XBDRV

Software Reference
for System II XBUS Devices

Software version 2.20
Printed 10/1/98

Device Index

AD1 Dual A/D Converter 1401
AD2 Instrumentation A/D Converter 1601
Auto-detecting calls 1801
CG1 Clock Generator 401
DA1 Dual D/A Converter 1301
DA3-2/4/8 Multi-Channel Instrumentation D/A 1701
DB4 Digital Biological Amplifier 2301
DD1 Stereo Analog Interface 1501
ET1 Event Timer 601
HTI Head Tracker Interface 2201
PA4 Programmable Attenuator 201
PD1 Power SDAC 2101
PF1 Programmable Filter 901
PI1 Parallel Interface 701
PI2 Parallel Interface 1001
PM1 Power Multiplexer 2001
SD1 Spike Discriminator 501
SS1 Programmable Signal Switcher 1901
SW2 Cosine Switch 301
TG6 Timing Generator 1101
WG1 Waveform Generator 801
WG2 Waveform Generator 1201
XB1 Device Caddie 101

(this page intentionally left blank)

Table of Contents

DEVICE INDEX 1

INTRODUCTION 1

SOFTWARE SETUP 3

Turbo and Microsoft 'C' Support *3*

Turbo Pascal Support *4*

Support for Other Compilers 5

Controlling Two XBUSes *5*

XBDRV NUTS & BOLTS 7

Control Constants *7*

Device Type Identification Codes *8*

Miscellaneous Communication Constants *9*

Physical Drivers *10*

Command Form Output Procedures *13*

Error Handling *15*

Receiving Data Across XBUS *17*

PROCEDURE DESCRIPTIONS 19

Description Format *20*

XB1 DEVICE CADDIE 101

XB1init(cprt) 101

XB1flush( ) 103

XB1device(dev, din) 103

XB1gtrig( ) 103

XB1ltrig(int rn) 104

XB1version(int dev, int din) 104

XB1select(int cprt) 106

PA4 PROGRAMMABLE ATTENUATOR 201

PA4atten(din, level) 201

PA4setup(din, base, step) 201

PA4auto(din) 202

PA4man(din) 202

PA4mute(din) 202

PA4nomute(din) 202

PA4ac(din) 203

PA4dc(din) 203

PA4read(din) 203
SW2 COSINE SWITCH 301
SW2on(din) 301
SW2off(din) 301
SW2ton(din) 302
SW2toff(din) 302
SW2shape(din, scode) 303
SW2rftime(din, rftime) 303
SW2trig(din, tcode) 304
SW2dur(din, dur) 304
SW2status(din) 305
SW2clear(din) 305
CG1 CLOCK GENERATOR 401
CG1go(din) 401
CG1tgo(din) 401
CG1stop(din) 402
CG1trig(din, tcode) 402
CG1reps(din, nreps) 403
CG1period(din, period) 403
CG1active(din) 405
SD1 SPIKE DISCRIMINATOR 501
SD1go(din) 501
SD1stop(din) 501
SD1use_enable(din) 501
SD1no_enable(din) 502
SD1hoop(din, hnum, scode, dly, wid, upper, lower) 502
SD1numhoops(din, numhoops) 503
SD1count(din) 504
SD1up(din, cbuf) 504
SD1down(din, cbuf) 505
ET1 EVENT TIMER 601
ET1clear(din) 601
ET1go(din) 601
ET1stop(din) 602
ET1read16(din) 602
ET1read32(din) 603
ET1drop(din) 603

ET1report(din) 603
ET1active(din) 604
ET1compare(din) 604
ET1evcount(din) 605
ET1blocks(din, numblocks) 605
ET1mult(din) 607
ET1xlogic(din, lmask) 608
PI1 PARALLEL INTERFACE 701
Overview *701*
PI1clear(din) 703
PI1outs(din, omask) 703
PI1logic(din, logout, login) 704
PI1write(din, bitcode) 704
PI1read(din) 705
PI1autotime(din, atmask, dur) 705
PI1map(din, incode, mapoutmask) 706
PI1latch(din, lmask) 709
PI1debounce(din) 710
PI1strobe(din) 711
PI1optread(din) 711
PI1optwrite(din, bitcode) 711
WG1 WAVEFORM GENERATOR 801
WG1on(din) 801
WG1off(din) 801
WG1ton(din) 802
WG1clear(din) 802
WG1amp(din, amp) 803
WG1freq(din, freq) 803
WG1swrt(din, swrt) 804
WG1phase(din, phase) 804
WG1dc(din, dc) 805
WG1shape(din, scode) 805
WG1dur(din, dur) 806
WG1rf(din, rf) 806
WG1trig(din, tcode) 807
WG1seed(din, seed) 807
WG1delta(din, delta) 808

WG1wave(din, wave, npts) ... 808
WG1status(din) ... 809
PF1 PROGRAMMABLE FILTER ... 901
Overview ... *901*
PF1freq(din, lpfreq, hpfreq) ... 902
PF1gain(din, lpgain, hpgain) ... 902
PF1bypass(din) ... 903
PF1nopass(din) ... 903
PF1fir16(din, bcoefs, ntaps) ... 904
PF1fir32(din, bcoefs, ntaps) ... 904
PF1iir32(din, bcoefs, acoefs, ntaps) ... 905
PF1biq16(din, bcoefs, acoefs, nbiqs) ... 906
PF1biq32(din, bcoefs, acoefs, nbiqs) ... 907
PF1type(din, type, n) ... 908
PF1begin(din) ... 909
PF1b16(din, coeff) ... 909
PF1a16(din, coeff) ... 910
PF1b32(din, coeff) ... 910
PF1a32(din, coeff) ... 910
PI2 PARALLEL INTERFACE ... 1001
Overview ... *1001*
PI2clear(din) ... 1003
PI2outs(din, omask) ... 1003
PI2logic(din, logout, login) ... 1004
PI2debounce(din, dbtime) ... 1004
PI2latch(din, lmask) ... 1005
PI2autotime(din, bitn, dur) ... 1005
PI2toggle(din, tmask) ... 1006
PI2map(din, bitn, mmask) ... 1006
PI2zerotime(din, bitmask) ... 1010
PI2gettime(din, bitn) ... 1010
PI2write(din, bitcode) ... 1012
PI2read(din) ... 1012
PI2setbit(din, bitmask) ... 1013
PI2clrbit(din, bitmask) ... 1013
PI2outsX(din, pnum) ... 1015
PI2writeX(din, pnum, bitcode) ... 1015

PI2readX(din, pnum) 1016
TG6 TIMING GENERATOR 1101
Overview *1101*
Binary/Hexadecimal Conversion Chart *1102*
TG6clear(din) 1103
TG6arm(din, snum) 1103
TG6go(din) 1103
TG6tgo(din) 1104
TG6stop(din) 1104
TG6baserate(din, brcode) 1104
TG6reps(din, rmode, rcount) 1105
TG6new(din, snum, lgth, dmask) 1106
TG6high(din, snum, _beg, _end, hmask) 1107
TG6low(din, snum, _beg, _end, lmask) 1108
TG6value(din, snum, _beg, _end, val) 1109
TG6dup(din, snum, s_beg, s_end, d_beg, ndup, dmask) 1110
TG6status(din) 1111
WG2 WAVEFORM GENERATOR 1201
WG2on(din) 1201
WG2off(din) 1201
WG2ton(din) 1202
WG2clear(din) 1202
WG2amp(din, amp) 1203
WG2freq(din, freq) 1203
WG2swrt(din, swrt) 1203
WG2phase(din, phase) 1204
WG2dc(din, dc) 1205
WG2shape(din, scode) 1205
WG2dur(din, dur) 1206
WG2rf(din, rf) 1206
WG2trig(din, tcode) 1207
WG2seed(din, seed) 1207
WG2delta(din, delta) 1208
WG2wave(din, wave, npts) 1208
WG2status(din) 1209
DA1 DUAL D/A CONVERTER 1301
DA1clear(din) 1301

DA1arm(din) 1301
DA1go(din) 1302
DA1stop(din) 1302
DA1tgo(din) 1302
DA1mode(din, mcode) 1303
DA1strig(din) 1304
DA1mtrig(din) 1304
DA1reps(din, nreps) 1305
DA1srate(din, sper) 1305
DA1speriod(din, sper) 1306
DA1npts(din, npts) 1307
DA1clkin(din, scode) 1308
DA1clkout(din, dcode) 1308
DA1clipon(din) 1309
DA1status(din) 1309
DA1clip(din) 1309
AD1 DUAL A/D CONVERTER 1401
AD1clear(din) 1401
AD1arm(din) 1401
AD1go(din) 1401
AD1stop(din) 1402
AD1tgo(din) 1402
AD1mode(din, mcode) 1403
AD1strig(din) 1403
AD1mtrig(din) 1404
AD1reps(din, nreps) 1404
AD1srate(din, sper) 1405
AD1speriod(din, sper) 1406
AD1npts(din, npts) 1406
AD1clkin(din, scode) 1407
AD1clkout(din, dcode) 1407
AD1clipon(din) 1408
AD1status(din) 1408
AD1clip(din) 1409
DD1 STEREO ANALOG INTERFACE 1501
DD1clear(din) 1501
DD1arm(din) 1501

DD1go(din) 1501
DD1stop(din) 1502
DD1tgo(din) 1502
DD1mode(din, mcode) 1503
DD1strig(din) 1504
DD1mtrig(din) 1504
DD1reps(din, nreps) 1505
DD1srate(din, sper) 1505
DD1speriod(din, sper) 1506
DD1npts(din, npts) 1507
DD1clkin(din, scode) 1507
DD1clkout(din, dcode) 1508
DD1clipon(din) 1508
DD1status(din) 1509
DD1clip(din) 1509
AD2 INSTRUMENTATION A/D CONVERTER 1601
AD2clear(din) 1601
AD2arm(din) 1601
AD2go(din) 1602
AD2stop(din) 1602
AD2tgo(din) 1602
AD2sh(din, shmode) 1603
AD2mode(din, mcode) 1604
AD2xchans(din, nchans) 1604
AD2gain(din, chan, gain) 1605
AD2strig(din) 1605
AD2mtrig(din) 1606
AD2reps(din, nreps) 1606
AD2srate(din, sper) 1607
AD2speriod(din, sper) 1608
AD2sampsep(din, sampsep) 1608
AD2npts(din, npts) 1609
AD2clkin(din, scode) 1609
AD2clkout(din, dcode) 1610
AD2status(din) 1610
AD2clip(din) 1611
DA3-2/4/8 MULTI-CHANNEL INSTRUMENTATION D/A 1701

DA3clear(din) ... 1701
DA3arm(din) ... 1701
DA3go(din) ... 1702
DA3stop(din) ... 1702
DA3tgo(din) ... 1703
DA3mode(din, mcode) ... 1703
DA3strig(din) ... 1704
DA3mtrig(din) ... 1704
DA3reps(din, nreps) ... 1705
DA3srate(din, sper) ... 1705
DA3speriod(din, sper) ... 1706
DA3npts(din, npts) ... 1707
DA3clkin(din, scode) ... 1707
DA3clkout(din, dcode) ... 1708
DA3clipon(din) ... 1708
DA3status(din) ... 1709
DA3clip(din) ... 1709
DA3setslew(din, slcode) ... 1710
DA3zero(din) ... 1711
AUTO-DETECTING CALLS ... 1801
SS1 PROGRAMMABLE SIGNAL SWITCHER ... 1901
Overview ... *1901*
SS1clear(din) ... 1901
SS1gainon(din) ... 1901
SS1gainoff(din) ... 1902
SS1mode(din, mcode) ... 1902
SS1select(din, chan, inpn) ... 1903
PM1 POWER MULTIPLEXER ... 2001
PM1clear(din) ... 2001
PM1config(din, ccode) ... 2001
PM1mode(din, mcode) ... 2002
PM1spkon(din, sn) ... 2002
PM1spkoff(din, sn) ... 2003
PD1 POWER SDAC ... 2101
Overview ... *2101*
PD1 Referencing Variables ... *2102*
PD1 Basic Calls ... *2104*

PD1clear(din) 2104
PD1arm(din) 2105
PD1go(din) 2105
PD1stop(din) 2105
PD1mode(din, mcode) 2106
PD1nstrms(din, nIB, nOB) 2107
PD1tgo(din) 2107
PD1strig(din) 2108
PD1mtrig(din) 2108
PD1reps(din, nreps) 2108
PD1srate(din, sper) 2109
PD1speriod(din, sper) 2109
PD1npts(din, npts) 2110
PD1clkin(din, scode) 2110
PD1clkout(din, dcode) 2110
PD1status(din) 2111
PD1syncall(din) 2111
Route Scheduling Calls 2112
PD1clrsched(din) 2112
PD1addsimp(din, src, des) 2113
PD1addmult(din, srclst[], sf[], nsrcs, des) 2114
PD1specIB(din, ibn, des) 2115
PD1specOB(din, obn, src) 2116
DSP Calls 2117
PD1idleDSP(din, dmask) 2117
PD1bypassDSP(din, dmask) 2117
PD1resetDSP(din, dmask) 2118
PD1lockDSP(din, dmask) 2118
PD1interpDSP(din, ifact, dmask) 2119
PD1checkDSPS(din) 2119
PD1whatDSP(din, dspid) 2120
Analog I/O Calls 2121
PD1clrIO(din) 2121
PD1setIO(din, dt1, dt2, at1, at2) 2121
PD1whatIO(din) 2122
Delay Processor Calls 2123
PD1clrDEL(din, n0, n1, n2, n3) 2123

PD1flushDEL(din) 2124
PD1interpDEL(din, ifact) 2124
PD1setDEL(din, tapn, dly) 2125
PD1latchDEL(din) 2125
PD1whatDEL(din) 2126
HTI HEAD TRACKER INTERFACE 2201
Overview *2201*
HTIclear(din) 2202
HTIgo(din) 2203
HTIstop(din) 2203
HTIboresight(din) 2203
HTIshowparam(din, pid) 2204
HTIreadAER(din, &az, &el, &roll) 2204
HTIfastAER(din, &az, &el, &roll) 2205
HTIreadXYZ(din, &x, &y, &z) 2206
HTIfastXYZ(din, &x, &y, &z) 2206
HTIreadone(din, pid) 2207
HTIreset(din, pid) 2207
HTIgetecode(din) 2208
HTIsetraw(din, nbytes, c1, c2) 2208
HTIwriteraw(din, cmdstr[]) 2209
HTIreadraw(din, maxchars, &buf) 2209
HTIisISO(din) 2210
DB4 DIGITAL BIOLOGICAL AMPLIFIER 2301
Overview *2301*
DB4clear(int din); 2301
DB4setgain(int din, int chan, float gain); 2301
DB4selgain(int din, int chan, int gs); 2302
DB4setfilt(int din, int chan, int ftype, float ffreq); 2303
DB4selfilt(int din, int chan, int ftype, int fs); 2304
DB4userfilt(int din, int chan, int fn, float coef[]); 2305
DB4setIT(int din, int chan, int it); 2306
DB4nchan(int din, int nc); 2306
DB4setTS(int din, float amp, float freq); 2307
DB4onTS(int din); 2307
DB4offTS(int din); 2308
DB4startIM(int din, int chan); 2308

DB4stopIM(int din); 2308
DB4readIM(int din, int pc); 2309
DB4impscan(int din, int tochan); 2310
DB4getclip(int din); 2311
DB4getstat(int din); 2311
DB4powdown(int din); 2312
DB4getgain(int din, int chan, int *sel); 2312
DB4getfilt(int din, int chan, int ft, int *sel); 2313
DB4getIT(int din, int chan); 2314
DB4getchmode(int din); 2314
DB4getmud(int din); 2315
Binary/Decimal Conversion Chart 2316
Generating Filter Coefficients with Matlab 2318

(THIS PAGE INTENTIONALLY LEFT BLANK)

Introduction

The XBUS driver software package, **XBDRV,** is a complete, high-level language interface for the XBUS product line. Although direct programming of XBUS devices is a straightforward task, the XBDRV package adds another level of ease and versatility. The XBDRV software is a complete library of driver routines, ready to use with Turbo 'C' and Microsoft 'C' compilers. The XBDRV software is provided in source code format for easy compilation and for advanced programmers who may wish to customize or create new drivers. The source code listing also makes it possible to port XBDRV to non-PC computers and other compiler platforms.

Once you have installed and configured your System II hardware, the ***Software Setup*** section has a simple 'C' example to help you get started in programming your XBUS devices right away. The ***XBDRV Nuts & Bolts*** section contains advanced programming information, which is necessary only if you wish to modify or write your own driver routines.

The ***Procedure Descriptions*** section outlines the driver routines and associated parameters for each XBUS device. As new devices and their drivers become available, document updates will be provided on a per-device basis. The driver source code is also listed with each device driver description.

The basic format of all the XBUS device driver routines is similar. Once you get familiar with a few of them, writing programs to control your XBUS devices will be easy and fast.

Software Setup

Before proceeding, be certain that you have completed all the steps outlined in the *System II Installation Guide* and have verified that your hardware is working properly.

Turbo and Microsoft 'C' Support

The XBDRV 'C' source code is provided in a format fully compatible with Borland Turbo 'C'. If you are using the Microsoft 'C' compiler, you should define MSC either in the compile command line, or at the top of the program. If you are programming under Microsoft WINDOWS, you should also define WINDOWS. The XBDRV is also compatible with other 'C' compilers. If you are using a different 'C' compiler, please refer to the user's guide for your compiler (see *Support for Other Compilers* below).

Test your software installation by writing a simple 'C' program. The program listed below initializes your XBUS hardware using the **XB1init** procedure, and setup a PA4 attenuation (if you have a PA4 module). In this example we will use the Turbo 'C' compiler:

1. Invoke the Turbo 'C' editor environment and set it up to use proper memory model and other compiler/linker options..

2. Create a project file and include the following files in the project:

```
xbtst1.c
xbdrv.c
```

3. Clear the editor and type in the following program:

```
#include <stdlib.h>
#include <stdio.h>
#include "xbdrv.h"

void main(void)
{
```

```
    int i;
    clrscr();
    if(!XB1init(USE_DOS))
    {
      printf("\n Error initializing XBUS \n");
      exit(0);
    }
    for(i=0; i<1000; i++)
      PA4atten(1,(float)i/10.0);
  }
```

4. Be certain the compiler is setup properly and then compile, link, and run the program.

If this simple program compiles, links, and runs, the software is installed properly. If you experience trouble, check all of your hardware and software installation work and then call TDT.

Turbo Pascal Support

If you are using Turbo Pascal you will have to compile the provided source file into a TPU file using the Turbo Pascal compiler. Once the XBDRV.TPU file has been created it should be moved to the appropriate directory and linked to your application using the Turbo Pascal USES line in your source code. When compiling the drivers be certain the *'Range Checking'* compiler feature is *disabled*.

Test your software setup by writing a simple PASCAL program. The program shown below initializes your XBUS hardware using the **XB1init** procedure and programs the PA4 programmable attenuator to count from 00.0 to 99.9dB. Invoke the Turbo compiler and type in the following lines:

```
program  xbtst1;
uses crt,xbdrv;

var i:integer;

begin
  if not XB1init(USE_DOS) then
  begin
    writeln('Error initializing XBUS.');
    halt;
```

```
      end;

      for i:=0 to 999 do
        PA4atten(1,i/10.0)
    end.
```

If this program compiles, links, and runs, the driver software is installed properly. If you experience trouble, check all of your hardware and software installation work and then call TDT.

Support for Other Compilers

Because the XBDRV drivers are provided in 'C' source code format, they can be used with a variety of compilers and operating systems. In an effort to make the source code compatible with a wider range of compilers, the following directive constants can be used to customize the XBDRV source code:

MSC	turns on Microsoft 'C' compatibility
COMP32	support for 32 bit compilers
WINDOWS	removes all error display using *printf*

The XBDRV source code includes support for compilation with a 32-bit compiler such as WATCOM Version 9.0. When using this compiler, the MSC, COMP32, and INTMOVE defines must be made. Note that the INTMOVE define replaces the 16-bit assembly in IO.ASM with equivalent 'C' code.

Controlling Two XBUSes

Under most circumstances, the XBDRV software will be used to control a single XBUS connected to either the TDT, AP2 card, or a standard PC serial port. Under this condition the setup outlined in the 'Startup Guide' using the DOS parameter works best. Call XB1init(USE_DOS) at the start of each application program to initialize your system using the DOS parameter,.

If the XBUSes are interfaced to the PC via the AP2 card, two XBUSes can be controlled from a single application program. When using XBDRV with two XBUSes, the DOS parameter is not used. Under this condition the DOS utilities, such as XBCOMINI.EXE, must be called twice with a command line argument specifying each AP2. For example, the following calls should be made in your AUTOEXEC.BAT file to initialize a system with two AP2s and two XBUSes:

```
apld APa
apld APb
xbcomini APa
xbcomini APb
```

The XB1init call must also be made twice at the top of each application program. A typical program initialization might appear as follows:

```
if( !XB1init(APa) || !XB1init(APb))
{
  printf("\n Error initializing XBUS \n");
  exit(0);
}
```

XBUSes are then selected for control using the **XB1select** call. For example, to select the XBUS connected to APb, simply call XB1select(APb). XBUS command calls following this call will be sent to the XBUS that is controlled by APb.

XBDRV Nuts & Bolts

This section is intended for users with advanced programming skills who wish to customize or create new driver software routines. 'C' code listings are provided on disk. This section covers the portion of XBDRV.C which governs communication to and from XBUS devices and error handling. Code for individual device driver routines is listed in each device's section.

Commands and data are sent to XBUS devices using two procedures: one called **shortform**, which is used for handling short-form commands; and a second called **standform**, which is used when the standard-form communication protocol is needed. Also, an error handling procedure and a show-error procedure are used globally. For receiving data from devices, there are three procedures to upload single-byte, integer, and long integer values. These procedures are called **get8()**, **get16()**, and **get32()**, respectively.

Control Constants

XBDRV has several control constants which may be altered to better suit the user's needs. These constants will be found at the top of the XBDRV.H file. It is recommended these constants be altered only if problems occur while using XBDRV.

```
/*********************************************************
  User programmable control constants
 *********************************************************/
#define XB_MAX_DEV_TYPES    32
#define XB_MAX_OF_EACH_TYPE 16
#define XB_MAX_NUM_RACKS   120
#define XB_TIMEOUT        3000
#define XB_MAX_TRYS          3
#define XB_WAIT_FOR_ID      40
```

Device Type Identification Codes

XBDRV defines a unique Device Type Identification Code for each XBUS device. In the identification procedure, these codes are used as the index to the *xcode* matrix, which is where the XBUS Location Numbers (XLNs) are stored.

```
/*********************************************************
  Device ID codes
 *********************************************************/
#define PA4_CODE      0x01
#define SW2_CODE      0x02
#define CG1_CODE      0x03
#define SD1_CODE      0x04
#define ET1_CODE      0x05
#define PI1_CODE      0x06
#define UI1_CODE      0x07
#define WG1_CODE      0x08
#define PF1_CODE      0x09
#define TG6_CODE      0x0a
#define PI2_CODE      0x0b
#define WG2_CODE      0x0c
#define VC1_CODE      0x0d
#define SS1_CODE      0x0e
#define PM1_CODE      0x0f
#define HTI_CODE      0x1a

#define DA1_CODE      0x10
#define AD1_CODE      0x11
#define DD1_CODE      0x12
#define DA2_CODE      0x13
#define AD2_CODE      0x14
#define AD3_CODE      0x15
#define DA3_CODE      0x16
#define PD1_CODE      0x17
```

Miscellaneous Communication Constants

The XBDRV.H header file contains a number of constants used for miscellaneous communication procedures. These include a series of constants used to specify the communication port to be used, and the standard XBUS communications constants.

```
/***********************************************************
  Global communication/control constants
 ***********************************************************/
#define COM1     1
#define COM2     2
#define COM3     3
#define COM4     4
#define COM_AP2  5
#define COM_AP2x 6
#define COM_APa  5
#define COM_APb  6
#define USE_DOS  0

#define COM1base 0x3f8
#define COM2base 0x2f8
#define COM3base 0x3e8
#define COM4base 0x2e8
#define APabase  0x230
#define APbbase  0x250

#define SNOP          0x00
#define VER_REQUEST   0x06
#define XTRIG         0x07
#define IDENT_REQUEST 0x08
#define HOST_RTR      0x09

#define ARB_ERR       0xC0
#define HOST_ERR      0xC1
#define ERR_ACK       0xC2
#define SLAVE_ACK     0xC3
#define HARD_RST      0xC5
```

```
#define SLAVE_ERR     0xC6
#define BAUD_LOCK     0xCA
#define ARB_ACK       0xCB
#define ARB_ID        0xCC
#define ARB_RST       0xCD
#define GTRIG         0xD2
#define LTRIG         0xD3
#define ARB_VER_REQ   0xD4
```

Physical Drivers

Because XBDRV is written in standard 'C', it can be easily ported to other hardware and software platforms. However, when an IBM PC or compatible is not being used, a few short procedures must be written that are specific to that computer hardware. These 'Physical Drivers' are responsible for driving the RS232 port of the host computer.

Two physical drivers are used by XBDRV to drive the PC's RS232 interface: **iosend** and **iorec**. These procedures handle the sending and receiving of single-byte data to and from the RS232 port. In addition to these procedures, code must be written that initializes the port to the required protocol. Brief descriptions of the **iosend** and **iorec** procedures are given below:

iosend

The **iosend** procedure sends a single byte of data out the RS232 serial port. The procedure first checks if the serial port hardware is ready to send the byte of data. If after a 'TIME-OUT' period the port does not become ready, the procedure fails and sends an error message to the video monitor.

iorec

This procedure checks the serial port hardware for any received data. If data is available it is returned, otherwise the value of -1 is returned.

NOTE: The **iosend** and **iorec** procedures provided with XBDRV can drive all COM ports on the PC, as well as the RS232 port on TDT's AP2. This versatility is typically not needed in user-written physical drivers.

```
/****************************************************************
  Physical Drivers.

****************************************************************/
void iosend(qint datum)
{
  long ii;


  if(!xb_eflag)
  {
              if(cc<COM_APa)
    {
      ii=0;
      do
      {
       ii++;
       if(ii>xbtimeout)
       {
         showerr("Transmission Time-Out on COM Port.");
         break;
       }
      }
                   while(!(inportb(cpa[xbsel]+5) & 0x20));
                   outportb(cpa[xbsel],datum);
    }
    else
    {
      ii=0;
      do
      {
       ii++;
       if(ii>xbtimeout)
       {
         showerr("Transmission Time-Out on COM Port.");
         break;
       }
      }
      while(!(inportb(cpa[xbsel]) & 0x02));
      outportb(cpa[xbsel]+1,datum);
    }
  }
}


qint iorec( void )
{
```

```
  if(xb_eflag)
    return(0);

       if(cc<COM_APa)
  {
    if(inportb(cpa[xbsel]+5) & 0x01)
    {
      return(inportb(cpa[xbsel]));
    }
    else
    {
      return(-1);
    }
  }
  else
  {
    if(inportb(cpa[xbsel]) & 0x01)
    {
      return(inportb(cpa[xbsel]+1));
    }
    else
    {
      return(-1);
    }
  }
}
```

Command Form Output Procedures

XBDRV uses two procedures for sending XBUS commands in the *form* acknowledged by XBUS. The procedure **shortform** is used to send single-byte commands, and the **standform** procedure is employed for multi-byte commands. In both procedures, the common variable *xln* is set using the **getxln** procedure, and in the **standform** procedure, a checksum is calculated and appended to the transmission. Finally, both procedures use the **processerr** procedure for error handling.

```
void getxln(qint din, qint dtype, char descrip[])
{
       xln=xbcode[dtype][din][xbsel];
  strcpy(xcaller,descrip);
  if(!xln && !xb_eflag)
  {
    showerr("  Non-present device referenced.");
  }
}

void shortform(qint val)
{
  qint i,j;
  long k;

  if(!xb_eflag)
  {
    j=0;
    do
    {
      cmd[0] = val;
      j++;
      iosend(xln & 0x7f);
      iosend(val);
      k=0;
      do
      {
       i=iorec();
       k++;
      }while(i<0 && k < xbtimeout);

      if(i!=SLAVE_ACK)
       processerr(i,j);

    }while(i!=SLAVE_ACK && j<=XB_MAX_TRYS && !xb_eflag);
  }
}

void standform( unsigned char bufp[], qint n)
{
  qint i,j,x,cs;
  long k;

  if(!xb_eflag)
  {
    if(n>63)
```

```
    {
      showerr("Too many datum being moved with Standard Form.");
    }
    else
    {
      j=0;
      n++;
      do
      {
       j++;
       cs = 0;
       iosend(xln & 0x7f);
       iosend(0x40 | n);
       for(x=0; x<(n-1); x++)
       {
         iosend(bufp[x]);
         cs+=(qint)bufp[x];
       }
       iosend(cs);

       k=0;
       do
       {
         i=iorec();
         k++;
       }while(i<0 && k < xbtimeout);

       if(i!=SLAVE_ACK)
         processerr(i,j);

      }while(i!=SLAVE_ACK && j<=XB_MAX_TRYS && !xb_eflag);
    }
  }
}
```

Error Handling

XBDRV uses two procedures for handling communication errors. The **processerr** procedure is called when the SLAVE_ACK response is not received after a command is sent. **Processerr** attempts to decode the error.

If a data integrity error flag (ARB_ERR) is detected, the procedure returns after checking the *number-of-errors* (ne) counter. If this count exceeds NUM_TRYS, a transmission error message will be displayed and the program will abort.

If a SLAVE_ERR flag is detected, the **processerr** procedure will receive the error message from the slave device and display it on the video monitor using the **showerr** procedure.

```
void processerr( qint li, qint ne)
{
  long x;
  qint i,j;
  char sss[100];

  switch(li)
  {
    case ARB_ERR:
    {
      if(ne>XB_MAX_TRYS)
      {
        XB1flush();
        showerr("Communication errors.");
        break;
      }
      else
      {
        XB1flush();
      }
      break;
    }

    case SLAVE_ERR:
    {
      tdtdisab();
      j=0;
      x=0;
      do
      {
        i=iorec();
        if(i>0)
        {
          sss[j]=(i);
         j++;
         x=0;
       }
       x++;
```

```
      }while(i!=0 && j<200 && x<xbtimeout);
      sss[j]=0;
      enable();
      showerr(sss);
      break;
    }

    default:
    {
      if(ne>XB_MAX_TRYS)
        showerr("No XBUS device responding at XBL accessed.");
      break;
    }
  }
}

void showerr( char eee[])
{
  qint rn,pn,i;

  enable();
  rn = xln >> 2;
  pn = (xln & 3) + 1;
  sprintf(xb_err[0]," ");
  sprintf(xb_err[1],"XBUS Error:    Rack(%d),
Position(%d).",rn,pn);
  sprintf(xb_err[2],"    %s",xcaller);
  sprintf(xb_err[3],"    %s",eee);
  sprintf(xb_err[4]," ");
  sprintf(xb_err[5],"");
  xb_eflag = 1;

  if(!xb_emode)
  {
    i = 0;
    do
    {
      printf("%s \n",xb_err[i]);
      i++;
    }while(xb_err[i][0]!=0);
    exit(0);
  }
}
```

Version 2 of XBDRV allows you to trap XBDRV errors in your program. You can then terminate the program or take action to correct the error. Consult the *APOS/XBDRV Release Notes* for further details.

Receiving Data Across XBUS

XBDRV has three procedures used for receiving device data. The **get8()** procedure receives a single byte from the host device while **get16()** and **get32()** handle integer and long values, respectively.

Any time an XBUS device responds with data, the amount and type of data are always known by both the XBUS device and the HOST computer. When an XBUS command requires a device response, the response data is always sent immediately after the SLAVE_ACK. Therefore, on computer platforms where incoming serial data is NOT buffered automatically, it is necessary to hold system interrupts until the XBUS communication is complete.

```
unsigned char get8(void)
{
  qint i;
  long k;

  if(xb_eflag)
    return(0);

  k=0;
  do
  {
    i=iorec();
    k++;
  }while(i<0 && k < xbtimeout);
  if(i<0)
  {
    showerr("Device not responding with byte value after
SLAVE_ACK.");
    i=0;
  }
  return((unsigned char)i);
}

qint get16(void)
{
  qint i,bn;
  long k;
  qint *rv;
  unsigned char x[2];

  if(xb_eflag)
    return(0);

  rv = (qint *)x;
  k=0;
  for(bn=0; bn<2; bn++)
  {
    do
    {
      i=iorec();
      k++;
    }while(i<0 && k < xbtimeout);
```

```
    if(i>=0)
    {
      x[bn] = i;
      k=0;
    }
    else
    {
      showerr("Device not responding with integer value after
SLAVE_ACK.");
      *rv = 0;
    }
  }
  return(*rv);
}

unsigned long get32(void)
{
  qint i,bn;
  unsigned long k;
  unsigned long *rv;
  unsigned char x[4];

  if(xb_eflag)
    return(0);

  rv = (unsigned long *)x;
  k=0;
  for(bn=0; bn<4; bn++)
  {
    do
    {
      i=iorec();
      k++;
    }while(i<0 && k < xbtimeout);
    if(i>=0)
    {
      x[bn] = i;
      k=0;
    }
    else
    {
      showerr("Device not responding with long value after
SLAVE_ACK.");
      *rv = 0;
    }
  }
  return(*rv);
}
```

Procedure Descriptions

XBUS procedures are divided into groups based on the device being programmed. The following XBUS devices are currently supported by XBDRV:

- XB1 XBUS Device Caddie (0x00)
- PA4 Programmable Attenuator (0x01)
- SW2 Cosine Switch (0x02)
- CG1 Clock Generator (0x03)
- SD1 Spike Discriminator (0x04)
- ET1 Event Timer (0x05)
- PI1 Parallel Interface (0x06)
- WG1 Waveform Generator (0x08)
- PF1 Programmable Filter (0x09)
- TG6 Timing Generator (0x0a)
- PI2 Parallel Interface II (0x0b)
- WG2 Waveform Generator II (0x0c)
- SS1 Programmable Signal Switcher (0x0e)
- PM1 Power Multiplexer (0x0f)
- HTI Head Tracker Interface (0x1a)
- DA1 Dual D/A Converter (0x10)
- AD1 Dual A/D Converter (0x11)
- DD1 Stereo Analog Interface (0x12)
- AD2 500KHz Inst. A/D (0x14)
- DA3 Inst. D/A Converter (0x16)
- PD1 Power SDAC Convolver (0x17)
- HTI Head Tracker Interface (0X1a)
- DB4 Digital Bio-Amp (0X1b)

NOTE: The values within parentheses are the XBUS Device Type Identifiers (DTI). XBDRV declares constants equal to these values in the following way:

```
          #define dev_CODE dti
example:  #define PA4_CODE 0x01
```

Description Format

Each **XBDRV** procedure will be listed and described according to a specific format. Following is an illustration of this format for the **PA4atten** operation. The significance of each line is given after this example.

