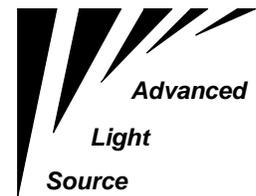

ALS controls

Accessing the BPM FADs

Chris Timossi



Advanced Light Source Computer Controls Group.

Chris Timossi.
Lawrence Berkeley Lab
ms 10-110
(510) 486-5385

Contents

Overview	3
Beam Position Monitor Hardware.....	3
Beam Position Monitor Software (ILC).....	3
Controlling a FAD.....	4
Single Trigger Mode.....	4
Continuous Trigger Mode.....	4
Simultaneous Access by Multiple Programs.....	5
Validity of readings.....	5
Using FADLIB	7
Description.....	7
Subroutines.....	7
Structures.....	10
Constants.....	10
Sample Code.....	11
Glossary of Terms	19
Index	21

Overview

Beam Position Monitor Hardware

The **BPMs** are high resolution position monitors placed around the accelerator. Physically, the BPM electronics monitors 4 pickups called **buttons**. The signals from these buttons are made available directly as video outputs (sometimes summed to produce an intensity monitor) and as inputs to an **ILC** via 4 'slow' 16 bit A/Ds. The A/D is referred to as slow since it will only be reliable when more than about one millisecond of beam is present (enough beam to charge up a capacitor). For the 32 monitors in the Booster Ring and the 96 monitors in the storage ring the button signals are also brought into Fast A/Ds connected to two separate 1024 deep FIFOs. The Fast A/D or **FADs**, can digitize the data to 8 bit accuracy at the revolution rate of the accelerator giving a turn by turn profile of the position for 1023 consecutive turns. The two FIFOs are labeled 'FAST' and 'SLOW'. The Fast FIFO always contains the data digitized at the full revolution rate of the booster or storage ring. The **Slow FIFO**, can be programmed to digitize every 10, 100, or 1000 turns giving a longer time record of the acceleration cycle. Once the FADs have been enabled, data continuously flows through the FIFOs until an external halt trigger is generated (usually by an external source like a Stanford Timer). The data is then frozen and available to the control system via the ILC. To be able to adjust to a wide range of beam intensities, the gain of the button amplifiers has to be set so that the incoming signal remains in the linear portion of the amplifiers range.

Beam Position Monitor Software (ILC)

An ILC is included in each BPM chassis to provide local processing and communication to the main control system. The ILC performs the following local processing of the button data: in **slow mode** (using the 16 bit A/D), the ILC averages the readings and calculates the x and y positions using constants supplied by its local database. The ILC also implements a **calibration** routine which attempts to achieve the highest accuracy of reading by compensating for the gain and offset differences of the four amplifiers. For the FAD's (or FAD mode), the ILC also performs conversion to position.

Controlling a FAD

At the workstation or PC level, the FADs can be controlled by high level calls to a library or by direct manipulation of a special control byte in the BPM database. In either case, it is important to understand the steps involved in controlling the FADs.

Single Trigger Mode

Application is responsible for enabling the FADs

Both the Storage Ring FADs and the Booster Ring FADs can be operated in single trigger mode. In single trigger mode, the application must initialize a FAD any time new data is needed. The process looks like:

Initializing

In this step the FAD is enabled to take data. The slow FIFO is given its rate to digitize: every turn, every 10 turns every 100 turns, every 1000 turns. Once the FAD is initialized, data will flow through the FIFOs until a **Halt Trigger** is detected by the electronics.

Triggering

After the FAD has been initialized, a check must be made to see if the external trigger has occurred before any data is read from the FIFO. Once the FAD has triggered, the data from the fifos are read into the ILC and does not change again until the fad is re-initialized.

Set Read Mode

The ILC can be told to deliver data from either the fast or slow FIFO and to deliver the raw 4 button A/D reading for each turn or to deliver the calculated x or y positions.

Wait for data

Once the trigger has occurred, the data will still take time to be sent from the ILC to the main database in the **CMM**.

Read Data

The data is valid and is read by the requesting application. The Read Mode can be changed and different data requested.

