:

E1432 Host Interface Library

Function Reference

A.05.00

Written by Eric Backus

NAME

e1432mon − monitor for E1432 firmware messages

SYNOPSIS

e1432mon [-fuV] [-L laddr] [-t timeout] [-v verbosity]

DESCRIPTION

E1432mon is a debugging program for monitoring messages fromthe E1432 firmware. As it executes, the E1432 firmware prints certain status messages to an internal buffer. If the *e1432mon* program is not running, these status messages are thrown away. The *e1432mon* program monitors for these status messages, and prints them out. The amount of status information printed by the E1432 firmware is controlled by the *e1432_set_internal_debug* program.

Note that running *e1432mon* can slow down your measurement, since the process of printing these debug messages takes time. It is quite possible for *e1432mon* to cause errors, if the internal debug level in the E1432 firmware is such that messages get printed during critical interrupt handlers.

Also, this program should **not** be run during any measurement that uses the local bus. The reasons for this are somewhat obscure. First of all, the local bus lines are positioned on the VXI P2 connector such that activity on the VXI D32 data lines will cause glitches on the local bus lines. (This is a consequence of the geometry of the lines on the VXI P2 connector and the shielding on the connector of the E1432 module - so all varieties of VXI mainframe will show this problem.) Normally D32 is used to access the E1432 module, but we disable D32 access when local bus measurements are running. However, this means that each time a program wants to read a 32-bit register, two 16-bit reads must be done. If the *e1432mon* program executes between these two 16-bit reads, it will corrupt the value of the second 32-bit read, causing problems in the measurement you are trying to run.

Command Line Parameters:

-f	Force e1432mon to tell the E1432 firmware that it's attached to a tty. Useful when running e1432mon from a script.
-L *logical_addr*	Specifies the logical address of the **E1432**. The default value is 8.
-t *timeout*	The e1432mon program will timeout and exit after *timeout* seconds. A value of zero (the default) disables the timeout. This timeout can be useful when running e1432mon from a script.
-u	Display usage message.
-v *verbosity*	Specifies the verbosity level. The default value is 2. This is used only to debug the *e1432mon* program itself. Normally, *e1432mon* prints only status messages from the E1432 firmware, or an error message if the communication between *e1432mon* and the firmware is corrupted. When the verbosity level is set higher than 2, *e1432mon* also prints other information as it executes.
-V	Print version info.

RETURN TYPE

E1432mon returns 0 upon success, or returns non-zero if an error is detected.

SEE ALSO

e1432_set_internal_debug(3)

NAME

e1432supp − print system information to aid in debugging E1432 problems e1432_sys_info − print system information to aid in debugging E1432 problems

SYNOPSIS

e1432supp

DESCRIPTION

E1432supp and *e1432_sys_info* print system information which may help when debugging E1432, E1433, and E1434 problems. They print things like the revision date of the E1432 host interface libraries and the VXI system configuration.

E1432_sys_info is exactly the same as *e1432supp*. The original name was *e1432supp*, but we decided that *e1432_sys_info* is a better name for what it does.

E1432supp is really a shell script, so it is useful only on HP-UX systems. This script does not understand any command-line options.

The last part of the script assumes that there is an E1432, E1433, or E1434 module at VXI logical address 8. If there is not, several error messages will be printed (which can be ignored).

RETURN TYPE

E1432supp returns 0 if the final e1432mon bootup output is successful, returns non-zero otherwise.

NAME

hostdiag − test and diagnose E1432 hardware

SYNOPSIS

hostdiag [-hPsuvV] [-f file] [-L laddr] [-S [cage:]slot] [-O list]

DESCRIPTION

Hostdiag is a program for testing and diagnosing E1432/3/4 hardware. It will find and diagnose most hardware failures. By default, it tests the module at VXI logical address 8.

Note that when testing a module with a source, signals will be output during the course of testing. Therefore, it may be wise to disconnect the source outputs when running *Hostdiag*.

Command Line Parameters:

-h Does a quick, half-hearted pass at testing by bypassing the tests which involve downloading code to the module.

-f *file* Uses "file" as the source of code to download to the module instead of the default `/opt/e1432/lib/sema.bin`.

-L *logical_addr*
Specifies the logical address of the module to be tested. The default value is 8. The -L option and the -S option are mutually exclusive.

-O *option_list*
Tests the module against a model options list. For example -O "E1432,1DE,AYF" tests the module as an 8 channel E1432A with the tachometer option. The model number and the options can be found on the serial number plate(s) on the right side of the module. Without this option, *hostdiag* only tests what it finds present. Hardware which has failed in such a way that it appears to be absent will not be detected without this option.

-P Prints only a pass/fail message - no diagnostic printouts.

-s Additionally runs the "standard input/output" tests. Sources finish testing with 1 VPk, 1 KHz sine on each output for manual verification of output functionality. Input testing (both E1432 and E1433 input SCAs and the Tachometer input) assumes 1 VPk, 1 KHz sine input on each channel. This allows testing of additional portions of the signal path which inaccessible from the internal tests.

-S *[cage:]slot*
Test the module in the vxi cardcage, *cage*, vxi slot, *slot*. *cage* defaults to 1 if not specified and simply counts up from the root cardcage, which is 1 (ie. the second cardcage is *cage* = 2). Default is to test the module at logical address 8. The -L option and the -S option are mutually exclusive. The -S option is not available on Windows platforms.

-u Display usage message.

-v Specifies the verbose printing. Normally, *hostdiag* does not print anything unless an error is found. With this option, *hostdiag* prints status messages as it operates. This option also enables additional diagnostic information which is not of much use outside the factory.

-V Print version info.

It should be noted that *hostdiag* attempts to determine if portions of a module are broken. It is not guaranteed to find all hardware problems. It is *not* a performance test or system verification test.

RETURN TYPE

Hostdiag returns 0 upon success, or returns non-zero if an error is detected.

NAME

hwblkio − do low-level I/O with the E1432

SYNOPSIS

hwblkio [-@CcdfhlLmMpRstuvWxy]

DESCRIPTION

This program is used to perform low-level I/O operations with an E1432.

This program is an extended version of the *hwblkio* program distributed with the E1485 programmer's toolkit, and with the HP3565 programmer's toolkit. Additional options have been added to allow it to work well with the E1432 VXI module. The program still has the capability to communicate with an E1485 VXI module or HP3565 system. However, those capabilities are not documented here.

This program reads or writes a block of data from the 96002 processor of an **E1432** module. The data transfer is done at a very low level, using interrupts, so that any firmware executing in the 96002 is completely unaware that the transfer is taking place. This makes *hwblkio* a good debugging tool.

Hwblkio will write a block to the E1432 if the **-W** command line argument is used. The data is first read from the file specified by the **-f** option. If the size is not specified, as much of the input file will be output as possible.

Hwblkio will read a block from the E1432 if the **-R** option is used. The data will then be written to the standard output. The data is almost always binary, so it is best to redirect the output to a file or a program like *od(1)*.

If neither **-R** or **-W** are specified, *hwblkio* will do nothing.

At any time, the behavior of *hwblkio* is governed by parameters specified on the command line and by parameters specified in the optional start-up file **${HOME}/.hwiorc.** Command line arguments take precedence over parameters specified in the start-up file.

Command Line Parameters:

-@ *block_addr*	This specifies the CPU address to access. *block_addr* may either be specified in decimal or in hex (hex must include a leading 0x).
-C	Disable reading of start-up file. All required parameters must be specified as command line arguments.
-c	Display all parameters. No I/O operations will take place. This is useful for debugging.
-d *interface_addr*	This options specifies the interface address. This parameter is usually something like "vxi" when the channel type is VXI, or "hpib" when the channel type is remote VXI.
-f *file_name*	Specifies the input file, when writing data to the E1432 module.
-h *hpib_addr*	Specifies the HP-IB address of the **E1406** when the channel type is remote VXI.
-l	Access DSP L memory (L memory is a concatenation of X and Y memory). The **-@** option specifies the offset into L memory, and **-s** specifies the size in 64-bit words.
-L *logical_addr*	Specifies the logical address of the **E1432**.
-m *mod_type*	This option specifies the type of module being accessed. *mod_type* must be specified as **E1432**.
-M *chan_type*	Specifies the HWIO channel type. The valid entries for this field are "vxi" and "remote_vxi".

-p Access DSP program memory. The **-s** option specifies the size in 32-bit words.

-R This enables the reading of a block.

-s *npoint* Specifies the size. *npoint* is normally in 32-bit words, but it is 64-bit words when reading from L memory. *npoint* may be specified in decimal or in hex (hex must include a leading 0x).

-t *timeout* This option specifies the HWIO timeout in microseconds.

-u Display usage message.

-v Turn verbose mode on. This is useful for debugging.

-W This enables the writing of a block.

-x Access DSP X RAM. **-s** specifies the size in 32-bit words.

-y Access DSP Y RAM. **-s** specifies the size in 32-bit words.

Start-up file:

The following are parameters that can be set in the start-up file.

interface_addr *interface_addr*
Use specified interface. There is no default for this parameter. See **-d** command line option.

timeout *timeout* Timeout in microseconds. A timeout of zero, will never timeout. If not specified, a timeout of 5 seconds will be used. See **-t** command line option.

hpib_hw_addr *hpib_addr*
HP-IB address of **E1406**. This field is used with the HWIO channel type is remote VXI. See **-h** command line option.

hwio_channel_type *hwio_channel_type*
The HWIO channel type. This may be specified as "vxi" or "remote_vxi". See **-M** command line option.

module_type *module_type*
Specifies the type of module to be accessed. This is the same as the **-m** command line option, and should be set to **E1432**.

vxi_logical_addr *laddr*
Logical address. See **-L** command line option.

All blank lines and lines beginning with the **#** character are ignored. Since *hwblkio* is not the only program to access the start-up file, parameters defined but not used by these programs will be ignored.

FILES

${HOME}/.hwiorc

EXAMPLES

An example of **${HOME}/.hwiorc** is as follows:

```
hwio_channel_type    vxi
interface_addr       vxi
vxi_logical_addr     8
# This is used by hwinstall
exec_file            /opt/e1432/lib/sema.bin
# This shouldn't be needed, but hwblkio is stupid
# But then, hwzap is even stupider - it ignores this line
module_type          e1432
```

```
# This is needed for hwzap
which_rom            boot
```

To read and display 16 32-bit words from the **E1432**, at address 0x00001000 of X memory, type:

```
hwblkio -R -x -@0x1000 -s16 | xd | less
```

RETURN VALUE

Hwblkio returns 0 upon success and non-zero if an error is detected.

SEE ALSO

hwinstall(1), hwzap(1)

NAME

hwinstall − install firmware into E1432 RAM

SYNOPSIS

hwinstall [**-cCSuw**] [**-d** interf_addr] [**-L** logical_addr] [**-M** hwio_chan_type] [**-f** op_sys_file] [**-h** hpib_addr] [**-t** timeout] [**-v** vmode]

DESCRIPTION

The *hwinstall* program is used to install firmware into RAM in an E1432 module.

This program is an extended version of the *hwinstall* program distributed with the E1485 programmer's toolkit, and with the HP3565 programmer's toolkit. Additional options have been added to allow it to work well with the E1432 VXI module. The program still has the capability to communicate with an E1485 VXI module or HP3565 system. However, those capabilities are not documented here.

At any time, the behavior of *hwinstall* is governed by parameters specified on the command line and by parameters specified in the optional start-up file **${HOME}/.hwiorc**. Command line arguments take precedence over parameters specified in the start-up file.

Command Line Parameters:

-c	Display setup and exit. This is useful for diagnostics.
-C	Disable the reading of the RC file. This is useful if *hwinstall* is embedded in another program and the caller wishes to setup the parameters and ignore the RC file.
-d *interf_addr*	Specifies the interface address. This parameter is usually something like "vxi" when the channel type is VXI, or "hpib" when the channel type is remote VXI.
-f *file*	Specifies the file containing the firmware image to install. Normally, this should be set to `/opt/e1432/lib/sema.bin`.
-h *hpib_addr*	Specifies the HP-IB address of the **E1406** when the channel type is remote VXI.
-L *logical_addr*	Specifies the logical address of the **E1432**.
-M *chan_type*	Specifies the HWIO channel type. The valid entries for this field are "vxi" and "remote_vxi".
-S	Do not do error checking after the firmware is initialized. When this option is specified, *hwinstall* will return immediately without waiting for the firmware to initialize itself, unless the **-w** option is also specified.
-t *timeout*	Specifies a timeout for all HWIO operations in microseconds.
-u	Display a usage message.
-w	Wait for the firmware to initialize itself before exiting. The wait will time out after approximately 10 seconds. By default, *hwinstall* will wait for the firmware to initialize itself. This option is only useful if the **-S** option has also been specified.
-v *mode*	Specifies the verbose mode, or debug level. With a verbose mode of 0, no output will occur. With a verbose mode of 1, only error messages will be displayed. With a verbose mode of 5, much information is displayed. The default is 3.

Start-up file:

The following are parameters that can be set in the start-up file.

interface_addr *interface_addr*
specifies the interface address. See **-d** command line option.

timeout *timeout* specifies the timeout in microseconds. See **-t** command line option.

hpib_hw_addr *hpib_addr*
specifies the HP-IB address of the **E1406** command module. See **-h** command line option.

exec_file *file* specifies the file containing the firmware image to install. See **-f** command line option.

hwio_channel_type *hwio_channel_type*
The HWIO channel type. This may be specified as "vxi" or "remote_vxi". See **-M** command line option.

vxi_logical_addr *laddr*
Logical address. See **-L** command line option.

All blank lines and lines beginning with the # character are ignored. Since *hwinstall* is not the only program to access the start-up file, parameters defined but not used by *hwinstall* are ignored.

The *hpib_addr* and *interface_addr* parameters must be specified as command-line options or in the start-up file--they do not have default values. The *timeout* parameter is not required, the default is 5 seconds. The *file* parameter should be set to `/opt/e1432/lib/sema.bin`. Note that the default is `/usr/e1485/lib/spos`, which is not useful for the E1432 module.

The default behavior when installing the firmware is to wait for the firmware to initialize itself. The **-S** command-line option may be used to disable waiting for the firmware to initialize. In this event, *hwinstall* will not wait for the firmware to initialize itself unless the **-w** command-line option is specified. This is all a little convoluted, but this is how hwinstall has always worked with other products.

RETURN VALUE

Hwinstall returns 0 upon success, or non-zero if an error is detected.

FILES

${HOME}/.hwiorc
/opt/e1432/lib/sema.bin

EXAMPLES

An example of the **${HOME}/.hwiorc** file is:

```
hwio_channel_type    vxi
interface_addr       vxi
vxi_logical_addr     8
# This is used by hwinstall
exec_file            /opt/e1432/lib/sema.bin
# This shouldn't be needed, but hwblkio is stupid
# But then, hwzap is even stupider - it ignores this line
module_type          e1432
# This is needed for hwzap
which_rom            boot
```

SEE ALSO

hwblkio(1), hwzap(1)

NAME

hwzap − program E1432 flash ROM

SYNOPSIS

hwzap [options]

DESCRIPTION

Hwzap is used to program the E1432 flash ROM. The flash ROM contains boot code for the 96002 processor on the substrate board, and calibration constants used during system calibration.

This program is an extended version of the *hwzap* program distributed with the E1485 programmer's toolkit, and with the HP3565 programmer's toolkit. Additional options have been added to allow it to work well with the E1432 VXI module. The program still has the capability to communicate with an E1485 VXI module or HP3565 system. However, those capabilities are not documented here.

A copy of the E1432 96002 boot code (a binary image) can be found in `/opt/e1432/lib/core1P.bin`. This boot code is small; its main purpose is to set things up for downloading the real firmware.

The behavior of *hwzap* is governed by parameters specified on the command line and by parameters specified in the optional start-up file **${HOME}/.hwiorc**. Command line arguments take precedence over parameters specified in the start-up file.

Command Line Parameters:

-c	Display setup and exit. This is useful for diagnostics.
-C	Disable the reading of the RC file. This is useful if *hwzap* is embedded in another program and the caller wishes to set the parameters and ignore the RC file.
-d *interface_addr*	Specifies the interface address. This parameter is usually something like "vxi" when the channel type is VXI, or "hpib" when the channel type is remote VXI.
-f *dfile*	Specifies the file containing the binary image to program.
-h *hpib_addr*	Specifies the HP-IB address of the **E1406** when the channel type is remote VXI.
-L *logical_addr*	Specifies the logical address of the **E1432**.
-m *mod_type*	Specifies the type of VXI module. Should be specified as "e1432".
-M *chan_type*	Specifies the HWIO channel type. The valid entries for this field are "vxi" and "remote_vxi".
-R *which_rom*	Specifies which ROM to program. This field must be specified as "boot".
-t *timeout*	Specifies a timeout for all I/O operations in microseconds.
-u	Display usage message.
-v *vmode*	Specifies the verbose mode, or debug level. With a verbose mode of 0, no output will occur. With a verbose mode of 1, only error messages will be displayed. With a verbose mode of 5, much information is displayed. The default is 3.

Start-up file:

The following are parameters that can be set in the start-up file.

interface_addr *interface_addr*
: specifies the interface address. See **-d** command line option.

timeout *timeout* specifies the timeout in microseconds. See **-t** command line option.

hpib_hw_addr *hpib_addr*
specifies the HP-IB address of the **E1406** command module. See **-h** command line option.

hwio_channel_type *hwio_channel_type*
The HWIO channel type. This may be specified as "vxi" or "remote_vxi". See **-M** command line option.

vxi_logical_addr *laddr*
Logical address. See **-L** command line option.

downloadable_file *dfile*
Specifies the file containing the binary image to zap into the ROM. See **-f** command line option.

which_rom *which_rom*
Specifies the ROM to program. See **-R** command line option.

rom_image_file *file* Specifies the file which contains the image to program. See **-f** command line option.

All blank lines and lines beginning with the **#** character are ignored. Since *hwzap* is not the only program to access the start-up file, parameters defined but not used by *hwzap* are ignored.

EXAMPLE

An example of zapping the flash rom with the default ROM image is:

```
hwzap -f /opt/e1432/lib/core1P.bin -R boot -m e1432
```

RETURN TYPE

Hwzap returns 0 upon success, or returns non-zero if an error is detected.

FILES

${HOME}/.hwiorc
/opt/e1432/lib/core1P.bin

SEE ALSO

hwinstall(1)

NAME

progopt − install options in E1432 hardware

SYNOPSIS

progopt [-RuVZ] [-A optStr] [-D optStr] [-L laddr]

DESCRIPTION

Progopt is a program that allows a user to install some E1432 hardware options, as well as read the hardware serial number.

Command Line Parameters:

-A *optStr*

Add option string <optStr> to option list. Typically this is a hexadecimal codeword for a particular option.

-D *optStr*

Delete option string <optStr> from option list. This can be used to delete an incorrectly entered option or codeword.

-L *laddr*

Talk to logical address <laddr>, default 8.

-R Read the option list from hardware. Codewords will appear literally, not as the option that they encode.

-S Read the serial string from hardware. This serial string is typically needed to generate the codeword for a particular option for a particular module.

-u Display usage message.

-V Print version information.

-Z Zero (delete) all options from the option list in hardware. This should not be done unless the information is available to reprogram the correct option string(s).

RETURN TYPE

Progopt returns 0 upon success, or returns non-zero if an error is detected.

NAME

ptman − on-line manual reader

SYNOPSIS

ptman [-u] [-l] [-r] [-b base_dir] [-p pager] title ...

DESCRIPTION

Ptman is used to access the on-line **E1432** Host Interface Library manual.

This program is an extended version of the *ptman* program distributed with the E1485 programmer's toolkit, and with the HP3565 programmer's toolkit. Additional options have been added to allow it to work well with the E1432 VXI module. The program still has the capability to provide E1485 or HP3565 manual pages.

The simplest usage of *ptman* is to display the manual page of any **E1432** function or program when the name of the function or program is known. For example, the command:

```
ptman ptman
```

will perform the following:

1. Search for the on-line manual data base files, located under the product base directories. *Ptman* searchs under the "/opt/e14*", "/usr/e14*" and "/usr/hp3565" directories. This allows the *ptman* program to be used to display manual pages from the E1432, E1431, E1430, E1485, and HP3565 products.
2. In the located "mandb" files, ptman searches for a reference to the function *ptman*.
3. Format the manual page via nroff, using the -man macro package.
4. If the output of ptman is being sent to a tty device (i.e. the screen) the formatted man page is sent to the standard input of the pager more(1). If the output of ptman is being sent to a pipe, the formatted manual page is sent directly to the standard output.

A second typical use of *ptman* is to display a manual page when the name of the function or program is not quite known. Suppose you want to know the name of the **E1432** function that will read data from the module. All you remember is that it starts with programs that start with the string 'e1432_read':

```
ptman -l e1432_read
```

From the output of this command, the user can pick out the desired function, then read the man page via the following:

```
ptman e1432_read_float32_data
```

The operation of *ptman* may be modified via environment variable and command line parameters. Command line parameters have presidence over environment variables.

The following command line parameters are supported:

-b *base_dir* Use *base_dir* as the product base directory. The default is "**/opt/e14***", "**/usr/e14***" and "**/usr/hp3565**" directories. Ptman checks under the "lib" subdirectory for the "mandb" file.

-l List file names only. This option is used in the second example above.

-p *pager* Use *pager* as the pager program. The default pager is "**more**". Pager may also be specified using the **PAGER** environment variable.

-r Displays the revision of the manual data base.

-u Display usage message.

the following environment variables are supported:

PTMAN_BASEDIR
Sets the product base directory. See above for default

PAGER Sets the pager to use. See above for default

NOTES

Currently, *ptman* may not terminate after scrolling through the man pages. If this occurs, hit the **break** key to exit *ptman*.

RETURN VALUE

Ptman returns 0 upon success, or non-zero if an error is detected.

FILES

/opt/e14*/lib/mandb
/usr/bin/col
/usr/bin/more
/usr/bin/nroff
/usr/bin/tbl
/usr/e14*/lib/mandb
/usr/hp3565/lib/mandb
/usr/lib/macros/an

SEE ALSO

man(1)

NAME

srcutil − program E1432 source board flash ROM

SYNOPSIS

srcutil [options]

DESCRIPTION

Srcutil is used to get hardware and firmware revision information, and is used to program the E1432 source board flash ROM. The revision information is displayed on stdout. The flash ROM contains boot code, and signal generation code for the 56002 processor on the source board.

A copy of the E1432 source board rom image (a binary image) can be found in `/opt/e1432/arbsrc/srcrom.bin`.

The behavior of *srcutil* is governed by optional parameters specified on the command line.

Optional Command Line Parameters:

-f *dfile* Specifies the file containing the binary image to program.

-L *logical_addr* Specifies the logical address of the **E1432**.

-P Specifies that the flash ROM is to be programmed.

-u Display usage message.

EXAMPLE

An example of getting source revision information is:

```
/opt/e1432/arbsrc/srcutil
```

An example of programming the source flash ROM is:

```
/opt/e1432/arbsrc/srcutil -P
```

RETURN TYPE

Srcutil returns 0 upon success, or returns non-zero if an error is detected.

FILES

/opt/e1432/arbsrc/srcrom.bin

SEE ALSO

e1432_src_prog_romimage(3)

NAME

e1432_arm_measure − Manually arm, move E1432s from IDLE to ARM state

SYNOPSIS

```
SHORTSIZ16 e1432_arm_measure(E1432ID hw, SHORTSIZ16 ID,
                             SHORTSIZ16 wait_after)
```

DESCRIPTION

e1432_arm_measure moves all modules in the group from the **IDLE** state to the **ARM** state.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

This function performs a "manual arm", and does not need to be called when the group is set to "auto arm". This function is called for the first time after *e1432_init_measure*, and then, when in block mode, after each data block has been read out of the module. See the "Measurement setup and control" section earlier in this manual, for a detailed description of the measurement states.

This function waits for all modules to be in the **IDLE** state, before proceeding further, and it will return an error if this state is not reached after a limited time. After the call to *e1432_arm_measure* completes successfully, the measurement will proceed further, as the trigger event occurs (see *e1432_trigger_measure*).

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If the measurement involves more than one module, it is mandatory that a *group ID* be used, rather than a *channel ID*.

wait_after determines whether this function will wait for the module to actually move beyond the **ARM** state. If zero, the function does not wait; if non-zero, the function waits.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_trigger_measure, e1432_arm_measure_master_finish

NAME

e1432_arm_measure_master_finish − Master side measurement manual arm
e1432_arm_measure_master_setup − Master side measurement manual arm
e1432_arm_measure_slave_finish − Slave side measurement manual arm

SYNOPSIS

```
SHORTSIZ16 e1432_arm_measure_master_finish(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_arm_measure_master_setup(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_arm_measure_slave_finish(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

These functions are *not* normally needed by the typical application. They are provided for use only when a measurement must use multiple VXI mainframes. The typical single-mainframe application should simply use *e1432_arm_measure* instead.

These functions are used in conjunction with *e1432_arm_measure* to manually arm a multi-mainframe measurement.

Instead of using these functions, it is possible to use *e1432_set_mmf_delay* instead. However, *e1432_set_mmf_delay* is not as reliable.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*.

e1432_arm_measure_master_setup is used to set up modules in the master mainframe of a multi-mainframe system, before doing the manual arm. The *ID* parameter to this function should be a channel that is in the master mainframe, or a group containing only channels in the master mainframe.

e1432_arm_measure_slave_finish is used to verify that modules in the slave mainframes of a multi-mainframe system have all successfully armed. This should be called after doing the manual arm. The *ID* parameter to this function should be a group containing the channels from the slave mainframes.

e1432_arm_measure_master_finish is used to clean up the multi-mainframe manual arm process. This should be done at the end of the manual arm process. The *ID* parameter to this function should be the same as the *ID* that was originally passed to *e1432_arm_measure_master_setup*.

The following sequence should be used to reliably manual arm all modules in a multi-mainframe setup:

```
e1432_arm_measure_master_setup(hw, master_id);
e1432_arm_measure(hw, global_id, 0);
e1432_arm_measure_slave_finish(hw, slave_id);
e1432_arm_measure_master_finish(hw, master_id);
```

In the above code, "master_id" is the ID of a channel or group in the master mainframe; "global_id" is the ID of a group containing all channels in the measurement (the same ID that was originally given to *e1432_init_measure*); and "slave_id" is the ID of a group containing the channels in the slave mainframes. Note that a real application should check the return values of these functions for errors.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_init_measure_master_finish, e1432_arm_measure, e1432_trigger_measure, e1432_set_mmf_delay, e1432_multimain

NAME

e1432_assign_channel_numbers − Preset and assign IDs to E1432s
e1432_assign_channels − Assign IDs to E1432s

SYNOPSIS

```
SHORTSIZ16 e1432_assign_channel_numbers(SHORTSIZ16 nmod,
                                        SHORTSIZ16 *la_list,
                                        E1432ID *hw)
SHORTSIZ16 e1432_assign_channels(SHORTSIZ16 nmod,
                                 SHORTSIZ16 *la_list,
                                 E1432ID *hw,
                                 int preset)
```

DESCRIPTION

One of *e1432_assign_channel_numbers* or *e1432_assign_channels* must be called exactly once, following a call to *e1432_init_io_driver*, in order to declare to the library the logical addresses of the E1432 modules that will be used.

These two functions are identical, except that *e1432_assign_channels* has an additional parameter to specify whether the modules should all be preset. A normal user typically would use *e1432_assign_channel_numbers*, which is exactly equivalent to:

```
e1432_assign_channels(nmod, la_list, hw, 1)
```

nmod is the count of logical addresses passed in the second parameter, *la_list*. This number should be between **0** and **255**. A value of zero means that the function should free all memory allocated by a previous call to *e1432_assign_channel_numbers* or *e1432_assign_channels*, and do nothing else.

la_list is the pointer to the list of logical addresses to be used by the library. Logical addresses have values ranging between **1** and **255**. All modules in the *la_list* are preset to their power-up state, as with *e1432_preset*. There is no requirement that the *la_list* array be in numerically increasing order. The channel numbers will start at one in the first logical address and increase in value in the same order that the logical addresses are found in the array.

The function returns in *hw* a hardware ID which is used when calling most other E1432 Host Interface library functions.

preset specifies whether the preset the modules. If non-zero, the modules are preset before the function returns.

This function checks the existence of an E1432 module at each of the logical addresses given in *la_list*, and allocates logical channel identifiers for each channel in all of the E1432s. Input channels, source channels, and tach/trigger channels are kept logically separated. Channel numbers for each type of channel are numbered starting from one, so there will be input channels 1 through M, source channels 1 through N, and tach/trigger channels 1 through P, where M is the number of input channels, N is the number of source channels, and P is the number of tach/trigger channels.

As an example, suppose two logical addresses 100 and 101 are passed to the function, and the logical address 100 has two 4-channel input SCAs and an Option AYF 2-channel tachometer input, while logical address 101 has three 4-channel input SCAs and an Option 1D4 single-channel source board. In this case, input channel IDs 1 through 8 are assigned to the eight input channels at logical address 100, while input channel IDs 9 through 20 are assigned to the twelve input channels at logical address 101. Tach/trigger channel IDs number 1 and 2 are assigned to the two tach/trigger channels at logical address 100, and Source channel ID number 1 is assigned to the source channel at logical address 101.

This function dynamically allocates memory using *malloc(3)*, in order to store the internal structures kept for each channel and each module.

When programming for multiple mainframe environments, where two or more mainframes are connected by an MXI interface, there are some limitations and special cases that need to be accounted for. The TTLTRG lines that are used by the E1432 to sync multiple module measurements are unidirectional through the MXI interface, so this restricts which modules can drive the TTLTRG lines. The e1432_read_xxx_data() function should use a group ID, not just channel IDs. See the manual page *e1432_multimain(5)* for more info.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_create_channel_group, e1432_delete_all_chan_groups, e1432_init_io_driver, e1432_preset, e1432_e1431_diff(5), e1432_id(5), e1432_multimain(5)

NAME

e1432_auto_range − Automatically set the input range

SYNOPSIS

```
SHORTSIZ16 e1432_auto_range(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ64 meas_time)
```

DESCRIPTION

e1432_auto_range attempts to set the input range of a single channel or group of channels. It tries to pick a range which puts the input signal near full scale but which does not overload the input.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

meas_time specifies how long to monitor the input channels, in seconds, before deciding whether the range is OK, or needs to be increased, or decreased. If the range needs to be changed on any channel, then the channels are again monitored for *meas_time* seconds and checked to see if the range needs to be changed or not. This is done in a loop until all input channels reach a range at which there is neither an overload nor an under-range.

As a special case, if the *meas_time* is specified as exactly zero, the auto range will estimate an appropriate measurement time. The estimate uses the formula: meas_time=blocksize*0.1/span.

The algorithm normally will change the input ranges up or down in order to reach the "best" range. However, *e1432_set_auto_range_mode* can be used to modify this algorithm. If the auto range mode is set to **E1432_AUTO_RANGE_MODE_UP** for a given channel, then the auto range will only increase the input range (or leave it the same) but will not decrease the range setting for that channel. Similarly, if the auto range mode is set to **E1432_AUTO_RANGE_MODE_DOWN** for a given channel, then the auto range will only decrease the input range (or leave it the same) but will not increase the range setting.

If a channel is at the maximum range setting and the input is still overloaded, the channel is left at the maximum range and no error is generated. If a channel is at the minimum range setting and the input is still under-ranged, the channel is left at the minimum range and no error is generated.

If this function is called when a measurement is running, it will change the ranges of the channels "on the fly", leaving the measurement running with the input channels at the new ranges.

If this function is called when no measurement is running, it will internally run a measurement, do the auto-range, and then stop the internal measurement. The internal measurement is mostly like any other measurement, but it temporarily turns off VME interrupts from the module, disables the Local Bus, and deactivates any sources, so that the auto-range will not have any unexpected effects outside of the module.

Due to DSP limitations, auto-range does **NOT** work on an E1433 if the current clock frequency (as set by *e1432_set_clock_freq*) is greater than 128 kHz. If an auto-range is attempted when the clock frequency is greater than 128 kHz, an error is returned.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_auto_range_mode, e1432_set_clock_freq

NAME

e1432_auto_zero − Null out DC offset

SYNOPSIS

```
SHORTSIZ16 e1432_auto_zero(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_auto_zero attempts to null out the DC offset of a single channel or group of channels.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

For input channels, an auto-zero involves grounding the input amplifier and measuring the resulting offset voltage. The measured offset is saved and used to correct future measurements. For the E1432 and E1433 input SCAs, an auto-zero also involves measuring the amplitude accuracy, and a gain constant is saved and used to correct future measurements.

For source channels, an auto-zero involves disconnecting the external connector, programming the source to produce zero volts, and then measuring the actual voltage produced. The measured offset is saved and used to correct future measurements.

For tach channels, an auto-zero is not done.

Doing an auto-zero (for either source or input channels) will abort any currently-running measurement. In general, it is better to auto-zero a group of channels all at once rather than auto-zero the individual channels separately, because the group auto-zero can do most of the work in parallel and will therefore not take as long.

When firmware is initially loaded into a module with *e1432_install*, all input channels are auto-zeroed, **but source channels are not**.

In general, the best auto-zero results will be obtained if all parameters for the input or source channel are set up prior to doing the auto-zero. For example, if the measurement will use a clock frequency of 65536 Hz, it is best to auto-zero the channels after setting the clock frequency to 65536 Hz. Typically, if an auto-zero is going to be done, it is done after setting up all parameters but before starting a measurement with *e1432_init_measure*.

Doing an auto-zero after changing clock frequency is important for input channels in the E1432 module, because the gain of the E1432's ADC varies slightly with clock frequency. This is less of a problem for the input channels in an E1433 module.

In a multi-module measurement, it is best to make sure all modules (or at least all source modules) have stopped running a measurement (using *e1432_reset_measure*) before doing an auto-zero. Otherwise, it is possible that an auto-zero in one module could cause an active source in a different module to produce an output signal.

For the E1434 or option 1D4 source channels additional steps are required before auto-zero if the filter frequency of the anti-alias digital filter has been changed. These steps are necessary to get the correct filter path set up in the source before doing the auto-zero, so that the auto-zero works correctly.

1. Set the source output mode to grounded. This prevents any glitching at the source BNC, but it causes the source to drive the module's CALOUT line.

2. Set the source arm mode to manual, so the source never really starts, so all we drive onto the CALOUT line is DC.
3. Set the module to not drive CALOUT onto the VXI sumbus, so we can't accidentally mess up anyone else's use of the VXI sumbus.
4. Start and stop a measurement. This gets the correct signal path loaded into the source, without glitching the output.
5. Do the auto-zero.
6. Restore the sumbus setting, the source output mode, and the arm mode.

You could do this procedure for all auto-zeros, even when the filter frequency does not change. It takes a little longer, though.

EXAMPLE

```
/* This code is required when auto-zeroing a source channel, if the
   source filter frequency changes.  Otherwise, all that is
   necessary is to call e1432_auto_zero(hw, group). */

/* Set manual arm, output grounded, sumbus off */
CHECK(e1432_set_arm_mode(hw, group, E1432_MANUAL_ARM));
CHECK(e1432_set_source_output(hw, group, E1432_SOURCE_OUTPUT_GROUNDED));
CHECK(e1432_set_sumbus(hw, group, E1432_SUMBUS_OFF));
/* Start and stop measurement, then auto-zero */
CHECK(e1432_init_measure(hw, group));
CHECK(e1432_reset_measure(hw, group));
CHECK(e1432_auto_zero(hw, group));
/* Restore desired sumbus, output, and arm settings */
CHECK(e1432_set_sumbus(hw, group, E1432_SUMBUS_OFF));
CHECK(e1432_set_source_output(hw, group, E1432_SOURCE_OUTPUT_NORMAL));
CHECK(e1432_set_arm_mode(hw, group, E1432_AUTO_ARM));
```

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

NAME

e1432_block_available − Return status of data FIFO

SYNOPSIS

```
SHORTSIZ16 e1432_block_available(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_block_available returns one of three status conditions:

- A positive number if a block of data is available to be read. Use *e1432_read_raw_data*, *e1432_read_float32_data*, or *e1432_read_float64_data* to read the data.

- Zero if a block of data is not available.

- A (negative) error number if an error occurred.

When the E1432 is set to block mode, using *e1432_set_data_mode* with **E1432_BLOCK_MODE**, the module stops acquiring data after one block of data for each active channel has accumulated in the FIFO. The module does not acquire more data until the block has been read out of the FIFO.

When the E1432 is set to overlap block mode, using *e1432_set_data_mode* with **E1432_DATA_MODE_OVERLAP_BLOCK**, the module acquires data continuously (as with continuous mode, below) but will stop momentarily if the FIFO fills up, so there will never be a FIFO overflow.

When the E1432 is set to continuous mode, using *e1432_set_data_mode* with **E1432_CONTINUOUS_MODE**, the module continues to acquire data until the E1432 is re-initialized with *e1432_init_measure* or the FIFO overflows because data is not read out fast enough.

If the FIFO overflows in continuous mode, data acquisition stops. Data remaining in the FIFO is valid and can be read. *e1432_block_available* returns 1 as long as at least a block of data remains in the FIFO. Calls to *e1432_block_available* when the FIFO is empty because of data overflow will return the error **ERR1432_FIFO_OVERRUN**.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

The usage of *e1432_block_available* varies depending upon whether a group ID or channel a ID is used. If a group ID is used, *e1432_block_available* will return positive when every channel in the group has a block ready for reading.

When given a group ID, the *e1432_block_available* function internally does some group-related operations to ensure that the group remains synchronized properly. For this reason, **it is almost always better to use a group ID than a channel ID for this function**.

If a channel ID is used, *e1432_block_available* will return positive when that channel has a block that is ready to be read.

The concept of having a block of data ready makes sense only for input channels, not for tach or source channels, so *e1432_block_available* normally ignores any tach or source channels that are present in the group *ID*. However, there is one exception to this rule. In a multi-module RPM-arm or order tracking measurement, if one module has no active input channels, but does have active tach channels, the tach channels must be in the group ID passed to *e1432_block_available*. The reason that that this is necessary is to ensure that the RPM arms and the tach data stay synchronized between all of the modules.

EXAMPLE

```
/* wait for data, handle errors */
while(!(error = e1432_block_available(groupID)));
if (error < 0)
    call_error_handler_routine();
else
    e1432_read_raw_data(groupID, buffer, size, &actualCount);
```

RESET VALUE

Not applicable.

RETURN VALUE

Return positive if successful, 0 if no data available, a (negative) error number if error. The two most likely errors returned will be ERR1432_FIFO_OVERRUN and ERR1432_TACH_BUFFER_OVERFLOW.

SEE ALSO

e1432_read_raw_data, e1432_read_float32_data, e1432_read_float64_data, e1432_get_raw_tachs, e1432_send_tachs

NAME

e1432_cached_parm_update − Update parameters cached by the host library

SYNOPSIS

```
SHORTSIZ16 e1432_cached_parm_update(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

When reading data from an E1432 module using *e1432_read_xxx_data*, the host library must know the values of several E1432 parameters, such as blocksize. Rather than query the E1432 module each time the data transfer is done, the host library saves (caches) the value of these parameters internally, and uses the saved value when doing the data transfer. This reduces the overhead of the data transfer functions. This function, *e1432_cached_parm_update*, is what actually saves the parameter values in the host library internal data structures.

e1432_cached_parm_update is normally called internally by *e1432_init_measure*, and is therefore not normally needed by the end user. We provide it for use in special cases where it may be useful.

An example of when this might be useful is if one host computer is responsible for setting up E1432 measurement parameters and starting a measurement, but a second computer is used to actually transfer the E1432 data. The second computer must use *e1432_cached_parm_update* after the first computer sets up the measurement, to update the internally cached parameters on the second computer, so that the data transfer will work properly.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. Typically a group ID is used.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_read_float32_data, e1432_read_float64_data, e1432_read_raw_data

NAME

e1432_channel_group_add − Add a channel to a channel group
e1432_channel_group_remove − Remove a channel from a channel group

SYNOPSIS

```
SHORTSIZ16 e1432_channel_group_add(E1432ID hw, SHORTSIZ16 group_id,
                                   SHORTSIZ16 chan_id)
SHORTSIZ16 e1432_channel_group_remove(E1432ID hw, SHORTSIZ16 group_id,
                                      SHORTSIZ16 chan_id)
```

DESCRIPTION

e1432_channel_group_add adds a channel to an existing channel group. If the channel is already a member of the channel group, this function does nothing.

e1432_channel_group_remove removes a channel from an existing channel group. If the channel is not currently a member of the channel group, this function does nothing.

There is nothing wrong with deleting all channels from a channel group using *e1432_channel_group_remove*. It is valid to have a group with no channels in it.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

group_id is the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*.

chan_id is the ID of a single channel.

When adding channels to or deleting channels from a channel group, these functions attempt to preserve the current "trigger master" setting if there is one. However, with *e1432_channel_group_remove* it is possible that the trigger master module is no longer in the group. In this case, a new trigger master is picked arbitrarily.

RESET VALUE

Not applicable.

RETURN VALUE

e1432_channel_group_add and *e1432_channel_group_remove* return 0 if successful, or a (negative) error number otherwise.

SEE ALSO

e1432_assign_channel_numbers, e1432_create_channel_group, e1432_delete_channel_group, e1432_set_trigger_master

NAME

e1432_check_overloads − Check for overload and under-range

SYNOPSIS

```
SHORTSIZ16 e1432_check_overloads(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *any, SHORTSIZ16 *comm,
                                 SHORTSIZ16 *diff, SHORTSIZ16 *half)
```

DESCRIPTION

e1432_check_overloads queries the overload/under-range registers and returns status information using four pointers. The first pointer location *any* will be non-zero if any overload occured (the *any* parameter does not get set for half-range condition, only overload). The other three pointer locations will be filled with arrays of TRUE/FALSE flags, one per active channel in the original channel list used in e1432_create_channel_group(). Any of the four pointers can be NULL if that information is not wanted.