Example:

(1) **PA4atten(din, level)**

(2)	**Op Code**	PA4_ATT {0x20}
(3)	**Prototype**	void PA4atten(int *din*, float *level*);
(4)	**Description**	Programs the specified PA4 with the attenuation level specified. The attenuation *level* is rounded to the nearest tenth of a dB.
(5)	**Arguments**	*din* Device index number. *level* Attenuation in dB (0.0 ... 99.9).
(6)	**Example**	`PA4atten( 1, 20.0);` Programs PA4_(1) with an attenuation of 20.0dB.

Description:

(1) The **XBDRV** operation format and name. Usually, this procedure name will be very similar to the device Op Code.

(2) Corresponding device Op Code. This is the primary device operation code used to implement the required device function. Refer to individual device information sheets for more information on device Op Codes.

(3) 'C' prototype.

(4) Brief description of what the procedure does. This line will also disclose any limitations or special notes associated with the procedure.

(5) Description of argument(s). This description includes acceptable ranges for each argument.

(6) An example call (sometimes with other supporting calls) is given for each procedure, along with a brief description of what the call will do.

XB1 Device Caddie

XB1init(cprt)

Op Code NA

Prototype int XB1init(int *cprt*);

Description Initializes the XBUS system and returns true if successful. If the *cprt* parameter is passed with USE_DOS (0), the procedure checks the XBCOM DOS environment variable to identify and initialize the communication method being used. This DOS selection method can be overridden by specifying a communications port code in the *cprt* parameter. This override method is only recommended if you intend to control two XBUSes from a single PC.

Once the port is identified and initialized, the procedure then begins identifying XBUS modules until it finds an XB1 device caddie with NO programmable modules. Each time XB1init detects a device it sets the variable *xbcode* equal to the device's XBUS Location Number (XLN).

Arguments *cprt* -- XBUS Port

USE_DOS	Use DOS environment variable XBCOM (typical).
APa	Initialize XBUS on APa.
APb	Initialize XBUS on APb.

Example

```
XB1init(cprt);
```

If a single XB1 caddie contained two PA4s and two SW2s the *xbcode* variable would be set as follows:

xbcode[PA4_CODE][1][XBN] = 4
xbcode[PA4_CODE][2][XBN] = 5
xbcode[SW2_CODE][1][XBN] = 6
xbcode[SW2_CODE][2][XBN] = 7

All other *xbcode* values would be 0. Also see description for the **XB1device** command.

NOTE: XLNs range from 4-127 and XBN is the XBUS number 0 or 1 (try 0).

XB1flush( )

Op Code SNOP {0x00}

Prototype void XB1flush(void);

Description Flushes the communications data stream by sending 40 SNOPs (zeros).

Arguments None.

Example
```
XB1flush();
```
This will send 40 SNOPs to the communication stream.

XB1device(dev, din)

Op Code NA

Prototype int XB1device(int *dev*, int *din*);

Description Returns the XLN of the specified device if it was detected by **XB1init**, otherwise returns zero. Use the device code constants PA4_CODE, SW2_CODE, etc. in the *dev* parameter position. This procedure is handy for programs designed to run on different systems.

Arguments
dev Device type code. (See header file)
din Device index number.

Example
```
if(XB1device(PA4_CODE, 1))
  PA4atten(1,20.0);
```
The above code will check if PA4_(1) exists and if it does, the **PA4atten** call will set its attenuation level to 20.0 dB.

XB1gtrig( )

Op Code GTRIG {0xD3}

Prototype void XB1gtrig(void);

Description Sends a global trigger to all XBUS devices. Certain XBUS modules can be told to wait for this trigger before starting or stopping to provide precise software synchronization.
NOTE: This command works only with XBUS hardware versions 3.0 and above.

Arguments None.

Example
```
CG1tgo(1);
CG1tgo(2);
XB1gtrig();
```
This triggers CG1_(1) and CG1_(2) at the same time.

XB1ltrig(int rn)

Op Code LTRIG {0xD4}

Prototype void XB1ltrig(int *rn*);

Description Sends a local trigger to all XBUS devices in the specified XB1 Caddie rack number. Certain XBUS modules can be told to wait for this trigger before starting or stopping to provide precise software synchronization.
NOTE: This command works only with XBUS hardware versions 3.0 and above.

Arguments *rn* XB1 Caddie rack number.

XB1version(int dev, int din)

Op Code NA

Prototype int XB1version(int *dev*, int *din*);

Description Returns the hardware version of the specified XB1 Caddie or XBUS device module. If *dev* <= 0, returns the version of caddie (arbiter) *din*.
NOTE: This command works only with XBUS hardware versions 3.0 and above.

Arguments *dev* Device type ID code.

din Device index number or caddie number.

Example

```
XB1version(0x11, 1);
```

This returns version number of AD1_(1).

XB1select(int cprt)

Op Code NA

Prototype void XB1select(int *cprt*);

Description When controlling a system with two AP2s and two XBUSes, XB1select( ) is used to toggle control between the XBUSes. Do not use this call unless you have two AP2s and two XBUSes in a single computer.

Arguments *cprt* -- XBUS Command Port.

APa	Select XBUS connected to APa.
APb	Select XBUS connected to APb.

Example

```
XB1select(APa);
```

This call selects the XBUS connected to APa. Commands following this call will take effect in this selected XBUS.

PA4 Programmable Attenuator

PA4atten(din, level)

Op Code PA4_ATT {0x20}

Prototype void PA4atten(int *din*, float *level*);

Description Programs the specified PA4 with the attenuation level specified. The attenuation *level* is rounded to the nearest tenth of a dB.

Arguments
din Device index number
level Attenuation in dB (0.0 ... 99.9).

Example
```
PA4atten( 1, 20.0);
```
Programs PA4_(1) with an attenuation of 20.0 dB.

PA4setup(din, base, step)

Op Code PA4_SETUP {0x17}

Prototype void PA4setup(int *din*, float *base*, float *step*);

Description Configures the STEP mode on the PA4. The *base* parameter sets a base attenuation calibration level, and *step* specifies the attenuation step size.

Arguments
din Device index number.
base Base attenuation in dB (0.0 ... 99.9).
step Attenuation step size in dB (0.1 ... 10.0).

Example
```
PA4setup( 1, 20.0, 3.0);
```
This call would set the PA4 STEP mode to have 20.0 dB of base attenuation and step sizes of 3.0 dB.

PA4auto(din)

Op Code PA4_AUTO {0x13}

Prototype void PA4auto(int *din*);

Description Programs the specified PA4 in AUTO mode to update attenuation dynamically while you dial in.

Arguments *din* Device index number.

Example
```
PA4auto(2);
```
This sets PA4_(1) in AUTO mode.

PA4man(din)

Op Code PA4_MAN {0x14}

Prototype void PA4man(int *din*);

Description Programs the PA4 to operate in MANUAL update mode (enter the attenuation to take effect).

Arguments *din* Device index number.

Example
```
PA4man(1);
```
This sets PA4_(1) in MANUAL mode.

PA4mute(din)

Op Code PA4_MUTE {0x15}

Prototype void PA4mute(int *din*);

Description Mutes the specified PA4.

Arguments *din* Device index number.

Example
```
PA4mute(1);
```

PA4nomute(din)

Op Code PA4_NOMUTE {0x16}

Prototype void PA4nomute(int *din*);

Description Takes PA4 out of MUTE mode. If the PA4 was not in MUTE mode the call will have no effect.
Arguments *din* Device index number.
Example `PA4nomute(1);`

PA4ac(din)

Op Code PA4_AC {0x11}
Prototype void PA4ac(int *din*);
Description Places the specified PA4 in AC coupling mode.
Arguments *din* Device index number.
Example `PA4ac(1);`
This places PA4_(1) in AC coupling mode.

PA4dc(din)

Op Code PA4_DC {0x12}
Prototype void PA4dc(int *din*);
Description Places the specified PA4 in DC coupling mode.
Arguments *din* Device index number.
Example `PA4dc(1);`
This places PA4_(1) in DC coupling mode.

PA4read(din)

Op Code PA4_READ {0x18}
Prototype float PA4read(int *din*);
Description Returns the current attenuation setting from the specified PA4.
Arguments *din* Device index number.
Example

```
printf(" Current attenuation: %f \n",
PA4read(1));
```

This displays the current attenuation setting of PA4(1).

SW2 Cosine Switch

SW2on(din)

Op Code SW2_ON {0x11}

Prototype void SW2on(int *din*);

Description When gating is controlled from software, **SW2on** tells the specified SW2 to start gating its input signal ON with the specified shape. **SW2on** has no effect when the SW2 is programmed for external enable gating control (see **SW2trig**).

Arguments *din* Device index number.

Example

```
SW2trig(1,COMPUTER);
SW2on(1);
```

This will turn on SW2_(1).

SW2off(din)

Op Code SW2_OFF {0x12}

Prototype void SW2off(int *din*);

Description When gating is controlled from software, **SW2off** tells the specified SW2 to start gating its input signal OFF with the specified shape. **SW2off** has no effect when the SW2 is programmed for external enable gating control (see **SW2trig**).

Arguments *din* Device index number.

Example

```
SW2off(1);
```

This will turn off SW2_(1).

SW2ton(din)

Op Code SW2_TON {0x13}

Prototype void SW2ton(int *din*);

Description Similar to **SW2on**, but causes the specified SW2 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
SW2trig(1,COMPUTER);
SW2ton(1);
SW2ton(2);
XB1gtrig();      /* Global trigger */
```

This will cause SW2_(1) and SW2_(2) to begin gating ON at precisely the same time.

SW2toff(din)

Op Code SW2_TOFF {0x14}

Prototype void SW2toff(int *din*);

Description Similar to **SW2off**, but causes the specified SW2 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
SW2toff(1);
SW2toff(2);
XB1gtrig();      /* Global trigger */
```

This will cause SW2_(1) and SW2_(2) to begin gating OFF at precisely the same time.

SW2shape(din, scode)

Op Code SW2_SHAPE {0x16}

Prototype void SW2shape(int *din*, int *scode*);

Description Selects the SW2 ON/OFF gating shape. The valid shape types are listed below.

Arguments *din* Device index number.

scode Shape type code (0...7).

ONOFF	ON/OFF Switch.
COS2	Cos^2 function.
COS4	Cos^4 function.
COS6	Cos^6 function.
RAMP	Linear ramp function.
RAMP2	$Ramp^2$ function.
RAMP4	$Ramp^4$ function.
RAMP6	$Ramp^6$ function.

Example

```
SW2shape(1,COS2);
SW2on(1);
delay(1000);
SW2off(1);
```

This will program SW2_(1) to gate its input ON with a cos^2 shape and then OFF after about one second with the same shape.

SW2rftime(din, rftime)

Op Code SW2_RFTIME {0x15}

Prototype void SW2rftime(int *din,* float *rftime*);

Description Sets the rise and fall time of the gate shape. The time is taken between the 10% and 90% points of minimum and maximum gate amplitude.

Arguments *din* Device index number.

rftime Rise and fall time of gate in ms (0.5 ... 99.9).

Example See **SW2dur**.

SW2trig(din, tcode)

Op Code SW2_TRIG {0x17}

Prototype void SW2trig(int *din*, int *tcode*);

Description Programs the gating control mode of the SW2. The valid enable types are listed below.

Arguments

din	Device index number.	
tcode	Trigger type code (0...4).	
0	MANUAL/ COMPUTER	Manual or software Control of gating.
1	POS_EDGE	Rising-edge external trigger.
2	NEG_EDGE	Falling-edge external trigger.
3	POS_ENABLE	High level external enable.
4	NEG_ENABLE	Low level external enable.

Example See **SW2dur**.

SW2dur(din, dur)

Op Code SW2_DUR {0x18}

Prototype void SW2dur(int *din,* float *dur*);

Description Sets/enables the SW2's self-timing feature. When the self-timer is enabled, the SW2 will gate ON when a valid trigger is detected and gate OFF after the specified duration. Specify a duration of zero for continuous output after triggering.

NOTE: **SW2dur** does not work when using POS_ENABLE or NEG_ENABLE gating control.

Arguments

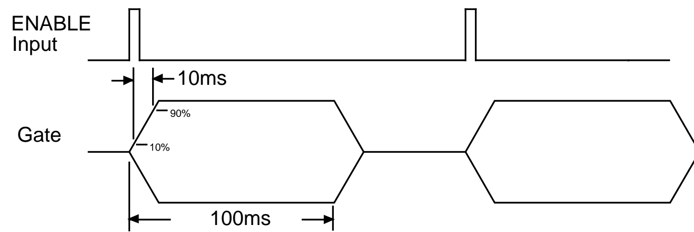
din Device index number.

dur Duration of gate in ms (1...9999).

Example

```
SW2trig(1,POS_EDGE);
SW2dur(1,100.0);
SW2rftime(1,10.0);
SW2shape(1,RAMP);
SW2on(1);
```

This example will produce the gate shown below in response to an external trigger:



SW2status(din)

Op Code SW2_STATUS {0x19}

Prototype int SW2status(int *din*);

Description Returns the status of the specified SW2.

Return Values:

OFF = 0	SW2 is off (no output).
ON = 1	SW2 gate fully on.
RISING = 2	Gate is in rising state.
FALLING = 3	Gate is in falling state.

Arguments *din* Device index number.

Example

```
if(SW2status(1)==ON);
  SW2off(1);
```

Turns SW2(1) off if it is on.

SW2clear(din)

Op Code SW2_CLEAR {0x20}

Prototype void SW2clear(int *din*);

Description Clears the specified SW2 and resets it to the factory default setup:

Triggering:	MANUAL/COMPUTER
Gate Shape:	Cos^2 function
Rise/Fall time:	10.0 ms
Gate duration:	Continuous

Arguments *din* Device index number.

Example

```
SW2clear(1);
```

This will reset SW2_(1) to factory default.

CG1 Clock Generator

CG1go(din)

Op Code CG1_GO {0x11}

Prototype void CG1(int *din*);

Description Tells the specified CG1 to begin producing the programmed output.

Arguments *din* Device index number.

Example

```
CG1trig(1,POS_EDGE);
CG1go(1);
```

This issues a manual trigger on CG1_(1).

CG1tgo(din)

Op Code CG1_TGO {0x19}

Prototype void CG1tgo(int *din*);

Description Similar to **CG1go**, but causes the specified CG1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
CG1trig(1,NONE);
CG1tgo(1);
CG1tgo(2);
XB1gtrig();            /* Global trigger */
```

This will cause CG1_(1) and CG1_(2) to begin clocking at precisely the same time.

CG1stop(din)

Op Code CG1_STOP {0x12}

Prototype void CG1stop(int *din*);

Description Places the specified CG1 in IDLE mode.

Arguments *din* Device index number.

Example

```
CG1stop(1);
```

This will stop CG1_(1) from producing output.

CG1trig(din, tcode)

Op Code CG1_TRIG {0x14}

Prototype void CG1trig(int *din*, int *tcode*);

Description Programs the trigger/enable input of the CG1. The valid trigger types are listed below.

NOTE: When using an enable type triggering the CG1 repetition counter is disabled (see **CG1reps**).

Arguments

din Device index number.

tcode Trigger type code (1...5).

POS_EDGE	Rising edge triggered.
NEG_EDGE	Falling edge triggered.
POS_ENABLE	High level enabled.
NEG_ENABLE	Low level enabled.
NONE	Free running.

Example

```
CG1trig(1,POS_EDGE);
```

This will program CG1_(1) to rising edge triggered. If no input is applied to the CG1 'ENABLE' input, a following **CG1go** call will manually trigger the device.

CG1reps(din, nreps)

Op Code CG1_REPS {0x13}

Prototype void CG1reps(int *din*, int *nreps*);

Description Programs the number of periods of the specified waveform the CG1 will output each time it is triggered. Remember that this feature is disabled when the trigger/enable input is programmed to POS_ENABLE or NEG_ENABLE (see **CG1trig**).

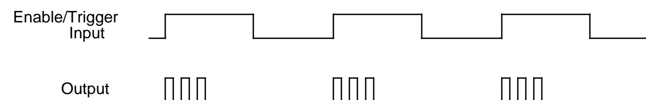
Arguments *din* Device index number.

nreps Number of periods to output (0...30000). Use *nreps*=0 for continuous output.

Example

```
CG1trig(1,POS_EDGE);
CG1reps(1,3);
```

The following output will be generated if the CG1 is programmed with the above two calls, and its trigger input is clocked from an external source.



CG1period(din, period)

Op Code CG1_PERIOD {0x15}

Prototype void CG1period(int *din*, float *period*);

Description Sets the period (frequency) of the CG1 output with a 50% duty cycle.

Arguments *din* Device index number.

period Clock cycle period in µs (1 ... $20 \cdot 10^6$).

Example

```
CG1period(1,100.0);
```

This will program the CG1 to generate a 10kHz clock signal with 50% duty cycle.

CG1puse(din, on_t, off_t)

Op Code CG1_PULSE {0x16}

Prototype void CG1pulse(int *din*, float *on_t*, float *off_t*);

Description Programs the CG1 to generate a signal with a variable duty cycle. The *on_t* parameter specifies the time the output will be high, while *off_t* indicates how long the signal will be low. The overall period of the waveform is *on_t* + *off_t*.

Arguments

din Device index number.

on_t Output high duration in μseconds (2000 ... 20,000,000).

off_t Output low duration in μseconds (2000 ... 20,000,000).

Example

```
CG1pulse( 1, 7500.0, 2500.0);
```

This call will program the CG1 to generate a 100Hz signal with a 75% duty cycle.

CG1patch(din, pcode)

Op Code CG1_PATCH {0x18}

Prototype void CG1patch(int *din*, int *pcode*);

Description Programs the CG1 to patch its output to one of the XBUS trigger/clocking signals.

Arguments

din Device index number.

pcode Patch selection code (1...5).

XTRG1	XBUS Trig/Sync #1.
XTRG2	XBUS Trig/Sync #2.
XCLK1	XBUS Clock #1.
XCLK2	XBUS Clock #2.
EXT	Front Panel BNC only.

Example

```
CG1patch(1,XCLK1);
```

This call will program CG1_(1) to assert its output to XBUS clock line #1.

CG1active(din)

Op Code CG1_ACTIVE {0x17}

Prototype char CG1active(int *din*);

Description Returns false if the selected CG1 is in idle mode or is waiting for a trigger; otherwise CG1active returns true.

Arguments *din* Device index number.

Example The CG1active call can be used to check if a pulse sequence has finished so response data can be collected, as illustrated below:

```
if (!CG1active(1))
{  read in spike times }
```

(this page intentionally left blank)

SD1 Spike Discriminator

SD1go(din)

Op Code SD1_GO {0x11}

Prototype void SD1go(int *din*);

Description Tells the SD1 to begin spike discrimination; i.e., takes the device out of IDLE mode. Note that this is the default state of the SD1.

Arguments *din* Device index number.

Example
```
SD1go(1);
```
Programs SD1 to begin looking for spikes.

SD1stop(din)

Op Code SD1_STOP {0x12}

Prototype void SD1stop(int *din*);

Description Places the SD1 in IDLE mode. In this mode, no pulses are sent from the **First** or **Last** outputs. Use the SD1's IDLE mode if your Time-Stamper/ Counter does not have an enable input.

Arguments *din* Device index number.

Example
```
SD1stop(1);
```

SD1use_enable(din)

Op Code SD1_ENABLE {0x13}

Prototype void SD1use_enable(int *din*);

Description Tells the specified SD1 to use its ENABLE input. When active, this input will 'enable' spike discrimination whenever it is high. Discrimination is disabled (IDLE mode) when the input is low.

Arguments *din* Device index number.

Example
```
SD1use_enable(1);
```

SD1no_enable(din)

Op Code SD1_NOENABLE

Prototype void SD1no_enab(int *din*);

Description Tells the SD1 to ignore its ENABLE input.

Arguments *din* Device index number.

Example

```
SD1no_enable(1);
```

SD1hoop(din, hnum, scode, dly, wid, upper, lower)

Op Code SD1_HOOP {0x15}

Prototype void SD1hoop(int *din*, int *hnum*, int *scode*, float *dly*, float *wid*, float *upper*, float *lower*);

Description Specifies a discrimination hoop on the SD1. The *dly* and *wid* parameters have no significance when *hnum* = 1. Hoop 1 serves as a trigger, with adjustable level and slope only. Set the lower level of hoop 1 to just under the upper level. The values of *dly* and *wid* for Hoops 2-5 must be set so that the hoops do not overlap in time (see below).

Arguments

din Device index number.

hnum Hoop Number (1...5).

scode Slope selection code (1...5).

RISE	Positive Slope.
FALL	Negative Slope.
PEAK	Pos. to Neg. Transition.
VALLEY	Neg. to Pos. Transition.
ANY	Pos. or Neg. Slope.

dly Delay time from Hoop 1 (0.10...5.20 ms).
$dly_i + wid_i + .08\text{ms} < dly_{i+1}$; i = *hnum*

wid Time width of hoop (0.011...0.830 ms).

upper Upper voltage level of hoop (-9.8...9.9V).

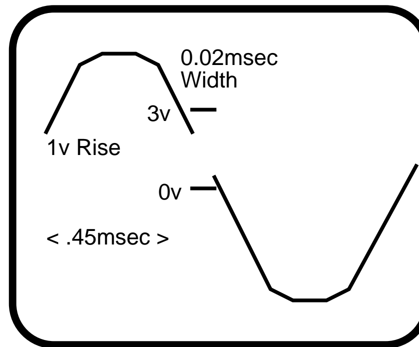
lower Lower voltage level of hoop (-9.9...9.8V), *lower* < *upper* .

Example

```
SD1hoop(1, 1, RISE, 0.0, 0.0, 1.0, 0.9);
SD1hoop(1, 2, FALL, 0.45, 0.02, 3.0, 0.0);
```

```
SD1numhoops(1,2);
```

The figure shown below illustrates what you should see on your scope after making the above series of SD1 calls. NOTE: You must connect a +/- 5 volt, 1KHz tone to the SD1 input and trigger your scope from the first output.



SD1numhoops(din, numhoops)

Op Code SD1_NUMHOOPS {0x16}

Prototype void SD1numhoops(int *din*, int *numhoops*);

Description Tells the SD1 how many hoops to use when discriminating spikes. Be careful not to tell the SD1 to use more hoops than you have specified using the SD1hoop call.

Arguments

din	Device index number.
numhoops	Number of hoops (1...5).

Example See the example given for **SD1hoop**.

SD1count(din)

Op Code SD1_COUNT {0x17}

Prototype int SD1count(int *din*);

Description The SD1 automatically keeps a count of the number of accepted spikes it discriminates. The **SD1count** call can be used to retrieve that count.

NOTE: The count is reset to zero each time **SD1go** is called. **SD1count** will return zero if Hoop 1 only is used.

Arguments *din* Device index number.

Example

```
SD1go(1);
{delay for 1 second.}
SD1stop(1);
printf("\n\nSpikes = %d\n\n", SD1count(1));
```

The above example will print the number of accepted spikes generated in a one second period.

SD1up(din, cbuf)

Op Code SD1_UP {0x18}

Prototype void SD1up(int *din,* char *cbuf[]*);

Description 'Uploads' the current configuration from the specified SD1. The configuration is sent as a binary image 32 bytes long.

Arguments *din* Device index number.

cbuf[] Pointer to a character buffer at least 32 bytes long.

Example

```
char cbuf[32];
SD1up(1,cbuf);
```

Uploads current SD1 configuration into PC memory.

SD1down(din, cbuf)

Op Code SD1_DOWN {0x19}

Prototype void SD1down(int *din*, char *cbuf[]*);

Description 'Downloads' a configuration from PC memory. The configuration is sent as a binary image 32 bytes long. NOTE: The only reasonable source of this binary image is to upload it from the SD1.

Arguments
din Device index number.
cbuf[] Pointer to a character buffer at least 32 bytes long.

Example

```
char cbuf[32];
SD1up(1,cbuf);
  { save cbuf to disk }
  { three days later }
  { load cbuf from disk }
SD1down(1,cbuf);
```

SD1up loads SD1_(1) configuration to a PC buffer *cbuf*, which can be saved to a file. Later this configuration can be retrieved and put in a PC buffer *cbuf* again. **SD1down** uses *cbuf* to configure SD1_(1).

(this page intentionally left blank)

ET1 Event Timer

ET1clear(din)

Op Code ET1_CLEAR {0x11}

Prototype void ET1clear(int *din*);

Description Resets the ET1 to the default factory configuration. The ET1 default configuration is as follows:

- Event Timing from Single Input.
- No Block Counting.
- Normal Enable.
- FIFO Cleared.

Arguments *din* Device index number.

Example

```
ET1clear(1);
```

Resets ET1_(1).

ET1go(din)

Op Code ET1_GO {0x15}

Prototype void ET1go(int *din*);

Description Takes the ET1 out of IDLE mode and tells it to begin recording/counting events.

NOTE: The ET1's timer/counter is automatically reset and a block marker (zero) is placed in the FIFO memory.

Arguments *din* Device index number.

Example

```
ET1go(1);
delay(100);
ET1stop(1);
```

This block of code will program the ET1 to record events for approximately 100 milliseconds.

ET1stop(din)

Op Code ET1_STOP {0x16}

Prototype void ET1stop(int *din*);

Description Places the ET1 in idle mode. In this mode, the ET1 timer/counter is halted and no data is written to the FIFO memory.

Arguments *din* Device index number.

Example See **ET1go**.

ET1read16(din)

Op Code ET1_READ {0x22}

Prototype int ET1read16(int *din*);

Description Reads a 16-bit value from the ET1's FIFO memory. If the FIFO is empty, the value of -1 is returned. Because only 16 bits are sent from the ET1, the **ET1read16** procedure is faster than **ET1read32**; however, the upper 16 bits of data are lost and short rollover times can be a problem.

Arguments *din* Device index number.

Example

```
i=0;
ET1go(1);
do
{
     tbuf[i] = ET1read16(1);
     if(tbuf[i] != -1) i++;
}while(i<100);
ET1stop(1);
```

This program block will record 100 event times in the PC buffer *tbuf*.

ET1read32(din)

Op Code ET1_READ32 {0x21}

Prototype unsigned long ET1read32(int *din*);

Description Reads a 32-bit timer/counter value from ET1 FIFO. If the FIFO is empty, a value of -1 is returned.

Arguments *din* Device index number.

Example See example for **ET1read16**.

ET1drop(din)

Op Code ET1_DROP {0x18}

Prototype void ET1drop(int *din*);

Description Clears the ET1's FIFO memory.

Arguments *din* Device index number.

Example

```
ET1go(1);
delay(100);
ET1stop(1);
for(i=0; i<10; i++)
tbuf[i]=ET1read32(1);
ET1drop(1);
```

This program will record events for about 100 milliseconds and then read the first 10 times into the PC array, *tbuf*. **ET1drop** is then called to clear any remaining event times.

ET1report(din)

Op Code ET1_REPORT {0x20}

Prototype int ET1report(int *din*);

Description Returns the number of timer/counter values stored in the ET1's FIFO.

Arguments *din* Device index number.

Example

```
ET1go(1);
do{}while(ET1report(1)<100);
```

```
ET1stop(1);
```

This code segment will program the ET1 to record events until 100 values are loaded into the ET1's FIFO.

ET1active(din)

Op Code ET1_ACTIVE {0x17}

Prototype int ET1active(int *din*);

Description Returns 1 if the specified ET1 is recording events, and returns 0 if the ET1 is IDLE.

Arguments *din* Device index number.

Example See example under **ET1blocks**.

ET1compare(din)

Op Code ET1_COMPARE {0x13}

Prototype void ET1compare(int *din*);

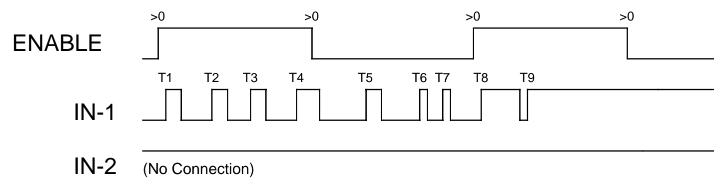
Description Places the specified ET1 in the COMPARISON mode. In this mode the enable input line is used to delimit two recording intervals: one when the enable line is high and a second when the enable line is low. On each transition of the enable input, the timer/counter is reset and a block marker (0) is written to the FIFO memory. See the example below for more information.

Arguments *din* Device index number.

Example

```
ET1compare(1);
ET1go(1);
delay(100);
ET1stop(1);
```

Hypothetical ET1 input:



If this program is run with the hypothetical input shown the resulting FIFO contents would be:
0, T1, T2, T3, T4, 0, T5, T6, T7, 0, T8, T9, 0 ....
NOTE: In the above example, the IN-2 input was left floating (high), so *all* event times on IN-1 were recorded in the FIFO. If IN-2 is used as an **acceptance pulse** for recording an event time, it must go high before the next event occurs on IN-1.

ET1evcount(din)

Op Code ET1_EVCOUNT {0x14}

Prototype void ET1evcount(int *din*);

Description Places the ET1 in Event Counter mode. In this mode the ET1 will count events rather than time stamp them. Use **ET1clear** to return the ET1 to Event Timing mode.

Arguments *din* Device index number.

Example If the line, `ET1evcount(1)`, is added to the beginning of the program example given for **ET1compare,** the FIFO contents would change to: 0, 4, 3, 2 ....

ET1blocks(din, numblocks)

Op Code ET1_BLOCKS {0x19}

Prototype void ET1blocks(int *din,* int *numblocks*);

Description Enables and programs the ET1's block counting feature. When enabled, this feature will cause the ET1 to time/count events until a specified number of blocks have been recorded to the FIFO memory. Set *numblocks* to 0 to disable the block counting feature.

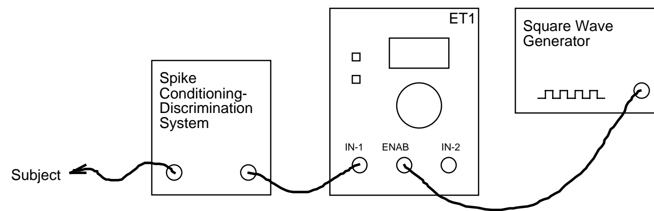
Arguments

din	Device index number.
numblocks	Number of blocks to record (0..32000).

Example