Continuous Trigger Mode

In continuous mode, 20 turns of data is continually acquired by the ILC. Applications just read the data.

The Storage ring FADs can operate in a new mode called **continuous trigger** mode. In this mode, new data is read into the ILC from the FIFO every time a halt trigger occurs. In addition, the first 20 turns from the FIFO are delivered to the main database for use by applications in less than .1 sec so that the data can be displayed at the rate of Injection. The access to the data involves the following steps:

Initializing

This step is only done once to enable the FADs and select the rate of the slow FIFO.

Set Read Mode

Data can be read either from the fast or slow fifo. Either x AND y data OR the sum of the 4 button signals can be selected. This is different from the booster which reports x OR y OR 4 button values. Once the mode is set, the data sent from the ILC to the CMM. The data is marked with a counter that is incremented each time a halt trigger is detected at the ILC.

Read Data

Data from the FADs is available in the CMM. The data changes at the rate of the External Halt trigger. X & Y position data is returned in integer format in units of meters; the application must divide by 1000 to get the value in millimeters. The SUM data is also in integer format (0-1020).

Simultaneous Access by Multiple Programs

From the above discussion, it is apparent that different applications can collide over the control of the FADs. Conflicts can occur with programs that change the Read Mode (x,y,raw), the frequency of the slow fifo, or the trigger mode (single, continous). Many different applications can read the data, however, once the FADs have been setup. Also, when the BPMs are told to report FAD data, the rate at which the slow mode (high accuracy) position data is reported back to an application is degraded.

Validity of readings

Many factors can influence the validity of the data read from the fads. Some error conditions cannot or are not detected by the routines. The most typical problem is incorrect gain setting of the BPMs. If the gain is too high, the position read by the routines can be far different than the actual position. Another problem is the position of the external halt trigger which stops the capture of data into the FIFOs; for the booster, the triggering can be set to time during which no beam occurs.

Using FADLIB

Description

Fadlib is a windows Dynamic Link Library (DLL) that can be used to access the BPM FADs for high level control. The routines in the library are generally divided between Storage Ring and Booster Ring. The DLL looks up BPM names the first time that BPM is accessed and stores the index for subsequent access. The DLL also maintains other state informations such: whether the name was found, whether the fad succeeded in initialization, etc.

Applications that use fadlib must have access to light40 (the main file server).

1. If not done already, assign y: to \\light40\controls and z to \\light40\rootd.
2. Put z:\lib\controls\fadlib.lib (the import library) in the LIB enviornment variable.
3. \opstat\rbin must be in the executable path.
4. \include\controls\fadlib.h has the C prototypes and manifest constants

Subroutines

Arguments:

Error Code returned

Sector: 1-4 (BR) 1-12 (SR)

Number: 1-8

DataType: fast/slow; xy/raw

Slow Fad Frequency: 1-4

Data Array returned: floats (BR) or ints (SR)

The first argument is almost always an Error Code that is used by the routine to report the status of the operation. The number returned can be decoded by a call to 'dbgeterr' (see sample code) so that a reasonable error message can be displayed. To control a specific FAD, the routines usually accept a 'sector number' followed by a 'BPM number'. For the booster, the sector number is from 1 to 4 for the storage ring the number is 1 to 12. In both cases the BPM number is 1 to 8. In addition, some routines are capable of controlling multiple FADs (for example all the storage ring fads). For these routines the convention is to set sector and bpm number to 0. The routines return arrays or structures of a variety of types but the booster routines usually return floating point arrays (single precision) whereas the storage ring routines return arrays of integers (since the FADs only digitize to 8 bit accuracy).

The routines fall in 3 broad categories: initialization, set read mode, and read data.

/* initialization routines */

int _far _pascal _export

init_br_fad(int _far *Err, int sect, int nr, int freq) ;

Enable fad (sect,nr) and set the frequency of the slow fifo.

int _far _pascal _export

init_sr_fad(int _far *Err, int sect, int nr, int freq);

Enable fad (sect,nr) and set the frequency of the slow fifo. Setting both sector and number to 0, initializes all fads.

