigned num_eng, unsigned num_aux, unsigned num_chan, unsigned num_tf)
  Read a 4 dimensional block of histogram data.
- int **xsp3_histogram_read3d** (int path, u_int32_t *buffer, unsigned x, unsigned y, unsigned t, unsigned dx, unsigned dy, unsigned dt)
  Read a 3 dimensional block of histogram data with auxiliary and channel combined into "y" to suit xspress3.server.
- int **xsp3_calib_histogram_read3d** (int path, u_int32_t *buffer, unsigned x, unsigned y, unsigned t, unsigned dx, unsigned dy, unsigned dt, int calib)
  Read a 3 dimensional block of histogram data with auxiliary and channel combined into "y" to suit xspress3.server, from either normal or cal_event histograms.

- int **xsp3_histogram_read_chan** (int path, u_int32_t *buffer, unsigned chan, unsigned eng, unsigned aux, unsigned tf, unsigned num_eng, unsigned num_aux, unsigned num_tf)
  Read histogram data from a single channel.
- int **xsp3_histogram_circ_ack** (int path, unsigned chan, unsigned tf, unsigned num_chan, unsigned num_tf)
  Acknowledge finished reading of MCA and scalar data for a given channel and time frame range in circular buffer mode.
- int64_t **xsp3_histogram_get_circ_overrun** (int path, int chan, int64_t *firstP)
  Read how many overrun frames have been detected in circular buffer mode.
- int **xsp3_histogram_get_tf_status** (int path, int chan, unsigned tf, **Xsp3TFStatus** *tf_status)
  Read Time frame status struct especially in circular buffer mode for 1 time frame.
- int **xsp3_histogram_get_tf_status_block** (int path, int chan, unsigned tf, unsigned ntf, **Xsp3TFStatus** *tf_status)
  Read Time frame status struct for several time frames, especially in circular buffer mode.
- int **xsp3_histogram_get_tf_markers** (int path, int chan, unsigned tf, unsigned num_tf, int *markers)
  Read Time frame markers, especially in circular buffer mode.
- int **xsp3_i2c_read_adc_temp** (int path, int card, float *temp)
  Read Temperature from all 4 LM75s on 1 or all ADC boards.
- int **xsp3_i2c_read_mid_plane_temp** (int path, float *temp)
  Read Temperature from all 3 LM75s on the Xspress 4 mid plane.
- int **xsp3_i2c_set_adc_temp_limit** (int path, int card, int critTemp)
  Set over Temperature threshold for all 4 LM75s on 1 or all ADC boards.
- int **xsp3_i2c_read_fem_temp** (int path, int card, float *temp)
  Read Temperature of FEM PCB and Virtex-5 FPGA using LM82s on 1 or all boards.
- int **xsp3_itfg_setup** (int path, int card, int num_tf, u_int32_t col_time, int trig_mode, int gap_mode)
  Setup Internal time frame generator (ITFG) for a series of equal length frames.
- int **xsp3_itfg_setup2** (int path, int card, int num_tf, u_int32_t col_time, int trig_mode, int gap_mode, int acq_in_pause, int marker_period, int marker_frame)
  Setup Internal time frame generator (TFG) for a series of equal length frames, with access to generation 2 features.
- int **xsp3_itfg_get_setup** (int path, int card, int *num_tf, u_int32_t *col_time, int *trig_mode, int *gap_mode)
  Retrieve settings of the Internal time frame generator (TFG).
- int **xsp3_itfg_stop** (int path, int card)
  Stop the Internal time frame generator (TFG).
- int **xsp3_itfg_start** (int path, int card)
  Re-start the Internal time frame generator (TFG).
- int **xsp3_save_settings** (int path, char *dir_name)
  Save all xspress3 settings into a series of files.
- int **xsp3_restore_settings_and_clock** (int path, char *dir_name, int force_mismatch)
  Restore all xspress3 settings from series of files including configuring the clocks.
- int **xsp3_restore_settings** (int path, char *dir_name, int force_mismatch)

Restore all xspress3 settings from series of files.

- int **xsp3_scaler_get_num_tf** (int path)
  Read the maximum number of time frames of scaler data that can be stored with the current memory allocation within the FEM.
- int **xsp3_scaler_read** (int path, u_int32_t *dest, unsigned scaler, unsigned chan, unsigned t, unsigned n_scalers, unsigned n_chan, unsigned dt)
  Read a block of scaler data.
- int **xsp3_scaler_check_progress** (int path)
  This function checks how many time frames have been processed.
- int64_t **xsp3_scaler_check_progress_details** (int path, **Xsp3ErrFlag** *flagsP, int quiet, int64_t *furthest_frame)
  This function checks how many time frames have been processed.
- int **xsp3_scaler_check_desc** (int path, int card)
  This function checks the DMA descriptors for the Scaler stream to check how many time frames of scalers have been successfully transferred to memory.
- int **xsp3_scaler_read_sf** (int path, u_int32_t *dest, unsigned scaler, unsigned first_sf, unsigned chan, unsigned t, unsigned n_scalers, unsigned n_sf, unsigned n_chan, unsigned dt)
  Read a block of scaler data with sub-frames feature.
- int **xsp3_sys_log_start** (int path, char *fname, int period, int max_count, **Xsp3SysLogFlags** flags)
  Start the temperature and FPGA supply voltage logging thread.
- int **xsp3_sys_log_continue** (int path)
  *Restart the temperature and FPGA supply voltage logging thread after it has been paused with **xsp3_sys_log_pause()**.*
- int **xsp3_sys_log_stop** (int path)
  Stop the temperature and FPGA supply voltage logging thread, close files.
- int **xsp3_sys_log_pause** (int path)
  *Pause the temperature and FPGA supply voltage logging thread stopping the thread but keeping the file open to allow **xsp3_sys_log_continue()***
- int **xsp3_read_scope_data** (int path, int card)
  Read scope data from 10G Ethernet using UDP protocol for Xspress 3/4 and 1G Ethernet for Xspress 3 Mini This is the common API for all generations.
- int **xsp3_write_playback_data** (int path, int card, u_int32_t *buffer, size_t nbytes, int no_retry)
  Write playback data into DRAM via the 10G Ethernet link or 1G Ethernet.
- int **xsp3_diag_histogram_read3d** (int path, u_int32_t *buffer, unsigned x, unsigned y, unsigned t, unsigned dx, unsigned dy, unsigned dt)
  Read a 3 dimensional block of diagnostic histogram data with aux and chan combined into "y" to suit xspress3.server.

FUNCTION DOCUMENTATION

int xsp3_calib_histogram_read3d (int *path*, u_int32_t * *buffer*, unsigned *x*, unsigned *y*, unsigned *t*, unsigned *dx*, unsigned *dy*, unsigned *dt*, int *calib*)

Read a 3 dimensional block of histogram data with auxiliary and channel combined into "y" to suit xspress3.server, from either normal or cal_event histograms.

Supported on all generations, for Xspress 3 Mini the normal mode BRAM histogram do not have any auxiliary data, so this calls xsp3m_histogram_read_frames, to read the data from the Mini over the 1G Ethernet link..

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
buffer	A pointer to the buffer for the returned data
x	Starting energy bin.
y	Channel and aux combined, with incrementing addresses covering aux and then channel
t	Starting time frame
dx	Number of energy bins.
dy	Number of the combined channel & aux
dt	Number of time frames
calib	1 => Access cal_event histograms0 => Access normal histogram

RETURNS:

XSP3_OK or a negative error code.

int xsp3_clocks_setup (int *path*, int *card*, int *clk_src*, int *flags*, int *tp_type*)

Set up ADC and data processing clock source.

The clock source can come from various places. At power up and from some debugging the clock source can be from the FPGA board clocks. However, for normal operation the ADC and data processing clock has to be from the ADC board. This is setup by this command. It should be done after xsp3_config, but before anything else. Setup for each generation is different, but this function front all 3 worker functions. Note that **xsp3_restore_settings** cannot restore the clock configuration. However **xsp3_restore_settings_and_clock()** can be used.

following definitions describe the valid values for the parameter clk_src

```
#define XSP3_CLK_SRC_INT                0       //!< channel processing clock comes from fpga
processor (testing only)
#define XSP3_CLK_SRC_XTAL               1       //!< adc and channel processing clock from
crystal on the ADC board (normal single board or master operation).
#define XSP3_CLK_SRC_EXT                2       //!< adc and channel processing clock from
lemo clock connector on ADC board (slave boards)
#define XSP3_CLK_SRC_FPGA               3       //!< not implemented, for future expansion

#define XSP3M_CLK_SRC_CDCM61004         0x10    //!< Xspress 3 Mini or Xspress 4 Clock Source
CDCM61004 on ADC board
#define XSP3M_CLK_SRC_LMK61E2           0x11    //!< Xspress 3 Mini or Xspress 4 Clock Source
LMK61E2 on ADC board

#define XSP4_CLK_SRC_MIDPLN_CDCM61004   0x20    //!< Xspress 4 Clock Source CDCM61004 on Mid
plane board
#define XSP4_CLK_SRC_MIDPLN_LMK61E2     0x21    //!< Xspress 4 Clock Source LMK61E2 on Mid
Plane board
```

The following definitions describe the valid values for flags which are or'ed together to make the flags parameter.

```
#define XSP3_CLK_FLAGS_MASTER       (1<<0)      // this clock generate clocks for other
boards in the system
#define XSP3_CLK_FLAGS_NO_DITHER    (1<<1)      // disables dither within the ADC
#define XSP3_CLK_FLAGS_STAGE1_ONLY  (1<<2)      // performs stage of the lmk 03200 setup,
does not enable zero delay mode
#define XSP3_CLK_FLAGS_NO_CHECK     (1<<3)      // don't check for lock detect from lmk 03200
#define XSP3_CLK_FLAGS_TP_ENB       (1<<4)      // enable test pattern
#define XSP3_CLK_FLAGS_DIS_OVER_TEMP (1<<5)     // Disable Over temperature protection on ADC
Board
#define XSP3_CLK_FLAGS_SHUTDOWN0    (1<<6)      // Shutdown ADC channel 0
#define XSP3_CLK_FLAGS_SHUTDOWN123  (1<<7)      // Shutdown ADC channels 123
#define XSP3_CLK_FLAGS_SHUTDOWN4    (1<<8)      // Shutdown ADC channel 4 (middle (unused?))
#define XSP3_CLK_FLAGS_SHUTDOWN5678 (1<<9)      // Shutdown ADC channel5678 last 4
#define XSP3_CLK_FLAGS_SHUTDOWN1    (1<<10)     // Shutdown ADC channel 1  Xspress 3 Mini and
Xspress 4
#define XSP3_CLK_FLAGS_SHUTDOWN2    (1<<11)     // Shutdown ADC channel 2  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN3    (1<<12)     // Shutdown ADC channel 3  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN5    (1<<13)     // Shutdown ADC channel 5  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN6    (1<<14)     // Shutdown ADC channel 6  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN7    (1<<15)     // Shutdown ADC channel 7  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN8    (1<<16)     // Shutdown ADC channel 8  Xspress 4 only
#define XSP3_CLK_FLAGS_SHUTDOWN9    (1<<17)     // Shutdown ADC channel 9  Xspress 4 only

#define XSP3_CLK_FLAGS_TS_SYNC      (1<<30)     // Restart time stamp on Run on card 0
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the cards in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card is less than 0 then all cards are selected.
clk_src	Clock source
flags	Setup flags (see definition)
tp_type	Test pattern type in the IO spartan FPGAs, Xspress 3 only.

RETURNS:

0 for success (Xspress 3), Clock frequency (Xspress 3 mini and Xspress 4) or a negative error code.

int xsp3_close (int *path*)

Close all open paths in the system.

Currently full details of tidying various shared objects and threads are not fully supported so this function is a place holder for future work.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_config (int *ncards*, int *num_tf*, char * *baseIPaddress*, int *basePort*, char * *baseMACaddress*, int *num_chan*, int *create_module*, char * *modname*, int *debug*, int *card_index*)

Configure and initialise the complete xspress3 system. This is the normal configuration entry point for Xspress 3. The number of cards and number of time frames parameters must be specified. The debug and create_module parameters are boolean and should be specified as 0 or 1. All other parameters can be used as default by specifying either NULL or -1.

The functions checks if the parameters of this configuration have been previously used. If so the system is re-initialised and the existing path is returned. Otherwise if it's a unique set of cards to configure allow the configuration. If a different debug flag is set for the case where the same parameters are used this may overwrite another connections debug preference.

This function normally creates all the shared memory objects and starts the histogramming thread(s).