```
ET1clear(1);
ET1compare(1);
ET1blocks(1,100);
ET1go(1);
do{}while(ET1active(1));
/* Read in ET1 values */
```

Running this program with the ET1 connected as shown below will cause the system to record 100 blocks of data into the ET1's FIFO memory. Note that the COMPARISON mode has been selected so spontaneous activity is recorded in every other data block.



ET1mult(din)

Op Code ET1_MULT {0x12}

Prototype void ET1mult(int *din*);

Description When the optional EE1 Event Timer Expander is being used, **ET1mult** places the ET1 in Multi-Channel mode. In this mode, an 8-bit pattern indicating which of the EE1's inputs were triggered is stored with each event time as the upper byte of the 32-bit time stamp (i.e., the event times/counts are truncated to 24 bits). Bits corresponding to triggered inputs are set to 0 (other bits are 1).

NOTE: You must read event times with **ET1read32** into an *unsigned* long integer variable when using multi-channel mode or the bit pattern information will be lost.

Arguments *din* Device index number.

Example `ET1mult(1)`

Places ET1_(1) in Multi-Channel mode.

ET1xlogic(din, lmask)

Op Code ET1_MULT {0x1A}

Prototype void ET1xlogic(int *din*, int *lmask*);

Description When the optional EE1 Event Timer Expander is being used, **ET1xlogic** selects the active triggering edge (positive or negative) for each of the eight inputs.

Arguments *din* Device index number.

lmask Logic mask for inputs:

LMASK

	(msb)							(lsb)
BIT #:	7	6	5	4	3	2	1	0
INPUT #:	8	7	6	5	4	3	2	1

BIT=0 => pos edge
BIT=1 => neg edge

Example

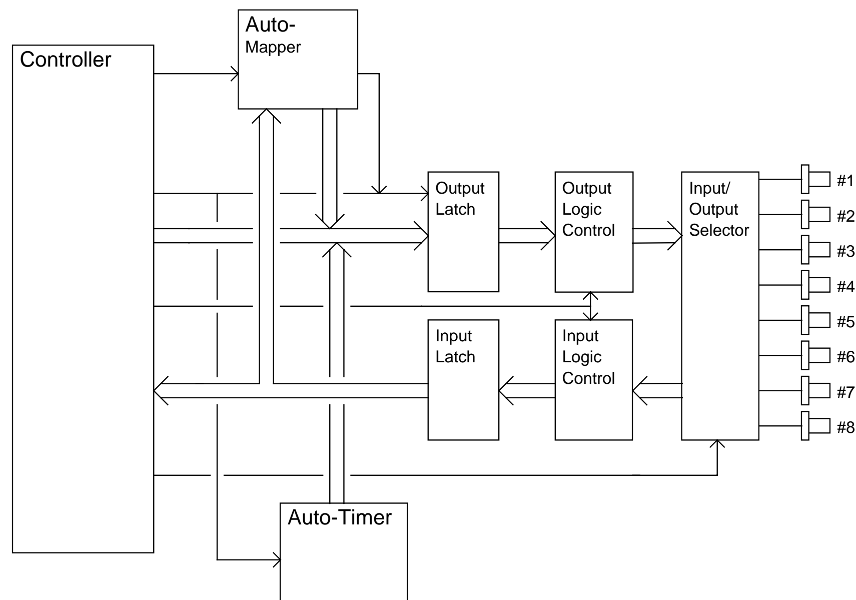
```
ET1mult(1)
ET1xlogic(1,0x0f)
```

Places ET1_(1) in Multi-Channel mode (using the EE1) and selects negative-edge triggering on inputs 1-4 and positive-edge triggering on inputs 5-8.

PI1 Parallel Interface

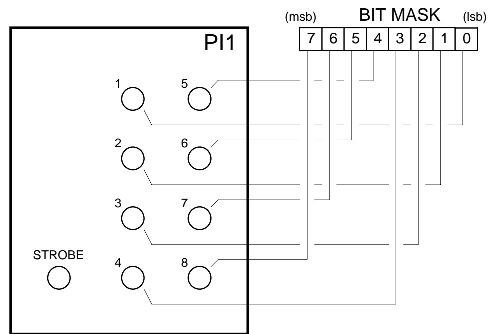
Overview

The PI1 is a smart parallel interface utilizing a dedicated microprocessor to drive eight "smart" TTL I/O lines. A logical diagram for the structures actually implemented in software is shown in the following diagram.



PI1 Logical Diagram.

The PI1 programming procedures make excessive use of mask-type arguments, whereby each bit in an 8-bit number is used to enable or disable a given feature on a specific I/O bit. The following diagram illustrates the correlation between mask bits and the PI1's front panel BNCs.



(msb)
BIT MASK
(lsb)
7
6
5
4
3
2
1
0
PI1
1
5
2
6
3
7
STROBE
4
8

PI1clear(din)

Op Code PI1_CLEAR {0x11}

Prototype void PI1clear(int *din*);

Description Resets the specified PI1 to the factory default configuration. The PI1 default configuration is as follows:

- All inputs with Positive Logic.
- No Mapping or Auto-Timing.
- No input Debouncing or Latching.
- Internal Strobe.

Arguments *din* Device index number.

Example

```
PI1clear(1);
```

Resets PI1_(1).

PI1outs(din, omask)

Op Code PI1_OUTS {0x12}

Prototype void PI1outs(int *din,* int *omask*);

Description Specifies certain I/O bits to be outputs. A set bit within *omask* will make the corresponding I/O line an output while all reset bits will indicate input lines.

Arguments *din* Device index number.
omask Designation Mask (1=Output, 0=Input).

Example

```
PI1outs(1,0xf0);
```

This call will program PI1_(1) to make I/O lines 1 through 4 inputs and lines 5 through 8 outputs.

PI1logic(din, logout, login)

Op Code PI1_LOGIC {0x13}

Prototype void PI1logic(int *din,* int *logout,* int *login*);

Description Specifies a logic mask applied to PI1 inputs and outputs. All operations within the PI1 are assumed to be operating with positive logic (i.e., 1 => high => true => on; 0 => low => false => off). The logic of any I/O line can be inverted by calling **PI1logic** with a 1 in the proper bit mask position.

NOTE: Bits of *logout,* which correspond to I/O lines programmed as inputs, will have no significance. The same is true of *login*.

Arguments

din Device index number.

logout Ouput Logic Designation Mask (0=Positive, 1=Negative).

login Input Logic Designation Mask (0 = Positive, 1=Negative).

Example Suppose the PI1 is interfaced to a subject response switch that pulls I/O line #1 low ('false' in positive logic) each time the button is pressed. The PI1 can be programmed to reverse the logic of this line with the following procedure call:

```
PI1logic( 1, 0x00, 0x01);
```

So, pressing the button will be interpreted as 'true'.

PI1write(din, bitcode)

Op Code PI1_WRITE {0x14}

Prototype void P1write(int *din,* int *bitcode*);

Description Writes the specified bit code to PI1 output lines. Note that any outputs specified to have negative logic will have output levels inverted from the *bitcode* specified for that line. Also, bits corresponding to I/O lines programmed as inputs have no significance.

Arguments *din* Device index number.
bitcode Output bit code.

Example

```
PI1clear(1);
PI1outs(1, 0xff);
PI1logic(1, 0x0f, 0x00);
PI1write(1, 0xff);
```

After the above program is run, LEDs #1 through #4 will be off while LEDs #5 through #8 will be on.

PI1read(din)

Op Code PI1_READ {0x15}

Prototype int PI1read(int *din*);

Description Reads the current inputs lines. Note that any inputs designated to have negative logic will read as 1 when the input is low and 0 when the input is high.

Arguments *din* Device index number.

Example

```
PI1clear(1);
PI1logic(1, 0x00, 0xfe);
i = PI1read(1);
```

If input #1 is held low while inputs #2 through #7 are allowed to float high, the **PI1read** function will return 0x00.

PI1autotime(din, atmask, dur)

Op Code PI1_AUTOTIME {0x16}

Prototype void PI1autotime(int *din,* int *atmask,* float *dur*);

Description Enables and programs the PI1's Auto-Time feature. When this feature is enabled the PI1 will automatically reset specified output lines after a preset period of time. All I/O lines share the same timer, which is started each time the outputs are written to by the host computer or the Auto-Mapping feature. Specify a *dur*ation of 0 to disable the Auto-Time feature.

Arguments

din	Device index number.
dur	Duration in milliseconds (0.0...3200.0).

Example See example under **PI1map**.

PI1map(din, incode, mapoutmask)

Op Code PI1map {0x17}

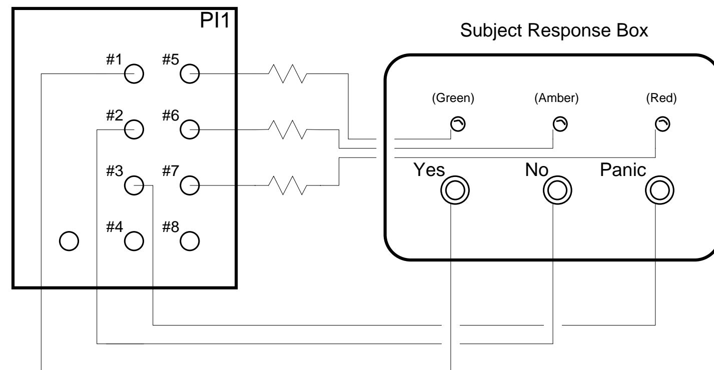
Prototype void PI1map(int *din,* int *incode,* int *mapoutmask*);

Description Programs the PI1's Auto-Mapping feature. When enabled the PI1 will automatically set specified output bits when a specified input pattern is present at the device's inputs. Note that this feature only sets output bits. However, the actual level at the device output can be inverted using the **PI1logic** call. The *incode* argument specifies the input pattern that must be matched exactly. If a match is detected the PI1 will automatically set the bits specified in *mapoutmask*.

Arguments

din	Device index number.
incode	Input match value.
mapoutmask	Output Control Mask.

Example Suppose the PI1 is going to be used to interface a user response box to the host computer through XBUS. The box will be connected to the PI1 as shown in the following diagram.



Typical subject response box.

The typical subject response box shown here has three momentary, normally open, push-button switches and three LEDs. The switches are connected with one terminal to ground and the other to the PI1 I/O line. LEDs are wired with the cathode (-) lead going to ground and the anode (+) lead going through a 470Ω resistor to the PI1 I/O line shown.

Let's suppose we want the response box to operate in the following way:

- Whenever a single button is pressed the corresponding LED should light up.
- The LED should remain illuminated while the button is pressed and for 100 milliseconds after it is released.
- If more than one button is pressed at the same time all LEDs should light up to indicate an error.
- Button activity should be latched to avoid missed subject responses.

- PI1 debouncing should be used to avoid false button reads.

The following program shows how the PI1 should be programmed to realize the system behavior described above. In this example the name of the button pressed is displayed on the screen for as long as the button is pressed.

```
#include <stdlib.h>
#include <conio.h>
#include "xbdrv.h"

#define YES   0x01
#define NO    0x02
#define PANIC 0x04

void main()
{
  int i;

  clrscr();
  XB1init(USE_DOS);

  printf("PI1 Subject Response Box
              Demonstration...");
  printf("\n\n    (Press any key to quit)");

  /* Clear to factory default                */
  PI1clear(1);

  /* Make lines #4 thru #8 outputs. Note unused lines are
     made outputs to simplify input ecoding. */
  PI1outs(1,0xf8);

  /* Flip the logic for the buttons inputs   */
  PI1logic(1,0x00,0x07);

  /* Latch all of the input lines            */
  PI1latch(1,0x07);

  /* Enable input debouncing                 */
  PI1debounce(1);

  /* Autotime all output lines               */
  PI1autotime(1,0x70,500.0);

  /* Send logic table for LED activity       */
  PI1map(1,0x01,0x10);
  PI1map(1,0x02,0x20);
  PI1map(1,0x03,0x70);
  PI1map(1,0x04,0x40);
  PI1map(1,0x05,0x70);
  PI1map(1,0x06,0x70);
  PI1map(1,0x07,0x70);

  do
  {
    i=PI1read(1);
    if(i)
    {
      switch(i)
      {
```

```
        case YES:
        {
          gotoxy(10,10);
          printf("[ Yes ]     ");
          break;
        }
        case NO:
        {
          gotoxy(10,10);
          printf("[ NO ]      ");
          break;
        }
        case PANIC:
        {
          gotoxy(10,10);
          printf("[ PANIC ]      ");
          break;
        }
        default:
        {
          gotoxy(10,10);
          printf("! Error !     ");
        }
      }
    }
    else
    {
      gotoxy(10,10);
      printf("                ");
    }
    delay(1000);            /* Dummy delay */
  }while(!kbhit());
  (void)getch();
  }
```

PI1latch(din, lmask)

Op Code PI1_LATCH {0x18}

Prototype void PI1latch(int *din,* int *lmask*);

Description Programs the PI1's Input Latching feature. This feature will latch selected (*lmask*) input bits, which become true (after optional logic inversion) until they are acknowledged by the host computer. The latch is automatically cleared when the PI1 inputs are read from the host, and then set again as soon as any input line in *lmask* is detected true on the next strobe cycle.

Arguments *din* Device index number.

lmask Selection mask (1 = Latch, 0 = No Latch).

Example See example under **PI1map**.

PI1debounce(din)

Op Code PI1_DEBOUNCE {0x19}

Prototype void PI1debounce(int *din*);

Description Enables PI1's Debouncing feature on all input lines.

Arguments *din* Device index number.

Example See example under **PI1map**.

PI1strobe(din)

Op Code PI1_STROBE {0x1A}

Prototype void PI1strobe(int *din*);

Description Enables the PI1's External Strobe input. By default the PI1 uses an internal 10KHz timer to sample the PI1 inputs and update the PI1 outputs. This internal rate generator can be replaced by an external TTL signal with a frequency of up to 10KHz.

Arguments *din* Device index number.

Example `PI1strobe(1);`

This enables PI1_(1) External Strobe input.

PI1optread(din)

Op Code PI1_OPTREAD {0x1B}

Prototype int PI1optread(int *din*);

Description Reads the PE1's optically isolated input port (if installed).

Arguments *din* Device index number.

Example `i = PI1optread(1).`

This call reads the state of PI1/PE1_(1)'s optically isolated input.

PI1optwrite(din, bitcode)

Op Code PI1_OPTWRITE {0x1C}

Prototype void PI1optwrite(int *din,* int *bitcodel*);

Description Writes the value *bitcode* to the PE1's optically isolated output port (if installed).

Arguments

din Device index number.

bitcode 8 bit value (0..255)

Example `PI1optwrite(1,0xff).`

This call sets all bits on PI1/PE1_(1)'s optically isolated output.

WG1 Waveform Generator

WG1on(din)

Op Code WG1_ON {0x11}

Prototype void WG1on(int *din*);

Description Tells the specified WG1 to begin producing the programmed wave shape. When using the external trigger/enable, **WG1on** acts as an overall device enable (use **WG1off** to disable the WG1). When the trigger mode is set to NONE, **WG1on** will activate the WG1 output signal.

Arguments *din* Device index number.

Example

```
WG1trig(1,NONE);
WG1on(1);
```

This will set no trigger (free run) and turn on WG1_(1) output.

WG1off(din)

Op Code WG1_OFF {0x12}

Prototype void WG1off(int *din*);

Description Places the specified WG1 in IDLE mode. **WG1off** will act as an overall system disabler when the external trigger/enable is used. When triggering is set to NONE, **WG1off** will turn off the device's output signal.

Arguments *din* Device index number.

Example

```
WG1off(1);
```

This will turn off WG1_(1) output.

WG1ton(din)

Op Code WG1_TON {0x21}

Prototype void WG1ton(int *din*);

Description Similar to **WG1on**, but causes the specified WG1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
WG1trig(1,NONE);
WG1ton(1);
WG1ton(2);
XB1gtrig();            /* Global trigger */
```

This will cause WG1_(1) and WG1_(2) to turn ON at precisely the same time.

WG1clear(din)

Op Code WG1_CLEAR {0x13}

Prototype void WG1clear(int *din*);

Description Clears the specified WG1 and resets it to the factory default setup (Gaussian noise, 9.99V amplitude, 0.0V DC shift, continuous phase, free run, 1.0 ms rise/fall time, output off).

Arguments *din* Device index number.

Example

```
WG1clear(1);
```

This clears WG1_(1) and sets factory defaults.

WG1amp(din, amp)

Op Code WG1_AMP {0x14}

Prototype void WG1amp(int *din,* float *amp*);

Description Sets the output waveform amplitude.

Note With WG1 hardware versions lower than 4.0, when the WG1 is generating Gaussian noise at its max level, the amplitude display will read 9.99 on the display. Decreasing the amplitude to 5.0 will not reduce the level of the noise by 6.0dB. This is corrected with WG1 hardware versions 4.0 and higher.

Arguments *din* Device index number.

amp Amplitude in volts (0.00...9.99).

Example

```
WG1amp(1, 5.00);
```

This call sets WG1_(1) output amplitude to 5.00 V_p.

WG1freq(din, freq)

Op Code WG1_FREQ {0x15}

Prototype void WG1freq(int *din,* float *freq*);

Description Sets the sine wave frequency.

Arguments *din* Device index number.

freq Frequency in Hz (1...20,000).

Example

```
WG1shape(1,SINE);
WG1freq(1, 5000);
WG1on(1);
```

This programs WG1_(1) to produce a sine wave output with a frequency of 5kHz.

WG1swrt(din, swrt)

Op Code WG1_SWRT {0x16}

Prototype void WG1swrt(int *din,* float *swrt*);

Description Sweeps the sine wave frequency for 'chirp' tones.

Arguments
din Device index number.
swrt Frequency sweep rate in Hz/s (10 ... 9,999).

Example

```
WG1shape(1,SINE);
WG1freq(1, 200);
WG1swrt(1, 1000);
WG1on(1);
```

This code programs WG1_(1) to produce a swept-frequency sine wave, starting at 200Hz. After one second, the frequency will be 1.2 kHz.

NOTE: The frequency will continue increasing until the WG1 is gated off, or until **WG1off** is issued.

WG1phase(din, phase)

Op Code WG1_PHASE {0x17}

Prototype void WG1phase(int *din,* float *phase*);

Description Sets the onset phase of the WG1's sine wave generator as well as the seed value for the WG1's internal random number generator. Using the phase call, the WG1 can be made to generate frozen noise signals each time the device is triggered. To generate non-frozen noise call WG1phase(1, -1).

Arguments
din Device index number.
phase Phase in degrees (0.00...360.00). Use (-1) to set the phase to continuous.

Example

```
WG1trig(1, POS_ENABLE);
WG1shape(1, SINE);
WG1freq(1, 200);
WG1phase(1, 90);
WG1on(1);
```

This programs WG1_(1) to produce a 200Hz sine wave with a 90 degree onset phase, synchronized with the Enable input.

WG1dc(din, dc)

Op Code WG1_DC {0x18}

Prototype void WG1dc(int *din*, float *dc*);

Description Sets the DC shift level in the output waveform. NOTE: This shift remains on the output even when the signal is gated off. DC shifting is useful when generating AM modulated signals.

Arguments
din Device index number.
dc DC shift level in volts (-9.9...9.9).

Example

```
WG1dc(1, -1.0);
```

This call programs WG1_(1) output to have a -1.0 V DC shift.

WG1shape(din, scode)

Op Code WG1_SHAPE {0x19}

Prototype void WG1shape(int *din*, int *scode*);

Description Selects the WG1 waveform shape. The valid shape types are listed below.

Arguments
din Device index number.
scode Shape type code (1..4).

GAUSS	Gaussian noise.
UNIFORM	Uniform noise.
SINE	Sine wave.
WAVE	User-defined shape.

Example

```
WG1amp(1,6.00);
WG1shape(1,GAUSS);
WG1on(1);
```

This will program WG1_(1) to produce a Gaussian noise signal with a +/- 6 volt maximum amplitude.

WG1dur(din, dur)

Op Code WG1_DUR {0x1A}

Prototype void WG1dur(int *din,* float *dur*);

Description Sets/enables the WG1's self-timing feature. When the self-timer is enabled the signal will gate on when a valid trigger is detected and gate off after the specified duration. Specify a duration of zero for continuous output.

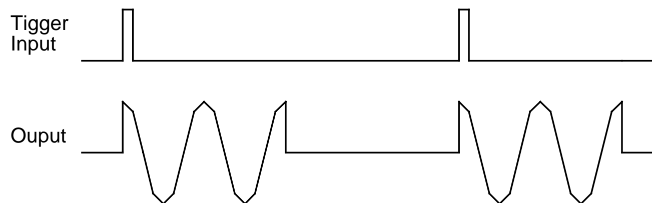
Arguments *din* Device index number.

dur Duration of gate in ms (1...30,000).

Example

```
WG1trig(1,POS_EDGE);
WG1rf(1,0.0);
WG1shape(1,SINE);
WG1freq(1, 1000);
WG1phase(1, 90.0);
WG1dur(1, 2);
WG1on(1);
```

This program will produce the output shown below:



WG1rf(din, rf)

Op Code WG1_RF {0x1B}

Prototype void WG1rf(int *din,* float *rf*);

Description Sets the rise and fall time of the built-in gating feature.

NOTE: The WG1 has limited gating abilities. Only the following gate rise/fall times will give guaranteed results:

0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
10., 20., 30., 40., 50., 60., 70., 80., 90.

Arguments
din Device index number.
rf Rise and fall time of gate in ms (0.1...90.0).

Example
```
WG1rf(1, 3.0);
```
This sets a 3.0 ms rise and fall time to WG1_(1).

WG1trig(din, tcode)

Op Code WG1_TRIG {0x1C}

Prototype void WG1trig(int *din*, int *tcode*);

Description Programs the ENABLE/TRIGGER input of the WG1. The valid trigger types are listed below.
NOTE: The duration feature works only with EDGE type (see **WG1dur)** and NONE (manual) triggering.

Arguments
din Device index number.
tcode Trigger type code (1..5).

POS_EDGE	Rising edge triggered.
NEG_EDGE	Falling edge triggered.
POS_ENABLE	High level enabled.
NEG_ENABLE	Low level enabled.
NONE	Free running.

Example
```
WG1trig(1,POS_EDGE);
```
This will program WG1_(1) to rising edge triggered.

WG1seed(din, seed)

Op Code WG1_SEED {0x1D}

Prototype void WG1seed(int *din,* long *seed*);

Description Sets the random number generator seed value.

Arguments *din* Device index number.
seed Random number seed value (1...2^32).

Example

```
WG1shape(1,UNIFORM);
WG1seed(1, 1000000);
WG1on(1);
```

This programs WG1_(1) to output uniform random noise with the generator seeded by 1,000,000.

WG1delta(din, delta)

Op Code WG1_DELTA {0x1E}

Prototype void WG1delta(int *din,* long *delta*);

Description Sets a very precise sine wave frequency.

Arguments *din* Device index number.
delta $= 2^{32} \cdot$ (frequency in Hz)/79277.419 Hz
$= 54176.426 \cdot$ (frequency in Hz)

Example

```
WG1shape(1,SINE);
WG1delta(1,(long)(54176.426 * 100.0));
WG1on(1);
```

This code programs WG1_(1) to produce a 100Hz sine wave.

WG1wave(din, wave, npts)

Op Code WG1_WAVE {0x1F}

Prototype void WG1wave(int *din,* int *wave[],* int *npts*);

Description Uploads a user-defined waveform into the WG1 sample buffer from a host PC.

Arguments *din* Device index number.
wave Pointer to waveform buffer.
npts number of buffer points to use (1...800).

Example

```
c=32767/400
for(i=-400;i<400;i++)
  wave[i]=i*c;
WG1wave(1,wave,800);
```

```
WG1shape(1,WAVE);
WG1on(1);
```

This uploads an 800 point, zero DC shift ramp function into the WG1_(1) and replays the contents of the sample buffer until gated off.

WG1status(din)

Op Code WG1_STATUS {0x20}

Prototype int WG1status(int *din*);

Description Returns the status of the specified WG1. WARNING: Calling **WG1status** while the device is generating tones or converting a user waveform will cause clicks in the WG1 output waveform. Do not use **WG1status** while the device is in one of these two modes.

Return Values:

OFF = 0 WG1 is off (no output).
ON = 1 WG1 output gated fully on.
RISING = 2 Output gate is in rising state.
FALLING = 3 Output gate is in falling state.

Arguments *din* Device index number.

Example

```
if(WG1status(1)==ON);
  WG1off(1);
```

Turns WG1(1) off if it is on.

(this page intentionally left blank)

PF1 Programmable Filter

Overview

The PF1 is a digital programmable filter. You can use the built-in filter functions or can load your own. Built-in filters functions include:

- **PF1freq**
- **PF1gain**

If you wish to load your own filter coefficients, use one of the following functions. Each function combines basic PF1 function calls to implement one of three filter architectures:

- **PF1fir16**
- **PF1fir32**
- **PF1iir32**
- **PF1biq16**
- **PF1biq32**

The remaining commands are the basic functions that download user-specified filter coefficients to the PF1. In general, you should use the preceding routines, since they accept floating-point coefficient arrays.

- **PF1type**
- **PF1begin**
- **PF1b16**
- **PF1a16**
- **PF1b32**
- **PF1a32**

Filter coefficients can be obtained from a variety of signal processing software packages. Refer to PF1 supplemental documentation for further information.

PF1freq(din, lpfreq, hpfreq)

Op Code PF1_FREQ {0x19}

Prototype void PF1freq(int *din*, int *lpfreq*, int *hpfreq*);

Description Selects corner frequencies of built-in lowpass and highpass filters and commences filtering.

Arguments
din Device index number.
lpfreq lowpass corner freq. (50...15,000), 0 OFF.
hpfreq highpass corner freq. (50...15,000), 0 OFF.

Example

```
PF1freq(1, 600, 1400);
```

This call tells the PF1_(1) to start filtering with a lowpass corner frequency of 600Hz and highpass corner frequency of 1.4kHz, which is equivalent to a notch-filter with 800Hz bandwidth, and centered at about 1kHz.

PF1gain(din, lpgain, hpgain)

Op Code PF1_GAIN {0x1A}

Prototype void PF1gain(int *din*, int *lpgain*, int *hpgain*);

Description Select gains of built-in lowpass and highpass filters.

Arguments
din Device index number.
lpgain lowpass filter gain.
hpgain highpass filter gain.

_0DB	1
_6DB	2
_12DB	3
_18DB	4
_24DB	5

Example

```
PF1gain(1, _0DB, _6DB);
```

This call will tell the PF1_(1) to set the lowpass filter gain to 0 dB and the highpass filter gain to 6 dB.

PF1bypass(din)

Op Code PF1_BYPASS {0x12}

Prototype void PF1bypass(int *din*);

Description Bypasses the main digital filter (oversampling circuit remains active so bandwidth is 20kHz).

Arguments *din* Device index number.

Example
```
PF1bypass(1);
```
This call lets the input signal bypass the digital filtering process on PF1_(1).

PF1nopass(din)

Op Code PF1_NOPASS {0x13}

Prototype void PF1nopass(int *din*);

Description Switches PF1 output off.

Arguments *din* Device index number.

Example
```
PF1nopass(1);
```
This call will turn off the PF1_(1) output.

➢ The following five functions simplify PF1 programming by combining basic PF1 calls to load and run a filter in a single call.

PF1fir16(din, bcoefs, ntaps)

Prototype void PF1fir16(int *din*, float *bcoefs[]*, int *ntaps*);

Description Downloads a *floating-point* array, containing the desired FIR tap coefficients, and starts the filtering process. Floating-point to integer conversion is done by the routine. Precision is 16-bit and coefficients must lie within the 1.15 format range: $-1.0 \leq b_i < +1.0$

Arguments
din Device index number.
bcoefs Array containing feed-forward coefficients: $[b_0, b_1, b_2,$ ☺ $, b_N]$ N = filter order.
ntaps Number of delay line taps (= N+1).

Example

```
PF1fir16(1, bcoefs, 64);
```

This downloads a 64-tap FIR to PF1_(1) and starts the filtering process.

PF1fir32(din, bcoefs, ntaps)

Prototype void PF1fir32(int *din*, float *bcoefs[]*, int *ntaps*);

Description Downloads a *floating-point* array, containing the desired FIR tap coefficients, and starts the filtering process. Floating-point to integer conversion is done by the routine. Precision is 32-bit and coefficients must lie within the 2.30 format range: $-2.0 \leq b_i < +2.0$

Arguments
din Device index number.
bcoefs Array containing feed-forward coefficients: $[b_0, b_1, b_2,$ ☺ $, b_N]$ N = filter order.
ntaps Number of delay line taps (= N+1).

Example
```
PF1fir32(1, bcoefs, 64);
```
This downloads a 64-tap FIR to PF1_(1) and starts the filtering process.

PF1iir32(din, bcoefs, acoefs, ntaps)

Prototype void PF1iir32(int *din*, float *bcoefs[]*, float *acoefs[]*, int *ntaps*);

Description Downloads two *floating-point* arrays, containing feedforward and feedback tap coefficients for a Direct-II IIR filter, and starts the filtering process all in a single call. Floating-point to integer conversion is done by the routine. Precision is 32-bit and coefficients must lie within the 8.24 format range: $-128.0 \leq b_i, a_i < +128.0$

WARNING: Direct-II IIR's are inherently limited to approximately 8-th order (depending on filter type).

Arguments

din Device index number.

bcoefs Array of feed-forward (numerator) coeff's: $[b_0, b_1, b_2,$ ☺ $, b_N]$ N = filter order.

acoefs Array of feed-back (denominator) coeff's: $[a_0, a_1, a_2,$ ☺ $, a_N]$ N = filter order. Note that a_0 is always sent, but not actually used.

ntaps Number of delay line taps (= N+1).

Example
```
PF1iir32(1, bcoefs, acoefs, 5);
```
This downloads a 4-th order IIR to PF1_(1) and starts the filtering process. Each array contains 5 coefficients.

PF1biq16(din, bcoefs, acoefs, nbiqs)

Prototype void PF1biq16(int *din*, float *bcoefs[]*, float *acoefs[]*, int *nbiqs*);

Description Downloads two *floating-point* arrays, containing feedforward and feedback tap coefficients for a Cascaded Bi-quad IIR filter, and starts the filtering process all in a single call. Floating-point to integer conversion is done by the routine. Precision is 16-bit. The coefficients in the two arrays must be ordered appropriately for the bi-quad architecture and lie within the 2.14 format range: $-2.0 \le b_i, a_i < +2.0$

Arguments

din Device index number.

bcoefs Array of feed-forward (numerator) coeff's: $[b_{10}, b_{11}, b_{12}, b_{20}, b_{21}, b_{22},$ ☺ $b_{m0}, b_{m1}, b_{m2}]$

acoefs Array of feed-back (denominator) coeff's: $[a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22},$ ☺ $a_{m0}, a_{m1}, a_{m2}]$, N = 2m = filter order. Note that the a_{i0}'s are always sent, but not actually used.

nbiqs Number of 2nd-order (3 tap) bi-quad stages (= m).

Example

```
PF1biq16(1, bcoefs, acoefs, 4);
```

This will download an 8-th order Bi-quad IIR to PF1_(1) and begin the filtering process. Each array contains 3 x 4 = 12 coefficients.

PF1biq32(din, bcoefs, acoefs, nbiqs)

Prototype void PF1biq32(int *din*, float *bcoefs[]*, float *acoefs[]*, int *nbiqs*);

Description Downloads two *floating-point* arrays, containing feedforward and feedback tap coefficients for a Cascaded Bi-quad IIR filter, and starts the filtering process all in a single call. Floating-point to integer conversion is done by the routine. Precision is 32-bit. The coefficients in the two arrays must be ordered appropriately for the bi-quad architecture and lie within the 2.30 format range: $-2.0 \le b_i, a_i < +2.0$

Arguments

din Device index number.

bcoefs Array of feed-forward (numerator) coeff's: $[b_{10}, b_{11}, b_{12}, b_{20}, b_{21}, b_{22}, \ldots b_{m0}, b_{m1}, b_{m2}]$

acoefs Array of feed-back (denominator) coeff's: $[a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}, \ldots a_{m0}, a_{m1}, a_{m2}]$, N = 2m = filter order. Note that the a_{i0}'s are always sent, but not actually used.

nbiqs Number of 2nd-order (3 tap) bi-quad stages (= m).

Example

```
PF1biq32(1, bcoefs, acoefs, 4);
```

This will download an 8-th order Bi-quad IIR to PF1_(1) and begin the filtering process. Each array contains 3 x 4 = 12 coefficients.

- The remaining functions are the basic calls used by the filter functions to upload coefficients to the PF1. In general, use the preceeding functions, since they accept floating-point coefficient arrays.

PF1type(din, type, n)

Op Code PF1_TYPE {0x11}

Prototype void PF1type(int *din*, int *type*, int *n*);

Description Selects filter type/algorithm and commences filtering.

Arguments

din Device index number.

n Number of delay line taps,
Number of biquad stages for BIQ16/32.

type Filter type code (16/32 16- or 32-bit prec.):

Code:		Type:	Coeff. Fmt:	Max. Taps/Order:
FIR16	1	FIR	1.15	160/159
FIR32	2	FIR	2.30	64/63
IIR32	3	IIR	8.24	8/7
BIQ16	4	bi-quad IIR	2.14	57/38
BIQ32	5	bi-quad IIR	2.30	21/14

Example

```
PF1begin(1);
{load filter coefficients}
PF1type(1, FIR16, 32);
```

This will tell the PF1_(1) to start filtering using a 16-bit precision FIR algorithm. The filter's 32 tap coefficients must be loaded first.

PF1begin(din)

Op Code PF1_BEGIN {0x14}

Prototype void PF1begin(int *din*);

Description Sets mode for downloading tap coefficients. Current filtering process continues until **PF1type** is called.

Arguments *din* Device index number.

Example See **PF1type**.

PF1b16(din, coeff)

Op Code PF1_b16 {0x15}

Prototype void PF1b16(int *din*, int *coeff*);

Description Downloads a 16-bit, feed-forward (numerator) tap coefficient.

NOTE: Floating-point to integer scale value depends on coefficient format required by desired filter type—see **PF1type** code constants.

Arguments *din* Device index number.

coeff Filter coefficient.

Example

```
PF1begin(din); /* set PF1 to load mode */
for(i=0; i<ntaps; i++)
{
  PF1b16(din, (int)(bcoes[i]*0x8000+0.5));
}
PF1type(din, FIR16, ntaps); /* start filter */
```

This example shows how coefficients are scaled and downloaded to the PF1 for a 16-bit precision FIR filter. Values in `bcoes[]` are floating-point between ± 1, as required for the 1.15 format for this filter type.

PF1a16(din, coeff)

Op Code PF1_a16 {0x17}

Prototype void PF1a16(int *din*, int *coeff*);

Description Downloads a 16-bit, feed-back (denominator) tap coefficient.