/* single trigger mode */

void _far _pascal _export

wait_br_fad_trigger(int _far *Err, int sect, int nr, int maxwait);

Waits until FAD (sect,nr) receives a halt trigger after an initialization. 'maxwait' is ignored.

void _far _pascal _export

wait_sr_fad_trigger(int _far *Err, int sect, int nr, int maxwait);

Waits until FAD (sect,nr) receives a halt trigger after an initialization. 'maxwait' is ignored.

/* set read mode */

void _far _pascal _export

set_br_fadmode(int _far *Err, int _far *datatype, int sect, int nr);

Sets the read mode of the (sect,nr) fad for the booster. Possible read modes are to return: raw, x, or y data from the fast or slow FIFO either full (1023 turns) or abbreviated (20 turn xy data combined) data.

void _far _pascal _export

set_sr_fadmode(int _far *Err, int _far *datatype, int sect, int nr);

Sets the read mode of the (sect,nr) fad for the storage ring. Possible read modes are to return: raw or x and y data from the fast or slow FIFO either full 1023 turns, in single trigger mode, or 20 turns in continuous trigger mode.

void _far _pascal

start_br_fads(int _far * ErrCode, int _far *datatype, unsigned long fad_mask);

A special call to handle: initializing, waiting for trigger, setting datatype, and waiting for valid data for a random assortment of booster FADS. The routine is passed a fad_mask with a bit set for each of the 32 bpms in the booster ring. A value of 0xffffffff sets all FADs in the booster ring; a value of 0x01 sets the first one, etc.

void _far _pascal _export

select_sr_fad_fifo(int _far *Err, int _far *datatype, int sect, int nr);

Similar to set_sr_fadmode, but it accepts sect = nr = 0 as a command to set all FADs in the storage ring to the desired data type.

void _far _pascal _export

DbSetSynd(UBYTE2 _far *Err, int EventType, int ListNr, UBYTE2 _far *Cnt);

Meant to handle access of multiple database items in a list labeled ListNr, synchronized on event EventType. For the Storage ring FADs, ListNr = 1 and EventType =1 synchronizes all the storage ring fads so that subsequent calls to DbGetSync returns synchronized data from all the FADs. When DbSetSync is called, the DLL records a fiducial count for each of the BPMs in the storage ring; the value of this count for the first bpm is returned in Cnt. DbSetSync must be called once before DbGetSync is called to read the data.

/* data access */

int _far _pascal _export

DbGetSync(UBYTE2 _far *Err, int ListNr, UBYTE1 _far *dataArray, int Nr, UBYTE2 _far *Cnt);

Can be used to get synchronized data from a list (ListNr) of database items. For storage ring fads, ListNr=1 returns abbreviated (20 turn) data in one of two formats: in xy format dataArray contains 40 integers for each FAD; the 40 integers are organized as 20 pairs of xy data in meters. In 'sum' format, dataArray contains 20 integers for each fad; each integer represents the sum of 4 button signals (0-1020). Cnt contains the value of the counter of the first bpm; this is the count that is used for synchronization by FADLIB, but is not used by the application typically.

Returns the number of bpm's that failed to sync.

int _far _pascal _export

GetFadStatus(int type, char _far FadName[][DB_FULLNAMELEN+1], UBYTE4 _far FadIndex[], UBYTE2 _far FadStat[], int Nr);

Retrieves an 8 bit encoded status byte, a bpm name, and a bpm index ilc (of the bp channel) for each bpm in the storage ring. The status byte contains a bit set to 1 for each of the following successful operations: bpm name found (1), frequency set (2), fad enabled (4), data mode set (8), SetSync succeeded (16), and GetSync succeeded(32).

void _far _pascal _export

read_br_fad_raw(UBYTE2 _far *ErrCode, int _far *start_turn, int _far *end_turn, int sect, int nr, UBYTE1 _far *Buts);

Returns the value of the FAD (sect,nr) for the booster. start_turn should be 1 and end_turn should be 1023. Buts contains an array of 4096 bytes organized as button1, 2, 3 and 4 for each turn. Byte values range from 0 to 255.