This function is meant to be used for real-time updating of the current overload status of channels. It returns the "recent" overload status of the channels. "Recent" is defined to be one blocksize worth of data points. Unlike the E1431 Host Interface library, calling this function does not clear the overload status. Also unlike the E1431 function, this function does not have separate information about ADC overloads versus other kinds of differential overloads. Also unlike the E1431 function, this function provides information about whether a channel is under-ranged.

To instead get the overload status for a particular block of data, use *e1432_set_append_status*, and get the overload status from the trailer data.

If this function is given a group ID, it returns overload status only for active channels in that group. The arrays pointed to by *comm*, *diff*, and *half* must be large enough to hold information for all of these active channels.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group or single channel.

any is a pointer to SHORTSIZ16 that is set by any overload (i.e. if any element of *comm* or *diff* is set). *any* can be NULL if the information is not desired.

comm is a pointer to SHORTSIZ16 that is set by a common mode input overload. *comm* can be NULL if the information is not desired.

diff is a pointer to SHORTSIZ16 that is set by a differential mode input overload. *diff* can be NULL if the information is not desired.

half is a pointer to SHORTSIZ16 that is set when the signal is not under-ranged. *half* can be NULL if the information is not desired.

EXAMPLE

```
SHORTSIZ16 any, error;
SHORTSIZ16 comm[5];
SHORTSIZ16 diff[5];
SHORTSIZ16 half[5];

/* assuming groupID is a group with 5 active channels */
error = e1432_check_overloads(hw, groupID, &any, comm, diff, half);
```

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_read_raw_data, e1432_set_append_status

NAME

e1432_check_src_shutdown − Check for source shutdown

SYNOPSIS

```
SHORTSIZ16 e1432_check_src_shutdown(E1432ID hw, SHORTSIZ16 chanID)

SHORTSIZ16 e1432_check_src_overload(E1432ID hw, SHORTSIZ16 chanID)

SHORTSIZ16 e1432_check_src_overread(E1432ID hw, SHORTSIZ16 chanID)

SHORTSIZ16 e1432_check_src_underrun(E1432ID hw, SHORTSIZ16 chanID)
```

DESCRIPTION

e1432_check_src_shutdown returns a 1 if the queried source is shutdown. It returns a 0 if not shutdown.

e1432_check_src_overload returns a 1 if the queried source has been overdriven by an external signal. It returns a 0 if not.

e1432_check_src_overrread returns a 1 if the queried source has run out of arbdata. It returns a 0 if not.

e1432_check_src_underrun returns a 1 if the queried source has run out of real-time (results in the output signal being turned off). It returns a 0 if not.

These checks are used to check the cause of an E1432_IRQ_SRC_STATUS interrupt.

For *e1432_check_src_overread*, the cause is cleared by its associated check function. Another interrupt should not occur for the same cause until that cause has been cleared. This also allows the causes to be latched in case polling is desired instead of an interrupt.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

chanID is the ID of a group or single channel.

RESET VALUE

Not applicable.

RETURN VALUE

Return 1 if the check is true, a 0 if the check is false, a (negative) error number otherwise.

SEE ALSO

e1432_check_src_arbrdy

NAME

e1432_create_channel_group − Create a group of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_create_channel_group(E1432ID hw, SHORTSIZ16 nchan,
                                      SHORTSIZ16 *chan_list)
```

DESCRIPTION

e1432_create_channel_group creates and initializes a channel group. A channel group allows you to issue commands to several E1432 channels at once, simplifying system setup. You can overlap channel groups. The state of an individual E1432 channel that is in more than one channel group, is determined by the most recent operation performed on any group to which this channel belongs.

As a side effect, this function makes all input and tach channels in the channel group active, and all source channels in the channel group inactive. Unlike the E1431 library, this function does not inactivate other channels within the modules that the channels are in. Also unlike the E1431 library, this function does not preset the channels in the new group.

An additional side effect of this function is that it resets the "auto_group_meas" parameter for each module that has a channel in the channel list. Most applications use the default value of "auto_group_meas", so most applications will not care about this. See *e1432_set_auto_group_meas* for details about this parameter.

This function dynamically allocates memory to keep track of the groups which have been created. This memory can be freed by *e1432_delete_channel_group* and *e1432_delete_all_chan_groups*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

nchan is the count of channels passed in the third parameter, *chan_list*. This number should be between **0** and the maximum number of channels available.

chan_list is the pointer to the list of logical channels identifiers of the group to be created. This list must be in ascending order and contain no repeats. If *chan* is zero, then this parameter is ignored and can be set to NULL.

To supply the ID of an input channel, the input channel number is given as an argument to the **E1432_INPUT_CHAN()** macro. For backwards compatibility with the E1431, the macro currently does nothing. To supply the ID of a source channel, the source channel number is given as an argument to the **E1432_SOURCE_CHAN()** macro. To supply the ID of a tach/trigger channel, the tach/trigger channel number is given as an argument to the **E1432_TACH_CHAN()** macro.

It is legal to have a mixture of input, source, and tach channels in one group. It is legal to create a group that has no channels in it.

As an example, to create a group consisting of the first three input channels and the eighth and ninth input channels, the code would like something like this:

```
SHORTSIZ16 chan_list[5];
SHORTSIZ16 input_group;

chan_list[0] = E1432_INPUT_CHAN(1);
chan_list[1] = E1432_INPUT_CHAN(2);
chan_list[2] = E1432_INPUT_CHAN(3);
chan_list[3] = E1432_INPUT_CHAN(8);
chan_list[4] = E1432_INPUT_CHAN(9);
```

```
input_group = e1432_create_channel_group(hw, 5, chan_list);
```

To create a group consisting of the first two source channels, the code would look something like this:

```
SHORTSIZ16 chan_list[2];
SHORTSIZ16 source_group;

chan_list[0] = E1432_SOURCE_CHAN(1);
chan_list[1] = E1432_SOURCE_CHAN(2);
source_group = e1432_create_channel_group(hw, 2, chan_list);
```

RESET VALUE

Not applicable.

RETURN VALUE

If successful, this function returns the *ID* of the group that was created, which is then used to reference the channel group in most other functions in this library. Because a group ID is always negative, if an error occurs a positive error number is returned by this function. This is the only function that returns a positive error number.

SEE ALSO

e1432_assign_channel_numbers, e1432_channel_group_add, e1432_channel_group_remove, e1432_delete_channel_group, e1432_set_auto_group_meas, e1432_id(5)

NAME

e1432_debug_level − Enable/disable register write printout

SYNOPSIS

```
void e1432_debug_level(SHORTSIZ16 level)
```

DESCRIPTION

e1432_debug_level controls printing of low-level register accesses to E1432 modules. When enabled, the debug message includes the register number, and the new contents being read or written. This function, in conjunction with the **E1432 Hardware Reference Manual** allows detailed examination of the sequence of register writes as well as the contents of the E1432 registers at the bit level.

level is should be set to **0** for no debug printing, **1** to print all register writes, or **2** to print register reads and writes.

RESET VALUE

After a reset, *level* is set to **0**, printing of register writes is disabled.

RETURN VALUE

This function does not return a value.

SEE ALSO

e1432_trace_level, e1432_set_internal_debug, e1432_print_errors

NAME

e1432_delete_channel_group − Delete a group of E1432 channels
e1432_delete_all_chan_groups − Delete all groups of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_delete_channel_group(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_delete_all_chan_groups(E1432ID hw)
```

DESCRIPTION

e1432_delete_channel_group deletes a group previously made using *e1432_create_channel_group*. After this call, the deleted group ID should not be used anymore.

e1432_delete_all_chan_groups deletes all channel groups.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*.

RESET VALUE

Not applicable.

RETURN VALUE

e1432_delete_all_chan_groups always returns 0. *e1432_delete_channel_group* returns 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_assign_channel_numbers, e1432_create_channel_group, e1432_channel_group_add, e1432_channel_group_remove

NAME

e1432_display_state − Dump E1432 module states in easy-to-read format

SYNOPSIS

```
void e1432_display_state(E1432ID hw)
```

DESCRIPTION

e1432_display_state uses printf to dump the current state of each module that was in the logical address list when *e1432_assign_channels* was called. The "current state" is just the state that the E1432 interface library stores - most of the parameters of the E1432 module are stored in the module itself and are not displayed by this function.

This function also prints information on the groups that have been created, and which channels belong to each group.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

RESET VALUE

Not applicable.

RETURN VALUE

None

SEE ALSO

e1432_assign_channels

---- NOT FOR GENERAL USE ----

NAME

e1432_dsp_exec_query − Send commands, receive responses from SCA DSP(s)

SYNOPSIS

```
SHORTSIZ16 e1432_dsp_exec_query(E1432ID hw, SHORTSIZ16 ID,
                      LONGSIZ32 exec_cmd,
                      LONGSIZ32 exec_data_size, LONGSIZ32 *exec_data,
                      LONGSIZ32 query_data_size, LONGSIZ32 *query_data)
```

DESCRIPTION

e1432_dsp_exec_query sends commands and optional data to an individual SCA DSP or a group of SCA DSPs and optionally reads the response(s).

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained by a call to *e1432_create_channel_group*, or the ID of a single channel.

exec_cmd is the host interrupt vector that will be sent in conjunction with the command data.

exec_data_size is the size of the command data block to be sent. If it is zero or *exec_data* is NULL, no command data will be sent.

exec_data is a pointer to the 32 bit command data.

query_data_size is the number of words of response expected from each DSP after the command executes. If it is zero or *query_data* is NULL, no response will be expected.

query_data is a pointer to the 32 bit an array which will receive the response data. It must be long enough to include the responses from all DSPs in *ID*, that is its size in 32 bit words must be at least *query_data_size* * number of DSPs addressed.

For *E1433* input channels, *exec_data* is first optionally sent across the host data port. Then *exec_cmd* is sent to the the host interrupt vector port. Execution then waits for the *CIP* (Command In Progress) bit to be cleared from the DSP. Then, the *query_data* is optionally read through the host data port. This is repeated for each channel in *ID*, since there is one DSP per channel.

This function is not currently implemented for any other SCA DSPs.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise. **_NO_ID**, if *ID* is not a valid channel or group ID. **_BUS_ERROR**, if any of the underlying register accesses fail. **_BUFFER_TOO_SMALL**, if the buffer requirements of *exec_data* and *query_data* exceeds the fixed internal buffer. **_SCA_FIRMWARE_ERROR** and **_SCA_HOSTPORT_FAIL** are symptomatic of the command failing on the DSP or the wrong amount of response data being expected.

SEE ALSO

e1432_sca_dsp_download

NAME

e1432_finish_measure − Clean up the VXI bus after a measurement is done

SYNOPSIS

```
SHORTSIZ16 e1432_finish_measure(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_finish_measure is used to clean up after the end of a measurement. It is similar to *e1432_reset_measure* in that it stops all inputs and sources, but in addition it also turns off any signals that modules in the group may be sending onto the VXI bus.

e1432_finish_measure does not disable the channels in the group. Depending on what your application will be doing next, it is sometimes a good idea to also call *e1432_set_active(hw, ID,* **E1432_CHANNEL_OFF***)* in addition to *e1432_finish_measure*, to disable the channels in the group.

This function sets the clock source for each module in the group to internal (see *e1432_set_clock_source*); stops each module from driving the clock or sync/trigger line onto the VXI backplane (see *e1432_set_multi_sync* and *e1432_set_clock_master*); sets the local bus mode to pipe mode (see *e1432_set_lbus_mode* and *e1432_reset_lbus*); and disables interrupt generation (see *e1432_set_interrupt_mask* and *e1432_set_interrupt_priority*).

This call stops a previous group measurement from interfering with a current group measurement if the two groups happen to share TTLTRG lines for module synchronization.

See *e1432_init_measure* for a description of multi-module measurements.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_reset_measure

NAME

e1432_get_current_data − Get current measurement data block

SYNOPSIS

```
SHORTSIZ16 e1432_get_current_data(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 data_type, SHORTSIZ16 data_size,
                                  void **data, LONGSIZ32 *actual_count)
```

DESCRIPTION

e1432_get_current_data gets the most recent measurement data blocks for the channel(s) selected by *ID*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It specifies which channel(s) to get current measurement data for.

data_type specifies which type of measurement data is requested. Currently, only **E1432_OCTAVE_DATA** is supported.

data_size specifies the type of numerical format that is requested. Currently, only **E1432_DATA_SIZE_FLOAT32** and **E1432_DATA_SIZE_FLOAT64** are supported.

data points to an array of pointers to the memory blocks which are to receive the measurement data. All channels specified by *ID* must have a pointer in the array pointed to by *data*. The memory blocks pointed to by those pointers must have enough space to receive the measurement data. The channel ordering is the same as that provided in the e1432_create_channel_group function call used to create *ID*, with no gaps for unused channels.

For *data_type* **E1432_OCTAVE_DATA**, with linear averaging, as set by *e1432_set_octave_avg_mode* or Octave hold modes other than **E1432_OCTAVE_HOLD_MODE_OFF**, as set by *e1432_set_octave_hold_mode*, data may be either the instantaneous Octave value or the averaged/hold mode result, depending on which is most recently available.

The actual number of data points read into each block is returned in the memory location pointed to by *actual_count*.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize

NAME

e1432_get_current_rpm − Get the current RPM of a tachometer channel
e1432_get_data_rpm − Get the arming RPM of last data block
e1432_get_next_arm_rpm − Get the arming RPM for the next data block

SYNOPSIS

```
SHORTSIZ16 e1432_get_current_rpm(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *rpm)
SHORTSIZ16 e1432_get_data_rpm(E1432ID hw, SHORTSIZ16 ID,
                              FLOATSIZ32 *rpm)
SHORTSIZ16 e1432_get_next_arm_rpm(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *rpm)
```

DESCRIPTION

e1432_get_current_rpm returns the current RPM of an active tachometer channel into the variable pointed to by *rpm*.

e1432_get_data_rpm returns the trigger RPM for the last data block into the variable pointed to by *rpm*. This information is also available in the data block trailer if the trailer has been enabled using *e1432_set_append_status*. This is only meaningful in one of the RPM arming modes set by *e1432_set_arm_mode*.

e1432_get_next_arm_rpm returns the arming RPM for the next data block that will be acquired. The measurement will arm when this RPM value is reached.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is a channel ID of a specific tachometer channel.

rpm is a pointer to a 32-bit float which will be filled in with the requested RPM value.

RESET VALUE

After a reset, *rpm* is set to 0.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_arm_mode, e1432_set_append_status

NAME

e1432_get_current_value − Get current Peak value, RMS value, or FIFO available count

SYNOPSIS

```
SHORTSIZ16 e1432_get_current_value(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 value_type,
                                   FLOATSIZ32 *value)
```

DESCRIPTION

e1432_get_current_value gets the Peak or RMS current value for of a single channel or group of channels *ID*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

The *value_type* should be one of:

E1432_CURRENT_VAL_PEAK,
indicating that the current Peak value is to be returned.

E1432_CURRENT_VAL_RMS,
indicating that the current RMS value is to be returned.

E1432_CURRENT_VAL_FIFO_AVAIL,
indicating that the current amount of data in the FIFO is to be returned.

value is a pointer to a single floating point variable in the case of a single channel *ID*, or a pointer to an array of floating point variables large enough to accept one current value result for each of the channels in channel group *ID*.

When *value* is **E1432_CURRENT_VAL_PEAK** or **E1432_CURRENT_VAL_RMS**, if a channel is not enabled or if the requested value computation has not been enabled with *e1432_set_peak_mode* or *e1432_set_rms_mode*, 0 will be returned in the corresponding *value*.

The functions *e1432_set_peak_decay_time*, *e1432_set_rms_avg_time*, and *e1432_set_rms_decay_time* control the time responses of the peak and RMS current values.

The weighting, selected by *e1432_set_weighting*, is applied to the peak and RMS current values.

The **E1432_CURRENT_VAL_PEAK** or **E1432_CURRENT_VAL_RMS** values are currently available only with the E1433. **E1432_CURRENT_VAL_FIFO_AVAIL** works for both the E1432 and E1433.

RESET VALUE

After a reset, Peak and RMS value computation is not enabled, and the FIFO is not running, so *e1432_get_current_value* will only return zero valued results.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_peak_mode, e1432_set_rms_mode, e1432_set_peak_decay_time, e1432_set_rms_avg_time, e1432_set_rms_decay_time, e1432_set_weighting.

NAME

e1432_get_decimation − Get decimation factor

SYNOPSIS

```
SHORTSIZ16 e1432_get_decimation(E1432ID hw, SHORTSIZ16 ID,
                                LONGSIZ32 *dec)
```

DESCRIPTION

e1432_get_decimation gets the current decimation rate of a channel. This is the ratio between the "sample0" sample clock frequency, and the effective sample rate of the data being acquired by the channel.

The sample clock frequency is the clock frequency that is connected to one of the VXI TTLTRG lines, if needed to synchronize several E1432 modules. This clock frequency is normally, but not always, the fundamental rate at which data is collected from the ADCs on the SCAs. The effective sample rate is determined by the span specified with *e1432_set_span*, and by the SCAs present in the hardware.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

dec is a pointer to 32-bit int, and will be filled with the current decimation factor.

RESET VALUE

The reset value of *dec* is one.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_span, e1432_get_clock_freq

NAME

e1432_get_error_string − return pointer to last error string
e1432_fill_error_string − copy last error string to buffer

SYNOPSIS

```
char *e1432_get_error_string(void)
SHORTSIZ16 e1432_fill_error_string(char *string, SHORTSIZ16 max,
                                   SHORTSIZ16 *actual);
```

DESCRIPTION

e1432_get_error_string returns a pointer to a string holding a one line description of the last error condition.

e1432_fill_error_string returns the same information as *e1432_get_error_string*, but it copies it to a passed-in buffer, instead of returning a pointer to the string. This works better when calling from Visual Basic.

string is a pointer to a buffer allocated by the caller.

len specifies the size of the *string* buffer. The function will not write more than this many characters, including the trailing null, to the *string*.

actual points to a value that will be filled in by *e1432_fill_error_string*. The value will be the length of the string written, not including the trailing null.

EXAMPLE

```
SHORTSIZ16 error;

error = e1432_set_span(ID, span);
if (error)
    (void) printf("error: %d %s0, error, e1432_get_error_string());
```

RESET VALUE

Not applicable.

RETURN VALUE

e1432_get_error_string returns a pointer to the error string. *e1432_fill_error_string* returns zero.

SEE ALSO

e1432_print_errors

NAME

e1432_get_fwrev − Get revision of "sema.bin" file

SYNOPSIS

```
SHORTSIZ16 e1432_get_fwrev(char *path, LONGSIZ32 *fwrev)
```

DESCRIPTION

e1432_get_fwrev returns the firmware revision for the file specified by *path*. This file should be a `sema.bin` file that could be installed in an E1432/E1433/E1434 module.

path is the path to the `sema.bin` file, including the `sema.bin` filename. The typical value to use for *path* would be `/opt/e1432/lib/sema.bin`.

fwrev should point to a 32-bit integer. This integer will get filled with the firmware revision for the `sema.bin` file. The revision is of the form "19970416". It is a decimal integer containing a four-digit year followed by a two-digit month followed by a two digit day.

If this `sema.bin` file is installed in an E1432/E1433/E1434 module using *e1432_install*, and then if *e1432_get_hwconfig* is used to get information about the module, the *fw_rev* field in the *e1432_hwconfig* structure will be equal to the *fwrev* returned by this function.

The typical user will probably not need to use this function. Its purpose it to provide a way to compare a `sema.bin` file in the host computer with the firmware that is executing in a module, to see which is newer.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_hwconfig, e1432_install

NAME

e1432_get_group_info − Get information about a group

SYNOPSIS

```
SHORTSIZ16 e1432_get_group_info(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 item, SHORTSIZ16 *answer,
                                SHORTSIZ16 size)
```

DESCRIPTION

hpe1432_getGroupInfo is used to return selected information about a channel group that has been previously created using hpe1432_createChannelGroup(). The information is returned in an array that must be allocated by the caller of *hpe1432_getGroupInfo*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

item is used to select which data is returned. See the list below.

answer is a pointer to a single answer or the start of an array of answers. Be sure to allocate enough space for the returing array of data. **HPE1432_GROUP_INFO_NUM_CHANNELS** can be used to determine how large an array is needed.

size is the maximum number of ellements allowed for the returning data.

The choices for item are;

HPE1432_GROUP_INFO_NUM_MODULES Number of modules

HPE1432_GROUP_INFO_LIST_MODULES List of modules

HPE1432_GROUP_INFO_NUM_CHANNELS Number of channels

HPE1432_GROUP_INFO_LIST_CHANNELS List of channels

HPE1432_GROUP_INFO_NUM_INPUTS Number of inputs

HPE1432_GROUP_INFO_LIST_INPUTS List of inputs

HPE1432_GROUP_INFO_NUM_SOURCES Number of sources

HPE1432_GROUP_INFO_LIST_SOURCES List of sources

HPE1432_GROUP_INFO_NUM_TACHS Number of tachs

HPE1432_GROUP_INFO_LIST_TACHS List of tachs

RESET VALUE

Not applicable

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_create_channel_group

NAME

e1432_get_hwconfig − Get module configuration

SYNOPSIS

```
SHORTSIZ16 e1432_get_hwconfig(SHORTSIZ16 modCount,
                              SHORTSIZ16 *logAddrList,
                              struct e1432_hwconfig *hwconfig)
```

DESCRIPTION

e1432_get_hwconfig returns information about the modules found at each logical address given. This function will return an error if any of the modules do not yet have firmware installed (firmware is installed by the *e1432_install* function). This means that the *e1432_get_hwconfig* function is a convenient way to test if *e1432_install* needs to be done. See the *e1432_install* manual page for example code of how to do this.

modCount specifies how many different modules to examine.

logAddrList is an array of VXI logical addresses. This array must be as long as *modCount*. Each logical address in this array will be examined.

hwconfig is an array of pointers to structures of type struct e1432_hwconfig. This array must be as long as *modCount*, and the pointers must point to already allocated structures. Each structure is filled in with information about the corresponding logical address from the *logAddrList* array. If any logical address does not contain an E1432 module, this function returns an error.

The *e1432_hwconfig* structure contains the following fields:

man_id, the VXI Manufacturer ID of the module. The Manufacturer ID for Hewlett-Packard is 0xfff.

model_code, the VXI Model Code of the module. The model code for E1432 is 0x201. The model code for E1433 is 0x202. The model code for E1434 is 0x203.

hw_rev, the hardware revision. The value is a four-decimal-digit date code.

bootrom_rev, the boot ROM revision. This is a number which represents the date of the boot ROM code. The number is of the form 19950721 (unless the boot ROM is very old, in which case the value is some number that doesn't look anything like a date code).

fw_rev, the firmware revision. This is a number which represents the date the firmware was created. The number is of the form 19950721.

sca_id, an array of IDs for the SCAs that are present. The value will be one of:

SCA ID Values	
Define (in e1432.h)	Meaning
E1432_SCA_ID_NONE	No SCA in this slot
E1432_SCA_ID_TACH	Option AYF Tachometer
E1432_SCA_ID_CLARINET	Option 1D4 Single-channel Source
E1432_SCA_ID_VIBRATO	E1432 51.2 kHz Input
E1432_SCA_ID_SONATA	E1433 196 kHz Input
E1432_SCA_ID_CLARION	E1434 Source

sca_rev, an array of revision codes for the SCAs that are present. The interpretation of this value depends upon the ID of the SCA. For a **E1432_SCA_ID_VIBRATO**, the value is a number between 0 and 5 (higher is more recent). For a **E1432_SCA_ID_SONATA**, the value is a number between 0 and 4 (higher is more recent). For a **E1432_SCA_ID_TACH**, a value of 0 means that the board does not have the ability

to do system triggering or monitor the analog tach input; a value of 1 means that the board can system trigger but can't monitor the analog tach input, and a value of 2 means that the board can system trigger and can monitor the analog tach input. All shipped tach boards should have a rev of 2 or higher.

bob_id, an array of IDs for any break-out boxes that may be connected to the SCAs. Possible values are:

Break-out Box ID Values	
Define (in e1432.h)	Meaning
E1432_BOB_ID_NONE	No smart break-out box detected
E1432_BOB_ID_CHARGE_PROTO	Prototype Charge/ICP break-out box
E1432_BOB_ID_CHARGE	Charge/ICP break-out box
E1432_BOB_ID_CHARGE2	B-version of Charge/ICP break-out box
E1432_BOB_ID_MIKE_PROTO	Prototype Microphone break-out box
E1432_BOB_ID_MIKE	Microphone break-out box
E1432_BOB_ID_MIKE2	B-version of Microphone break-out box

total_chans, the total number of input, source, and tach channels present in the module.

input_chans, the total number of input channels present in the module.

source_chans, the total number of source channels present in the module.

tach_chans, the total number of tach channels present in the module.

oct_present, which is non-zero if the module has the Octave option (1D1) installed.

lbus_present, which is non-zero if the module supports VXI local-bus transfers.

dram_size, which is the total size of DRAM, in terms of 32-bit words.

a24_used, which is the total amount of A24 space used by this module, in bytes. This value will either be 1MB or 256KB, depending on the VXI interface ROM present in the module.

serial, which is a string containing the serial number of the module.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_assign_channel_numbers, e1432_install

NAME

e1432_get_xxx_limits − Get parameter limits

SYNOPSIS

```
SHORTSIZ16 e1432_get_arm_time_interval_limits(E1432ID hw,
                                              SHORTSIZ16 ID,
                                              FLOATSIZ32 *min,
                                              FLOATSIZ32 *max,
                                              FLOATSIZ32 *def,
                                              FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_avg_number_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
                                       FLOATSIZ32 *max,
                                       FLOATSIZ32 *def,
                                       FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_avg_update_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
                                       FLOATSIZ32 *max,
                                       FLOATSIZ32 *def,
                                       FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_avg_weight_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
                                       FLOATSIZ32 *max,
                                       FLOATSIZ32 *def,
                                       FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_blocksize_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_cal_dac_limits(E1432ID hw,
                                    SHORTSIZ16 ID,
                                    FLOATSIZ32 *min,
                                    FLOATSIZ32 *max,
                                    FLOATSIZ32 *def,
                                    FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_cal_voltage_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_center_freq_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_clock_freq_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
```

```
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_decimation_undersamp_limits(E1432ID hw,
                                                 SHORTSIZ16 ID,
                                                 FLOATSIZ32 *min,
                                                 FLOATSIZ32 *max,
                                                 FLOATSIZ32 *def,
                                                 FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_delta_order_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_fifo_size_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_filter_settling_time_limits(E1432ID hw,
                                                 SHORTSIZ16 ID,
                                                 FLOATSIZ32 *min,
                                                 FLOATSIZ32 *max,
                                                 FLOATSIZ32 *def,
                                                 FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_internal_debug_limits(E1432ID hw,
                                           SHORTSIZ16 ID,
                                           FLOATSIZ32 *min,
                                           FLOATSIZ32 *max,
                                           FLOATSIZ32 *def,
                                           FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_interrupt_priority_limits(E1432ID hw,
                                               SHORTSIZ16 ID,
                                               FLOATSIZ32 *min,
                                               FLOATSIZ32 *max,
                                               FLOATSIZ32 *def,
                                               FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_max_order_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_meas_time_length_limits(E1432ID hw,
                                             SHORTSIZ16 ID,
                                             FLOATSIZ32 *min,
                                             FLOATSIZ32 *max,
                                             FLOATSIZ32 *def,
                                             FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_overlap_limits(E1432ID hw,
                                    SHORTSIZ16 ID,
                                    FLOATSIZ32 *min,
```

```
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_peak_decay_time_limits(E1432ID hw,
                                            SHORTSIZ16 ID,
                                            FLOATSIZ32 *min,
                                            FLOATSIZ32 *max,
                                            FLOATSIZ32 *def,
                                            FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_rms_avg_time_limits(E1432ID hw,
                                         SHORTSIZ16 ID,
                                         FLOATSIZ32 *min,
                                         FLOATSIZ32 *max,
                                         FLOATSIZ32 *def,
                                         FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_rms_decay_time_limits(E1432ID hw,
                                           SHORTSIZ16 ID,
                                           FLOATSIZ32 *min,
                                           FLOATSIZ32 *max,
                                           FLOATSIZ32 *def,
                                           FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_span_limits(E1432ID hw,
                                 SHORTSIZ16 ID,
                                 FLOATSIZ32 *min,
                                 FLOATSIZ32 *max,
                                 FLOATSIZ32 *def,
                                 FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_trigger_delay_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          FLOATSIZ32 *min,
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_triggers_per_arm_limits(E1432ID hw,
                                             SHORTSIZ16 ID,
                                             FLOATSIZ32 *min,
                                             FLOATSIZ32 *max,
                                             FLOATSIZ32 *def,
                                             FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_xfer_size_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_amp_scale_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_coupling_freq_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          FLOATSIZ32 *min,
```

```
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_duty_cycle_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
                                       FLOATSIZ32 *max,
                                       FLOATSIZ32 *def,
                                       FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_filter_freq_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_input_offset_limits(E1432ID hw,
                                         SHORTSIZ16 ID,
                                         FLOATSIZ32 *min,
                                         FLOATSIZ32 *max,
                                         FLOATSIZ32 *def,
                                         FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_pre_arm_rpm_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_ramp_rate_limits(E1432ID hw,
                                      SHORTSIZ16 ID,
                                      FLOATSIZ32 *min,
                                      FLOATSIZ32 *max,
                                      FLOATSIZ32 *def,
                                      FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_range_limits(E1432ID hw,
                                  SHORTSIZ16 ID,
                                  FLOATSIZ32 *min,
                                  FLOATSIZ32 *max,
                                  FLOATSIZ32 *def,
                                  FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_range_charge_limits(E1432ID hw,
                                         SHORTSIZ16 ID,
                                         FLOATSIZ32 *min,
                                         FLOATSIZ32 *max,
                                         FLOATSIZ32 *def,
                                         FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_range_mike_limits(E1432ID hw,
                                       SHORTSIZ16 ID,
                                       FLOATSIZ32 *min,
                                       FLOATSIZ32 *max,
                                       FLOATSIZ32 *def,
                                       FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_rpm_high_limits(E1432ID hw,
                                     SHORTSIZ16 ID,
                                     FLOATSIZ32 *min,
```

```
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_rpm_interval_limits(E1432ID hw,
                                            SHORTSIZ16 ID,
                                            FLOATSIZ32 *min,
                                            FLOATSIZ32 *max,
                                            FLOATSIZ32 *def,
                                            FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_rpm_low_limits(E1432ID hw,
                                        SHORTSIZ16 ID,
                                        FLOATSIZ32 *min,
                                        FLOATSIZ32 *max,
                                        FLOATSIZ32 *def,
                                        FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_rpm_smoothing_limits(E1432ID hw,
                                             SHORTSIZ16 ID,
                                             FLOATSIZ32 *min,
                                             FLOATSIZ32 *max,
                                             FLOATSIZ32 *def,
                                             FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_sine_freq_limits(E1432ID hw,
                                         SHORTSIZ16 ID,
                                         FLOATSIZ32 *min,
                                         FLOATSIZ32 *max,
                                         FLOATSIZ32 *def,
                                         FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_sine_phase_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          FLOATSIZ32 *min,
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_source_blocksize_limits(E1432ID hw,
                                                SHORTSIZ16 ID,
                                                FLOATSIZ32 *min,
                                                FLOATSIZ32 *max,
                                                FLOATSIZ32 *def,
                                                FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_source_centerfreq_limits(E1432ID hw,
                                                 SHORTSIZ16 ID,
                                                 FLOATSIZ32 *min,
                                                 FLOATSIZ32 *max,
                                                 FLOATSIZ32 *def,
                                                 FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_source_seed_limits(E1432ID hw,
                                           SHORTSIZ16 ID,
                                           FLOATSIZ32 *min,
                                           FLOATSIZ32 *max,
                                           FLOATSIZ32 *def,
                                           FLOATSIZ32 *step)
      SHORTSIZ16 e1432_get_source_span_limits(E1432ID hw,
                                           SHORTSIZ16 ID,
                                           FLOATSIZ32 *min,
```

```
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_srcbuffer_size_limits(E1432ID hw,
                                           SHORTSIZ16 ID,
                                           FLOATSIZ32 *min,
                                           FLOATSIZ32 *max,
                                           FLOATSIZ32 *def,
                                           FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_tach_decimate_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          FLOATSIZ32 *min,
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_tach_holdoff_limits(E1432ID hw,
                                         SHORTSIZ16 ID,
                                         FLOATSIZ32 *min,
                                         FLOATSIZ32 *max,
                                         FLOATSIZ32 *def,
                                         FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_tach_max_time_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          FLOATSIZ32 *min,
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_tach_ppr_limits(E1432ID hw,
                                     SHORTSIZ16 ID,
                                     FLOATSIZ32 *min,
                                     FLOATSIZ32 *max,
                                     FLOATSIZ32 *def,
                                     FLOATSIZ32 *step)
SHORTSIZ16 e1432_get_trigger_level_limits(E1432ID hw,
                                          SHORTSIZ16 ID,
                                          SHORTSIZ16 which,
                                          FLOATSIZ32 *min,
                                          FLOATSIZ32 *max,
                                          FLOATSIZ32 *def,
                                          FLOATSIZ32 *step)
```

DESCRIPTION

For every numerical parameter understood by the E1432 module, there is a corresponding *e1432_get_xxx_limits* function. This function can be used by an application to provide appropriate limits on a user-interface. In general, the limits returned by this function reflect the limits that the current hardware is capable of supporting. In cases where several parameters interact, the interaction is **not** accounted for when providing the limits.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If *ID* is a group ID, then all channels in that group must have the same parameter limits, or the function will return an error.

Some parameters are global for an entire module, other parameters are channel-specific. For module-wide parameters, the channel or channels specified by the *ID* parameter are used only to determine which E1432 module is being queried. In the above list of functions, *blocksize* through *triggers_per_arm* are module-wide parameters, while *amp_scale* through *trigger_level* are channel-specific.

min is a pointer to a 32-bit float which will be filled with the minimum valid value for the parameter.

max is a pointer to a 32-bit float which will be filled with the maximum valid value for the parameter.

def is a pointer to a 32-bit float which will be filled with the default (reset) value of the parameter.

step is a pointer to a 32-bit float which will be filled with the step-size between valid values of the parameter.

If the parameter can take on any floating-point value between the *min* and *max*, then *step* will be set to zero. If the parameter takes on discrete values that are evenly spaced, then *step* will be set to the spacing between valid values of the parameter. (For example, the interrupt priority can use any of the values 0, 1, 2, 3, 4, 5, 6, or 7, so the *step* is set to **1** in this case.) If the parameter takes on discrete values that are not evenly spaced, then the *step* is set to **-1**.

In some cases, there are no valid values for a parameter. This happens, for example, with *e1432_get_tach_holdoff_limits* when the channel specified is not a tach channel. In cases like this, the *min* is set larger than *max*, and the *step* is set to **-2**. The function does **not** return an error in this case.

In general, the limit values returned will depend on what SCAs are installed on the E1432, how much RAM is available on the E1432, and what hardware options are present.

All of the functions documented on this manual page take exactly the same form of parameters, except for *e1432_get_trigger_level_limits*. This function has an additional parameter *which*, which must be one of **E1432_TRIGGER_LEVEL_LOWER** or **E1432_TRIGGER_LEVEL_UPPER**.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_blocksize_current_max, e1432_get_fifo_size_current_max

NAME

e1432_get_meas_state − Get measurement state

SYNOPSIS

```
SHORTSIZ16 e1432_get_meas_state(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *state)
```

DESCRIPTION

e1432_get_meas_state gets the current state of the measurement loop that runs in an E1432 module.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If the ID is a group ID, and if the modules in the group are not all at the same measurement state, then this function tries to figure out the "lowest" state that they are all in, and returns that. When determining the "lowest" state, this function tries to take into account the fact that **IDLE** follows **CONVERT_WAIT2**, and tries to come up with the state of the module which is furthest behind in the measurement loop.

state is a pointer to a 16-bit int, and will be filled with the current measurement state. The measurement state will be one of:

E1432_MEAS_STATE_TESTED
E1432_MEAS_STATE_BOOTING
E1432_MEAS_STATE_BOOTING_WAIT1
E1432_MEAS_STATE_BOOTING_WAIT2
E1432_MEAS_STATE_BOOTED
E1432_MEAS_STATE_SYNC
E1432_MEAS_STATE_SYNC_WAIT1
E1432_MEAS_STATE_PRE_ARM
E1432_MEAS_STATE_PRE_ARM_WAIT2
E1432_MEAS_STATE_IDLE
E1432_MEAS_STATE_ARM
E1432_MEAS_STATE_ARM_WAIT1
E1432_MEAS_STATE_ARM_WAIT2
E1432_MEAS_STATE_TRIGGER
E1432_MEAS_STATE_CONVERT
E1432_MEAS_STATE_CONVERT_WAIT1
E1432_MEAS_STATE_CONVERT_WAIT2
E1432_MEAS_STATE_ERROR
E1432_MEAS_STATE_ERROR_WAIT1
E1432_MEAS_STATE_ERROR_WAIT2
E1432_MEAS_STATE_TPUT

The following is more details about the measurement state than you really wanted to know:

E1432_MEAS_STATE_TESTED

No measurement running. This state is what the module goes to at boot-up, and after e1432_reset_measure. Sync/Trig high, module waiting for it to go low to indicate start of measurement. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_BOOTING

Measurement starting, module doing setup and testing parameters for consistency. Sync/Trig held low.

E1432_MEAS_STATE_BOOTING_WAIT1

Module done doing setup, waiting for all SCAs to be ready for measurement sync. Sync/Trig held low.

E1432_MEAS_STATE_BOOTING_WAIT2

Module done doing setup, all SCAs ready for measurement sync. Module releases Sync/Trig line, and is waiting for the line to go high.

E1432_MEAS_STATE_BOOTED

Sync/Trig high, module waiting for it to go low to indicate the measurement sync. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_SYNC

Measurement sync just happened, all inputs start collecting data. Sync/Trig held low. Fall through to next state.

E1432_MEAS_STATE_SYNC_WAIT1

Module waiting for input digital filters to finish settling. Sync/Trig held low.

E1432_MEAS_STATE_PRE_ARM

Module done waiting for digital filter settling, now waiting for measurement pre-arm. Sync/Trig held low. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_PRE_ARM_WAIT2

Module done waiting for pre-arm. Module releases Sync/Trig line, and is waiting for the line to go high.

E1432_MEAS_STATE_IDLE

Sync/Trig high, module waiting for it to go low to indicate measurement arm. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_ARM

Measurement arm just happened. Sync/Trig held low. Fall through to next state. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_ARM_WAIT1

Module waiting for pre-trigger acquisition to complete, and for sources to be ready for trigger. Sync/Trig held low.

E1432_MEAS_STATE_ARM_WAIT2

Pre-trigger acquisition done, sources ready for trigger. Module releases Sync/Trig line, and is waiting for the line to go high.

E1432_MEAS_STATE_TRIGGER

Sync/Trig high, module waiting for it to go low to indicate measurement trigger. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_CONVERT

Measurement trigger just happened. Sync/Trig held low. Fall through to next state. A meas state change interrupt is generated when the module reaches this state from a different state.

E1432_MEAS_STATE_CONVERT_WAIT1

Module waiting for input data block to be acquired, and for source burst to complete. Sync/Trig held low. If the module is in continuous mode, it stays here until FIFO overflow. If the module is in overlap block or overlap free-run mode, it does not wait for input data block to be acquired.

E1432_MEAS_STATE_CONVERT_WAIT2

Previous state completed, Sync/Trig line released, module waiting for it to go high. The next state will be TESTED if the module was in continuous mode, or IDLE otherwise.

E1432_MEAS_STATE_DUMMY

Dummy state, never reached.

E1432_MEAS_STATE_ERROR **E1432_MEAS_STATE_ERROR_WAIT1**
E1432_MEAS_STATE_ERROR_WAIT2

Transitional states reached only briefly if an error happens during a measurement. The next state will be TESTED.

E1432_MEAS_STATE_DUMMY2

Dummy state, never reached.

E1432_MEAS_STATE_TPUT

Used only when doing thruput to the local bus, and at the same time sending data over VME (the data port parameter set to

E1432_SEND_PORT_LBUS_EAVES). If an RPM armed or time armed measurement is done, eventually the RPM armed or time armed measurement completes. However, this does not cause the time data that is sent to the local bus to stop. Instead, the local bus data continues, and the measurement state switches to **E1432_MEAS_STATE_TPUT**. The local bus data continues until the measurement is reset.

RESET VALUE

After a reset, the measurement state is **E1432_MEAS_STATE_TESTED**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure

NAME

e1432_get_octave_blocksize − Get current Octave data blocksize

SYNOPSIS

```
SHORTSIZ16 e1432_get_octave_blocksize(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 *octave_blocksize)
```

DESCRIPTION

e1432_get_octave_blocksize returns the current Octave data blocksize of the modules(s) selected into a memory location pointed to by *octave_blocksize*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to query.

Since the e1432_set_octave_start_freq and e1432_set_octave_stop_freq functions change the number of data points transferred by the e1432_read_XXXXXXX_data functions, the *e1432_get_octave_blocksize* function has been provided to supply the value needed for the *size* parameter of the e1432_read_XXXXXXX_data functions.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_current_data

NAME

e1432_get_raw_tachs − Read raw tach time data from a tachometer channel

SYNOPSIS