NOTE: Floating-point to integer scale value depends on coefficient format required by desired filter type—see **PF1type** code constants.

Arguments *din* Device index number.

coeff Filter coefficient.

Example See **PF1b16**.

PF1b32(din, coeff)

Op Code PF1_b32 {0x16}

Prototype void PF1b32(int *din*, int *coeff*);

Description Downloads a 32-bit, feed-forward (numerator) tap coefficient.

NOTE: Floating-point to integer scale value depends on coefficient format required by desired filter type—see **PF1type** code constants.

Arguments *din* Device index number.

coeff Filter coefficient.

Example See **PF1b16**.

PF1a32(din, coeff)

Op Code PF1_a32 {0x18}

Prototype void PF1a32(int *din*, int *coeff*);

Description Downloads a 32-bit, feed-back (denominator) tap coefficient. NOTE: Floating-point to integer scale value depends on coefficient format required by desired filter type—see **PF1type** code constants.

Arguments *din* Device index number.

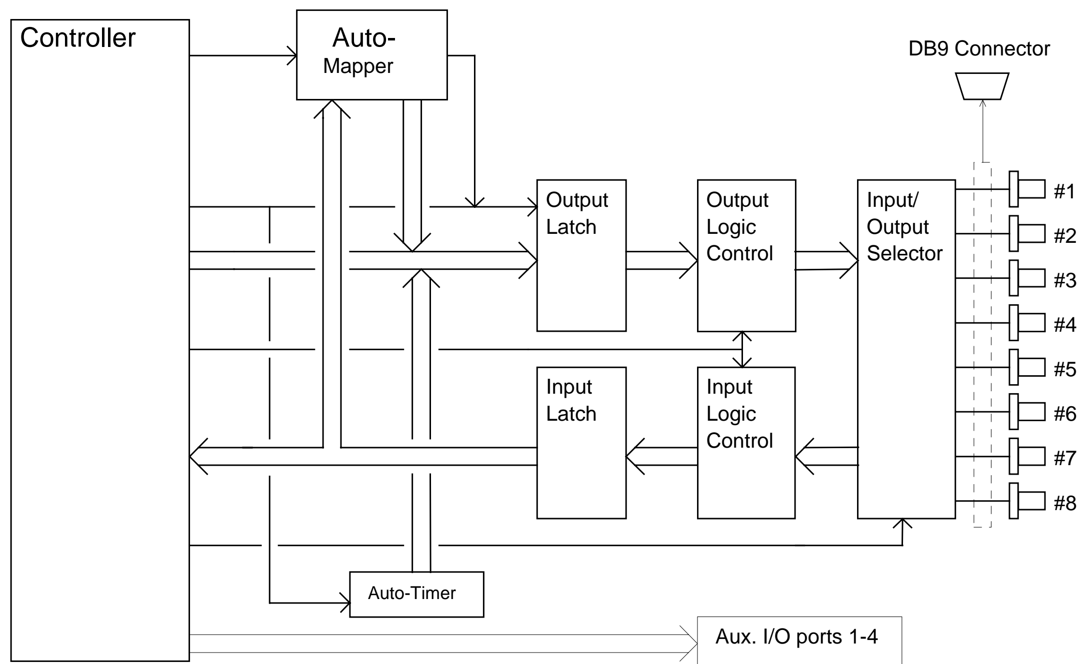
coeff Filter coefficient.

Example See **PF1b16**.

PI2 Parallel Interface

Overview

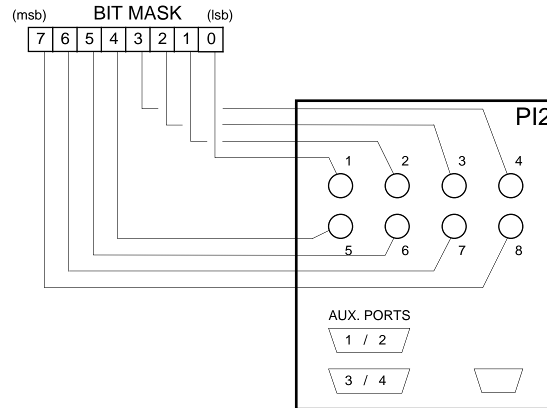
The PI2 is a smart parallel interface utilizing a dedicated microprocessor to drive eight "smart" TTL I/O lines and four auxiliary 8-bit ports for general purpose digital I/O. A logical diagram for the structures actually implemented in software is shown in the following diagram.



PI2 Logical Diagram.

The PI2 programming procedures make extensive use of mask-type arguments, whereby each bit in an 8-bit number is used to enable or disable a given feature on a specific I/O bit. The following diagram illustrates the relation between mask bits and the PI2's front panel BNCs.

The four Auxiliary I/O Ports are accessed using the last three command calls in this section. All other commands apply only to the eight main "smart" I/O lines.



➢ The four Auxiliary I/O Ports are accessed using the following three command calls:

- **PI2outsX**
- **PI2writeX**
- **PI2readX**

All other commands apply only to the eight main "smart" I/O lines.

PI2clear(din)

Op Code PI2_CLEAR {0x11}

Prototype void PI2clear(int *din*);

Description Resets the specified PI2 to the factory default configuration. The PI2 default configuration is as follows:

- Auxilliary ports: All inputs
- Smart Ports:
  a) All inputs with Positive Logic.
  b) No Mapping or Auto-Timing.
  c) No input Debouncing or Latching.

Arguments *din* Device index number.

Example

```
PI2clear(1);
```

Resets PI2_(1).

PI2outs(din, omask)

Op Code PI2_OUTS {0x12}

Prototype void PI2outs(int *din*, int *omask*);

Description Specifies certain I/O bits to be outputs. A set bit within *omask* will make the corresponding I/O line an output while all reset bits will indicate input lines.

Arguments *din* Device index number.
omask Designation Mask (1=Output, 0=Input).

Example

```
PI2outs(1,0xf0);
```

This call will program PI2_(1) to make I/O lines 1 through 4 inputs and lines 5 through 8 outputs.

PI2logic(din, logout, login)

Op Code PI2_LOGIC {0x13}

Prototype void PI2logic(int *din,* int *logout,* int *login*);

Description Specifies a logic mask applied to PI2 inputs and outputs. All operations within the PI2 are assumed to be operating with positive logic (i.e., 1 => high => true => on; 0 => low => false => off). The logic of any I/O line can be inverted by calling PI2logic with a 1 in the proper bit mask position.

NOTE: Bits of *logout,* which correspond to I/O lines programmed as inputs, will have no significance. The same is true of *login*.

Arguments

din Device index number.

logout Ouput Logic Designation Mask (0=Positive, 1=Negative).

login Input Logic Designation Mask (0 = Positive, 1=Negative).

Example Suppose the PI2 is interfaced to a subject response switch that pulls I/O line #1 low ('false' in positive logic) each time the button is pressed. The PI2 can be programmed to reverse the logic of this line with the following procedure call:

```
PI2logic( 1, 0x00, 0x01);
```

So, pressing the button will be interpreted as 'true'.

PI2debounce(din, dbtime)

Op Code PI2_DEBOUNCE {0x16}

Prototype void PI2debounce(int *din*, int *dbtime*);

Description Enables PI2's Debouncing feature on all input lines.

Arguments

din Device index number.

dbtime Debounce time in ms (0...255).

Example See example under **PI2map**.

PI2latch(din, lmask)

Op Code PI2_LATCH {0x18}

Prototype void PI2latch(int *din,* int *lmask*);

Description Programs the PI2's Input Latching feature. This feature will latch selected (*lmask*) input bits which become true (after optional logic inversion), until they are acknowledged by the host computer. The latch is automatically cleared when the PI2 inputs are read from the host, and then set again as soon as any input line in *lmask* is detected true on the next strobe cycle.

Arguments
din Device index number.
lmask Input bits to latch (1 = Latch, 0 = No Latch).

Example See example under **PI2map**.

PI2autotime(din, bitn, dur)

Op Code PI2_AUTOTIME {0x17}

Prototype void PI2autotime(int *din,* int *bitn,* float *dur*);

Description Enables and programs the PI2's Auto-Time feature. When this feature is enabled the PI2 will automatically reset a specified output line after a period of time. Each output line has its own timer which is started each time it is written to by the host computer or the Auto-Mapping feature.

Arguments
din Device index number.
bitn Bit number (0...7)
dur Duration in ms (0...32,000).

Example See example under **PI2map**.

PI2toggle(din, tmask)

Op Code PI2_LATCH {0x21}

Prototype void PI2toggle(int *din,* int *tmask*);

Description Programs the PI2's Output Toggling feature, used with **PI2autotime**. This feature will cause selected (*tmask*) output bits to toggle back on (set) after the same *dur*ation specified in **PI2autotime**.

Arguments
din Device index number.
tmask Output bits to toggle.

Example

```
PI2clear(1);
PI2outs(1, 0x0f);
PI2autotime(1, 0, 500);
PI2autotime(1, 1, 250);
PI2toggle(1, 0x03);
```

After the above program is run, LEDs #1 and #2 (bits 0 and 1) will go on and off at 1Hz and 0.5Hz, respectively.

PI2map(din, bitn, mmask)

Op Code PI2map {0x1D}

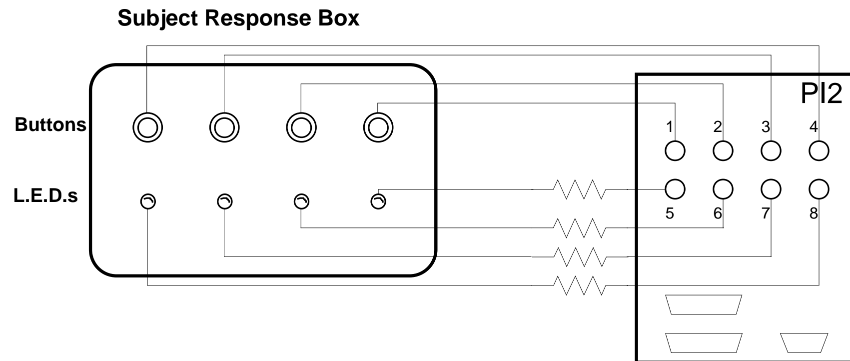
Prototype void PI2map(int *din,* int *bitn,* int *mmask*);

Description Programs the PI2's Auto-Mapping feature to set specific output bits when a specific input pattern is present at the device's inputs. Note that this feature only "sets" output bits—the actual level at the device output can be inverted using the **PI2logic** call. The *mmask* argument specifies the input bits that must be "true" for a valid match (other bits are ignored). If a match is detected, the PI2 will set the *bitn* specified.

Arguments
din Device index number.
bitn Bit number of output to set.
mmask Input map mask.

Example

Suppose the PI2 is going to be used to interface a user response box to the host computer through the XBUS. The box will be connected to the PI2 as shown in the following diagram.



Typical Subject Response Box

The subject response box shown above has four momentary, normally open, push-button switches and four LEDs. The switches are connected with one terminal to ground and the other to the PI2 I/O line shown above. LEDs are wired with the cathode (-) lead going to ground and the anode (+) lead going through a 470Ω resistor to the PI2 I/O line shown.

Let's suppose we want the response box to operate in the following way:

- Whenever a single button is pressed the corresponding LED should light up.
- The LED should remain illuminated while the button is pressed and for 100 milliseconds after it is released.
- If more than one button is pressed at the same time, the software should indicate an error.

- Button activity should be latched to avoid missed subject responses.

- PI2 debouncing should be used to avoid false button reads.

- Response times should be measured.

The following program shows how the PI2 should be programmed to realize the system behavior described above. In this example the name of the button pressed is displayed on the screen as long as the button is pressed.

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "xbdrv.h"

#define YES   0x01
#define NO    0x02
#define PANIC 0x04

void main()
{
  int i,j,t1;

  clrscr();
  XB1init(USE_DOS);

  printf("PI2 Subject Response Box Demonstration...");
  printf("\n\n    (Press any key to quit)");

  PI2clear(1);           /* Clear to factory default      */
  PI2outs(1,0xf8);       /* Make lines 4-8 outputs
  */
  PI2logic(1,0x00,0x07); /* Neg. logic for button inputs  */
  PI2latch(1,0x07);      /* Latch all of the input lines  */
  PI2debounce(1,10);     /* Enable input debouncing (10ms)*/

  /* Autotime output lines 5-7 (bits 4-6)   */
  for(i=4;i<7;i++)
    PI2autotime(1,i,100);

  /* Now send maps for LED activity:*/
  PI2map(1,4,YES);   /* bit 4 (output 5) on for YES   */
  PI2map(1,5,NO);    /* bit 5 (output 6) on for NO    */
  PI2map(1,6,PANIC); /* bit 6 (output 7) on for PANIC */

  PI2zerotime(1,0x07);   /* zero timers on all input lines*/

  do
  {
    i=PI2read(1);    /* read inputs                */
    if(i)            /* button pressed             */
    {
```

```
      switch(i)      /* process response           */
      {
        case YES:
       {
             t1=PI2gettime(1,0);
             gotoxy(10,10);
             printf("[ Yes ]   response time t=%d ms   ",t1);
          break;
        }
        case NO:
       {
             t1=PI2gettime(1,1);
             gotoxy(10,10);
             printf("[ NO ]   response time =%d ms   ",t1);
          break;
        }
        case PANIC:
       {
             t1=PI2gettime(1,2);
             gotoxy(10,10);
             printf("[ PANIC ]   response time =%d ms   ",t1);
          break;
        }
        default:
        {
          gotoxy(10,10);
             printf("! Error !                   ");
        }
      } /* end switch */
    } /* end if */

    delay(1000);  /* Delay before accept next response */
    gotoxy(10,10);
    printf("Ready                                    ");
    /* read to clear spurious inputs latched during processing: */
    if(i)    PI2read(1);
    /* zero the time of the inputs just pressed: */
    PI2zerotime(1,i);

  }while(!kbhit());
  (void)getch();
}
```

Note that the input timers are reset individually after a button is pressed. This allows each response time to be measured from the last response of that type.

PI2zerotime(din, bitmask)

Op Code PI2_ZEROTIME {0x1A}

Prototype void PI2zerotime(int *din,* int *bitmask*);

Description Zeroes the PI2's Input Timing feature. This feature is used with **PI2gettime** to determine the time any input line goes "true". Each line has a timer, so times can be zeroed independently.

Arguments

din	Device index number.
bitmask	Bit mask of input timers to zero.

Example See example under **PI2gettime**.

PI2gettime(din, bitn)

Op Code PI2_GETTIME {0x1A}

Prototype int PI2gettime(int *din,* int *bitn*);

Description Returns the time (in ms) an input line became "true" from the time PI2zerotime was last called. Each line has a timer, so times can be read individually. Input event times are latched so they can be read any time before another zeroing. Maximum time is 32 seconds. WARNING: The time for a particular input will read zero for 2 milliseconds after that input is read true using **PI2read**. To avoid possible false readings always wait at least 2 milliseconds after a bit reads true before trying to read its time.

Arguments

din	Device index number.
bitn	Bit number of input to read.

Example

```
PI2clear(1);
getch();                /* wait for key hit   */
PI2zerotime(1, 0x03);   /* zero timers on bits*/
                        /* 0 and 1            */
delay(2000);            /* wait 2 seconds     */
t0=PI2gettime(1, 0);    /* get time of bit 0  */
t1=PI2gettime(1, 1);    /* get time of bit 1  */
```

This program waits for a keyboard hit, then zeros the input timers on bits 0 and 1 (Inputs 1 and 2) and then waits 2 seconds for a response before getting the times. **PI2gettime** returns 0 if the input bit has not gone true.

PI2write(din, bitcode)

Op Code PI2_WRITE {0x14}

Prototype void PI2write(int *din,* int *bitcode*);

Description Writes a specific bit pattern to the main output lines. Note that any outputs specified to have negative logic will have output levels inverted from the *bitcode* specified for that line. Also, bits corresponding to I/O lines programmed as inputs have no significance.

Arguments *din* Device index number.

bitcode Output bit code.

Example

```
PI2clear(1);
PI2outs(1, 0xff);
PI2logic(1, 0x0f, 0x00);
PI2write(1, 0xff);
```

After the above program is run, LEDs #1 through #4 will be off while LEDs #5 through #8 will be on.

PI2read(din)

Op Code PI2_READ {0x15}

Prototype int PI2read(int *din*);

Description Reads the bit pattern on the main input lines. Note that any inputs designated to have negative logic will read as 1 when the input is low and 0 when the input is high.

Arguments *din* Device index number.

Example

```
PI2clear(1);
PI2logic(1, 0x00, 0xfe);
i = PI2read(1);
```

If input 1 is held low while inputs 2 through 7 are allowed to float high, **PI2read** will return 0x00.

PI2setbit(din, bitmask)

Op Code PI2_SETBIT {0x18}

Prototype void PI2setbit(int *din,* int *bitmask*);

Description Similar to **PI2write**, but sets output lines specified in a bit mask and leaves the others unchanged (logical OR). Note that outputs specified for negative logic will have "low" output levels if set. Also, bits corresponding to Smart Port lines programmed as inputs have no significance.

Arguments

din	Device index number.
bitmask	Output bit code.

Example

```
PI2clear(1);
PI2outs(1, 0xf0);
PI2setbit(1, 0x60);
```

The above code segment will illuminate LEDs 2 and 3 on a standard TDT response box.

PI2clrbit(din, bitmask)

Op Code PI2_CLRBIT {0x19}

Prototype void PI2clrbit(int *din,* int *bitmask*);

Description Similar to **PI2write**, but clears output lines specified by a bit mask and leaves the others unchanged. Note that outputs specified to have negative logic will have "high" output levels if cleared. Also, bits in the mask corresponding to I/O lines programmed as inputs have no significance.

Arguments

din	Device index number.
bitmask	Output bit code.

Example

```
PI2clear(1);
PI2outs(1, 0xf0);
PI2setbit(1, 0xf0);
PI2clrbit(1, 0x60);
```

The above code segment will briefly flash all LEDs (1-4) on a standard TDT response box, then will turn off LEDs 2 and 3.

- The four Auxiliary I/O Ports are accessed using the following three command calls. (All previous commands apply only to the eight "smart" I/O lines.)

PI2outsX(din, pnum)

Op Code PI2_OUTS {0x1E}

Prototype void PI2outsX(int *din*, int *pnum*);

Description Configures an auxiliary I/O port to be an output.

Arguments *din* Device index number.

pnum Port number (1...4).

Example

```
PI2outsX(1,2);
```

This call will make auxiliary I/O port 2 an output port.

PI2writeX(din, pnum, bitcode)

Op Code PI2_WRITEX {0x1F}

Prototype void PI2write(int *din*, int *pnum*, int *bitcode*);

Description Writes the specified bit code to an auxiliary port specified as output.

Arguments *din* Device index number.

pnum Port number (1..4).

bitcode Output bit code (0..0xff).

Example

```
PI2clear(1);
PI2outsX(1,2);
PI2writeX(1, 2, 0xff);
```

This will set all bits on auxiliary port 2 high.

PI2readX(din, pnum)

Op Code PI2_READX {0x20}

Prototype int PI2readX(int *din*, int *pnum*);

Description Reads the current input lines. Note that any inputs designated to have negative logic will read as 1 when the input is low and 0 when the input is high.

Arguments *din* Device index number.

pnum Port number.

Example

```
PI2clear(1);
i = PI2readX(1, 1);
```

This will read the 8-bit pattern on auxiliary port 1.

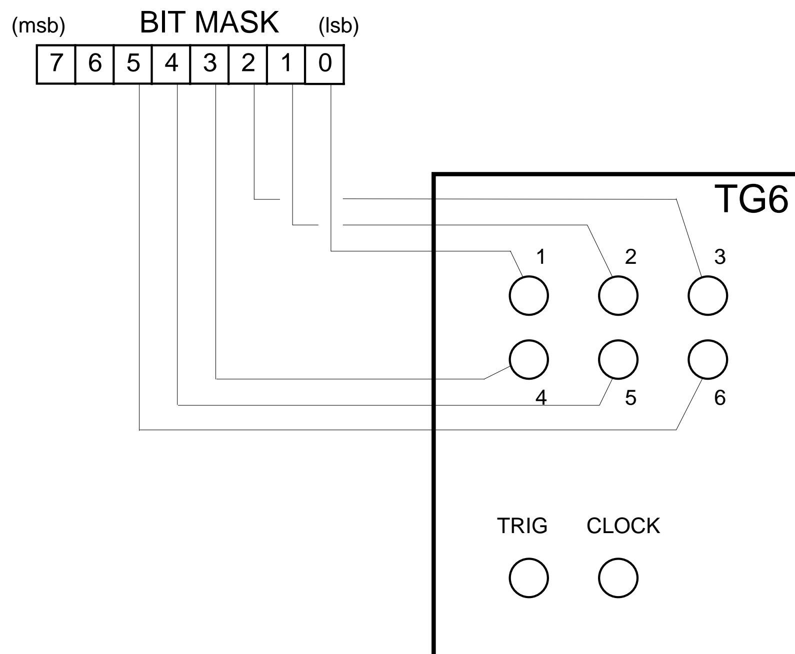
TG6 Timing Generator

Overview

The TG6 is a timing generator utilizing a dedicated microprocessor and high speed static RAM to produce up to six arbitrary timing sequences.

The TG6 programming procedures make extensive use of mask-type arguments, whereby each bit in a 6-bit number, or mask, is used to set specific output high or low for a specific interval. Since the TG6 is a TTL device, when a channel is set high, it will produce a voltage between 2.8V and 5V. When a channel is set low, its voltage will be between 0V and 0.8 V.

The following diagram illustrates the relation between mask bits and the TG6's front panel BNCs. Thus, bit mask position '0' corresponds to channel 1 of the TG6, and bit mask position '4' corresponds to channel 5 of the TG6. Bit mask positions 6 and 7 are not used in setting the TG6.



Binary/Hexadecimal Conversion Chart

This chart can assist you in determining the appropriate hexadecimal code for selecting different combinations of channels. In 'C' hexadecimal numbers are preceded by '`0x`'. In Pascal hexadecimal numbers are preceded by '`$`'.

Binary	Decimal	Hexadecimal (hex)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Examples:

To set a mask for channels 2, 4, and 6 (00101010 in binary), you would use a mask number of 2A in hex. This can be easily determined from the chart above, because 0010 (the left-hand four digits of the binary code) is 2 in hex, and 1010 (the right-hand four digits of the binary code) is A in hex. Thus, put together, 00101010 equals 2A in hex.

To set a mask for channels 2, 3, and 4 (00001110 in binary), you would use a mask number of 14 decimal, or 0E in hex. In 'C' you would write `0x0E`, in Pascal you would write `$0E`.

TG6clear(din)

Op Code TG6_CLEAR {0x11}

Prototype void TG6clear(int *din*);

Description Clears all timing sequences and sets all outputs low. Default configuration is as follows:

- 1µs base clock period.
- Single rep. after trigger.

Arguments *din* Device index number.

Example

```
TG6clear(1);
```

This resets TG6_(1).

TG6arm(din, snum)

Op Code TG6_ARM {0x12}

Prototype void TG6arm(int *din,* int *snum*);

Description Arms the TG6 to generate the specified timing sequence when triggered to start (externally or from software).

Arguments *din* Device index number.
snum Sequence number (0...7).

Example See **TG6go**.

TG6go(din)

Op Code TG6_GO {0x13}

Prototype void TG6go(int *din*);

Description Triggers the currently armed timing sequence (see **TG6arm**).

Arguments *din* Device index number.

Example

```
TG6arm(1,3);
TG6go(1);
```

This will arm and start timing sequence number three.

TG6tgo(din)

Op Code TG6_TGO {0x14}

Prototype void TG6tgo(int *din*);

Description Prepares the TG6 to receive a global or local software trigger to synchronize it with other XBUS devices (see **XB1gtrig** and **XB1ltrig**).

Arguments *din* Device index number.

Example

```
TG6arm(1,3);
TG6tgo(1);/*ready for global software trig.*/
{arm and tgo other relevant devices}
XB1gtrig();/*issue global software trig.*/
```

This will arm and start the TG6 timing sequence and other devices (readied with their corresponding **tgo** commands) simultaneously.

TG6stop(din)

Op Code TG6_STOP {0x15}

Prototype void TG6stop(int *din*);

Description Stops the TG6 from software after completion of the current sequence repetition.

Arguments *din* Device index number.

Example See **TG6status**.

TG6baserate(din, brcode)

Op Code TG6_BASERATE {0x16}

Prototype void TG6baserate(int *din*, int *brcode*);

Description Sets the TG6's base clock rate for output timing sequences. Sequence patterns are specified in terms of integers 'ticks' of this clock.

Arguments *din* Device index number.
brcode Base rate time period:

_100ns	0
_1µs	1
_10µs	2
_100µs	3
_1ms	4
_EXT	7

Example See example under **TG6new**.

TG6reps(din, rmode, rcount)

Op Code TG6_REPS {0x1C}

Prototype void TG6reps(int *din*, int *rmode*, int *rcount*);

Description Sets the sequence repetition mode and count. The timing sequence can be made to run (1)continuously after triggering, (2) for a set number of repetitions after a trigger or (3) once each trigger for a set number of repeat triggers.

Arguments *din* Device index number.
rmode Repetition mode:

CONTIN_REPS	0
TRIGGED_REPS	1

rcount Repetition count (0...32,000).
0 => infinite repetitions.

Example 1
Continuous

```
TG6reps(1, CONTIN_REPS, 0);
TG6arm(1,3);
TG6go(1);
```

TG6_(1) will produce timing sequence number 3 until **TG6stop** is called.

Example 2
Repeat n times

```
TG6reps(1, CONTIN_REPS, 10);
TG6arm(1,3);
TG6go(1);
```

TG6_(1) will produce timing sequence number 3 for 10 repetitions. It must be armed before triggering again.

Example 3
Once for n trigs

```
TG6reps(1, TRIGGED_REPS, 10);
TG6arm(1,3);
/* TG6 ready for ext. TTL trigger... */
```

TG6_(1) will produce timing sequence number 3 *once* on each external trigger for 10 triggers, and then stop responding to triggers.

TG6new(din, snum, lgth, dmask)

Op Code TG6_NEW {0x17}

Prototype void TG6new(int *din*, int *snum*, int *lgth*, int *dmask*);

Description Defines a new timing sequence in a segment of the TG6's memory. Up to eight separate sequences can be defined and accessed independently and are retained until power is turned off or a **TG6clear** is issued. Each sequence is 6 bits wide (one bit for each output) and the total length of all seven sequences cannot exceed 32,767 'ticks' of the base clock (period is set with **TG6baserate**).

Arguments

din Device index number.

snum Sequence number (0...7).

lgth Sequence length (0...32767) 'ticks'.

dmask Default data mask (0...63/0x3f) dec/hex (the corresponding outputs are set default high).

Example

```
TG6baserate(1, _1ms);
TG6new(1, 3, 200, 0x05);
```

This defines an area of TG6 memory for a new timing sequence (#3) which will be 200 ms long. The outputs 1 and 3 (bits 0 and 2) are set high, and outputs 2, 4, 5, and 6 (bits 1, 3, 4, and 5) are set low: 0x05 hex = 000101 binary. The default mask setting facilitates active high or low logic implementation using **TG6high** and **TG6low**.

TG6high(din, snum, _beg, _end, hmask)

Op Code TG6_HIGH {0x18}

Prototype void TG6high(int *din*, int *snum*, int *_beg*, int *_end*, int *hmask*);

Description Sets selected TG6 outputs high for an interval of a timing sequence. The outputs to set high correspond to true bits (1's) in *hmask* (other outputs are unchanged) and the time period is specified by the *_beg* and *_end* parameters.

Arguments

din	Device index number.
snum	Sequence number (0...7).
_beg	Seq. position to begin at (0 ... seq. lgth - 1).
_end	Seq. position to end at (*_beg*+1 ... seq. lgth - 1).
hmask	Bit locations to set high (0...63/0x3f) dec/hex.

Example 1

```
TG6high(1, 1, 10, 11, 0x01);
```

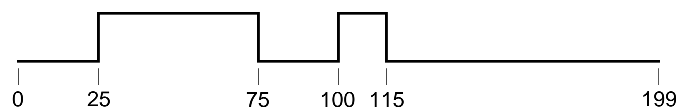
Channel 1 will produce a one-count pulse at time 10.

Example 2

```
TG6baserate(1, _1ms);
TG6new(1, 3, 200, 0x00);
TG6high(1, 3, 25, 75, 0x0a);
TG6high(1, 3, 100, 115, 0x0a);
TG6high(1, 3, 50, 130, 0x20);
```

TG6new defines a new timing sequence (#3) and sets all channels default low. **TG6high** sets outputs 2, 4 and 6 to produce the timing sequence shown below:

TG6 Outputs 2 and 4



TG6 Output 6

All other outputs remain unchanged from their default states.

TG6low(din, snum, _beg, _end, lmask)

Op Code TG6_LOW {0x19}

Prototype void TG6low(int *din*, int *snum*, int *_beg*, int *_end*, int *lmask*);

Description Sets selected TG6 outputs low for an interval of a timing sequence. The outputs to set low correspond to true bits (1's) in *lmask* (other outputs are unchanged) and the time period is specified by the *_beg* and *_end* parameters.

Arguments

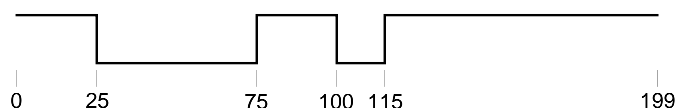
din	Device index number.
snum	Sequence number (0...7).
_beg	Seq. position to begin at (0 ... seq. lgth - 1).
_end	Seq. position to end at (*_beg*+1 ... seq. lgth - 1).
lmask	Bit locations to set low (0...63/0x3f) dec/hex.

Example

```
TG6baserate(1, _1ms);
TG6new(1, 3, 200, 0x05);
TG6low(1, 3, 25, 75, 0x05);
TG6low(1, 3, 100, 115, 0x05);
```

This defines a new timing sequence (#3) and sets outputs 1 and 3 (set default high by **TG6new**) to produce the timing sequence shown below:

TG6 Outputs 1 and 3



All other outputs remain unchanged from their default states.

TG6value(din, snum, _beg, _end, val)

Op Code TG6_VALUE {0x1a}

Prototype void TG6value(int *din*, int *snum*, int *_beg*, int *_end*, int *val*);

Description Sets a specific bit pattern on TG6 outputs for an interval of a timing sequence. It combines the actions of **TG6high** and **TG6low** by setting *all* outputs to the bit pattern of *val* (1=high, 0=low) over the time period specified by the *_beg* and *_end* parameters.

NOTE: Unlike **TG6high** and **TG6low**, you cannot set or clear certain bits without affecting others.

Arguments

din Device index number.

snum Sequence number (0...7).

_beg Seq. position to begin at (0 ... seq. lgth - 1).

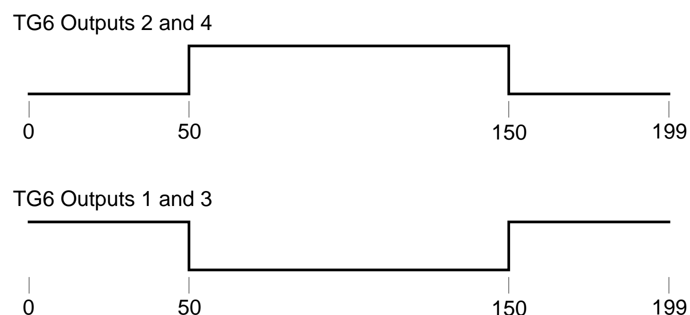
_end Seq. position to end at (*_beg*+1 ... seq. lgth - 1).

val Bit pattern to set (0...63/0x3f) dec/hex.

Example

```
TG6baserate(1, _1ms);
TG6new(1, 3, 200, 0x05);
TG6value(1, 3, 50, 150, 0x0a);
```

This defines a new timing sequence (#3) and sets outputs 1 and 3 (set default high by **TG6new**), and outputs 2 and 4 (set default low by **TG6new**) to produce the timing sequence shown below:



TG6dup(din, snum, s_beg, s_end, d_beg, ndup, dmask)

Op Code TG6_DUP {0x1B}

Prototype void TG6dup(int *din*, int *snum*, int s_*beg*, int s_*end*, int *d_beg*, int *ndup*,int *dmask*);

Description Duplicates a section of a timing sequence throughout the rest of the sequence memory on specified outputs. This feature simplifies creation of long, repetitive 'master' timing sequences on certain TG6 outputs. Arbitrary sequences can then be specified on other outputs using **TG6high** and **TG6low**.

Arguments

din	Device index number.
snum	Sequence number (0...7).
s_beg	Begin. of section to dup. (0 ... seq. lgth - 1).
s_end	End of section to dup. (*s_beg*+1 ... seq. lgth - 1).
d_beg	Position to begin dup. (*s_end*+1 ... seq. lgth - 1).
ndup	No. of duplications (0...32000).
dmask	Bit positions to dup. (0...63/0x3f) dec/hex.

Example 1 The following example illustrates the creation of a 500kHz clock on channel 1 of the TG6:

500kHz clock

```
TG6baserate(1, _1us);
TG6new(1, 2, 100, 0x00);              100-count seq.
TG6high(1, 2, 0, 1, 0x01);            one clock pulse
TG6dup (1, 2, 0, 2, 2, 50, 0x01);     make 50 copies
TG6reps (1, CONTIN_REPS, 0);          play continuously
```

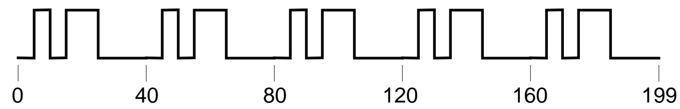
Example 2

Arbitrary seqs.