void _far _pascal _export

read_sr_fad_raw(UBYTE2 _far *ErrCode, int _far *start_turn, int _far *end_turn, int sect, int nr, UBYTE1 _far *Buts);

Returns the raw button values of the FAD (sect,nr) for the storage ring. start_turn should be 1 and end_turn should be 1023. Buts contains an array of 4096 bytes organized as a byte for button1, 2, 3 and 4 for each turn. Byte values range from 0 to 255.

void _far _pascal

read_br_fad_xy(UBYTE2 _far *ErrCode, int _far *start_i, int _far *end_i, int sect, int nr, float _far *ByteArray);

Returns the x or y position values of the FAD (sect,nr) for the booster. start_turn should be 1 and end_turn should be 1023. ByteArray contains an array of 1024 floating point values representing x or y in millimeters. Values range from 0 to 30 (30 or larger indicate failure).

void _far _pascal _export

read_sr_fad_xy(UBYTE2 _far *ErrCode, int _far *start_turn, int _far *end_turn, int sect, int nr, int _far *ByteArray);

Returns the x or y position values of the FAD (sect,nr) for the storage ring. start_turn should be 1 and end_turn should be 1023. Buts contains an array of 2048 integer values organized in pairs of x, y values measured in meters. Typical range is 0 to 30,000. Numbers of 30,000 or greater indicate failure.

void _far _pascal

read_br_fad_quick(UBYTE2 _far *ErrCode, int sect, int nr, FAD_QUICK_TYPE _far *fad);

Reads data from one booster ring fad in a special abrieved data mode called 'quick' mode. In quick mode, a booster fad ilc just returns 20 turns of xy or 4 button raw data using the FAD_QUICK_TYPE structure.

void _far _pascal _export

ffad_read(UBYTE2 _far *ErrCode, int sect, int nr, FADCONTXY _far *fad);

Reads one storage ring fad (sect,nr) in the continuous trigger mode. The data is returned in the FADCONTXY structure.

Structures

```
typedef struct {
    float x;
    float y;
} FADPOSTYPE;
```

```
typedef struct {
    FADPOSTYPE pos[FADNTURN];
    UBYTE1 b1[FADNTURN];
    UBYTE1 b2[FADNTURN];
    UBYTE1 b3[FADNTURN];
    UBYTE1 b4[FADNTURN];
} FAD_QUICK_TYPE;
```

```
typedef struct {
    UBYTE2 cnt;
    int x[FADNTURN];
    int y[FADNTURN];
} FADCONTXY;
```

```
typedef struct {
    UBYTE2 cnt;
    UBYTE2 sum[FADNTURN];
} FADCONTSUM;
```

Constants

```
#define MAXBRBPMS 32
#define FIRSTRBPM 32
#define LASTSRBPM 127
#define MAXSRBPMS 96
#define NRTURNS 1024
#define FADNTURN 20
```

```

#define BRFADTYPE 0
#define SRFADTYPE 1
#define FADNAMELEN 13

/* triggering & handshake */
#define FADASKTRIG 50
#define FADTRIGGERED 200
#define FADNOTTRIGGERED 201
#define FADRESETCMND 0
#define FADRESET 100
/* set with init_br_fad*/
#define FADFREQ1 1
#define FADFREQ10 2
#define FADFREQ100 3
#define FADFREQ1000 4
/* fast mode for br only */
#define FADQUICKFAST 5
#define FADQUICKSLOW 6
/* set with set_br_fadmode(FF means return FADNTURN turns)*/
#define FADRAW 20
#define FADX 21
#define FADY 22
#define FADRAWSLOW 30
#define FADXSLOW 31
#define FADYSLOW 32
#define FADNTURNRAW 40
#define FADNTURNXY 41
#define FFADXYFASTS 45
#define FFADXYSLWS 46
#define FFADSUMFASTS 47
#define FFADSUMSLWS 48
/* continuous mode stuff set with set_fad_fifo (sr only)*/
#define FFADXYFAST 5
#define FFADXYSLW 6
#define FFADSUMFAST 7
#define FFADSUMSLW 8