```
* Using the default baseIPaddress, for the host server, of 192.168.0.1 the following values
are calculated for the complete xspress3 system.
*
* Xspress 3 1Gb tcp IP address      card 1      192.168.0.2
*                                   card 2      192.168.0.3
*                                   card 3      192.168.0.4
*                                   ...
*
* host 10Gb udp IP address          card 1      192.168.0.65
*                                   card 2      192.168.0.69
*                                   card 3      192.168.0.73
*                                   ...
*
* xspress3 10Gb udp IP address      card 1      192.168.0.66
*                                   card 2      192.168.0.70
*                                   card 3      192.168.0.74
*                                   ...
*
* Using the default MAC address of 02.00.00.00.00.00 the following values are calculated for
the complete xspress3 system.
*
* Xspress 3 MAC address             card 1      02.00.00.00.00.00
*                                   card 2      02.00.00.00.00.02
*                                   card 3      02.00.00.00.00.04
*                                   ...
*
* For all Xspress 3 cards the default basePort for the tcp connection is 30123 and for the
udp connection 30124.
* For Xspress 3 mini and Xspress 4 the tcp port default is 30124 (TCP). By default the
library tries both to determine if it is talking to an Xspress 3 or Xspress 3 mini/Xspress 4
*
```

PARAMETERS:

ncards	The number of xspress3 cards that constitute the xspress3 system, between 1 and **XSP3_MAX_CARDS**.
num_tf	The maximum number of time frames when used as 4096 energy bins and no auxiliary data. If the memory is arranged differently (using ROI or Auxiliary data) then the actual number of time frames that will fit can go up or down
baseIPaddress	The IP address of the host server in dotted quad format (e.g. 192.168.0.1) from which all other addresses are calculated or NULL to use the default
basePort	The IP port number or -1 to use the default
baseMACaddress	The base MAC address (e.g. 02.00.00.00.00) for card 1, from which all other card MAC addresses are calculated or NULL to use the default
num_chan	The number of channels to be configured in the xspress3 system. Between 1 and (ncards * **XSP3_MAX_CHANS_PER_CARD**) or -1 to configure all available channels.
create_module	A boolean flag to determine whether to create the scope mode data module. It usual to enable this, though some memory may be saved by not doing this.
modname	The module name for the scope mode data module or NULL to use the default. Using the default is recommended.
debug	An integer flag to turn debug messages off/on/verbose.
card_index	Starting card number. i.e. 0 = card1, 1 = card2 etc. -1 assumes card_index 0)

RETURNS:

a handle to the top level of the xspress3 system or a negative error code.

int xsp3_config_init (int *ncards*, int *num_tf*, char * *baseIPaddress*, int *basePort*, char * *baseMACaddress*, int *num_chan*, int *create_module*, char * *modname*, int *debug*, int *card_index*, int *do_init*)

Configure the complete Xspress 3 system with flag to specify whether the system in initialised.

This configuration entry point is used by imgd to monitor Xspress 3 without overwriting the initialisation.

PARAMETERS:

ncards	the number of xspress3 cards that constitute the xspress3 system, between 1 and **XSP3_MAX_CARDS**.
num_tf	the maximum number of time frames when used as 4096 energy bins and no auxiliary data. If the memory is arranged differently (using RoI or Auxiliary data) then the actual number of tf that will fit can go up or down
baseIPaddress	the IP address of the host server in dotted quad format (e.g. 192.168.0.1) from which all other address's are calculated or NULL to use the default
basePort	the IP port number or -1 to use the default
baseMACaddress	The base MAC address (e.g. 02.00.00.00.00) for card 0, from which all other card MAC address's are calculated or NULL to use the default Specify 00.00.00.00.00.00 to disable UDP connection ot this instance, to co-exist with other libxspress3 instance in xspress3.server, EPICS etc.
num_chan	The number of channels to be configured in the xspress3 system. between 1 and (ncards * **XSP3_MAX_CHANS_PER_CARD**) or -1 to configure all available channels.
create_module	A boolean flag to determine whether to create the scope mode data module. It usual to enable this, though some memory may be saved by not doing this.
modname	The module name for the scope mode data module or NULL to use the default. Using the default is recommended.
debug	An integer flag to turn debug messages off/on/verbose.

card_index	Starting card number ie 0 = card1, 1 = card2 etc. -1 assumes card_index 0)
do_init	A boolean flag to specify whether the FPGA registers and BRAMS are initialised. Usually set, but cleared if a second progress (e.g. imgd) is to view data without overwriting a setup done by xspress3.server or EPICS.

RETURNS:

a handle to the top level of the xspress3 system or a negative error code.

int xsp3_diag_histogram_read3d (int *path*, u_int32_t * *buffer*, unsigned *x*, unsigned *y*, unsigned *t*, unsigned *dx*, unsigned *dy*, unsigned *dt*)

Read a 3 dimensional block of diagnostic histogram data with aux and chan combined into "y" to suit xspress3.server.

Only supported on Xspress 3 Mini.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
buffer	a pointer to the buffer for the returned data
x	is the start energy
y	is the channel and aux combined, converting aux and then channel
t	is the start time frame
dx	is the number of energies
dy	is the number of the combined channel & aux
dt	is the number of time frames

RETURNS:

XSP3_OK or a negative error code.

int xsp3_dma_config_memory (int *path*, int *card*, int *layout*)

Configure the memory layout connected to the processing FPGA

At present for Xspress 3 it is a fixed layout so there is little point calling this. For Xspress 3 Mini this is used much more. The system can use the memory for 1 or 2 channels and firmware regions of interest can be used. With changes to the number of channels and the size of each spectrum if ROI are used, the software adjusts the memory usage between spectra, scalars and DMA descriptors so the number of time frames of spectra and scalars match. the number of channels is when this is called as part of **xsp3_config()** and the size of each spectra is set when this is called as part of **xsp3_format_run()**. Normally user code will not need to call this directly. For Xspress 3 Mini more detailed control of the layout has been schemed, but is not fully tested. For the options considered see:

```
#define XSP3_CONF_MEM_OVERALL(x)             ((x)&0xFF)
#define XSP3_CONF_MEM_GET_OVERALL(x)         ((x)&0xFF)

#define XSP3_CONF_MEM_PLAYBACK_SCOPE(x)      (((x)&0x3F)<<8)
#define XSP3_CONF_MEM_GET_PLAYBACK_SCOPE(x)  (((x)>>8)&0x3F)

#define XSP3_CONF_MEM_HIST_OPTIONS(x)        (((x)&0x3F)<<14)
#define XSP3_CONF_MEM_GET_HIST_OPTIONS(x)    (((x)>>14)&0x3F)
#define XSP3_CONF_MEM_HIST_FRAME_SCALARS     0
#define XSP3_CONF_MEM_HIST_ALL_SCALARS       1
#define XSP3_CONF_MEM_HIST_10FRAME_DIAG      4
#define XSP3_CONF_MEM_HIST_ALL_DIAG          5

#define XSP3_CONF_MEM_NBITS_ENG(x)           (((x)&0xF)<<20)
#define XSP3_CONF_MEM_GET_NBITS_ENG(x)       (((x)>>20)&0xF)

#define XSP3_CONF_MEM_NUM_CHAN(x)            (((x)&0x3F)<<24)
#define XSP3_CONF_MEM_GET_NUM_CHAN(x)        (((x)>>24)&0x3F)
```

Xspress 4 currently two layouts are supported. Layout 0 is used for all normal operation. Layout 1 uses a very restricted part of the DRAM for low level testing only.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, Specify card = -1 to apply the config to all cards in a system.
layout	0 for default layout, otherwise depends on generation.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_dma_get_memory_config (int *path*, int *card*)

Get last known memory configuration layout as set by xsp3_dma_config_memory.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number 0..num_cards-1 in system.

RETURNS:

Memory layout for current xspress system or -1 on error.

int xsp3_format_run (int *path*, int *chan*, int *aux1_mode*, int *aux1_thres*, int *aux2_cont*, int *flags*, int *aux2_mode*, int *nbits_eng*)

Set format of the data to be histogrammed.

The output histogram format is a multi-dimensional space. first dimension is the photon energy. second dimension (if present) is the auxiliary data 1 aux1. third dimension (if present) is the auxiliary data 2 aux2. fourth dimension is the time frame.

The following definitions describe the aux1 mode options

```
#define XSP3_FORMAT_AUX1_MODE_NONE          0       //!< No Auxilliary data 1
#define XSP3_FORMAT_AUX1_MODE_PILEUP        7       //!< 1 bit of Aux1, set when pileup
detected for Width or hardware
#define XSP3_FORMAT_AUX1_MODE_CSHR          10      //!< 4 or 6 bits showing the 4 or 6
CShare status bit.
#define XSP3_FORMAT_AUX1_MODE_CSHR_ENC      11      //!< 3 bit 0..5 showing the priority
encoded CShare status bits, otherwise coded 7 is "good"
#define XSP3_FORMAT_AUX1_MODE_GOOD_GRADE    15      //!< 0 bits, but Masks MCA (and scalers)
for events with min < thres
#define XSP3_FORMAT_AUX1_MODE_TIMESTAMPED   100     //!< 0 bits, Auxiliary data is replaced
by timestamp. Not written to hardware.
```

The following definitions describe the aux2 mode options

```
#define XSP3_FORMAT_AUX2_ADC                    0
#define XSP3_FORMAT_AUX2_WIDTH                  1
#define XSP3_FORMAT_AUX2_RST_START_ADC          2
#define XSP3_FORMAT_AUX2_NEB_RST_ADC            3
#define XSP3_FORMAT_AUX2_NEB_RST_RST_START_ADC  4
#define XSP3_FORMAT_AUX2_TIME_FROM_RST          5
#define XSP3_FORMAT_AUX2_NEB_RST_TIME_FROM_RST  6
#define XSP3_FORMAT_AUX2_NEB_RST_TIME_FROM_RSTX8 7
```

The definition of the flag bits are shared with the format register. Currently the only bit which can be set here is XSP3_FORMAT_PILEUP_REJECT. If set, events which are identified as pileup from either the

trigger hardware or by event width (see **xsp3_build_pileup_time**) are discarded from the in-window scalers and MCA spectra.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) if chan is less than 0 then all channels are selected.
aux1_mode	Special debug mode (set to 0 for user operation)
aux1_thres	Threshold special debug value (set to 0 for user operation)
aux2_cont	Set bits for ADC range (set to 0 for user operation)
flags	Flags for pileup rejection (set to 0 for user operation) and can include charge sharing mode.
aux2_mode	Set to 0 for user operation.
nbits_eng	Number bits of energy (default 12)

RETURNS:

The number of time frames of the requested configuration that fits into memory.

int xsp3_format_sub_frames (int *path*, int *chan*, int *just_good*, int *just_good_thres*, int *flags*, int *nbits_eng*, int *num_sub_frames*, int *ts_divide*)

Set format of the data to be histogrammed when working with sub-time-frames calculated from event timestamping the output histogram format is a multidimensional space.

The first dimension is the photon energy, second is the calculated subtime-frame, third dimension is the time frame.

The definition of the flag bits are shared with the format register. Currently the only bit which can be set here is XSP3_FORMAT_PILEUP_REJECT. If set, events which are identified as pileup from either the trigger hardware or by event width (see **xsp3_build_pileup_time**) are discarded from the in-window scalers and MCA spectra.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) if chan is less than 0 then all channels are selected.
just_good	Keep (in MCA) just events with specified minimum number of samples.
just_good_thres	Threshold for minimum number of samples to keep in just_good mode
flags	Flags for pileup rejection (set to 0 for user operation), can include cshare mode
nbits_eng	Number bits of energy (default 12)
num_sub_frames	Number of subframes in sub time frame mode (< 1 to disable)
ts_divide	Divisor from time stamp to sub frame number in sub-frame mode.

RETURNS:

The number of top level time frames of the requested configuration that fits into memory, each one contains sub_frames addressed using aux in **xsp3_histogram_read4d()** etc.

int xsp3_get_bins_per_mca (int *path*)

Get the number of bins per MCA configured in the xspress3 system.

The correct value is stored in the top level path of the system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the number of bins per MCA for the current configuration of the xspress3 system or a negative error code.

int xsp3_get_chans_per_card (int *path*)

Get the number of channels available per xspress3 card.

The correct value is stored in the top level path of the system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the number of channels available per xspress3 card or a negative error code.

int xsp3_get_disable_threading (int *path*)

Get disable multi threading flags.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the disable multi threading flags. See **XSP3_MT_FLAGS** or a negative error code.

char* xsp3_get_error_message ()

Get the last error message that was generated by the xspress3 system.

Whenever a function returns an error code a detailed error message is sprintfed into the error_message buffer. It can be retrieved using this call. The maximum length of this message is **XSP3_MAX_MSG_LEN**.

RETURNS:

A pointer to the statically allocated error message, which will remain valid only until the next error occurs.

int xsp3_get_format (int *path*, int *chan*, int * *nbins_eng*, int * *nbins_aux1*, int * *nbins_aux2*, int * *nbins_tf*)

Get the format control parameters.

Retrieves the parameters set by **xsp3_format_run**. These are retrieved from the current Xsp3Sys data structures, so will not reflect changes made by another program. To force these values to be read from the hardware use **xsp3_read_format()**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
nbins_eng	Pointer to return the number of energy bins -usually 4096 unless RoI are in use.
nbins_aux1	Pointer to return the number of aux1 bins - usually 1.
nbins_aux2	Pointer to return the number of aux2 bins - usually 1.
nbins_tf	Pointer to return the number of time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_glob_time_statA (int *path*, int *card*, u_int32_t * *time*)

Get the timing status register A.

following definitions describe the layout of the global time A register