```
TG6baserate(1, _1ms);
TG6new(1, 3, 200, 0x00);
TG6high(1, 3, 5, 10, 0x01);
TG6high(1, 3, 15, 25, 0x01);
TG6dup(1, 3, 0, 40, 40, 4, 0x01);
TG6high(1, 3, 30, 125, 0x02);
TG6high(1, 3, 80, 150, 0x04);
```

This creates a 'master' timing sequence on output 1 and two other sequences on outputs 2 and 3 as shown below:

TG6 Output 1



TG6 Output 2

TG6 Output 3

TG6status(din)

Op Code TG6_STATUS {0x1D}

Prototype int TG6status(int *din*);

Description Returns the output status of the specified TG6.

Arguments *din* Device index number.

Return Values:

IDLE	0	TG6 stopped.
ARM	1	TG6 armed, ready for trigger.
ACTIVE	2	Timing sequence in progress.
LAST	3	Last rep. of timing sequence in progress.

Example

```
TG6arm(1,0);
TG6go(1);
do{
  if(kbhit()) TG6stop(1);
}while(TG6status(1)!=IDLE);
```

The do-loop waits until TG6_(1) stops (returns IDLE). If a key is pressed during the loop, the TG6_(1) will stop upon completion of the present sequence repetition.

(this page intentionally left blank)

WG2 Waveform Generator

Note: The WG2 must be installed in a caddie that is communicating with the host computer at 38400 baud.

WG2on(din)

Op Code WG2_ON {0x11}

Prototype void WG2on(int *din*);

Description Tells the specified WG2 to begin producing the programmed wave shape. When using the external trigger/enable, **WG2on** acts as an overall device enable (use **WG2off** to disable the WG2). When the trigger mode is set to NONE, **WG2on** will activate the WG2 output signal.

Arguments *din* Device index number.

Example

```
WG2trig(1,NONE);
WG2on(1);
```

This will turn on the WG2_(1). WG2_(1) will produce waveform as configured prior to this call.

WG2off(din)

Op Code WG2_OFF {0x12}

Prototype void WG2off(int *din*);

Description Places the specified WG2 in IDLE mode. **WG2off** will act as an overall system disabler when the external trigger/enable is used. When triggering is set to NONE, **WG2off** will turn off the device's output signal.

Arguments *din* Device index number.

Example

```
WG2off(1);
```

This will stop WG2_(1) from producing output signal.

WG2ton(din)

Op Code WG2_TON {0x21}

Prototype void WG2ton(int *din*);

Description Similar to **WG2on**, but causes the specified WG2 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
WG2trig(1,NONE);
WG2ton(1);
WG2ton(2);
XB1gtrig();            /* Global trigger */
```

This will cause WG2_(1) and WG2_(2) to turn ON at precisely the same time.

WG2clear(din)

Op Code WG2_CLEAR {0x13}

Prototype void WG2clear(int *din*);

Description Clears the specified WG2 and resets it to the factory default setup.

Arguments *din* Device index number.

Example

```
WG2clear(1);
```

This clears WG2_(1) and resets it.

WG2amp(din, amp)

Op Code WG2_AMP {0x14}

Prototype void WG2amp(int *din,* float *amp*);

Description Sets the output waveform amplitude.

Note With WG2 hardware versions lower than 4.0, when the WG2 is generating Gaussian noise at its max level, the amplitude display will read 9.99 on the display. Decreasing the amplitude to 5.0 will not reduce the level of the noise by 6.0dB. This is corrected with WG2 hardware versions 4.0 and higher.

Arguments
din Device index number.
amp Amplitude in volts (0.00...9.99).

Example

```
WG2amp(1, 5.00);
```

This sets WG2_(1) output amplitude to 5.00 V_p.

WG2freq(din, freq)

Op Code WG2_FREQ {0x15}

Prototype void WG2freq(int *din,* float *freq*);

Description Sets the sine wave frequency.

Arguments
din Device index number.
freq Frequency in Hz (1...20,000).

Example

```
WG2shape(1,SINE);
WG2freq(1, 5000);
WG2on(1);
```

This programs WG2_(1) to produce a sine wave output with a frequency of 5kHz.

WG2swrt(din, swrt)

Op Code WG2_SWRT {0x16}

Prototype void WG2swrt(int *din,* float *swrt*);

Description Sweeps the sine wave frequency for 'chirp' tones.

Arguments *din* Device index number.

swrt Frequency sweep rate in Hz/s (10...9,999).

Example

```
WG2shape(1,SINE);
WG2freq(1, 200);
WG2swrt(1, 1000);
WG2on(1);
```

This programs WG2_(1) to produce a swept-frequency sine wave, starting at 200Hz. After one second, the frequency will be 1.2 kHz. NOTE: The frequency continues increasing until the WG2 is gated off, or until **WGoff** is issued.

WG2phase(din, phase)

Op Code WG2_PHASE {0x17}

Prototype void WG2phase(int *din,* float *phase*);

Description Sets the onset phase of the WG2's sine wave generator as well as the seed value for the WG2's internal random number generator. Using the phase call, the WG2 can be made to generate frozen noise signals each time the device is triggered. To generate non-frozen noise call WG2phase(1, -1).

Arguments *din* Device index number.

phase Phase in degrees (0.00...360.00). Use (-1) to set the phase to continuous.

Example

```
WG2trig(1, POS_ENABLE);
WG2shape(1, SINE);
WG2freq(1, 200);
WG2phase(1, 90);
WG2on(1);
```

This programs WG2_(1) to produce a 200Hz sine wave with a 90 degree onset phase, synchronized with the Enable input.

WG2dc(din, dc)

Op Code WG2_DC {0x18}

Prototype void WG2dc(int *din*, float *dc*);

Description Sets the DC shift level in the output waveform. NOTE: This shift remains on the output even when the signal is gated off. DC shifting is useful when generating AM modulated signals.

Arguments
din Device index number.
dc DC shift level in volts (-9.9...9.9).

Example
```
WG2dc(1, -1.0);
```
This programs WG2_(1) output to have a -1.0 V DC shift.

WG2shape(din, scode)

Op Code WG2_SHAPE {0x19}

Prototype void WG2shape(int *din*, int *scode*);

Description Selects the WG2 waveform shape. The valid shape types are listed below.

Arguments
din Device index number.
scode Shape type code (1...4).

GAUSS	Gaussian noise.
UNIFORM	Uniform noise.
SINE	Sine wave.
WAVE	User-defined shape.

Example
```
WG2amp(1,6.00);
WG2shape(1,GAUSS);
WG2on(1);
```
This will program WG2_(1) to produce a Gaussian noise signal with a +/- 6 volt maximum amplitude.

WG2dur(din, dur)

Op Code WG2_DUR {0x1A}

Prototype void WG2dur(int *din*, float *dur*);

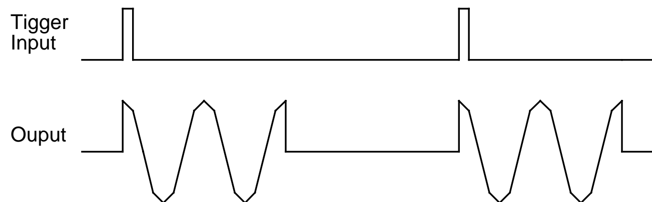
Description Sets/enables the WG2's self-timing feature. When the self-timer is enabled the signal will gate on when a valid trigger is detected and gate off after the specified duration. Specify a duration of zero for continuous output.

Arguments
din Device index number.
dur Duration of gate in ms (1...30,000).

Example

```
WG2trig(1,POS_EDGE);
WG2rf(1,0.0);
WG2shape(1,SINE);
WG2freq(1, 1000);
WG2phase(1, 90.0);
WG2dur(1, 2);
WG2on(1);
```

This program will produce the output shown below:



WG2rf(din, rf)

Op Code WG2_RF {0x1B}

Prototype void WG2rf(int *din*, float *rf*);

Description Sets the rise and fall time of the built-in gating feature.

NOTE: The WG2 has limited gating abilities. Only the following gate rise/fall times will give guaranteed results:

0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
10., 20., 30., 40., 50., 60., 70., 80., 90.

Arguments
din Device index number.
rf Rise and fall time of gate in ms (1.0...90.0).

Example
```
WG2rf(1, 3.0);
```
This call sets the rise and fall times of 3.0 ms to WG2_(1)'s gating circuit.

WG2trig(din, tcode)

Op Code WG2_TRIG {0x1C}

Prototype void WG2trig(int *din*, int *tcode*);

Description Programs the ENABLE/TRIGGER input of the WG2. The valid trigger types are listed below.
NOTE: The duration feature works only with EDGE type (see **WG2dur)** and NONE (manual) triggering.

Arguments
din Device index number.
tcode Trigger type code (1 ... 5).

POS_EDGE	Rising edge triggered.
NEG_EDGE	Falling edge triggered.
POS_ENABLE	High level enabled.
NEG_ENABLE	Low level enabled.
NONE	Free running.

Example
```
WG2trig(1,POS_EDGE);
```
This will program WG2_(1) to rising edge triggered.

WG2seed(din, seed)

Op Code WG2_SEED {0x1D}

Prototype void WG2seed(int *din,* long *seed*);

Description Sets the random number generator seed value.

Arguments
din Device index number.
seed Random number seed value ($1...2^{32}$).

Example

```
WG2shape(1,UNIFORM);
WG2seed(1, 1000000);
WG2on(1);
```

This sets WG2_(1) as a uniform noise generator and seeds it with 1,000,000.

WG2delta(din, delta)

Op Code WG2_DELTA {0x1E}

Prototype void WG2delta(int *din,* long *delta*);

Description Sets a precise sine wave frequency.

Arguments

din Device index number.

delta $= 2^{32} \cdot$ (frequency in Hz)/55854.5 Hz
$= 76895.636 \cdot$ (frequency in Hz)

Example

```
WG2shape(1,SINE);
WG2delta(1,(long)(76895.636 * 100.0));
WG2on(1);
```

Programs WG2_(1) to produce a 100Hz sine wave.

WG2wave(din, wave, npts)

Op Code WG2_WAVE {0x1F}

Prototype void WG2wave(int *din,* int *wave[],* int *npts*);

Description Uploads a user-defined waveform into the WG2 sample buffer from a host PC.

Arguments

din Device index number.

wave Pointer to waveform buffer.

npts number of buffer points to use (1 ... 800).

Example

```
c=32767/400
for(i=-400;i<400;i++)
  wave[i+400]=i*c;
WG2wave(1,wave,800);
WG2shape(1,WAVE);
WG2on(1);
```

This uploads an 800 point, zero DC shift ramp function into the WG2_(1) and replays the contents of the sample buffer until gated off.

WG2status(din)

Op Code WG2_STATUS {0x20}

Prototype int WG2status(int *din*);

Description Returns the status of the specified WG2.

WARNING: Calling **WG2status** while the device is generating tones or converting a user waveform will cause clicks in the WG2 output waveform. Do not use **WG2status** while the device is in one of these two modes.

Return Values:

OFF = 0	WG2 is off (no output).
ON = 1	WG2 output gated fully on.
RISING = 2	Output gate is in rising state.
FALLING = 3	Output gate is in falling state.

Arguments *din* Device index number.

Example

```
if(WG2status(1)==ON);
   WG2off(1);
```

Turns WG2(1) off if it is on.

(this page intentionally left blank)

DA1 Dual D/A Converter

DA1clear(din)

Op Code DA1_CLEAR {0x1E}

Prototype void DA1clear(int *din*);

Description Clears the specified DA1 module and resets it to factory default settings.

Factory Defaults:

- Sampling rate: 1/20µs = 50kHz
- # of conversions: 1000
- Channel mode: DAC1
- Single triggering.
- Clip feature disabled.
- Zero output.

Arguments *din* Device index number.

Example See example under **DA1go**.

DA1arm(din)

Op Code DA1_ARM {0x13}

Prototype void DA1arm(int *din*);

Description Prepares the specified DA1 module to wait for a trigger to begin D/A conversion. **DA1arm** must be issued before the DA1 can be triggered (through either software or an external trigger source) to start D/A conversion.

Arguments *din* Device index number.

Example See example under **DA1go**.

DA1go(din)

Op Code DA1_GO {0x11}

Prototype void DA1go(int *din*);

Description Triggers the specified DA1 from software to begin D/A conversion. **DA1arm** must be issued first to 'prime' the DA1.

Arguments *din* Device index number.

Example

```
DA1clear(1);
DA1arm(1);
DA1go(1);
```

This example will reset DA1_(1) to its default setting and begin conversion.

DA1stop(din)

Op Code DA1_STOP {0x12}

Prototype void DA1stop(int *din*);

Description Forces the specified DA1 to stop D/A conversion immediately.

NOTE: **DA1stop** will not zero the DAC outputs—a **DA1clear** must be issued to re-zero, if necessary.

Arguments *din* Device index number.

Example

```
delay(1000);
DA1stop(1);
```

These lines added to the example under **DA1go** will cause the DA1_(1) to stop conversion after a short delay period.

DA1tgo(din)

Op Code DA1_TGO {0x21}

Prototype void DA1tgo(int *din*);

Description Similar to **DA1go**, but causes the specified DA1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering features are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
DA1arm(1);
DA1arm(2);
DA1tgo(1);
DA1tgo(2);
XB1gtrig();            /* Global trigger */
```

This will first prepare both DA1_(1) and DA1_(2) to wait for a same trigger and begin conversion at precisely the same time.

DA1mode(din, mcode)

Op Code DA1_MODE {0x14}

Prototype void DA1mode(int *din*, int *mcode*);

Description Selects which of the specified DA1's two channels will be used for conversion. The channel mode for D/A conversion is specified by *mcode*.

Arguments *din* Device index number.

mcode Mode code:

DAC1	0x01	D/A channel 1
DAC2	0x02	D/A channel 2
DUALDAC	0x03	Both D/A channels
FASTDAC	0x10	Fast single-channel D/A

The FASTDAC option places the DA1 into a special 500kHz sampling rate mode using channel 1 only.

Example

```
DA1mode(1,DAC1);
```

This call will set up DA1_(1) to use D/A channel 1.

DA1strig(din)

Op Code DA1_STRIG {0x1A}

Prototype void DA1strig(int *din*);

Description Sets up the specified DA1 for single triggering of a conversion buffer. **DA1strig** can be issued while the DA1 is active (see example under **DA1mtrig**).

Arguments *din* Device index number.

Example

```
DA1strig(1);
DA1arm(1);
```

This will cause DA1_(1) to begin converting on an external trigger or **DA1go**, but will not re-trigger unless the **DA1arm** is issued again.

DA1mtrig(din)

Op Code DA1_MTRIG {0x19}

Prototype void DA1mtrig(int *din*);

Description Sets up the specified DA1 for multiple external triggering of a conversion buffer.

NOTE: **DA1strig** can be issued while the DA1 is in multiple triggering mode to place it back into single triggering mode. This provides a way to stop the DA1 from software after the next buffer conversion is complete without disabling the external triggering source.

Arguments *din* Device index number.

Example

```
DA1mtrig(1);
DA1arm(1);
```

This will cause DA1_(1) to begin converting at each external trigger. The DA1 will not re-trigger until the present conversion cycle is complete.

DA1reps(din, nreps)

Op Code DA1_REPS {0x1C}

Prototype void DA1reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the DA1 can be re-triggered after a **DA1arm** is issued. Valid only in multiple trigger mode.

Arguments
din Device index number.
nreps Repeat triggers before rearming (0...60,000).

Example
```
DA1reps(1,100);
```
If this line is added to the previous example, DA1_(1) will respond only to the first 100 triggers received.

DA1srate(din, sper)

Op Code DA1_SRATE {0x15}

Prototype void DA1srate(int *din*, float *sper*);

Description Sets the sampling rate of the specified DA1 by assigning the sampling period. *Use only when compatibility with older hardware is required (see Note below).*

Arguments
din Device index number.
sper Sampling period (5µs...2000µs), (2µs...2000µs) for FASTDAC mode.

Example
```
DA1srate(1,20.0);
```
Sets the sampling period of DA1_(1) to 20µs, which corresponds to a sampling frequency of 50kHz.

Note DA1 derives the programmed sampling period from a 12.5 MHz base clock. This means that realizable sample periods are quantized to 0.08µs. For example, calling **DA1srate**(1, 7.0) will actually cause the device to sample every 6.96µs, because the rate must be derived from an integer number of cycles of the base clock (0.08µs). Also, the rate will be truncated down to the nearest integer number of 12.5MHz clock cycles, so calling **DA1srate**(1, 5.1)

will result in a 5.04µs sampling period not a 5.12µs as might be expected.

With XBDRV versions 2.00 or lower, **DA1srate** will pass the specified sample period to the nearest tenth of a microsecond. So if you make a call such as **DA1srate**(1, 6.98), it will be rounded and sent to the DA1 as 7.0µs, and the DA1 will truncate it down to the nearest integer number of 0.08µs, which is 6.96µs. XBDRV versions 2.01 and higher has provided the **DA1speriod** call to set sample period and return the actual sample period to correct this problem.

DA1speriod(din, sper)

Op Code DA1_SPERIOD {0x22}

Prototype float DA1speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified DA1. It returns the actual sample period.

Arguments

din Device index number.

sper Sampling period (5µs...2000µs), (2µs...2000µs) for FASTDAC mode.

Example

```
printf("  %f ", DA1speriod(1,5.00));
```

Outputs 5.04 to the screen and sets the sampling rate of DA1_(1) to 1/5.04µs = 198.413kHz.

Note **DA1speriod** corrects the round-off problem associated with **DA1srate** call, while the base clock limitation still apply (See note under **DA1srate**). **DA1speriod** call is available for DA1 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

DA1npts(din, npts)

Op Code DA1_NPTS {0x18}

Prototype void DA1npts(int *din*, long *npts*);

Description Sets the number of samples the specified DA1 will convert after it is triggered.

Arguments

din Device index number.

npts Number of conversions (1...2^{32}).

Example

```
DA1npts(1,500000);
```

This call will set up DA1_(1) to perform 500,000 conversions which corresponds to 10 seconds at 50kHz sampling rate.

DA1clkin(din, scode)

Op Code DA1_CLKIN {0x16}

Prototype void DA1clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified DA1. This is useful for synchronizing DA1 conversions with other devices.

Arguments *din* Device index number.

scode Clock source (1...4):

INTERNAL	1	Internal sample clock
EXTERNAL	2	Front panel BNC
XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line

Example

```
DA1clkin(1,EXTERNAL);
```

This call will set up DA1_(1) to use its front panel BNC for the sampling clock.

DA1clkout(din, dcode)

Op Code DA1_CLKOUT {0x17}

Prototype void DA1clkout(int *din*, int *dcode*);

Description Asserts the DA1's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with DA1 conversions.

Arguments *din* Device index number.

dcode Patch line code (3...5):

XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line
NONE	5	No clock patch

Example

```
DA1clkin(1,INTERNAL);
DA1clkout(1,XCLK1);
```

This call will set up DA1_(1) to use its internal sampling clock and assert it to the XCLK1 line on the XBUS, where it can be used to synchronize other devices on the bus.

DA1clipon(din)

Op Code DA1_CLIPON {0x1F}

Prototype void DA1clipon(int *din*);

Description Enables the DAC clip detection feature. If the D/A conversion data ever reach their maximum absolute value (±7FFF), the front-panel Clip light will trip and stay on until a **DA1arm** is issued.

NOTE: If this feature is enabled, the maximum sampling rate is limited to 100kHz.

Arguments *din* Device index number.

Example See example under **DA1clip**.

DA1status(din)

Op Code DA1_STATUS {0x1B}

Prototype int DA1status(int *din*);

Description Returns the status of the specified DA1.

Return Values:

IDLE	0	DA1 idle, no conversion in progress.
ARM	1	DA1 armed, ready for trigger.
ACTIVE	2	DA1 conversion in progress.

Arguments *din* Device index number.

Example

```
do{}while(DA1status(1)==ACTIVE);
```

This will cause the program to pause while DA1_(1) finishes converting the present sample buffer.

DA1clip(din)

Op Code DA1_CLIP {0x1D}

Prototype int DA1clip(int *din*);

Description Returns the status of the clip detector.

Return Values:

0 If no clip has occurred.

1 If a clip has occurred.

Arguments *din* Device index number.

Example

```
DA1clipon(1);
DA1arm(1);
```

```
DA1go(1);
do{ if(DA1clip(1)) DA1stop(1); }
  while(DA1status(1)==ACTIVE);
```

This example will stop DA1_(1) if the D/A clips at any time.

le

AD1 Dual A/D Converter

AD1clear(din)

Op Code AD1_CLEAR {0x1E}

Prototype void AD1clear(int *din*);

Description Clears the specified AD1 and resets it to factory default settings.

Arguments *din* Device index number.

Defaults
- Sampling rate: 1/20µs = 50kHz
- # of conversions: 1000
- Channel mode: ADC1
- Single triggering.
- Clip feature disabled.

Example See example for **AD1go**.

AD1arm(din)

Op Code AD1_ARM {0x13}

Prototype void AD1arm(int *din*);

Description Prepares the specified AD1 for A/D conversion. **AD1arm** must be issued before the AD1 can be triggered (from software or externally) to start conversion.

Arguments *din* Device index number.

Example See example for **AD1go**.

AD1go(din)

Op Code AD1_GO {0x11}

Prototype void AD1go(int *din*);

Description Triggers the specified AD1 from software to begin A/D conversion. **AD1arm** must be issued first to 'prime' the AD1.

Arguments *din* Device index number.

Example

```
AD1clear(1);
AD1arm(1);
```

```
AD1go(1);
```

This example will reset AD1_(1) to its default setting and begin conversion.

AD1stop(din)

Op Code AD1_STOP {0x12}

Prototype void AD1stop(int *din*);

Description Forces the specified AD1 to stop A/D conversion immediately.

Arguments *din* Device index number.

Example

```
delay(1000);
AD1stop(1);
```

These lines added to the example given for **AD1go** will cause the AD1_(1) to stop after a short delay period.

AD1tgo(din)

Op Code AD1_TGO {0x21}

Prototype void AD1tgo(int *din*);

Description Similar to **AD1go**, but causes the specified AD1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
AD1arm(1);
AD1arm(2);
AD1tgo(1);
AD1tgo(2);
XB1gtrig();            /* Global trigger */
```

This will cause AD1_(1) and AD1_(2) to begin conversion at precisely the same time.

AD1mode(din, mcode)

Op Code AD1_MODE {0x14}

Prototype void AD1mode(int *din*, int *mcode*);

Description Selects which of the specified AD1's two channels will be used for conversion. The channel mode for A/D conversion is specified by *mcode*.

Arguments

din Device index number.

mcode Mode code:

ADC1	0x04	A/D channel 1
ADC2	0x08	A/D channel 2
DUALADC	0x0C	Both A/D channels

Example

```
AD1mode(1,ADC1);
```

This call will set up AD1_(1) to use A/D channel 1.

AD1strig(din)

Op Code AD1_STRIG {0x1A}

Prototype void AD1strig(int *din*);

Description Sets up the specified AD1 for single triggering of a conversion buffer. **AD1strig** can be issued while the AD1 is active (see example under **AD1mtrig**).

Arguments

din Device index number.

Example

```
AD1strig(1);
AD1arm(1);
```

This will cause AD1_(1) to begin converting on an external trigger or **AD1go**, but will not re-trigger unless **AD1arm** is issued again.

AD1mtrig(din)

Op Code AD1_MTRIG {0x19}

Prototype void AD1mtrig(int *din*);

Description Sets up the specified AD1 for multiple external triggering of a conversion buffer.

NOTE: **AD1strig** can be issued while the AD1 is in multiple triggering mode to place it back into single triggering mode. This provides a way to stop the AD1 from software after the next buffer conversion is complete without disabling the external triggering source.

Arguments *din* Device index number.

Example

```
AD1mtrig(1);
AD1arm(1);
```

This will cause AD1_(1) to begin converting at each external trigger. The AD1 will not re-trigger until conversion of the present buffer is complete.

AD1reps(din, nreps)

Op Code AD1_REPS {0x1C}

Prototype void AD1reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the AD1 can be re-triggered after an **AD1arm** is issued. Valid only in multiple trigger mode.

Arguments *din* Device index number.

nreps Repeat triggers before rearming (0...60,000)

Example

```
AD1reps(1,100);
```

If this line is added to example given for **AD1mtrig**, AD1_(1) will respond only to the first 100 triggers received.

AD1srate(din, sper)

Op Code AD1_SRATE {0x15}

Prototype void AD1srate(int *din*, float *sper*);

Description Sets the sampling rate (in microseconds per sample) of the specified AD1. *Use only when compatibility with older hardware is required (see Note below).*

Arguments
din Device index number.
sper Sampling period (5µs...2000µs).

Example

```
AD1srate(1,20.0);
```

Sets the sampling rate of AD1_(1) to 1/20µs = 50kHz.

Note AD1 derives the programmed sampling period from a 12.5 MHz base clock. This means that realizable sample periods are quantized to 0.08µs. For example, calling **AD1srate**(1, 7.0) will actually cause the device to sample every 6.96µs, because the rate must be derived from an integer number of cycles of the base clock (0.08µs). Also, the rate will be truncated down to the nearest integer number of 12.5MHz clock cycles, so calling **AD1srate**(1, 5.1) will result in a 5.04µs sampling period not a 5.12µs as might be expected.

With XBDRV versions 2.00 or lower, **AD1srate** will pass the specified sample period to the nearest tenth of a microsecond. So if you make a call such as **AD1srate**(1, 6.98), it will be rounded and sent to the AD1 as 7.0µs, and the AD1 will truncate it down to the nearest integer number of 0.08µs, which is 6.96µs. XBDRV versions 2.01 and higher has provided the **AD1speriod** call to set sample period and return the actual sample period to correct this problem.

AD1speriod(din, sper)

Op Code AD1_SPERIOD {0x22}

Prototype float AD1speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified AD1. It returns the actual sample period.

Arguments
din Device index number.
sper Sampling period (5µs...2000µs).

Example
```
printf("  %f ", AD1speriod(1,5.00));
```
Outputs 5.04 to the screen and sets the sampling rate of AD1_(1) to 1/5.04µs = 198.413kHz.

Note **AD1speriod** corrects the round-off problem associated with **AD1srate** call, while the base clock limitation still apply (See note under **AD1srate**). **AD1speriod** call is available for AD1 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

AD1npts(din, npts)

Op Code AD1_NPTS {0x18}

Prototype void AD1npts(int *din*, long *npts*);

Description Sets the number of samples the specified AD1 will convert after it is triggered.

Arguments
din Device index number.
npts Number of conversions (1...2^{32}).

Example
```
AD1npts(1,500000);
```
This call will set up AD1_(1) to perform 500,000 conversions, which corresponds to 10 seconds at 50kHz sampling rate.

AD1clkin(din, scode)

Op Code AD1_CLKIN {0x16}

Prototype void AD1clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified AD1. This is useful for synchronizing AD1 conversions with other devices.

Arguments *din* Device index number.

scode Clock source (1...4):

INTERNAL	1	Internal sample clock
EXTERNAL	2	Front panel BNC
XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line

Example

```
AD1clkin(1,EXTERNAL);
```

This call will set up AD1_(1) to use its front panel BNC for the sampling clock.

AD1clkout(din, dcode)

Op Code AD1_CLKOUT {0x17}

Prototype void AD1clkout(int *din*, int *dcode*);

Description Asserts the AD1's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with AD1 conversions.

Arguments *din* Device index number.

dcode Patch line code (3 ... 5):

XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line
NONE	5	No clock patch

Example

```
AD1clkin(1,INTERNAL);
AD1clkout(1,XCLK1);
```

This call will set up AD1_(1) to use its internal sampling clock and assert it to the XCLK1 line on the XBUS, where it can be used to synchronize other devices on the bus.

AD1clipon(din)

Op Code AD1_CLIPON {0x1F}

Prototype void AD1clipon(int *din*);

Description Enables the ADC clip detection feature. If the A/D conversion data ever reach their maximum absolute value (±7FFF), the front-panel Clip light will trip and stay on until the next **AD1arm** is issued.

NOTE: If this feature is enabled, the maximum sampling rate is limited to 100kHz.

Arguments *din* Device index number.

Example See example under **AD1clip**.

AD1status(din)

Op Code AD1_STATUS {0x1B}

Prototype int AD1status(int *din*);

Description Returns the status of the specified AD1.

Return Values:

IDLE	0	AD1 idle, no conversion in progress.
ARM	1	AD1 armed, ready for trigger.
ACTIVE	2	AD1 conversion in progress.

Arguments *din* Device index number.

Example

```
do{}while(AD1status(1)==ACTIVE);
```

This will cause the program to pause while AD1_(1) finishes converting the present sample buffer.

AD1clip(din)

Op Code AD1_CLIP {0x1D}

Prototype int AD1clip(int *din*);

Description Returns the status of the clip detector.

Return Values:

0 If no clip has occurred.

1 If a clip has occurred.

Arguments *din* Device index number.

Example

```
AD1clipon(1);
AD1arm(1);
AD1go(1);
do{ if(AD1clip(1)) AD1stop(1); }
  while(AD1status(1)==ACTIVE);
```

This example will stop AD1_(1) if the A/D clips at any time.

(this page intentionally left blank)

DD1 Stereo Analog Interface

DD1clear(din)

Op Code DD1_CLEAR {0x1E}

Prototype void DD1clear(int *din*);

Description Clears the specified DD1 and resets it to factory default settings.

Arguments *din* Device index number.

Defaults
- Sampling rate: 1/20µs = 50kHz
- # of conversions: 1000
- Channel mode: DAC1
- Single triggering.
- Clip feature disabled.

Example See example for **DD1go**.

DD1arm(din)

Op Code DD1_ARM {0x13}

Prototype void DD1arm(int *din*);

Description Prepares the specified DD1 for A/D - D/A conversion. **DD1arm** must be issued before the DD1 can be triggered (from software or externally) to start conversion.

Arguments *din* Device index number.

Example See example for **DD1go**.

DD1go(din)

Op Code DD1_GO {0x11}

Prototype void DD1go(int *din*);

Description Triggers the specified DD1 from software to begin A/D - D/A conversion. **DD1arm** must be issued first to 'prime' the DD1.

Arguments *din* Device index number.

Example
```
DD1clear(1);
DD1arm(1);
```

```
DD1go(1);
```

This example will reset DD1_(1) to its default setting and begin conversion.

DD1stop(din)

Op Code DD1_STOP {0x12}

Prototype void DD1stop(int *din*);

Description Forces the specified DD1 to stop A/D - D/A conversion immediately.

NOTE: **DD1stop** will not zero the DAC outputs, a **DD1clear** must be issued to re-zero, if necessary.

Arguments *din* Device index number.

Example

```
delay(1000);
DD1stop(1);
```

These lines added to the example above will cause the DD1_(1) to stop after a short delay period.