```
SHORTSIZ16 e1432_get_raw_tachs(E1432ID hw, SHORTSIZ16 ID,
                               unsigned long *buffer,
                               LONGSIZ32 size, LONGSIZ32 *actualCount)
```

DESCRIPTION

e1432_get_raw_tachs returns a block of raw tach times into the buffer pointed to by *buffer*. When the tach buffer on the E1432 is more than half full the **E1432_IRQ_TACHS_AVAIL** bit in the **E1432_IRQ_STATUS2_REG** register is set. This bit can be polled or interrupted on to know when to get the tach times

NOTE: This status bit will be set when **either** tach channel's buffer is over half full; so if both tach channels are active, the *e1432_get_raw_tachs* must be called for both to clear this bit. If a second tach channel's tach times are not going to be read, then this channel should be made to be inactive with *e1432_set_active*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID must be the ID of a single tachometer channel.

buffer is a pointer to the array for returned data.

size is the size, in data points, of *buffer*.

Note: always make this size less than or equal to the actual allocated memory for *buffer* or the function may overrun your *buffer*.

actualCount is a pointer to a long integer. It is set to the actual number of raw tach times transferred into *buffer*. It will always be less than or equal to *size* and may be zero, if no new tach times were accumulated, or there has been an overflow in the tach buffer internal to the tach daughter card. A tach overflow can be determined by a set **E1432_STATUS2_TACH_OVERFLOW** bit in the **E1432_IRQ_STATUS2_REG** register.

NOTE: If there has been a tach overflow, there still will be up to **E1432_TACH_RAW_SIZE** tach edge times stored in the module's raw tach buffer that are available for internal processing (i.e. order track arm points); so the **E1432_STATUS2_TACH_OVERFLOW** measurement error is not returned by the *e1432_block_available* function until all the stored tach edges are processed and the input data associated with the are transferred to the host.

Raw tach times are just latched values of a 32 bit counter driven by a tachometer clock. The frequency of this clock is obtained by a call to *e1432_get_tach_clock_freq*. The following snipet of code shows how to convert raw times into seconds, accounting for tach counter rollover:

```
  SHORTSIZ16 error, count;
  unsigned long buffer[BUF_SIZE];
  double tachTimes[BUF_SIZE];
long tachTimeLast = 0;
float tachFreq;
double tachTimeOffset = 0.0, tachTimeStep;

  e1432_get_tach_clock_freq(hw, ID, &tachFreq);
tachTimeStep = 1.0 / (double)tachFreq;
```

```
error = e1432_get_raw_tachs(hw, ID, buffer, BUF_SIZE, &count);
if(error) {
    printf("e1432_get_raw_tachs returned error = %d0, error);
    exit(-1);
}

/* convert raw tach times to seconds */
for(i = 0; i < count; i++)
{
    if(buf[i] < tachTimeLast) /* tach counter rollover has occured */
    {
      tachTimeOffset += E1432_TACH_WRAP_COUNT * tachTimeStep;
    }
    tachTimeLast = buf[i];
    tachTimes[i] = tachTimeOffset + (double)buf[i] * tachTimeStep;
}
```

NOTE: Once this function is called the first time either it or the function *e1432_send_tachs* **must be called regularly to read tach values out of the module. If tach times are not read out often enough, a tach buffer overflow will happen, which will overwrite the internal tach buffer and cause** *e1432_block_available* **to return ERR1432_TACH_BUFFER_OVERFLOW**.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_scale, e1432_set_append_status, e1432_set_data_size, e1432_set_data_port, e1432_send_tachs.

NAME

e1432_get_register_address − Get memory mapped address of E1432 register

SYNOPSIS

```
SHORTSIZ16 e1432_get_register_address(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 regOffset,
                                      volatile SHORTSIZ16 **addr)
```

DESCRIPTION

e1432_get_register_address returns the address of a register on a single E1432, identified by one of its channels, *ID*. This address can be used to directly access the register in question. **Note:** bus errors that occur when accessing a register from an address returned by this function are not trapped; so extreme care must be exercised when using the address returned.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a single channel.

regOffset is the offset of the register relative to the base address of the E1432. Register offsets have defines of the form **E1432_**<*register_name*>**_REG** in the file **e1432.h**. For instance, E1432_VXI_ID_REG is at register offset 0. The list of E1432 registers, and their detailed description, may be found in the hardware appendix.

addr is a pointer to a pointer to a SHORTSIZ16.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_read_register

NAME

e1432_get_samples_to_pre_arm − Get number of samples to pre_arm condition

SYNOPSIS

```
SHORTSIZ16 e1432_get_samples_to_pre_arm(E1432ID hw, SHORTSIZ16 ID,
                                        LONGSIZ32 *samples)
```

DESCRIPTION

e1432_get_samples_to_pre_arm returns the number of samples in the data FIFO from the first sample to the pre_arm condition. This information can be used to find the tachometer times of the first sample of a block of data.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

samples is the number of samples from the first sample to the pre_arm condition.

RESET VALUE

0

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_raw_tachs, e1432_get_tach_delay, e1432_get_append_status

NAME

e1432_get_scale − Calculate scale factor for current board settings

SYNOPSIS

```
SHORTSIZ16 e1432_get_scale(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ64 *scale)
```

DESCRIPTION

This function is not needed by most users. The typical application uses *e1432_read_float32_data* or *e1432_read_float64_data* to get input data from the E143x modules, and those functions already scale their results correctly to volts or picoCoulombs. In fact, those functions internally call *e1432_get_scale* to get the correct scale factor to use.

e1432_get_scale returns a scaling factor to convert raw input data into volts. The data from *e1432_read_raw_data* should be multiplied by this scale factor to get voltage (or picoCoulombs if the input channel is in charge mode). The scale factor accounts for break-out box settings, range, span, and multi-pass filtering effects. A call with group ID will return **ERR1432_PARAMETER_UNEQUAL** if all channels do not have the same scale factor. **NOTE: the scale factor can only by applied to raw time data .... frequency and order data are already scaled to volts internally.**

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group or single channel.

It is possible to set up an input channel to monitor the tach signal seen on a tach channel input. Normally, when this is done the tach channel drives CALOUT, the substrate connects CALOUT to CALIN, and the input channel monitors CALIN. (See *e1432_set_input_high* and *e1432_set_calin* for details.) The scale factor from *e1432_get_scale* will correctly account for the tach board and calibration scale factors.

It is also possible to set up a tach channel to drive the VXI sumbus, and have the sumbus drive CALIN, and have an input channel monitor CALIN. For this to work correctly, pass **E1432_CALIN_SUMBUS_TACH** (not **E1432_CALIN_SUMBUS**) to *e1432_set_calin*, to ensure that the input channel can figure out the correct scale factor.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_read_float32_data, e1432_read_float64_data, e1432_read_raw_data, e1432_set_data_size, e1432_set_input_high

NAME

e1432_get_tach_clock_freq − Get the tachometer channel clock frequency

SYNOPSIS

```
SHORTSIZ16 e1432_get_tach_clock_freq(E1432ID hw, SHORTSIZ16 ID,
                                     FLOATSIZ32 *freq)
```

DESCRIPTION

e1432_get_tach_clock_freq returns the clock frequency of the counters on the tachometer channels into the variable pointed to by *freq*. The tachometer option measures raw times of tach crossings by latching the value these 32 bit counters. The clock frequency returned by this function is used in conjunction with raw tach times obtained by a call to *e1432_get_raw_tachs* to calculate the times of these tach crossing in seconds.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is an ID on a module containing a tachometer channel.

freq is a pointer to a 32-bit float which will be filled in with the requested clock frequency.

RESET VALUE

After a reset, *freq* is set to 19,660,800 Hz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_raw_tachs

NAME

e1432_get_tach_delay − Get the delay time from sytem sync to first sample

SYNOPSIS

```
SHORTSIZ16 e1432_get_tach_delay(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 *delay)
```

DESCRIPTION

e1432_get_tach_delay returns the correction to the tach times such that they will be expressed relative to the time of the first sample in the data FIFO. The tachometer times are all relative to the time when the system SYNC line starts a measurement. Data from the input modules starts filling the data FIFO at some time after that because of ADC settling times and digital filter delays. Adding *delay* to the tach times calculated from the raw tach times returned by *e1432_get_raw_tachs* will yield tach times that are relative to the first sample of data in a measurement. Conversely, the negative of *delay* is the time between SYNC and the first sample of data.

NOTE: the first sample in the FIFO is not the same as the first sample in the first block. The first block of data starts after the pre_arm conditions are met. To find the number of samples in the FIFO prior to the pre_arm condition, call the function *e1432_get_samples_to_pre_arm*. The number of samples from pre_arm to the first sample in each block is found by summing the *gap* fields found in the data trailer of each block.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

delay is the desired correction in seconds to be added to tach times to make them relative to the first sample of data in the FIFO.

The following sample code shows how to calculate the tach time of the first sample in the first block of data:

```
float   delay, tach_correction, span;
long    pre_arm_samples, gap_sum = 0, count;
float   data[BLOCK_SIZE];
struct e1432_trailer trailer;

e1432_set_append_status(hw, ID, E1432_APPEND_STATUS_ON);

/* start measurement */

e1432_get_tach_delay(hw, ID, &delay);
e1432_get_samples_to_pre_arm(hw, ID, &pre_arm_samples);
e1432_get_span(hw, ID, &span);

/* wait for first data block */
while(e1432_block_available(hw, input) <= 0)
    ;

error = e1432_read_float32_data(hw, ID,
    E1432_TIME_DATA, data, BLOCK_SIZE, &trailer, &count);

/* samples from pre_arm to first in block */
if (zoom)
{
```

```
            gap_sum += trailer.gap / 2;
            tach_correction = delay -
                (float)(pre_arm_samples + gap_sum) / (1.28 * span);
        }
        else
        {
            gap_sum += trailer.gap;
            tach_correction = delay -
                (float)(pre_arm_samples + gap_sum) / (2.56 * span);
        }
```

RESET VALUE

0

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_raw_tachs, e1432_get_samples_to_pre_arm, e1432_get_append_status

NAME

e1432_get_trig_corr − Get trigger to sample delay correction

SYNOPSIS

```
SHORTSIZ16 e1432_get_trig_corr(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 *trig_corr)
```

DESCRIPTION

e1432_get_trig_corr gets the trigger to sample delay correction for the current triggered measurement. This delay is from the trigger event to the next sample point. It includes both the delay from trigger to the next sample clock and the delay from trigger to the next decimated output point. It is normalized such that 1.0 is equivalent to one (decimated) sample interval.

The delay returned by *e1432_get_trig_corr* is not absolute. Other, fixed delays are present, which depend on setup conditions. The delay returned by this function is suitable for correcting time record position such that the trigger event is accurately repeatable from sample to sample. One application of this function is correcting data prior to time averaging.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

trig_corr is a pointer to 32-bit float, and will be filled with the current trigger to sample correction value. This is the same parameter as the trig_corr entry in the trailer.

RESET VALUE

The value of *delay* is not valid until after a triggered measurement.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_append_status

NAME

e1432_get_meas_warning − get warnings from the measurement

SYNOPSIS

```
SHORTSIZ16 e1432_get_meas_warning(E1432ID hw, SHORTSIZ16 ID,
            SHORTSIZ16 *warning, unsigned long size, LONGSIZ32 *actualCount)
```

DESCRIPTION

e1432_get_meas_warning returns zero or more warning codes from a measurement running in the E1432 module into a user buffer. Warnings are considered less serious than errors, which will stop a measurement. Currently the only measurements which return warnings are rpm arming and Order Tracking measurements. The warnings are defined in *err1432.h*. Warning messages can be obtained using *e1432_get_warning_string*. Warnings are currently issued when arming points are missed or lost because of the following reasons:

WARN1432_LOST_NOT_ENOUGH_TACHS is issued when there is not enough tach edges present to do the resampling calculation.

WARN1432_LOST_TOO_MANY_TACHS is issued when the tach pulses are coming too fast to accomodate. The internal tach time buffers have wrapped and erased some of the tach times needed for the resampling calculation.

WARN1432_LOST_TOO_MANY_POINTS_REQUIRED is issued when the number of data points needed for a resampling calculation is greater than the size of the channel's span buffer in the FIFO.

WARN1432_LOST_DATA_SHIFTED_OUT_FIFO is issued when the measurement is falling behind and the data needed in the FIFO has already been overwritten by newer data before the FIFO could be stopped.

WARN1432_LOST_NOT_ENOUGH_DATA_FIFO is issued when there is not enough data in the span buffer in the FIFO to do the resampling calculation.

WARN1432_LOST_RPM_TOO_HIGH is issued when the RPM calculated from the tach input is greater than maximum allowed for the top span of data in the FIFO. This maximum is (top_span / max_order) * 60.0.

WARN1432_LOST_RPM_TOO_LOW is issued when the RPM calculated from the tach input is lower than minimum allowed for the bottom span of data in the FIFO. This minimum is (bottom_span / max_order) * 60.0.

WARN1432_LOST_RESAMPLE_ERROR is issued when an internal computational error occurs in the resampling calculation. This should not happen.

WARN1432_LOST_FIFO_EMPTIED is issued when a stopped FIFO has finished emptying it's data. The FIFO is stopped when the addition of new data would overwrite data needed for already queued arm points.

WARN1432_RPM_RAMP_TO_FAST is issued when the tach RPM changes to fast to be able to correctly resample the input data into the order domain.

A bit defined by **E1432_IRQ_MEAS_WARNING** is set in the status register, **E1432_IRQ_STATUS2_REG** when there are warnings available. This bit can be interrupted on.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is a group ID of the channels involved in the measurement.

warning is a pointer to an array into which the warning codes are placed.

size is the size, in elements, of *warning*.

Note: always make this size less than or equal to the actual allocated memory for *warning* or the function may overrun the array.

actualCount is a pointer to a long integer. It is set to the actual number of warning codes transferred into *warning*. It will always be less than or equal to *size* and may be zero, if no new warnings were accumulated.

The following code illustrates how to query for errors, warnings and end of measurment while looking for a block available:

```
#define WARNING_MAX     100

      /* Wait for block available, checking for errors, warnings and end */
      do
      {
          e1432_get_meas_state(hw, inputs, &meas_state);
          if(meas_state == E1432_MEAS_STATE_TESTED)
          {
            printf("Measurement finished.0);
            exit(0);
          }

            e1432_read_register(hw, inputs, E1432_IRQ_STATUS2_REG, &status);
          if(status & E1432_IRQ_MEAS_ERROR)
          {

              if(status & E1432_STATUS2_TACH_OVERFLOW)
                (void) printf("Tach buffer overflowed0);
            else
                (void) printf("Fifo overflowed0);
            exit(-1);
          }

          if(status & E1432_IRQ_MEAS_WARNING)
          {
            /* read out all measurement warnings */
            while(status & E1432_IRQ_MEAS_WARNING)
            {
                e1432_get_meas_warning(hw, inputs, warning, WARNING_MAX,
                                                  &warningCount);

                if(warningCount)
                {
                        printf("%d Warning", warningCount);
                  if(warningCount > 1) printf("s");
                  printf(":0);
                }
```

```
                    for(i=0; i < warningCount; i++)
                      printf("   %s0, e1432_get_warning_string(warning[i]);

                    e1432_read_register(hw, inputs,
                                 E1432_IRQ_STATUS2_REG, &status);
                  }
                }


            }while((status & E1432_IRQ_BLOCK_READY) == 0);
```

RESET VALUE

Not applicable.

RETURN VALUE

Returns 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_warning_string

NAME

e1432_get_warning_string − get string associated with a measurement warning

SYNOPSIS

```
char *e1432_get_warning_string(SHORTSIZ16 warning)
```

DESCRIPTION

e1432_get_warning_string returns a string that is a brief description of the measurement warning, *warning*, which has been returned by the *e1432_get_meas_warning* function.

warning is a warning number.

RESET VALUE

Not applicable.

RETURN VALUE

Returns a string.

SEE ALSO

e1432_get_meas_warning

NAME

e1432_init_io_driver − Initialize I/O driver
e1432_set_interface_addr − Set SICL interface name
e1432_uninit_io_driver − Close SICL files and free all memory

SYNOPSIS

```
SHORTSIZ16 e1432_init_io_driver(void)
SHORTSIZ16 e1432_set_interface_addr(const char *name)
SHORTSIZ16 e1432_uninit_io_driver(void)
```

DESCRIPTION

e1432_init_io_driver must be the first routine called when using the E1432 library. It performs whatever initialization the I/O driver (for example, SICL) needs for the environment in which this library is running.

e1432_set_interface_addr is the one function which can be called before *e1432_init_io_driver*. Call *e1432_set_interface_addr* before *e1432_init_io_driver* to change the SICL interface name from the default of "vxi" to some other interface name. See your SICL documentation for more information. SICL is the 'Standard Instrument Interface Library' that the E1432 interface library calls to communicate with the E1432 hardware.

name specifies the SICL interface name. If this name is longer than **E1432_SICL_NAME_MAX** - 1, it is silently truncated to that length. If *name* is specified as the empty string, then the default name ("vxi") will be used.

e1432_uninit_io_driver is the inverse of *e1432_init_io_driver*. It frees all memory that the library has allocated internally, unmaps VXI shared memory windows, and closes all file descriptors that the library openned. After this call, no other calls to the E1432 host interface library should be made (except for another *e1432_init_io_driver* to re-open the library).

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise

SEE ALSO

e1432_assign_channel_numbers

NAME

e1432_init_measure − Initialize measurement, move E1432s to IDLE state

SYNOPSIS

```
SHORTSIZ16 e1432_init_measure(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_init_measure_to_booted(E1432ID hw, SHORTSIZ16 ID,
                                        SHORTSIZ16 wait_after)
SHORTSIZ16 e1432_init_measure_finish(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 wait_after)
```

DESCRIPTION

e1432_init_measure places all modules in the group into the idle state. The modules can be in any state before the call so *e1432_init_measure* can be used to abort a current measurement.

After the call to *e1432_init_measure* completes successfully, the E1432 will ready for arming and triggering (see *e1432_arm_measure* and *e1432_trigger_measure*). If auto arm or auto trigger are on, the hardware may proceed beyond the idle state without further intervention.

Overload flags are reset by *e1432_init_measure*, see *e1432_check_overloads*.

e1432_init_measure_to_booted is for the special case of the library user wanting to initialize the modules half way, (to booted state), performing some communications with other hardware on the VXI bus, and then finishing the measurement initialization with *e1432_init_measure_finish*. Most users will need only the *e1432_init_measure* to perform data collection with the E1432.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

wait_after is used only with *e1432_init_measure_to_booted* and *e1432_init_measure_finish*. If this is set to non-zero, the function will not return until the module has completed moving to the **BOOTED** state (for *e1432_init_measure_to_booted*) or the **IDLE** state (for *e1432_init_measure_finish*). If it is zero, then the functions will return as soon as they are done requesting that the measurement state move, not waiting for the module to actually complete moving to the next measurement state.

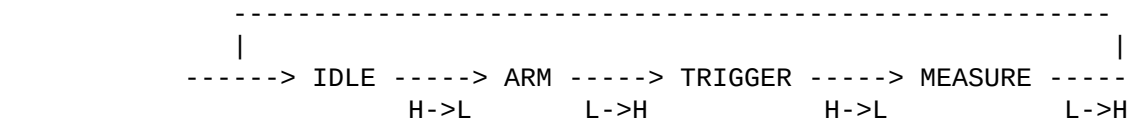
The measurement itself consists of two phases, the measurement initialization, and the measurement loop. Each of these phases consists of several states, through which the measurement progresses. The transition from one state to the next is tied to a transition in the SYNC line (one of the TTLTRG lines on the VXI backplane). This SYNC line is "wired-ORed" such that all E1432s in a multiple module system must release it for it to go high. Only one E1432 is required to pull the SYNC line low. In a single E1432 system, the SYNC line is local to the module and is not tied into a TTLTRG line on the VXI backplane.

The measurement initialization states, and the corresponding SYNC line transitions (with 'H' for high, 'L' for Low) are:

```
TESTED ---> BOOTING ---> BOOTED ---> SYNC ---> PRE_ARM ---> IDLE
       H->L         L->H        H->L                       L->H
```

This complete measurement sequence initialization, from **TESTED**, through **BOOTING**, **BOOTED**, **SYNC**, **PRE_ARM**, and ultimately **IDLE**, can be performed with a call to the function *e1432_init_measure*.

The progression of measurement loop states and the corresponding SYNC line transitions are:



In the **IDLE** state the E1432 clears the FIFO An E1432 remains in the **IDLE** state until it sees a high to low transition of the SYNC line. If any of the E1432 is programmed for auto arming, with *e1432_set_auto_arm*, the SYNC line immediately goes low, exiting the **IDLE** state for the ARM state. The E1432 may also be moved to the ARM state by an explicit call to the function, *e1432_arm_measure*.

Upon entering the **ARM** state, the E1432 starts saving new data in it's FIFO. It remains in the **ARM** state until the SYNC signal goes high. If an E1432 is programmed with a pre-trigger delay, it collects enough data samples to satify this pre-trigger delay, and then releases the SYNC line. If no pre-trigger delay has been programmed, the SYNC line goes high immediately. When all E1432s in a system have released the SYNC line allowing it to go high, a transition to the **TRIGGER** state occurs.

Upon entering the **TRIGGER** state an E1432 continues collecting data into the FIFO, discarding any data prior to the pre-trigger delay. An E1432 remains in the **TRIGGER** state until it sees a high to low transition of the SYNC line. The SYNC line is pulled low by any E1432 which encounters a trigger condition and is programmed to pull the SYNC line. If any E1432 is programmed for auto triggering, with *e1432_set_auto_trigger*, the SYNC line goes low immediately, exiting the **TRIGGER** state. The SYNC line may also be pulled low by an explicit call to the function, *e1432_trigger_measure*.

Upon entering the **MEASURE** state an E1432 continues to collect data. The E1432 also presents the first data from the FIFO to the selected output port, making it available to the controller to read. The E1432 holds the SYNC line low as long as it it actively collecting data.

Note: When in the one of the RPM arming/triggering modes in a multiple module system, the slave modules move from state to state in a different manner. They change states in response to a command passed from the master through the host computer. See *e1432_set_arm_mode* for details of this behavior.

In block mode the E1432 module stops taking data as soon as a block of data has been collected, including any programmed pre or post trigger delays. In continuous mode, the E1432 stops taking data when the FIFO overflows. After the FIFO overflow, the E1432 will remain in the **MEASURE** state, not taking data, until the FIFO is emptied, and then it will move to the **TESTED** state. For more information about data modes, see the *e1432_set_data_mode* manual page.

A channel group that spans more than one module will need to be configured to use the TTLTRG trigger lines on the VXI bus for inter-module communication. This configuration is automatically performed in the *e1432_init_measure* call unless defeated using *e1432_set_auto_group_meas. See the e1432_set_auto_group_meas* manual page for details on what setups are done automatically.

There are 4 pairs of VXIBUS TTLTRG lines that can be used for multi-module synchronization. The pair is selected using the *e1432_set_ttltrg_clock* and *e1432_set_ttltrg_satrg* (be sure to use a group ID for those functions, to ensure that all modules in the group are set to the same two TTLTRG lines).

After a multi-module measurement has completed, the modules are left connected to the VXI TTLTRG lines. This helps subsequent multi-module measurements start more quickly. However, it can also cause problems if a module (call it module 'A') is driving the TTL trigger lines and a different group is started which also drives the TTLTRG lines, and that different group does not include module 'A'. Module 'A' will conflict and prevent the different group from functioning properly. In this case a call to *e1432_finish_measure* for module 'A' will disconnect module 'A' from the TTLTRG lines, allowing the

new group to function properly.

Note that if the new group includes all modules of the old group, the conflict will not occur since *e1432_init_measure* will reset all modules as needed. Also note that single module groups do not drive the TTLTRG lines, so single modules groups are immune from causing or receiving this conflict.

At the start of a measurement that uses local bus, it is important to properly reset the local bus before starting the measurement. Before resetting the local bus, make sure all E1432 modules are no longer running any previous measurement, by calling *e1432_reset_measure*. Then the local bus interface on all VXI cards must be reset, and then they must all be un-reset. The order in which the cards must be reset and un-reset is important. After all that, the measurement can be started with *e1432_init_measure*.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_arm_measure, e1432_trigger_measure, e1432_finish_measure, e1432_reset_measure, e1432_check_overloads, e1432_reset_lbus, e1432_set_auto_group_meas, e1432_set_ttltrg_clock, e1432_set_ttltrg_satrg

NAME

e1432_init_measure_master_finish − Master side measurement init e1432_init_measure_master_setup − Master side measurement init e1432_init_measure_slave_finish − Slave side measurement init e1432_init_measure_slave_middle − Slave side measurement init e1432_init_measure_slave_setup − Slave side measurement init

SYNOPSIS

```
SHORTSIZ16 e1432_init_measure_master_finish(E1432ID hw, SHORTSIZ16 ID,
                                            SHORTSIZ16 wait_after)
SHORTSIZ16 e1432_init_measure_master_setup(E1432ID hw, SHORTSIZ16 ID,
                                           SHORTSIZ16 wait_after)
SHORTSIZ16 e1432_init_measure_slave_finish(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_init_measure_slave_middle(E1432ID hw, SHORTSIZ16 ID)
SHORTSIZ16 e1432_init_measure_slave_setup(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

These functions are *not* normally needed by the typical application. They are provided for use in special circumstances involving multiple VXI mainframes and multiple processes running the measurements in those mainframes. The typical application should simply use *e1432_init_measure* instead.

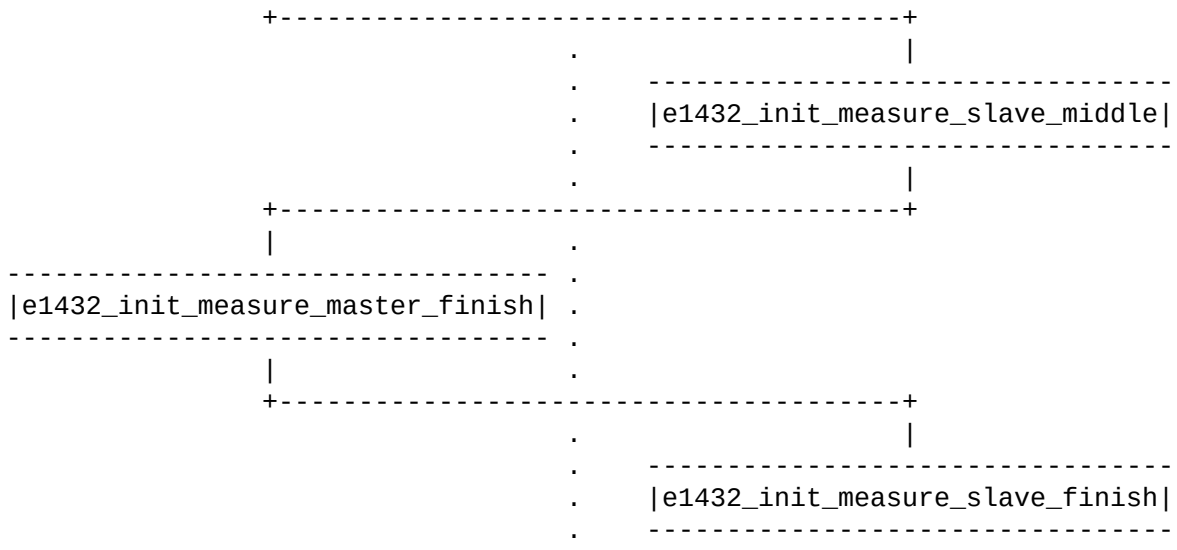
These functions take the place of calling the (much simpler) *e1432_init_measure* function. Please see the documentation on *e1432_init_measure* for more detail on what happens when a measurement is started.

The five functions *e1432_init_measure_slave_setup*, *_slave_middle*, *_slave_finish*, *_master_setup*, and *_master_finish* are used for making measurements for the special case of multiple groups of modules sharing common TTLTRG lines for clocking and sync.

These groups will each run in a separate UNIX (NT?) process (see below if you care why).

The application program must establish interprocess communications to allow these five functions to be called in the following order.

One process controlling one of the module groups must be designated as the "Master". This should be the group that is at the head of the TTL trigger line routing. VXI mainframe extenders currently have a one way routing of TTL lines between mainframes so it is important to watch directions and which group is driving TTL sync.



Why is this so complex? Why use separate processes? Normally you wouldn't. But some applications might want to deal with separate channel groups using separate processes, and still provide for all of the modules in all of the groups running synchronously. This is what is needed to deal with that.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_arm_measure_master_finish, e1432_set_mmf_delay

NAME

e1432_install − Install firmware into E1432 RAM

SYNOPSIS

```
SHORTSIZ16 e1432_install(SHORTSIZ16 modCount, SHORTSIZ16 *logAddrList,
                         LONGSIZ32 flags, const void *path);
```

DESCRIPTION

e1432_install installs firmware into E1432 RAM. This must be done after powerup or a hard reset. Until this is done, the module is unable to run a measurement or set or query parameter values.

modCount specifies how many different modules to install firmware into. The installation is done in parallel as much as possible, so it is faster to call *e1432_install* and give it several modules, rather than call *e1432_install* once for each module.

logAddrList is an array of VXI logical addresses. This array must be as long as *modCount*. Firmware will be installed in each E1432 module specified by this array.

flags is used to modify the installation process. This value is a bit field, with each bit signifying something different. Normally, *flags* should be specified as zero. Values that are understood are:

E1432_INSTALL_FASTEST This performs no handshaking while downloading the firmware. *This fails except when the E1406 command module is being used*.

E1432_INSTALL_FASTER This downloads the first 4K block of firmware with handshaking, and then uses no handshaking for the rest of the firmware.

E1432_INSTALL_FAST This is similar to **E1432_INSTALL_FASTER**, but it uses a slower transfer method for non-handshaked transfers.

E1432_INSTALL_FROM_MEM
Normally, *e1432_install* installs data from a file into the E1432 module. If this bit is set, then instead *e1432_install* installs data from a passed-in memory buffer specified by the *path* parameter. See the "DOWNLOADING FROM A BUFFER" section below.

E1432_INSTALL_SYSCALLS This specifies that the *e1432mon* program is already running, and the installed firmware should wait for the monitor to acknowledge each debug print (and all other system calls).

None of the **FAST** flags above will work over a fast interface, such as MXI or embedded V/743.

path specifies the name of a file that contains the firmware to download (except when **E1432_INSTALL_FROM_MEM** is set in the *flags* parameter; see section "DOWNLOADING FROM A BUFFER" below). Normally, this path should be set to `/opt/e1432/lib/sema.bin`. This binary file is an exact image of the firmware that will execute in the E1432 substrate 96002.

DOWNLOADING FROM A BUFFER

Normally, *e1432_install* opens a file and downloads the contents of the file to the E1432. This is generally the easiest and most efficient way to perform the download. However, in special circumstances (for example, when controlling an E1432 from an E1485), it may be inconvenient to use a file. To allow for this possibility, the **E1432_INSTALL_FROM_MEM** bit in the *flags* parameter may be used. This bit indicates that a file will not be used for downloading, and that the *path* parameter is not a filename but is instead a pointer to a structure of type *e1432_install_from_mem*. This structure has two fields: an *nbyte* field which

specifies the number of bytes to install, and a *data* field which is a pointer to the actual data.

It is up to the application to set these two fields in the structure pointed to by *path*. In the case of an E1485 program, the contents of `/opt/e1432/lib/sema.bin` could be converted to ASCII, embedded into the source code of the E1485 program, and *data* could be set to point to this embedded array. Note that this would make the E1485 program quite large.

IS DOWNLOADING NECESSARY?

The process of downloading code into the E1432 module usually takes more than five seconds. Since it is so slow, an application programmer generally wants to download code only when absolutely necessary. The best way to tell if the download is necessary is to use the *e1432_get_hwconfig* function. If *e1432_get_hwconfig* returns an error, then downloading is necessary. If *e1432_get_hwconfig* is successful, then downloading is generally not necessary. The structure returned by a successful call to *e1432_get_hwconfig* contains a field that specifies the firmware revision of the code running in the E1432 module.

Here is a typical code fragment to deal with downloading, which will download code into the E1432 only if needed:

```
struct e1432_hwconfig hwconfig;
SHORTSIZ16 laddr[E1432_MOD_MAX];
SHORTSIZ16 nmod;

nmod = 1;
laddr[0] = 8;

if (e1432_print_errors(0) < 0)
    return -1;
status = e1432_get_hwconfig(nmod, laddr, hwconfig);
if (e1432_print_errors(1) < 0)
    return -1;
if (status < 0)
    if (e1432_install(nmod, laddr, 0, "/opt/e1432/lib/sema.bin") < 0)
        return -1;
```

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_assign_channel_numbers, e1432_get_hwconfig, e1432_init_io_driver

NAME

e1432_install_file − Specify the location of the downloadable

SYNOPSIS

```
SHORTSIZ16 e1432_install_file(const char *install_file, SHORTSIZ16 from_mem)
```

DESCRIPTION

e1432_install_file informs the library of the location of the 96000 downloadable firmware file.

install_file must point to a string which is the location of the 96000 downloadable firmware file, such as `/opt/e1432/lib/sema.bin`. The library also looks in the same directory for the Octave SCA DSP downloadable, `soct.bin`, so this function is useful for specifying its location. *install_file* is later used as the location of the 96000 downloadable firmware file when the *e1432_install* function is called the *path* string pointer is set to null.

install_file is also set to the value of *path* when *e1432_install* is called with *path* a valid string.

from_mem should be set to 0. Behavior is undefined otherwise.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_meas, e1432_install

NAME

e1432_octave_ctl − Send Octave measurement control command

SYNOPSIS

```
SHORTSIZ16 e1432_octave_ctl(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 cmd)
```

DESCRIPTION

e1432_octave_ctl sends an Octave measurement command to the modules(s) selected.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to send the command to.

cmd must be one of **E1432_OCTAVE_CTL_STOP**, **E1432_OCTAVE_CTL_RESTART** or **E1432_OCTAVE_CTL_CONTINUE**.

e1432_octave_ctl is currently inactive.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_get_octave_blocksize, e1432_get_current_data

NAME

e1432_pre_arm_measure − Manually move E1432s from PRE_ARM to IDLE state

SYNOPSIS

```
SHORTSIZ16 e1432_pre_arm_measure(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 wait_after)
```

DESCRIPTION

e1432_pre_arm_measure moves all modules in the group from the **PRE_ARM** state to the **IDLE** state.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

This function performs a "manual pre_arm", and does not need to be called unless the group's pre-arm mode is E1432_MANUAL_ARM. This function is called after *e1432_init_measure*. This function waits for all modules to be in the **PRE_ARM** state, before proceeding further, and it will return an error if this state is not reached after a limited time.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If the measurement involves more than one module, it is mandatory that a *group ID* be used, rather than a *channel ID*.

wait_after determines whether this function will wait for the module to actually move beyond the **PRE_ARM** state. If zero, the function does not wait; if non-zero, the function waits.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_set_pre_arm_mode, e1432_arm_measure

NAME

e1432_preset − Preset parameters to default values

SYNOPSIS

```
SHORTSIZ16 e1432_preset(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_preset sets a single channel or a group of channels back to initial default values.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID identifies a group of channels or a single channel to be preset to initial values.

This function is called by *e1432_assign_channel_numbers* to preset all parameters to their default values.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_assign_channel_numbers

NAME

e1432_print_errors − Enable/disable function error printout

SYNOPSIS

```
SHORTSIZ16 e1432_print_errors(SHORTSIZ16 enable)
```

DESCRIPTION

e1432_print_errors enables/disables the printing of error messages when any function returns an error.

If the library is used in a host computer environment, the errors are output to stderr (normally the console screen). If the library is used in the E1485A environment, the error messages are output to a terminal connected to the RS-232 port available on this module.

It is normal while developing code to include a call to *e1432_print_errors* enabling error printing early in the code. Once the code has been fully debugged, you may choose to remove this call.

enable is a boolean, with the value being either nonzero (to enable printing) or zero (to disable printing).

RESET VALUE

After a reset, *enable* is set to **0**. Error printing is disabled.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_trace_level

NAME

e1432_read_raw_data − Read raw data from E1432 channel
e1432_read_float32_data − Read scaled float data from E1432 channel
e1432_read_float64_data − Read scaled float data from E1432 channel

SYNOPSIS

```
SHORTSIZ16 e1432_read_raw_data(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 which, void *buffer,
                               LONGSIZ32 size,
                               struct e1432_trailer *trailer,
                               LONGSIZ32 *actualCount)
SHORTSIZ16 e1432_read_float32_data(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 which, FLOATSIZ32 *buffer,
                                   LONGSIZ32 size,
                                   struct e1432_trailer *trailer,
                                   LONGSIZ32 *actualCount)
SHORTSIZ16 e1432_read_float64_data(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 which, FLOATSIZ64 *buffer,
                                   LONGSIZ32 size,
                                   struct e1432_trailer *trailer,
                                   LONGSIZ32 *actualCount)
```

DESCRIPTION

e1432_read_raw_data returns a block of raw, *UN*-scaled data. *buffer* should be either a pointer to SHORTSIZ16, LONGSIZ32, or FLOATSIZ32, depending upon the setting of the data size (see *e1432_set_data_size*).

e1432_read_float32_data returns a block of 32 bit floating point data. The data is properly scaled to volts (or picoCoulombs if the input channel is in charge mode).

e1432_read_float64_data returns a block of 64 bit floating point data. The data is properly scaled to volts (or picoCoulombs if the input channel is in charge mode).

e1432_read_raw_data, *e1432_read_float32_data* and *e1432_read_float64_data* can be called with either a group ID or a channel ID. If called with a group ID the buffer must be large enough to hold all data from all channels in the group. These functions may return **ERR1432_BUS_ERROR** if there is no data ready, so use *e1432_block_available* to determine data readiness before calling a read data function. These functions will attempt to read one blocksize worth of data as set by *e1432_set_blocksize*.

If channel IDs are used, any of these functions has to be called as many times as there are active channels in the group for which the data acquisition has just been performed. Also, the channels MUST be called in order of the channel list used to create the group. The channel ID does not really specify which channel's data to read - it just indicates which module to read from, and the module sends the next active channel. Each channel must be read completely before the next channel data is available. All these restrictions are dealt with by using a group ID instead of channel ID.

These data transfers are performed using the VME bus. If the data port is set to Local Bus, there is no need to read data via VME, and these functions should not be used at all.

If *e1432_set_append_status* has been used to turn on the status trailer, then a trailer is read from the module after reading the block of data. There are two ways to get this trailer data.

One way is to have the trailer appended to the end of each data block. If this is done, then each block of data will have a trailer of eight 32-bit words appended to it. The data buffer must be large enough to accomodate the additional data. The *size* that is passed to this function should be either 8 or 16 larger, for each channel, than it would otherwise be. If the current data size (as selected by *e1432_set_data_size*) is

E1432_DATA_SIZE_16, then 16 should be added to the size. Otherwise, 8 should be added to the size. This will ensure that the correct amount of data is read after each data block. The reason that the number varies is that the trailer information is always in the same format, regardless of the current data size setting.

The other way to get trailer data is to use the *trailer* parameter. If this is done, then the *size* should NOT try to account for the additional trailer data. The *trailer* parameter should point to an array of *e1432_trailer* structures, one for each channel whose data will be read by this function.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a group or single channel.

which specifies what type of data to read. **E1432_TIME_DATA** causes time data to be read.

E1432_FREQ_DATA causes the results of an FFT of the time data to be read.

E1432_RESAMP_DATA causes resampled time data to be read. For this data to be available, the resampling calculation must be enabled by e1432_set_calc_data.

E1432_OCTAVE_DATA causes Octave date to be read. For this data to be available, Octave measurements must be turned on vi *e1432_set_octave_mode*. **E1432_OCTAVE_DATA** is not valid for *e1432_read_raw_data* calls. Octave data is in mean-squared form. A square root must be taken to make it RMS.

E1432_ORDER_DATA causes the results of an FFT of the resampled time data (order data) to be read. For this data to be available, the order calculation must be enabled by e1432_set_calc_data.

NOTE: The frequency data and order data are normally sent up as complex data, real and imaginary pairs for each point. Since the FFT is a Real Valued Transform (RVT) on *blocksize* points of input time data (or resampled time data in the case of order calculations), the result normally would have one extra point in it (*blocksize*/2 + 1), called the Fs/2 point which is necessary if an inverse FFT is to be done in the host. To save buffer space in the E1432, the real part of the Fs/2 point is put into the imaginary part of the DC frequency bin (first point in the block). This is possible because the imaginary parts of the DC point the Fs/2 point are zero. The result of this packing is *blocksize*/2 complex pairs, or *blocksize* points. When one of the averaging modes is turned on with *e1432_set_avg_mode* that cause only magnitude squared data to be passed up, the resulting number of points is *blocksize*/2. The Fs/2 information is lost with this type of averaging.

buffer is a pointer to the array for returned data.

size is the size, in data points, of *buffer*. If *size* is less than the block size set with *e1432_set_blocksize*, some channel data will be left on the E1432 and will corrupt future reads.

Note: always make this size less than or equal to the actual allocated memory for *buffer* or the function may overrun your *buffer*.

trailer is a pointer to a structure of type *struct e1432_trailer*. This parameter is ignored unless append status is on (see *e1432_set_append_status*). If append status is on, and if this parameter is non-NULL, and if the size is equal to the *blocksize* as specified by *e1432_set_blocksize*, then the *trailer* structure is filled in with trailer data. If the *ID* is a group ID, then *trailer* must point to an array of structures, one for each channel in the group.

actualCount is a pointer to a long integer. It is set to the actual number of data points transferred into

buffer. It will always be less than or equal to *size*.

Data Transfer Sizes		
Function	Data size	Bytes per data point
e1432_read_raw_data	16	2
e1432_read_raw_data	32	4
e1432_read_float32_data	any	4
e1432_read_float64_data	any	8

Data size is set using *e1432_set_data_size*.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_scale, e1432_set_append_status, e1432_set_data_size, e1432_set_data_port

NAME

e1432_read_i2c − Read from input I2C port
e1432_write_i2c − Write to input I2C port

SYNOPSIS

```
SHORTSIZ16 e1432_read_i2c(E1432ID hw, SHORTSIZ16 ID,
                          SHORTSIZ16 addr, SHORTSIZ16 *value)
SHORTSIZ16 e1432_write_i2c(E1432ID hw, SHORTSIZ16 ID,
                           SHORTSIZ16 addr, SHORTSIZ16 value)
```

DESCRIPTION

This is a low-level function which will not normally be needed by an application programmer. These functions are not necessary for the E3242A Charge Break-Out Box, nor for the E3243A Microphone Break-Out Box.

To use the input channel on an E1432 module, a cable must connect a "break-out" box to the small connectors on the front panel of the E1432. Several break-out boxes are available. Some are passive, and provide only a direct connection from a BNC connector to the E1432 input channel. Some are "smart", and can perform some analog signal conditioning before sending the signal to the E1432 input channel.

The smarter break-out boxes use I2C to communicate setup information between the E1432 module and the break-out box. For the most part, the E1432 module has built-in knowledge of how to talk to a break-out box, and will encapsulate the functionality of the break-out box. However, future break-out boxes might provide functionality that the E1432 module is unaware of. In this case, it may be useful for a host program to directly program the break-out box by reading and writing on the I2C bus. *e1432_read_i2c* and *e1432_write_i2c* provide this capability.

e1432_read_i2c will read any I2C address on the I2C bus attached to a particular SCA (the *ID* identifies which SCA).

e1432_write_i2c will write any value to any I2C address on the I2C bus attached to a particular SCA.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. Each SCA in an E1432 module can have a different break-out box attached. The *ID* identifies which break-out box to talk to - the ID of any channel of an SCA identifies the break-out box attached to that SCA.

addr specifies the I2C address to talk to. I2C addresses contain seven bits of address, which should be shifted up one bit so that the bottom bit of the address is always zero. The bits above the bottom eight are ignored.

When reading, *value* is a pointer to a 16-bit integer which will be filled in with the value read. The value read is really only an 8-bit quantity.

When writing, *value* is a 16-bit integer which specifies the value to write. Only the bottom eight bits are actually written.

RESET VALUE

None.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

NAME

e1432_read_register − Read 16-bit register from a single E1432
e1432_write_register − Write 16-bit register to a single E1432
e1432_read32_register − Read 32-bit register from a single E1432
e1432_write32_register − Write 32-bit register to a single E1432

SYNOPSIS

```
SHORTSIZ16 e1432_read_register(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 regOffset, SHORTSIZ16 *data)
SHORTSIZ16 e1432_write_register(E1432ID hw, SHORTSIZ16 ID,
                                LONGSIZ32 regOffset, SHORTSIZ16 data)
SHORTSIZ16 e1432_read32_register(E1432ID hw, SHORTSIZ16 ID,
                                 LONGSIZ32 regOffset, LONGSIZ32 *data)
SHORTSIZ16 e1432_write32_register(E1432ID hw, SHORTSIZ16 ID,
                                  LONGSIZ32 regOffset, LONGSIZ32 data)
```

DESCRIPTION

e1432_read_register reads the contents of a 16-bit register of an E1432 module, identified by the ID from any one of its channels.

e1432_write_register writes data to a particular 16-bit register of an E1432 module, identified by the ID from any one of its channels.

e1432_read32_register reads the contents of a 32-bit register of an E1432 module, identified by the ID from any one of its channels.

e1432_write32_register writes data to a particular 32-bit register of an E1432 module, identified by the ID from any one of its channels.

Normally, these functions trap bus errors that might occur while reading or writing a register, and return the error **ERR1430_BUS_ERROR** if a bus error occurs. This behavior can be changed by using the *e1432_set_try_recover* function.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a single channel which is used to identify the module.

regOffset is the offset of the register relative to the base address of the E1432. Register offsets have defines of the form **E1432_**<*register_name*>**_REG** in the file **e1432.h**. For instance, **E1432_VXI_ID_REG** is at register offset 0. The list of E1432 registers, and their detailed description, may be found in appendix A of the HP E1432A users's guide.

data is the value of the data, to be read from, or to be written into, the register.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_register_address, e1432_set_try_recover

NAME

e1432_reenable_interrupt − Reenable interrupts

SYNOPSIS

```
SHORTSIZ16 e1432_reenable_interrupt(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

When VME interrupts are being used, an E1432 module will do a VME interrupt when an event matching the current interrupt mask occurs. Once it has done this interrupt, the module will not do any more VME interrupts until re-enabled with *e1432_reenable_interrupt*. Normally, the last thing a host computer's interrupt handler should do is call *e1432_reenable_interrupt*.