```
#define XSP3_GLOB_TSTAT_A_FRAME(x)          (((x)>>0)&0xFFFFFF) //!< Read the current frame
number value.
#define XSP3_GLOB_TSTAT_A_ITFG_COUNTING(x)  (((x)>>24)&0x1)     //!< Read whether the ITFG is
currently enabling counting
#define XSP3_GLOB_TSTAT_A_ITFG_RUNNING(x)   (((x)>>25)&0x1)     //!< Read whether the ITFG is
currently running.
#define XSP3_GLOB_TSTAT_A_ITFG_PAUSED(x)    (((x)>>26)&0x1)     //!< Read whether the ITFG is
paused wait for software or hardware trigger
#define XSP3_GLOB_TSTAT_A_ITFG_FINISHED(x)  (((x)>>27)&0x1)     //!< Read whether the ITFG
has finished a run. This clears when the bit XSP3_GLOB_TIMA_RUN is negated using {@link
xsp3_histogram_stop()}.
```

Reading this register allows the current time frame to be followed. However processing of data may lag the time frame slightly. To see how much data is ready to read use **xsp3_scaler_check_progress_details()**. register also allows the Internal TFG status to be read and is the correct way to determine the ITFG status.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
time	Pointer to return the timing status a register. See **XSP3_GLOBAL_TIME_STATUS_A**

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_glob_timeA (int *path*, int *card*, u_int32_t * *time*)

Get the time framing control register.

The following definitions describe the layout of the global time A register

```
#define XSP3_GLOB_TIMA_TF_SRC(x)        ((x)&7)     //!< Sets Time frame info source see
XSP3_GTIMA_SRC_*.
#define XSP3_GLOB_TIMA_F0_INV           (1<<3)      //!< Invert Frame Zero signal polarity to
make signal active low, resets time frame when sampled low by leading edge of Veto.
#define XSP3_GLOB_TIMA_VETO_INV         (1<<4)      //!< Invert Veto signal polarity to make
signal active low, counts when Veto input is low.
#define XSP3_GLOB_TIMA_ENB_SCALER       (1<<5)      //!< Enables scalers.
#define XSP3_GLOB_TIMA_ENB_HIST         (1<<6)      //!< Enables histogramming.
#define XSP3_GLOB_TIMA_LOOP_IO          (1<<7)      //!< Loop TTL_IN(0..3) to TTL_OUT(0..3)
for hardware testing (only).
#define XSP3_GLOB_TIMA_NUM_SCAL_CHAN(x) (((x)&0xF)<<8)  //!< Sets the number of channels of
scalers to be transferred to memory by the DMA per time frame.
#define XSP3_GLOB_TIMA_SW_MARKERS(x)    (((x)&3)<<12)   //!< Set Software markers for use
(testing) in circular buffer mode.
#define XSP3_GLOB_TIMA_FRAME_CAPTURE    (1<<14)         //!< Enable Frame Capture mode in
circular buffer mode.
#define XSP3_GLOB_TIMA_FROM_RADIAL      (1<<15)         //!< Enable from Radial trigger
signals for Xspress 4 with backplane builds.
#define XSP3_GLOB_TIMA_DEBOUNCE(x)      (((x)&0xFF)<<16)    //!< Set debounce time in 80 MHz
cycles to ignore glitches or ringing on Frame0 or Framing signal from any source.
#define XSP3_GLOB_TIMA_ALT_TTL(x)       (((x)&0xF)<<24)     //!< Alternate uses of the TTL
Outputs (including channel in windows signals etc).
#define XSP3_GLOB_TIMA_RUN              (1<<31)     //!< Overall Run enable signal, set after
all DMA channels have been configured.
#define XSP3_GLOB_TIMA_PB_RST           (1<<30)     //!< Resets Playback FIFO as part of
clean start.
#define XSP3_GLOB_TIMA_COUNT_ENB        (1<<29)     //!< In software timing
(XSP3_GTIMA_SRC_FIXED) mode enable counting when high. Transfers scalers on falling edge.
After first frame, increments time frame on rising edge.
#define XSP3_GLOB_TIMA_ITFG_RUN         (1<<28)     //!< From versions 11/8/2014 onwards this
is a separate Run signal to the internal time frame generator, which could be used to crash
stop the ITFG before stopping the rest.
#define XSP3_GLOB_TIMA_GET_TF_SRC(x)    ((x)&7)     //!< Sets Time frame info source see
XSP3_GTIMA_SRC_*.
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
time	Pointer to return the time framing control register

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_glob_timeFixed (int *path*, int *card*, u_int32_t * *time*)

Get fixed time frame register.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
time	fixed Pointer to return time frame register

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_max_num_chan (int *path*)

Get the maximum number of channels currently available in system, auto-sensed at configuration.

The correct value is stored in the top level path of the system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the maximum number of channels for the current configuration of the xspress3 system or a negative error code.

int xsp3_get_num_cards (int *path*)

Get the number of xspress3 cards currently configured in the system.

The correct value is stored in the top level path of the system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the number of cards currently configured in the xspress3 system or a negative error code.

int xsp3_get_num_chan (int *path*)

Get the number of channels currently configured in the system.

This value is passed into the **xsp3_config()** routine or the number of channels auto-sensed . The correct value is stored in the top level path of the system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the number of configured channels in the xspress3 system or a negative error code.

int xsp3_get_num_chan_used (int *path*, int *card*)

Get the number of channels used on a given xspress3 card.

The value comes from each leaf in a composite system.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number 0..num_cards-1 (0 in a single card system)

RETURNS:

the number of channels available per Xspress 3 card or a negative error code.

int xsp3_get_revision (int *path*)

Get firmware revision.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the revision or a negative error code.

int xsp3_get_run_flags (int *path*)

Get the run mode flags.

Retrieve run mode flags as set by **xsp3_set_run_flags**.

following definitions describe the valid values for the stream

```
#define XSP3_RUN_FLAGS_PLAYBACK 1       //!< Enable build of descriptors and start of DMA for
Playback DMA
#define XSP3_RUN_FLAGS_SCOPE    2       //!< Enable build of descriptors and start of DMA for
Scope Mode DMA(s)
#define XSP3_RUN_FLAGS_SCALERS  4
#define XSP3_RUN_FLAGS_HIST     8
#define XSP3_RUN_FLAGS_DIAG_HIST 0x10
```

```
#define XSP3_RUN_FLAGS_CIRCULAR_BUFFER 0x100    //!< Enable recircualting buffer mode.
```

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the value of run flags

int xsp3_get_trig_in_term (int *path*, int *card*, int * *flags*)

Get 50 Ohm termination on user trigger inputs.

This is supported on Xspress 3 Mini and Xspress 4. It is reasonable that user code would set this to suit the triggering requirements of a given experiment. For Xspress 3 Mini there are only two user TTL inputs controlled by flag bits 0 and 1. For Xspress 4 there are 4 user TTL inputs controlled by flag bits 0 to 3.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number of -1 for all. However, for XSpress4 this is usually card 0. However passing card = -1 will access card 0 only for mid-plane based Xspress4 systems.
flags	Bitwise mask 1 to enable termination resistor, otherwise Hi-Z

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_trig_out_term (int *path*, int *card*, int * *flags*)

Get 50 Ohm series termination on user trigger outputs.

This is supported on Xspress 3 Mini and Xspress 4 if the hardware requires it. It is the reasonable that user code would set this to suit the triggering requirements of a given experiment. However this restored by xsp3_restore_settings in case the signals are used for daisy-chained veto or global reset For Xspress 3 Mini there are only two user TTL inputs controlled by flag bits 0 and 1. For Xspress 4 the are 4 User TTL outputs controlled by bits 0..3 of flags. Outputs 2 and 3 are also output as LVDS.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number of -1 for all.
flags	Bitwise mask 1 to enable series termination resistor, 0 => direct drive.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_get_window (int *path*, int *chan*, int *win*, u_int32_t * *low*, u_int32_t * *high*)

Get the scaler window settings.

Retrieve the in-window scalers limits as set by **xsp3_set_window**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
win	The window (0 or 1).
low	Pointer to return low window threshold.

high	Pointer to return high window threshold.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_arm (int *path*, int *card*)

Start system ready for histogramming but do not software enable histogramming.

For non-farm mode UDP core versions the UDP port and frame numbers are reset as the scope mode also uses the UDP connection on a different port. In software timing mode, counting is left disabled, start with **xsp3_histogram_continue()** In ITFG Mode, the ITFG gets a run signal but no CountEnb. If trigger mode = XSP3_ITFG_TRIG_MODE_SOFTWARE or XSP3_ITFG_TRIG_MODE_SOFTWARE_ONLY_FIRST the system will immediately pause. Start with **xsp3_histogram_continue()**

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	is the number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. card == -1 causes all cards in a multi card system to start.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_circ_ack (int *path*, unsigned *chan*, unsigned *tf*, unsigned *num_chan*, unsigned *num_tf*)

Acknowledge finished reading of MCA and scalar data for a given channel and time frame range in circular buffer mode.

For Xspress 3 and Xspress 4 the state words in the xsp3_circ_buff<sys> data module are cleared to release the time frames. This is purely a status module. Not clearing this will not stop data being collected, but the status flags returned by **xsp3_scaler_check_progress_details** will not be correct if this is not used. However, for Xspress 3 Mini the state in within the DMA descriptors and clearing the

complete bit to re-use the descriptors is compulsory. For compatibility the user code should use this function. Also note that in Xspress 3 Mini the histogram and frame descriptors are shared between frames on a card. Clearing either channel on a given card will acknowledge BOTH (all) channels on that card. The user code must be sure that both channels on a card have been read out before acknowledging either of them. Calling **xsp3_histogram_circ_ack()** with num_chan >1 will correctly acknowledge each card with channels in the specified range. However calling **xsp3_histogram_circ_ack()** twice to acknowledge the same frames on 2 channels on the same card will give an error at the 2nd call, to warn the developer of these features.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	Is the number of the first channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1).
tf	Starting time frame
num_chan	Number of channels to mark as read.
num_tf	Number of time frames

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_clear (int *path*, int *first_chan*, int *num_chan*, int *first_frame*, int *num_frames*)

Clears a section of the histogramming data buffer ready to start histogramming the next experiment.

For Xspress 3 and Xspress 4 this is mandatory. For Xspress 3 Mini it can be omitted, however, it is easier to see triggering problems if this is used.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
first_chan	First channel to start clearing from.

num_chan	Number of channels to clear.
first_frame	First frame number to start clearing from.
num_frames	Number of frames to clear.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_continue (int *path*, int *card*)

Continue counting from armed or paused when time framing is in software controlled mode.

The System must have been armed first with **xsp3_histogram_start()** or **xsp3_histogram_arm()** or paused with **xsp3_histogram_pause()**. When using the ITFG in XSP3_ITFG_TRIG_MODE_SOFTWARE or XSP3_ITFG_TRIG_MODE_SOFTWARE_ONLY_FIRST modes, use this to continue from armed. To trigger the ITFG from subsequent pauses, call **xsp3_histogram_pause()** then **xsp3_histogram_continue()** to negate and assert CountEnb.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card is less than 0 then all cards are selected. However, generally cards will be daisy-chained so this should only be necessary on card 0.

RETURNS:

XSP3_OK or a negative error code.

int64_t xsp3_histogram_get_circ_overrun (int *path*, int *chan*, int64_t * *firstP*)

Read how many overrun frames have been detected in circular buffer mode.

If chan specifies one channel, the number of overruns are reported. If chan = -1 to specify all channels, the code scans all channels and reports the first dropped frame and the worst number of dropped frames. For Xspress3Mini the system stops at the first over-run. xsp3_histogram_get_circ_overrun returns 1 in this case, though many more frames may have been lost.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	Channel number in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) or -1 to scan all channels
firstP	Pointer to return the first overrun frame. Specify NULL to just return the number of over run frames.

RETURNS:

Number of overrun frames, 0 if none lost or a negative error code.

int xsp3_histogram_get_tf_markers (int *path*, int *chan*, unsigned *tf*, unsigned *num_tf*, int * *markers*)

Read Time frame markers, especially in circular buffer mode.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	is the number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1). In normal operation all channels should give the same information, so reading just channel 0 should surfice.
tf	First time frame. In circular mode this will be wrapped round to use the circular buffer.

num_tf	Number of time frames. In circular mode this will be wrapped round to uwe the circular buffer.
markers	Pointer to buffer to return an array of ints to return the time frame markers.