DD1tgo(din)

Op Code DD1_TGO {0x21}

Prototype void DD1tgo(int *din*);

Description Similar to **DD1go**, but causes the specified DD1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
DD1arm(1);
DD1arm(2);
DD1tgo(1);
DD1tgo(2);
XB1gtrig();            /* Global trigger */
```

This will cause DD1_(1) and DD1_(2) to begin conversion at precisely the same time.

DD1mode(din, mcode)

Op Code DD1_MODE {0x14}

Prototype void DD1mode(int *din*, int *mcode*);

Description Selects which of the specified DD1's four channels will be used for conversion. The combination of A/D and D/A channels to be used is specified by the lower four bits of *mcode*.

Arguments

din Device index number.

mcode Mode code (0x00...0x0F):

DAC1	0x01	D/A channel 1
DAC2	0x02	D/A channel 2
DUALDAC	0x03	Both D/A channels
ADC1	0x04	A/D channel 1
ADC2	0x08	A/D channel 2
DUALADC	0x0C	Both A/D channels
FASTDAC	0x10	Fast single-channel D/A
ALL	0x0F	All channels converting

Other combinations are achieved by directly setting bits 0-3 of *mcode*:

bit-3	bit-2	bit-1	bit-0
ADC2	ADC1	DAC2	DAC1

or by adding the above defined constants as in the following example.

NOTE: FASTDAC places the DD1 into a special D/A channel 1 only mode with 500kHz sampling rate, and cannot be combined with other options.

Example
```
DD1mode(1,DAC1+ADC2);
```
This call will set up DD1_(1) to use D/A channel 1 and A/D channel 2.

DD1strig(din)

Op Code DD1_STRIG {0x1A}

Prototype void DD1strig(int *din*);

Description Sets up the specified DD1 for single triggering of a conversion buffer. **DD1strig** can be issued while the DD1 is active (see example under **DD1mtrig**).

Arguments *din* Device index number.

Example
```
DD1strig(1);
DD1arm(1);
```
This will cause DD1_(1) to begin converting on an external trigger or **DD1go**, but will not re-trigger unless **DD1arm** is issued again.

DD1mtrig(din)

Op Code DD1_MTRIG {0x19}

Prototype void DD1mtrig(int *din*);

Description Sets up the specified DD1 for multiple external triggering of a conversion buffer.
NOTE: **DD1strig** can be issued while the DD1 is in multiple external triggering mode to place it back into single triggering mode. This provides a way to stop the DD1 from software after the next buffer conversion is complete without disabling the external triggering source.

Arguments *din* Device index number.

Example

```
DD1mtrig(1);
DD1arm(1);
```

This will cause DD1_(1) to begin converting at each external trigger. The DD1 will not re-trigger until conversion of the present buffer is complete.

DD1reps(din, nreps)

Op Code DD1_REPS {0x1C}

Prototype void DD1reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the DD1 can be re-triggered after a **DD1arm** is issued. Valid only in multiple trigger mode.

Arguments
din Device index number.
nreps Repeat triggers before re-arming (0...60,000)

Example

```
DD1reps(1,100);
```

If this line is added to the example given for **DD1mtrig**, DD1_(1) will respond only to the first 100 triggers received.

DD1srate(din, sper)

Op Code DD1_SRATE {0x15}

Prototype void DD1srate(int *din*, float *sper*);

Description Sets the sampling rate (microseconds per sample) of the specified DD1. *Use only when compatibility with older hardware is required (see Note below).*

Arguments
din Device index number.
sper Sampling period (5.4µs...2000µs), (2µs...2000µs) for FASTDAC mode.

Example

```
DD1srate(1,20.00);
```

Sets the sampling rate of DD1_(1) to 1/20µs = 50kHz.

Note DD1 derives the programmed sampling period from a 12.5 MHz base clock. This means that realizable sample periods are quantized to 0.08µs. For example, calling **DD1srate**(1, 7.0) will actually cause

the device to sample every 6.96µs, because the rate must be derived from an integer number of cycles of the base clock (0.08µs). Also, the rate will be truncated down to the nearest integer number of 12.5MHz clock cycles, so calling **DD1srate**(1, 5.9) will result in a 5.84µs sampling period not a 5.92µs as might be expected.

With XBDRV versions 2.00 or lower, **DD1srate** will pass the specified sample period to the nearest tenth of a microsecond. So if you make a call such as **DD1srate**(1, 6.98), it will be rounded and sent to the DD1 as 7.0µs, and the DD1 will truncate it down to the nearest integer number of 0.08µs, which is 6.96µs. XBDRV versions 2.01 and higher has provided the **DD1speriod** call to set sample period and return the actual sample period to correct this problem.

DD1speriod(din, sper)

Op Code DD1_SPERIOD {0x22}

Prototype float DD1speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified DD1. It returns the actual sample period.

Arguments

din Device index number.

sper Sampling period (5.4µs...2000µs), (2µs...2000µs) for FASTDAC mode.

Example

```
printf("  %f ", DD1speriod(1,5.00));
```

Outputs 5.04 to the screen and sets the sampling rate of DD1_(1) to 1/5.04µs = 198.413kHz.

Note **DD1speriod** corrects the round-off problem associated with **DD1srate** call, while the base clock limitation still apply (See note under **DD1srate**). **DD1speriod** call is available for DD1 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

DD1npts(din, npts)

Op Code DD1_NPTS {0x18}

Prototype void DD1npts(int *din*, long *npts*);

Description Sets the number of samples the specified DD1 will convert after it is triggered.

Arguments
din Device index number.
npts Number of conversions (1...2^{32}).

Example

```
DD1npts(1,500000);
```

This call will set up DD1_(1) to perform 500,000 conversions, which corresponds to 10 seconds at 50kHz sampling rate.

DD1clkin(din, scode)

Op Code DD1_CLKIN {0x16}

Prototype void DD1clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified DD1. This is useful for synchronizing DD1 conversions with other devices.

Arguments
din Device index number.
scode Clock source (1...4):

INTERNAL	1	Internal sample clock
EXTERNAL	2	Front panel BNC
XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line

Example

```
DD1clkin(1,EXTERNAL);
```

This call will set up DD1_(1) to use its front panel BNC for the sampling clock.

DD1clkout(din, dcode)

Op Code DD1_CLKOUT {0x17}

Prototype void DD1clkout(int *din*, int *dcode*);

Description Asserts the DD1's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with DD1 conversions.

Arguments
din Device index number.
dcode Patch line code (3...5):

XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line
NONE	5	No clock patch

Example

```
DD1clkin(1,INTERNAL);
DD1clkout(1,XCLK1);
```

This call will set up DD1_(1) to use its internal sampling clock and assert it to the XCLK1 line on the XBUS, where it can be used to synchronize other devices on the bus.

DD1clipon(din)

Op Code DD1_CLIPON {0x1F}

Prototype void DD1clipon(int *din*);

Description Enables the DD1 clip detection feature. If the A/D or D/A conversion data ever reach their maximum absolute value (±7FFF), the front-panel Clip light will trip and stay on until the next **DD1arm** is issued.

NOTE: If this feature is enabled, the maximum sampling rate is limited to 100kHz.

Arguments *din* Device index number.

Example See example under **DDclip**.

DD1status(din)

Op Code DD1_STATUS {0x1B}

Prototype int DD1status(int *din*);

Description Returns the status of the specified DD1.

Return Values:

IDLE	0	DD1 idle, no conversion in progress.
ARM	1	DD1 armed, ready for trigger.
ACTIVE	2	DD1 conversion in progress.

Arguments *din* Device index number.

Example

```
do{}while(DD1status(1)==ACTIVE);
```

This will cause the program to pause while DD1_(1) finishes converting the present sample buffer.

DD1clip(din)

Op Code DD1_CLIP {0x1D}

Prototype int DD1clip(int *din*);

Description Returns the status of the clip detector.

Return Values:

0 If no clip has occurred

1 If a clip has occurred.

Arguments *din* Device index number.

Example

```
DD1clipon(1);
DD1arm(1);
DD1go(1);
do{ if(DD1clip(1)) DD1stop(1); }
  while(DD1status(1)==ACTIVE);
```

This example will stop DD1_(1) if either the A/D or D/A clip at any time.

(this page intentionally left blank)

AD2 Instrumentation A/D Converter

AD2clear(din)

Op Code AD2_CLEAR {0x1E}

Prototype void AD2clear(int *din*);

Description Clears the specified AD2 and resets it to factory default settings.

Factory Defaults:

- Sampling rate: 1/20µs = 50kHz
- Sample separation: 2.1µs
- # of conversions: 1000
- Sample/hold mode: OFF
- Channel mode: ADC1
- Single triggering.

Arguments *din* Device index number.

Example See example under **AD2go**.

AD2arm(din)

Op Code AD2_ARM {0x13}

Prototype void AD2arm(int *din*);

Description Prepares the specified AD2 for A/D conversion. **AD2arm** must be issued before the AD2 can be triggered (from software or externally) to start conversion.

Arguments *din* Device index number.

Example See example for **AD2go**.

AD2go(din)

Op Code AD2_GO {0x11}

Prototype void AD2go(int *din*);

Description Triggers the specified AD2 from software to begin A/D conversion. **AD2arm** must be issued first to 'prime' the AD2.

Arguments *din* Device index number.

Example

```
AD2clear(1);
AD2arm(1);
AD2go(1);
```

This example will reset AD2_(1) to its default setting and begin conversion.

AD2stop(din)

Op Code AD2_STOP {0x12}

Prototype void AD2stop(int *din*);

Description Forces the specified AD2 to stop A/D conversion immediately.

Arguments *din* Device index number.

Example

```
delay(1000);
AD2stop(1);
```

These lines added to the example above will cause the AD2_(1) to stop after a short delay period.

AD2tgo(din)

Op Code AD2_TGO {0x23}

Prototype void AD2tgo(int *din*);

Description Similar to **AD2go**, but causes the specified AD2 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices. NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
AD2arm(1);
```

```
AD2arm(2);
AD2tgo(1);
AD2tgo(2);
XB1gtrig();          /* Global trigger */
```

This will cause AD2_(1) and AD2_(2) to begin conversion at precisely the same time.

AD2sh(din, shmode)

Op Code AD2_SH {0x20}

Prototype void AD2sh(int *din*, int *shmode*);

Description Enables or disables the individual sample-hold amplifiers on the AD2's four main channels, or on the external multiplexer channels of MX1SH. Disabling the sample-holds results in a slight improvement of signal-to-noise ratio. Enabling the sample-holds results in simultaneous sampling of all four channels.

NOTE: If used, **AD2sh** must *precede* an **AD2mode** or **AD2xchans** command.

Arguments
din Device index number.
shmode Sample-hold mode: OFF-0, ON-1.

AD2mode(din, mcode)

Op Code AD2_MODE {0x14}

Prototype void AD2mode(int *din*, int *mcode*);

Description Selects which of the specified AD2's four channels will be used for conversion. The combination of A/D channels to be used is specified by bits 2-5 of the *mcode* parameter.

Arguments
din Device index number.
mcode Mode code (0x04...0x3C):

ADC1	0x04	A/D channel 1
ADC2	0x08	A/D channel 2
ADC3	0x10	A/D channel 3
ADC4	0x20	A/D channel 4
DUALADC	0x0C	A/D channels 1&2

Other combinations are achieved by directly setting bits 2-5 of *mcode*:

bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
ADC4	ADC3	ADC2	ADC1	0	0

or by adding the above defined constants as in the following example.

Example
```
AD2mode(1,ADC1+ADC3);
```
This call will set up AD2_(1) to use A/D channels 1 and 3.

AD2xchans(din, nchans)

Op Code AD2_XCHANS {0x22}

Prototype void AD2xchans(int *din*, int *nchans*);

Description Instructs the AD2 to use external multiplexer input channels of the MX1 or MX1SH. If used, **AD2xchans** is issued *instead* of **AD2mode**, and the AD2's four front panel BNC inputs cannot be used.

Arguments
din Device index number.
nchans Number of external channels to use (1-128).

Example
```
AD2xchans(1,16);
```

This call will set up AD2_(1) to use 16 multiplexed external channels. Two external multiplexer modules (8 channels each) must be installed in the XBUS.

Note Call this function after **AD2clear** and *before* **mrecord**. Otherwise channels will be shifted.

AD2gain(din, chan, gain)

Op Code AD2_GAIN {0x1F}

Prototype void AD2gain(int *din*, int *chan*, int *gain*);

Description Sets the gain factor which the AD2 will use on a specific input channel. Unless specified with **AD2gain**, channel gains default to x1.

NOTE: If used, **AD2gain** must *follow* an **AD2mode** or **AD2xchans** command.

NOTE: Gain factors x16 through x128 are disabled with factory default. Use of these gains may degrade system performance and is discouraged. These gains can be enabled by singling the jumper on the lower left of the back of AD2.

Arguments

din Device index number.

chan Channel to set gain on (1...128).

gain Channel gain factor (0...7).

x1 - 0	x16 - 4
x2 - 1	x32 - 5
x4 - 2	x64 - 6
x8 - 3	x128 - 7

Example

```
for(i=1,i<5,i++)
  AD2gain(1, i, x8);
```

This will set the gain factor on channels 1-4 of AD2_(1) to x8.

AD2strig(din)

Op Code AD2_STRIG {0x1A}

Prototype void AD2strig(int *din*);

Description Sets up the specified AD2 for single triggering of a conversion buffer. **AD2strig** can be issued while the AD2 is active (see example under **AD2mtrig**).

Arguments *din* Device index number.

Example

```
AD2strig(1);
AD2arm(1);
```

This will cause AD2_(1) to begin converting on an external trigger or **AD2go**, but will not re-trigger unless an **AD2arm** is issued again.

AD2mtrig(din)

Op Code AD2_MTRIG {0x19}

Prototype void AD2mtrig(int *din*);

Description Sets up the specified AD2 for multiple external triggering of a conversion buffer.

NOTE: **AD2strig** can be issued while the AD2 is in multiple external triggering mode to place it back into single triggering mode. This provides a way to stop the AD2 from software after the next buffer conversion is complete without disabling the external triggering source.

Arguments *din* Device index number.

Example

```
AD2mtrig(1);
AD2arm(1);
```

This will cause AD2_(1) to begin converting at each external trigger. The AD2 will not re-trigger until conversion of the present buffer is complete.

AD2reps(din, nreps)

Op Code AD2_REPS {0x1C}

Prototype void AD2reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the AD2 can be re-triggered after an **AD2arm** is issued. Valid only in multiple trigger mode.

Arguments *din* Device index number.

nreps Repeat triggers before rearming (0...60,000).

Example

```
AD2reps(1,100);
```

If this line is added to the example given for **AD2mtrig**, AD2_(1) will respond only to the first 100 triggers received.

AD2srate(din, sper)

Op Code AD2_SRATE {0x15}

Prototype void AD2srate(int *din*, float *sper*);

Description Sets the per-channel sampling rate (in terms of microseconds per sample) of the specified AD2. *Use only when compatibility with older hardware is required (see Note below).* See also **AD2sampsep**.

Arguments
din Device index number.
sper Sampling period (2µs...2000µs).

Example

```
AD2srate(1,20.0);
```

Sets the per-channel sampling rate of AD2_(1) to 1/20µs = 50kHz.

Note AD2 derives the programmed sampling period from a 12.5 MHz base clock. This means that realizable sample periods are quantized to 0.08µs. For example, calling **AD2srate**(1, 7.0) will actually cause the device to sample every 6.96µs, because the rate must be derived from an integer number of cycles of the base clock (0.08µs). Also, the rate will be truncated down to the nearest integer number of 12.5MHz clock cycles, so calling **AD2srate**(1, 5.1) will result in a 5.04µs sampling period not a 5.12µs as might be expected.

With XBDRV versions 2.00 or lower, **AD2srate** will pass the specified sample period to the nearest tenth of a microsecond. So if you make a call such as **AD2srate**(1, 6.98), it will be rounded and sent to the AD2 as 7.0µs, and the AD2 will truncate it down to the nearest integer number of 0.08µs, which is 6.96µs. XBDRV versions 2.01 and higher has provided the **AD2speriod** call to set sample period and return the actual sample period to correct this problem.

AD2speriod(din, sper)

Op Code AD2_SPERIOD {0x24}

Prototype float AD2speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified AD2. It returns the actual sample period.

Arguments *din* Device index number.

sper Sampling period (5µs...2000µs).

Example

```
printf("  %f ", AD2speriod(1,5.00));
```

Outputs 5.04 to the screen and sets the sampling rate of AD2_(1) to 1/5.04µs = 198.413kHz.

Note **AD2speriod** corrects the round-off problem associated with **AD2srate** call, while the base clock limitation still apply (See note under **AD2srate**). **AD2speriod** call is available for AD2 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

AD2sampsep(din, sampsep)

Op Code AD2_SAMPSEP {0x21}

Prototype void AD2sampsep(int *din*, float *sampsep*);

Description Sets the separation time between individual channel samples for the specified AD2.

NOTE: Care must be taken not to set the sampling rate too fast, or the AD2 will not be able to finish sampling all specified channels in time. The period specified by **AD2srate** must satisfy the following:

period >= (# *channels* + 1) · (*sample separation*).

Arguments *din* Device index number.

sampsep Sample separation (2.1µs...1000µs).

Example

```
AD2xchans(1,30);
AD2sampsep(1,5.0);
AD2srate(1,155.0);
```

This call will set up AD2_(1) to use 30 external channels, and set the time between the samples taken on each channel to 5µs. The sampling rate has been selected with a period which allows completion of all 30 channel samples.

AD2npts(din, npts)

Op Code AD2_NPTS {0x18}

Prototype void AD2npts(int *din*, long *npts*);

Description Sets the number of samples the specified AD2 will convert after it is triggered.

Arguments *din* Device index number.

npts Number of conversions (1...2^{32}).

Example

```
AD2npts(1,500000);
```

This call will set up AD2_(1) to perform 500,000 conversions.

AD2clkin(din, scode)

Op Code AD2_CLKIN {0x16}

Prototype void AD2clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified AD2. This is useful for synchronizing AD2 conversions with other devices.

Arguments *din* Device index number.

scode Clock source (1...4):

INTERNAL 1 Internal sample clock

EXTERNAL	2	Front panel BNC
XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line

Example `AD2clkin(1,EXTERNAL);`

This call will set up AD2_(1) to use its front panel BNC for the sampling clock.

AD2clkout(din, dcode)

Op Code AD2_CLKOUT {0x17}

Prototype void AD2clkout(int *din*, int *dcode*);

Description Asserts the AD2's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with AD2 conversions.

Arguments *din* Device index number.

dcode Patch line code (3...5):

XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line
NONE	5	No clock patch

Example

```
AD2clkin(1,INTERNAL);
AD2clkout(1,XCLK1);
```

This call will set up AD2_(1) to use its internal sampling clock and assert it to the XCLK1 line on the XBUS, where it can be used to synchronize other devices on the bus.

AD2status(din)

Op Code AD2_STATUS {0x1B}

Prototype int AD2status(int *din*);

Description Returns the status of the specified AD2.

Return Values:

IDLE	0	AD2 idle, no conversion in progress.
ARM	1	AD2 armed, ready for trigger.
ACTIVE	2	AD2 conversion in progress.

Arguments *din* Device index number.

Example `do{}while(AD2status(1)==ACTIVE);`

This will cause the program to pause while AD2_(1) finishes converting the present sample buffer.

AD2clip(din)

Op Code AD2_CLIP {0x1D}

Prototype int AD2clip(int *din*);

Description Returns the status of the clip detector.

Return Values: number of clips that have occurred.

Arguments *din* Device index number.

Example

```
AD2arm(1);
AD2go(1);
do{ if(AD2clip(1) > 4) AD2stop(1); }
  while(AD2status(1)==ACTIVE);
```

This example will stop AD2_(1) if the A/D clips more than four times.

(this page intentionally left blank)

DA3-2/4/8 Multi-Channel Instrumentation D/A

DA3clear(din)

Op Code DA3_CLEAR {0x1E}

Prototype void DA3clear(int *din*);

Description Clears the specified DA3 and resets it to factory default settings.

Defaults:

Sampling rate:	1/20µs = 50kHz
# of conversions:	1000
Channel mode:	DAC1

Single triggering.

Clip feature disabled.

Slew Correction: Off (no load cap.)

Arguments *din* Device index number.

Example See example for **DA3go**.

DA3arm(din)

Op Code DA3_ARM {0x13}

Prototype void DA3arm(int *din*);

Description Prepares the specified DA3 for D/A conversion. **DA3arm** must be issued before the DA3 can be triggered (from software or externally) to start conversion.

Arguments *din* Device index number.

Example See example for **DA3go**.

DA3go(din)

Op Code DA3_GO {0x11}

Prototype void DA3go(int *din*);

Description Triggers the specified DA3 from software to begin D/A conversion. **DA3arm** must be issued first to 'prime' the DA3.

Arguments *din* Device index number.

Example

```
DA3clear(1);
DA3arm(1);
DA3go(1);
```

This example will reset DA3_(1) to its default setting and begin conversion.

DA3stop(din)

Op Code DA3_STOP {0x12}

Prototype void DA3stop(int *din*);

Description Forces the specified DA3 to stop D/A conversion immediately.

NOTE: **DA3stop** will not zero the DAC outputs—a **DA3clear** must be issued to re-zero, if necessary.

Arguments *din* Device index number.

Example

```
delay(1000);
DA3stop(1);
```

These lines added to the example given for **DA3go** will cause the DA3_(1) to stop after a short delay period.

DA3tgo(din)

Op Code DA3_TGO {0x21}

Prototype void DA3tgo(int *din*);

Description Similar to **DA3go**, but causes the specified DA3 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.

NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example

```
DA3arm(1);
DA3arm(2);
DA3tgo(1);
DA3tgo(2);
XB1gtrig();            /* Global trigger */
```

This will cause DA3_(1) and DA3_(2) to begin conversion at precisely the same time.

DA3mode(din, mcode)

Op Code DA3_MODE {0x14}

Prototype void DA3mode(int *din*, int *mcode*);

Description Selects which of the specified DA3's channels will be used for conversion. The channel mode for D/A conversion is specified by *mcode*.

Arguments *din* Device index number.

mcode Mode code:

DAC1	0x01	D/A channel 1
DAC2	0x02	D/A channel 2
DAC3	0x04	D/A channel 3
DAC4	0x08	D/A channel 4
DAC5	0x10	D/A channel 5
DAC6	0x20	D/A channel 6
DAC7	0x40	D/A channel 7
DAC8	0x80	D/A channel 8
FASTDAC3	0x00	Fast D/A channel 1 only

NOTE: FASTDAC3 mode must be used for single-channel D/A at the maximum 500kHz sampling rate.

When using multiple channels at *aggregate* sampling rates faster than about 200kHz, the D/A channels should be utilized *in sequence*. For example, with 4 channels at 50kHz per channel, DA3 channels 1-4 should be specified.

Example
```
DA3mode(1,DAC1+DAC3);
```
Sets up DA3_(1) to use D/A channels 1 and 3.

DA3strig(din)

Op Code DA3_STRIG {0x1A}

Prototype void DA3strig(int *din*);

Description Sets up the specified DA3 for single triggering of a conversion buffer. **DA3strig** can be issued while the DA3 is active (see example under **DA3mtrig**).

Arguments *din* Device index number.

Example
```
DA3strig(1);
DA3arm(1);
```
This will cause DA3_(1) to begin converting on an external trigger or **DA3go**, but will not re-trigger unless the **DA3arm** is issued again.

DA3mtrig(din)

Op Code DA3_MTRIG {0x19}

Prototype void DA3mtrig(int *din*);

Description Sets up the specified DA3 for multiple external triggering of a conversion buffer.

NOTE: **DA3strig** can be issued while the DA3 is in multiple external triggering mode to place it back into single triggering mode. This provides a way to stop the DA3 from software after the next buffer conversion is complete, without disabling the external triggering source.

Arguments *din* Device index number.

Example

```
DA3mtrig(1);
DA3arm(1);
```

This will cause DA3_(1) to begin converting at each external trigger. The DA3 will not re-trigger until the present conversion cycle is complete.

DA3reps(din, nreps)

Op Code DA3_REPS {0x1C}

Prototype void DA3reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the DA3 can be re-triggered after a **DA3arm** is issued. Valid only in multiple trigger mode.

Arguments
din Device index number.
nreps Repeat triggers before rearming (0...60,000).

Example

```
DA3reps(1,100);
```

If this line is added to the previous example, DA3_(1) will respond only to the first 100 triggers received.

DA3srate(din, sper)

Op Code DA3_SRATE {0x15}

Prototype void DA3srate(int *din*, float *sper*);

Description Sets the per-channel sampling rate of the specified DA3. *Use only when compatibility with older hardware is required (see Note below).*

Arguments
din Device index number.
sper Sampling period (2µs...2000µs).

NOTE: The *aggregate* sampling rate (# of channels x per-channel sampling rate) cannot exceed 500kHz.

Example

```
DA3srate(1,20.0);
```

This call will set the sampling rate of DA3_(1) to 1/20µs = 50kHz.

Note DA3 derives the programmed sampling period from a 12.5 MHz base clock. This means that realizable sample periods are quantized to 0.08µs. For example, calling **DA3srate**(1, 7.0) will actually cause the device to sample every 6.96µs, because the rate must be derived from an integer number of cycles of the base clock (0.08µs). Also, the rate will be truncated down to the nearest integer number of 12.5MHz clock cycles, so calling **DA3srate**(1, 5.1) will result in a 5.04µs sampling period not a 5.12µs as might be expected.

With XBDRV versions 2.00 or lower, **DA3srate** will pass the specified sample period to the nearest tenth of a microsecond. So if you make a call such as **DA3srate**(1, 6.98), it will be rounded and sent to the DA3 as 7.0µs, and the DA3 will truncate it down to the nearest integer number of 0.08µs, which is 6.96µs. XBDRV versions 2.01 and higher has provided the **DA3speriod** call to set sample period and return the actual sample period to correct this problem.

DA3speriod(din, sper)

Op Code DA3_SPERIOD {0x24}

Prototype float DA3speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified DA3. It returns the actual sample period.

Arguments

din Device index number.

sper Sampling period (2µs...2000µs),

NOTE: the aggregate sampling rate (# of channels X per-channel sampling rate) cannot exceed 500 kHz.

Example

```
printf("  %f ", DA3speriod(1,5.00));
```

Outputs 5.04 to the screen and sets the sampling rate of DA3_(1) to 1/5.04µs = 198.413kHz.

Note **DA3speriod** corrects the round-off problem associated with **DA3srate** call, while the base clock limitation still apply (See note under **DA3srate**). **DA3speriod** call is available for DA3 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

DA3npts(din, npts)

Op Code DA3_NPTS {0x18}

Prototype void DA3npts(int *din*, long *npts*);

Description Sets the number of samples the specified DA3 will convert after it is triggered.

Arguments

din Device index number.

npts Number of conversions (1...2^{32}).

Example

```
DA3npts(1,500000);
```

This call will set up DA3_(1) to perform 500,000 conversions, which corresponds to 10 seconds at 50kHz sampling rate.

DA3clkin(din, scode)

Op Code DA3_CLKIN {0x16}

Prototype void DA3clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified DA3. This is useful for synchronizing DA3 conversions with other devices.

Arguments

din Device index number.

scode Clock source (1...4):

INTERNAL	1	Internal sample clock
EXTERNAL	2	Front panel BNC
XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line

Example

```
DA3clkin(1,EXTERNAL);
```

This call will set up DA3_(1) to use its front panel BNC for the sampling clock.

DA3clkout(din, dcode)

Op Code DA3_CLKOUT {0x17}

Prototype void DA3clkout(int *din*, int *dcode*);

Description Asserts the DA3's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with DA3 conversions.

Arguments
din Device index number.
dcode Patch line code (3...5):

XCLK1	3	XBUS patch line
XCLK2	4	XBUS patch line
NONE	5	No clock patch

Example

```
DA3clkin(1,INTERNAL);
DA3clkout(1,XCLK1);
```

This call will set up DA3_(1) to use its internal sampling clock and assert it to the XCLK1 line on the XBUS, where it can be used to synchronize other devices on the bus.

DA3clipon(din)

Op Code DA3_CLIPON {0x1F}

Prototype void DA3clipon(int *din*);

Description Enables the DAC clip detection feature. If the D/A conversion data ever reach their maximum absolute value (±7 FFF), the front-panel clip light will trip and stay on until a **DA3arm** is issued.
NOTE: If this feature is enabled, the maximum sampling rate is limited to 100kHz.

Arguments *din* Device index number.

Example See example under **DA3clip**.

DA3status(din)

Op Code DA3_STATUS {0x1B}

Prototype int DA3status(int *din*);

Description Returns the status of the specified DA3.

Return Values:

IDLE	0	DA3 idle, no conversion in progress.
ARM	1	DA3 armed, ready for trigger.
ACTIVE	2	DA3 conversion in progress.

Arguments *din* Device index number.

Example

```
do{}while(DA3status(1)==ACTIVE);
```

This will cause the program to pause while DA3_(1) finishes converting the present sample buffer.

DA3clip(din)

Op Code DA3_CLIP {0x1D}

Prototype int DA3clip(int *din*);

Description Returns the status of the clip detector.

Return Values:

0 If no clip has occurred

1 If a clip has occurred.

Arguments *din* Device index number.

Example

```
DA3clipon(1);
DA3arm(1);
DA3go(1);
do{ if(DA3clip(1)) DA3stop(1); }
  while(DA3status(1)==ACTIVE);
```

This example will stop DA3_(1) if the D/A clips at any time.

DA3setslew(din, slcode)

Op Code DA3_SETSLEW {0x22}

Prototype void DA3setslew(int *din*, int *slcode*);

Description Sets the amount of slew distortion correction by selecting the load capacitance placed on the D/A converter outputs. The capacitance is selected by the bits in *slcode*.

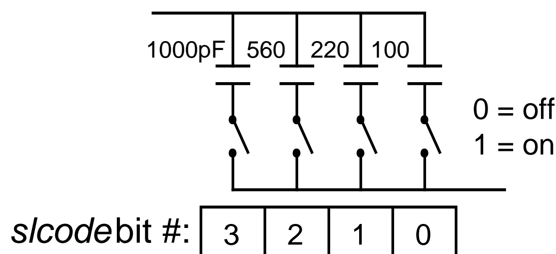
Arguments

din Device index number.

slcode Slew correction code:

AUTOSLEW 0x00 Same as 0x04

The lower four bits in *slcode* select DAC output capacitance as follows:



Call **DA3clear** to turn slew correction off.

Example

```
DA3setslew(1,0x09);
```

Sets an output load capacitance of 1100pF on the DAC outputs of DA3_(1).

DA3zero(din)

Op Code DA3_ZERO {0x23}

Prototype int DA3zero(int *din*);

Description Zeroes all DA3 outputs.

Arguments *din* Device index number.

Example

```
DA3go(1);
do {
   if( kbhit() )
      { DA3stop(1); DA3zero(1); }
}while(DA3status(1)==ACTIVE);
```

This example will stop DA3_(1) and zero all outputs if a key is pressed during D/A conversion.

(this page intentionally left blank)

Auto-Detecting Calls

XBDRV includes a series of procedures that are helpful to those writing applications for use with different TDT modules. The current System II line includes three D/A and three A/D module options. Although each device has its own special features they are programmed using nearly identical XBUS calls.

To make an application program automatically detect and control the installed A/D and D/A module, one might include a conditional block such as the one shown below:

```
/* implements D/A stop command */
if(XB1device(DD1_CODE, 1))
  DD1stop(1);
else
  if(XB1device(DA1_CODE, 1))
    DA1stop(1);
  else
    if(XB1device(DA3_CODE, 1))
      DA3stop(1);
```

Obviously, this implementation would be difficult to manage. To address this problem XBDRV includes a set of auto-detecting calls that implement this functionality using the more efficient *switch* (*case*) statement.

The auto-detecting calls set up two global variables, *dadev* and *addev,* when the *DAclear* and *ADclear* calls are made. These variables are then used in subsequent DA... and AD... calls to select the installed hardware device and to call the appropriate procedure. The following example illustrates this implementation:

```
/* DAstop call */
void DAstop( qint din)
{
  switch(dadev)
  {
    case DD1_CODE:
      DD1stop(din);
      break;

    case DA1_CODE:
      DA1stop(din);
      break;

    case DA3_CODE:
      DA3stop(din);
      break;

case PD1_CODE:
      PD1stop(din);
      break;

    default:
      xcaller = (char *)&"DAstop(dn)";
      showerr("No D/A device IDed.");
  }
}
```

When using the auto-detecting calls, a few points must be noted. First, the *DAclear* and *ADclear* calls are not optional; they must be called so that the *dadev* and *addev* variables will be set properly. Secondly, differences between devices must be handled by the application program. For example, if the DA3's slew reduction is to be enabled when this device is present, the following statements must be included in the application program:

```
if(XB1device( DA3_CODE, 1))
  DA3setslew(1, AUTOSLEW);
```

Another situation that must be handled by the application program is when the DD1 or PD1 is installed. In this case, the D/A and A/D are the same device.

Although the auto-detecting calls add a compatibility advantage to your programs, they also add complexity. TDT does **not** recommend using these calls unless it is essential that your application runs on multiple TDT platforms.

The following is a list of available auto-detecting calls. For more detailed information on specifications, examples, and notes, please refer to this document for discussions on device-specific driver calls. Note that calls not common to all D/A or A/D devices are omitted.

D/A Calls:	A/D Calls:
DAclear(din)	**ADclear(din)**
DAarm(din)	**ADarm(din)**
DAgo(din)	**ADgo(din)**
DAstop(din)	**ADstop(din)**
DAtgo(din)	**ADtgo(din)**
DAmode(din, mcode)	**ADmode(din, mcode)**
DAstrig(din)	**ADstrig(din)**
DAmtrig(din)	**ADmtrig(din)**
DAreps(din, nreps)	**ADreps(din, nreps)**
DAsrate(din, sper)	**ADsrate(din, sper)**
DAsperiod(din, sper)	**ADsperiod(din, sper)**
DAnpts(din, npts)	**ADnpts(din, npts)**
DAclkin(din, scode)	**ADclkin(din, scode)**
DAclkout(din, dcode)	**ADclkout(din, dcode)**
DAstatus(din)	**ADstatus(din)**

The following are issues to keep in mind when using the auto-detecting calls:

• DAs	The D/A and A/D modules do not share all of the same mode codes. For example, an error will result if a *DAmode(1, DAC4)* call is made to a system having DA1 module.
• DA3	**DA zero** is not available
• DA3	Uses the FASTDAC3 constant instead of FASTDAC
• DD1, PD1	Call either **DAmode** or **ADmode**, but not both. Include both the A/D and D/A constants in the function call. e.g.: `DAmode(1, ADC2+DAC1);`
• PD1	You cannot use A/D or D/A channels 3 and 4 with the **DAmode** (or **PD1mode**) functions.
• PD1	The PD1 does not have FASTDAC mode.

SS1 Programmable Signal Switcher

Overview

The SS1 is a versatile signal switcher utilizing a dedicated microprocessor and solid state switches to provide bi-directional, multi-channel switching. The SS1 can be configured as a quad 2-to-1, dual 4-to-1, or single 8-to-1 switch. A 10 dB gain also can be applied at inputs 1, 2, 5, and 6.

SS1clear(din)

Op Code SS1_CLEAR {0x11}

Prototype void SS1clear(int *din*);

Description Clears the current switching mode and sets the SS1 to the following factory default configuration:

- All switches off.
- 0dB gain on all channels.
- Quad 2-to-1 switching mode.

Arguments *din* Device index number.