Events that would have caused an interrupt, but which are blocked because *e1432_reenable_interrupt* has not yet been called, will be saved. After *e1432_reenable_interrupt* is called, these saved events will cause an interrupt, so that there is no way for the host to "miss" an interrupt. However, the module will only do one VME interrupt for all of the saved events, so that the host computer will not get flooded with too many interrupts.

For things like "E1432_IRQ_BLOCK_READY", which are not events but are actually states, the module will do an interrupt after *e1432_reenable_interrupt* only if the state is still present. This allows the host computer's interrupt handler to potentially read multiple scans from an E1432 module, and not get flooded with block ready interrupts after the fact.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

RESET VALUE

None.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_interrupt_mask, e1432_intr(5)

NAME

e1432_reset − Reset (group of) E1432 modules

SYNOPSIS

```
SHORTSIZ16 e1432_reset(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_reset resets the (group of) channel(s) specified by *ID*. This is a hard reset done by setting the RESET bit in the VXI control register. This function should not be needed by normal applications.

To simply restore default settings to a group of channels, use the *e1432_preset* function instead.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_preset

NAME

e1432_reset_lbus − Reset or enable Local Bus
e1432_get_lbus_reset − Return reset status of local bus

SYNOPSIS

```
SHORTSIZ16 e1432_reset_lbus(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 state)
SHORTSIZ16 e1432_get_lbus_reset(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *state)
```

DESCRIPTION

e1432_reset_lbus controls the reset state of the Local Bus. **E1432_RESET_LBUS_ON** puts it into reset and **E1432_RESET_LBUS_OFF** allows the Local Bus to function.

e1432_get_lbus_reset returns the reset state of the Local Bus, into the location pointed to by *state*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state must be either **E1432_RESET_LBUS_ON** or **E1432_RESET_LBUS_OFF**.

At the start of a measurement that uses local bus, it is important to properly reset the local bus before starting the measurement. Before resetting the local bus, make sure all E1432 modules are no longer running any previous measurement, by calling *e1432_reset_measure*. Then the local bus interface on all VXI cards must be reset, and then they must all be un-reset. The order in which the cards must be reset and un-reset is important. After all that, the measurement can be started with *e1432_init_measure*.

RESET VALUE

After a reset the Local Bus is set to **E1432_RESET_LBUS_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_lbus_mode

NAME

e1432_reset_measure − Reset current measure, move E1432s to TESTED

SYNOPSIS

```
SHORTSIZ16 e1432_reset_measure(E1432ID hw, SHORTSIZ16 ID)
```

DESCRIPTION

e1432_reset_measure moves the E1432 measurement state machine to the **TESTED** state.

This function call is not normally needed because *e1432_init_measure* prepares the modules for data gathering.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If the measurement involves more than one module, it is mandatory that a *group ID* be used, rather than a *channel ID*. Using a group ID guarantees that all modules in the group move synchronously to the **TESTED** state.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure

---- NOT FOR GENERAL USE ----

NAME

e1432_sca_dsp_download − Download program and data to SCA DSP(s)

SYNOPSIS

```
SHORTSIZ16 e1432_sca_dsp_download(E1432ID hw, SHORTSIZ16 ID,
                 LONGSIZ32 length, LONGSIZ32 *ldata)
```

DESCRIPTION

e1432_sca_dsp_download downloads program and optional data to an individual SCA DSP or a group of SCA DSPs.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained by a call to *e1432_create_channel_group*, or the ID of a single channel.

length is the size of the download data block to be sent.

ldata is a pointer to the 32 bit download data.

For *E1433* input channels, for hardware implementation reasons, this operation downloads to both channels in a channel pair, such as channels 1 & 2. If both channels in the pair are included in *ID*, the operation will be performed only once on that channel pair.

As a 56002, a *E1433* input channel DSP initially expects 256 program words, to be loaded starting at P:0 and then executed. If *length* is less than 256, NOPs will be appended to complete the download. If X:, Y:, and/or higher P: memory are to be loaded, a secondary loader must be included in the initial 256 words to properly place the additional download data.

Calling *e1432_sca_dsp_download* with either *length* set to zero or *ldata* set to NULL will result in the standard SCA DSP code being restored at the next *e1432_init_measure* call.

Since this function temporarily uses memory in the module which is also used for the arbitray source data, this function must preceed the function calls to pre-load the arbitrary source buffers using *e1432_write_srcbuffer_data*.

This function is not currently implemented for any other SCA DSPs.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise. **_NO_ID**, if *ID* is not a valid channel or group ID. **_BUS_ERROR**, if any of the underlying register accesses fail. **_BUFFER_TOO_SMALL**, if the internal buffer is too small to accomodate *length* words. **_SCA_FIRMWARE_ERROR** and **_SCA_HOSTPORT_FAIL** are symptomatic of the downloaded firmware behaving unexpectedly or the wrong amount of download data being sent with respect to the downloaded secondary loader expectations.

SEE ALSO

e1432_dsp_exec_query

NAME

e1432_selftest − Perform self test of hardware
e1432_install_file − Set self test module downloadable
e1432_set_diag_print_level − Set self test print level

SYNOPSIS

```
SHORTSIZ16 e1432_selftest(SHORTSIZ16 mods, SHORTSIZ16 *las,
                          SHORTSIZ16 tests, void *opts)
SHORTSIZ16 e1432_install_file(const char *fileName, SHORTSIZ16 from_mem)
SHORTSIZ16 e1432_set_diag_print_level(SHORTSIZ16 print_level)
```

DESCRIPTION

e1432_selftest performs a selftest of the hardware. It returns 0 if all tests pass and a negative error code if the self test fails.

e1432_install_file specifies the module downloadable file used by *e1432_selftest* and should be called before *e1432_selftest*.

e1432_set_diag_print_level sets the diagnostic output level of *e1432_selftest* and should be called before *e1432_selftest*.

mods is the number of modules to be tested.

las is the pointer to a logical address or array of logical addresses.

tests specifies the testing level. Valid values of *tests* are **E1432_SELFTEST_BASIC**, **E1432_SELFTEST_FULL**, and **E1432_SELFTEST_FULL_STD_IO**. **E1432_SELFTEST_BASIC** specifies fairly quick testing of basic functionality. **E1432_SELFTEST_FULL** specifies basic testing plus additional testing using the module downloadable file specified with the *e1432_install_file* function. **E1432_SELFTEST_FULL_STD_IO** specifies basic testing, testing using the module downloadable, plus testing assuming testing assuming 1 Volt Peak, 1 kHz sine on all input and tachometer channels and placing 1 Volt Peak, 1 kHz sine on source output for manual verification of source outputs.

opts should be either NULL or a pointer to a string describing the hardware configuration to be tested against. This configuration string is a comma separated list of model and options. An example is "E1432A,1DE,AYF" to specify an 8 channel E1432A with the tachometer option. The model number and the options can be found on the serial number plate(s) on the right side of the module. If *opts* is NULL, *e1432_selftest* will test what it can find.

fileName points to a string which is the path to the module downloadable to be used in **E1432_SELFTEST_FULL** and **E1432_SELFTEST_FULL_STD_IO** testing.

from_mem should be set to 0.

print_level controlls the level of printing in *e1432_selftest*. A setting of 0 or lower completely disables printouts. A setting of 1 results in only terse, failure messages. A setting of 2 causes printout of the testing steps as well as detailed diagnostic messages. A setting of 3 or more is for printout used in the development of the *e1432_selftest* code itself and should not be used.

e1432_selftest underlies the functionality of the *hostdiag* program.

It should be noted that *e1432_selftest* attempts to determine if portions of a module are broken. It is not guaranteed to find all hardware problems. It is *not* a performance test or system verification test.

Coverage

E1432A, E1433A, E1434A models.
Channel add/delete options.
ANM and ANC DRAM options as well as standard DRAM configurations.
AYF tachometer option and 1D4 source option.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

NAME

e1432_send_tachs − Send raw tach times from master to slave modules

SYNOPSIS

```
SHORTSIZ16 e1432_send_tachs(E1432ID hw, SHORTSIZ16 groupID, SHORTSIZ16 tachID,
                            unsigned long *buffer, int size, LONGSIZ32 *count)
```

DESCRIPTION

e1432_send_tachs gets the available new raw tachometer times from the master module and passes them on to all the slave modules in the order tracking mode in a multi-module system. Optionally, it can also return the values of these raw tach times into a user buffer. This function is used to allow the module with the tach board to be the master module, providing raw tach times to all other modules in the system. These tach times are necessary for calculating resampled time data on slave modules.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

groupID is the ID of a group of modules that was obtained with a call to *e1432_create_channel_group*.

tachID is the ID of a **single** tach channel.

buffer is a pointer to a buffer that will recieve the raw tach times. If this is **NULL**, no tach times will be returned, but they will be transmitted to the slave module(s). If this pointer is not NULL, the *size* passes the size of *buffer*.

size is the size of *buffer*. Ignored if *buffer* is **NULL**.

count is a pointer the location into which the number of raw tach times transferred is placed.

The master module with the tach board in the group must have been enabled by a call to *e1432_set_trigger_master*. This call puts all other modules in the group into slave mode. Each time *e1432_send_tachs* is called, all raw tach times collected by the master module since the last call to the function are sent to the slave module(s) in a system. This function can be called at any time, but should be called relatively frequently, since it serves to "sync" all of the modules in a group doing resampling.

NOTE: Once this function is called the first time either it or the function *e1432_get_raw_tachs* **must be called regularly to read tach values out of the module. If tach times are not read out often enough, a tach buffer overflow will happen, which will overwrite the internal tach buffer and cause** *e1432_block_available* **to return ERR1432_TACH_BUFFER_OVERFLOW**.

RESET VALUE

Not applicable.

RETURN VALUE

Returns 0 if successful, a (negative) error number otherwise. The error, **ERR1432_NO_GROUP**, will be returned if *groupID* is not a group ID. **ERR1432_NO_CHANNEL** will be returned if a trigger master has not been set for the group, *groupID* and when the *tachID* is not a valid tach channel.

SEE ALSO

e1432_set_trigger_master, e1432_set_data_mode, e1432_get_raw_tachs

NAME

e1432_send_trigger − Send trigger from master to slave modules

SYNOPSIS

```
SHORTSIZ16 e1432_send_trigger(E1432ID hw, SHORTSIZ16 groupID)
```

DESCRIPTION

e1432_send_trigger gets the data FIFO index of a trigger from the master module and passes it on to all the slave modules in the RPM arming/trigger mode in a multi-module system. This is used to allow the module with the tach board to be the master module, providing a trigger to all other modules in the system.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

groupID is the ID of a group of modules that was obtained with a call to *e1432_create_channel_group*.

The master module with the tach board in the group must have been enabled by a call to *e1432_set_trigger_master*. This call puts all other modules in the group into slave mode. Each time the master module notices a trigger in the RPM arming/trigger mode, the data FIFO index of the trigger is saved and the **E1432_IRQ_TRIGGER** bit is set in the **E142_IRQ_STATUS2_REG** register. This bit can used to enable an interrupt or can be polled by the host computer. This scheme only works when the data mode has been set to **E1432_DATA_MODE_OVERLAP_BLOCK** by a call to *e1432_set_data_mode*. The following fragment of code illustrates how to poll for a master trigger and pass it on to the slaves.

```
/* <masterChan> is a channel in the master module, <groupID> is a group
 * ID that includes the channel, <masterChan>, and at least one channel
 * from each of the slave modules.
 */
error = e1432_set_trigger_master( hw, masterChan,
                          E1432_TRIGGER_MASTER_ON );
if( error ) return error;

  ......

do      /* wait for trigger from master */
{
    error = e1432_read_register( hw, masterChan,
                      E1432_IRQ_STATUS2_REG, &status );
    if( error ) return error;
}while( ( status & E1432_IRQ_TRIGGER ) == 0 );

error = e1432_send_trigger( hw, groupID );
if( error ) return error;
```

RESET VALUE

Not applicable.

RETURN VALUE

Returns 0 if successful, a (negative) error number otherwise. The error, **ERR1432_NO_GROUP**, will be returned if *ID* is not a group ID. **ERR1432_NO_CHANNEL** will be returned if a trigger master has not been set for the group, *ID*.

SEE ALSO

e1432_set_trigger_master, e1432_set_data_mode

NAME

e1432_set_active − Set a group or channel active
e1432_get_active − Get group or channel active state

SYNOPSIS

```
SHORTSIZ16 e1432_set_active(E1432ID hw, SHORTSIZ16 ID,SHORTSIZ16 state)
SHORTSIZ16 e1432_get_active(E1432ID hw, SHORTSIZ16 ID,SHORTSIZ16 *state)
```

DESCRIPTION

An application can use the E1432 interface library to communicate with several E1432 modules, each of which has multiple channels. Obviously, the application will not always want to use all of the channels. The *e1432_set_active* function is used to specify which of the channels the application wants to use.

Channels that are "inactive" are not used at all in a measurement. The active/inactive status of a channel can’t be changed while a measurement is running. For an input channel, if it is inactive then it will not assert trigger, and its data will not get put into the FIFO. For a tach channel, if it is inactive then the RPM of that channel is not monitored and the raw tach times can’t be read by the host computer. For a source channel, if it is inactive then the channel will not produce an output signal.

Changing an input channel from active to inactive, or from inactive to active, will abort any currently running measurement. Changing a source channel from inactive to active will not start a source, only *e1432_init_measure* does that. However, changing a source channel from active to inactive stops the source immediately. Changing a tach channel from active to inactive, or from inactive to active, will have no effect until the start of the next measurement.

Compare this with *e1432_set_enable*, which can enable or disable data from a input channel while a measurement is running.

e1432_get_active returns the state of a channel or group.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

If the *ID* is a channel ID, that one channel is set to *state*.

The *state* can be either E1432_CHANNEL_ON, E1432_CHANNEL_OFF, or E1432_CHANNEL_MAYBE. If *state* is E1432_CHANNEL_MAYBE, then source channels are set to E1432_CHANNEL_OFF, while input and tach channels are set to E1432_CHANNEL_ON.

A call to *e1432_create_channel_group* automatically calls *e1432_set_active* using parameter E1432_CHANNEL_MAYBE for the new group so in the simple case of one group, the programmer never needs to call *e1432_set_active*.

Unlike the E1432 library, in the E1431 library calling this function sets all channels in each module to E1431_CHANNEL_OFF, and then sets the specified channels to *state*. This "feature" is undesirable and is not duplicated in the E1432 library. The E1431 library does not have a E1431_CHANNEL_MAYBE parameter.

RESET VALUE

See above.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_create_channel_group, e1432_set_enable, e1432_e1431_diff

NAME

e1432_set_amp_scale − Set source amplitude scale of E1432 channels
e1432_get_amp_scale − Get current source amplitude scale of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_set_amp_scale(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 amp_scale)
SHORTSIZ16 e1432_get_amp_scale(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 *amp_scale)
```

DESCRIPTION

e1432_set_amp_scale sets the source amplitude scale factor, of a single channel or group of channels *ID*, to the value given in *amp_scale*.

e1432_get_amp_scale returns the current value of the source amplitude scale factor, of a single channel or group of channels *ID*, into a memory location pointed to by *amp_scale*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

amp_scale is the amplitude scale factor, which must be between zero and one.

For input channels, amplitude scale is not used.

For source channels, the actual source amplitude generated is equal to the full-scale range (set by e1432_set_range) multiplied by the amplitude scale factor. The *range* specifies an overall maximum signal level (typically on a range DAC reserved for this purpose) which can't be changed instantaneously during output. The *amplitude scale factor* has finer resolution and can be adjusted instantaneously during output.

For tach channels, neither range nor amplitude scale are used.

RESET VALUE

After a reset, the source *amp_scale* is set to one.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_range, e1432_get_amp_scale_limits

NAME

e1432_set_analog_input − Set all analog input parameters

SYNOPSIS

```
SHORTSIZ16 e1432_set_analog_input(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 mode, SHORTSIZ16 source,
                                  SHORTSIZ16 state, SHORTSIZ16 coupling,
                                  FLOATSIZ32 range)
```

DESCRIPTION

e1432_set_analog_input sets many of the parameters associated with the analog input section of an E1432 or group of E1432s, effectively combining a number of individual parameter function calls. Please refer to the documentation for the individual parameter functions for more specific information.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode determines the input mode in the front end. This parameter may also be set with *e1432_set_input_mode*. Changing the input mode for one channel causes the input mode for all channels within that SCA to change. See *e1432_set_input_mode* for more details.

source selects the input to the ADC. **E1432_INPUT_HIGH_NORMAL** selects the front panel connector. **E1432_INPUT_HIGH_GROUNDED** grounds the ADC input. **E1432_INPUT_HIGH_CALIN** selects the module's CALIN line. **E1432_INPUT_HIGH_BOB_CALIN** selects the module's CALIN line via the cal connection in a break-out box.

This parameter may also be set with *e1432_set_input_high*.

state determines the state of the analog anti-alias filter in the front end. The analog anti-alias filters of the E143x modules can't be disabled, so this value must be set to **E1432_ANTI_ALIAS_ANALOG_ON**. This parameter may also be set with *e1432_set_anti_alias_analog*.

coupling determines the AC or DC coupling mode of the input. Using **E1432_COUPLING_DC** will connect the input directly to the amplifier. **E1432_COUPLING_AC** inserts a series capacitor between the input and the amplifier. This parameter may also be set with *e1432_set_coupling*.

range is the full scale range in volts (or in picoCoulombs if the input mode is set to charge mode). Signal inputs whose absolute value is larger than full scale will generate an ADC overflow error. The possible values for *range* depend on the type of channel being programmed. The actual range that is set will be the nearest legal range value that is greater than or equal to the value specified by the *range* parameter.

If the mode is set to voltage or ICP, this function uses *e1432_set_range* to set the range. If the mode is set to charge mode, this function uses *e1432_set_range_charge* to set the charge-amp range. If the mode is set to microphone mode, this function uses *e1432_set_range_mike* to set the microphone range.

The corresponding E1431 function takes an extra parameter to specify whether the ground path is floating or grounded. This function does not take this parameter, since the E1432 hardware can't set this programmatically. Instead, there is a switch which controls the grounding, on the break-out box that connects to the

inputs.

RESET VALUE

After a reset, *mode* is set to **E1432_INPUT_MODE_VOLT**. The *source* is set to **E1432_INPUT_SOURCE_BNC**, *state* is set to **E1432_ANTI_ALIAS_ANALOG_ON**, *coupling* is set to **E1432_COUPLING_DC**, and *range* is set to the maximum legal value for each channel.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_anti_alias_analog, e1432_set_coupling, e1432_set_input_mode, e1432_set_input_high, e1432_set_range, e1432_set_range_charge, e1432_set_range_mike.

NAME

e1432_set_anti_alias_analog − Enable/disable analog anti-alias filter
e1432_get_anti_alias_analog − Get current state of analog anti-alias filter

SYNOPSIS

```
SHORTSIZ16 e1432_set_anti_alias_analog(E1432ID hw, SHORTSIZ16 ID,
                                       SHORTSIZ16 state)
SHORTSIZ16 e1432_get_anti_alias_analog(E1432ID hw, SHORTSIZ16 ID,
                                       SHORTSIZ16 *state)
```

DESCRIPTION

Unfortunately, the analog anti-alias filters can't be disabled on any input or source channels in the E143x modules. Originally this function was going to be used to disable the analog anti-alias filters, but now this function is essentially useless.

See *e1432_set_anti_alias_digital* for disabling the digital anti-alias filters, which **can** be done.

If this function is called, the parameters must obey the following:

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained by a call to *e1432_create_channel_group*, or the ID of a single channel.

state must be set to **E1432_ANTI_ALIAS_ANALOG_ON**, otherwise an error is generated.

RESET VALUE

After a reset, *state* is set to **E1432_ANTI_ALIAS_ANALOG_ON**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_create_channel_group, e1432_set_analog_input, e1432_set_anti_alias_digital

NAME

e1432_set_anti_alias_digital − Enable/disable digital anti-alias filter
e1432_get_anti_alias_digital − Get current state of digital anti-alias filter

SYNOPSIS

```
SHORTSIZ16 e1432_set_anti_alias_digital(E1432ID hw, SHORTSIZ16 ID,
                                        SHORTSIZ16 state)
SHORTSIZ16 e1432_get_anti_alias_digital(E1432ID hw, SHORTSIZ16 ID,
                                        SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_anti_alias_digital enables or disables the digital anti-alias filter, of a single channel or group of channels *ID*, depending on the value given in *state*. The analog counterpart to the digital anti-alias filter is controlled with another function, *e1432_set_anti_alias_analog*.

e1432_get_anti_alias_digital returns the current state of the digital anti-filter, of a single channel or group of channels *ID*, into a memory location pointed to by *state*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state determines the state of the digital anti-alias filter.

E1432_ANTI_ALIAS_DIGITAL_ON enables the digital anti-alias filters. These filters are elliptic IIR filters, which provide a very flat pass-band and very good stop-band attenuation. These are the default filters.

E1432_ANTI_ALIAS_DIGITAL_OFF disables the digital anti-alias filters.

E1432_ANTI_ALIAS_DIGITAL_BES enables the use of "Bessel" anti-alias digital filters. These filters are IIR digital filters which have nearly linear phase and very little overshoot and ringing. For this reason, these filters are good for doing time-domain measurements. However, these filters have several drawbacks when used for frequency-domain measurements. These filters cut the bandwidth (the 3-dB point) of a measurement to roughly half of what it would be with the elliptical filters, they provide only about 70 dB of alias rejection, and they are not nearly as flat in the frequency domain as the elliptical filters.

When using **E1432_ANTI_ALIAS_DIGITAL_BES**, the actual 3-dB frequency of the data is roughly half of the span setting set by *e1432_set_span*. The effective data rate is still 2.56 times the span setting. It may be useful to increase the oversample rate by a factor of two using the *e1432_set_decimation_oversample*. This will not change the 3-dB frequency of the data, but it will increase the effective data rate to 5.12 times the span setting, doubling the amount of data acquired in a given amount of time.

Changing the input digital filters while a measurement is running will stop the measurement.

For input channels, disabling the digital filters results in data that is not alias-protected in the frequency domain, and therefore this is not usually a good idea for input channels. For source channels, disabling the digital filters results in an output signal that may have high-frequency components in it. For tach channels, there is no digital filter and this function returns an error.

The E1432 and E1433 input SCAs support this parameter to control the digital filtering done by the DSP chip in the SCAs. However, note that this enables and disables *only* the decimation digital filters. It does not disable or change the digital filter that is built into the Delta-Sigma ADCs (there is no way to disable this filter and still have a working ADC), so there is still some digital filtering applied to the data even when you attempt to completely disable the digital filters.

On the E1432, this parameter applies to all channels on an SCA, and can't be set separately for each channel. This restriction is not needed for the E1433.

Disabling the input digital filters is not usually a good idea because aliasing can result. Depending on the application, it may be better to use **E1432_ANTI_ALIAS_DIGITAL_BES**, or the *e1432_set_decimation_undersamp* function may be useful instead. This function provides a different way to prevent use of the digital anti-alias filters, by slowing down the ADC clock on the E1433.

Source 20-Bit Mode

The E1432/3/4 Option 1D4 single-channel source and the E1434 source normally have an interpolation digital filter immediately before the output DAC. This interpolation digital filter uses 16-bit input data, so normally the arb source data path is only 16 bits wide.

This interpolation digital filter can be disabled using the *e1432_set_anti_alias_digital* function, resulting in a 20-bit signal path (the DAC can accept 20-bit data words). When this interpolation filter is disabled, the DAC output is not completely alias protected by the 25.6 kHz analog output filter. To get better alias protection, the analog output filter must be switched to 6.4 kHz (see *e1432_set_filter_freq*). If the source is in random mode, the selected span should be 6.4 kHz or less. If the source is in arb mode, either the span should be set to 6.4 kHz or less, or the arb data should be band-limited by the host computer to 6.4 kHz or less.

There is only one 20-bit signal path for each 2-channel SCA in an E1434, on the first of the two SCA channels. Please refer to the module block diagram in the E1434A Users Guide.

To enable the 20-bit mode, use the following function call sequence: *e1432_set_active*, *e1432_set_source_mode*, *e1432_set_span*, *e1432_set_anti_alias_digital*, *e1432_set_filter_freq*.

RESET VALUE

After a reset, *state* is set to **E1432_ANTI_ALIAS_DIGITAL_ON**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_anti_alias_analog, e1432_set_filter_freq, e1432_set_decimation_undersamp

NAME

e1432_set_append_status − Enable/disable appending trailer onto data
e1432_get_append_status − Get current state of append trailer switch

SYNOPSIS

```
SHORTSIZ16 e1432_set_append_status(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 append)
SHORTSIZ16 e1432_get_append_status(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 *append)
```

DESCRIPTION

e1432_set_append_status controls the appending of a trailer of status data to each block of data. When turned appending is turned on, each block of data transfered out of the E1432 module (either to Local Bus or VME Bus) will have an extra eight 32-bit words appended to the end of the block. These extra words contain status information about that block of data.

e1432_get_append_status returns the current state of append status, of a single channel or group of channels *ID*, into a memory location pointed to by *append*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

append selects whether or not status information is appended to a data block. Specifying **E1432_APPEND_STATUS_ON** means that an extra block of status information is appended to the end of each data block transferred. **E1432_APPEND_STATUS_OFF** disables this feature. This parameter may also be set with *e1432_set_data_format*.

When **E1432_APPEND_STATUS_ON** is selected, the user is responsible for reading the additional trailer data from the module. There are two ways of doing this, both of which are explained on the read-data manual pages (*e1432_read_raw_data*, *e1432_read_float32_data*, and *e1432_read_float64_data*).

One way is to add either 8 or 16 to the size parameter sent to one of the read-data functions (*e1432_read_raw_data*, *e1432_read_float32_data*, and *e1432_read_float64_data*). If the current data size (as selected by *e1432_set_data_size*) is **E1432_DATA_SIZE_16**, then 16 should be added to the size. Otherwise, 8 should be added to the size. This will ensure that the correct amount of data is read after each data block. The reason that the number varies is that the trailer information is always in the same format, regardless of the current data size setting. When reading the data, don't forget to allocate enough room for the extra trailer data!

The other way to get trailer data is to use the *trailer* parameter. If this is done, then the size parameter to the read data function should NOT try to account for the additional trailer data. The *trailer* parameter should point to an array of *e1432_trailer* structures, one for each channel whose data will be read by this function.

The format of the block of trailer information is given by the *e1432_trailer* structure, which is defined in the <e1432.h> include file. The structure contains eight fields, each of which is 32-bits wide. The eight fields are:

trig_corr, which specifies the time between the trigger event and the actual start of data in the block. The time is normalized to the data sample interval. This time is not absolute. Other, fixed delays are present, which depend on setup conditions. This field is a 32-bit float.

zoom_corr, which specifies the phase of the local oscillator at the start of the block. This is for future use when zoom is implemented. It is currently filled with zero. This field is a 32-bit float.

rpm1, which specifies the RPM on the first tach channel in the module. This field contains the RPM only if there is, in fact, a tach channel, and that tach channel is enabled with *e1432_set_active*. The field contains zero otherwise. This field is a 32-bit float.

rpm2, which specifies the RPM on the second tach channel in the module. This field contains the RPM only if there is, in fact, a second tach channel, and that tach channel is enabled with *e1432_set_active*. The field contains zero otherwise. This field is a 32-bit float.

gap, which specifies the number of samples between the start of this scan and the start of the previous scan. For the very first scan of a measurement, the value is the number of samples between the system sync and the start of the first block. This field is a 32-bit integer.

When doing a zoomed measurement (see *e1432_set_zoom*), the gap field is set to twice the number of samples between the start of the scan and the start of the previous scan. This allows the time between successive blocks to be calculated using:

delay = gap / (span * 2.56)

This works for both zoom and non-zoom, due to the differing definitions of span when doing a zoomed measurement.

info, which is a bit-field containing information about the block. This field is a 32-bit integer. Here are the bits that may be present:

E1432_TRAILER_INFO_NOT_UNDERRANGE, which indicates that at least some of the data in this block is above the under-range threshold, so the next lower input range would overload. If this bit is zero, then the next lower input range would not have overloaded due to a differential overload.

E1432_TRAILER_INFO_OVERLOAD, which indicates that at least some of the data in this block is overloaded and therefore may not accurately reflect the input signal. This bit is set for both differential and common-mode overloads. For differential overloads, increasing the input range may remove the overload.

E1432_TRAILER_INFO_OVERLOAD_COMM, which indicates that at least some of the data in this block is overloaded due to a common-mode overload. If this bit is set, then the above **E1432_TRAILER_INFO_OVERLOAD** bit is also set.

E1432_TRAILER_INFO_TRIGGER, which indicates that this input channel was enabled to trigger and detected a trigger level crossing within this block. This does not necessarily mean that this input channel was the trigger for this data block, since there might have been another input trigger by another input channel first.

E1432_TRAILER_INFO_SETTLED, which indicates that the data in this block was collected with properly settled hardware.

E1432_TRAILER_INFO_STOP, which indicates that this is the last scan of data in the measurement.

E1432_TRAILER_INFO_OT_RAMP_TOO_FAST, which indicates that the tachometer signal was changing frequency too fast over the time that resampled data (revolution domain) was being calculated.

E1432_TRAILER_INFO_DEC_2_MASK, which is a set of five bits that, when shifted down by **E1432_TRAILER_INFO_DEC_2_SHIFT**, specify the number of decimate-by-two decimation passes are being performed on the input data before it is sent to the host.

E1432_TRAILER_INFO_DEC_5, which indicates whether a decimate-by-five decimation pass is being performed on the input data before it is sent to the host.

E1432_TRAILER_INFO_CHAN_MASK, which is a set of ten bits that, when shifted down by **E1432_TRAILER_INFO_CHAN_SHIFT**, specify the channel number for the data block. This channel number is off by one from the channel number used in the call to *e1432_create_channel_group*. The "off-by-one" is so that the trailer channel value starts at zero, not one, so that the trailer can efficiently encode all possible channel numbers.

E1432_TRAILER_INFO_TYPE_MASK, which is a set of three bits that, when shifted down by **E1432_TRAILER_INFO_TYPE_SHIFT**, specify the type of data in the data block. A value of zero means time data, a value of (**E1432_FREQ_DATA** - **E1432_TIME_DATA**) means frequency data, and so on. The values are the same as the data type that is passed to *e1432_read_raw_data*, with the value of **E1432_TIME_DATA** subtracted out. This subtraction allows the value to be encoded efficiently in three bits.

peak, which is the Peak value. The *peak_mode* parameter in the *e1432_set_peak_mode* function call determines determines what kind of processing is performed to produce the value returned in *peak*. A *peak_mode* of **E1432_PEAK_MODE_OFF** turns Peak detection off and *peak* will be set to 0 as a result. When *peak_mode* is **E1432_PEAK_MODE_BLOCK**, *peak* is the peak of the full span data across the time period of the block. When *peak_mode* is **E1432_PEAK_MODE_FILT**, *peak* is same as that returned by *e1432_get_current_value* but sampled at the at the trigger point or the closest point within the time period of the block.

rms, which is the RMS value. The *rms_mode* parameter in the *e1432_set_rms_mode* function call determines determines what kind of processing is performed to produce the value returned in *rms*. A *rms_mode* of **E1432_RMS_MODE_OFF** turns RMS processing off and *rms* will be set to 0 as a result. When *rms_mode* is **E1432_RMS_MODE_BLOCK**, *rms* is the rms of the full span data across the time period of the block. When *rms_mode* is **E1432_RMS_MODE_FILT**, *rms* is same as that returned by *e1432_get_current_value* but sampled at the at the trigger point or the closest point within the time period of the block.

The weighting, set by *e1432_set_weighting* applies to both *peak* and *rms*. Both *peak* and *rms* are only available at clock frequencies of 65,536 or less. Currently both *peak* and *rms* are the result of processing the full span data, not the decimated data.

The trailer data is one way to get status information about the data, and is the most common way to get this information. An alternative is to use the **E1432_DATA_SIZE_32_SERV** value for data size (see *e1432_set_data_size*), though this does not contain as much information.

RESET VALUE

After a reset, *append* is set to **E1432_APPEND_STATUS_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_read_raw_data, e1432_read_float32_data, e1432_read_float64_data, e1432_set_clock_freq, e1432_set_data_size, e1432_set_peak_mode, e1432_set_rms_mode, e1432_set_weighting, e1432_set_zoom.

NAME

e1432_set_arm_channel − Select a tach channel as the arm channel
e1432_get_arm_channel − Get the current arm state of a tach channel

SYNOPSIS

```
SHORTSIZ16 e1432_set_arm_channel(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 state)
SHORTSIZ16 e1432_get_arm_channel(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_arm_channel enables/disables a single tach channel as the arming and pre-arming channel for RPM armed measurements.

e1432_get_arm_channel returns the arming state of a tachometer channel into a memory location pointed to by *chanID*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is the ID of a single tachometer channel.

state enables or disables the tach channel as the arming channel. The legal values of this parameter are **E1432_CHANNEL_ON** and **E1432_CHANNEL_OFF**.

In the RPM arming modes (**E1432_ARM_RPM_RUNUP**, **E1432_ARM_RPM_RUNDOWN**, and **E1432_ARM_RPM_DELTA**) the arm channel and the active trigger channel must be the same, otherwise an error will be reported.

RESET VALUE

After a reset, the first tachometer channel is set to **E1432_CHANNEL_ON**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_trigger_channel, e1432_set_trigger_mode, e1432_set_arm_mode, e1432_set_pre_arm_mode

NAME

e1432_set_arm_mode − Set auto arm state
e1432_get_arm_mode − Get current auto arm state
e1432_set_auto_arm − Obsolete name for e1432_set_arm_mode
e1432_get_auto_arm − Obsolete name for e1432_get_arm_mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_arm_mode(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 armState)
SHORTSIZ16 e1432_get_arm_mode(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 *armState)
SHORTSIZ16 e1432_set_auto_arm(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 armState)
SHORTSIZ16 e1432_get_auto_arm(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 *armState)
```

DESCRIPTION

e1432_set_arm_mode sets the arm mode, of a single channel or group of channels *ID*, to the value given in *armState*.

e1432_get_arm_mode returns the current value of the arm mode, of a single channel or group of channels *ID*, into a memory location pointed to by *armState*.

e1432_set_auto_arm and *e1432_get_auto_arm* are identical to *e1432_set_arm_mode* and *e1432_get_arm_mode* respectively. These functions are provided for compatibility with the E1431 Host Interface library, and should not be used by new code.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

armState determines which arm event will allow the module to advance from the IDLE state into the ARM state.

E1432_MANUAL_ARM sets the module to wait for a arm event to occur either from the system (SYNC line), or from the *e1432_arm_measure* command, in order to perform the transition.

E1432_AUTO_ARM sets the module to perform the transition as soon as it enters the IDLE state. When in the order tracking mode, the measurement does not re-arm until the results of the previous arm/trigger have been uploaded by the host. This is done to prevent accumulation of so many trigger points the the internal data FIFO starts to overwrite the older trigger points. The data in order tracking mode is usually not continuous for this reason. If continuous resampled data is wanted, use the **E1432_AUTO_ARM_CONTINUOUS** in order tracking.

E1432_AUTO_ARM_CONTINUOUS is used only in the order tracking mode to continuously resample data. This mode currently only works with the **E1432_TACH_EDGE_TRIGGER** auto trigger mode, which will produce continuously resampled data. If the host does not upload data fast enough to remain in real time, the internal data FIFO will stop automatically before it overwrites the oldest trigger points. The FIFO will then empty and the **E1432_IRQ_MEAS_ERROR** and **E1432_STATUS2_TACH_OVERFLOW** bits set in the status register when the FIFO is completely empty.

There are three RPM arming modes.

E1432_ARM_RPM_RUNUP sets the module to do an initial arm as soon as the RPM from the tachometer board rises above the level set by the *e1432_set_rpm_low* function. After the initial arm, each successive arm occurs after the RPM increases by the amount set by the *e1432_set_rpm_interval* function.

E1432_ARM_RPM_RUNDOWN sets the module to do an initial arm as soon as the RPM from the tach board falls below the value set by the *e1432_set_rpm_high* function. After the initial arm, each successive arm occurs after the RPM decreases by the amount set by the *e1432_set_rpm_interval* function.

E1432_ARM_RPM_DELTA sets the module to do an initial arm as soon as the RPM falls within the values set by the *e1432_set_rpm_low* and *e1432_set_rpm_high* functions. After the initial arm, each successive arm occurs after the RPM changes by the amount set by the *e1432_set_rpm_interval* function.

E1432_ARM_TIME arms the module at regular time intervals set by *e1432_set_arm_time_interval*. The length of the measurement is set by *e1432_set_meas_time_length*.

RESET VALUE

After a reset, *armState* is set to **E1432_AUTO_ARM**. Note that this is different than the E1431, which defaults to **E1432_MANUAL_ARM**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_arm_measure, e1432_set_rpm_low, e1432_set_rpm_high, e1432_set_rpm_interval, e1432_set_interrupt, e1432_get_meas_warning

NAME

e1432_set_arm_time_interval − Set time interval for time arming mode
e1432_get_arm_time_interval − Get time interval for time arming mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_arm_time_interval(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 arm_time_interval)
SHORTSIZ16 e1432_get_arm_time_interval(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 *arm_time_interval)
```

DESCRIPTION

e1432_set_arm_time_interval sets the time between arm points when in the time arming mode. It is a global parameter applying to all channels in a single module. Once started, a measurement will arm at multiples of this interval until the elapsed time exceeds the measurement time set by the *e1432_set_meas_time_length* function.

e1432_get_arm_time_interval returns the current value of the arm time interval into a memory location pointed to by *arm_time_interval*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of tach channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

arm_time_interval is the time interval between arm points when the arm mode is set to **E1432_ARM_TIME** by the *e1432_set_arm_mode* function. When not in order tracking mode, the resolution of time arming points is four milliseconds, limited by the resolution of the internal timer interrupt.

For input channels and source channels, this parameter is not used.