RETURNS:

0 on success or a negative error code.

int xsp3_histogram_get_tf_status (int *path*, int *chan*, unsigned *tf*, Xsp3TFStatus * *tf_status*)

Read Time frame status struct especially in circular buffer mode for 1 time frame.

Use **xsp3_histogram_get_tf_status_block** for a more efficient way to read status for many time frames, especially for Xspress3Mini. For a faster way to read just the markers consider **xsp3_histogram_get_tf_markers**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1). In normal operation all channels should give the same information, so reading just channel 0 should suffice.
tf	Time frame. In circular mode this will be wrapped round to use the circular buffer.
tf_status	Pointer to **Xsp3TFStatus** to return the time frame status.

RETURNS:

0 on success or a negative error code.

int xsp3_histogram_get_tf_status_block (int *path*, int *chan*, unsigned *tf*, unsigned *ntf*, Xsp3TFStatus * *tf_status*)

Read Time frame status struct for several time frames, especially in circular buffer mode.

This is the preferred mode for accessing multiple time frames, particularly for Xspress 3 Mini. For a faster way to read just the markers consider **xsp3_histogram_get_tf_markers**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1). In normal operation all channels should give the same information, so reading just channel 0 should suffice.
tf	Time frame. In circular mode this will be wrapped round to use the circular buffer.
ntf	Number of time frames of status to read
tf_status	Pointer to **Xsp3TFStatus** to return the time frame status.

RETURNS:

0 on success or a negative error code.

int xsp3_histogram_is_any_busy (int *path*)

Check if any histogramming thread for this system is busy or waiting for data.

Usually the histogramming thread will keep up with the list of events and will idle in a blocking read. If the system is used at high rate and CPU load or page faults stall the histogram thread may take some time to finish processing data from the network stack buffers. To safely stop the system, the supply of histogram data should be stopped either by the external timing generator negating the count enable input or by calling **xsp3_histogram_stop()** to negate the Run bit in the timing control register. After this the calling program should repeat sleeping 10 ms and checking this busy flag until it is seen to be not busy twice on the run. For the current XSP3_FEATURE_OUTPUT_FORMAT_HEIGHTS64 variants using **xsp3_scaler_check_progress_details()** is preferred.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

1 if busy 0 idle or a negative error code.

int xsp3_histogram_pause (int *path*, int *card*)

Pause counting in series of software timed frames.

The next **xsp3_histogram_continue** will increment time frame and continue counting. This leaves DMAs running ready for next frame. The System must have started first with **xsp3_histogram_start()** This is also used to negate CountEnb before calling xsp3_histogram_continue when using the ITFG with software triggering.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card is less than 0 then all cards are selected. However, generally cards will be daisy-chained so this should only be necessary on card 0.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_read3d (int *path*, u_int32_t * *buffer*, unsigned *x*, unsigned *y*, unsigned *t*, unsigned *dx*, unsigned *dy*, unsigned *dt*)

Read a 3 dimensional block of histogram data with auxiliary and channel combined into "y" to suit xspress3.server.

Supported on all generations, for Xspress 3 Mini the normal mode BRAM histogram do not have any auxiliary data, so this calls xsp3m_histogram_read_frames, to read the data from the Mini over the 1G Ethernet link..

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
buffer	A pointer to the buffer for the returned data
x	Starting energy bin.
y	Channel and aux combined, with incrementing addresses covering aux and then channel
t	Starting time frame
dx	Number of energy bins.
dy	Number of the combined channel & aux
dt	Number of time frames

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_read4d (int *path*, u_int32_t * *buffer*, unsigned *eng*, unsigned *aux*, unsigned *chan*, unsigned *tf*, unsigned *num_eng*, unsigned *num_aux*, unsigned *num_chan*, unsigned *num_tf*)

Read a 4 dimensional block of histogram data.

This is supported on all generations. For Xspress 3 Mini the normal mode BRAM histogram do not have any auxiliary data, so this calls xsp3m_histogram_read_frames, to read the data from the ZYNQ over the 1G Ethernet link.

```
* Data set is 4 dimensional. Read any 4d cuboid from this 4d data volume
* Dimension 1: Energy typically 0..4095 unless RoI are used to reduce volume
* Dimension 2: Any auxiliary data, start=0, num=1 for normal operation, can be position on
ADC ramp.
* Dimension 3: Channel number
* Dimension 4: Time frame
*
*                   <------- Dim1 ------ Energy -------------------->
* 0 1   2   3   ....        4093    4094    4095
* T=0   Chan 0 Aux 0    0   1   2   3   ....        4093    4094    4095
*       Chan 0 Aux 1    4096            ....                        8191
*       Chan 0 Aux 2    8192            ....                        12287
*       Chan 0 Aux ...  12288           ...                         16383
```

```
*
* T=0   Chan 1 Aux 0    nbins_aux1*nbin_aux2+0      ....    nbins_aux1*nbin_aux2+4095
*       Chan 1 Aux 1    nbins_aux1*nbin_aux2+4096   ....    nbins_aux1*nbin_aux2+8191
*       Chan 1 Aux 2
*       Chan 1 Aux ...
*
* T=0   Chan 2 Aux 0    2*nbins_aux1*nbin_aux2+0    ....    2*nbins_aux1*nbin_aux2+4095
*       Chan 2 Aux 1    2*nbins_aux1*nbin_aux2+4096 ....    2*nbins_aux1*nbin_aux2+8191
*       Chan 2 Aux 2
*       Chan 2 Aux ...
*
*   .....
* T=1   Chan 0 Aux 0    1*num_chan*nbins_aux1*nbin_aux2+0   ....
1*num_chan*nbins_aux1*nbin_aux2+4095
*       Chan 0 Aux 1
*       Chan 0 Aux 2
*       Chan 0 Aux ...
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
buffer	Buffer pointer to receive the histogram data.
eng	Starting energy bin
aux	Starting aux point - usually 0.
chan	Starting channel number.
tf	Starting time frame
num_eng	Number of energies
num_aux	Number of aux data - usually 1.
num_chan	Number of channels
num_tf	Number of time frames

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_read_chan (int *path*, u_int32_t * *buffer*, unsigned *chan*, unsigned *eng*, unsigned *aux*, unsigned *tf*, unsigned *num_eng*, unsigned *num_aux*, unsigned *num_tf*)

Read histogram data from a single channel.

Supported on all generations, for Xspress 3 Mini the normal mode BRAM histogram do not have any auxiliary data, so this calls xsp3m_histogram_read_frames to read the data from the Mini over the 1G Ethernet link.

```
* For Xspress 3 and Xspress 4 the raw data arrangement is stored in a /dev/shm shared data
area per channel.
* Each channel's data region is a 3 d volume: energy, aux and time.
* The module header and imgd display program only understand this as  2d : energy , aux and
time combined
*
* This interface supports the native format of the data from xspress3.
* Data set is 3 dimensional. Read any 3-d cuboid from this 3-d data volume
* Dimension 1: Energy typically 0..4095 unless RoI are used to reduce volume
* Dimension 2: Any auxiliary data, start=0, num=1 for normal operation, can be position on
ADC ramp
* Dimension 3: Time frame
*
*                    <--------- Dim1 -------- Energy -------------------->
*                        0   1   2   3   ....        4093    4094    4095
* T=0   Aux 0            0   1   2   3   ....        4093    4094    4095
*       Aux 1            4096            ....                        8191
*       Aux 2            8192            ....                        12287
*       Aux ...          12288           ...                         16383
*
* T=1   Aux 0       nbins_aux1*nbin_aux2+0      ....    nbins_aux1*nbin_aux2+4095
*       Aux 1       nbins_aux1*nbin_aux2+4096   ....    nbins_aux1*nbin_aux2+8191
*       Aux 2
*
* T=2   Aux 0       2*nbins_aux1*nbin_aux2+0    ....    2*nbins_aux1*nbin_aux2+4095
*       Aux 1       2*nbins_aux1*nbin_aux2+4096 ....    2*nbins_aux1*nbin_aux2+8191
*       Aux 2
*       Aux ...
*
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
buffer	A pointer to the buffer for the returned data
eng	Starting energy bin.
aux	Starting aux point.
chan	Channel number in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1).
tf	Starting time frame.

num_eng	Number of energies.
num_aux	Number of aux data points.
num_tf	Number of time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_start (int *path*, int *card*)

Start system and enable counting.

This is the normal way to start the system, if counting is either under external control or to start immediately. For Xspress 3 the UDP port is set first as it is shared with scope mode and the UDP frame numbers are reset. Then start histogramming after resetting the UDP port and frame numbers. In software timing mode , start counting immediately. In ITFG mode the ITFG receives are started and CountEnb trigger and will start immediately unless set to wait for an external trigger.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	is the number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. card == -1 causes all cards in a multi card system to start.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_histogram_stop (int *path*, int *card*)

Stop histogramming.

Clears the run bit and count enable bit in the global timing register. The system will need to be fully restarted using **xsp3_histogram_start** etc. after stopping like this. Any buffered events will start to be sent to the UDP RX threads.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_i2c_read_adc_temp (int *path*, int *card*, float * *temp*)

Read Temperature from all 4 LM75s on 1 or all ADC boards.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card <0 then the temperatures are read from all the cards in the system.
temp	Pointer to array to store data.

RETURNS:

XSP3_OK

int xsp3_i2c_read_fem_temp (int *path*, int *card*, float * *temp*)

Read Temperature of FEM PCB and FPGA using LM82s on 1 or all boards.

This is supported only on Xspress 3. The preferred solution is to call {link **xsp3_read_xadc()**} which calls this for Xspress 3 and presents a common API for all generations.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card is the number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
temp	Pointer to array to store data, first LM82 internal, then Virtex-5 die, then next FEM card temperatures..

RETURNS:

XSP3_OK

int xsp3_i2c_read_mid_plane_temp (int *path*, float * *temp*)

Read Temperature from all 3 LM75s on the Xspress 4 mid plane.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
temp	Pointer to array to store data.

RETURNS:

XSP3_OK

int xsp3_i2c_set_adc_temp_limit (int *path*, int *card*, int *critTemp*)

Set over Temperature threshold for all 4 LM75s on 1 or all ADC boards.

If the ADC board goes over temperature then the power will shutdown. For Xspress 3 and Xspress 3 Mini the power LED will flash red.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. card <0 causes the value to be written to all cards in the system.
critTemp	Temperature in degrees to turn off power.

RETURNS:

XSP3_OK

int xsp3_init_roi (int *path*, int *chan*)

Initialise the regions of interest.

Removes any existing regions of interest.

This needs to be done if the RoI has been set and the user wishes to switch back to seeing the full spectra. It would be combined with a call to **xsp3_format_run** with nbits_eng=12.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	is the number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)

RETURNS:

XSP3_OK or a negative error code.

int xsp3_itfg_get_setup (int *path*, int *card*, int * *num_tf*, u_int32_t * *col_time*, int * *trig_mode*, int * *gap_mode*)

Retrieve settings of the Internal time frame generator (ITFG).

The minimal ITFG (currently the only version designed) can create a series of time frames all of the same length. This functions allows the settings as last set by **xsp3_itfg_setup()** to be read back.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card of the system containing the ITFG
num_tf	Pointer to return the number of time frames.
col_time	Collection time per frame in ADC clock cycles, typically 80 MHz, but confirm period with **xsp3_get_clock_period()**.
trig_mode	Pointer to return the trigger modes defined by **XSP3_ITFG_TRIG_MODE**.
gap_mode	Pointer to return frame to Frame gap moved defined by **XSP3_ITFG_GAP_MODE**

RETURNS:

XSP3_OK or a negative error code.

int xsp3_itfg_setup (int *path*, int *card*, int *num_tf*, u_int32_t *col_time*, int *trig_mode*, int *gap_mode*)

Setup Internal time frame generator (ITFG) for a series of equal length frames.