Example

```
SS1clear(1);
```

Resets SS1_(1) to factory default mode (quad 2-to-1 mode, no gain, all channels are off).

SS1gainon(din)

Op Code SS1_GAINON {0x12}

Prototype void SS1gainon(int *din*);

Description Turns on the 10 dB gain at inputs 1, 2, 5, and 6. NOTE: The SS1 is a bi-directional device; however, 10 dB gain is valid only when signal is applied at left 8 channels. These are marked as INPUT channels.

WARNING: When selecting 10 dB gain, do not exceed ±10V peak input/output level. Otherwise, unpredectable operation will be resulted.

Arguments *din* Device index number.

Example
```
SS1gainon(3);
```
This will turn on the gain of the SS1 having *din*=3.

SS1gainoff(din)

Op Code SS1_GAINOFF {0x13}

Prototype void SS1gainoff(int *din*);

Description Turns off the 10 dB gain at inputs 1, 2, 5 and 6. See **SS1gainon**.

Arguments *din* Device index number.

Example
```
SS1gainoff(5);
```
This will turn off the gain of the SS1 having *din*=5.

SS1mode(din, mcode)

Op Code SS1_MODE {0x14}

Prototype void SS1mode(int *din* , int *mcode*);

Description Sets the SS1's switching mode.

Arguments *din* Device index number.

mcode Device switching mode, pre-defined as:

QUAD_2_1 0
DUAL_4_1 1
SING_8_1 2

Example
```
SS1mode(1, DUAL_4_1);
```
This sets the SS1 as a dual 4-to-1 (or 1-to-4) switch.

SS1select(din, chan, inpn)

Op Code SS1_SELECT {0x15}

Prototype void SS1select(int *din* , int *chan*, int *inpn*);

Description Connects the specified input channel (*inpn*) to the specified output channel (*chan*).

Arguments

din Device index number.

chan Device channel number (0 through 3, marked as A through B).

inpn Device input number (In quad 2-to-1 mode, *inpn* is 1 or 2; in dual 4-to-1 mode, *inpn* can be 1 through 4; in single 8-to-1 mode, *inpn* can be 1 through 8).

Example

```
SS1clear(1);
SS1gainon(1);
SS1mode(1, DUAL_4_1);
SSlselect(1, 2, 1);
```

This will clear the SS1 and reset it to factory default; turn on the 10 dB gain at inputs 1, 2, 5, and 6; set the SS1 as a dual 4-to-1 switch (inputs 5 and 6 are addressed as inputs 1 and 2); and connect input 1 (marked as input 5) to output 2 (marked as C).

NOTE: The eight inputs are marked as 1 through 8, as in signal 8-to-1 mode. While in dual 4-to-1 mode, the actual inputs numbers are 1, 2, 3 and 4 (inputs 5 through 8 should be addressed as inputs 1 through 4). Similarly, in quad 2-to-1 mode, input numbers are 1 and 2. That is, inputs 3 and 4, 5 and 6, and 7 and 8 should be addressed as inputs 1 and 2, respectively.

The four outputs (marked as A through D) are always addressed as 0 through 3. In dual 4-to-1 mode, only 0 and 2 can be addressed, since outputs 1 and 3 are disabled (their addresses are 0 or 2). Similarly, only output 0 can be addressed in single 8-to-1 mode, since outputs 1 through 3 are disabled (connected to output 0).

(this page intentionally left blank)

PM1 Power Multiplexer

Note: Since PM1 connection is critical to its performance, please refer to the PM1 specification sheet for correct connection.

PM1clear(din)

Op Code PM1_CLEAR {0x11}

Prototype void PM1clear(int *din*);

Description Clears the PM1 to its factory default settings:
- All speakers off
- Exclusive mode
- Mono configuration

Arguments *din* Device index number.

Example
```
PM1clear(3);
```
Resets PM1_(3) to factory default mode (see above).

PM1config(din, ccode)

Op Code PM1_CONFIG {0x12}

Prototype void PM1config(int *din* , int *ccode*);

Description Sets the PM1 configuration to either mono or stereo. This call has no affect when device is running in common switching mode (see below). Also, the manual MONO/STEREO switch should be set to the desired operating mode.

Arguments *din* Device index number.
ccode Device configuration code, use:
PM1_MONO 1
PM1_STEREO 0

Example see below.

PM1mode(din, mcode)

Op Code PM1_MODE {0x13}

Prototype void PM1mode(int *din* , int *mcode*);

Description Programs the PM1 to run in either Exclusive or Common mode. In common mode, any number of speakers can be switched on at the same time. In exclusive mode only one speaker per channel will be allowed on at any given time. The PM1 will automatically turn off other speakers before turning on the specified speaker.

Warning: damage to amplifiers etc. may result if more than one speaker (load) is placed in parallel across its outputs. Refer to your amplifier documentation for pertinent information.

Arguments *din* Device index number.

mcode Device mode code, use:

EXCLUSIVE = 1

COMMON = 0

Example

```
PM1clear(1);
PM1config(1, PM1_STEREO);
PM1mode(1, COMMON);
```

This set of calls will setup the PM1 for two channel operation in common mode.

PM1spkon(din, sn)

Op Code PM1_SPKON {0x14}

Prototype void PM1spkon(int *din* , int *sn*);

Description Turns the specified speaker on.

Arguments *din* Device index number.

sn Speaker number (1..16). The following constants can also be used to specify a PM1 speaker:

SN1, SN2... SN16 for speakers 1..16.

CHAN_A = 0

CHAN_B = 8

Example

```
PM1clear(1);
PM1config(1, PM1_STEREO);
```

```
PM1spkon(1, CHAN_A+SN3);
PM1spkon(1, CHAN_B+SN7);
```

Turns on the 3rd speaker of Channel-A and the 7th speaker (17) of Channel-B. Note: the device is programmed to operate in STEREO mode.

PM1spkoff(din, sn)

Op Code PM1_SPKOFF {0x15}

Prototype void PM1spkoff(int *din* , int *sn*);

Description Turns the specified speaker off.

Arguments

din Device index number.

sn Speaker number (1 ... 16). See above.

Example

```
PM1spkoff(1, 3);
PM1spkoff(1, 15);
```

Turns off the speakers turned on in the above example.

(this page intentionally left blank)

PD1 Power SDAC

Overview

Unlike other XBUS devices, the PD1 not only contains a controlling DSP processor, but it also includes a secondary controlling processor that handles the PD1 auto-routing function. This secondary processor is responsible for interfacing with the PD1 and controlling all PD1 resources. The PD1's secondary processor and its associated circuits are collectively referred to as the ***Real Time Router*** or **RTR**.

Because there are numerous XBUS calls required to control the PD1, they have been broken into logical groups based on their function and/or the PD1 resource they control. These groups and a brief description are listed below:

1. **PD1 Basic Calls** -- This group includes the PD1 calls that function in a similar manner to other D/A -- A/D calls. These 'overlapping' calls include: PD1clear, PD1mode, PD1srate and PD1arm. Basic Calls provide the PD1 with D/A -- A/D functionality and make it compatible with other TDT converter modules.

2. **Route Scheduling Calls** -- Route schedules are specified using this group of calls. Included are calls for scheduling simple routes and multi-routes. Also included are calls specifying hard-wired connections for Inbound and Outbound streams.

3. **Convolver DSP Calls** -- These calls send commands to the PD1's complex of DSP processors.

4. **Analog I/O Calls** -- The PD1 is typically equipped with 2 or 4 channels of A/D and D/A. The calls in this group are used to initialize and configure these hardware resources.

5. **Delay Processor Calls** -- The PD1 delay processor (optional) is controlled with this group of calls.

6. **Utility Calls** -- These functions are used by other PD1 calls and should NOT be called directly in user source code.

PD1 Referencing Variables

XBDRV includes a number of constants (macros) to be used when programming the PD1. These constants are used to bracket and identify data records placed on data streams being sent to the PD1 from the AP2. The defined constants include: MONO, STEREO, _START, _STOP, LSYNC and GSYNC. Refer to *PD1 User's Guide* for more information on the use of these constants.

Also included in XBDRV are a number of global variables used to identify various PD1 resources and their ports. Variable arrays are used instead of constants (macros) to allow for indexed referencing of various PD1 resources. For example, to program the first eight delay taps of delay channel-0 to 0, 10, 20... 70 millisecond delays, respectively, a for-next loop can be used:

```
PD1srate(1, 20.0);
PD1clrDEL(1, 8, 0, 0, 0);
for(i=0; i<8; i++)
  PD1setDEL(1, TAP[i][0], i * 10);
```

Note that resource variables and thus resources and their ports are always numbered starting from zero instead of one. At first this may seem awkward, however, after writing a few code loops such as the one shown above, the advantages of zero-based referencing become apparent. The following variables are included in XBDRV's PD1 section:

DSPid[0..27]	Convolving DSP selector
DSPin[0..27]	DSP mono input port
DSPinL[0..27]	DSP left channel input port
DSPinR[0..27]	DSP right channel input port
DSPout[0..27]	DSP mono output port
DSPoutL[0..27]	DSP left channel output port
DSPoutR[0..27]	DSP right channel output port
COEF[0..27]	DSP coefficient input port
DELin[0..3]	Delay Processor input port
DELout[0..31][0..3]	Delay tap output port (0..31 selects tap)
TAP[0..31][0..3]	Delay tap specification input port
DAC[0..3]	D/A Converter input port
ADC[0..3]	A/D Converter output port
IB[0..15]	Inbound data stream
OB[0..15]	Outbound data stream

The variables shown above are initialized when **PD1clear** is called. These variables will NOT be valid until **PD1clear** has been called.

PD1 Basic Calls

The PD1 Power SDAC can be programmed and operated much like other TDT D/A -- A/D converters. The PD1 XBUS drivers described here include a PD1mode call to make the PD1 compatible with the DD1, thus allowing the use of standard DA and AD auto-detecting calls.

PD1clear(din)

Op Code	PD1_CLEAR {0x1E}
Prototype	void PD1clear(int *din*);
Description	Clears the specified PD1 and resets it to factory default settings. Also resets the RTR, Analog Interface and Router Schedule by calling: **PD1resetRTE, PD1clrIO and PD1clrsched**. Refer to specifications on these calls for more information.
Arguments	*din* Device index number.
Defaults	- Sampling rate: 1/20µs = 50kHz - # of conversions: 1000 - Single triggering. * Unlike other D/A--A/D devices PD1mode must be called to enable a particular configuration of DAC and ADC channels. See **PD1mode.**
Example	See **PD1mode**.

PD1arm(din)

Op Code PD1_ARM {0x13}

Prototype void PD1arm(int *din*);

Description Prepares the specified PD1 for A/D - D/A conversion. **PD1arm** must be issued before the PD1 can be triggered (from software or externally) to start conversion. **PD1arm** also calls **PD1flushRTE**. Refer to this call's specification for more information.

Arguments *din* Device index number.

Example See **PD1mode**.

PD1go(din)

Op Code PD1_GO {0x11}

Prototype void PD1go(int *din*);

Description Triggers the specified PD1 from software to begin A/D - D/A conversion. **PD1arm** must be issued first to 'prime' the PD1.

Arguments *din* Device index number.

Example See **PD1mode**.

PD1stop(din)

Op Code PD1_STOP {0x12}

Prototype void PD1stop(int *din*);

Description Forces the specified PD1 to stop A/D - D/A conversion immediately.
NOTE: **PD1stop** will not zero the DAC outputs, a **PD1clear** or **PD1clrIO** must be issued to zero the outputs.

Arguments *din* Device index number.

Example

```
PD1stop(1)
```

This call immediately halts all A/D and D.A conversion on PD1_(1).

PD1mode(din, mcode)

Op Code NA

Prototype void PD1mode(int *din*, int *mcode*);

Description This call has been added to make the PD1 compatible with other TDT D/A -- A/D devices.

PD1mode assumes your PD1 has two DAC and two ADC channels, similar to the DD1. PD1mode creates the routing schedule necessary to use the specified channels. For example, calling *PD1mode (1, DAC2 + DUALADC)* will result in the following commands being sent to the PD1:

```
PD1clrsched(1);
PD1nstrms(1, 1, 2);
PD1specIB(1, IB[0], DAC[1]);
PD1specOB(1, OB[0], ADC[0]);
PD1specOB(1, OB[1], ADC[1]);
```

Arguments

din Device index number.

mcode Any combination of the DAC1/2 and ADC1/2 constants (see **DD1mode** for a description)

Note You must use the PD1 function calls (nstrms, specIB, specOB, etc.) to make use of the third and fourth channels.
FASTDAC mode is not available on the PD1.

Example PD1clear(1);

```
PD1mode(1, DAC1);
{ required AP2 play call }
PD1arm(1);
PD1go(1);
```

This example will reset PD1_(1) to its default setting and begin conversion from DAC channel 1.

PD1nstrms(din, nIB, nOB)

Op Code PD1_NSTRMS {0x14}

Prototype void PD1nstrms(int *din*, int *nIB*, int *nOB*);

Description The PD1nstrms call is used to tell the PD1 how many Inbound (AP2==>PD1) and Outbound (PD1==>AP2) data streams it should process. Be careful to match the number specified here with the play and record specifications made on the AP2. For example, if two Inbound streams are specified with PD1nstrms then *dplay* should be used setup the AP2 to feed the proper number of data streams. Note, this procedure also calls **PD1nstrmsRTE**.

Arguments

din Device index number.

nIB Number of Inbound data streams (0..16)

nOB Number of Outbound data streams (0..16)

Example

```
PD1nstrms(1, 2, 0);
```

This call will program the PD1 to process two channels of Inbound data (D/A data) and no Outbound data (A/D data).

PD1tgo(din)

Op Code PD1_TGO {0x21}

Prototype void PD1tgo(int *din*);

Description Similar to **PD1go**, but causes the specified PD1 to wait for a global or local XBUS trigger before taking action (see **XB1gtrig()** and **XB1ltrig()**). Useful for synchronized triggering of multiple XBUS devices.
NOTE: Global and local triggering are available only on XBUS hardware versions 3.0 or higher.

Arguments *din* Device index number.

Example See **DD1tgo**.

PD1strig(din)

Op Code PD1_STRIG {0x1A}

Prototype void PD1strig(int *din*);

Description Sets up the specified PD1 for single triggering of a conversion buffer. **PD1strig** can be issued while the PD1 is active (see example under **PD1mtrig**).

Arguments *din* Device index number.

Example See **DD1strig**.

PD1mtrig(din)

Op Code PD1_MTRIG {0x19}

Prototype void PD1mtrig(int *din*);

Description Sets up the specified PD1 for multiple external triggering of a conversion buffer.

NOTE: **PD1strig** can be issued while the PD1 is in multiple triggering mode to place it back into single triggering mode. This provides a way to stop the PD1 from software after the next buffer conversion is complete without disabling the external triggering source.

Arguments *din* Device index number.

Example See **DD1tmtrig**.

PD1reps(din, nreps)

Op Code PD1_REPS {0x1C}

Prototype void PD1reps(int *din*, unsigned int *nreps*);

Description Sets the number of times the PD1 can be re-triggered after a **PD1arm** is issued. Valid only in multiple trigger mode.

Arguments *din* Device index number.

nreps Repeat triggers before re-arming (0...60,000)

Example See **DD1reps**.

PD1srate(din, sper)

Op Code PD1_SRATE {0x15}

Prototype void PD1srate(int *din*, float *sper*);

Description Sets the sampling rate of the specified PD1. *Use only when compatibility with older hardware is required (see Note under DD1srate).*

Arguments

din	Device index number.
sper	Sampling period (5µs...2000µs) or (6.5µs...2000µs) when A/D is used.

Example

```
PD1srate(1,20.00);
```

Sets the sampling rate of PD1_(1) to 1/20µs = 50kHz.

PD1speriod(din, sper)

Op Code PD1_SPERIOD {0x26}

Prototype float PD1speriod(int *din*, float *sper*);

Description Sets the sampling period (microseconds per sample) of the specified PD1. It returns the actual sample period.

Arguments

din	Device index number.
sper	Sampling period (5µs...2000µs) or (6.5µs...2000µs) when A/D is used.

Example

```
printf("  %f ", PD1speriod(1,5.00));
```

Outputs 5.04 to the screen and sets the sampling rate of PD1_(1) to 1/5.04µs = 198.413kHz.

Note **PD1speriod** corrects the round-off problem associated with **PD1srate** call, while the base clock limitation still apply (See note under **PD1srate**). **PD1speriod** call is available for PD1 of hardware versions 5.0 and above, and XBDRV versions 2.01 and above.

PD1npts(din, npts)

Op Code PD1_NPTS {0x18}

Prototype void PD1npts(int *din*, long *npts*);

Description Sets the number of samples the specified PD1 will convert after it is triggered.

Arguments
din Device index number.
npts Number of conversions ($1...2^{32}$).

Example

```
PD1npts(1,-1);
```

This call tells the PD1 to convert the maximum number of points possible. This maximum number corresponds to about 24 hours at 50KHz.

PD1clkin(din, scode)

Op Code PD1_CLKIN {0x16}

Prototype void PD1clkin(int *din*, int *scode*);

Description Selects the source of the sampling clock for the specified PD1. This is useful for synchronizing PD1 conversions with other devices.

Arguments
din Device index number.
scode Clock source (1...4):
See **DD1clkin**.

Example See **DD1clkin**.

PD1clkout(din, dcode)

Op Code PD1_CLKOUT {0x17}

Prototype void PD1clkout(int *din*, int *dcode*);

Description Asserts the PD1's internal sampling clock onto an XBUS internal patch line. This is useful for synchronizing other devices with PD1 conversions.

Arguments
din Device index number.
dcode Patch line code (3...5):
See **DD1clkin**.

Example See **DD1clkout**.

PD1status(din)

Op Code PD1_STATUS {0x1B}

Prototype int PD1status(int *din*);

Description Returns the status of the specified PD1.

Return Values:

IDLE	0	PD1 idle, no conversion in progress.
ARM	1	PD1 armed, ready for trigger.
ACTIVE	2	PD1 conversion in progress.

Arguments *din* Device index number.

Example See **DD1status**.

PD1syncall(din)

Op Code RTE_SYNCALL {0x0110}

Prototype void PD1syncall(int *din*);

Description Sends a global sync command to all PD1 resources causing them to latch any previously loaded filter coefficients and/or delay constants. Calling PD1syncall is equivalent to sending the value GSYNC to the PD1 on a data stream.

Arguments *din* Device index number.

Example

```
PD1srate(1, 20.0);
PD1clrDEL(1, 3, 0, 0, 0);
PD1setDEL(1, TAP[0][0], 50);
PD1selDEL(1, TAP[0][1], 100);
PD1selDEL(1, TAP[0][2], 150);
{possibly load some filter coefs}
PD1syncall(1);
```

The code shown above will program channel-0 of the delay processor to generate 1ms, 2ms and 3ms delays from its first three tap outputs. It then alludes to filter coef loading. The PD1syncall call is then used to synchronously latch all parameters.

Route Scheduling Calls

The PD1's Real Time Router (RTR) is programmed to run a specified routing schedule on each tick of the sampling clock. The routing schedule is programmed using the following set of XBDRV calls. Route order within each routing schedule is determined by simple routing rules. Refer to *PD1 User's Guide* for more information on these rules.

PD1clrsched(din)

Op Code	RTE_CLRSCHED {0x0106}	
Prototype	void	PD1clrsched(int *din*);
Description	Clears the current routing schedule and readies the PD1's RTR for receiving routing calls. It is important that all PD1 route scheduling calls be made immediately following the **PD1clrsched** call. Do NOT intermix calls in this group with other PD1 calls.	
Arguments	*din*	Device index number.
Example	See **PD1addsimp**.	

PD1addsimp(din, src, des)

Op Code RTE_ADDSIMP {0x0107}

Prototype void PD1addsimp(int *din*, int *src*, int *des*);

Description Adds a simple route to the current route schedule. Simple routes are used to interconnect two PD1 resources when there is a single source (no mixing). Be careful to follow ordering rules when scheduling routes involving DSPs (see *PD1 User's Guide*).

Arguments

din	Device index number.
src	Route source port. Use: DSPout*, DELout or ADC.
des	Route destination port. Use: DSPin*, COEF, DELin, TAP or DAC.

Example

```
PD1clrsched(1);
PD1addsimp(1, DSPout[0], DAC[0]);
PD1addsimp(1, ADC[0], DSPin[0]);
```

These three calls will program a routing schedule that will pass the signal from ADC[0] through DSP[0] and out DAC[0]. Note the ordering of the scheduling calls whereby the DSP is read before it is written. Refer to *PD1 User's Guide* for more information on rules governing route scheduling.

PD1addmult(din, srclst[], sf[], nsrcs, des)

Op Code RTE_ADDMULT {0x0108}

Prototype void PD1addmult(int *din*, int *srclst[]*, float *sf[]*, int *nsrcs*, int *des*);

Description Adds a multi-route (mixing route) to the current route schedule. Multi-routes are used to connect multiple output ports to a single input port. Multi-routes support individual scaling for each input.

Arguments

din Device index number.

srclst[] Integer array containing list of source ports.

sf[] Floating point array containing a list of source scale factors (1.0 <= sf <= -0.99976).

nsrcs Number of sources in srclst (1..13).

des Destination port.

Example

```
int srclstL[8];
int srclstR[8];
float sf[8];

PD1clrsched(1);
for(i=0; i<8; i++)
{
  srclstL[i] = DSPoutL[i];
  srclstR[i] = DSPoutR[i];
  sf[i] = 0.125;
}
PD1addmult(1, srclstL, sf, 8, DAC[0]);
PD1addmult(1, srclstR, sf, 8, DAC[1]);
```

The program segment shown above will program the PD1 RTR to run two multi-routes. The first will sum the left channel outputs from DSPs 0 through 7 and send the result to DAC[0]'s input. The second multi-route will sum the right channel outputs from the same DSPs and send it to the input of DAC[1]. All inputs will be scaled by 0.125 or 1/8.

PD1specIB(din, ibn, des)

Op Code RTE_SPECIB {0x0109}

Prototype void PD1specIB(int *din,* int *ibn,* int *des*);

Description Use this call to specify the destination resource port for an Inbound data stream (AP2 ==> PD1). If the stream will be dynamically assigned there is no need to call this procedure.

Arguments

din Device index number.

ibn Inbound stream number. Use: IB[0..15].

des Destination port.

Example

```
PD1clear(1);
PD1nstrms(1, 1, 0);
PD1clrsched(1);
PD1specIB(1, IB[0], DAC[1]);
```

This program segment will program the PD1 as a single channel D/A converter. Data will be converted from DAC[1] (second D/A channel).

PD1specOB(din, obn, src)

Op Code RTE_SPECOB {0x0112}

Prototype void PD1specOB(int *din,* int *obn,* int *src*);

Description Use this call to specify the source resource port for an Outbound data stream (PD1 ==> AP2). Outbound streams cannot be dynamically assigned and therefore must be specified using PD1specOB. Refer to *PD1 User's Guide* for rules on specifying Outbound steams.

Arguments

din Device index number.

ibn Inbound stream number. Use: IB[0..15].

des Destination port.

Example

```
PD1clear(1);
PD1nstrms(1, 0, 2);
PD1clrsched(1);
PD1specIB(1, OB[0], ADC[0]);
PD1specIB(1, OB[1], ADC[1]);
```

This program segment will program the PD1 as a two channel A/D converter.

DSP Calls

When running in convolver mode, DSPs are programmed primarily through data records sent to their coefficient inputs. A DSP is reset to this basic mode with the **PD1resetDSP** call (see below). Each DSP can be made to run in either bypassed or no-pass mode using the **PD1bypass** and **PD1idle** calls explained below. The following calls make extensive use of a bit mask to specify DSPs within the PD1. In a specification bit mask, bit position zero (LSBit) corresponds to DSP[0] and bit-1 to DSP[1], etc. Specifying 1 in a mask bit position sends the command to the appropriate DSP.

PD1idleDSP(din, dmask)

Op Code RTE_IDLEDSP {0x0116}

Prototype void PD1idleDSP(int *din*, long *dmask*);

Description Places the specified DSP(s) in IDLE mode. In this mode, output ports will always read zero and all input ports are ignored.

Arguments
din Device index number.
dmask DSP specification mask (see above).

Example
```
PD1idleDSP(1, 0x3);
```
This call will place DSPs 0 and 1 in idle mode.

PD1bypassDSP(din, dmask)

Op Code RTE_BYPASSDSP {0x010c}

Prototype void PD1bypassDSP(int *din*, long *dmask*);

Description Places the specified DSP(s) in BYPASS mode. In this mode, DSPinL ==> DSPoutL and DSPinR ==> DSPoutR. The COEF input port will be ignored.

Arguments
din Device index number.
dmask DSP specification mask (see above).

Example
```
PD1bypassDSP(1, 0x30);
```
This call will place DSPs 4 and 5 in bypass mode.

PD1resetDSP(din, dmask)

Op Code RTE_RESETDSP {0x010b}

Prototype void PD1resetDSP(int *din*, long *dmask*);

Description Resets the specified DSP(s) into basic convolution mode. In this mode, the each DSP can be controlled by data records sent to its COEF input port. After being properly loaded with coefficients, DSPs running in this mode will convolve data at their input ports with coefficient data and write the result to the output ports.

Arguments *din* Device index number.
dmask DSP specification mask (see above).

Example
```
PD1resetDSP(1, 0x0fffffff);
```
Resets all DSPs installed in system.

PD1lockDSP(din, dmask)

Op Code RTE_LOCKDSP {}

Prototype void PD1lockDSP(int *din*, long *dmask*);

Description Locks the specified DSP so that it will not respond to global sync commands.

Arguments *din* Device index number.
dmask DSP specification mask (see above).

Example
```
PD1lockDSP(1, 0x3);
```
This call will lock DSPs 0 and 1.

PD1interpDSP(din, ifact, dmask)

Op Code RTE_INTERPDSP {}

Prototype void PD1lockDSP(int *din*, int *ifact*, long *dmask*);

Description Turns interpolation on. Interpolation is used to provide a smooth transition from one set of filter coefficients to another.

Arguments

din	Device index number.
ifact	Interpolation factor (usu 500-1000).
dmask	DSP specification mask (see above).

Example

```
PD1interpDSP(1, 1000, 0x1)
```

Turns on coefficient interpolation for DSP[0] and sets the interpolation factor to 1000.

PD1checkDSPS(din)

Op Code PD1_CHECKDSP {0x24}

Prototype long PD1checkDSPS(int *din*);

Description Returns a long bit mask specifying all functioning DSPs installed in the PD1. You MUST call **PD1bypassDSP**(1, 0x0FFFFFFF) before calling **PD1checkDSPS** or the test will fail for any non-bypassed DSPs.

Arguments

din	Device index number.

Example

```
dmask = PD1checkDSPS(1);
printf ("Functioning DSPs:%X%X",
   (int)(dmask >> 16), (int)(dmask & 0xffff));
```

In a PD1 system with six convolver blocks (12 DSPs) the above code will generate the following output:

Functioning DSPs: 00FFF

PD1whatDSP(din, dspid)

Op Code RTE_WHATDSP {0x01115}

Prototype qint PD1whatDSP(int *din*, int *dspid*);

Description This call returns micro-code version and device identification information for the specified DSP. Refer to *PD1 User's Guide* for more information on decoding the returned value.

Arguments

din Device index number.

dspid DSP selection ID. Use: DSPid[0..27].

Example See *PD1 User's Guide*.

Analog I/O Calls

The PD1 can be configured with a variety of D/A--A/D options. At a minimum, a typical PD1 will be equipped with two channels of DAC and two channels of ADC. The installed analog interface can be programmed using the following calls.

PD1clrIO(din)

Op Code RTE_CLRIO {0x0117}

Prototype void PD1clrIO(int *din*);

Description Clears all DAC outputs to zero volts and clears the digital portion of the analog interface back to zero.

Arguments *din* Device index number.

Example

```
PD1stop(1);
PD1clrIO(1);
```

This code is typical of how the PD1 can be stopped asynchronously and then its DAC outputs forced to zero volts.

PD1setIO(din, dt1, dt2, at1, at2)

Op Code RTE_SETIO {0x0117}

Prototype void PD1clrIO(qint *din,* float *dt1,* float *dt2,* float *at1,* float *at2*);

Description Sets up LED indicator thresholds for the DAC and ADC channels. By default the yellow LEDS will light for signals greater than 0.04 volts and the red LEDS will light when voltages are greater than 5.0 volts. These defaults can be overridden using PD1setIO.

Arguments

din Device index number.
dt1 Yellow LED threshold for DACs (0..10.0).
dt2 Red LED threshold for DACs (0..10.0).
at1 Yellow LED threshold for ADCs (0..10.0).
at2 Red LED threshold for ADCs (0..10.0).

Example

```
PD1setIO(1, 0.01, 9.99, 0.01, 9.99);
```

This above call will set the red LED thresholds to 9.99 volts and the yellow LED thresholds to 0.01 volts.

PD1whatIO(din)

Op Code RTE_WHATIO {0x01114}

Prototype qint PD1whatIO(int *din*);

Description This call returns micro-code version and device information for the installed analog I/O hardware. Refer to *PD1 User's Guide* for more information on decoding the returned value.

Arguments *din* Device index number.

Example See *PD1 User's Guide*.

Delay Processor Calls

The PD1 is optionally equipped with the DP2 delay processor. This four channel delay is capable of generating multiple, arbitrary delay taps from each of four delay channels. The following calls are used to program the delay processor hardware. Refer to *PD1 User's Guide* for more information on using the DP2.

PD1clrDEL(din, n0, n1, n2, n3)

Op Code RTE_CLRDEL {0x0103}

Prototype void PD1clrDEL(int *din*, int *n0*, int *n1*, int *n2*, int *n3*);

Description Clears the delay processor and specifies the number of taps to be processed on each channel. PD1clrDEL will flush the delay history (see **PD1flushDEL**), clear all delay time specifications to zero and turn off interpolation (see below). In addition, the n0..n3 arguments are used to indicate the number of delay taps each channel should process. Refer to *PD1 User's Guide* for limitations associated with delay tap processing.

Arguments

din Device index number.

n0,n1,n2,n3 Processing specifiers for each delay channel 0 through 3.

Example

```
PD1clrDEL(1, 13, 8, 0, 0);
```

This call will clear the delay processor and tell it to process delay taps 0..12 on channel 0 and 0..7 on channel 1.

PD1flushDEL(din)

Op Code RTE_FLUSHDEL {0x010f}

Prototype void PD1flushDEL(int *din*);

Description Clears all delay history buffers to zero. Should be used each time the PD1 is rearmed for operation using the delay processor.

Arguments *din* Device index number.

Example None.

PD1interpDEL(din, ifact)

Op Code RTE_INTERPDEL {0x0111}

Prototype void PD1interpDEL(int *din*, int *ifact*);

Description Calling **PD1interpDEL** enables the delay interpolator and sets the interpolation factor. Refer to *PD1 User's Guide* for more information on the DP2's delay interpolator.

Arguments *din* Device index number.

ifact Interpolation factor (1..32767).

ifact = 32767/(number-of-samples).

Example

```
PD1interpDEL(1, (int)(32767.0 / 100.0));
```

The above call will enable the DP2 delay interpolator and set the delay interpolation speed to approximately 100 samples.

PD1setDEL(din, tapn, dly)

Op Code RTE_SETDEL {0x0104}

Prototype void PD1setDEL(int *din*, int *tapn*, int *dly*);

Description This procedure is used to specify the amount of delay to be used for the indicated delay tap output.

Arguments

din Device index number.

tapn Delay tap specifier. Use: TAP[0..31][0..3].

dly Number of samples to delay (0..32767)

Example

```
{ Assume a sample rate of 50KHz or 50 samples
  per millisecond }
PD1clrDEL(1, 0, 0, 4, 0);
PD1setDEL(1, TAP[0][2], 50 * 0);
PD1setDEL(1, TAP[1][2], 50 * 12);
PD1setDEL(1, TAP[2][2], 50 * 24);
PD1setDEL(1, TAP[3][2], 50 * 36);
PD1latchDEL(1);
```

The code listed above will clear the DP2 and set the first four delay taps of channel-2 to 0, 12, 24 and 36 milliseconds. See below for explanation of **PD1latchDEL**.

PD1latchDEL(din)

Op Code RTE_LATCHDEL {0x010e}

Prototype void PD1latchDEL(int *din*);

Description This call is used to latch any previously specified delay times into the DP2. Refer to *PD1 User's Guide* for more information.

Arguments

din Device index number.

Example See previous example

PD1whatDEL(din)

Op Code	RTE_WHATDEL {0x0113}	
Prototype	qint	PD1whatDEL(int *din*);
Description	This call returns micro-code version and device information for the installed delay hardware. Refer to *PD1 User's Guide* for more information on decoding the returned value.	
Arguments	*din*	Device index number.
Example	See *PD1 User's Guide*.	