RESET VALUE

After a reset, the *arm_time_interval* defaults to 0.1 seconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_arm_time_interval_limits, e1432_set_meas_time_length

NAME

e1432_set_auto_group_meas − Select auto group set up
e1432_get_auto_group_meas − Get state of auto group set up

SYNOPSIS

```
SHORTSIZ16 e1432_set_auto_group_meas(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 state)
SHORTSIZ16 e1432_get_auto_group_meas(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_auto_group_meas is used to control the automatic multi-module setup performed in *e1432_init_measure*. Since the default is to perform the automatic set up, you probably will never need to call this function. If this function is used to turn off the automatic group setup, each module in a group will need to be set up by your program. For more information about multiple module groups, see the Multiple Module Groups section.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state is either **E1432_AUTO_GROUP_MEAS_ON** or **E1432_AUTO_GROUP_MEAS_OFF**.

When *state* is **E1432_AUTO_GROUP_MEAS_ON**, *e1432_init_measure* automatically takes care of setting up the multi_sync, clock_master, and clock_source parameters for the modules in a measurement, and setting the clock frequency for the non-master modules to match the master module. If there is only one E1432 module, the parameters are set to avoid using the VXI TTLTRG lines, like this:

```
e1432_set_multi_sync(hw, id, E1432_MULTI_SYNC_OFF);
e1432_set_clock_master(hw, id, E1432_MASTER_CLOCK_OFF);
e1432_set_clock_source(hw, id, E1432_CLOCK_SOURCE_INTERNAL);
```

If there are multiple modules, *e1432_init_measure* picks one module to pbe the clock master, and the parameters are set like this:

```
e1432_set_multi_sync(hw, id, E1432_MULTI_SYNC_ON);
for all non-master modules:
    e1432_set_clock_master(hw, slave_id, E1432_MASTER_CLOCK_OFF);
    e1432_set_clock_source(hw, slave_id, E1432_CLOCK_SOURCE_VXI);
    e1432_set_clock_freq(hw, slave_id, clock_freq_of_master_module);
for the master module:
    e1432_set_clock_master(hw, master_id, E1432_MASTER_CLOCK_ON);
    e1432_set_clock_source(hw, master_id, E1432_CLOCK_SOURCE_INTERNAL);
```

When *state* is **E1432_AUTO_GROUP_MEAS_OFF**, *e1432_init_measure* does not do any of the above setups, and the application is responsible for setting up the multi_sync, clock_master, clock_source, and clock_freq parameters.

If the clock and sync/arm/trigger lines are connected to the VXI backplane, they use two of the VXI TTLTRG lines. The choice of which TTLTRG lines are used is determined by the *e1432_set_ttltrg_clock* and *e1432_set_ttltrg_satrg* functions.

RESET VALUE

Each group defaults to E1432_AUTO_GROUP_MEAS_ON

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_init_measure, e1432_create_channel_group, e1432_set_clock_source, e1432_set_clock_freq, e1432_set_clock_master, e1432_set_multi_sync, e1432_set_ttltrg_clock, e1432_set_ttltrg_satrg

NAME

e1432_set_auto_range_mode − Set auto-range mode
e1432_get_auto_range_mode − Get auto-range mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_auto_range_mode(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_auto_range_mode(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_auto_range_mode sets the auto-range mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_auto_range_mode returns the current value of the auto-range mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode specifies the auto-range mode. The valid values are:

E1432_AUTO_RANGE_MODE_DEF During an auto-range, the input range will be adjusted up or down until the best range setting is found.

E1432_AUTO_RANGE_MODE_UP During an auto-range, the input range will never be decreased.

E1432_AUTO_RANGE_MODE_DOWN During an auto-range, the input range will never be increased.

RESET VALUE

After a reset, the auto-range mode is set to **E1432_AUTO_RANGE_MODE_DEF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_auto_range

NAME

e1432_set_auto_trigger − Set auto trigger state
e1432_get_auto_trigger − Get current auto trigger state

SYNOPSIS

```
SHORTSIZ16 e1432_set_auto_trigger(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 trigState)
SHORTSIZ16 e1432_get_auto_trigger(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *trigState)
```

DESCRIPTION

e1432_set_auto_trigger sets the trigger state, of a single channel or group of channels *ID*, to the value given in *trigState*.

e1432_get_auto_trigger returns the current value of the trigger state, of a single channel or group of channels *ID*, into a memory location pointed to by *trigState*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

trigState determines which trigger event will allow the module to advance from the TRIGGER state into the MEASURE state. **E1432_MANUAL_TRIGGER** sets the module to wait for a trigger event to occur either locally or from the system (SYNC line), or from the *e1432_trigger_measure* command, in order to perform the transition. **E1432_AUTO_TRIGGER** sets the module to perform the transition as soon as it enters the TRIGGER state. **E1432_TACH_EDGE_TRIGGER** is used in order tracking or RPM triggering to trigger at the next tach edge after the arming point.

RESET VALUE

After a reset, *trigState* is set to **E1432_AUTO_TRIGGER**. Note that this is different than the E1431, which defaults to **E1432_MANUAL_TRIGGER**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_trigger_measure

NAME

e1432_set_avg_mode − Set averaging mode
e1432_get_avg_mode − Get averaging mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_avg_mode(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_avg_mode(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_avg_mode sets the data averaging mode. At this time averaging is only allowed on the frequency data that results from the FFT calculations which are turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** parameter. Averaging allows the data from multiple scans to be averaged together before being uploaded to the host computer. This method both reduces noise on the data and reduces the amount of data uploaded to the host. The number of averages is set with the *e1432_set_avg_number* function. The rate at which intermediate results in an average are sent to the host is controlled by *e1432_set_avg_update*.

Although averaging is performed only on frequency data, it is still possible to read time data out of the module when averaging is taking place. Whenever frequency data is made available to the host (as controlled by *e1432_set_avg_update*), time data is also made available. The time data is not averaged, it is just the most recent time block (the block corresponding to the last frequency block that was averaged into the average results).

e1432_get_avg_mode returns the current averaging mode.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode selects one of the following averaging modes:

E1432_AVG_NONE turns off averaging.

E1432_AVG_RMS selects a mode where the sum of the squares of the real and complex components of the frequency data (magnitude squared) is averaged point by point for the number of times specified by *e1432_set_avg_number*.

E1432_AVG_LIN selects a mode where the real and complex components of the frequency data are averaged separately for the number of times specified by *e1432_set_avg_number*.

E1432_AVG_EXPO selects an exponential averaging mode where a weighted sum of the squares of the real and complex components of the frequency data is averaged for the number of times, N, specified by *e1432_set_avg_number*. The weighting factor is set by the *e1432_set_avg_weight* function. The algorithm for this weighted average is:

result = new point for n = 1 (first point)

result(n) = (((weight - 1.0) * result(n-1)) + new point) / weight for n = 2 -> N

where n = point number in the average and N = total number in the average.

E1432_AVG_PEAK selects a mode where the maximum values of the sum of the squares of the real and complex components of the frequency data is saved over the number of times specified by *e1432_set_avg_number*.

NOTE: the number of the averaged frequency data points for the sum of squares modes are one half that of the un-averaged and E1432_AVG_LIN modes, since the real and imaginary components are combined to form a single magnitude number. Instead of BLOCKSIZE points, the magnitude squared modes will only have BLOCKSIZE/2 points.

RESET VALUE

After a reset, *mode* is set to **E1432_AVG_NONE**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_avg_number, e1432_set_avg_update, e1432_set_avg_weight, e1432_set_calc_data

NAME

e1432_set_avg_number − Set average number
e1432_get_avg_number − Get average number

SYNOPSIS

```
SHORTSIZ16 e1432_set_avg_number(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 number)
SHORTSIZ16 e1432_get_avg_number(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *number)
```

DESCRIPTION

e1432_set_avg_number sets the number of scans of data to be averaged. At this time averaging is only allowed on the frequency data that results from the FFT calculations which are turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** parameter. Averaging allows the data from multiple scans to be averaged together before being uploaded to the host computer. This method both reduces noise on the data and reduces the amount of data uploaded to the host. Setting the average number to zero allows continuous averaging until the measurement is stopped. The averaging method is set with the *e1432_set_avg_mode* function. The rate at with intermediate results in an average are sent to the host is controlled by *e1432_set_avg_update*.

Although averaging is performed only on frequency data, it is still possible to read time data out of the module when averaging is taking place. Whenever frequency data is made available to the host (as controlled by *e1432_set_avg_update*), time data is also made available. The time data is not averaged, it is just the most recent time block (the block corresponding to the last frequency block that was averaged into the average results).

e1432_get_avg_number returns the average number.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

number selects the total number of scans in the average.

NOTE: the number of the averaged frequency data points for the sum of squares modes are one half that of the un-averaged and E1432_AVG_LIN modes, since the real and imaginary components are combined to form a single magnitude number. Instead of BLOCKSIZE points, the magnitude squared numbers will only have BLOCKSIZE/2 points.

RESET VALUE

After a reset, *number* is set to **10**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_avg_mode, e1432_set_avg_update, e1432_set_avg_weight, e1432_set_calc_data

NAME

e1432_set_avg_update − Set average update rate
e1432_get_avg_update − Get average update rate

SYNOPSIS

```
SHORTSIZ16 e1432_set_avg_update(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 update)
SHORTSIZ16 e1432_get_avg_update(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *update)
```

DESCRIPTION

e1432_set_avg_update sets the update rate of intermediate results of an ongoing average. At this time averaging is only allowed on the frequency data that results from the FFT calculations which are turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** parameter. Averaging allows the data from multiple scans to be averaged together before being uploaded to the host computer. This method both reduces noise on the data and reduces the amount of data uploaded to the host. The averaging method is set with the *e1432_set_avg_mode* function. The total number of scan in an average is controlled by *e1432_set_avg_number*.

Although averaging is performed only on frequency data, it is still possible to read time data out of the module when averaging is taking place. Whenever frequency data is made available to the host (as controlled by *e1432_set_avg_update*), time data is also made available. The time data is not averaged, it is just the most recent time block (the block corresponding to the last frequency block that was averaged into the average results).

e1432_get_avg_update returns the average update rate.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_updates*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

update selects the update rate to the host of intermediate results in an average. A value of 1 means every intermediate result is sent to the host, 2 means every other result, 3 means every third result, etc. The final result in an ongoing average is always sent to the host even if is not a multiple of the update rate.

NOTE: the number of the averaged frequency data points for the sum of squares modes are one half that of the un-averaged and E1432_AVG_LIN updates, since the real and imaginary components are combined to form a single magnitude update. Instead of BLOCKSIZE points, the magnitude squared updates will only have BLOCKSIZE/2 points.

RESET VALUE

After a reset, *update* is set to **1**.

RETURN VALUE

Return 0 if successful, a (negative) error update otherwise.

SEE ALSO

e1432_set_avg_mode, e1432_set_avg_number, e1432_set_avg_weight, e1432_set_calc_data

NAME

e1432_set_avg_weight − Set average weight
e1432_get_avg_weight − Get average weight

SYNOPSIS

```
SHORTSIZ16 e1432_set_avg_weight(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 weight)
SHORTSIZ16 e1432_get_avg_weight(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *weight)
```

DESCRIPTION

e1432_set_avg_weight sets the weight factor for the weighted average mode, **E1432_AVG_EXPO**, set by the e1432_set_avg_mode function. At this time averaging is only allowed on the frequency data that results from the FFT calculations which are turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** parameter. Averaging allows the data from multiple scans to be averaged together before being uploaded to the host computer. This method both reduces noise on the data and reduces the amount of data uploaded to the host. The averaging update rate is set with the *e1432_set_avg_update* function. The total number of scans in an average is set by *e1432_set_avg_number*.

e1432_get_avg_weight returns the average weight factor.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_weights*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

weight selects the weight factor for the average. The algorithm for therate weighted average is:

result = new point for n = 1 (first point)

result(n) = (((weight - 1.0) * result(n-1)) + new point) / weight for n = 2 -> N

where n = point number in the average and N = total number in the average.

RESET VALUE

After a reset, *weight* is set to **1.0**.

RETURN VALUE

Return 0 if successful, a (negative) error weight otherwise.

SEE ALSO

e1432_set_avg_mode, e1432_set_avg_number, e1432_set_avg_update, e1432_set_calc_data

NAME

e1432_set_blocksize − Set measurement blocksize
e1432_get_blocksize − Get current measurement blocksize
e1432_get_blocksize_current_max − Get current maximum blocksize

SYNOPSIS

```
SHORTSIZ16 e1432_set_blocksize(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 blocksize)
SHORTSIZ16 e1432_get_blocksize(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 *blocksize)
SHORTSIZ16 e1432_get_blocksize_current_max(E1432ID hw, SHORTSIZ16 ID,
                                           LONGSIZ32 *max)
```

DESCRIPTION

e1432_set_blocksize sets the measurement blocksize, of a single channel or group of channels *ID*, to the value given in *blocksize*. If a measurement is in progress while calling this function, the measurement is aborted.

e1432_get_blocksize returns the current value of the measurement blocksize, of a single channel or group of channels *ID*, into a memory location pointed to by *blocksize*.

When the data port is set to VME (see *e1432_set_data_port*), then this *blocksize* is the number of data samples produced, for each active channel, each time there is a trigger. These samples are put together into a contiguous block, one block for each active channel, and sent to the VME bus.

When the data port is set to local bus eavesdrop, (see *e1432_set_data_port*), this *blocksize* is again the number of data samples in a block sent to the VME bus.

The transfer size for local bus transfers (when the data port is either local bus, or local bus eavesdrop) is specified by *e1432_set_xfer_size*. However, the default value for that is zero which means to use this *blocksize*.

e1432_get_blocksize_current_max returns the maximum valid value for the blocksize, given the amount of DRAM available, the current number of active channels, and the current settings of the calc data, data size, data port, append status, and fifo size parameters. The value is returned into the memory location pointed to by *max*. If the *ID* is a group ID, then *e1432_get_blocksize_current_max* returns the minimum of the maximum blocksizes of the modules in the group.

This same "current maximum blocksize" can also be used to determine the current maximum value for the overlap.

To get the maximum value for fifo size, use *e1432_get_fifo_size_current_max*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

blocksize selects the number of sample points in a block. The minimum legal value is 1; the maximum depends on how much RAM is available, how many channels are active in a module, and whether the module is doing FFTs or is in zoom mode. When the module is doing FFTs, the following restrictions on

blocksize hold:

Minimum: 64 non-zoom 32 zoom
Maximum: 8192 non-zoom 4096 zoom
Must be a power of two

The blocksize parameter should not include the size of the appended status data, as defined in *e1432_set_append_status*. This parameter may also be set with *e1432_set_data_format*.

NOTE: When doing an order track measurement, the following relationship must hold between these parameters:

max_order <= blocksize * delta_order / 5.12

or an **ERR1432_ILLEGAL_BLOCK_ORDER_COMBO** error will be issued when the measurement starts.

When the data size is set to **E1432_DATA_SIZE_16**, which is the default, the *blocksize* will be rounded down to an even number (but a *blocksize* of one will get rounded up to two to avoid getting a zero block-size).

When doing local-bus transfers, the minimum blocksize is four.

RESET VALUE

After a reset, the measurement *blocksize* is set to **1024** (1K).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_append_status, e1432_set_calc_data, e1432_set_data_format, e1432_set_data_port, e1432_set_data_size, e1432_set_delta_order, e1432_set_max_order, e1432_set_overlap, e1432_set_xfer_size, e1432_get_blocksize_limits, e1432_get_fifo_size_current_max

NAME

e1432_set_calc_data − Set the granularity of resampled time data
e1432_get_calc_data − Get the granularity of resampled time data

SYNOPSIS

```
SHORTSIZ16 e1432_set_calc_data(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 data_type)
SHORTSIZ16 e1432_get_calc_data(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *data_type)
```

DESCRIPTION

e1432_set_calc_data sets the type of calculated data available from a measurement.

e1432_get_calc_data returns the type of calculated data into the variable pointed to by *data_type*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

data_type specifies the type of data calculation. The available options:

E1432_DATA_TIME performs no additional calculation on the input data. Only input time data will be available to the host.

E1432_DATA_FREQ performs a FFT calculation on the input data. The size of the FFT is limited to the following blocksizes:

Minimum: 64 non-zoom 32 zoom
Minimum: 8192 non-zoom 4096 zoom

and must be a power of two. Only input time data and its FFT are available to the host.

E1432_DATA_RESAMP_TIME resamples the input data into the revolution domain. This is the method to enable order tracking calculations to be done in the E1432 module. Only input time data and its resampled time data are available to the host.

E1432_DATA_ORDER performs a FFT calculation on the resampled input data, producing an order spectrum. The size of the FFT is limited to a minimum blocksize of 64 and a maximum blocksize of 4096. Choosing this option also enables the calculation of the resampled input data upon which it depends. The input time data, resampled time data, and order spectrum are all available to the host. **NOTE:** no FFTed input time data is available.

This function only enables what types of data are calculated and available to the host, not what is actually sent to the host. The data sent to the host is enabled or disabled by *e1432_set_enable*. The default condition is that all data available is sent to the host. If that is not what is wanted, *e1432_set_enable* must be used to prevent some of the data from being sent to the host. For example, if only input time data and order data are wanted in the host and not resampled time data, use *e1432_set_calc_data* to set the calculated data to **E1432_DATA_ORDER** and use *e1432_set_enable* to enable input time data, disable resampled time data and enable order data.

NOTE: See the function *e1432_read_float32_data* for a discussion of the relationship of blocksize of the FFT data and averaging modes.

RESET VALUE

After a reset, *data* is set to **E1432_DATA_TIME**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_max_order, e1432_set_delta_order, e1432_set_enable.

NAME

e1432_set_cal_dac − Set calibration DAC value
e1432_get_cal_dac − Get current value of calibration DAC

SYNOPSIS

```
SHORTSIZ16 e1432_set_cal_dac(E1432ID hw, SHORTSIZ16 ID,
                             SHORTSIZ16 cal_dac)
SHORTSIZ16 e1432_get_cal_dac(E1432ID hw, SHORTSIZ16 ID,
                             SHORTSIZ16 *cal_dac)
```

DESCRIPTION

This is a low-level function that is normally not used. Instead, use *e1432_set_cal_voltage* to control the calibration DAC output voltage.

e1432_set_cal_dac sets the E1432 internal calibration DAC value, of a single channel or group of channels *ID*, to the value given in *cal_dac*.

e1432_get_cal_dac returns the current setting of the internal calibration DAC, of a single channel or group of channels *ID*, into a memory location pointed to by *cal_dac*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

cal_dac determines the DAC setting. This should be an integer between -2048 and 2047. This value is written directly to the calibration DAC. The voltage produced at the DAC output depends on the setting of *calin* (see *e1432_set_calin* and *e1432_set_cal_voltage*).

RESET VALUE

After a reset, *cal_dac* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calin, e1432_set_cal_voltage, e1432_set_source_output, e1432_set_sumbus

NAME

e1432_set_cal_voltage − Set calibration DAC voltage
e1432_get_cal_voltage − Get current value of calibration DAC voltage

SYNOPSIS

```
SHORTSIZ16 e1432_set_cal_voltage(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 cal_voltage)
SHORTSIZ16 e1432_get_cal_voltage(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *cal_voltage)
```

DESCRIPTION

e1432_set_cal_voltage sets the E1432 internal calibration DAC voltage, of a single channel or group of channels *ID*, to the value given in *cal_voltage*.

e1432_get_cal_voltage returns the current internal calibration DAC voltage, of a single channel or group of channels *ID*, into a memory location pointed to by *cal_voltage*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

cal_voltage determines the voltage to set. The valid range depends on the setting of the *calin* parameter (see *e1432_set_calin*). If *calin* is E1432_CALIN_DC_LO, then the valid range is -0.4838 to +0.4835 volts. For any other value of *calin*, the valid range is -15 to +14.9927 volts.

This specified voltage is converted to the appropriate calibration DAC setting, and the DAC is set using *e1432_set_cal_dac*.

RESET VALUE

After a reset, *cal_voltage* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calin, e1432_set_cal_dac, e1432_set_source_output, e1432_set_sumbus

NAME

e1432_set_calin − Set driver for the CALIN line
e1432_get_calin − Get current value of CALIN driver

SYNOPSIS

```
SHORTSIZ16 e1432_set_calin(E1432ID hw, SHORTSIZ16 ID,
                           SHORTSIZ16 calin)
SHORTSIZ16 e1432_get_calin(E1432ID hw, SHORTSIZ16 ID,
                           SHORTSIZ16 *calin)
```

DESCRIPTION

e1432_set_calin sets the driver for the internal CALIN line, of a single channel or group of channels *ID*, to the value given in *calin*. The CALIN line is sent to all SCAs, the optional source board, and the break-out box.

e1432_get_calin returns the current setting of the internal calibration DAC, of a single channel or group of channels *ID*, into a memory location pointed to by *calin*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

calin determines the driver for the CALIN line. This should be one of:

E1432_CALIN_OPEN, to have nothing drive the line.

E1432_CALIN_GROUND, to ground the line.

E1432_CALIN_DC_HI, to have the calibration DAC high range drive the line (the DAC can then drive between +-15 volts).

E1432_CALIN_DC_LO, to have the calibration DAC low range drive the line (the DAC can then drive between +-0.48 volts).

E1432_CALIN_SUMBUS, to have the VXI SUMBUS, amplified by a factor of 3, drive the line.

E1432_CALIN_SUMBUS_TACH, which is identical to **E1432_CALIN_SUMBUS**, except that it indicates that the SUMBUS is being driven by an E1432 tach channel. This can be used when monitoring an E1432 tach channel, and is a hint to the input channel that the scale factor should be adjusted properly for the tach board. See *e1432_set_input_high* for more information on monitoring tach channels.

E1432_CALIN_CALOUT, to have the internal CALOUT line drive the line. The CALOUT line can be driven by the optional source board, or by the option tach board. When monitoring tach channels, **E1432_CALIN_CALOUT** is a more direct connection to the input channels, and will result in smaller DC offset. However, it works only within a single E1432 module.

RESET VALUE

After a reset, *calin* is set to **E1432_CALIN_GROUND**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_input_high, e1432_set_cal_voltage, e1432_set_source_output, e1432_set_sumbus

NAME

e1432_set_center_freq - Set zoom center frequency
e1432_get_center_freq - Get zoom center frequency

SYNOPSIS

```
SHORTSIZ16 e1432_set_center_freq(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 freq)
SHORTSIZ16 e1432_get_center_freq(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *freq)
```

DESCRIPTION

e1432_set_center_freq sets the center frequency for "zooming". An error will result when attempting to set the center frequency higher than what can be supported by the module. See the discussion of zooming under the *e1432_set_zoom* function for center frequency limitations and the relationship between center frequency, span and sample frequency.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

freq is the center frequency, in Hertz.

This function sets the center frequency of all input channels in the module referred to by *ID*. By default, all source channels in the module referred to by *ID* will also get this same center frequency. The source channels can be given a center frequency different than the input channels by using *e1432_set_source_centerfreq*, which overrides *e1432_set_center_freq* on source channels.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_zoom, e1432_get_center_freq_limits, e1432_set_span, e1432_set_clock_freq, e1432_set_source_centerfreq

NAME

e1432_set_clock_freq − Set sample clock frequency
e1432_get_clock_freq − Get sample clock frequency

SYNOPSIS

```
SHORTSIZ16 e1432_set_clock_freq(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 freq)
SHORTSIZ16 e1432_get_clock_freq(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 *freq)
```

DESCRIPTION

e1432_set_clock_freq sets the fundamental sample clock frequency used by all input and source channels specified by the *D*. The substrate is capable of creating a wide variety of sample clock frequencies; however, each input or source channel generally supports only a limited range of frequencies.

This sample clock frequency is the clock frequency that is connected to one of the VXI TTLTRG lines, if needed to synchronize several E1432 modules. This clock frequency is normally, but not always, the fundamental rate at which data is collected from the ADCs on the SCAs.

The clock frequency determines which measurement spans are available, and must therefore be set before setting the span with *e1432_set_span*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

freq is the clock frequency, in Hertz.

Internal Clock Source

The way the specified *freq* is used depends on the setting of the *e1432_set_clock_source* parameter. The default clock source, **E1432_CLOCK_SOURCE_INTERNAL**, means that this module is generating the sample clock internally. In this case, the clock frequency must be one of the frequencies that the E1432 module can generate.

For the E1432 51.2 kHz input SCA, the valid clock frequencies (in Hz) that can be generated are:

40960 41938.6 44122.1 48000 49152 50000 51200 52400.9 61440 62500 64000 65536 66666.6 76800 78125 80000 81920 96000 98304 100000 102400

For the E1433 196 kHz input SCA, the valid clock frequencies (in Hz) that can be generated are all of those listed above, plus the following:

122880 125000 128000 133333.3 153600 156250 163840 192000 196608

Some of the above clock frequencies (40960, 49152, 51200, 65536) are useful when frequency-domain measurements are being performed. They are useful because they give round numbers when divided by powers of two. Some of them (48000, 50000, 64000, 66666.6, 80000) are useful when time-domain measurements are being performed. They are useful because their reciprocal is a round number, or perhaps because when decimated by a power of two, their reciprocal is a round number.

Other frequencies in the list may have special applications. For example, 61440 Hz might be useful when doing frequency-domain measurements involving 60Hz power line harmonics and sub-harmonics.

For the E1433 196 kHz input SCA, the clock is used directly by the ADC, and the maximum valid span is clock_freq / 2.56.

For the E1432 51.2 kHz input SCA, the clock is used directly if it is 51200 Hz or less. This means that the largest valid span will be clock_freq / 2.56. If the clock frequency is larger than 51200 Hz, the clock is effectively divided by two before use. This means that the largest valid span will be clock_freq / 5.12. See the manual page for *e1432_set_span* for more details.

For the E1434 source, and the Option 1D4 source board, the valid clock frequencies are the same as those for the E1432 51.2 kHz input SCA, except that the maximum allowed clock frequency is 65536 Hz.

External Clock Sources

If the *e1432_set_clock_source* parameter is set to **E1432_CLOCK_SOURCE_VXI**, **E1432_CLOCK_SOURCE_EXTERNAL**, or **E1432_CLOCK_SOURCE_EXTERNALN**, it means that this module is not generating the clock internally. In this case, the clock frequency coming into the module must still be within the range of 40960 Hz to 102400 Hz (for the E1432) or 40960 Hz to 196608 Hz (for the E1433), or 40960 Hz to 65536 Hz (for the E1434). The *freq* parameter to *e1432_set_clock_freq* must match the frequency coming into the module, and the frequency coming into the module must be a fixed frequency so that the module's internal phase locked loop can lock to the external clock.

The module will use the specified *freq* to determine the valid spans. For the E1433 196 kHz input SCA, the clock is used directly by the ADC, and the maximum valid span is clock_freq / 2.56.

For the E1432 51.2 kHz input SCA, the clock is used directly if it is 51200 Hz or less. This means that the largest valid span will be clock_freq / 2.56. If the clock frequency is larger than 51200 Hz, the clock is effectively divided by two before use. This means that the largest valid span will be clock_freq / 5.12. The maximum allowed clock frequency is 102400 Hz. See the manual page for *e1432_set_span* for more details.

For the Option 1D4 source board, the maximum valid clock frequency is 65536 Hz.

RESET VALUE

The default clock frequency is 51200 Hz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_clock_freq_limits, e1432_set_span, e1432_set_clock_source

NAME

e1432_set_clock_master − Set sample clock source
e1432_get_clock_master − Get current value of sample clock source

SYNOPSIS

```
SHORTSIZ16 e1432_set_clock_master(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 state)
SHORTSIZ16 e1432_get_clock_master(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *state)
```

DESCRIPTION

A typical measurement doesn't generally need to deal with *e1432_set_clock_master* at all. Normally, the call to *e1432_init_measure* automatically takes care of setting the clock master correctly for all the modules in a measurement, by internally calling *e1432_set_clock_master* for each module. This automatic setup can be disabled using the *e1432_set_auto_group_meas* function.

e1432_set_clock_master controls whether the E1432 module drives the VXI bus clock line. The clock line is one of the VXI TTLTRG lines, as selected by *e1432_set_ttltrg_clock*.

e1432_get_clock_master returns the current value of the master clock for a single channel or group of channels *ID*, into a memory location pointed to by *state*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state determines the state of the master clock. **E1432_MASTER_CLOCK_ON** drives the sample clock from this module onto the VXI bus. **E1432_MASTER_CLOCK_OFF** turns off the sample clock drive.

RESET VALUE

After a reset, *state* is set to **E1432_MASTER_CLOCK_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_clock, e1432_set_auto_group_meas, e1432_init_measure, e1432_set_clock_source

NAME

e1432_set_clock_source − Set sample clock source
e1432_get_clock_source − Get current value of sample clock source

SYNOPSIS

```
SHORTSIZ16 e1432_set_clock_source(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 source)
SHORTSIZ16 e1432_get_clock_source(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *source)
```

DESCRIPTION

A typical measurement doesn't generally need to deal with *e1432_set_clock_source* at all. Normally, the call to *e1432_init_measure* automatically takes care of setting the clock source for all the modules in a measurement, by internally calling *e1432_set_clock_source* for each module. This automatic setup can be disabled using the *e1432_set_auto_group_meas* function.

e1432_set_clock_source sets the source of the sample clock, of a single channel or group of channels *ID*, to the value given in *source*. If a measurement is in progress while calling this function, the measurement is aborted.

e1432_get_clock_source returns the current value of the sample clock source, of a single channel or group of channels *ID*, into a memory location pointed to by *source*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

source determines the source for the sample clock.

E1432_CLOCK_SOURCE_INTERNAL selects the internal sample clock. This is the default. With this clock source, the clock frequency programmed by *e1432_set_clock_freq* will get rounded to one of the valid clock frequencies that the E1432 hardware can generate.

E1432_CLOCK_SOURCE_INT_VXI10 is nearly the same as **E1432_CLOCK_SOURCE_INTERNAL**. It is different only if the clock frequency (programmed by *e1432_set_clock_freq*) is set to one of the valid clock frequencies that is a sub-multiple of 10 MHz. Normally when the clock frequency is one of these values, the module divides down an internal 10 MHz crystal oscillator to produce the sample clock. The **E1432_CLOCK_SOURCE_INT_VXI10** value causes the module to instead divide down the VXI backplane 10 MHz clock line to produce the sample clock. Usually you don't want to do that, because the internal crystal is more accurate, more stable, and less noisy. If you switch to the backplane 10 MHz line, the module may not meet all of its specifications. But special applications might want to force the use of the backplane 10 MHz line.

E1432_CLOCK_SOURCE_VXI selects the VXI bus sample clock. This assumes that something else out there is producing the sample clock on one of the VXI TTLTRG lines. The clock frequency coming in can be anything within the range of 40960 Hz to 196608 Hz (max 102400 Hz for E1432, max 196608 Hz for E1433), but the clock frequency coming in must match the clock frequency told to the module with *e1432_set_clock_freq*. The clock frequency coming in must be a fixed frequency so that the internal phase locked loop can accurately lock to this frequency.

E1432_CLOCK_SOURCE_EXTERNAL and **E1432_CLOCK_SOURCE_EXTERNALN** select the external sample clock, positive true and negative true respectively. The clock frequency coming in can be anything within the range of 40960 Hz to 196608 Hz (max 102400 Hz for E1432, max 196608 Hz for E1433, max 65536 for E1434). The clock frequency coming in must match the clock frequency told to the module with *e1432_set_clock_freq*, and the clock frequency coming in must be constant so that the module's PLL can lock to it successfully.

The external sample clock input is present on any E143x module except those which have the optional 1D4 source board. This input is an SMB connector labeled "ExSamp". The external sample clock input is a TTL input, so the external clock signal must have TTL signal levels.

To use the external sample clock input, you will need to use *e1432_set_auto_group_meas*, and also program the clock master and multi-sync parameters for all the E143x modules in the measurement. See *e1432_set_auto_group_meas* for more information.

E1432_CLOCK_VXI_DEC_3 selects the VXI bus clock, divided by 3, provided by some other clock master. The **E1432_CLOCK_VXI_DEC_3** mode was provided for use with the E1431 module, which generates a clock that is 196608 Hz, which is three times 65536. To use the same clock line for both an E1431 and an E1432, the E1431 must be the clock master, and the E1432 must use **E1432_CLOCK_VXI_DEC_3** to get a clock frequency that it can use. However, the hardware to implement this option on the E1432 module has never been tested and the use of this option is not recommended or supported.

For clock sources external to the module, *e1432_set_clock_frequency* must be set to the frequency being input to the module in order for the internal phase locked loop to be configured correctly.

Note that if *e1432_init_measure* is subsequently called and *e1432_set_auto_group_meas* has not been set to **E1432_AUTO_GROUP_MEAS_OFF**, the clock source will be reset to a default value. See *e1432_init_measure* and *e1432_set_auto_group_meas* for more detailed information.

Other External Sample Clocks

If you have an E143x module that does not have the external sample clock connector on the front panel, it may be possible to use the TRIG IN connector on the front panel of a V/743 or E1482B MXI card to provide an external sample clock. Perhaps a VXLink card has this as well. In this case, you would use use **E1432_CLOCK_SOURCE_VXI** for all E1432 modules in the measurement.

To use the TRIG IN connector on the V/743 or E1482B MXI card would probably require using some low-level SICL function calls to set it up correctly. I haven't actually done this myself, but I believe you use the ivxitrigroute function to do this with a V/743, and you instead use the file `/usr/pil/etc/vxi16/oride.cf` to configure a MXI card to do this.

RESET VALUE

After a reset, *source* is set to **E1432_CLOCK_SOURCE_INTERNAL**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_auto_group_meas, e1432_init_measure, e1432_set_clock_freq, e1432_set_ttltrg_clock

NAME

e1432_set_coupling − Set input coupling to AC or DC
e1432_get_coupling − Get current state of input coupling

SYNOPSIS

```
SHORTSIZ16 e1432_set_coupling(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 coupling)
SHORTSIZ16 e1432_get_coupling(E1432ID hw, SHORTSIZ16 ID,
                              SHORTSIZ16 *coupling)
```

DESCRIPTION

e1432_set_coupling sets the input coupling, of a single channel or group of channels *ID*, to the value given in *coupling*.

e1432_get_coupling returns the current state of the input coupling, of a single channel or group of channels *ID*, in a memory location pointed to by *coupling*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

coupling determines the AC or DC coupling mode of the input. In addition, it can control the high-pass filter that is present on some Charge and Microphone Break-out Boxes. The valid values for *coupling* are:

E1432_COUPLING_DC, which DC couples the input (and turns off the high-pass filter if one is present on the Charge or Microphone Break-out Box).

E1432_COUPLING_AC, which AC couples the input (and turns off the high-pass filter if one is present on the Charge or Microphone Break-out Box).

E1432_COUPLING_DC_BOB_HP, which DC couples the input, and also turns on the high-pass filter on the Charge or Microphone Break-out Box. This value is valid only if there is a Charge Break-out Box with option 402 or a Microphone Break-out Box with option 402 connected to the input. In addition, this value is valid only when the Charge Break-out Box is in charge mode or the Microphone Break-out Box is in Microphone mode (see *e1432_set_input_mode*).

E1432_COUPLING_AC_BOB_HP, which AC couples the input, and also turns on the high-pass filter on the Charge or Microphone Break-out Box. This value is valid only if there is a Charge Break-out Box with option 402 or a Microphone Break-out Box with option 402 connected to the input. In addition, this value is valid only when the Charge Break-out Box is in charge mode or the Microphone Break-out Box is in Microphone mode (see *e1432_set_input_mode*).

On an E1432 input, the AC coupling filter is a simple one-pole analog filter with a cutoff of about 0.7 Hz, which gives it a settling time constant of about 150 ms.

On an E1433 input, the AC coupling filter is implemented using feedback from the input DSP back to a DAC connected to the analog signal path. This complicated setup allows the corner frequency of the AC coupling to be changed programmatically (see *e1432_set_coupling_freq*), allows tight phase-matching between E1433 input channels, and does a better job of removing DC offsets. The settling time constant for this filter is roughly 0.25/f0 seconds, where f0 is the programmed corner frequency.

The output of a Charge Break-out Box in Charge mode is always inherently AC coupled, regardless of the setting of this *coupling* parameter. However, there may be a residual DC offset coming out of the Charge BoB. If the input coupling is set to AC coupling, this DC offset should be (at least partially) removed. If

the input coupling is set to *E1432_COUPLING_AC_BOB_HP*, or *E1432_COUPLING_DC_BOB_HP*, then the output of the Charge BoB has a high-pass filter applied to it, which will help remove low-frequency signals. The high-pass filter is a third-order Butterworth filter with a cutoff frequency of about 10 Hz.

The output of a Microphone Break-out Box in Microphone mode is always inherently AC coupled, regardless of the setting of this *coupling* parameter. However, there may be a residual DC offset coming out of the Microphone BoB. If the input coupling is set to AC coupling, this DC offset should be (at least partially) removed. If the input coupling is set to *E1432_COUPLING_AC_BOB_HP* or *E1432_COUPLING_DC_BOB_HP*, then the output of the Microphone BoB has a high-pass filter applied to it, which will help remove low-frequency signals. The high-pass filter is a third-order Butterworth filter with a cutoff frequency of about 22.4 Hz. For input range settings greater than 5 Volts, the high-pass filter is disconnected due to hardware limitations (which may be changed in the future), even when the *coupling* is set to enable the high-pass filter.

For source channels, this parameter is not used, since it is not possible to AC couple the output of the source. An attempt to set this parameter will generate an error.

For tach channels, this parameter is not used, since it is not possible to AC couple the tach input. An attempt to set this parameter will generate an error.

RESET VALUE

After a reset, *coupling* is set to **E1432_COUPLING_DC**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_analog_input, e1432_set_coupling_freq, e1432_set_input_mode

NAME

e1432_set_coupling_freq − Set AC coupling frequency
e1432_get_coupling_freq − Get current AC coupling frequency

SYNOPSIS

```
SHORTSIZ16 e1432_set_coupling_freq(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 coupling_freq)
SHORTSIZ16 e1432_get_coupling_freq(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 *coupling_freq)
```

DESCRIPTION

e1432_set_coupling_freq sets the AC coupling frequency, of a single channel or group of channels *ID*, to the value given in *coupling_freq*.

e1432_get_coupling_freq returns the current value of the AC coupling frequency, of a single channel or group of channels *ID*, into a memory location pointed to by *coupling_freq*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

coupling_freq is the desired AC coupling frequency in Hz. This frequency is the 3 dB point of the high-pass filter that implements AC coupling.

The AC coupling frequency is used only for input channels - source and tach channels do not have AC coupling filters.

The AC coupling frequency is only controllable on E1433 196 kHz input channels. E1432 51.2 kHz input channels have a fixed AC coupling filter at roughly 1 Hz. Any attempt to set the AC coupling frequency of an E1432 input channel will generate an error. However, *e1432_get_coupling_freq* will return 1.0 for E1432 input channels.

The AC coupling frequency is used only when an input channel is AC coupled (see *e1432_set_coupling*). The AC coupling frequency is settable over a continuous range of 0.1 Hz to 100 Hz.

RESET VALUE

After a reset, input channels have the *coupling_freq* set to 1 Hz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_coupling_freq_limits, e1432_set_coupling

NAME

e1432_set_data_format − Set all data format parameters, except data port

SYNOPSIS

```
SHORTSIZ16 e1432_set_data_format(E1432ID hw, SHORTSIZ16 ID,
                                 LONGSIZ32 blocksize, SHORTSIZ16 size,
                                 SHORTSIZ16 mode, SHORTSIZ16 append)
```

DESCRIPTION

e1432_set_data_format sets all of the parameters associated with the data format section of an E1432 or group of E1432s.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

blocksize selects the number of sample points in a block. A number between 1 and an upper limit that depends on the amount of memory installed is a valid value for *blocksize*. This parameter should not take into account the size of the appended status data. This parameter may also be set with *e1432_set_blocksize*.

size selects the number of bits of precision for the fixed point, two's complement data outputs from the E1432. The legal values for this parameter are **E1432_DATA_SIZE_16**, **E1432_DATA_SIZE_32**, and **E1432_DATA_SIZE_32_SERV**. This parameter may also be set with *e1432_set_data_size*.

mode selects whether the E1432's data collection operates in block mode or continuous mode. **E1432_BLOCK_MODE** selects block transfer mode. **E1432_CONTINUOUS_MODE** means data collection will be continuous. This parameter may also be set with *e1432_set_data_mode*.

append selects whether or not status information is appended to a data block. Specifying **E1432_APPEND_STATUS_ON** means that an extra block of status information is appended to the end of each data block transferred. **E1432_APPEND_STATUS_OFF** disables this feature. This parameter may also be set with *e1432_set_append_status*.

RESET VALUE

After a reset, *blocksize* is set to **1024** (1K), *size* is set to **E1432_DATA_SIZE_16**, *mode* is set to **E1432_BLOCK_MODE**, and *append* is set to **E1432_APPEND_STATUS_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_append_status, e1432_set_blocksize, e1432_set_data_mode, e1432_set_data_size

NAME

e1432_set_data_mode − Set data collection mode
e1432_get_data_mode − Get current data collection mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_data_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_data_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_data_mode sets the data collection mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_data_mode returns the current state of the data collection mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode selects whether the E1432's data collection operates in block mode, overlap block mode, overlap freerun mode, or continuous mode.

E1432_BLOCK_MODE selects block transfer mode. In this mode, the E1432 will stop collecting data, and go to the **IDLE** state, as soon as one scan of data has been collected and transferred to the host.

E1432_DATA_MODE_OVERLAP_BLOCK selects overlap block mode. The main difference between overlap block mode and block mode is that overlap block mode allows more arms and triggers to occur before an already-acquired block is sent to the host. Each time a trigger occurs, the trigger time is remembered so that the data for that trigger can eventually get sent to the host. There is no longer a constraint that the trigger must occur after the end of the previous block, so overlapping blocks are possible (hence the name "overlap block mode"). As in continuous mode, there is an overlap parameter which controls how much overlap is allowed between consecutive blocks.

By using *e1432_set_fifo_size*, overlap block mode can be configured to act exactly like block mode. If the FIFO size is set the same as the block size then overlap block mode becomes identical to block mode. There are no visible differences at all.

E1432_DATA_MODE_OVERLAP_FREERUN selects overlap freerun mode. This mode is like overlap block mode in that it allows an arm or trigger to occur before an already-acquired block is sent to the host, and it allows overlapping blocks. However, this mode tries not to queue up pending triggers, so that the data sent to the host is the most recent data available.

E1432_CONTINUOUS_MODE selects continuous data mode. In this mode, the E1432 stays in the **MEASURE** state and collects data until the FIFO overflows or the measurement is stopped by the host. In that case it goes to the **IDLE** state.

This parameter may also be set with *e1432_set_data_format*.

RESET VALUE

After a reset, *mode* is set to **E1432_DATA_MODE_OVERLAP_BLOCK**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_data_format, e1432_set_fifo_size

NAME

e1432_set_data_port − Set data port to VME or Local Bus
e1432_get_data_port − Get current data port

SYNOPSIS

```
SHORTSIZ16 e1432_set_data_port(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 port)
SHORTSIZ16 e1432_get_data_port(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *port)
```

DESCRIPTION

e1432_set_data_port sets a single channel or group of channels *ID*, to deliver data either on the VME backplane or on the Local Bus (for high speed data transfers), or both, depending on the value of *port*.

In any mode that uses the Local Bus, the group of E1432s must be contiguous in one mainframe and positioned immediately to the left of the module that is to receive the local bus data. The module on the extreme left generates data and the others append their data to the data which is pipelined through them, from left to right.

e1432_get_data_port returns the current value of the data port, of a single channel or group of channels *ID*, into a memory location pointed to by *port*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

port determines the path the E1432 uses to make its data available to the host. The following three modes are valid:

E1432_SEND_PORT_VME causes data to be accessed by the host via the **VME bus**. The E1432 module is capable of using D32 and has a fairly fast VME interface, so this mode is sufficient for many applications.

E1432_SEND_PORT_LBUS causes data to be transmitted via the **Local Bus**. This mode is only valid if the E1432 module actually has a local bus interface, which is option UGV for the E1432 module. The local bus interface is significantly faster than the VME interface, and using the local bus frees up the VME bus for other uses. However, the local bus only connects modules which are adjacent to each other, and transfers data only left to right.

Only time data is sent to the local bus, and the enable settings from *e1432_set_enable* are completely ignored.

E1432_SEND_PORT_LBUS_EAVES causes data to be transmitted both to the local bus and the VME bus. This mode is only valid if the E1432 module actually has a local bus interface, which is option UGV for the E1432 module. This mode is typically used to send data to a disk via the local bus, while allowing a host computer to monitor the data at the same time. It is possible to have the E1432 module send time data to the local bus, while simultaneously FFTing the data and sending the FFT results to the VME bus.

EAVESDROP DETAILS

When doing eavesdrop, the local bus data ignores *e1432_set_data_mode* and always uses continuous input data. The user uses *e1432_set_data_mode* to control what data mode (continuous, overlap, block) is sent to VME. The "continuous" mode is valid, but it won't quite be continuous mode because the measurement won't be aborted if the host can't keep up with the VME data. Instead, some of the data is skipped over and not sent to the host.

When doing eavesdrop, the user uses *e1432_set_enable* to control which channels and data types he is interested in. These are the channels and data types which get sent to VME. The local bus data ignores *e1432_set_enable*, and always sends all active input channels and always sends only raw time data.

When doing eavesdrop, the user uses *e1432_set_calc_data* to specify what calculations (resampling, FFT) are done. This applies only to the VME data. The local bus data is always just raw time data.

When doing eavesdrop, *e1432_set_blocksize* controls the blocksize of the data going to VME, but NOT the blocksize of data going to local bus. The local bus thruput uses a separate *e1432_set_xfersize* to control the local bus blocksize. NOTE: this is not yet implemented, currently blocksize applies in all cases. Non-eavesdrop local-bus measurements will use xfersize as well. However, the default for xfersize will be zero, which will mean to use the value of blocksize. This will make things work in a backwards compatible way.

When doing eavesdrop, *e1432_set_overlap* does not apply to the local bus data. The local bus data is continuous with an effective overlap of 0. For VME data, *e1432_set_overlap* is used the same as it normally is when not doing eavesdrop.

When doing eavesdrop, if the local bus thruput falls behind far enough, the input data FIFOs will fill up and the measurement will get a FIFO overflow. This will abort the measurement (including the VME eavesdropping) just like it does in any other case of FIFO overflow with continuous data.

When doing eavesdrop, if the VME data transfer falls behind, then some of the input data is just skipped over and is not sent to VME. No errors are generated. However, the VME data transfer always gets a complete scan (a complete block from all enabled channels). In addition, the "gap" field in the trailer block is filled in correctly, so the user will know the gap between one scan and the next. See *e1432_set_append_status* for details about the trailer data.

Because of the above, local bus transfers will have priority over VME data calculations and transfers. The module slows down the rate at which VME receives data blocks rather than cause the local bus thruput to fall behind.

When doing eavesdrop, normally the VME data is not continuous, and there is an arm and trigger for each VME data block, just like when doing a non-local-bus measurement. These arms and triggers have no effect on the continuous local bus data, except that the very first arm/trigger starts the continuous local bus data. There is no way to use the local bus data to reconstruct where those arms and triggers after the first happened, so there is no way to reconstruct exactly which blocks got sent to VME during the eavesdrop.

If the module is doing order tracking, and therefore is in multi-pass mode, the data sent to local bus is taken from the top span only. Also, when the module is doing order tracking, the data sent to local bus is not oversampled data, even if oversample is turned on. This is done so that the data sent to disk is always the same (top span non-oversampled time data) regardless of the order tracking setup.