The minimal internal TFG (currently only version 1 and 2 designed) can create a series of time frames all of the same length. To use the ITFG, set timing source using **xsp3_set_glob_timeA()** with **XSP3_GTIMA_SRC_INTERNAL**. The ITFG can run a burst of frames immediate from start, a burst of frames from 1 trigger or can wait for the trigger to start each frame. This trigger can from software or hardware. start immediately from software, set trig_mode = and XSP3_ITFG_TRIG_MODE_BURST start the system calling **xsp3_histogram_start()**. arm the system and then start a burst of frame from software, set trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE_ONLY_FIRST and start the system

calling **xsp3_histogram_arm()**. Then trigger by calling **xsp3_histogram_continue()** then **xsp3_histogram_pause()**. arm the system and then trigger each frame from software, set trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE and start the system calling **xsp3_histogram_arm()**. Then trigger by calling **xsp3_histogram_continue()** then **xsp3_histogram_pause()**. Note it is necessary to pulse the CountEnb signal high and low calling both functions, but the time between the 2 calls does not give the collection time when using the ITFG. run a burst of frames with the first frame triggered by hardware, set trig_mode = XSP3_ITFG_TRIG_MODE_HARDWARE_ONLY_FIRST. The system is started using **xsp3_histogram_start()** and then triggers on the rising edge of TTL_IN(1). This can be switched to the falling edge using XSP3_GLOB_TIMA_VETO_INV To trigger each frame from hardware, set trig_mode = XSP3_ITFG_TRIG_MODE_HARDWARE and start the system using **xsp3_histogram_start()**. The system then triggers each fixed length frame on the rising edge of TTL_IN(1). This can be switched to the falling edge using XSP3_GLOB_TIMA_VETO_INV in **xsp3_set_glob_timeA()**. The de-bounce time can also be used to clean the trigger pulse to the internal TFG. a multi card system, it would be usual to make one card the master and distribute the veto signal (from TTFG_OUT(0..3) to the other cards, so the software does not broadcast settings to all cards. The other cards are set with **XSP3_GTIMA_SRC_TTL_VETO_ONLY** with the de-bounce time set to ignore any ringing but the cleanly detect the gaps between frames, see gap_mode and **XSP3_ITFG_GAP_MODE**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card of the system containing the ITFG.
num_tf	Number of time frames.
col_time	Collection time per frame in 80 MHz clock cycles.
trig_mode	Trigger modes defined by **XSP3_ITFG_TRIG_MODE**.
gap_mode	Frame to Frame gap mode defined by **XSP3_ITFG_GAP_MODE**

RETURNS:

XSP3_OK or a negative error code.

int xsp3_itfg_setup2 (int *path*, int *card*, int *num_tf*, u_int32_t *col_time*, int *trig_mode*, int *gap_mode*, int *acq_in_pause*, int *marker_period*, int *marker_frame*)

Setup Internal time frame generator (TFG) for a series of equal length frames, with access to generation 2 features.

The minimal internal TFG (currently the only version designed, with 2 generations) can create a series of time frames all of the same length. To use the ITFG, set timing source using **xsp3_set_glob_timeA()** with **XSP3_GTIMA_SRC_INTERNAL**. The ITFG can run a burst of frames immediate from start or can wait for the trigger to start each frame. This trigger can from software or hardware. start immediately from software, set trig_mode = and XSP3_ITFG_TRIG_MODE_BURST start the system calling **xsp3_histogram_start()**. arm the system and then start a burst of frame from software, set trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE_ONLY_FIRST and start the system calling **xsp3_histogram_arm()**. Then trigger by calling **xsp3_histogram_continue()** then **xsp3_histogram_pause()**. arm the system and then trigger each frame from software, set trig_mode = XSP3_ITFG_TRIG_MODE_SOFTWARE and start the system calling **xsp3_histogram_arm()**. Then trigger by calling **xsp3_histogram_continue()** then **xsp3_histogram_pause()**. Note it is necessary to pulse the CountEnb signal high and low calling both functions, but the time between the 2 calls does not give the collection time when using the ITFG. run a burst of frames with the first frame triggered by hardware, set trig_mode = XSP3_ITFG_TRIG_MODE_HARDWARE_ONLY_FIRST. The system is started using **xsp3_histogram_start()** and then triggers on the rising edge of TTL_IN(1). This can be switched to the falling edge using XSP3_GLOB_TIMA_VETO_INV To trigger each frame from hardware, set trig_mode = XSP3_ITFG_TRIG_MODE_HARDWARE and start the system using **xsp3_histogram_start()**. The system then triggers each fixed length frame on the rising edge of TTL_IN(1). This can be switched to the falling edge using XSP3_GLOB_TIMA_VETO_INV in **xsp3_set_glob_timeA()**. The de-bounce time can also be used to clean the trigger pulse to the internal TFG. a multi card system, it would be usual to make one card the master and distribute the veto signal (from TTFG_OUT(0..3) to the other cards, so the software does not broadcast settings to all cards. The other cards are set with **XSP3_GTIMA_SRC_TTL_VETO_ONLY** with the de-bounce time set to ignore any ringing but the cleanly detect the gaps between frames, see gap_mode and **XSP3_ITFG_GAP_MODE**. 2 adds the option to count while waiting for triggers. This can be used to count between rising edges of the (usually external) trigger signal. The timing part of the ITFG is not used. It just provides the accurate gaps between frames, and makes the collection time as long as possible and independent of the mark:space ratio of the external trigger. Generation 3 extends this to wait for one or more triggers per frame. The number of triggers to count per frame is specified as the count time. There are 2 variations of this mode. XSP3_ITFG_TRIG_ACQ_PAUSED_ALL acquire when waiting for trigger, including while waiting for the first trigger. A unknown length "rubbish" is collected from the software start to the first trigger. _ITFG_TRIG_ACQ_PAUSED_EXCL_FIRST acquire when waiting for trigger, except when waiting for first trigger. So first trigger starts frame 0. Then 1 or more triggers are counted to finish frame 0 and moved onto frame 1. This is often what is required. 2 and 3 also optionally (VHDL conf option) provides test markers to be used testing the markers and

Frame/Capture signals in recirculating buffer mode. This uses the mark_period and marker_frame arguments.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card of the system containing the ITFG.
num_tf	Number of time frames.
col_time	Collection time per frame in ADC clock cycles, typically 80 MHz, but confirm period with **xsp3_get_clock_period()**.
trig_mode	Trigger modes defined by **XSP3_ITFG_TRIG_MODE**.
gap_mode	Frame to Frame gap mode defined by **XSP3_ITFG_GAP_MODE**
acq_in_pause	(0 ..2) Enable acquire in pause mode, **XSP3_ITFG_ACQ_PAUSED**.
marker_period	0 to disable feature, otherwise specifies the period in FRAMES of a repeating test marker.
marker_frame	Specifies the phase in FRAMES (0...period-1) of a repeating marker.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_itfg_start (int *path*, int *card*)

Re-start the Internal time frame generator (ITFG).

This re-starts the Internal TFG after being stopped with **xsp3_itfg_stop()** . This could be used to run the ITFG several times before filling the Xspress memory. If the ITFG has been used to time or count 1 row of a raster scan, with circular buffer mode, this may be an efficient way to run long raster scans.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card of the system containing the ITFG.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_itfg_stop (int *path*, int *card*)

Stop the Internal time frame generator (TFG).

This stops the Internal TFG mid experiment. Any end of experiment processing in the firmware is not stopped, so the histogram threads will run to completion.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card of the system containing the ITFG.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_read_format (int *path*, int *chan*, int *diag_hist*, int * *num_used_chanP*, int * *nbins_eng*, int * *nbins_aux1*, int * *nbins_aux2*, int * *nbins_tf*)

Read the format control parameters from the hardware so this will see any changes set by another process.

The values are unpacked from the registers and interpreted. They are stored in Xsp3Sys and returned into the pointers passed.

Retrieves the parameters set by **xsp3_format_run**.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

chan	is the number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
diag_hist	0 => Normal histogram. 1 => Diagnostic histogram in Xspress 3 Mini only.
num_used_chanP	Pointer to return the number of channels in use. Extracted from Memory layout for Xspress3Mini only, otherwise as set by **xsp3_set_num_chan()**
nbins_eng	Pointer to return the number of bins of energy - typically 4096 unless RoI are in use.
nbins_aux1	Pointer to return the number of bins of aux1 data - 1 when aux1 data is not is use, includes Charge sharing when enabled.
nbins_aux2	Pointer to return the number of bins of aux2 data - 1 when aux2 data is not is use.
nbins_tf	Pointer to return the number of time frames. Returns smaller of the number of histogram and scalar time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_read_scope_data (int *path*, int *card*)

Read scope data from 10G Ethernet using UDP protocol for Xspress 3/4 adn 1G Ethernet fro Xspress 3 Mini This is the common API for all generations.

The data is read from the XSPREES* card and stored in the scope mode data module. For multi card system multi threads can be used, unless the multi-threading flags disable this.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has

	been passed to **xsp3_config()** at xspress3 system configuration. If card <0 the scope mode data is read for all cards using a thread per card.

RETURNS:

number of frames greater than 0 is success else 0 or negative error code is fail.

int xsp3_read_xadc (int *path*, int *card*, int *first*, int *num*, u_int32_t * *data*)

Read system monitor function from all generations of Xspress3, so far as supported.

This is the preferred option for a common function to access all 3 generations.

Xspress 4 supports all the parameter described by **XSP4_XADC_OFFSETS.** The function returns the values as integer mV and mdegC.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Is the number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
first	First data to read see **XSP4_XADC_OFFSETS**
num	Number if data items to return
data	Integer array of length (XSP3_XADC_MAX+1) to return values coded as mV and mdegC

RETURNS:

XSP3_OK or a negative error code.

int xsp3_restore_settings (int *path*, char * *dir_name*, int *force_mismatch*)

Restore all xspress3 settings from series of files.

All register, and BRAM settings are restored. Register setting are copied into Xsp3Sys as necessary for the software processing RX Threads. Dead time correction parameters are restored, but the user code should specify the beam energy. For Xspress 3 Mini and Xspress 4 the ADC gain switches and Offset DACS are also restored.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dir_name	Directory with saved settings file.
force_mismatch	Force restore if major revision of saved file does not match the firmware revision.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_restore_settings_and_clock (int *path*, char * *dir_name*, int *force_mismatch*)

Restore all xspress3 settings from series of files including configuring the clocks.

All register, BRAM settings are restored and the clock setups are read and a full clock setup performed. For compatibility with systems already in the field, if the system Xspress 3 and there is no clock file, the software falls back to a default ADC board clock setup. For Xspress 3 Mini and Xspress 4 the ADC gain switches and Offset DACS are also restored.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dir_name	Directory with saved settings file.
force_mismatch	Force restore if major revision of saved file does not match the firmware revision.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_save_settings (int *path*, char * *dir_name*)

Save all xspress3 settings into a series of files.

The directory specified must exist. Any existing files will be overwritten.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dir_name	Directory to save settings file.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_scaler_check_desc (int *path*, int *card*)

This function checks the DMA descriptors for the Scaler stream to check how many time frames of scalers have been successfully transferred to memory.

Now recommend using **xsp3_scaler_check_progress** as this checks full system.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.

RETURNS:

The number of complete time frames or a negative error code.

int xsp3_scaler_check_progress (int *path*)

This function checks how many time frames have been processed.

For hardware scalers, it check the DMA descriptors for the Scaler stream to check how many time frames of scalers have been successfully transferred to memory. For 64 bit event list histogramming it looks at the internal data structures to see the latests frame seen assuming up to here is finished. For XSpress3 Mini it checks both scalers and histogram frames. For a multi card system it checks all cards and returns the smallest number of completed frames. For more detailed information about the progress and more error detection see **xsp3_scaler_check_progress_details()**

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

The number of complete time frames or a negative error code.

int64_t xsp3_scaler_check_progress_details (int *path*, Xsp3ErrFlag * *flagsP*, int *quiet*, int64_t * *furthest_frame*)

This function checks how many time frames have been processed.

For hardware scalers, it check the DMA descriptors for the Scaler stream to check how many time frames of scalers have been successfully transferred to memory. For 64 bit event list histogramming it looks at the internal data structures to see the latest frame seen, assuming up to here is finished. If the software is currently histogramming into frame n, frames0...n-1 are finished. This function will return n. With recirculating buffer mode, it uses the extended measure of time frame including recirculation. For Xspress 3 Mini it checks both scalers and histogram frames. For a multi card system it checks all cards and returns the smallest number of completed frames.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
flagsP	Pointer to return error flags. See **Xsp3ErrFlag**
quiet	Disable printed messages, all information must be decoded from the flags

furthest_frame	Reports the highest frame number reached, ignoring any dropped frames.

RETURNS:

The number of successfully completed time frames or a negative error code.

int xsp3_scaler_get_num_tf (int *path*)

Read the maximum number of time frames of scaler data that can be stored with the current memory allocation within the FEM.

The number of time frames may be limited by the scalers or by the MCA memory (see **xsp3_get_format**).

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

number of time frames or a negative error code.

int xsp3_scaler_read (int *path*, u_int32_t * *dest*, unsigned *scaler*, unsigned *chan*, unsigned *t*, unsigned *n_scalers*, unsigned *n_chan*, unsigned *dt*)

Read a block of scaler data.