```

Sample Code

Example of Continuous Mode read of Storage Ring FADs

```

#ifdef _WINDOWS
#include <windows.h>
#endif
#include <stdio.h>
#include <ptypes.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <dbdefine.h>
#include <time.h>
#include <fadlib.h>
#include <llinkc.h>

```

```

void far pascal REPORT(int, int, char _far *);
#define dbgeterr(x,y) REPORT(x, 0, y)

struct {
    int x[FADNTURN];
    int y[FADNTURN];
} FadXY[MAXSRBPMS];

struct {
    int sum[FADNTURN];
} FadSum[MAXSRBPMS];

char Names[MAXSRBPMS][DB_FULLNAMELEN+1];
UBYTE4 Indices[MAXSRBPMS];
UBYTE2 Status[MAXSRBPMS];

/* illustrates the use of CONTINUOUS MODE READ for STORAGE RING FADs */
void main(int argc, char *argv[] )
{
    int ErrCode, sr, nr, freq, i, AccessMode, tries, fails, failures, fadnr;
    int firstsr, lastsr, sum, Nr;
    UBYTE2 Cnt;
    static char ErrStr[80];
    float x, y;

    /* access the 'sum of buttons' from the FAST fifo */
    AccessMode = FFADSUMFAST;
    /* AccessMode = FFADXYFAST; */
    freq=1;
    failures = 0;
    firstsr = 1;
    lastsr = 12;

    /* initialize all fads */
    printf("initializing all fads\n");
    failures = init_sr_fad( &ErrCode, 0, 0, freq);
    if (ErrCode) {
        dbgeterr(ErrCode, ErrStr);
        printf("cant init fads\n");
        printf("%s\n", ErrStr);
    }
    printf ("%u Failed to initialize\n", failures);

    /* set all fads to cont. read (sum) */

select_sr_fad_fifo( &ErrCode, &AccessMode, 0, 0 );
    if (ErrCode) {
        dbgeterr(ErrCode, ErrStr);
        printf("cant select fad: %d %d\n", sr, nr);
        printf("%s\n", ErrStr);
    }
}

```

```

        /* SET sync */
printf("setsync\n");
DbSetSync( &ErrCode, 1, 1, &Cnt);
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("Some FADs failed to sync\n");
    printf("%s\n", ErrStr);
}
printf("count %u\n\n", Cnt);

GETSYNC:
for (i = 1; i <= 1; i++) {
    printf("getsync\n");
    fails = 0;
    while ((fadnr=DbGetSync( &ErrCode, 1, (UBYTE1 *)&FadSum, 100, &Cnt)) && fails < 1) {
        if (ErrCode) {
            printf("%d fads failed to sync\n", fadnr);
            dbgeterr(ErrCode, ErrStr);
            fails ++;
        }
    }
    printf("count %u \n\n", Cnt);
}
Nr = MAXSRBPMS;
GetFadStatus(SRFADTYPE, Names, Indices, Status, Nr);

if ( AccessMode == FFADSUMFAST ) {
    for ( sr = firstsr; sr <= lastsr; sr++ ) {
        printf("sr %d: ", sr);
        for ( nr = 1; nr <= 8; nr++ ) {
            i = (sr-1)*8+nr-1;
            sum = FadSum[i].sum[0];
            printf("%u ", sum, Status[i]);
            if (Status[i] != 0x3f)
                printf("(%x)t", Status[i]);
            else
                printf("\t");
        }
        printf("\n");
    }
} else {
    for ( sr = firstsr; sr <= lastsr; sr++ ) {
        printf("sr %d:", sr );
        for ( nr = 1; nr <= 8; nr++ ) {
            i = (sr-1)*8+nr-1;
            x = FadXY[i].x[0]/1000.0;
            y = FadXY[i].y[0]/1000.0;
            printf("(%.4f,%.4f):%x\t", x, y, Status[i]);
        }
        printf("\n");
    }
}
goto GETSYNC;