```
Eavesdrop summary:

                        Local bus data          VME data
-----------------------------------------------------------------------
e1432_set_data_mode     Ignored, local bus      Used
```

	data is always continuous	
e1432_set_enable	Ignored, local bus data is all channels, raw time data only	Used to control what is sent to VME bus
e1432_set_calc_data	Ignored, local bus data is raw time data only	Used to control what is sent to VME bus
e1432_set_blocksize	Ignored	Used
e1432_set_xfersize	Used	Ignored
e1432_set_overlap	Ignored, overlap is effectively 0	Used
"Falling behind"	If we fall behind far enough, a FIFO overflow will abort the measurement	Skip over input data in order to catch up
Priority	Local bus xfer is first priority	Uses only whatever processing time is left, even if it means falling behind
Arm/trigger	First one starts thruput, ignored after that	Used for each block

RESET VALUE

After a reset, *port* is set to **E1432_SEND_PORT_VME**

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_lbus_mode, e1432_reset_lbus, e1432_set_active, e1432_set_append_status e1432_set_enable, e1432_set_calc_data, e1432_set_data_mode, e1432_set_blocksize, e1432_set_xfersize, e1432_set_overlap

NAME

e1432_set_data_size − Set size of samples to 16 or 32 bits
e1432_get_data_size − Get current value of data size

SYNOPSIS

```
SHORTSIZ16 e1432_set_data_size(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 size)
SHORTSIZ16 e1432_get_data_size(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *size)
```

DESCRIPTION

e1432_set_data_size sets the sample data size, of a single channel or group of channels *ID*, to the value given in *size*.

The data size is mostly an internal issue. Normally, data is read from the E1432 module using either *e1432_read_float32_data* or *e1432_read_float64_data*, and these functions will convert the raw E1432 data into the requested format, regardless of the setting of data size.

e1432_get_data_size returns the current value of the sample data size, of a single channel or group of channels *ID*, into a memory location pointed to by *size*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

size selects the number of bits per sample. The output is fixed point, two's complement data. The legal values for this parameter are **E1432_DATA_SIZE_16**, **E1432_DATA_SIZE_32**, **E1432_DATA_SIZE_32_SERV**, and **E1432_DATA_SIZE_FLOAT32**. This parameter may also be set with *e1432_set_data_format*.

All formats are integer formats except **E1432_DATA_SIZE_FLOAT32**. Choosing 16-bit precision allows for more data throughput on the bus. Choosing 32 bits allows more dynamic range. Choosing 32 bits with *service* (**E1432_DATA_SIZE_32_SERV**) provides additional information in the lower eight bits of each sample. **E1432_DATA_SIZE_FLOAT32** means that data from the channels will be already-scaled-to-volts 32-bit IEEE floating point numbers.

Note: When the module is doing order tracking, then only the 32-bit integer modes are valid.

All time data read from the E1432 module will be in the format specified. Trailer data at the end of a block, if present, is in a fixed format that is not affected by this data size function. (This trailer data is only present if it is explicitly turned on with *e1432_set_append_status*.) Also, frequency data, resampled time data, and order data are not affected by the data size parameter, and are always sent as 32-bit floating-point values.

The **E1432_DATA_SIZE_32_SERV** mode provides information about each sample in the block, by embedding the information in the lower eight bits of each sample. The bottom four bits (bit number 0 through 3) contain the digital filter pass count. Bit 4 is set if this sample had an overload, bit 5 is set if this sample corresponds to a trigger, bit 6 is set if this sample caused the input channel to assert trigger, and bit 7 is set if the input signal is above approximately half-scale. Bit 7 is only implemented on the E1432 inputs, not on the E1433 inputs.

An alternative (and typically much more convenient) way to get information about the data is to enable trailer data at the end of the block, using *e1432_set_append_status*.

For the E1433 196 kHz input SCA, using **E1432_DATA_SIZE_32** or **E1432_DATA_SIZE_FLOAT32** will not work when the sample clock frequency (as set by *e1432_set_clock_freq*) is greater than 165 kHz, when using all eight channels. Use either **E1432_DATA_SIZE_16** or **E1432_DATA_SIZE_32_SERV**, or use fewer than eight channels, to work around this.

RESET VALUE

After a reset, *size* is set to **E1432_DATA_SIZE_16**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_append_status, e1432_set_data_format

NAME

e1432_set_delta_order − Set the granularity of resampled time data
e1432_get_delta_order − Get the granularity of resampled time data

SYNOPSIS

```
SHORTSIZ16 e1432_set_delta_order(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 delta)
SHORTSIZ16 e1432_get_delta_order(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *delta)
```

DESCRIPTION

e1432_set_delta_order sets the granularity of the resampled time data. This data is in the revolution domain. The *delta* parameter is the spacing expressed as a fraction of a tachometer revolution (order) of each element of the resampled data from the order tracking measurement. Resampled data calculation is activated by the *e1432_set_calc_data* function.

e1432_get_delta_order returns the resampled time data spacing into the variable pointed to by *delta*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

delta specifies the resampled time data spacing.

NOTE: When doing an order track measurement, the following relationship must hold between these parameters:

max_order <= blocksize * delta_order / 5.12

or an **ERR1432_ILLEGAL_BLOCK_ORDER_COMBO** error will be issued when the measurement starts.

RESET VALUE

After a reset, *delta* is set to 0.1.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_delta_order_limits, e1432_set_blocksize, e1432_set_calc_data, e1432_set_max_order

NAME

e1432_set_decimation_bandwidth − Set data decimation bandwidth
e1432_get_decimation_bandwidth − Get current data decimation bandwidth

SYNOPSIS

```
SHORTSIZ16 e1432_set_decimation_bandwidth(E1432ID hw, SHORTSIZ16 ID,
                                          SHORTSIZ16 decBw)
SHORTSIZ16 e1432_get_decimation_bandwidth(E1432ID hw, SHORTSIZ16 ID,
                                          SHORTSIZ16 *decBw)
```

DESCRIPTION

This function is provided only for backward compatibility with the E1431 Host Interface library. All new code should use *e1432_set_span* instead.

e1432_set_decimation_bandwidth sets the decimation bandwidth, of a single channel or group of channels *ID*, to the value given in *decBw*. The span is derived from the decimation bandwidth, and sample clock (Fs) as follows: span = Fs / (2.56 * 2***decBw*).

Decimation allows data reduction on oversampled data, saving only those points needed to reconstruct the waveform. A decimation of 2 keeps every other data point, a decimation of 4 keeps every fourth data point, etc. The bandwidth of the data must be reduced at the same time to prevent aliasing.

e1432_get_decimation_bandwidth returns the current value of the decimation bandwidth, of a single channel or group of channels *ID*, into a memory location pointed to by *decBw*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

decBw selects the bandwidth of the filter and the amount of decimation applied to the signal. The possible values for this parameter range from **0** to **9**. This parameter may also be set with *e1432_set_decimation_filter*.

RESET VALUE

After a reset, *decBw* is set to **0**, therefore leading to a full span of FS/2.56

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_span, e1432_set_decimation_filter

NAME

e1432_set_decimation_filter − Set most decimation filter parameters

SYNOPSIS

```
SHORTSIZ16 e1432_set_decimation_filter(E1432ID hw, SHORTSIZ16 ID,
                                       SHORTSIZ16 output,
                                       SHORTSIZ16 state,
                                       SHORTSIZ16 decBw)
```

DESCRIPTION

This function is provided only for backward compatibility with the E1431 Host Interface Library. All new code should use *e1432_set_span*, *e1432_set_decimation_output*, and *e1432_set_anti_alias_digital* instead.

e1432_set_decimation_filter sets all of the parameters associated with the decimation filter section of an E1432 or group of E1432s, except the filter settling time.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained by a call to *e1432_create_channel_group*, or the ID of a single channel.

output selects the type of output from the decimation filter. Using **E1432_ONEPASS** for this parameter selects the output of the last filter in the chain. This is the normal operating mode of the filter. Using **E1432_MULTIPASS** causes an output consisting of the time multiplexed outputs of all cascaded filters equal to or narrower than the programmed bandwidth. This parameter may also be set with *e1432_set_decimation_output*.

state determines the state of the digital anti-alias filter. **E1432_ANTI_ALIAS_DIGITAL_ON** enables it, and **E1432_ANTI_ALIAS_DIGITAL_OFF** bypasses it. This parameter may also be set with *e1432_set_anti_alias_digital*.

decBw selects the bandwidth of the filter and the amount of decimation applied to the signal. The possible values for this parameter range from 0 to 9. This parameter may also be set with *e1432_set_decimation_bandwidth*, but a better way to set the amount of decimation is to call *e1432_set_span*.

RESET VALUE

After a reset, *state* is set to **E1432_ANTI_ALIAS_DIGITAL_ON**, *output* is set to **E1432_ONEPASS**, and *decBw* is set to **0** (i.e. maximum span).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_anti_alias_digital, e1432_set_decimation_bandwidth, e1432_set_decimation_output, e1432_set_span

NAME

e1432_set_decimation_output − Set single or multi-pass filter output
e1432_get_decimation_output − Get current state of filter output

SYNOPSIS

```
SHORTSIZ16 e1432_set_decimation_output(E1432ID hw, SHORTSIZ16 ID,
                                       SHORTSIZ16 output)
SHORTSIZ16 e1432_get_decimation_output(E1432ID hw, SHORTSIZ16 ID,
                                       SHORTSIZ16 *output)
```

DESCRIPTION

e1432_set_decimation_output sets the filter output, of a single channel or group of channels *ID*, to single or multi-pass, based on the value given in *output*.

e1432_get_decimation_output returns the current status of the filter output, of a single channel or group of channels *ID*, into a memory location pointed to by *output*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

output selects the type of output from the digital filters. The digital filters consist of a cascaded chain of sections, each decimating the data stream by a factor of two and reducing its bandwidth by a factor of two.

Using **E1432_ONEPASS** for this parameter selects the output of the last filter in the chain. This is the normal operating mode of the filter.

Using **E1432_MULTIPASS** causes an output consisting of the time multiplexed outputs of all cascaded filters equal to or narrower than the programmed bandwidth. This mode is useful when gathering data for synchronous analysis, or octave measurements.

The E1431 Host Interface library, but not the E1432 Host Interface library, allows an addition value for *output* which enables both multi-pass and oversampled data. For the E1432 Host Interface library, this can be achieved by using **E1432_MULTIPASS**, and then calling *e1432_set_decimation_oversample* to enable oversampled data.

This parameter may also be set with *e1432_set_decimation_filter*.

When in the multi-pass mode on an E1432, the data size must be one of the 32-bit sizes (see *e1432_set_data_size*). If the data size is set to **E1432_DATA_SIZE_32_SERV**, then each data sample is tagged with a 4 bit *passcount*. The passcount is placed over the least significant data bits (bits 0 to 3) of the 32-bit data sample. This passcount can be used to decode which samples come from which span of data. On an E1433, you are not required to use the 32-bit data size when in multi-pass mode, but generally multi-pass mode is only useful when the data size is **E1432_DATA_SIZE_32_SERV**.

Note: On an E1432, when operating in multi-pass mode and oversampled mode, the top span is limited to 10kHz and the number of channels is limited to eight. At spans below 10kHz the full 16 channels can be used.

RESET VALUE

After a reset, *output* is set to **E1432_ONEPASS**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_data_size, e1432_set_decimation_oversample, e1432_set_anti_alias_digital

NAME

e1432_set_decimation_oversample − Set digital filter oversample
e1432_get_decimation_oversample − Get digital filter oversample

SYNOPSIS

```
SHORTSIZ16 e1432_set_decimation_oversample(E1432ID hw, SHORTSIZ16 ID,
                                           SHORTSIZ16 oversample)
SHORTSIZ16 e1432_get_decimation_oversample(E1432ID hw, SHORTSIZ16 ID,
                                           SHORTSIZ16 *oversample)
```

DESCRIPTION

e1432_set_decimation_oversample specifies whether the digital filters of a set of channels should produce oversampled output.

e1432_get_decimation_oversample returns the current status of the filter oversample, of a single channel or group of channels *ID*, into a memory location pointed to by *oversample*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

oversample selects whether the digital filters should produce oversampled output. **E1432_DECIMATION_OVERSAMPLE_OFF** specifies non-oversampled output, which is what is normally desired. **E1432_DECIMATION_OVERSAMPLE_ON** specifies output that is oversampled by a factor of two. This oversampling is only possible if the span is less than the top span for the current clock frequency.

Note: On an E1432, when operating in multi-pass mode and oversampled mode, the top span is limited to 10kHz and the number of channels is limited to eight. At spans below 10kHz the full 16 channels can be used.

RESET VALUE

After a reset, *oversample* is set to **E1432_DECIMATION_OVERSAMPLE_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_decimation_output, e1432_set_span, e1432_set_clock_freq

NAME

e1432_set_decimation_undersamp − Set digital filter undersample amount
e1432_get_decimation_undersamp − Get digital filter undersample amount

SYNOPSIS

```
SHORTSIZ16 e1432_set_decimation_undersamp(E1432ID hw, SHORTSIZ16 ID,
                                          SHORTSIZ16 undersamp)
SHORTSIZ16 e1432_get_decimation_undersamp(E1432ID hw, SHORTSIZ16 ID,
                                          SHORTSIZ16 *undersamp)
```

DESCRIPTION

This function is not needed by most normal users. It works only for the E1433 module, is useful only in certain specific applications, and can reduce the alias protection normally provided by the E1433 module.

e1432_set_decimation_undersamp causes the input ADC to get run at a slower clock frequency than normal. The input ADC clock frequency is normally specified by *e1432_set_clock_freq*. When *e1432_set_decimation_undersamp* is used, the normal clock frequency is divided by the undersample amount.

The slower "undersampled" clock frequency means that data is sampled at a slower rate, reducing the amount of decimation filtering that must be done by the input signal processors to produce data at the span specified by *e1432_set_span*. If the undersample amount is equal to the total amount of decimation that must be done, then the input signal processors will not have to do any decimation filtering. This setting has the advantage of not running the data through the normal non-linear-phase decimation filters, which can be useful for time-domain measurements.

e1432_get_decimation_undersamp returns the current amount of the undersampling, of a single channel or group of channels *ID*, into a memory location pointed to by *undersamp*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

undersamp specifies the amount of undersampling. A value of 1 means no undersampling. Higher values specify more undersampling. This undersampling is supported only the the E1433 module. A value greater than 1 will result in an error on an E1432 module.

The *undersamp* amount will be rounded up to the next highest power of two. The resulting undersample amount must not be larger than 16, or an error will be produced. In addition, if the undersample reduces the effective ADC clock frequency to less than 7812.5 Hz (the minimum specified by the ADC manufacturer), an error will be produced.

The undersample amount must not be larger than the amount of decimation needed to produce data at the current span. The undersampling can be done when the module is producing "oversampled" data (see *e1432_set_decimation_oversample*). In this case, the undersample amount must be *less* than the amount of decimation needed to produce data at the current span, so that the input decimation filters run at least once, so that oversampled data can be produced.

As an example, suppose the clock frequency is 65536 Hz and the current span is 6400 Hz. The data rate is 6400 * 2.56 = 16384 samples per second, and the decimation filters must do two passes of decimate-by-

two, for a total decimation of four, to produce that. In this case, the undersample value can be 1, 2, or 4. The value of 4 causes no decimation filtering to take place. If oversample is turned on, then the undersample value must be either 1 or 2.

Undersampling by more than one has the following limitations and drawbacks:
* It works only on E1433 modules
* No zoom can be done
* Alias protection is reduced, potentially by a great deal

Because of these drawbacks, this setting is probably only useful in special circumstances.

Even if the undersample amount prevents the non-linear-phase decimation filters from being used, the ADC itself has a linear-phase digital filter which can't be disabled. This is similar to all Delta-Sigma converters, which all have a built-in anti-alias filter.

Because these ADC filters are linear-phase, they produce less distortion of the input time waveform than typical non-linear-phase decimation filters. For this reason, they are generally preferred for time-domain applications.

The undersampling reduces alias protection, but in ways that may not be a problem for many applications. The aliasing is **only** at 32 times the effective sample clock frequency, due to the ADC's built-in digital filter.

For example, suppose the sample clock frequency is set to 65536 Hz, and the undersample amount is 2. The effective sample rate of the data produced by the input ADCs is then 32768 Hz. Any potential alias products would be at 32 * 32768, which is about 1 MHz. Most normal measurement signals have no energy at this frequency, so the aliasing is probably not a problem.

Furthermore, even if there was energy out at this frequency, it is still somewhat attenuated by the analog anti-alias filters. The amount of attenuation varies with clock frequency. Typical alias attenuation to expect is given in the following table.

Typical Undersample Alias Attenuation

Effective Clock Freq	Minimum Alias Freq	Typical Alias Protection
65536 Hz	2 MHz	>90 dB
32768 Hz	1 MHz	70 dB
16384 Hz	524 kHz	35 dB
8192 Hz	262 kHz	0 dB

RESET VALUE

After a reset, *undersamp* is set to 1.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_decimation_output, e1432_set_decimation_oversample, e1432_set_span, e1432_set_clock_freq, e1432_set_anti_alias_digital

NAME

e1432_set_duty_cycle − Set source burst duty cycle
e1432_get_duty_cycle − Get source burst duty cycle

SYNOPSIS

```
SHORTSIZ16 e1432_set_duty_cycle(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 duty_cycle)
SHORTSIZ16 e1432_get_duty_cycle(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 *duty_cycle)
```

DESCRIPTION

e1432_set_duty_cycle sets the burst duty cycle for a source channel. This duty cycle is used only when the source mode (set by *e1432_set_source_mode*) is a burst mode, such as **E1432_SOURCE_MODE_BSINE**, or **E1432_SOURCE_MODE_BRAND**.

e1432_get_duty_cycle returns the current duty cycle for a single channel or group of channels *ID*, into a memory location pointed to by *duty_cycle*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

duty_cycle specifies the duty cycle for a burst. The duty cycle is a fraction that must be between zero and one. The total burst period is equal to the current blocksize; the duty cycle specifies the fraction of that burst that the source is on. If the source ramp rate is non-zero, the source will ramp up and down *within* the duty-cycle portion of the burst period.

Obviously, this function is not useful when talking to input or tach channels. Only source channels have a burst duty cycle.

RESET VALUE

After a reset, *duty_cycle* is set to 0.5.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_blocksize, e1432_set_ramp_rate, e1432_set_source_mode

NAME

e1432_set_enable − Enable or disable data for an input channel or group
e1432_get_enable − Get group or channel enable value

SYNOPSIS

```
SHORTSIZ16 e1432_set_enable(E1432ID hw, SHORTSIZ16 ID,
                            SHORTSIZ16 data_type, SHORTSIZ16 setting)
SHORTSIZ16 e1432_get_enable(E1432ID hw, SHORTSIZ16 ID,
                            SHORTSIZ16 data_type, SHORTSIZ16 *setting)
```

DESCRIPTION

This function enables or disables data from an input channel. If data is enabled, then *e1432_block_available* specifies when data is available and the data transfer functions (*e1432_read_xxx_data*) are used to read the data. If data is disabled, data from the specified channel is not made available to the host computer.

This parameter can be changed while a measurement is running, to allow the host computer to look at only some of the data being collected by the E1432 module. Compare this with *e1432_set_active*, which completely enables or disables a channel and which can't be changed while a measurement is running.

e1432_get_enable returns the current enable value for a channel or group.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

If the *ID* is a channel ID, that one channel is set to *setting*.

data_type specifies what type of data is being enabled or disabled. The valid values are **E1432_ENABLE_TYPE_TIME**, **E1432_ENABLE_TYPE_FREQ**, **E1432_ENABLE_TYPE_RESAMPLE** and **E1432_ENABLE_TYPE_ORDER**. The type of data that is available for transfer to the host is determined by the *e1432_set_calc_data* function, which must be called before the start of a measurement. If the data type has not been made available with this function, then enabling its transmission to the host will be ignored. For instance, if resampling calculations are not being done for an input channel, then the enable value for E1432_ENABLE_TYPE_RESAMPLE for that channel will be ignored.

The *setting* can be either **E1432_ENABLE_ON** or **E1432_ENABLE_OFF**. The default value is **E1432_ENABLE_ON**.

This parameter is used only by input channels. Attempting to set or change it for tach or source channels will result in an error.

If a particular **E1432_ENABLE_TYPE_xxx** is set to **E1432_ENABLE_ON**, then the user wants to receive data of that type for the channels referred to by the *ID*. If the **E1432_ENABLE_TYPE_xxx** is set to **E1432_ENABLE_OFF**, then the user does not want to receive data of that type for the channels referred to by the *ID*.

This enable setting for a particular channel is used only if the channel is active, as set by *e1432_set_active*. (Inactive input channels never produce data, are not sent to local bus, and can't trigger.) Changing active/inactive on a channels stops a measurement. In contrast, the enable setting can be changed while a measurement is running without stopping the measurement.

The "enable" setting for a particular channel is used only for VME data transfers. The enable setting is completely ignored for local bus transfers, so all active channels are always transferred to the local bus.

The "enable" setting has no effect on input triggering - an input can trigger whether data is enabled or not.

If a measurement is running, any changes to the "enable" setting take place immediately. (If the E1432 is in the MIDDLE of transferring a scan of data to the host, then an error is generated and the changes to the "enable" settings are not made.) If the E1432 has already asserted block available but has not started transferring data to the host, then the change *does* take place immediately.

If *e1432_set_enable* is used to disable all data from all channels in a group, then block available will no longer get asserted in **E1432_IRQ_STATUS2_REG**, and *e1432_block_available* will return 0 for any channel that is disabled.

If interrupts are being used for data transfer, then interrupts should probably be blocked around the call to *e1432_set_enable*, so that the interrupt handler can know if the new enable settings are in effect.

The E1432 module ignores the "enable" settings when deciding whether to assert an interrupt. This means that a BLOCK READY interrupt will happen even when all data from all channels is disabled. This may seem undesirable at first, but it is actually quite useful when doing interrupt-driven multi-module measurements. Because the interrupt will happen even if all channels in a module are disabled, a single module can be used to generate the BLOCK READY interrupts, even when all of the channels in that module are disabled. It is the responsibility of the application to **not** call *e1432_read_xxx_data* when there is not actually any data available. Note that *e1432_block_available* will correctly return 0 for the disabled channels, and 0 for the group ID, so this function can be used inside an interrupt handler to determine if data is really available.

CORNER CASES

Unfortunately, there are a couple corner cases where the data enable functions don’t work as cleanly as they should, and unfortunately it is difficult to explain these corner cases in a simple way. Fortunately, these really are corner cases which typical applications will probably never see or care about.

These corner cases are an issue only if the application tries to disable *all* data from *all* channels in one of the modules of a multi-module measurement. **Single-module measurements are not affected**.

Corner Case 1

If an application disables all data from all channels of a module in a multi-module measurement, then there is no data for the host application to read out of that module. But the module must still remain synchronized with other modules in the multi-module measurement that do have enabled data. To ensure synchronization, the *e1432_block_available* function does some behind-the-scenes communication with the modules. When a scan of data is ready, the *e1432_block_available* function tells the completely disabled module to pretend that data was just read from it. Then, when the application reads data from the other modules in the measurement, all the modules will stay synchronized.

This is all fine, but it means that **if there are any completely disabled modules, once** *e1432_block_available* **has returned true (and has therefore told completely disabled modules to act like data was already read), then it is too late to change the enabled channels**. If an application tries to enable data from the completely disabled module at this time, the modules may not stay synchronized, and possibly the measurement may hang waiting for the next block available.

One reason this is not typically a problem is that applications typically read data immediately after calling *e1432_block_available*, if the function tells them that data is ready.

Corner Case 2

If an application disables all data from all channels of all modules in multi-module measurement, then there

is no data for the host to read at all. In this case, *e1432_block_available* will always return zero, because no data is ever ready for the host.

However, the modules still need to stay synchronized, so that when data is eventually enabled by the application, that data is synchronized properly.

For a multi-module measurement, this synchronization is performed by *e1432_block_available*. **So it is necessary for the application to regularly call** *e1432_block_available* **even though no data is enabled**.

One reason this is not typically a problem is that few applications actually try to disable all data from all channels. Another reason this is not typically a problem is that many application already call *e1432_block_available* frequently. Even an interrupt-driven application will not generally have problems with this, because the modules will continue to generate data available interrupts even though no data is enabled, so the interrupt handler will typically end up calling *e1432_block_available*.

RESET VALUE

See above.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_block_available, e1432_create_channel_group, e1432_set_enable, e1432_e1431_diff, e1432_set_calc_data

NAME

e1432_set_fifo_size − Set data fifo size e1432_get_fifo_size − Get current data fifo size e1432_get_fifo_size_current_max − Get current maximum fifo size

SYNOPSIS

```
SHORTSIZ16 e1432_set_fifo_size(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 fifo_size)
SHORTSIZ16 e1432_get_fifo_size(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 *fifo_size)
SHORTSIZ16 e1432_get_fifo_size_current_max(E1432ID hw, SHORTSIZ16 ID,
                                           LONGSIZ32 *max)
```

DESCRIPTION

e1432_set_fifo_size sets the data fifo size, of a single channel or group of channels *ID*, to the value given in *fifo_size*.

e1432_get_fifo_size returns the current value of the measurement fifo_size, of a single channel or group of channels *ID*, into a memory location pointed to by *fifo_size*.

e1432_get_fifo_size_current_max returns the maximum valid value for the fifo size, given the amount of DRAM available, the current number of active channels, and the current settings of the calc data, data size, data port, append status, and fifo size parameters. The value is returned into the memory location pointed to by *max*. If the *ID* is a group ID, then *e1432_get_fifo_size_current_max* returns the minimum of the maximum fifo sizes of the modules in the group.

This same "current maximum fifo size" can also be used to determine the current maximum value for the pre-trigger delay (set by *e1432_set_trigger_delay*). In addition, the maximum value for the local bus transfer size (set by *e1432_set_xfer_size*) is equal to the maximum fifo size, minus enough room for a data trailer if data trailers are enabled.

To get the maximum value for blocksize, use *e1432_get_blocksize_current_max*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

fifo_size selects the number of sample points in each active channel's data fifo. The minimum valid value is zero, which means to make the data fifo as large as possible. If the value is not zero, it must be at least as large as the *blocksize* (see *e1432_set_blocksize*). The maximum valid value depends on how much RAM is available and how many channels are active in the module, and can be found by using *e1432_get_fifo_size_current_max*.

If the *fifo_size* is not zero, it is rounded up to the next power of two greater than or equal to the specified value.

If this parameter is changed while a measurement is running, it will not have any effect until the start of the next measurement.

RESET VALUE

After a reset, the *fifo_size* is set to **zero**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_blocksize, e1432_get_blocksize_current_max, e1432_get_fifo_size_limits, e1432_set_trigger_delay, e1432_set_xfer_size

NAME

e1432_set_filter_freq − Set filter frequency of E1432 channels
e1432_get_filter_freq − Get filter frequency of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_set_filter_freq(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 filter_freq)
SHORTSIZ16 e1432_get_filter_freq(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *filter_freq)
```

DESCRIPTION

e1432_set_filter_freq sets the filter frequency, of a single channel or group of channels *ID*, to the value given in *filter_freq*. Normally this is the cutoff frequency of the analog filter on the output of a source, or the cutoff frequency of an input filter on a charge input.

e1432_get_filter_freq returns the current value of the filter frequency, of a single channel or group of channels *ID*, into a memory location pointed to by *filter_freq*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

filter_freq is the filter frequency in Hertz.

For input channels, filter frequency is not generally used. Most input channels have a fixed analog filter that can't be adjusted. However, the Charge Break-out Box has a 2 kHz input filter available on the charge input. This function can be used to switch this 2 kHz filter in and out.

For source channels, the filter frequency generally specifies the cutoff frequency of the analog filter at the output of the source. For the Option 1D4 single-channel source board, this function can be used to switched between a 6.4 kHz and a 25.6 kHz output filter.

For tach channels, filter frequency is not generally used.

RESET VALUE

After a reset, the *filter_freq* is set to the minimum valid filter frequency for each channel, except source channels which are set to 25.6 kHz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_filter_freq_limits, e1432_bob

NAME

e1432_set_filter_settling_time − Change default filter settling time
e1432_get_filter_settling_time − Get current filter settling time

SYNOPSIS

```
SHORTSIZ16 e1432_set_filter_settling_time(E1432ID hw, SHORTSIZ16 ID,
                                          FLOATSIZ32 setlTime)
SHORTSIZ16 e1432_get_filter_settling_time(E1432ID hw, SHORTSIZ16 ID,
                                          FLOATSIZ32 *setlTime)
```

DESCRIPTION

e1432_set_filter_settling_time sets the filter settling time, of a single channel or group of channels *ID*, to the value given in *setlTime*.

e1432_get_filter_settling_time returns the current value of the filter settling time, of a single channel or group of channels *ID*, into a memory location pointed to by *setlTime*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

setlTime is the time given to the filter to settle its output. This time has an impact on the time spent in the **SETTLING** state of the measurement loop: The E1432 will wait for that time to elapse, before moving to the **IDLE** state. This parameter is given in seconds, and may be set to any value between **0.0** seconds, and **10,000.0** seconds.

The actual settling time will be the greater of *setlTime* and the internally computed settling time, which accounts for ADC settling and digital filter settling.

The two primary uses for this functionality are to handle AC coupling settling and Octave band settling, both of which are not handled by the internally computed settling time, due to the large times which may be involved.

AC coupling settling for the E1432 front ends should be set to 2.2 seconds for full protection.

Worst case AC coupling settling for the E1433 should be 5 seconds for the slewing of the active coupling plus 2.1/*coupling_freq*, where *coupling_freq* is set by the *e1432_set_coupling_freq* function.

Settling for the lowest octave band, *octave_start_freq*, as set by *e1432_set_octave_start_freq*, should be 7.1/*octave_start_freq* for full octave measurements and 21.6/*octave_start_freq* for one third octave measurements. Settling in exponential Octave average mode, as set by *e1432_set_octave_avg_mode*, should be an additional 7 times the exponential average time constant, as set by *e1432_set_octave_time_const*. Allowing for settling is of particular importance to obtaining accurate measurement results for the Octave hold modes, as set by *e1432_set_octave_hold_mode*.

RESET VALUE

After a reset, *setlTime* is set to **0.0** samples.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_filter_settling_time_limits

NAME

e1432_set_input_high − Set source of input signal
e1432_get_input_high − Get current source of input signal

SYNOPSIS

```
SHORTSIZ16 e1432_set_input_high(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 high)
SHORTSIZ16 e1432_get_input_high(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *high)
```

DESCRIPTION

e1432_set_input_high sets the input to the ADC, of a single channel or group of channels *ID*, to the value given in *high*.

e1432_get_input_high returns the current input high selection, of a single channel or group of channels *ID*, into a memory location pointed to by *high*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

high controls the analog input to the input channel. **E1432_INPUT_HIGH_NORMAL** selects the front panel connector. **E1432_INPUT_HIGH_GROUNDED** grounds the input. **E1432_INPUT_HIGH_CALIN** selects the module's CALIN line. **E1432_INPUT_HIGH_BOB_CALIN** selects the module's CALIN line via the cal connection in a break-out box. **E1432_INPUT_HIGH_CALOUT** is the same as **E1432_INPUT_HIGH_NORMAL**, except that the input is also connected to the module's CALOUT line.

For backwards compatibility with the E1431 host interface library, the following obsolete values for *high* are also accepted: **E1432_INPUT_SOURCE_BNC**, which is the same as **E1432_INPUT_HIGH_NORMAL**; **E1432_INPUT_SOURCE_SUMBUS**, which is the same as **E1432_INPUT_HIGH_CALIN**; and **E1432_INPUT_SOURCE_ZERO**, which is the same as **E1432_INPUT_HIGH_GROUNDED**.

On input channels, only **NORMAL**, **GROUNDED**, **CALIN**, and **BOB_CALIN** are supported. The **BOB_CALIN** value is valid only when a smart break-out box (such as a Charge Break-out Box or Microphone Break-out Box) is connected to the input.

On tach channels, only **NORMAL** and **CALOUT** are supported. The **CALOUT** setting can be used to monitor the analog signal connected to a tach input. To do this, the module's CALOUT line must be connected back to the module's CALIN line, using *e1432_set_calin*. In addition, an input channel must be told to monitor the CALIN line by using *e1432_set_input_high* on the input channel. Then the data collected by the input channel will show the signal on the tach input. Because there is only one CALOUT line, only one tach channel may drive the CALOUT line at any one time.

It is also possible to set up a tach channel to drive the VXI sumbus, and have the sumbus drive CALIN, and have an input channel monitor CALIN. This allows an input channel to monitor a tach channel from a different E1432 module, but results in a larger DC offset error and more noise. To make this work properly, pass **E1432_CALIN_SUMBUS_TACH** (not **E1432_CALIN_SUMBUS**) to *e1432_set_calin*, to ensure that the input channel can figure out the correct scale factor.

When monitoring tach signals via the CALIN line using either of the above methods, the tach signal bypasses most of the input range setting hardware. Because of this, the tach signal level that can be monitored is limited.

On an E1432 51.2 kHz input SCA, the maximum tach signal level when the tach trigger level is zero is about 6 volts. On an E1433 196 kHz input SCA, the maximum tach signal level when the tach trigger level is zero is about 2 volts. If the tach trigger level is larger than +-4.9 volts, then the tach board attenuates the tach signal, resulting in a five times greater range that can be monitored (30 volts for E1432, 10 volts for E1433).

This parameter is not used for source channels.

RESET VALUE

After a reset, *high* is set to **E1432_INPUT_HIGH_NORMAL**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_scale, e1432_set_analog_input, e1432_set_calin

NAME

e1432_set_input_low − Set grounding mode of input signal
e1432_get_input_low − Get grounding mode of input signal

SYNOPSIS

```
SHORTSIZ16 e1432_set_input_low(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 low)
SHORTSIZ16 e1432_get_input_low(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *low)
```

DESCRIPTION

e1432_set_input_low sets the grounding mode, of a single channel or group of channels *ID*, to the value given in *low*.

e1432_get_input_low returns the current input low selection, of a single channel or group of channels *ID*, into a memory location pointed to by *low*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

low controls the analog input to the input channel. **E1432_INPUT_LOW_GROUNDED** grounds the low-side of the channel. **E1432_INPUT_LOW_FLOATING** lets the low-side of the channel float, making the input a differential input.

For backwards compatibility with the E1431 host interface library, the following obsolete values for *low* are also accepted: **E1432_INPUT_GROUNDED**, which is the same as **E1432_INPUT_LOW_GROUNDED**; and **E1432_INPUT_FLOATING**, which is the same as **E1432_INPUT_LOW_FLOATING**.

Only input channels support the setting the grounding mode - source and tach channels are never floating, and are always grounded.

If a "smart" break-out box is attached to the channel, such as the E3242A Charge break-out box or the E3243A Microphone break-out box, then this function will tell the break-out box to set the grounding appropriately.

However, if a passive break-out box is attached (or no break-out box), there is no way to tell the break-out box what grounding to use, nor is there any way to ask the break-out box what the current grounding mode is. In this case, the parameter to *e1432_set_input_low* is accepted and simply saved - no hardware settings are changed and no errors are generated. In this case, a call to *e1432_get_input_low* will return the most recent value given to *e1432_set_input_low*.

RESET VALUE

After a reset, *low* is set to **E1432_INPUT_LOW_FLOATING**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_input_high

NAME

e1432_set_input_mode − Set input mode to volt or ICP
e1432_get_input_mode − Get current state of input mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_input_mode(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_input_mode(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_input_mode sets the input mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_input_mode returns the current value of the input mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode determines the input mode in the front end. **E1432_INPUT_MODE_VOLT** sets the volt input mode. **E1432_INPUT_MODE_ICP** sets the ICP input mode. **E1432_INPUT_MODE_CHARGE** sets the input mode to charge-amp mode. **E1432_INPUT_MODE_MIC** sets the input mode to microphone mode. **E1432_INPUT_MODE_MIC_200V** sets the input mode to microphone mode with 200V supply turned on.

This parameter may also be set with *e1432_set_analog_input*.

If there is not a smart break-out box connected to the channel, then changing the input mode for one channel causes the input mode for all channels within that SCA to change. If there is a smart break-out box present, then the input mode applies only to the specific channels specified.