This is the normal entry point to read scaler data without dead time correction. It provides a common API for all generations and variants and wraps the other code as necessary.

```
* The data block will comprise of
* time frame[t]     channel[chan]           scaler[scaler]              data[0]
*                                                                       ...
*                                             scaler[scaler+n_scaler-1]   data[n_scaler-1]
*                                                                       ...
*                       channel[chan+n_chan-1]  scaler[scaler]              data[(n_chan-
1)*n_scaler]
*                                                                       ...
*                                             scaler[n_scaler]
data[n_chan*n_scaler-1]
*                                                                       ...
* time frame[t+dt-1]    channel[chan]           scaler[scaler]              data[(dt-
1)*n_chan*n_scaler]
*                                                                       ...
*                                             scaler[scaler+n_scaler-1]   data[dt*(nchan-
1)*n_scaler-1]
*                                                                       ...
*                       channel[chan+n_chan-1]  scaler[scaler]              data[(dt*nchan-
1)*n_scaler]
```

```
*                                                                    ...
*                                                scaler[scaler+n_scaler-1]
data[dt*n_chan*n_scaler-1]
*
* where the scalers are defined as:
* scaler 0 - Time
* scaler 1 - ResetTicks
* scaler 2 - ResetCount
* scaler 3 - AllEvent
* scaler 4 - AllGood
* scaler 5 - InWIn 0
* scaler 6 - In Win 1
* scaler 7 - PileUp
* scaler 8 - Total Time
*
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dest	Pointer to the data buffer to receive scaler data.
scaler	First scaler number to read.
chan	First channel to read.
t	First time frame to read.
n_scalers	Number of scaler to read.
n_chan	Number of channels to read.
dt	Number of time frame to read.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_scaler_read_sf (int *path*, u_int32_t * *dest*, unsigned *scaler*, unsigned *first_sf*, unsigned *chan*, unsigned *t*, unsigned *n_scalers*, unsigned *n_sf*, unsigned *n_chan*, unsigned *dt*)

Read a block of scaler data with sub-frames feature.

This is the entry point to read scaler data without dead time correction, but only in sub frames mode. It provides a common API for all generations and variants and wraps the other code as necessary, though currenyl is only supported in Xspress [3|4] with 64 bit list output.

```
* The data block will comprise of
* time frame[t]     channel[chan]           first_sf        scaler[scaler]
data[0]
*                                                                    ...
*                                             first_sf        scaler[scaler+n_scaler-1]
data[n_scaler-1]
* time frame[t]     channel[chan]           first_sf+1      scaler[scaler]
data[n_scaler]
*                                                                    ...
*                                             first_sf+1      scaler[scaler+n_scaler-1]
data[2*n_scaler-1]
*                                                                    ...
* time frame[t]     channel[chan]           first_sf+n_sf-1 scaler[scaler]
data[(n_sf-1)*n_scaler]
*                                                                    ...
*                                             first_sf+n_sf-1 scaler[scaler+n_scaler-1]
data[n_sf*n_scaler-1]
*                                                                    ...
*                       channel[chan+n_chan-1]  scaler[scaler]              data[(n_chan-
1)*n_scaler]
*                                                                    ...
*                                             scaler[n_scaler]
data[n_chan*n_scaler-1]
*                                                                    ...
* time frame[t+dt-1]    channel[chan]           scaler[scaler]              data[(dt-
1)*n_chan*n_scaler]
*                                                                    ...
*                                             scaler[scaler+n_scaler-1]   data[dt*(nchan-
1)*n_scaler-1]
*                                                                    ...
*                       channel[chan+n_chan-1]  scaler[scaler]              data[(dt*nchan-
1)*n_scaler]
*                                                                    ...
*                                             scaler[scaler+n_scaler-1]
data[dt*n_chan*n_scaler-1]
* where the scalers are defined as:
* scaler 0 - Time
* scaler 1 - ResetTicks
* scaler 2 - ResetCount
* scaler 3 - AllEvent
* scaler 4 - AllGood
* scaler 5 - InWIn 0
* scaler 6 - In Win 1
* scaler 7 - PileUp
* scaler 8 - Total Time
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dest	Pointer to the data buffer to receive scaler data.

scaler	First scaler number to read.
first_sf	First Sub -fraem (0 id no sub-frames - but usually use xsp3_scalere_read()
chan	First channel to read.
t	First time frame to read.
n_scalers	Number of scaler to read.
n_sf	Number of sub-fraems to read (1 if no sub-frames)
n_chan	Number of channels to read.
dt	Number of time frame to read.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_scope_wait (int *path*, int *card*)

This function polls the DMA descriptors for the number of scope DMA engines in use to check whether all the data has been transferred.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card <0 the function waits until all cards in the system have finished.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_alt_ttl_out (int *path*, int *card*, int *alt_ttl*, int *force*)

Specify how the TTL LEMOS are used, when not used for board to board and/or Global reset functions.

PARAMETERS:

path	Handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number of -1 to duplicate to all cards.
alt_ttl	Mode for TTL outputs defined by XSP3_GLOBAL_TIMEA_ALT_TTL
force	Force overwrite of Alternate outputs even when software has detected it will break synchronisation or global reset

RETURNS:

0 or a negative error code.

int xsp3_set_disable_threading (int *path*, int *flags*)

Set Disable multi threading flags.

For debugging purposes it is possible to disable the multi-threading which is used to speed up particularly multi-card Xspress systems.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
flags	Replacement disable multi threading flags. See **XSP3_MT_FLAGS**

RETURNS:

0 on success or a negative error code.

int xsp3_set_glob_timeA (int *path*, int *card*, u_int32_t *time*)

Set-up the triggering or time framing control register.

This register is present in all generations of Xspress. This register should be set to control how the system is to be triggered. Some bits are under user control, others are set/cleared by **xsp3_system_start**. The timing control can either be from software (**XSP3_GTIMA_SRC_SOFTWARE**) or from the various hardware inputs. The most common hardware input is to use TTL input 1 to control exposure so the system counts while this input is high. (XSP3_GTIMA_SRC_TTL_VETO_ONLY). The polarity of the selected Veto (and frame0) signals can be inverted using **XSP3_GLOB_TIMA_VETO_INV** or **XSP3_GLOB_TIMA_F0_INV** respectively. For hardware testing and potentially to daisy chain cards the TTL inputs can be looped to the TTL output using **XSP3_GLOB_TIMA_LOOP_IO**. There is a de-bounce circuit on the Veto and Frame 0 input which digitally ignores pulses (high or low going) below a minimum width specified in 80 MHz clock cycles 0..255. This is set using **XSP3_GLOB_TIMA_DEBOUNCE(time)** . This is very useful to help with reflections on long cables with questionable termination, so is recommended.

For demonstration and easy integration into systems with existing scalers, some pulse outputs can be multiplexed onto the TTL outputs. See **XSP3_GLOB_TIMA_ALT_TTL()**. The following bits should be overwritten by the xsp3_system_start and associated xsp3_histogram_* functions and should not be set by the user code: **XSP3_GLOB_TIMA_ENB_SCALER**, **XSP3_GLOB_TIMA_ENB_HIST**, **XSP3_GLOB_TIMA_NUM_SCAL_CHAN(x)**, **XSP3_GLOB_TIMA_RUN**, **XSP3_GLOB_TIMA_PB_RST**, **XSP3_GLOB_TIMA_COUNT_ENB** following definitions describe the layout of the global time A register

```
#define XSP3_GLOB_TIMA_TF_SRC(x)        ((x)&7)     //!< Sets Time frame info source see
XSP3_GTIMA_SRC_*.
#define XSP3_GLOB_TIMA_F0_INV           (1<<3)      //!< Invert Frame Zero signal polarity to
make signal active low, resets time frame when sampled low by leading edge of Veto.
#define XSP3_GLOB_TIMA_VETO_INV         (1<<4)      //!< Invert Veto signal polarity to make
signal active low, counts when Veto input is low.
#define XSP3_GLOB_TIMA_ENB_SCALER       (1<<5)      //!< Enables scalers.
#define XSP3_GLOB_TIMA_ENB_HIST         (1<<6)      //!< Enables histogramming.
#define XSP3_GLOB_TIMA_LOOP_IO          (1<<7)      //!< Loop TTL_IN(0..3) to TTL_OUT(0..3)
for hardware testing (only).
#define XSP3_GLOB_TIMA_NUM_SCAL_CHAN(x) (((x)&0xF)<<8)  //!< Sets the number of channels of
scalers to be transferred to memory by the DMA per time frame.
#define XSP3_GLOB_TIMA_SW_MARKERS(x)    (((x)&3)<<12)   //!< Set Software markers for use
(testing) in circular buffer mode.
#define XSP3_GLOB_TIMA_FRAME_CAPTURE    (1<<14)         //!< Enable Frame Capture mode in
circular buffer mode.
#define XSP3_GLOB_TIMA_FROM_RADIAL      (1<<15)         //!< Enable from Radial trigger
signals for Xspress 4 with backplane builds.
#define XSP3_GLOB_TIMA_DEBOUNCE(x)      (((x)&0xFF)<<16)    //!< Set debounce time in 80 MHz
cycles to ignore glitches or ringing on Frame0 or framing signal from any source.
#define XSP3_GLOB_TIMA_ALT_TTL(x)       (((x)&0xF)<<24)     //!< Alternate uses of the TTL
Outputs (including channel in windows signals etc).
#define XSP3_GLOB_TIMA_RUN              (1<<31)     //!< Overall Run enable signal, set after
all DMA channels have been configured.
#define XSP3_GLOB_TIMA_PB_RST           (1<<30)     //!< Resets Playback FIFO as part of
clean start.
#define XSP3_GLOB_TIMA_COUNT_ENB        (1<<29)     //!< In software timing
(XSP3_GTIMA_SRC_FIXED) mode enable counting when high. Transfers scalers on falling edge.
After first frame, increments time frame on rising edge.
#define XSP3_GLOB_TIMA_ITFG_RUN         (1<<28)     //!< From versions 11/8/2014 onwards this
is a separate Run signal to the internal time frame generator, which could be used to crash
stop the ITFG before stopping the rest.
```

```
#define XSP3_GLOB_TIMA_GET_TF_SRC(x)    ((x)&7)     //!< Sets Time frame info source see
XSP3_GTIMA_SRC_*.
```

The time frame source bits are defined as:

```
#define XSP3_GTIMA_SRC_FIXED            0       //!< Fixed constant time frame now replaced
by Software timed but incremented time frame.
#define XSP3_GTIMA_SRC_SOFTWARE         0       //!< Timing controlled by software,
incrementing on each subsequent continue.
#define XSP3_GTIMA_SRC_INTERNAL         1       //!< Time frame from internal timing
generator (for future expansion).
#define XSP3_GTIMA_SRC_IDC              3       //!< Time frame incremented and rest by
signals from IDC expansion connector.
#define XSP3_GTIMA_SRC_TTL_VETO_ONLY    4       //!< Time frame incremented by TTL Input 1.
#define XSP3_GTIMA_SRC_TTL_BOTH         5       //!< Time frame incremented by TTL Input 1
and reset to Fixed register by TTL Input 0.
#define XSP3_GTIMA_SRC_LVDS_VETO_ONLY   6       //!< Time frame incremented by LVDS Input.
#define XSP3_GTIMA_SRC_LVDS_BOTH        7       //!< Time frame incremented and reset by LVDS
Inputs.
```

The Alternate TTL output modes are defined as:

```
#define XSP3_ALT_TTL_TIMING_VETO            0       //!< Output the currently selected Count
Enable Signal from Internal TFG or other inputs and replicate 4 times on TTL_OUT 0...3 Rev
1.22 onwards
#define XSP3_ALT_TTL_TIMING_ALL             1       //!< Output TTL_OUT(0)= Veto, (1)=Veto
(2) = Running, (3) = Paused from Internal TFG (when present)
#define XSP3_ALT_TTL_TIMING_VETO_RUN        2       //!< Output TTL_OUT(0)= ITFGRun, (1)=Veto
(2) = Veto, (3) = Veto
#define XSP3_ALT_TTL_TIMING_VETO_GR         4       //!< Output Global Reset on TTL_OUT(0)
and the currently selected Count Enable Signal from Internal TFG or other inputs and
replicate 3 times on TTL_OUT 1...3 Rev 1.26 onwards
#define XSP3_ALT_TTL_TIMING_ALL_GR          5       //!< Output Global Reset on TTL_OUT(0)
and  TTL_OUT(1)=Veto, (2) = Running, (3) = Paused from Internal TFG (when present)
#define XSP3_ALT_TTL_INWINDOW               0x8     //!< Output in-window signal for channels
0:3.
#define XSP3_ALT_TTL_INWINLIVE              0x9     //!< Output 2 in-window signals and 2
live signals.
#define XSP3_ALT_TTL_INWINLIVETOGGLE        0xA     //!< Output 2 in-window signals and 2
live signals toggling.
#define XSP3_ALT_TTL_INWINGOODLIVE          0xB     //!< Outputs in-window Allevent AllGood
and LiveLevel from Chan 0.
#define XSP3_ALT_TTL_INWINGOODLIVETOGGLE    0xC     //!< Outputs in-window Allevent AllGood
and Live toggling from Chan 0.
```

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card is less than 0 then all cards are selected.
time	The timing control register

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_glob_timeFixed (int *path*, int *card*, u_int32_t *time*)

Set the fixed time frame register - starting time frame.