```

```

/* check status */
printf ("done\n");

```

Example of Single Trigger access of Storage Ring Fads

```

//
// savefad.c: saves all 1024 turns of all 96 storage ring fads in 2
// files whose name is entered by the user. 'xy' and 'raw' are appended
// to the file name to store xy and raw data. To make an FFT easy,
// 1024 turns, rather than the 1023 that the fads really return,
// are stored. The 1024th entry is just copied from the 1023rd entry.
//
#include <stdio.h>
#include <string.h>
#include <report.h>
#include <dbdefine.h>
#include <time.h>
#include <linkc.h>
#include <fadlib.h>

#define dbgeterr(x,y) REPORT(x, 0, y)

void GetFileName(void);

struct {
    UBYTE2 cnt;
    UBYTE1 button[4][1024];
} FadRaw;

struct {
    UBYTE2 cnt;
    int x[1024];
    int y[1024];
} FadXY;

char ErrStr[81];
char Names[MAXSRBPMS][DB_FULLNAMELEN+1];
UBYTE4 Indices[MAXSRBPMS];
UBYTE2 Status[MAXSRBPMS];
static FILE *fh, *fhraw;

void main(argc,argv)
int argc;
char* argv[];
{
    int ErrCode, i, j, start_i, end_i, datatype, Nr;
    UBYTE2 ilc_array[32], delaytime;
    int nr, sr, freq, sr_off, failures;
    char type, ans[80];
    float sum, x, y;
    int firstsr, lastsr, turn, b;

```

```

freq = 1 ;
turn = 1;
start_i = 1; end_i = 1023;
delaytime = 200;
firstsr = 1; lastsr = 12;

Logon (&ErrCode, &Nr, ilc_array, "light46");
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("%s\n", ErrStr);
    exit(1);
}

GetFileName();

    /* initialize all fads */
printf("\nRemove trigger cable. Ready ? (y/n)\n");
scanf("%s", &ans);

//
//With trigger cable removed, arm all the fads
//
printf("initializing all fads\n");
failures = init_sr_fad ( &ErrCode, 0, 0, freq);
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("cant init fads\n");
    printf("%s\n", ErrStr);
}
if (failures > 0) {
    printf ("%u Failed to initialize\n\n",failures);
    GetFadStatus(SRFADTYPE, Names, Indices, Status, MAXSRBPMS);
}

sr = nr = 1;
printf("\nRestore trigger cable. Ready ? (y/n)\n");
scanf("%s", &ans);
//
// restore the cable. The FADs will trigger on the next FAD Halt Trigger
//
wait_sr_fad_trigger (&ErrCode, sr, nr, 100 );
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("cant get fad trigger for sector %d bpm %d \n",sr,nr);
    printf("%s\n", ErrStr);
    exit(1);
} else {
    printf("Triggered\n");
}

//
// For the storage ring, xy position come back in the same structure
// as integers

```

```

//
datatype =FADXSLOW;
select_sr_fad_fifo( &ErrCode, &datatype, 0, 0 );
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("cant select fad: %d %d\n", sr, nr);
    printf("%s\n", ErrStr);
                                //exit(1);
}
start_i = 1; end_i = 1023;
DELAY(&delaytime); // wait for data to arrive at CMM

// read all storage ring fads
for ( sr = firstsr; sr <= lastsr; sr++ ) {
    printf("sr %d:", sr );
    for ( nr = 1; nr <= 8; nr++ ) {
        printf(" %d", nr );
        fprintf(fh, "#%d\t%d", sr, nr);
        read_sr_fad_xy(&ErrCode, &start_i, &end_i, sr, nr, &FadXY);
        if (ErrCode) {
            dbgeterr(ErrCode, ErrStr);
            printf("%s", ErrStr);
            fprintf(fh, " Error");
        }
        fprintf(fh, "\n");
        printf("\n");
        for (turn = 0; turn < 1023; turn++) {
            x = ((ErrCode) ? 0.0 : FadXY.x[turn]/1000.0);
            y = ((ErrCode) ? 0.0 : FadXY.y[turn]/1000.0);
            fprintf(fh, "%d\t%.4f\t%.4f\n", turn, x, y);
        }
        fprintf(fh, "%d\t%.4f\t%.4f\n", turn, x, y);
    }
}

printf("\n");
fclose(fh);

fh = fhraw;
//
// Read raw values
//
datatype = FADRAW;

select_sr_fad_fifo( &ErrCode, &datatype, 0, 0 );
if (ErrCode) {
    dbgeterr(ErrCode, ErrStr);
    printf("cant select fad: %d %d\n", sr, nr);
    printf("%s\n", ErrStr);
                                //exit(1);
}