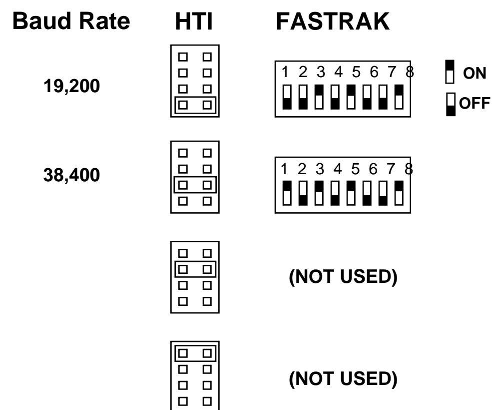
HTI Head Tracker Interface

Overview

HTI is an XBUS compatible interface to the Polhemus FASTRAK and ISOTRAK II head trackers. The HTI relieves the host PC from the tasks associated with receiving and decoding FASTRAK information. By default, the Head Tracker Interface will program the FASTRAK to run in 16-bit mode whereby azimuth, elevation and roll, as well as X, Y and Z information is available at a rate of up to 120 Hz. In this default mode, angles will be passed with a 0.02 degree accuracy and displacement information is available with a 0.03 cm accuracy.

The HTI also allows the host to communicate directly with the FASTRAK and provides a means for sending and receiving raw commands. For more information on using the Polhemus FASTRAK, refer to the FASTRAK user's manual.

When HTI is used to interface Polhemus FASTRAK, different baud rates can be selected by setting the I/O SELECT switches on the back of FASTRAK, and the jumpers on the back of HTI. We highly recommend use of 38,400 baud rate. The available HTI jumper settings and corresponding FASTRAK switch settings are illustrated on the following diagram.



The HTI does not require a NULL modem cable. The cable used to connect FASTRAK and PC is a NULL modem cable which cannot be used to interface HTI and FASTRAK.

HTI version 3.0 and earlier supports only FASTRAK. HTI versions 4.0 and later support both FASTRAK and ISOTRAK II*. The HTI power-up mode is always FASTRAK mode. Calling **HTIisISO(1)** will put the HTI in ISOTRAK II mode, and calling **HTIclear(1)** will put it back to FASTRAK mode. To manually switch the HTI from the FASTRAK mode to ISOTRAK II mode, or vice versa, press and hold the START/STOP button at the top-left of the HTI panel until the LED next to this button stops blinking. When the button is released, the display on the front panel will indicate the present mode.

HTIclear(din)

Op Code	HTI_CLEAR {0x11}	
Prototype	void	HTIclear(int *din*);
Description	Clears the specified HTI and resets it to factory default settings. In this mode information will be handled in the Polhemus 16-bit format.	
Arguments	*din*	Device index number.
Defaults	- Data format: 16-bit integer - Display azimuth rotation angle - HTI in ready mode - Translation unit: centimeter - Rotation unit: degree. - Mode: Fastrak	
Example	See **HTIreadXYZ**.	

* The HTI does not support Isotrak I, which was discontinued in 1993.

HTIgo(din)

Op Code HTI_GO {0x12}

Prototype void HTIgo(int *din*);

Description Places the FASTRAK in continuous transmit mode. Once in this mode the current positional information can be accessed using the following HTI calls: **HTIreadAER**, **HTIfastAER**, **HTIreadXYZ**, **HTIfastXYZ**, and **HTIreadone**.

Arguments *din* Device index number.

Example See **HTIreadXYZ**.

HTIstop(din)

Op Code HTI_STOP {0x13}

Prototype void HTIstop(int *din*);

Description Takes the FASTRAK out of continuos transmit mode.

Arguments *din* Device index number.

Example See **HTIreadXYZ**.

HTIboresight(din)

Op Code HTI_BORESIGHT {0x19}

Prototype void HTIboresight(int *din*);

Description Issues boresight (B1) command to the FASTRAK.

Arguments *din* Device index number.

Example See **HTIreadXYZ**.

HTIshowparam(din, pid)

Op Code HTI_SHOWPARAM {0x1a}

Prototype void HTIshowparam(int *din*, int *pid*);

Description Tells the specified HTI to display the indicated parameter on its LED display.

Arguments

din Device index number.

pid Parameter identification number:

P_AZ	1	AZimuth angle
P_EL	2	ELevation angle
P_ROLL	3	ROLL angle
P_X	4	X displacement
P_Y	5	X displacement
P_Z	6	Z displacement

Example See **HTIreadXYZ**.

HTIreadAER(din, &az, &el, &roll)

Op Code HTI_READAER {0x14}

Prototype void HTIreadAER(int *din*, float **az,* float **el,* float **roll*);

Description Reads the current rotation parameters (azimuth, elevation, and roll angles) from the specified HTI. The returned angles will be in degrees with about a 0.02 degree resolution. Note: this call requires that about 8 bytes of information be transferred between the HTI and the FASTRAK. If speed becomes an issue, try **HTIfastAER** and/or **HTIreadone**.

Arguments

din Device index number.

az Pointer to return azimuth.

el Pointer to return elevation.

roll Pointer to return roll.

Example See **HTIreadXYZ**.

HTIfastAER(din, &az, &el, &roll)

Op Code HTI_FASTAER {0x1d}

Prototype void HTIfastAER(int *din*, qint **az,* qint **el,* qint **roll*);

Description Reads the current rotation parameters (azimuth, elevation, and roll angles) from the specified HTI. The returned angles will be in degrees. This is the fast version of **HT1readAER**. The returned values will be 16-bit integers. The angles will only be valid between ±127 degrees and will have a 1 degree resolution.

Arguments

din	Device index number.
az	Pointer to return azimuth.
el	Pointer to return elevation.
roll	Pointer to return roll.

Example

```
int az,el,roll;
HTIclear(1);
HTIgo(1);
do{
  HTIfastAER(1, &az, &el, &roll);
  printf("Az: %4.2f  El: %4.2f  Ro: %4,2f \n",
                        az,el,roll);
}while(!kbhit());
HTIstop(1);
(void)getch();
```

HTIreadXYZ(din, &x, &y, &z)

Op Code HTI_READXYZ {0x15}

Prototype void HTIreadXYZ(int *din*, float **x,* float **y,* float **z*);

Description Reads the current displacement parameters (X, Y, and Z) from the specified HTI. The returned displacements will be in centimeters with about a 0.03 cm resolution.

Arguments

din	Device index number.
x	Pointer to return X.
y	Pointer to return Y.
z	Pointer to return Z.

Example

```
float x,y,z;
HTIclear(1);
HTIgo(1);
HTIboresight(1);
HTIshowparam(1, P_EL);
HTIreadAER(1, &az, &el, &roll);
HTIreadXYZ(1, &x, &y, &z);
HTIstop(1);
```

HTIfastXYZ(din, &x, &y, &z)

Op Code HTI_FASTXYZ {0x1e}

Prototype void HTIfastXYZ(int *din*, int **x,* int **y,* int **z*);

Description Reads the current head movement parameters (X, Y, and Z translations) from the specified HTI. This is the fast-read version of **HTIreadXYZ.** The returned values will be between ±127 centimeters with a 1 cm resolution.

Arguments

din	Device index number.
x	Pointer to return X.
y	Pointer to return Y.
z	Pointer to return Z.

Example

```
HTIfastXYZ(1, &x, &y, &z);
```

HTIreadone(din, pid)

Op Code HTI_READONE {0x1c}

Prototype float HTIreadone(int *din*, int *pid*);

Description Reads the specified parameter from the indicated HTI. The return value is in floating point format with the same units and resolution as the standard **HTIreadAER** and **HTIreadXYZ** calls.

Arguments

din Device index number.

pid Parameter identification number. See **HTIshowparam** for legal constants.

Example

```
HTIclear(1);
HTIgo(1);
do{
  printf("Roll: %4.2f \n",
             HTIreadone(1, P_ROLL);
}while(!kbhit());
HTIstop(1);
(void)getch();
```

HTIreset(din, pid)

Op Code HTI_RESET {0x1b}

Prototype void HTIreset(int *din*);

Description Sends ^Y to the specified HeadTracker. Note: care must be taken to allow 5 seconds for the FASTRAK to run through initialization.

Arguments

din Device index number;

Example

```
HTIreset(1);
delay(5000);
```

HTIgetecode(din)

Op Code HTI_GETECODE {0x1f}

Prototype int HTIgetecode(int *din*);

Description Returns the last error code transmitted from FASTRAK. Refer to FASTRAK document for more information..

Arguments *din* Device index number;

Example lasterr = `HTIgetecode(1);`

HTIsetraw(din, nbytes, c1, c2)

Op Code HTI_SETRAW {0x18}

Prototype void HTIsetraw(int *din, int nbytes, int c1, int c2*);

Description **HTIsetraw**, **HTIwriteraw**, and **HTIreadraw** provide a method for customizing the operation of the HTI/HeadTracker system. Note: if the HTI default mode of operation does not suit your needs, consider interfacing the FASTRAK directly to the host. Although these three calls allow for some amount of customization, they provide something short of a direct link to the FASTRAK. The **HTIsetraw** call provides the HTI with the basic structure of the output string programmed to be sent by the FASTRAK.

Arguments

din	Device index number;
nbytes	Number of bytes to send.
cl	First lockon character.
c2	Second lockon character.

Example See **HTIreadraw**.

HTIwriteraw(din, cmdstr[])

Op Code	HTI_WRITERAW {0x16}	
Prototype	void	HTIwriteraw(int *din*, char *cmdstr[]*);
Description	Sends a command direct to the FASTRAK.	
Arguments	*din*	Device index number.
	cmdstr[]	Character string buffer holding the FASTRAK command.
Example	See **HTIreadraw**.	

HTIreadraw(din, maxchars, &buf)

Op Code	HTI_READRAW {0x17}	
Prototype	void	HTIreadraw(int *din*, int *maxchars,* char **buf*);
Description	Reads the Output command direct from the specified HTI/FASTRAK.	
Arguments	*din*	Device index number.
	maxchars	Maximum number of characters that should be read.
	buf	Pointer to character buffer.

Example

```
char rtn_str[50];
HTIclear(1);
HTIwriteraw(1, "O1,2\r");
HTIsetraw(1, 21, '0', '1', 2);
HTIgo(1);
do{
  HTIreadraw(1, 25, rtn_str);
  printf("Displacement: %s \n",rtn_str);
}while(!kbhit());
HTIstop(1);
```

HTIisISO(din)

Op Code HTI_ISISO {0x20}

Prototype void HTIisISO(int *din*);

Description Switches the HTI from FASTRAK mode to ISOTRAK II mode.

Arguments *din* Device index number.

Example Following statements will set HTI to ISOTRAK II mode and run it.

```
HTIclear(1);
HTIisISO(1);
HTIgo(1);
```

DB4 Digital Biological Amplifier

Overview

This section describes the commands used to program the DB4 from a computer. See the DB4 User's Guide for information on the operation of the amplifier.

DB4clear(int din);

Op Code	DB4_CLEAR 0x11
Prototype	void DB4clear(int *din*);
Description	Clears the specified DB4 and resets it to last EEPROM settings. Gain, filter, and impedance threshold settings will not be reset.
Arguments	*din* Device index number.

DB4setgain(int din, int chan, float gain);

Op Code	DB4_GAIN 0x12
Prototype	float DB4setgain(int *din*, int *chan*, float *gain*);
Description	Sets gain on the specified channel of specified DB4 to nearest programmable gain setting. Returns actual gain value.
Arguments	*din* Device index number. *chan* CH1, CH2, CH3, CH4, or CHALL *gain* gain (x100 to x1,000,000)
Example	The following sets gain on channel 2 to 10,000x.

```
DB4setgain(1, CH2, 10000.0);
```

DB4selgain(int din, int chan, int gs);

Op Code DB4_GAIN 0x12

Prototype float DB4selgain(int *din*, int *chan*, int *gs*);

Description Sets gain on the specified channel of specified DB4 to the selected gain index. Returns actual gain value. Similar to DB4setgain(), but uses an index to specify the gain.

Arguments

din	Device index number.
chan	CH1, CH2, CH3, CH4, or CHALL
gs	gain setting index (0-36)

Example The following sets gain on all channels to gain setting 5, which corresponds to 600x.

```
DB4selgain(1, CHALL, 5);
```

Table of gain settings

Gain Index	Gain
0-8	100-900 (in 100x steps)
9-17	1,000-9,000 (in 1,000x steps)
18-26	10,000-90,000 (in 10,000x steps)
27-36	100,000-1,000,000 (in 100,000x steps)

DB4setfilt(int din, int chan, int ftype, float ffreq);

Op Code DB4_FILTER 0x13

Prototype float DB4setfilt(int *din*, int *chan*, int *ftype*, float *ffreq*);

Description Sets filter frequency on the specified channel of specified DB4 to nearest programmable filter frequency (see table below for programmable filter frequencies). Returns actual filter frequency.

Arguments

din	Device index number.
chan	CH1, CH2, CH3, CH4, or CHALL
ftype	F_HP (high-pass), F_LP (low-pass), F_NT (notch)
ffreq	corner frequency

Example This code sets the following filter frequencies for channel 1: High-pass 10 Hz, Low-pass 12,000 Hz, Notch 60 Hz.

```
DB4setfilt(1, CH1, F_HP, 10);
DB4setfilt(1, CH1, F_LP, 12000);
DB4setfilt(1, CH1, F_NT, 60);
```

DB4selfilt(int din, int chan, int ftype, int fs);

Op Code DB4_FILTER 0x13

Prototype float DB4selfilt(int *din*, int *chan*, int *ftype*, int *fs*);

Description Sets filter frequency on the specified channel of specified DB4 to the selected filter index (see table below for programmable filter settings and frequencies). Returns actual filter frequency. Similar to DB4setfilt(), but uses an index to specify the frequency.

Arguments

din Device index number.

chan CH1, CH2, CH3, CH4, or CHALL

ftype Filter type: F_HP (high-pass), F_LP (low-pass), F_NT (notch)

fs corner frequency setting

Example This code sets the following filter frequencies for channel 1: High-pass 10 Hz, Low-pass 12,000 Hz, Notch 60 Hz.

```
DB4selfilt(1, CH1, F_HP, 8);
DB4selfilt(1, CH1, F_LP, 33);
DB4selfilt(1, CH1, F_NT, 2);
```

High-Pass & Low-Pass Filter Index	Filter Frequency (Hz)	Notch Filter Index	Notch Filter Frequency (Hz)
0	0	0	0
1-5	5-9	1	50
6-14	10-90	2	60
15-23	100-900	3	100
24-32	1000-10000	4	120
33-37 (low-pass only)	11000-15000	5	150
		6	180

DB4userfilt(int din, int chan, int fn, float coef[]);

Op Code DB4_USERCOEF 0x14

Prototype void DB4userfilt(int *din*, int *chan*, int *fn*, float *coef[]*);

Description Downloads 2nd order Bi-quad IIR filter coefficients to specified filter bank in PD1.

Arguments

din	Device index number.
chan	CH1, CH2, CH3, CH4, or CHALL
fn	filter number (0, 1, 2) (corresponding to F_HP, F_LP, F_NT, respectively)
coef[]	filter coefficients [B0, B1, B2, A1, A2]

Notes The filter number represents one of three cascaded filter banks. User coefficients may be downloaded to any of these banks (i.e. one could download coefficients for a notch filter to filter number 0). See the end of this section for instructions on how to generate filter coefficients with Matlab. DB4getfilt() will not return an indication that user-specified coefficients have been loaded.

Example This example downloads filter coefficients for a 5 kHz notch filter to filter bank number 1.

```
float coef[5]= {0.9845,-1.3925,0.9845,
         -1.3925,0.9691};
DB4userfilt(1, CH2, 1, coef);
```

DB4setIT(int din, int chan, int it);

Op Code DB4_SETIT 0x15

Prototype void DB4setIT(int *din*, int *chan*, int *it*);

Description Sets the impedance threshold for specified channel. This threshold controls the behavior of the impedance LEDs on the headstage.

Arguments

din Device index number.

chan CH1, CH2, CH3, CH4, or CHALL

it impedance threshold (1-99 kOhms)

Example This example sets the impedance threshold on channel 1 to 5000 ohms (5k ohms) and channel 2 to 6000 ohms (6 kohms).

```
DB4setIT(1, CH1, 5);
DB4setIT(1, CH2, 6);
```

DB4nchan(int din, int nc);

Op Code DB4_NCHAN 0x16

Prototype void DB4nchan(int *din*, int *nc*);

Description Set to the number of channels in the headstage.

Arguments

din Device index number.

nc 2 or 4

Example This example configures the DB4 to 2 channel operation.

```
DB4nchan(1, 2);
```

DB4setTS(int din, float amp, float freq);

Op Code DB4_SETTS 0x17

Prototype void DB4setTS(int *din*, float *amp*, float *freq*);

Description Sets the amplitude and frequency of the test signal.

Arguments

din	Device index number.
amp	voltage (0-0.1V).
freq	frequency (1-5960 Hz).

Notes The test tone is played out of a 16-bit D/A. The amplitude is scaled over the 16-bits, so a 0.1 V signal will have the best quality (highest signal-to-noise). A 0.0001 V (0.1 mV) signal will have only about a 30 dB dynamic range.

Example This example sets the test signal on the headstage to 2000 Hz and 0.01 V, and then turns it on.

```
DB4setTS(1, 0.01, 2000);
DB4onTS(1);
```

DB4onTS(int din);

Op Code DB4_TSOO 0x18

Prototype void DB4onTS(int *din*);

Description Activates the test signal on the headstage connected to the specified controller.

Arguments *din* Device index number.

Example This example sets the test signal on the headstage to 2000 Hz and 0.01 V, and then turns it on.

```
DB4setTS(1, 0.01, 2000);
DB4onTS(1);
```

DB4offTS(int din);

Op Code DB4_TSOO 0x18

Prototype void DB4offTS(int *din*);

Description Turns off the test signal on the headstage connected to the specified controller.

Arguments *din* Device index number.

Example This example sets the test signal on the headstage to 2000 Hz and 0.01 V, and then turns it on. The test signal is turned off when the user presses a key.

```
DB4setTS(1, 0.01, 2000);
DB4onTS(1);
do{}while(!kbhit);
DB4offTS(1);
```

DB4startIM(int din, int chan);

Op Code DB4_STARTIM 0x19

Prototype void DB4startIM(int *din*, int *chan*);

Description Starts impedance monitoring by the headstage on the specified channel.

Arguments *din* Device index number.

chan CH1, CH2, CH3, CH4

Example see example under DB4readIM();

DB4stopIM(int din);

Op Code DB4_STOPIM 0x1a

Prototype void DB4stopIM(int *din*, int *chan*);

Description Stops impedance monitoring by the headstage.

Arguments *din* Device index number.

Example see example under DB4readIM();

DB4readIM(int din, int pc);

Op Code	DB4_READIM 0x1b	
Prototype	int	DB4readIM(int *din*, int *pc*);
Description	Returns impedance from the specified channel.	
Arguments	*din*	Device index number.
	pc	impedance of POS or NEG electrode and ground

Example The following example continuously monitors the impedance on headstage channel 1, until the user presses a key.

```
DB4startIM(1, 1);
int imp,imn;
do
{
imp= DB4readIM(1, POS);
imn= DB4readIM(1, NEG);
printf("Impedance Positive %i   Imp Neg
%i\n",imp, imn);
}while(!kbhit());
```

DB4impscan(int din, int tochan);

Op Code DB4_IMPSCAN 0x1f

Prototype int DB4impscan(int *din*, int *tochan*);

Description Scans impedance of multiple channels on HS4 headstage, and returns bitmask for which channels are above the impedance threshold.

Arguments

din Device index number.

tochan channel number to scan through (starting from channel 1). CH1, CH2, CH3, or CH4.

Notes The bit pattern is set by which channels exceed the impedance threshold (1 in bit pattern). Channels that are less than the impedance threshold have a 0 in the bit pattern. See the table at the end of this section for interpreting bit patterns. See the BioAmp User's Guide for information on impedance measurement.

	Negative				Positive			
Channel	4	3	2	1	4	3	2	1

Example The following example scans impedance from channel 1 through channel 3. If impmask=33 (corresponding to a bit pattern of 0010 0001), Channel 2 Negative electrode exceeds the impedance threshold, and Channel 1 Positive electrode exceeds the impedance threshold.

```
impmask = DB4impscan(1, CH3);
```

DB4getclip(int din);

Op Code DB4_GETCLIP 0x1c

Prototype int DB4getclip(int *din*);

Description Returns bit pattern for channel clip status.

Arguments *din* Device index number.

Notes The bit pattern is set by which channels are clipping (1 in bit pattern). Channels that are not clipping have a 0 in the bit pattern. See the table at the end of this section for interpreting bit patterns. See the BioAmp User's Guide for information on Digital and Analog clipping.

	Digital Clip				**Analog Clip**			
Channel	4	3	2	1	4	3	2	1

Example The following example reads the clip bitmask from the DB4. If bitpattern=238, then channels 2, 3, and 4 show both analog and digital clipping on all channels.

```
int bitpattern;
bitpattern=DB4getclip(1);
```

DB4getstat(int din);

Op Code DB4_GETSTAT 0x1d

Prototype int DB4getstat(int *din*);

Description Returns 0 if headstage is not on.
Returns 3 for low-impedance headstage.
Returns 65 for high-impedance headstage.

Arguments *din* Device index number.

Example The following example reads the status of the DB4.

```
int status;
status=DB4getstat(1);
```

DB4powdown(int din);

Op Code DB4_POWDOWN 0x1e

Prototype void DB4powdown(int *din*);

Description Turns off the HS4 headstage.

Arguments *din* Device index number.

Example The following example powers down the HS4 headstage.

```
DB4powdown(1);
```

DB4getgain(int din, int chan, int *sel);

Op Code DB4_GETGAIN 0x20

Prototype float DB4getgain(int *din*, int *chan*, int **sel*);

Description Returns the gain from the specified channel and loads *sel* with index of gain setting.

Arguments *din* Device index number.

chan CH1, CH2, CH3, CH4.

*sel Pointer to int variable to hold gain index.

Notes To just read the gain, and not load sel with the gain index enter 0 for **sel* (e.g.: `gain = DB4getgain(1, CH2, 0`);

Example The following example reads the gain from the DB4 channel 2 and loads the variable *sel* with the gain index

```
float gain;
int sel;
gain = DB4getgain(1, CH2, &sel);
```

DB4getfilt(int din, int chan, int ft, int *sel);

Op Code DB4_GETFILT 0x21

Prototype float DB4getfilt(int *din*, int *chan*, int *ft*, int **sel*);

Description Returns the filter frequency from the specified channel and filter bank, and loads *sel* with index of frequency setting.

Arguments

din	Device index number.
chan	CH1, CH2, CH3, CH4.
ft	Filter type: F_HP (high-pass), F_LP (low-pass), F_NT (notch)
*sel	Pointer to int variable to hold gain index.

Notes To just read the frequency, and not load *sel* with the frequency index enter 0 for **sel* (e.g.: `frequency = DB4getfilt(1, CH2, F_HP, 0)`);

Example The following example reads the frequency settings of the high-pass, low-pass, and notch filter banks from the DB4 channel 1 and 2, and load the variable *sel* with the gain index.

```
float hp, lp, nt;
for(ch=0; ch<=1; ch++);
{
hp = DB4getfilt(1, ch, F_HP, &sel);
printf("Ch %i HP %f, Sel %i  ",ch, hp, sel);
lp = DB4getfilt(1, ch, F_LP, &sel);
printf("LP %f, Sel %i  ", lp, sel);
nt = DB4getfilt(1, ch, F_NT, &sel);
printf("NT %f, Sel %i\n", nt, sel);
}
```

DB4getIT(int din, int chan);

Op Code DB4_GETIT 0x22

Prototype int DB4getIT(int *din*, int *chan*);

Description Returns the impedance threshold setting from the specified channel.

Arguments *din* Device index number.

chan CH1, CH2, CH3, CH4.

Example The following example reads the impedance threshold from the DB4 channel 1.

```
int IT;
IT=DB4getIT(1, CH1);
```

DB4getchmode(int din);

Op Code DB4_GETCHMODE 0x23

Prototype int DB4getchmode(int *din*);

Description Returns the channel mode from the specified DB4.

Arguments *din* Device index number.

DB4getmud(int din);

Op Code DB4_GETCHMODE 0x24

Prototype int DB4getchmode(int *din*);

Description Returns the last manual update made to DB4. The number returned consists of the the channel changed (upper 4 bits) and the function changed (lower 4 bits). Returns 0 if no manual update has been made. Value is reset to 0 after DB4getmud() is run.

Channel Constants

CH1	0x0
CH2	0x1
CH3	0x2
CH4	0x3
CHALL	0xa

Function Constants

MUD_GAIN	0x2
MUD_HP	0x3
MUD_LP	0x4
MUD_NT	0x5
MUD_IT	0x6
MUD_ALL	0xf

Arguments *din* Device index number.

Example The following code checks for the last change made manually to the DB4. The return value will be 0x12 (decimal 18) if the channel 2 gain has been changed.

```
int mud;
mud=DB4getmud(1);
```

Binary/Decimal Conversion Chart

This chart can assist you in decoding the bit pattern returned by DB4getclip(); and DB4impscan().

Binary	Decimal	Binary	Decimal
0000 0000	0		
0000 0001	1	0001 0000	16
0000 0010	2	0010 0000	32
0000 0011	3	0011 0000	48
0000 0100	4	0100 0000	64
0000 0101	5	0101 0000	80
0000 0110	6	0110 0000	96
0000 0111	7	0111 0000	112
0000 1000	8	1000 0000	128
0000 1001	9	1001 0000	144
0000 1010	10	1010 0000	160
0000 1011	11	1011 0000	176
0000 1100	12	1100 0000	192
0000 1101	13	1101 0000	208
0000 1110	14	1110 0000	224
0000 1111	15	1111 0000	240

Examples:
A decimal value of 12 corresponds to the bit pattern 0000 1100.
For DB4getclip(); this would mean that channels 3 and 4 have analog clipping at the headstage.

For decoding decimal values larger than those shown in this table, find the decimal value in the right hand column that is closest to, but not greater than, the returned value. The four most significant bits (i.e. the 4 left-most bits) for that decimal value form the four most significant bits of the returned value. To determine the four least significant bits, subtract that decimal value from the returned value. The remainder decimal value (which will be between 0 and 15) form the four least signficant bits (i.e. the 4 rightmost bits) of the bit pattern.

For example, the bit pattern for 238 would be the bit pattern for 224 (1110) followed by the bit pattern for 14 (i.e. 238-224, which is 1110). So, the bit pattern for 238 is 1110 1110.

When you are programming you should make use of the bit shifting features of your programming language (e.g. << and >> in C) to decode bit patterns automatically.

Generating Filter Coefficients with Matlab

The DB4 Controller accepts user downloaded filter coefficients for any of its three filter banks (see the programming guide for more information). It is quite simple to generate filter coefficients with Matlab.

The following commands show how to generate Butterworth filter coefficients from the Matlab command line. The sample rate is 48 kHz for the 2 channel HS4 headstage, and 24 kHz for the 4 channel HS4 headstage.

Prototype **[B A] = butter(order, Corner Frequency/sample rate)**

Low-pass Example (1000 Hz low-pass filter for 2-channel HS4)

[B A] = butter(2, 1000 / 48000)

Matlab returns:

B = 0.0010 0.0020 0.0010

A = 1.0000 -1.9075 0.9116

The first A coefficient is always assumed to be 1, and is not used in the DB4userfilt(); command.

High-pass Example (1200 Hz high-pass filter for 4-channel HS4)

[B A] = butter(2, 1200/24000, 'high')

Notch Example (1000 Hz Notch with 100 Hz band for 2-channel HS4)

This example generates a notch filter that can be loaded to one filter block. The notch is 1st order (i.e. 6dB per octave).

[B A] = butter(1, [950 1050]/48000, 'stop')

Band-pass Example (1000 to 5000 Hz band-pass filter for 4-channel HS4)

This example generates a band-pass filter that can be loaded to one filter block. The pass-band is 1st order (i.e. 6dB per octave).

[B A] = butter(1, [1000 5000]/24000)

MC1 Motor Controller

Overview

This section describes the commands used to program the MC1 from a computer. See the MC1 Tear Sheet for information on the operation of the controller through the front panel.

MC1clear(int din);

Prototype	void	MC1clear(int *din*);
Description	Clears the specified MC1.	
Arguments	*din*	Device index number.

MC1pos(int din, int pos);

Prototype	void	MC1pos(int din, int pos);
Description	Sets the position to which the motor will move when the MC1move() command is issued.	
Arguments	*din*	Device index number.
	pos	Position.

MC1vel(int din, int vel, int perm);

Prototype	void	MC1vel(int din, int vel, int perm);
Description	Sets the maximum velocity of the motor.	
Arguments	*din*	Device index number.
	vel	velocity in units/s
	perm	

MC1acc(int din, int acc, int perm);

Prototype void MC1acc(int din, int acc, int perm);

Description Sets the acceleration of the motor. The motor will accelerate until it reaches the maximum velocity set by the MC1vel() command.

Arguments

din Device index number.

acc acceleration in units/s/s

perm

MC1move(int din);

Prototype void MC1move(int din);

Description Initiates movement of the motor to the position set by the MC1pos() command.

Arguments *din* Device index number.

Example The following example sets a new position to which to move (20 degrees), and then intiates movement.

```
MC1pos(1, 20);
MC1move(1);
```

MC1syncmove(int din);

Prototype void MC1syncmove(int din);

Description

Arguments *din* Device index number.

Example

MC1gear(int din, float gratio);

Prototype void MC1gear(int din, float gratio);

Description Sets the gear ratio of the motor.

Arguments

din Device index number.

gratio Gear ratio

Example The following example sets the gear ratio of the motor to

MC1home(int din, int home);

Prototype void MC1home(int din, int home);

Description Sets the home position of the motor. This is the position the motor will return to when the MC1gohome() command is issued or the HOME button is pressed on the front of the MC1 controller.

Arguments

din Device index number.

home Home position of MC1

Example The following example sets the home position to 90 degrees and goes there with the MC1gohome() command.

```
MC1home(1, 90);
MC1gohome(1);
```

MC1boundry(int din, int minp, int maxp);

Prototype void MC1boundry(int din, int minp, int maxp);

Description Sets the boundaries of the motor to limit the movement of the arm to a particular range. This is often a safety feature, and prevents the arm from tangling up any cables that may be attached to it.

Arguments

din Device index number.

minp Minimum position

maxp Maximum position

Example The following example sets the arm to limit movement between –90 and +90 degrees.

```
MC1boundry(1, -90, 90);
```

MC1reference(int din, int refmode, int srchvel, int refpos);

Prototype void MC1reference(int din, int refmode, int srchvel, int refpos);

Description References the MC1 through the use of a click switch or manually.

Arguments

din	Device index number.
refmode	Reference Mode RM_MANUAL 1 RM_REFSWITCH 2
srchvel	Search velocity
refpos	Reference position

Example To reference the motor to zero degrees manually through the front panel. Press the REFERENCE button, then turn the dial until the motor is positioned to zero degrees. Press ENTER. To reference manually through programming, move to the position you want to reference using MC1pos() and MC1move(), then call MC1reference(); with the value of this reference position. This will rereference the last position set by MC1pos() to the value *refpos*.

MC1filter(int din, int par, int v);

Prototype void MC1filter(int din, int par, int v);

Description Sets the filter coefficients for the motor.

Arguments

din Device index number.

par Parameter

FP_Kp 0x08
FP_Ki 0x04
FP_Kd 0x02
FP_Ilim 0x01

v Value.

Notes These coefficients are specific to the motor you are using and the load that is on the motor. They should be set to provide smooth motion. See the motor catalog for approximate values that should work for your motor.

Example

MC1status(int din);

Prototype int MC1status(int din);

Description Checks the status of the motor controller.

Arguments

din Device index number.

STAT_OFF 0
STAT_STATIC 1
STAT_MOVE 2
STAT_ERROR 3

Example The following example waits for the motor to finish moving before playing a sound.

```
MC1pos(1, 20);
MC1move(1);
do{}while(MC1status(1)==STAT_MOVE);
// code to play sound
```

MC1curpos(int din);

Prototype int MC1curpos(int din);
Description Reads the current position of the motor.
Arguments *din* Device index number.
Example

MC1curvel(int din);

Prototype int MC1curvel(int din);
Description Reads the current velocity of the motor.
Arguments *din* Device index number.
Example

MC1go(int din);

Prototype void MC1go(int din);
Description
Arguments *din* Device index number.
Example

MC1stop(int din);

Prototype void MC1stop(int din);
Description Stops the MC1.
Arguments *din* Device index number.
Example

MC1kill(int din);

Prototype void MC1kill(int din);
Description Programmable kill switch for the MC1.
Arguments *din* Device index number.
Example

MC1zero(int din);

Prototype void MC1zero(int din);

Description Makes the MC1 go to the zero position.

Arguments *din* Device index number.

Example The following example reads the frequency settings of the high-pass, low-pass, and notch filter banks

MC1gohome(int din);

Prototype void MC1gohome(int din);

Description Makes the motor go to the home position (the position set by `MC1home`();).

Arguments *din* Device index number.

MC1goref(int din);

Prototype void MC1goref(int din);

Description Makes the motor go to the reference position.

Arguments *din* Device index number.

Example

MC1getparam(int din, int parcode);

Prototype int MC1getparam(int din, int parcode);

Description Reads one of several parameters from the MC1. This can be used to read values that may be set by the user from the front panel of the MC1, and then reset values in software.

Arguments *din* Device index number.

parcode velocity in units/s

PC_VEL	1
PC_ACC	2
PC_GEAR	3
PC_MINP	4
PC_MAXP	5
PC_HOME	6
PC_REFMODE	7

PC_SRCHVEL 8
PC_REFPOS 9
PC_Kp 10
PC_Ki 11
PC_Kd 12
PC_Ilim 13