Whenever the time frame is reset, at the beginning of a run or using the frame 0 input, the time frame resets to this value.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	The number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration. If card is less than 0 then all cards are selected.
time	The fixed time frame register

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_good_thres (int *path*, int *chan*, u_int32_t *good_thres*)

Set the threshold for the good event scaler.

The system provides 2 versions of all event scaler. The AllGood scaler only counts events above a noise threshold, specified here. The correct value for this threshold should be determined as part of the dead-time calibration and is usually restored by **xsp3_restore_settings_and_clock()**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	is the number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) if chan is less than 0 then all channels are selected.
good_thres	the threshold for the good event scaler

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_roi (int *path*, int *chan*, int *num_roi*, XSP3Roi * *roi*)

Sets the regions of interest.

This sets the region of interest BRAM to output only selected region(s) of the spectra, placing all other counts into the last bin. It will usually follow a call to **xsp3_format_run** with nbits_eng<12.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) If chan is less than 0 then all channels are selected.
num_roi	The number of regions of interest.
roi	A pointer to an array of **XSP3Roi** structures describing the details the region.

RETURNS:

total number of bins used or a negative error code.

int xsp3_set_run_flags (int *path*, int *flags*)

Set the run mode flags.

The run mode flags control which DMA engines are started by **xsp3_system_start** usually called from **xsp3_histogram_start**. The default run flags are XSP3_RUN_FLAGS_SCALERS | XSP3_RUN_FLAGS_HIST which should be enabled unless these features are specifically to be turned off to reduce data in special experiments. Note this is not fully supported. If playback of test data into the system is required, then flags must include XSP3_RUN_FLAGS_PLAYBACK. If saving of scope mode data to DRAM is required, then flags must include XSP3_RUN_FLAGS_SCOPE. Enabling either of these when not required will still work, but xsp3_system_start will take longer to run. There is also XSP3_RUN_FLAGS_CIRCULAR_BUFFER to enable the circular buffer option where supported.

following definitions describe the valid values for the stream

```
#define XSP3_RUN_FLAGS_PLAYBACK 1       //!< Enable build of descriptors and start of DMA for
Playback DMA
#define XSP3_RUN_FLAGS_SCOPE    2       //!< Enable build of descriptors and start of DMA for
Scope Mode DMA(s)
#define XSP3_RUN_FLAGS_SCALERS  4
#define XSP3_RUN_FLAGS_HIST     8
#define XSP3_RUN_FLAGS_DIAG_HIST 0x10

#define XSP3_RUN_FLAGS_CIRCULAR_BUFFER 0x100    //!< Enable recircualting buffer mode.
```

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.
flags	

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_timing (int *path*, Xsp3Timing * *setup*)

Setup the time framing control for a whole system.

This will replace individual calls xsp3_set_glob_timeA for each card.

PARAMETERS:

path	Handle to the top level of the xspress3 system returned from **xsp3_config()**.
setup	Pointer to structure of type **Xsp3Timing** containing the time framing setup.

RETURNS:

0 or a negative error code.

int xsp3_set_trig_in_term (int *path*, int *card*, int *flags*)

Set 50 Ohm termination on USER trigger inputs.

This is supported on Xspress 3 Mini and Xspress 4. It is the reasonable that user code would set this to suit the triggering requirements of a given experiment. For Xspress 3 Mini there are only two user

TTL inputs controlled by flag bits 0 and 1. For Xspress 4 there are 4 user TTL inputs controlled by flag bits 0 to 3.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number of -1 for all. However, for XSpress4 this is usually card 0. However passing card = -1 will access card 0 only for mid-plane based Xspress4 systems.
flags	Bitwise mask 1 to enable termination resistor, otherwise Hi-Z

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_trig_out_term (int *path*, int *card*, int *flags*)

Set 50 Ohm series termination on USER trigger outputs.

This is supported on Xspress 3 Mini and Xspress 4 if the hardware requires it. It is the reasonable that user code would set this to suit the triggering requirements of a given experiment. For Xspress 3 Mini there are only two user TTL inputs controlled by flag bits 0 and 1. For Xspress4 the are 4 User TTL outputs controlled by bits 0..3 wof flags. Outputs 2 and 3 are also output as LVDS.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Card number of -1 for all.
flags	Bitwise mask 1 to enable series termination resistor, 0 => direct drive.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_set_window (int *path*, int *chan*, int *win*, int *low*, int *high*)

Set the scalar windows.

The system provides in firmware or emulates 2 in-window scalars per channel to count typically fluorescence data. This is a feature of each experiment so should be set by the user code. The upper and lower limits are specified in energy bins using this function. They are stored independently on each channel, though this function can copy the same settings to all channels. Events are counted as in window provided low <= eng <= high. The high and low threshold are specified in energy bins out of the raw spectra and remain unchanged in Regions of Interest are used.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1). If chan is less than 0 then all channels are selected.
win	The window scaler (0 or 1)
low	The low window threshold (0 ... 4095)
high	The high window threshold (0 ... 409)

RETURNS:

XSP3_OK or a negative error code.

int xsp3_sys_log_continue (int *path*)

Restart the temperature and FPGA supply voltage logging thread after it has been paused with **xsp3_sys_log_pause()**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

XSP3_OK on sucess or a negative error code.

int xsp3_sys_log_pause (int *path*)

Pause the temperature and FPGA supply voltage logging thread stopping the thread but keeping the file open to allow **xsp3_sys_log_continue()**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

XSP3_OK on sucess or a negative error code.

int xsp3_sys_log_start (int *path*, char * *fname*, int *period*, int *max_count*, Xsp3SysLogFlags *flags*)

Start the temperature and FPGA supply voltage logging thread.

This is the main, often only interaction with this code required. The logging thread reads the temperatures and Voltages from each card and writes to the specified file. This is repeated at the specified or default period. When the file has repeated this max_count times, (default 40 days) the file rol over action occurs. The default file roll over is to rename the files using fname.1... fname.10 fname.10 is deleted, fname.9 is renamed rname.10, fname.8 is renamed fname.9 etc. If the flag Xsp3SysLog_KeepFiles is specified, when the log file reaches max_count, it is renamed using the last modified time, fname.YmdHM

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
fname	The file name for the log file, also used as the base name for rolled log files.
period	Sampling period in seconds or 0 to use the default (10 s)

period	Maximum number of samples in the file before a file rollover is applied or 0 to use the default. Set of flags **Xsp3SysLogFlags**. 0 for default activity.

RETURNS:

XSP3_OK on success or a negative error code.

int xsp3_sys_log_stop (int *path*)

Stop the temperature and FPGA supply voltage logging thread, close files.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

XSP3_OK on success or a negative error code.

int xsp3_write_playback_data (int *path*, int *card*, u_int32_t * *buffer*, size_t *nbytes*, int *no_retry*)

Write playback data into DRAM via the 10G Ethernet link or 1G Ethernet.

Provide common API for all generations.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
card	Number of the card in the xspress3 system, 0 for a single card system and up to (**xsp3_get_num_cards()** - 1) for a multi-card system, where this value has been passed to **xsp3_config()** at xspress3 system configuration.
buffer	Buffer containing playback data
nbytes	Number of bytes of data in buffer

no_retry	Disable retr of sending data if TX fails.

RETURNS:

greater or equal to 0 for success else negative error code

DEAD TIME CORRECTION

The dead time correction scheme processes the data in several stages.

FUNCTIONS

- int **xsp3_getDeadtimeCorrectionParameters** (int path, int chan, int *flags, double *processDeadTimeAllEventGradient, double *processDeadTimeAllEventOffset, double *processDeadTimeInWindowOffset, double *processDeadTimeInWindowGradient)
  Get the parameters for dead time energy correction.
- int **xsp3_getDeadtimeCorrectionParameters2** (int path, int chan, int *flags, double *processDeadTimeAllEventOffset, double *processDeadTimeAllEventGradient, double *processDeadTimeAllEventRateOffset, double *processDeadTimeAllEventRateGradient, double *processDeadTimeInWindowOffset, double *processDeadTimeInWindowGradient, double *processDeadTimeInWindowRateOffset, double *processDeadTimeInWindowRateGradient)
  Get the parameters for dead time energy depentent correction, with linear Tp dependency with ICR.
- int **xsp3_setDeadtimeCorrectionParameters** (int path, int chan, int flags, double processDeadTimeAllEventGradient, double processDeadTimeAllEventOffset, double processDeadTimeInWindowOffset, double processDeadTimeInWindowGradient)
  Set the parameters for dead time energy correction.
- int **xsp3_setDeadtimeCorrectionParameters2** (int path, int chan, int flags, double processDeadTimeAllEventOffset, double processDeadTimeAllEventGradient, double processDeadTimeAllEventRateOffset, double processDeadTimeAllEventRateGradient, double processDeadTimeInWindowOffset, double processDeadTimeInWindowGradient, double processDeadTimeInWindowRateOffset, double processDeadTimeInWindowRateGradient)
  Set the parameters for dead time energy correction, including rate dependence of Tp.
- int **xsp3_getDeadtimeCorrectionFlags** (int path, int chan, int *flags)
  Get the flags only for dead time energy correction.
- double **xsp3_getDeadtimeCalculationEnergy** (int path)
  Get the energy for dead time correction.
- int **xsp3_setDeadtimeCalculationEnergy** (int path, double energy)
  Set the energy for dead time energy correction.
- int **xsp3_calculateDeadtimeCorrectionFactors** (int path, u_int32_t *hardwareScalerReadings, double *dtcFactors, double *inpEst, int num_tf, int first_chan, int num_chan)
  Calculate the dead time correction factors from the scalers.

- int **xsp3_calculateDeadtimeCorrectionFactors_sf** (int path, u_int32_t *hardwareScalerReadings, double *dtcFactors, double *inpEst, int num_tf, int first_chan, int num_chan, int num_sub_frames)
  Calculate the dead time correction factors from the scalers when using sub-frames Given an array of the hardwareScalerReadings (XSP3_SW_NUM_SCALERS = 9 values per channel) for a group of channels and frames, this calculates the dead-time correction factor for each element.
- int **xsp3_scaler_dtc_read** (int path, double *dest, unsigned scaler, unsigned chan, unsigned t, unsigned n_scalers, unsigned n_chan, unsigned dt)
  Read a block of dead time corrected scaler data.
- int **xsp3_scaler_dtc_read_sf** (int path, double *dest, unsigned scaler, unsigned first_sf, unsigned chan, unsigned t, unsigned n_scalers, unsigned n_sf, unsigned n_chan, unsigned dt)
  Read a block of dead time corrected scaler data when using sub-frames.
- int **xsp3_hist_dtc_read4d** (int path, double *hist_buff, double *scal_buff, unsigned eng, unsigned aux, unsigned chan, unsigned tf, unsigned num_eng, unsigned num_aux, unsigned num_chan, unsigned num_tf)
  Read a 4 dimensional block of dead time corrected histogram data and optionally return the dead time corrected scaler data.
- int **xsp3_hist_dtc_read4d_sf** (int path, double *hist_buff, double *scal_buff, unsigned eng, unsigned aux, unsigned chan, unsigned tf, unsigned num_eng, unsigned num_aux, unsigned num_chan, unsigned num_tf)
  Read a 4 dimensional block of dead time corrected histogram data and optionally return the dead time corrected scaler data when sub-frames are in use.
- int **xsp3_hist_dtc_read3d** (int path, double *hist_buff, unsigned eng, unsigned y, unsigned tf, unsigned num_eng, unsigned dy, unsigned num_tf)
  Read a 3 dimensional block of dead time corrected histogram data to suit xspress3.server.

DETAILED DESCRIPTION

The dead time correction scheme processes the data in several stages.

The firmware contains scalars which measure the reset dead time in clock ticks and the total exposure time. These are used correct for reset dead time. software then estimates the total input rate by inverting _rate_out = total_rate_in * exp(-total_rate_in*Tp_all)

_all = _all e^-_all*_all This estimated input rate is then used to find the correction to apply to the goo (non-pileup) peaks in the MCA and the InWidnow scalars. _win_rate_out = in_win_rate_in * exp(-2*Tp_win*total_rate_in) _win = _win e^-2_all_win

In principle, for a flat beam fill pattern, Tp_all and Tp_win should be the same. However, with gapped beam filling patterns, better fits are achieved by fitting separate values. For some detectors, due to various effects, the apparent dead time shows an energy dependence. The code here characterises the event processing time Tp as a baseline value plus a linear energy dependence. To use this feature, the library must be told the nominal beamline energy (typically the K alpha peak energy) using **xsp3_setDeadtimeCalculationEnergy()** system can estimate the total input rate. This is either from the AllEvent scalar, which counts every time the system triggers or from the AllGood Scalar. Although the

AllEvent scalar correctly counts every time the system triggers and hence goes dead, with detectors that suffer event crosstalk or other low level noise mechanisms, this will include some "nuisance events" which make the detector out step up/down only a very small amount. If these "nuisance events" overlap (pile up with) a real event, they add very little to the measure energy of the real event and hence do not move the measured energy out of the InWindow region into a pileup peak. Hence in some cases the dead time fitting works better if these triggers are ignored in the dead time correction. The AllGood scalar counts only events above a set threshold to ignore the nuisance triggers. It can be used for the dead time corrections.