E1432_INPUT_MODE_CHARGE is valid only if a Charge break-out box is attached to the channel specified. In this mode, the full-scale setting is controlled by the *e1432_set_range_charge* function, which specifies the full scale in picoCoulombs. In this mode, all input data for this channel is in terms of picoCoulombs rather than volts. See *e1432_bob* for more information about break-out boxes.

E1432_INPUT_MODE_MIC and **E1432_INPUT_MODE_MIC_200V** are valid only if a Microphone break-out box is attached to the channel specified. In these two modes, the full-scale setting is controlled by the *e1432_set_range_mike* function. See *e1432_bob* for more information about break-out boxes.

There is not actually an ICP current source inside the E1432/E1433 input SCAs. Instead, a breakout box containing an ICP current source can be attached to the input. When this ICP breakout box is attached, then setting the *mode* to **E1432_INPUT_MODE_ICP** enables the ICP current source in the breakout box. If there is no ICP breakout box attached to the input, then setting **E1432_INPUT_MODE_ICP** does nothing.

RESET VALUE

After a reset, *mode* is set to **E1432_INPUT_MODE_VOLT**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_analog_input, e1432_set_range, e1432_set_range_charge, e1432_set_range_mike, e1432_bob

NAME

e1432_set_input_offset − Set input offset voltage
e1432_get_input_offset − Get current input offset voltage

SYNOPSIS

```
SHORTSIZ16 e1432_set_input_offset(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 input_offset)
SHORTSIZ16 e1432_get_input_offset(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *input_offset)
```

DESCRIPTION

e1432_set_input_offset sets the input offset voltage, of a single channel or group of channels *ID*, to the value given in *input_offset*.

e1432_get_input_offset returns the current value of the input offset voltage, of a single channel or group of channels *ID*, into a memory location pointed to by *input_offset*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

input_offset is the desired input offset in volts. This voltage is subtracted from the input signal, and can be used to remove a constant DC level from an input signal. This can be used instead of AC coupling if the signal's DC level is known ahead of time, thus avoiding AC filter settling times. It can also be used in addition to AC coupling, to help extend the range of the AC coupling filter and to help the AC coupling filter settle more quickly.

The input offset voltage is used only for input channels - source and tach channels do not have this feature.

The input offset voltage is only controllable on E1433 196 kHz input channels. E1432 51.2 kHz input channels do not have hardware to implement this feature. Any attempt to set the input offset voltage of an E1432 input channel will generate an error. However, *e1432_get_input_offset* will return 0.0 for E1432 input channels.

The input offset voltage is settable over a continuous range of 0 volts to 20 volts. The resolution is approximately 5 mV. Negative voltages are not valid. The E1433 196 kHz input hardware will clip the input signal if the signal exceeds about 21 volts, regardless of the input offset voltage setting. Unfortunately, this clipping is not detected by the hardware, so there is no overload indication in this case.

The input offset voltage is normally only useful when the input mode is voltage or ICP mode (see *e1432_set_input_mode*). In microphone or charge input modes, there is typically no offset that needs to be subtracted out. This function will still "work" when the input mode is microphone or charge mode, however the voltage that is subtracted out will not necessarily match the parameter that is applied, due to scaling performed by the break-out box. Therefore, it's typically best to set the input offset voltage to zero in microphone or charge modes.

RESET VALUE

After a reset, input channels have the *input_offset* set to 0 volts.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_input_offset_limits, e1432_set_coupling, e1432_set_input_mode

NAME

e1432_set_internal_debug − Set E1432 internal firmware debug level
e1432_get_internal_debug − Get E1432 internal firmware debug level

SYNOPSIS

```
SHORTSIZ16 e1432_set_internal_debug(E1432ID hw, SHORTSIZ16 ID,
                                    LONGSIZ32 lev)
SHORTSIZ16 e1432_get_internal_debug(E1432ID hw, SHORTSIZ16 ID,
                                    LONGSIZ32 *lev)
```

DESCRIPTION

e1432_set_internal_debug sets the internal debug level of the firmware executing in the E1432 modules referred to by *ID*.

e1432_get_internal_debug returns the current value of the internal firmware debug level, in the memory location pointed to by level.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

lev is the firmware debug level.

The firmware debug level determines how much information the internal firmware tries to print out as it executes. This use useful only for developers, and even then it is not always useful.

The internal debug value is a bit-mask, where each bit has a separate meaning. As the firmware gets optimized, some of these debugging bits have been eliminated. At this point, there are only a few bits that are still useful:

Internal Debug Bit Definitions	
Bit	Meaning
0x00000200	Print host commands and parameters
0x00000400	Print measurment state changes
0x00004000	Print source driver info
0x00008000	Print source register accesses

The printout from the firmware goes to a buffer inside the E1432 module. If nothing is monitoring this buffer, then the printout is just ignored and there is no point in setting the firmware debug level. Use the *e1432mon* program to monitor the printout buffer.

RESET VALUE

After a reset, *lev* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_debug_level, e1432_trace_level, e1432_print_errors, e1432_get_internal_debug_limits

NAME

e1432_set_interrupt − Set all interrupt parameters

SYNOPSIS

```
SHORTSIZ16 e1432_set_interrupt(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 priority, SHORTSIZ16 mask)
```

DESCRIPTION

e1432_set_interrupt sets all parameters associated with the interrupt capability of an E1432 or group of E1432s.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

priority specifies which of the seven VME interrupt lines to use; it is not actually a priority at all, it just specifies which line to use. The only legal values are 0 through 7. Specifying 0 turns the interrupt off. This parameter may also be set with *e1432_set_interrupt_priority*.

mask specifies the mask of events on which to interrupt. This mask is created by ORing together various condition bits. Refer to *e1432_set_interrupt_mask* for the definition of the conditions which may be part of the mask. This parameter may also be set with *e1432_set_interrupt_mask*.

RESET VALUE

After a reset, *priority* is set to **0** (none), and *mask* is set to **0** (all causes masked).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_reenable_interrupt, e1432_set_interrupt_mask, e1432_set_interrupt_priority

NAME

e1432_set_interrupt_mask − Set interrupt mask
e1432_get_interrupt_mask − Get current interrupt mask

SYNOPSIS

```
SHORTSIZ16 e1432_set_interrupt_mask(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 mask)
SHORTSIZ16 e1432_get_interrupt_mask(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 *mask)
```

DESCRIPTION

e1432_set_interrupt_mask sets the interrupt mask, of a single channel or group of channels *ID*, to the value given in *mask*.

e1432_get_interrupt_mask returns the current value of the interrupt mask, of a single channel or group of channels *ID*, into a memory location pointed to by *mask*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mask specifies the mask of events on which to interrupt. This parameter may also be set with *e1432_set_interrupt*. This mask is created by ORing together the various conditions defined in the following table (this table is also found on the *e1432_intr(5)* manual page).

Interrupt Mask Bit Definitions	
Define (in e1432.h)	Description
E1432_IRQ_BLOCK_READY	Scan of data ready in FIFO
E1432_IRQ_MEAS_ERROR	FIFO overflow or tach buffer overflow
E1432_IRQ_MEAS_STATE_CHANGE	Measurement state machine changed state
E1432_IRQ_MEAS_WARNING	Measurement warning
E1432_IRQ_OVERLOAD_CHANGE	Overload status changed
E1432_IRQ_SRC_STATUS	Source channel interrupt
E1432_IRQ_TACHS_AVAIL	Raw tach times ready for transfer to other modules
E1432_IRQ_TRIGGER	Trigger ready for transfer to other modules

The **E1432_IRQ_SRC_STATUS** interrupt is used for source channel overload, overread, underrun, and shutdown. When the source is in arb data mode, this interrupt is also used for the "ready for arb data" interrupt.

The **E1432_IRQ_MEAS_ERROR** currently is used only for a FIFO overflow. This normally can happen only when the module is in continuous mode (see *e1432_set_data_mode*). This will interrupt as soon as the FIFO overflows, but note that the FIFO still has useful data in it which can still be read by the *e1432_read_xxx_data* functions. *e1432_block_available* will not indicate that a FIFO overflow has occurred until all of the remaining data is read out of the FIFO. This bit is also used to indicate a raw tach buffer overflow, which is indicated by the **E1432_STATUS2_TACH_OVERFLOW** bit being set in the status register.

A second **E1432_IRQ_MEAS_ERROR** interrupt will happen after all data has been read out of a FIFO

that previously overflowed. This can be used to tell an application that no more data will be available, so the application can stop the measurement. For this second **E1432_IRQ_MEAS_ERROR** interrupt, the **E1432_IRQ_STATUS2_REG** will have the **E1432_STATUS2_FIFO_EMPTIED** bit set in it.

Once the mask has been set, and an interrupt occurs, the cause of the interrupt is obtained by reading the **E1432_IRQ_STATUS2_REG** register. The bit position of the interrupt mask and status registers match so the defines can be used to set and check IRQ bits. The **E1432_IRQ_STATUS2_REG** may have bits set for things that are not enabled to interrupt using this *e1432_set_interrupt_mask* function, so the register value should be binary-ANDed with the mask setting if you must determine exactly what caused the interrupt.

Once an interrupt occurs, the module will not interrupt again until *e1432_reenable_interrupt()* is called.

RESET VALUE

After a reset, *mask* is set to **0** (all causes disabled).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_block_available, e1432_reenable_interrupt, e1432_set_interrupt, e1432_check_src_arbrdy, e1432_check_src_shutdown, e1432_check_src_overload, e1432_check_src_overread, e1432_check_src_underrun, e1432_intr(5), e1432_get_meas_warning

NAME

e1432_set_interrupt_priority − Set interrupt priority
e1432_get_interrupt_priority − Get current interrupt priority

SYNOPSIS

```
SHORTSIZ16 e1432_set_interrupt_priority(E1432ID hw, SHORTSIZ16 ID,
                                        SHORTSIZ16 priority)
SHORTSIZ16 e1432_get_interrupt_priority(E1432ID hw, SHORTSIZ16 ID,
                                        SHORTSIZ16 *priority)
```

DESCRIPTION

e1432_set_interrupt_priority sets which VME interrupt line to use when interrupting. It is not actually a priority level, it just specifies which line to use.

e1432_get_interrupt_priority returns the current value of the interrupt priority, of a single channel or group of channels *ID*, into a memory location pointed to by *priority*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

priority specifies which of the seven VME interrupt lines to use. The only legal values are 0 through 7. Specifying 0 turns the interrupt off. This parameter may also be set with *e1432_set_interrupt*.

RESET VALUE

After a reset, *priority* is set to **0** (none).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_interrupt, e1432_get_interrupt_priority_limits, e1432_intr(5)

NAME

e1432_set_lbus_mode − Set mode for Local Bus
e1432_get_lbus_mode − Get current mode for Local Bus

SYNOPSIS

```
SHORTSIZ16 e1432_set_lbus_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_lbus_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_lbus_mode sets the Local Bus to one of five settings: *append*, *generate*, *insert*, *pipe*, or *consume*.

e1432_get_lbus_mode returns the current setting.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode must be one of the following:

E1432_LBUS_MODE_GENERATE causes the module to ignore and throw away any data that comes in (from the left) and to send its own data out (to the right). The local bus data stream is *generated* by the module.

E1432_LBUS_MODE_PIPE causes the module to send out all data that comes in. No additional data generated by the module is added to the local bus data stream. The local bus data stream is *piped* through the module.

E1432_LBUS_MODE_INSERT causes the module to send out its own data, followed by all data that comes in. The module's data is *inserted* at the beginning of the local bus data stream.

E1432_LBUS_MODE_APPEND causes the module to send out all data that comes in, followed by its own data. The module's data is *appended* to the end of the local bus data stream.

E1432_LBUS_MODE_CONSUME causes the module to keep the data that comes in, and not send any data out. The local bus data stream is *consumed* by this module.

RESET VALUE

After a reset the Local Bus is set to **E1432_LBUS_MODE_PIPE**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_reset_lbus, e1432_set_data_port

NAME

e1432_set_max_order − Set the upper limit for resampled time data
e1432_get_max_order − Get the upper limit for resampled time data

SYNOPSIS

```
SHORTSIZ16 e1432_set_max_order(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 max)
SHORTSIZ16 e1432_get_max_order(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 *max)
```

DESCRIPTION

e1432_set_max_order sets the upper frequency limit of the resampled time data. This data is in the revolution domain. The *max* parameter is the upper limit expressed in units of a tachometer revolution (order) of the resampled data from the order tracking measurement. The max order multiplied by the tach signal frequency is the upper frequency limit of a signal to the inputs that can be reliably analysed by the order tracking algorithm. Resampled data calculation is activated by the *e1432_set_calc_data* function.

e1432_get_max_order returns the resampled time data upper limit into the variable pointed to by *max*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

max specifies the resampled time data upper frequency limit.

NOTE: When doing an order track measurement, the following relationship must hold between these parameters:

max_order <= blocksize * delta_order / 5.12

or an **ERR1432_ILLEGAL_BLOCK_ORDER_COMBO** error will be issued when the measurement starts.

RESET VALUE

After a reset, *max* is set to 10.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_max_order_limits, e1432_set_blocksize, e1432_set_calc_data, e1432_set_delta_order

NAME

e1432_set_meas_time_length − Set measurement time length for time arming mode
e1432_get_meas_time_length − Get measurement time length for time arming mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_meas_time_length(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 meas_time_length)
SHORTSIZ16 e1432_get_meas_time_length(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 *meas_time_length)
```

DESCRIPTION

e1432_set_meas_time_length sets the length of time a measurement will run in the time arming mode. It is a global parameter applying to all channels in a single module. Once started, a measurement will arm at multiples of the time interval set by the *e1432_set_meas_time_length* function until the elapsed time exceeds this measurement time length.

e1432_get_meas_time_length returns the current value of the measurement time length into a memory location pointed to by *meas_time_length*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of tach channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

meas_time_length is the total measurement time length when the arm mode is set to **E1432_ARM_TIME** by the *e1432_set_arm_mode* function.

For input channels and source channels, this parameter is not used.

RESET VALUE

After a reset, the *meas_time_length* defaults to 30.0 seconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_meas_time_length_limits, e1432_set_arm_time_interval

NAME

e1432_set_mmf_delay − Set measurement mmf_delay
e1432_get_mmf_delay − Get current measurement mmf_delay

SYNOPSIS

```
SHORTSIZ16 e1432_set_mmf_delay(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 mmf_delay)
SHORTSIZ16 e1432_get_mmf_delay(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 *mmf_delay)
```

DESCRIPTION

These functions are needed only for multi-mainframe measurements. For some multi-mainframe measurements, synchronization between mainframes can be a problem. This problem can be worked around by telling modules in the master mainframe to delay their measurement state transitions.

e1432_set_mmf_delay sets a measurement delay time, of a single channel or group of channels *ID*, to the value given in *mmf_delay*.

e1432_get_mmf_delay returns the current value of the measurement delay time, of a single channel or group of channels *ID*, into a memory location pointed to by *mmf_delay*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mmf_delay specifies the delay in 4 ms increments. This is the time that the module will delay releasing the sync/trigger line, helping to ensure synchronization between modules in a multi-mainframe measurement. A suggested value of 5 should be used for modules in the master mainframe. Larger values will provide more assurance of synchronization, but will also slow down the measurement loop.

This parameter should be zero (which is the default) in all modules in the slave mainframes. It should be non-zero only for modules in the master mainframe.

RESET VALUE

After a reset, the measurement *mmf_delay* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_multimain, e1432_arm_measure_master_finish

NAME

e1432_set_multi_sync − Set multiple module system synchronization
e1432_get_multi_sync − Get current multiple module system sync state

SYNOPSIS

```
SHORTSIZ16 e1432_set_multi_sync(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 sync)
SHORTSIZ16 e1432_get_multi_sync(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *sync)
```

DESCRIPTION

A typical measurement doesn't generally need to deal with *e1432_set_multi_sync* at all. Normally, the call to *e1432_init_measure* automatically takes care of setting the multi-module setups for all the modules in a measurement, by internally calling *e1432_set_multi_sync* for each module. This automatic setup can be disabled using the *e1432_set_auto_group_meas* function.

e1432_set_multi_sync sets the multiple module system synchronization, of a single channel or group of channels *ID*, to the value given in *sync*.

e1432_get_multi_sync returns the current value of the multiple module system synchronization, of a single channel or group of channels *ID*, into a memory location pointed to by *sync*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

sync is used to select the synchronization. The E1432 supports synchronous operation among multiple E1432s by using a VXI TTLTRG line to drive all the modules in a system from the same clock. **E1432_MULTI_SYNC_OFF** sets the SYNC to be generated locally. **E1432_MULTI_SYNC_ON** sets the module to use the SYNC line from the VXI backplane, which is selected with *e1432_set_ttltrg_satrg*. This mode uses the SYNC line for multiple module synchronization capabilities including: booting of the digital filters, synchronization of the local oscillators, arming, and triggering.

For backwards compatibility with the E1431 library, the following obsolete values for *sync* are also accepted: **E1432_SYSTEM_SYNC_OFF**, which is the same as **E1432_MULTI_SYNC_OFF**; and **E1432_SYSTEM_SYNC_ON**, which is the same as **E1432_MULTI_SYNC_ON**.

RESET VALUE

After a reset, *sync* is set to **E1432_MULTI_SYNC_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_auto_group_meas, e1432_set_ttltrg_satrg

NAME

e1432_set_octave_avg_mode − Set Octave measurement mode
e1432_get_octave_avg_mode − Get Octave measurement mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_avg_mode(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 octave_avg_mode)
SHORTSIZ16 e1432_get_octave_avg_mode(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 *octave_avg_mode)
```

DESCRIPTION

e1432_set_octave_avg_mode sets the octave average mode of the modules(s) selected to the value given in *octave_avg_mode*.

e1432_get_octave_avg_mode returns the octave mode state of the modules(s) selected into a memory location pointed to by *octave_avg_mode*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_avg_mode must be one of **E1432_OCTAVE_AVG_MODE_EXP**, which selects exponential averaging, or **E1432_OCTAVE_AVG_MODE_LIN**, which selects linear (block) averaging.

Note: If octave_avg_mode is set to exponential, and the measurement arm mode is set to time arm or rpm arm, the first trigger will start the measurement and all additional triggers will be ignored.

RESET VALUE

After a reset, *octave_avg_mode* is set to **E1432_OCTAVE_AVG_MODE_EXP**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data

NAME

e1432_set_octave_hold_mode − Set Octave hold mode
e1432_get_octave_hold_mode − Get Octave hold mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_hold_mode(E1432ID hw, SHORTSIZ16 ID,
                                      SHORTSIZ16 octave_hold_mode)
SHORTSIZ16 e1432_get_octave_hold_mode(E1432ID hw, SHORTSIZ16 ID,
                                      SHORTSIZ16 *octave_hold_mode)
```

DESCRIPTION

e1432_set_octave_hold_mode sets the octave hold mode of the modules(s) selected to the value given in *octave_hold_mode*.

e1432_get_octave_hold_mode returns the octave hold mode state of the modules(s) selected into a memory location pointed to by *octave_hold_mode*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_hold_mode must be either **E1432_OCTAVE_HOLD_MODE_OFF**, or **E1432_OCTAVE_HOLD_MODE_MAX**. **E1432_OCTAVE_HOLD_MODE_OFF** turns off Octave hold mode. **E1432_OCTAVE_HOLD_MODE_MAX** selects the maximum value hold mode.

An octave hold mode of other than **E1432_OCTAVE_HOLD_MODE_OFF** works only when the octave average mode, selected by *e1432_set_octave_avg_mode*, is set to **E1432_OCTAVE_AVG_MODE_EXP**. Otherwise, it is ignored.

The hold mode computations begin place after the settling time, as set by *e1432_set_filter_settling_time*. Correct setting of filter_settling_time is essential for valid Octave hold mode results.

For implementation reasons, the Octave results returned by *e1432_get_current_data* may be either the instantaneous Octave data or hold mode data, depending on what is most recently available.

RESET VALUE

After a reset, *octave_hold_mode* is set to **E1432_OCTAVE_HOLD_MODE_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data e1432_set_filter_settling_time

NAME

e1432_set_octave_int_time − Set Octave linear average integration time
e1432_get_octave_int_time − Get Octave linear average integration time

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_int_time(E1432ID hw, SHORTSIZ16 ID,
                                     FLOATSIZ32 octave_int_time)
SHORTSIZ16 e1432_get_octave_int_time(E1432ID hw, SHORTSIZ16 ID,
                                     FLOATSIZ32 *octave_int_time)
```

DESCRIPTION

e1432_set_octave_int_time sets the average integration time of the modules(s) selected to the value given in *octave_int_time*.

e1432_get_octave_int_time returns the average integration time of the modules(s) selected into a memory location pointed to by *octave_int_time*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_int_time must be within .001953125 seconds minimum and 10,000 seconds maximum. For values below 1 second, it is rounded to the nearest negative power of two seconds. For values above 1 second, it is rounded to the nearest second. This parameter sets the average time for an Octave linear average. For exponential averages, this parameter has no effect.

Because of internal fifo issues, larger values of *octave_int_time* will be rejected at the beginning of a measurement with the **ERR1432_ILLEGAL_OCTAVE_INT_TIME** error if data other than E1432_ENABLE_TYPE_OCTAVE, such as E1432_ENABLE_TYPE_TIME, is selected using the e1432_set_enable function. These values depend on the amount of DRAM installed as well as the number of channels enabled, but generally fall in the range of .5 to a small number of seconds. This problem can be aleviated by adjusting *octave_time_step* to cause average updates to occur below this threshold. In general, it is best to not select data other than E1432_ENABLE_TYPE_OCTAVE when using an *octave_time_step* of any significant size.

RESET VALUE

After a reset, *octave_int_time* is set to 1.0 second.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data

NAME

e1432_set_octave_meas − Set Octave measurement on/off state
e1432_get_octave_meas − Get Octave measurement on/off state

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_meas(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 octave_meas)
SHORTSIZ16 e1432_get_octave_meas(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *octave_meas)
```

DESCRIPTION

e1432_set_octave_meas turns Octave measurements on/off for the modules(s) selected.

e1432_get_octave_meas returns the Octave measurment on/off state of the modules(s) selected into a memory location pointed to by *octave_meas*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_meas must either **E1432_OCTAVE_MEAS_OFF**, to turn Octave measurements off, or **E1432_OCTAVE_MEAS_ON**, to turn Octave measurements on.

Octave measurements can only be run on **E1433** modules with the **1D1** "Real Time Octave" option installed. Error **ERR1432_OPTION_NOT_INSTALLED** is returned when the measurement is started if the **1D1** option is not installed if input channels are enabled. Error **ERR1432_OCTAVE_VS_MODULE_TYPE** is returned when the measurement is started if the the module is not an **E1433** module if input channels are enabled.

For Octave measurements to be run, the clock frequencey must be set to 65526 by *e1432_set_clock_freq*.

Octave measurements utilize a special SCA DSP downloadable, `soct.bin`, which must either be located in `/opt/e1432/lib/` (or \HPE1432\LIB\ on PC systems) if the *hwinstall (1)* program was used to download the 96000 firmware or be in the same location as the 96000 firmware file if the *e1432_install* function is used to install the 96000 firmware. Alternatively, the *e1432_install_file* function can be used to inform the library of the location of the 96000 firmware file and hence the location of the Octave SCA DSP downloadable. The standard SCA DSP downloadable is restored when measurements are begun after selecting *octave_meas* of **E1432_OCTAVE_MEAS_OFF**.

Octave measurements with a trigger_delay, set by *e1432_set_trigger_delay*, other than 0 give indeterminate results.

Time and Octave data are related by the trigger condition. A trigger begins a linear average or an exponential average, as set by *e1432_set_octave_avg_meas*. Since exponential averages run forever, only one trigger is needed for an exponential average measurement.

Note: If octave_avg_mode is set to exponential, and the measurement arm mode is set to time arm or rpm arm, the first trigger will start the measurement and all additional triggers will be ignored.

For linear averages, subsequent triggers are ignored until the time block size, as set by *e1432_set_blocksize*,

(minus the overlap, as set by *e1432_set_overlap*) AND the integration time, as set by *e1432_set_octave_int_time*, are fulfilled. So, if the Time block size (minus overlap) is greater than the integration time, there will be gaps between the Octave averages. If the Time block size (minus overlap) is less than the integration time, then the overlap between Time blocks will be less than specified; there will be gaps between the time blocks if the Time block size is less than the integration time.

For exponential averages and updates on linear averages, as set by *e1432_set_octave_time_step*, a time block is chosen to fit the Octave update rate, so the Time blocks may be overlapped or have gaps between them, regardless of what the overlap is set to. This is one case where more than one time block is available for one trigger.

Peak and RMS values are placed in the trailer when Octave measurements are running and available via *e1432_get_current_value* calls. The trailer Peak and RMS values are "filtered", the equivalent of *e1432_get_current_value* at for the frame of octave data that they are attached to. There is no equivalent of **E1432_PEAK_MODE_BLOCK** or **E1432_RMS_MODE_BLOCK**.

The Octave SCA DSP downloadable is installed in the course of execution of this function. Since it temporarily uses memory in the module which is also used for the arbitray source data, this function must preceed the function calls to pre-load the arbitrary source buffers using *e1432_write_srcbuffer_data*.

RESET VALUE

After a reset, *octave_meas* is set to **E1432_OCTAVE_MEAS_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data, e1432_set_trigger_delay, e1432_install_file

NAME

e1432_set_octave_mode − Set Octave measurement mode
e1432_get_octave_mode − Get Octave measurement mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 octave_mode)
SHORTSIZ16 e1432_get_octave_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *octave_mode)
```

DESCRIPTION

e1432_set_octave_mode sets the octave mode of the modules(s) selected to the value given in *octave_mode*.

e1432_get_octave_mode returns the octave mode state of the modules(s) selected into a memory location pointed to by *octave_mode*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_mode must be either **E1432_OCTAVE_MODE_FULL** or **E1432_OCTAVE_MODE_THIRD**. **E1432_OCTAVE_MODE_FULL** selects full (1/1) Octave measurements. **E1432_OCTAVE_MODE_THIRD** selects third (1/3) Octave measurements.

RESET VALUE

After a reset, *octave_mode* is set to **E1432_OCTAVE_MODE_THIRD**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_meas, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data, e1432_set_trigger_delay

NAME

e1432_set_octave_start_freq − Set Octave start frequency band
e1432_get_octave_start_freq − Get Octave start frequency band
e1432_set_octave_stop_freq − Set Octave stop frequency band
e1432_get_octave_stop_freq − Get Octave stop frequency band

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_start_freq(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 octave_start_freq)
SHORTSIZ16 e1432_get_octave_start_freq(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 *octave_start_freq)
SHORTSIZ16 e1432_set_octave_stop_freq(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 octave_stop_freq)
SHORTSIZ16 e1432_get_octave_stop_freq(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 *octave_stop_freq)
```

DESCRIPTION

e1432_set_octave_start_freq and *e1432_set_octave_stop_freq* set the start frequency band and stop frequency band, respectively, of the modules(s) selected to the value given in *octave_start_freq* and *octave_stop_freq*, respectively.

e1432_get_octave_start_freq and *e1432_get_octave_stop_freq* return the start frequency band and stop frequency band, respectively, of the modules(s) selected into a memory location pointed to by *octave_start_freq* and *octave_stop_freq*, respectively..

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_start_freq and *octave_stop_freq* are rounded to the nearest third octave band center, with a minimum *octave_start_freq* of 3.15 Hz and a maximum *octave_stop_freq* of 20.0 kHz. The maximum *octave_start_freq* is 10.0 kHz or *octave_stop_freq*, whichever is lesser. The minimum *octave_stop_freq* is 6.3 Hz or *octave_start_freq*, whichever is greater. When a Full Octave measurement is running, the start and stop bands are the standard Full Octave bands containing *octave_start_freq* and *octave_stop_freq*, respectively.

Since *octave_start_freq* and *octave_stop_freq* change the number of data points transferred by the e1432_read_XXXXXXX_data functions, the *e1432_get_octave_blocksize* function has been provided to supply the value need for the *size* parameter of the e1432_read_XXXXXXX_data functions.

RESET VALUE

After a reset, *octave_start_freq* is set to 3.15 Hz and *octave_stop_freq* is set to 20.0 kHz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_set_filter_settling_time, e1432_get_current_data

NAME

e1432_set_octave_time_const − Set Octave exponential average time constant
e1432_get_octave_time_const − Get Octave exponential average time constant

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_time_const(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 octave_time_const)
SHORTSIZ16 e1432_get_octave_time_const(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 *octave_time_const)
```

DESCRIPTION

e1432_set_octave_time_const sets the exponential average time constant of the modules(s) selected to the value given in *octave_time_const*.

e1432_get_octave_time_const returns the exponential average time constant value of the modules(s) selected into a memory location pointed to by *octave_time_const*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_time_const must be within .0078125 seconds minimum and 1.0 seconds maximum. It is rounded to the nearest negative power of two seconds. It is the time that it takes the the exponential Octave average power to decay by a factor of "e".

RESET VALUE

After a reset, *octave_time_const* is set to .125 seconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_step, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_set_filter_settling_time, e1432_get_current_data

NAME

e1432_set_octave_time_step − Set Octave time step
e1432_get_octave_time_step − Get Octave time step

SYNOPSIS

```
SHORTSIZ16 e1432_set_octave_time_step(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 octave_time_step)
SHORTSIZ16 e1432_get_octave_time_step(E1432ID hw, SHORTSIZ16 ID,
                                      FLOATSIZ32 *octave_time_step)
```

DESCRIPTION

e1432_set_octave_time_step sets the octave time step of the modules(s) selected to the value given in *octave_time_step*.

e1432_get_octave_time_step returns the octave time step of the modules(s) selected into a memory location pointed to by *octave_time_step*.

This parameter is a "global" parameter. It applies to an entire module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

octave_time_step must be within .001953125 seconds minimum and 10,000 seconds maximum. It is rounded to the nearest negative power of two seconds.

octave_time_step sets the average update rate for both linear and exponential averages. If it is larger than the integration time, set by the e1432_set_octave_int_time function, for a linear average, only one upate occurs, at the end of the average.

Because of internal fifo issues, larger values of *octave_time_step* (if less than octave_int_time when averaging is linear) will be rejected at the beginning of a measurement with the **ERR1432_ILLEGAL_OCTAVE_TIME_STEP** error if data other than E1432_ENABLE_TYPE_OCTAVE, such as E1432_ENABLE_TYPE_TIME, is selected using the e1432_set_enable function. These values depend on the amount of DRAM installed as well as the number of channels enabled, but generally fall in the range of .5 to a small number of seconds. In general, it is best to not select data other than E1432_ENABLE_TYPE_OCTAVE when using an *octave_time_step* of any significant size.

RESET VALUE

After a reset, *octave_time_step* is set to .125 seconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_octave_mode, e1432_set_octave_avg_mode, e1432_set_octave_hold_mode, e1432_set_octave_start_freq, e1432_set_octave_stop_freq, e1432_set_octave_int_time, e1432_set_octave_time_const, e1432_octave_ctl, e1432_get_octave_blocksize, e1432_get_current_data

NAME

e1432_set_overlap − Set measurement overlap
e1432_get_overlap − Get current measurement overlap

SYNOPSIS

```
SHORTSIZ16 e1432_set_overlap(E1432ID hw, SHORTSIZ16 ID,
                             LONGSIZ32 overlap)
SHORTSIZ16 e1432_get_overlap(E1432ID hw, SHORTSIZ16 ID,
                             LONGSIZ32 *overlap)
```

DESCRIPTION

e1432_set_overlap sets the measurement overlap, of a single channel or group of channels *ID*, to the value given in *overlap*.

e1432_get_overlap returns the current value of the measurement overlap, of a single channel or group of channels *ID*, into a memory location pointed to by *overlap*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

overlap selects the number of data points that can overlap between one block and the next. The value may be positive or negative. A negative value for the *overlap* means that there is a separation between successive blocks. When the data size is set to **E1432_DATA_SIZE_16**, then the overlap will be rounded down to an even number. When the in order tracking mode the number of point of overlap are specified in terms of resampled time points.

When the data mode is set to **E1432_BLOCK_MODE**, then the *overlap* is not used. This is like the block mode of the E1431 and of the HP35652 and HP35655.

When the data mode is set to **E1432_CONTINUOUS_MODE**, then the *overlap* specifies the number of samples that overlap between successive data blocks. The maximum legal value is one less than the blocksize, while the minimum legal value is some large negative number. A negative overlap corresponds to a gap between one block and the next.

When the data mode is set to **E1432_DATA_MODE_OVERLAP_BLOCK**, the *overlap* specifies the point at which the module starts looking for a trigger for the next block of data. A positive value means that the trigger can happen before the end of the previous block. A negative value causes a gap after the previous block before a trigger can occur. Note that this just determines when the E1432 module reaches the **TRIGGER** state. Unlike in continuous mode, a trigger must still occur before a new data block is collected.

RESET VALUE

After a reset, the *overlap* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_blocksize, e1432_set_data_mode, e1432_get_overlap_limits

NAME

e1432_set_peak_decay_time − Set Peak detection decay time constant
e1432_get_peak_decay_time − Get current Peak detection decay time constant
e1432_set_rms_avg_time − Set RMS averaging decay time constant
e1432_get_rms_avg_time − Get current RMS averaging decay time constant
e1432_set_rms_decay_time − Set RMS decay time constant
e1432_get_rms_decay_time − Get current RMS decay time constant

SYNOPSIS

```
SHORTSIZ16 e1432_set_peak_decay_time(E1432ID hw, SHORTSIZ16 ID,
                                     FLOATSIZ32 peak_decay_time)
SHORTSIZ16 e1432_get_peak_decay_time(E1432ID hw, SHORTSIZ16 ID,
                                     FLOATSIZ32 *peak_decay_time)
SHORTSIZ16 e1432_set_rms_avg_time(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 rms_avg_time)
SHORTSIZ16 e1432_get_rms_avg_time(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *rms_avg_time)
SHORTSIZ16 e1432_set_rms_decay_time(E1432ID hw, SHORTSIZ16 ID,
                                    FLOATSIZ32 rms_decay_time)
SHORTSIZ16 e1432_get_rms_decay_time(E1432ID hw, SHORTSIZ16 ID,
                                    FLOATSIZ32 *rms_decay_time)
```

DESCRIPTION

e1432_set_peak_decay_time sets the exponential decay time constant, in seconds, for the Peak detection current value for a module, the time for a Peak value to decay to 37%, in voltage equivalent. This decay is only approximately exponential and breaks down under the conditions below.

e1432_set_rms_avg_time sets the exponential averaging time constant, in seconds, for the RMS current value for a module, the time for the internal RMS average value to decay to 37%, in power equivalent. The RMS power value is peak detected and *e1432_set_rms_decay_time* sets the exponential decay time constant, in seconds, power equivalent, for this peak detection and the peak detected value becomes the value returned by *e1432_get_current_value*. Both of these these decays are only approximately exponential and break down under the conditions below.

The Peak and RMS values returned by *e1432_get_current_value* are affected by these three parameters. The Peak and/or RMS values in the data trailer are also affected by these parameters when Peak detection is enabled using the **E1432_PEAK_MODE_FILT** parameter when calling the *e1432_set_peak_mode* function and/or RMS computations are enabled using the **E1432_RMS_MODE_FILT** parameter when calling the *e1432_set_rms_mode* function. The Peak and/or RMS values in the data trailer are uniformly weighted and not affected by these parameters when Peak detection is enabled using the **E1432_PEAK_MODE_BLOCK** parameter and/or RMS computations are enabled using the **E1432_RMS_MODE_BLOCK** parameter.

e1432_get_peak_decay_time, *e1432_get_rms_avg_time*, and *e1432_get_rms_decay_time* return the current value of their parameters to a floating point variable pointed by *peak_decay_time*, *rms_avg_time*, and *rms_decay_time* respectively.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel and is used to determine which module(s) the function call applies to.

peak_decay_time may be any value between **E1432_PEAK_DECAY_TIME_MIN** (0) seconds and **E1432_PEAK_DECAY_TIME_MAX** seconds inclusive.

Likewise, *rms_avg_time* may be any value between **E1432_RMS_AVG_TIME_MIN** (0) seconds and **E1432_RMS_AVG_TIME_MAX** seconds inclusive.

rms_decay_time may be any value between **E1432_RMS_DECAY_TIME_MIN** (0) seconds and **E1432_RMS_DECAY_TIME_MAX** seconds inclusive. The RMS peak decay time should be set to BE1432_RMS_DECAY_TIME_MIN (0) for the RMS current value to be a simple RMS exponential average, without peak detection.

With the correct choice of *rms_avg_time* and *rms_decay_time*, the "Slow", "Fast", and "Impulse" response characteristics of ANSI S1.4-1983 can be approximated.

When *peak_decay_time*, *rms_avg_time*, or *rms_decay_time* approach the neighborhood of 8-16 times the decimated sample interval (1 divided by the value returned by *e1432_get_span*), the exponential peak decay begins to break down and tends towards a uniform weighting. The length of the minimum, uniform weighting is 16 decimated samples if both Peak detection and RMS computation are turned on with *e1432_set_peak_mode* and *e1432_set_rms_mode*. If only one of these are turned on, the length of the minimum, uniform weighting is 8 decimated samples. Then decimated sample rate is doubled by each of: setting **E1432_DECIMATION_OVERSAMPLE_ON** with *e1432_set_decimation_oversample* and setting **E1432_MULTIPASS** with *e1432_set_decimation_output*.

Peak detection and RMS computations are currently available only on the E1433.

If any of these parameter are changed while a measurement is running, the change will not have any effect until the start of the next measurement.

RESET VALUE

After a reset, *peak_decay_time* is set to 1.5 seconds. *rms_avg_time* is set to 1.0 seconds and *rms_decay_time* to 0.0 seconds, approximating the "Slow" response characteristics of ANSI S1.4-1983.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_clock_freq, e1432_set_peak_mode, e1432_set_rms_mode, e1432_get_current_value.
e1432_set_weighting

NAME

e1432_set_peak_mode − Set Peak detection on/off
e1432_get_peak_mode − Get current state of Peak detection operation
e1432_set_rms_mode − Set RMS computation on/off
e1432_get_rms_mode − Get current state of RMS computation operation

SYNOPSIS

```
SHORTSIZ16 e1432_set_peak_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 peak_mode)
SHORTSIZ16 e1432_get_peak_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *peak_mode)
SHORTSIZ16 e1432_set_rms_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 rms_mode)
SHORTSIZ16 e1432_get_rms_mode(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *rms_mode)
```

DESCRIPTION

e1432_set_peak_mode and *e1432_set_rms_mode* turn on or off Peak detection and RMS computations, respectively, for the modules(s) selected.

e1432_get_peak_mode and *e1432_get_rms_mode* return the state of Peak detection and RMS computations, respectively, into a memory location pointed to by *peak_mode* and *rms_mode*, respectively.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. It is used to determine which module(s) in *hw* to set/query.

peak_mode must be one of **E1432_PEAK_MODE_OFF** to turn Peak detection off or either **E1432_PEAK_MODE_BLOCK** or **E1432_PEAK_MODE_FILT** to turn Peak detection on. The parameter, **E1432_RMS_MODE_ON** is an alias for **E1432_RMS_MODE_BLOCK** and included for backwards compatibility.

rms_mode must be one of **E1432_RMS_MODE_OFF** to turn RMS computations off or either **E1432_RMS_MODE_BLOCK** or **E1432_RMS_MODE_FILT** to turn RMS computations on. The parameter, **E1432_RMS_MODE_ON** is an alias for **E1432_RMS_MODE_BLOCK** and included for backwards compatibility.

Peak detection and RMS computations are only available at clock frequencies of 65,536 or less. Peak detection and RMS computations are performed at the full sample rate even when a lower span has been chosen. Peak and RMS values are available through the *e1432_get_current_value* function and in the trailer. The choice between **E1432_xxxx_MODE_BLOCK** or **E1432_xxxx_MODE_FILT** determines the computions used to place values in the trailer.

Because of conflicting demands on the module DSP resources, failures may occur for **E1432_PEAK_MODE_BLOCK** or **E1432_RMS_MODE_BLOCK** spans greater than 5 kHz with order tracking measurements, that is, both *decimation_output* = **E1432_MULTIPASS** and *decimation_oversample* = **E1432_DECIMATION_OVERSAMPLE_ON**. Similarly, **E1432_PEAK_MODE_FILT E1432_RMS_MODE_FILT** may fail for spans greater than 10 kHz with order tracking measurements. If both Peak and RMS computations are turned on, the failures may occur at lower spans.

Peak detection and RMS computations are currently available only on the E1433.

If either of these parameter are changed while a measurement is running, the change will not have any

effect until the start of the next measurement.

RESET VALUE

After a reset, *peak_mode* is set to **E1432_PEAK_MODE_OFF** and *rms_mode* is set to **E1432_RMS_MODE_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_clock_freq, e1432_set_peak_decay_time, e1432_set_rms_avg_time, e1432_set_rms_decay_time, e1432_get_current_value, e1432_set_weighting

NAME

e1432_set_ramp − Set debug data ramp
e1432_get_ramp − Get current value of debug data ramp

SYNOPSIS

```
SHORTSIZ16 e1432_set_ramp(E1432ID hw, SHORTSIZ16 ID,
                          SHORTSIZ16 ramp)
SHORTSIZ16 e1432_get_ramp(E1432ID hw, SHORTSIZ16 ID,
                          SHORTSIZ16 *ramp)
```

DESCRIPTION

e1432_set_ramp sets the debug data ramp, for a single channel or group of channels *ID*, to the value given in *ramp*.

e1432_get_ramp returns the current debug data ramp setting, of a single channel or group of channels *ID*, into a memory location pointed to by *ramp*.

This parameter is useful only for debugging, and may not even be useful for that. It's main purpose is to aid debugging of hardware and firmware problems at HP.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ramp determines whether the debug data ramp is turned on. The valid values are:

E1432_RAMP_OFF, to not have the ramp.

E1432_RAMP_ON, to turn on the ramp. When the ramp is on, all data from the input channels is thrown away, and replaced with artificial data which contains a smooth ramp. The ramp is duplicated for each input channel and starts over with each block read from the module. When the data size is set to **E1432_DATA_SIZE_16**, then the ramp starts at a raw data value of zero and increments by one lsb for each sample. When the data size is set to **E1432_DATA_SIZE_FLOAT32**, then the ramp starts at floating-point 1.0, and increments by a small value with each sample. When the data size is set to **E1432_DATA_SIZE_32** or **E1432_DATA_SIZE_32_SERV**, the ramp starts at zero and increments by 65536 lsbs for each sample. The low 16 bits of the data is all zero, except that the very bottom bits are set to the channel number.

When the ramp is on, it applies to all input time data transferred from the module, either to local bus or to to host computer. If eavesdropping is enabled then the ramp applies only to the data transferred to the host, and data going to disk may or may not have the ramp. We're not telling.

When the ramp is on and frequency data is enabled, the ramp may or may not replace the time data that is the input to the FFT. I think it will replace the time data if the host has enabled both time and frequency data to the host, and it will *not* if no time data is enabled to the host. Try it and let me know!

I have no idea what effect ramp will have when doing resampling. It probably will not be anything useful.

RESET VALUE

After a reset, *ramp* is set to **E1432_RAMP_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

NAME

e1432_set_ramp_rate − Set source ramp rate of E1432 channels
e1432_get_ramp_rate − Get source ramp rate of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_set_ramp_rate(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 ramp_rate)
SHORTSIZ16 e1432_get_ramp_rate(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 *ramp_rate)
```

DESCRIPTION

e1432_set_ramp_rate sets the source ramp-up and ramp-down rate, of a single channel or group of channels *ID*, to the value given in *ramp_rate*.

e1432_get_ramp_rate returns the current value of the source ramp rate, of a single channel or group of channels *ID*, into a memory location pointed to by *ramp_rate*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ramp_rate is the ramp-up rate. This is the total time for the ramp-up or ramp-down, in seconds.