```

```

DELAY(&delaytime);

GetFadStatus(SRFADTYPE, Names, Indices, Status, MAXSRBPMS);

for ( sr = firstsr; sr <= lastsr; sr++ ) {
    printf("sr %d:\n", sr);
    for ( nr = 1; nr <= 8; nr++ ) {
        fprintf(fh,"#%d\t%d",sr,nr);
        printf("(%u,%u)", sr, nr);
        read_sr_fad_raw(&ErrCode, &start_i, &end_i, sr, nr, (UBYTE1 *)&FadRaw);
        if (ErrCode) {
            dbgeterr(ErrCode, ErrStr);
            printf("%s\n", ErrStr);
            fprintf(fh,"%s\n", ErrStr);
        }
        fprintf(fh,"\n");
        printf("\n");
        for (turn = 0; turn < 1023; turn++) {
            fprintf(fh,"%d\t", turn);
            for (b = 0; b < 4; b++) {
                if (ErrCode)
                    fprintf(fh,"%u ", 0.0);
                else
                    fprintf(fh,"%u ", FadRaw.button[b][turn]);
                fprintf(fh,"\t");
            }
            fprintf(fh,"\n");
        }
        fprintf(fh,"\n");
    }
    fprintf(fh,"\n");
}

printf("%u turns\n",(end_i-start_i+1));

}

```


Glossary of Terms

A/D

Analog to Digital converter. For BPMs, converts the analog signal from the beam pickup to a digital value read by the ILC.

BPM

Beam Position Monitor. A non destructive position monitor using 4 pickups. There are 32 in the booster and 96 in the storage ring.

buttons

The pickups used by the beam position monitors.

calibration

To compensate for the differences in the amplifiers connected to the BPM buttons, the ILC can be told to perform a calibration.

CMM

Collector Micro Module located in the control room keeps the complete database. Applications read data from this data base.

continuous trigger

A mode of fast data acquisition. Storage ring fast ILCs can be told to acquire data every time a fast trigger is detected. In single trigger mode the fifo data is frozen until the fast is told by an application to initialize.

FAD

Fast Analog to Digital Converter used by BPMs to digitize signals from the pickups to 8 bit accuracy at the rate of the beam revolution.

Halt Trigger

A trigger brought into the FAD electronics that stops the FIFO data gathering. This trigger occurs once a second in the booster.

ILC

Intelligent Local Controller. The computer built into the BPM electronics to provide local control functions such as xy calculations and to communicate with the main control system.

Slow FIFO

Slow First-In-First-Out buffer that has space for 1023 entries used to store up to 1023 turns of data digitized at either the at 1,10,100, or 1000 times the beam revolution frequency.

slow mode

A slow mode reading refers to the 16 bit A/D reading valid when more than 1 ms of circulating beam is present.

triggering

A fad is 'triggered' when an external signal tells the fad to stop acquiring data.

Index

D

DbGetSync 9, 13
DbSetSync 8, 13

F

ffad_read 10

G

GetFadStatus 9, 13

I

init_br_fad 8, 11
init_sr_fad 8, 12

R

read_br_fad_quick 10
read_br_fad_raw 9
read_br_fad_xy 9
read_sr_fad_raw 9
read_sr_fad_xy 10

S

select_sr_fad_fifo 8, 12
set_br_fadmode 8, 11
set_sr_fadmode 8
start_br_fads 8

W

wait_br_fad_trigger 8
wait_sr_fad_trigger 8