FUNCTION DOCUMENTATION

int xsp3_calculateDeadtimeCorrectionFactors (int *path*, u_int32_t * *hardwareScalerReadings*, double * *dtcFactors*, double * *inpEst*, int *num_tf*, int *first_chan*, int *num_chan*)

Calculate the dead time correction factors from the scalers.

Given an array of the hardwareScalerReadings (XSP3_SW_NUM_SCALERS = 9 values per channel) for a group of channels and frames, this calculates the dead-time correction factor for each element. It assumes the hardwareScalerReadings are in the order returned by **xsp3_scaler_read**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
hardwareScalerReadings	Scaler data assumes all scalers are read from first_chan, for num_chan, repeated for num_tf read using **xsp3_scaler_read()**
dtcFactors	Pointer to return the dead time correction factors.
inpEst	Pointer to return the corrected estimate of the total input counts.
num_tf	Number of time frames of data.
first_chan	First channel to be processed.
num_chan	Number of channels to be processed.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_calculateDeadtimeCorrectionFactors_sf (int *path*, u_int32_t * *hardwareScalerReadings*, double * *dtcFactors*, double * *inpEst*, int *num_tf*, int *first_chan*, int *num_chan*, int *num_sub_frames*)

Calculate the dead time correction factors from the scalers when using sub-frames Given an array of the hardwareScalerReadings (XSP3_SW_NUM_SCALERS = 9 values per channel) for a group of channels and frames, this calculates the dead-time correction factor for each element.

It assumes the hardwareScalerReadings are in the order returned by **xsp3_scaler_read**

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
hardwareScalerReadings	Scaler data assumes all scalers are read from first_chan, for num_chan, repeated for num_tf read using **xsp3_scaler_read()**
dtcFactors	Pointer to return the dead time correction factors.
inpEst	Pointer to return the corrected estimate of the total input counts.
num_tf	Number of time frames of data.
first_chan	First channel to be processed.
num_chan	Number of channels to be processed.
num_sub_frames	Number of sub-frames

RETURNS:

XSP3_OK or a negative error code.

double xsp3_getDeadtimeCalculationEnergy (int *path*)

Get the energy for dead time correction.

The user code should tell the library at approximately what energy the system is operating. TODO: May calculate from the MCA.

PARAMETERS:

path	a handle to the top level of the xspress3 system returned from **xsp3_config()**.

RETURNS:

the energy (usually in keV) or negative error code

int xsp3_getDeadtimeCorrectionFlags (int *path*, int *chan*, int * *flags*)

Get the flags only for dead time energy correction.

These are often restored by **xsp3_restore_settings** so may not need to be handled by the user code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
flags	Pointer to return the processing flags. **XSP3_DTC_FLAGS**.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_getDeadtimeCorrectionParameters (int *path*, int *chan*, int * *flags*, double * *processDeadTimeAllEventGradient*, double * *processDeadTimeAllEventOffset*, double * *processDeadTimeInWindowOffset*, double * *processDeadTimeInWindowGradient*)

Get the parameters for dead time energy correction.

These are often restored by **xsp3_restore_settings** so may not need to be handled by the user code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.

chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
flags	Pointer to return the flags **XSP3_DTC_FLAGS**
processDeadTimeAllEventGradient	Pointer to return the energy dependence of the total rate estimating event processing time in seconds/keV.
processDeadTimeAllEventOffset	Pointer to return the baseline of the total rate estimating event processing time in seconds.
processDeadTimeInWindowOffset	Pointer to return the baseline of the in window estimating event processing time in seconds.
processDeadTimeInWindowGradient	Pointer to return the energy dependence of the in window estimating event processing time in seconds/keV.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_getDeadtimeCorrectionParameters2 (int *path*, int *chan*, int * *flags*, double * *processDeadTimeAllEventOffset*, double * *processDeadTimeAllEventGradient*, double * *processDeadTimeAllEventRateOffset*, double * *processDeadTimeAllEventRateGradient*, double * *processDeadTimeInWindowOffset*, double * *processDeadTimeInWindowGradient*, double * *processDeadTimeInWindowRateOffset*, double * *processDeadTimeInWindowRateGradient*)

Get the parameters for dead time energy dependent correction, with linear Tp dependency with ICR.

These are often restored by **xsp3_restore_settings** so may not need to be handled by the user code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1)
flags	Pointer to return the flags **XSP3_DTC_FLAGS**

processDeadTimeAllEventOffset	Pointer to return the baseline of the total rate estimating event processing time in seconds.
processDeadTimeAllEventGradient	Pointer to return the energy dependence of the total rate estimating event processing time in seconds/keV.
processDeadTimeAllEventRateOffset	Pointer to return the ICR baseline of the total rate estimating event processing time in seconds dependence on rate.
processDeadTimeAllEventRateGradient	Pointer to return the ICR baseline of the total rate estimating event processing time in seconds dependence on rate.
processDeadTimeInWindowOffset	Pointer to return the baseline of the in window estimating event processing time in seconds.
processDeadTimeInWindowGradient	Pointer to return the energy dependence of the in window estimating event processing time in seconds/keV.
processDeadTimeInWindowRateOffset	Pointer to return the baseline of the in window estimating event processing time in seconds dependence on rate.
processDeadTimeInWindowRateGradient	Pointer to return the energy dependence of the in window estimating event processing time in seconds/keV dependence on rate.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_hist_dtc_read3d (int *path*, double * *hist_buff*, unsigned *eng*, unsigned *y*, unsigned *tf*, unsigned *num_eng*, unsigned *dy*, unsigned *num_tf*)

Read a 3 dimensional block of dead time corrected histogram data to suit xspress3.server.

This function is used to allow xspress3.server to provide backwards compatibility with xspress2, but should not be used in new code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
hist_buff	Buffer pointer to receive the histogram data.
eng	Start energy bin.
y	Starting y position (combined aux and channel to suit xspress3.server)
tf	Starting time frame.
num_eng	Number of energy bins.
dy	Number of y (combined aux adn channel)
num_tf	Number of time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_hist_dtc_read4d (int *path*, double * *hist_buff*, double * *scal_buff*, unsigned *eng*, unsigned *aux*, unsigned *chan*, unsigned *tf*, unsigned *num_eng*, unsigned *num_aux*, unsigned *num_chan*, unsigned *num_tf*)

Read a 4 dimensional block of dead time corrected histogram data and optionally return the dead time corrected scaler data.

This is the usual method to access dead time corrected data. Both MCA and scaler data can be returned. If sub-frames are used, this code will call **xsp3_hist_dtc_read4d_sf()** and aux, num_aux become the first and number of sub-frames. The scal_buffer then grows in size to be XSP3_SW_NUM_SCALERS*num_aux*num_chan.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
hist_buff	Buffer pointer to receive the histogram data.

scal_buff	Buffer pointer to receive the scaler data or NULL if not required.
eng	Start energy bin.
aux	Starting aux point (usually 0)
chan	Number of the first channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1).
tf	Starting time frame.
num_eng	Number of energy bins.
num_aux	Number of aux data (usually 1).
num_chan	Number of channels.
num_tf	Number of time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_hist_dtc_read4d_sf (int *path*, double * *hist_buff*, double * *scal_buff*, unsigned *eng*, unsigned *aux*, unsigned *chan*, unsigned *tf*, unsigned *num_eng*, unsigned *num_aux*, unsigned *num_chan*, unsigned *num_tf*)

Read a 4 dimensional block of dead time corrected histogram data and optionally return the dead time corrected scaler data when sub-frames are in use.

This function is called by {link **xsp3_hist_dtc_read4d()**} or can be called directly.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
hist_buff	Buffer pointer to receive the histogram data.

scal_buff	Buffer pointer to receive the scaler data or NULL if not required.
eng	Start energy bin.
aux	Starting aux point (usually 0)
chan	Number of the first channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1).
tf	Starting time frame.
num_eng	Number of energy bins.
num_aux	Number of aux data (usually 1).
num_chan	Number of channels.
num_tf	Number of time frames.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_scaler_dtc_read (int *path*, double * *dest*, unsigned *scaler*, unsigned *chan*, unsigned *t*, unsigned *n_scalers*, unsigned *n_chan*, unsigned *dt*)

Read a block of dead time corrected scaler data.

If only scalers are required, this is the normal entry point for scalers. However, due to the overhead of reading the scalers from the FPGA board in some generations, if both scalers and spectra are required, please use **xsp3_hist_dtc_read4d**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dest	A pointer to the data buffer to receive the dead tiem corrected scaler data.

scaler	First scaler number to read.
chan	First channel to read.
t	First time frame to read.
n_scalers	Number of scaler to read.
n_chan	Number of channels to read.
dt	Number of time frame to read.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_scaler_dtc_read_sf (int *path*, double * *dest*, unsigned *scaler*, unsigned *first_sf*, unsigned *chan*, unsigned *t*, unsigned *n_scalers*, unsigned *n_sf*, unsigned *n_chan*, unsigned *dt*)

Read a block of dead time corrected scaler data when using sub-frames.

If only scalers are required, this is the normal entry point for scalers. However, due to the overhead of reading the scalers from the FPGA board in some generations, if both scalers and spectra are required, please use **xsp3_hist_dtc_read4d_sf**.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
dest	A pointer to the data buffer to receive the dead time corrected scaler data.
scaler	First scaler number to read.
first_sf	First sub-frame to read 0..num_sub_frames-1
chan	First channel to read.

t	First time frame to read.
n_scalers	Number of scaler to read.
n_sf	Number of sub-frame to read.
n_chan	Number of channels to read.
dt	Number of time frame to read.

RETURNS:

XSP3_OK or a negative error code.

int xsp3_setDeadtimeCalculationEnergy (int *path*, double *energy*)

Set the energy for dead time energy correction.

The user code should tell the library at approximately what energy the system is operating.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
energy	The energy in keV

RETURNS:

XSP3_OK or a negative error code.

int xsp3_setDeadtimeCorrectionParameters (int *path*, int *chan*, int *flags*, double *processDeadTimeAllEventGradient*, double *processDeadTimeAllEventOffset*, double *processDeadTimeInWindowOffset*, double *processDeadTimeInWindowGradient*)

Set the parameters for dead time energy correction.

These are often restored by **xsp3_restore_settings** so may not need to be handled by the user code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) If chan is less than 0 then the data is copied to all channels.
flags	Processing flags **XSP3_DTC_FLAGS**
processDeadTimeAllEventGradient	Energy dependence of the total rate estimating event processing time (seconds/keV).
processDeadTimeAllEventOffset	Baseline of the total rate estimating event processing time (seconds).
processDeadTimeInWindowOffset	Baseline of the in window estimating event processing time (seconds).
processDeadTimeInWindowGradient	Energy dependence of the in window estimating event processing time (seconds/keV).

RETURNS:

XSP3_OK or a negative error code.

int xsp3_setDeadtimeCorrectionParameters2 (int *path*, int *chan*, int *flags*, double *processDeadTimeAllEventOffset*, double *processDeadTimeAllEventGradient*, double *processDeadTimeAllEventRateOffset*, double *processDeadTimeAllEventRateGradient*, double *processDeadTimeInWindowOffset*, double *processDeadTimeInWindowGradient*, double *processDeadTimeInWindowRateOffset*, double *processDeadTimeInWindowRateGradient*)

Set the parameters for dead time energy correction, including rate dependence of Tp.

These are often restored by **xsp3_restore_settings** so may not need to be handled by the user code.

PARAMETERS:

path	A handle to the top level of the xspress3 system returned from **xsp3_config()**.
chan	The number of the channel in the xspress3 system, 0 to (**xsp3_get_num_chan()** - 1) If chan is less than 0 then the data is copied to all channels.
flags	Processing flags **XSP3_DTC_FLAGS**
processDeadTimeAllEventOffset	Baseline of the total rate estimating event processing time (seconds).
processDeadTimeAllEventGradient	Energy dependence of the total rate estimating event processing time (seconds/keV).
processDeadTimeAllEventRateOffset	Baseline of the total rate estimating event processing time (seconds/Hz).
processDeadTimeAllEventRateGradient	Energy dependence of the total rate estimating event processing time (seconds/Hz/keV).
processDeadTimeInWindowOffset	Baseline of the in window estimating event processing time (seconds).
processDeadTimeInWindowGradient	Energy dependence of the in window estimating event processing time (seconds/keV).

processDeadTimeInWindowRateOffset	Baseline of the in window estimating event processing time (seconds/Hz).
processDeadTimeInWindowRateGradient	Energy dependence of the in window estimating event processing time (seconds/Hz/keV).

RETURNS:

XSP3_OK or a negative error code.

INDEX