For input channels, ramp rate is not generally used.

For source channels, the ramp rate is usually used to ensure that the source signal starts and stops smoothly. Output signals will start at zero and then ramp up at the specified ramp_rate. Thus, the first output point will be zero and succeeding values will follow the ramp_rate (which is applied at the clock_freq sampling rate). NOTE: If the ramp_rate is set to zero, the first value output (at clock_freq rate) will still be zero. This is especially noticeable at top span when using an Arb mode of operation since the first arb data point will appear to have been over-written with zero.

For tach channels, ramp rate is not generally used.

RESET VALUE

After a reset, the *ramp_rate* is set to 1 second.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_ramp_rate_limits

NAME

e1432_set_range − Set range of E1432
e1432_get_range − Get current range of E1432

SYNOPSIS

```
SHORTSIZ16 e1432_set_range(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 range)
SHORTSIZ16 e1432_get_range(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 *range)
```

DESCRIPTION

e1432_set_range sets the range, of a single channel or group of channels *ID*, to the value given in *range*.

e1432_get_range returns the current value of the range, of a single channel or group of channels *ID*, into a memory location pointed to by *range*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

range is the full scale range in volts. Signal inputs whose absolute value is larger than full scale will generate an ADC overflow error. (Actually, there is several dB of overhead before the ADC will overflow, to avoid spurious overflow indications.)

The actual range that is set will be the nearest legal range value that is greater than or equal to the value specified by the *range* parameter.

For input channels, the *range* is used only when the input mode is Voltage or ICP (see *e1432_set_input_mode*). When the input mode is Charge, the *range_charge* parameter is used instead (see *e1432_set_range_charge*). When the input mode is microphone mode, the *range_mike* parameter is used instead (see *e1432_set_range_mike*).

For source channels the range specifies an overall maximum signal level (typically on a range DAC reserved for that purpose), and can't be changed instantaneously during output. To change the signal amplitude during output, use e1432_set_amp_scale, which can scale the output level by an (almost) arbitrary scale factor.

For tach channels, neither range nor amplitude are used.

This parameter may also be set with *e1432_set_analog_input*.

RESET VALUE

After a reset, input channels have the *range* set to 10 volts. Source channels have the *range* set to the minimum legal source range (and the source is also inactive, so no signal is produced).

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_analog_input, e1432_set_amp_scale, e1432_set_input_mode, e1432_get_range_limits, e1432_set_range_charge, e1432_set_range_mike, e1432_bob

NAME

e1432_set_range_charge − Set charge-amp range of E1432
e1432_get_range_charge − Get current charge-amp range of E1432

SYNOPSIS

```
SHORTSIZ16 e1432_set_range_charge(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 range_charge)
SHORTSIZ16 e1432_get_range_charge(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *range_charge)
```

DESCRIPTION

e1432_set_range_charge sets the charge-amp range, of a single channel or group of channels *ID*, to the value given in *range_charge*.

e1432_get_range_charge returns the current value of the charge-amp range, of a single channel or group of channels *ID*, into a memory location pointed to by *range_charge*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

range_charge is the full scale charge-amp range in picoCoulombs. Signal inputs whose absolute value is larger than full scale will generate an ADC overflow error. (Actually, there is several dB of overhead before the ADC will overflow, to avoid spurious overflow indications.)

The actual charge-amp range that is set will be the nearest legal range value that is greater than or equal to the value specified by the *range_charge* parameter.

The *range_charge* parameter applies only to input channels, and is used only when the input mode is **E1432_INPUT_MODE_CHARGE** (see *e1432_set_input_mode*). This is only possible when a Charge Break-out Box is connected to the input.

RESET VALUE

After a reset, the charge-amp range is set to 50000 pC.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_input_mode, e1432_set_range, e1432_get_range_charge_limits, e1432_set_range_mike, e1432_bob

NAME

e1432_set_range_mike − Set microphone range of E1432
e1432_get_range_mike − Get current microphone range of E1432

SYNOPSIS

```
SHORTSIZ16 e1432_set_range_mike(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 range_mike)
SHORTSIZ16 e1432_get_range_mike(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 *range_mike)
```

DESCRIPTION

e1432_set_range_mike sets the microphone range, of a single channel or group of channels *ID*, to the value given in *range_mike*.

e1432_get_range_mike returns the current value of the microphone range, of a single channel or group of channels *ID*, into a memory location pointed to by *range_mike*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

range_mike is the full scale microphone range in volts. Signal inputs whose absolute value is larger than full scale will generate an ADC overflow error. (Actually, there is several dB of overhead before the ADC will overflow, to avoid spurious overflow indications.)

The actual microphone range that is set will be the nearest legal range_mike value that is greater than or equal to the value specified by the *range_mike* parameter.

The *range_mike* parameter applies only to input channels, and is used only when the input mode is **E1432_INPUT_MODE_MIC** or **E1432_INPUT_MODE_MIC_200V** (see *e1432_set_input_mode*). This is only possible when a Microphone Break-out Box is connected to the input.

RESET VALUE

After a reset, the charge-amp range is set to 10 volts.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_input_mode, e1432_set_range, e1432_set_range_charge, e1432_get_range_mike_limits, e1432_bob

NAME

e1432_set_pre_arm_mode − Set pre-arm state
e1432_get_pre_arm_mode − Get current pre-arm state

SYNOPSIS

```
SHORTSIZ16 e1432_set_pre_arm_mode(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 armState)
SHORTSIZ16 e1432_get_pre_arm_mode(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *armState)
```

DESCRIPTION

e1432_set_pre_arm_mode sets the pre-arm mode, of a single channel or group of channels *ID*, to the value given in *armState*.

e1432_get_pre_arm_mode returns the current value of the pre-arm mode, of a single channel or group of channels *ID*, into a memory location pointed to by *armState*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

armState determines which arm event will allow the module to advance from the **PRE_ARM** state into the **IDLE** state.

E1432_MANUAL_ARM sets the module to wait for a pre-arm event to occur either from the system (SYNC line), or from the *e1432_pre_arm_measure* command, in order to perform the transition.

E1432_AUTO_ARM sets the module to perform the transition as soon as it enters the **PRE_ARM** state.

There are three RPM pre-arming modes:

E1432_ARM_RPM_RUNUP sets the module to do a pre-arm as soon as the RPM from the tachometer board rises above the level set by the *e1432_set_pre_arm_rpm* function.

E1432_ARM_RPM_RUNDOWN sets the module to do a pre-arm as soon as the RPM from the tach board falls below the value set by the *e1432_set_pre_arm_rpm* function.

E1432_ARM_RPM_DELTA sets the module to do a pre-arm as soon as the RPM changes by the amount set by the *e1432_set_pre_arm_rpm* functions.

There is also an external trigger pre-arm mode:

E1432_ARM_EXTERNAL specifies that the module should wait for an external trigger edge before moving on to the **IDLE** state.

If there is an option AYF tachometer board present, then the second channel on this board can be used as the external trigger. Note that this channel must be active (set by *e1432_set_active*) and enabled to assert trigger (by using *e1432_set_trigger_channel* with the **E1432_CHANNEL_PRE_ARM** parameter), in order for it to detect an external trigger edge in the pre-arm state.

If there is no option 1D4 source board present, then there is also an external trigger connector, **ExTrig** on the module's front panel which can be used for external triggering. This external trigger must be enabled with *e1432_set_trigger_ext* to allow it to detect an external trigger edge. (This external trigger input is a TTL input, so there is no way to program the trigger level.)

RESET VALUE

After a reset, *armState* is set to **E1432_AUTO_ARM**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_pre_arm_measure, e1432_set_arm_mode, e1432_set_pre_arm_rpm, e1432_set_trigger_ext, e1432_set_trigger_channel

NAME

e1432_set_pre_arm_rpm − Set pre-arm threshold RPM
e1432_get_pre_arm_rpm − Get pre-arm threshold RPM

SYNOPSIS

```
SHORTSIZ16 e1432_set_pre_arm_rpm(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 rpm)
SHORTSIZ16 e1432_get_pre_arm_rpm(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ32 *rpm)
```

DESCRIPTION

e1432_set_pre_arm_rpm sets the threshold value used by the RPM pre-arming modes of the *e1432_set_pre_arm_mode* function.

e1432_get_pre_arm_rpm returns the threshold pre-arm RPM into the variable pointed to by *rpm*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

rpm specifies the threshold RPM.

RESET VALUE

After a reset, *rpm* is set to 0.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_pre_arm_rpm_limits, e1432_set_pre_arm_mode

NAME

e1432_set_rpm_high − Set upper limit of arming RPM
e1432_get_rpm_high − Get upper limit of arming RPM

SYNOPSIS

```
SHORTSIZ16 e1432_set_rpm_high(E1432ID hw, SHORTSIZ16 ID,
                              FLOATSIZ32 rpm)
SHORTSIZ16 e1432_get_rpm_high(E1432ID hw, SHORTSIZ16 ID,
                              FLOATSIZ32 *rpm)
```

DESCRIPTION

e1432_set_rpm_high sets the upper limit of arming RPMs used by the RPM arming modes of the *e1432_set_arm_mode* function. The value entered is a floating point number, but it is truncated to a integer internally; so the value returned by *e1432_get_rpm_high* will always be an integer represented as a floating point number.

e1432_get_rpm_high returns the upper limit RPM into the variable pointed to by *rpm*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

rpm specifies the upper limit RPM.

NOTE: When in the order tracking measurement, the following relationship between these parameters must be observed:

rpm_high <= 60 * span / max_order

or a WARN1432_LOST_RPM_TOO_HIGH warning will be issued and an arming point lost.

Arming begins at this threshold on a rundown measurment or ends at this threshold on a runup measurement.

RESET VALUE

After a reset, *rpm* is set to 6000.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_rpm_high_limits, e1432_set_arm_mode, e1432_set_rpm_low, e1432_set_rpm_interval

NAME

e1432_set_rpm_interval − Set step interval of arming RPM
e1432_get_rpm_interval − Get step interval of arming RPM

SYNOPSIS

```
SHORTSIZ16 e1432_set_rpm_interval(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 rpm)
SHORTSIZ16 e1432_get_rpm_interval(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *rpm)
```

DESCRIPTION

e1432_set_rpm_interval sets the stepping interval of arming RPMs used by the RPM arming modes of the *e1432_set_arm_mode* function. The value entered is a floating point number, but it is truncated to a integer internally; so the value returned by *e1432_get_rpm_interval* will always be an integer represented as a floating point number.

e1432_get_rpm_interval returns the stepping interval RPM into the variable pointed to by *rpm*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

rpm specifies the stepping interval RPM.

Arming reoccurs after the RPM changes by this stepping interval from the previous RPM arming point.

RESET VALUE

After a reset, *rpm* is set to 25.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_rpm_interval_limits, e1432_set_arm_mode, e1432_set_rpm_low, e1432_set_rpm_high

NAME

e1432_set_rpm_low − Set lower limit of arming RPM
e1432_get_rpm_low − Get lower limit of arming RPM

SYNOPSIS

```
SHORTSIZ16 e1432_set_rpm_low(E1432ID hw, SHORTSIZ16 ID,
                             FLOATSIZ32 rpm)
SHORTSIZ16 e1432_get_rpm_low(E1432ID hw, SHORTSIZ16 ID,
                             FLOATSIZ32 *rpm)
```

DESCRIPTION

e1432_set_rpm_low sets the lower limit of arming RPMs used by the RPM arming modes of the *e1432_set_arm_mode* function. The value entered is a floating point number, but it is truncated to a integer internally; so the value returned by *e1432_get_rpm_low* will always be an integer represented as a floating point number.

e1432_get_rpm_low returns the lower limit RPM into the variable pointed to by *rpm*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

rpm specifies the lower limit RPM.

NOTE: When in the order tracking measurement, the following relationship between these parameters must be observed:

rpm_low >= (60 * span) / (64 * max_order)

or a WARN1432_LOST_RPM_TOO_LOW warning will be issued and an arming point lost.

Arming begins at this threshold on a runup measurment or ends at this threshold on a rundown measurement.

RESET VALUE

After a reset, *rpm* is set to 600.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_rpm_low_limits, e1432_set_arm_mode, e1432_set_rpm_high, e1432_set_rpm_interval

NAME

e1432_set_rpm_smoothing − Set tach time smoothing factor
e1432_get_rpm_smoothing − Get tach time smoothing factor

SYNOPSIS

```
SHORTSIZ16 e1432_set_rpm_smoothing(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 smoothing)
SHORTSIZ16 e1432_get_rpm_smoothing(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 *smoothing)
```

DESCRIPTION

e1432_set_rpm_smoothing sets the tach time smoothing factor used for calculating RPM values in the RPM arming modes.

e1432_get_rpm_smoothing returns the tach time smoothing factor into the variable pointed to by *smoothing*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

smoothing specifies the smoothing factor. It's legal values range from 0.0 to 1.0.

When the smoothing factor is non-zero, tach times are calculated using a weighted "average" of previous smoothed tach times and the new raw tach time obtained from the AYF tachometer option. This smoothing operation takes the "jitter" out of tach times obtained from real world tachometers. The smoothed tach time is calculated in the following fashion:

Traw = current raw tach time from tach board
Tn-1 = previously calculated tach time
Tn-2 = calculated tach time before that
Tn = current tach time to be calculated

Tn = Traw * (1.0 - smoothing) + (2Tn-1 - Tn-2) * smoothing

RESET VALUE

After a reset, *smoothing* is set to 0.0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_rpm_smoothing_limits

NAME

e1432_set_sample_mode − Set method of resampling time data in order tracking
e1432_get_sample_mode − Get method of resampling time data in order tracking

SYNOPSIS

```
SHORTSIZ16 e1432_set_sample_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_sample_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_sample_mode sets the method of resampling time data while in order tracking to either the standard computed resampling or resampling at each tach pulse. The latter method is useful when using tachometers/encoders with multiple pulses per revolution. This method resamples the time data at every tach pulse as opposed to the normal resampling method which computes the resampling points based on the delta order parameter and interpolated shaft positions.

The resample mode is a "global" parameter. It applies to an entire E1432/33 module (or group of modules if in a master/slave configuration) rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode selects the resample mode. The two modes are **E1432_RESAMP_NORMAL** for the normal mode and **E1432_RESAMP_AT_TACH** for the resample at tach times mode.

Resampling of time data only make sense in the order tracking mode of the E1432, which means that the *e1432_set_calc_data* function must be set to **E1432_DATA_RESAMP_TIME** or **E1432_DATA_ORDER**. If it is set to **E1432_DATA_ORDER**, the blocksize must be a power of two. The number of pulses per revolution as specified by the *e1432_set_tach_ppr* function is used to get the number of sampling points per revolution in the resample at tach mode and the delta order parameter is ignored. The value of smoothing set by the *e1432_set_rpm_smoothing* function is ignored when in the resample at tach mode.

In the resample at tach mode, the number of pulses per revolution must be set by the *e1432_set_tach_ppr* function. The *blocksize* parameter is set by *e1432_set_blocksize* to a value equal to the number of pulses per revolution times the number of revolutions of data wanted in a block. This number does not have to be a power of two **unless** the internal FFT calculation is enabled with the *e1432_set_calc_data* function.

Except for a special external trigger case mentioned below, the resample at tach mode ignores all arming and triggering modes. The module will continuously resample time data at the tach pulse times whenever the RPM is within the high and low values set by *e1432_set_rpm_high* and *e1432_set_rpm_low* respectively. If this causes the module to fall behind enough where new data coming into the data FIFO overwrites old data, the module will skip over some of the older data and then continue resampling in order to keep up. When it does this, it will issue a warning, **WARN1432_LOST_DATA_SHIFTED_OUT_FIFO**.

There is a special configuration of the two tach channels such that one channel can be used to start the resampling process, whose resample points come from the other channel. This is the external trigger option which allows each occurance of the resample at tach process to be initiated by a "top dead center" signal once per revolution, for instance. In this configuration, the first tach channel must be configured as the

resampling channel and the second as the trigger channel. An example of the code to do this is found in the demo programs by uncommenting the **EXT_TRIG** define statement.

The demo programs, at_tach.c and at_tach2.c, show how to set up the module to do resample at tach.

RESET VALUE

After a reset, *mode* is set to **E1432_SAMP_RESAMP**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calc_data, e1432_set_tach_ppr, e1432_get_meas_warning.

NAME

e1432_set_sine_freq − Set source sine frequency of E1432 channels
e1432_get_sine_freq − Get source sine frequency of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_set_sine_freq(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 sine_freq)
SHORTSIZ16 e1432_get_sine_freq(E1432ID hw, SHORTSIZ16 ID,
                               FLOATSIZ32 *sine_freq)
```

DESCRIPTION

e1432_set_sine_freq sets the source sine frequency, of a single channel or group of channels *ID*, to the value given in *sine_freq*. When the source is in **E1432_SOURCE_MODE_SINE** mode, this is the frequency that is generated.

e1432_get_sine_freq returns the current value of the source sine frequency, of a single channel or group of channels *ID*, into a memory location pointed to by *sine_freq*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

sine_freq is the sine frequency in Hertz.

For input channels, sine frequency is not used.

For source channels, the sine frequency is used when in **E1432_SOURCE_MODE_SINE** mode.

For tach channels, sine frequency is not used.

RESET VALUE

After a reset, the *sine_freq* is set to 1 kHz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_mode, e1432_set_sine_phase, e1432_get_sine_freq_limits

NAME

e1432_set_sine_phase − Set source sine start phase of E1432 channels
e1432_get_sine_phase − Get source sine start phase of E1432 channels

SYNOPSIS

```
SHORTSIZ16 e1432_set_sine_phase(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 sine_phase)
SHORTSIZ16 e1432_get_sine_phase(E1432ID hw, SHORTSIZ16 ID,
                                FLOATSIZ32 *sine_phase)
```

DESCRIPTION

e1432_set_sine_phase sets the source sine start phase, of a single channel or group of channels *ID*, to the value given in *sine_phase*. When the source is in sinewave generation mode, this is the starting phase of the sinewave that is generated.

e1432_get_sine_phase returns the value of the source sine phase, of a single channel or group of channels *ID*, into a memory location pointed to by *sine_phase*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

sine_phase is the starting sine phase in degrees. It must be between -360 and 360 degress. The sinusoid signal is expressed as sin(sine_freq*t + sine_phase). If sine_phase is changed during a measurement, the phase is incremented by the difference between sine_phase and the previous value of sine_phase. Note: sine_phase cannot be changed during a measurement when using the burst sine mode (**E1432_SOURCE_MODE_BSINE**) of operation.

For input channels, sine phase is not used.

For source channels, the sine phase is used when in sinewave mode.

For tach channels, sine phase is not used.

RESET VALUE

After a reset, the *sine_phase* is set to zero.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_sine_freq, e1432_get_sine_phase_limits

NAME

e1432_set_source_blocksize − Set source blocksize
e1432_get_source_blocksize − Get source blocksize

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_blocksize(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 blocksize)
SHORTSIZ16 e1432_get_source_blocksize(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 *blocksize)
```

DESCRIPTION

e1432_set_source_blocksize sets the source blocksize, of a single channel or group of channels *ID*, to the number of samples given in *blocksize*. All channels with in a scource SCA have the same blocksize.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

blocksize selects the number of sample points in a block. The default value is zero which means to use the measurement blocksize.

RESET VALUE

After a reset, the source *blocksize* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_blocksize, e1432_get_source_blocksize_limits

NAME

e1432_set_source_centerfreq - Set source zoom center frequency
e1432_get_source_centerfreq - Get source zoom center frequency

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_centerfreq(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 freq)
SHORTSIZ16 e1432_get_source_centerfreq(E1432ID hw, SHORTSIZ16 ID,
                                       FLOATSIZ32 *freq)
```

DESCRIPTION

e1432_set_source_centerfreq sets the center frequency for source "zoom". See discussion of source zooming under the *e1432_set_source_mode* function.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

freq is the source center frequency, in Hertz. The default value is zero which means to use the measurement center frequency that was specified by *e1432_set_center_freq*. Any non-zero value overrides the center frequency specified by *e1432_set_center_freq*, so that source channels may have a different center frequency than input channels.

RESET VALUE

After a reset, the source *freq* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_mode, e1432_get_source_centerfreq_limits, e1432_set_center_freq

NAME

e1432_set_source_cola − Enable or disable source constant-level output
e1432_get_source_cola − Get current state of source constant-level output

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_cola(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 source_cola)
SHORTSIZ16 e1432_get_source_cola(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *source_cola)
```

DESCRIPTION

e1432_set_source_cola enables or disables the source constant-level output, of a single channel or group of channels *ID*, to the value given in *source_cola*.

e1432_get_source_cola returns the current status of the source constant-level output, of a single channel or group of channels *ID*, into a memory location pointed to by *source_cola*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

source_cola specifies whether the constant-level output is enabled or not. The valid values are **E1432_SOURCE_COLA_OFF** to disable the constant-level output, **E1432_SOURCE_COLA_ON** to enable the constant-level output, and **E1432_SOURCE_COLA_DRPEPPER** for a refreshing new taste in colas.

Obviously, this function is not useful when talking to input or tach channels. Only source channels have a constant-level output.

For the Option 1D4 single-channel source, the "source_cola" output is shared with the "source_sum" input. Only one of these two may be enabled at any one time. For prototype Option 1D4 sources only, one of the two must be enabled at all times, and the default is for the constant-level output to be enabled.

RESET VALUE

After a reset, *cola* is set to **E1432_SOURCE_COLA_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_sum

NAME

e1432_set_source_mode − Set source mode
e1432_get_source_mode − Get current state of source mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_source_mode(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_source_mode sets the source mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_source_mode returns the current value of the source mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode determines the source mode. The valid values are **E1432_SOURCE_MODE_SINE** for sine mode, **E1432_SOURCE_MODE_BSINE** for burst sine mode, **E1432_SOURCE_MODE_RAND** for random noise, **E1432_SOURCE_MODE_BRAND** for burst random noise, **E1432_SOURCE_MODE_RANDZ** for zoomed random noise, **E1432_SOURCE_MODE_BRANDZ** for zoomed burst random noise, and **E1432_SOURCE_MODE_ARB** for an arbitrary signal. **E1432_SOURCE_MODE_BARB** for a burst arbitrary signal.

For **E1432_SOURCE_MODE_RANDZ** or **E1432_SOURCE_MODE_BRANDZ** the noise is digitally mixed (multiplied) by a complex sine wave. The frequency of the sine wave is set by *e1432_set_source_centerfreq* or *e1432_set_center_freq*. This results in a source signal with a spectrum that is centered around the frequency set by this function with a span set by *e1432_set_span* function or *e1432_set_source_span*.

There are limitations on the spans available in the zoomed modes, since these modes require more than twice as many digital filter operations to create. The top frequency is limited to one fifth of the normal top frequency for the module; i.e. if the normal frequency is 20Khz, the top frequency in zoom is limited to 4KHz. The center frequency is constrained to be below this top frequency limit.

For **E1432_SOURCE_MODE_ARB** and **E1432_SOURCE_MODE_BARB**, the host program must provide the data to use for the arbitrary signal, by using the *e1432_write_srcbuffer_data* function. The arb data must be reloaded into the buffers after setting which source channels are active.

For **E1432_SOURCE_MODE_BARB**, the source buffer mode must be set to **E1432_SRCBUFFER_PERIODIC_A** or **E1432_SRCBUFFER_PERIODIC_AB**.

For all of the burst modes, multiple source channels will burst at exactly the same time regardless of what form of triggering is used by the system.

For non-burst modes, an auto-triggered measurement will start the source at the system sync, so that it is already on by the time the inputs start taking data (though ramp-up may not yet be complete, depending on the ramp rate set by *e1432_set_ramp_rate*). In contrast, a non-auto-triggered measurement will start the source at the first trigger.

For random signal modes, **E1432_SOURCE_MODE_RAND**, **E1432_SOURCE_MODE_BRAND**, **E1432_SOURCE_MODE_RANDZ**, and **E1432_SOURCE_MODE_BRANDZ**, the nearly normal amplitude distribution results in occassional high peak amplitudes. To avoid clipping, etc., the actual random noise level is much less than the source level set using *e1432_set_amp_scale* and *e1432_set_range*.

Typical source levels for non burst random signals when *amp_scale* set to 1 are:

source mode	Vrms/Vrange	Vpeak/Vrms
base-band random top span	0.22	3.2
base-band random other spans	0.2	4.4
zoomed random	0.11	4.4

Burst random signal levels will be similar, during the burst on time.

There is no "off" source mode. To turn a source channel off, make it inactive using e1432_set_active.

When the source is off, it is normally low impedance to ground. To make the source high impedance, use the *e1432_set_source_output* function, with the **E1432_SOURCE_OUTPUT_OPEN** parameter.

There are currently some realtime restrictions that vary depending on the source mode used, whether parameters are changed during a measurement, and which source outputs are being used. Currently the conditions that will produce realtime source errors have been locked out by the following logic and will produce **ERR1432_SRC_REALTIME_RESTRICTION** errors if violated:

Lockouts for clock_freq:

Daughter board:
if ((ARB_CONT or ARB_PER_A or ARB_ONESHOT) and decimation) and
(clock_freq > 64000))

For SCA with 2 channels active:
if ((RANDZ or BRANDZ or ARB_CONT or ARB_PER_A or ARB_ONESHOT) and
(clock_freq > 51200))

Lockouts for changing parameters during a measurement:

Daughter Board:
if ((BSINE and clock_freq > 64000) or
((BRAND or RANDZ or BRANDZ or ARB (with decimation)) and
(clock_freq > 51200)))

SCA with 2 channels active:
if ((BSINE and clock_freq > 64000) or
((RAND(with decimation) or ARB) and clock_freq > 51200) or
((BRAND or RANSZ or BRANDZ) and (clock_freq > 50000)))

SCA with 1 channel active:
if ((BSINE or BRAND or BRANDZ) and (freq >64000))

Obviously, this function is not useful when talking to input or tach channels. Only source channels have a source mode.

RESET VALUE

After a reset, *mode* is set to **E1432_SOURCE_MODE_SINE**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_amp_scale, e1432_set_range, e1432_set_input_mode, e1432_set_active, e1432_set_source_output, e1432_set_source_span, e1432_set_source_centerfreq, e1432_set_srcbuffer_mode

NAME

e1432_set_source_output − Set source output type
e1432_get_source_output − Get current state of source output

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_output(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 output)
SHORTSIZ16 e1432_get_source_output(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 *output)
```

DESCRIPTION

e1432_set_source_output sets the source output, of a single channel or group of channels *ID*, to the value given in *output*.

e1432_get_source_output returns the current value of the source output, of a single channel or group of channels *ID*, into a memory location pointed to by *output*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

output determines the source output. The valid values are:

E1432_SOURCE_OUTPUT_NORMAL for normal source output, as defined by the source mode and other source parameters.

E1432_SOURCE_OUTPUT_GROUNDED to have the source output connector remain grounded, while internally connecting the source DAC to the CALOUT line in the module. Because the signal does not pass through the range attenuator or analog filter circuits, the signal amplitude is controlled only by **e1432_set_amp_scale**. Starting the source will leave the source output connector grounded, and drive the source signal (as defined by the source mode) onto the CALOUT line.

E1432_SOURCE_OUTPUT_OPEN to have the source remain open circuited even when the source is started. The impedance on the output is actually only about 1K ohm, because the power-fail decay circuit is still connected to the output.

E1432_SOURCE_OUTPUT_CAL to connect the source output connector to the module's internal CALOUT line. This allows the module's CALOUT line to be driven by an external signal applied at the source output connector.

E1432_SOURCE_OUTPUT_MULTI to connect the source output connector to the module's internal CALOUT line, and also connect the source DAC to the CALOUT line. This is a combination of the GROUNDED and CAL values above, and is useful for multi-mainframe calibration.

Obviously, this function is not useful when talking to input or tach channels. Only source channels have a source output.

RESET VALUE

After a reset, *output* is set to **E1432_SOURCE_OUTPUT_NORMAL**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_mode, e1432_set_calin, e1432_set_active, e1432_set_amp_scale

NAME

e1432_set_source_seed − Set seed for source random modes
e1432_get_source_seed − Get seed for source random modes

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_seed(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 source_seed)
SHORTSIZ16 e1432_get_source_seed(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *source_seed)
```

DESCRIPTION

e1432_set_source_seed sets the seed used by the source random modes, of a single channel or group of channels *ID*, to the value given in *source_seed*. This value is used when the source is in **E1432_SOURCE_MODE_RAND** or **E1432_SOURCE_MODE_BRAND** modes.

e1432_get_source_seed returns the current value of the source randomization seed, of a single channel or group of channels *ID*, into a memory location pointed to by *source_seed*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

source_seed is the seed value. Only the bottom eight bits of this seed value are currently used. However, each of the 256 possible seeds results in uncorrelated signals (i.e. a source channel with a seed of 0 and a source channel with a seed of 1 will produce random noise that is uncorrelated with each other).

For input channels, the source seed is not used.

For source channels, the source seed is used when in **E1432_SOURCE_MODE_RAND** or **E1432_SOURCE_MODE_BRAND** modes.

For tach channels, the source seed is not used.

Note: The reset value for this parameter is the same for all source channels. This means that unless the application sets the source seeds, all source channels will produces the same random data. Furthermore, unless the seed is changed between successive measurements, each source will produce the same data stream from measurement to measurement.

RESET VALUE

After a reset, the *source_seed* is set to 3.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_mode, e1432_get_source_seed_limits

NAME

e1432_set_source_span − Set source span
e1432_get_source_span − Get source span

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_span(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 span)
SHORTSIZ16 e1432_get_source_span(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 *span)
```

DESCRIPTION

e1432_set_source_span sets the source bandwidth. This specifies the maximum frequency at which the output signal will correctly track the signal that the source is attempting to generate.

The valid values for *span* depend of the current clock frequency, which is set by *e1432_set_clock_freq*. The clock frequency should be set before setting the span. Normally, the maximum valid span is max_span = clock_freq/2.56. Valid spans are max_span divided by powers of two, and max_span divided by five and by powers of two. The ratio between the span and the max_span is called the decimation factor.

The maximum number of decimate-by-two passes allowed by the source is 16, so the maximum decimation factor is 5*2^16.

The effective sample rate, which is the rate at which data is received from an input or used by a source, is normally equal to 2.56 times the span.

In certain cases, some of the frequencies above the maximum span may still contain valid alias-protected data. However, this function ignores the extra bandwidth, and pretends that the maximum span is always 1/2.56 times the effective sample rate.

Filter settling is not performed on source channels during the measurement initialization process when the decimation factor is greater than 5*2^9.

e1432_get_source_span returns the current span in Hertz. All source channels in the same SCA have the same span.

The default setting is 0 Hz. This is a special case, in which the source is actually set to the measurement span.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

span is the desired span, in Hz. It is rounded up to the next larger valid span.

This function is used in order to set a source channel to a different span than the input channels in a measurement. However, even if the module is an E1434 and has no input channels, it is a good idea to also use *e1432_set_span* to tell the module what span other input modules are using. That way, if the source module is doing source triggering, it knows how much decimation the inputs are doing and can ensure that the source trigger lines up with the decimated input data.

RESET VALUE

After a reset, each source channel is set to a span of 0 Hz.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_clock_freq, e1432_set_span, e1432_get_source_span_limits

NAME

e1432_set_source_sum − Enable or disable source sum input
e1432_get_source_sum − Get current state of source sum input

SYNOPSIS

```
SHORTSIZ16 e1432_set_source_sum(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 source_sum)
SHORTSIZ16 e1432_get_source_sum(E1432ID hw, SHORTSIZ16 ID,
                                SHORTSIZ16 *source_sum)
```

DESCRIPTION

e1432_set_source_sum enables or disables the source sum input, of a single channel or group of channels *ID*, to the value given in *source_sum*.

e1432_get_source_sum returns the current status of the source sum input, of a single channel or group of channels *ID*, into a memory location pointed to by *source_sum*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

source_sum specifies whether the sum input is enabled or not. The valid values are **E1432_SOURCE_SUM_OFF** to disable the sum input and **E1432_SOURCE_SUM_ON** to enable the sum input. The signal on the sum input is internally added to the output that the source would otherwise be produced.

Obviously, this function is not useful when talking to input or tach channels. Only source channels have a constant-level output. This function only works on the Option 1D4 single-channel source.

For the Option 1D4 single-channel source, the "source_sum" input is shared with the "source_cola" output. Only one of these two may be enabled at any one time. For prototype Option 1D4 sources only, one of the two must be enabled at all times, and the default is for the constant-level output to be enabled and the sum input to be disabled.

RESET VALUE

After a reset, *sum* is set to **E1432_SOURCE_SUM_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_cola

NAME

e1432_set_span − Set measurement span
e1432_get_span − Get measurement span

SYNOPSIS

```
SHORTSIZ16 e1432_set_span(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 span)
SHORTSIZ16 e1432_get_span(E1432ID hw, SHORTSIZ16 ID, FLOATSIZ32 *span)
```

DESCRIPTION

e1432_set_span sets the measurement bandwidth. For an input channel, the span specifies the maximum frequency at which valid alias-protected data will be received (frequencies above that are filtered out). For a source channel, the span specifies the maximum frequency at which the output signal will correctly track the signal that the source is attempting to generate.

The valid values for *span* depend of the current clock frequency, which is set by *e1432_set_clock_freq*. The clock frequency should be set before setting the span. Normally, the maximum valid span is max_span = clock_freq/2.56. Valid spans are max_span divided by powers of two, and max_span divided by five and by powers of two. The ratio between the span and the max_span is called the decimation factor.

NOTE: When in order tracking the max_span = clock_freq / 5.12, since the order tracking algorithm uses data oversampled by two. Also in the order tracking mode, twice as much data is passed from the input SCAs to the main processor. The outputs of all of the decimation filters in the input SCAs are output for the order tracking algorithm to use. This increased data load forces a restriction on the number of channels that can be used at the top order tracking spans ... only two SCA may be active on some of the top spans. In the E1432 this means only 8 channels residing on two SCAs may be active at the top span (10Khz at clock frequency of 51200). In the E1433 only 4 channels residing on two SCAs may be active above at span of 10Khz. Any more active channels will cause the error, **ERR1432_ILLEGAL_ORDER_CHANNEL_COMBO**.

For the E1432 51.2 kHz input SCA, the maximum number of decimate-by-two passes allowed is nine, so the maximum decimation factor is 5*2^9 and the minimum valid span is clock_freq/(2.56*5*2^9). If the clock frequency is larger than 51.2 kHz, then the E1432 input channel is unable to do a decimation factor of one; the minimum decimation factor is 2 and the maximum valid span is clock_freq/5.12.

For the E1433 196 kHz input SCA, the maximum number of decimate-by-two passes allowed is 12, so the maximum decimation factor is 5*2^12. Due to limits in the E1433 DSP processors, when the clock frequency is set higher than 128000 Hz, the E1432 input SCA is unable to do *any* decimation, so in this case the *only* valid span is clock_freq/2.56. Attempts to use decimation when the clock frequency is above 128000 Hz will result in an error when the measurement starts.

For the E1434 source, and the option 1D4 source board, the maximum number of decimate-by-two passes allowed is 16, so the maximum decimation factor is 5*2^16.

The effective sample rate, which is the rate at which data is received from an input or used by a source, is normally equal to 2.56 times the span. If the data is oversampled (see *e1432_set_decimation_oversample*), then the effective sample rate is 5.12 times the span.

In certain cases, some of the frequencies above the maximum span may still contain valid alias-protected data. This will be the case if the digital filters in an SCA have a sharper cutoff than the usual 1/2.56. This is the case with both the E1432 51.2 kHz input SCA and the E1433 196 kHz input SCA when at top span. The E1432 and E1433 top span filter cutoff is clock_freq/2.226 (23 kHz when the clock frequency is 51.2 kHz, 88.3 kHz when the clock frequency is 196.608 kHz). However, this e1432_set_span function ignores the extra bandwidth, and pretends that the maximum span is always 1/2.56 times the effective sample rate.

NOTE: There are further restrictions on allowable spans when in zoom mode. These are documented in the *e1432_set_zoom* function.

e1432_get_span returns the current span in Hertz. All channels of a module have the same span.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

span is the desired span, in Hz. It is rounded up to the next larger valid span.

The algorithm to pick what "valid" span to round to is different depending on whether a measurement is currently running or not. If no measurement is running, the span is chosen from all valid spans.

If a measurement is running, the span is required to use the same "divide by five" setting that the original span used when the measurement was started. For example, if the clock frequency is 51200 Hz, it is valid to switch from 5000 Hz to 10000 Hz spans when a measurement is running, but it is not valid to switch from 4000 Hz to 10000 Hz. The 4000 Hz span uses divide by five, while the 10000 Hz span does not.

Because the algorithm for rounding the span depends on whether a measurement is running, and is somewhat complicated anyway, it is generally a good idea to call *e1432_get_span* after setting the span. This will ensure that the host application knows the actual span that the module is using.

If the span is changed while a measurement is running, the module will flush any data from before the span change. All data read from the module after the span change will be data at the new span. When using trailer data (see *e1432_set_append_status*), the "gap" field in the trailer will not be exactly correct for the first block after the span change.

RESET VALUE

After a reset, each module is set to the maximum legal span.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_clock_freq, e1432_get_span_limits, e1432_set_center_freq

NAME

e1432_set_srcbuffer_mode − Set the source ARB buffer mode
e1432_get_srcbuffer_mode − Get the current source ARB buffer mode
e1432_set_srcbuffer_size − Set the source ARB buffer size
e1432_get_srcbuffer_size − Get the current source ARB buffer size
e1432_set_srcbuffer_init − Initialize the source ARB buffers
e1432_get_srcbuffer_init − Get the source ARB buffer initialization value

SYNOPSIS

```
SHORTSIZ16 e1432_set_srcbuffer_mode(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_srcbuffer_mode(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 *mode)
SHORTSIZ16 e1432_set_srcbuffer_size(E1432ID hw, SHORTSIZ16 ID,
                                    LONGSIZ32 size)
SHORTSIZ16 e1432_get_srcbuffer_size(E1432ID hw, SHORTSIZ16 ID,
                                    LONGSIZ32 *size)
SHORTSIZ16 e1432_set_srcbuffer_init(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 initmode)
SHORTSIZ16 e1432_get_srcbuffer_init(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 *initmode)
```

DESCRIPTION

These functions specify how the option 1D4 single-channel source will work, when the source is in mode **E1432_SOURCE_MODE_ARB** and mode **E1432_SOURCE_MODE_BARB**.

For all of these functions, *hw* must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

e1432_set_srcbuffer_mode sets the buffer operation mode for the source addressed. The values for *mode* are:

E1432_SRCBUFFER_PERIODIC_A specifies that the source will loop on one buffer of data (the A buffer). The buffer size is determined by *e1432_set_srcbuffer_size*. This source buffer mode works for both source mode **E1432_SOURCE_MODE_ARB** and source mode **E1432_SOURCE_MODE_BARB**.

E1432_SRCBUFFER_PERIODIC_AB specifies that the source will loop on the both the A and B buffer (concatenated). The total buffer size is twice that set by *e1432_set_srcbuffer_size*. This source buffer mode works for both source mode **E1432_SOURCE_MODE_ARB** and source mode **E1432_SOURCE_MODE_BARB**.

E1432_SRCBUFFER_CONTINUOUS specifies that data will be continually loaded into the source from the host and played back at a rate determined by the source frequency parameters. In the event that insufficient data is available, the source will replay its last data point and an source overread status condition will be set. This mode uses both A and B buffers, alternately. Note: The at least one of the two source buffers must be pre-loaded with data prior to starting a measurement. This source buffer mode works only for source mode **E1432_SOURCE_MODE_ARB**, and not for **E1432_SOURCE_MODE_BARB**.

E1432_SRCBUFFER_ONESHOT specifies that one buffer of data will be sent out, and then the source will stop. When the source stops, it goes back to zero volts output. The buffer size is determined by *e1432_set_srcbuffer_size*. Only the A buffer is used in this mode, and this buffer should be pre-loaded before starting a measurement. This source buffer mode works only for source mode **E1432_SOURCE_MODE_ARB**, and not for **E1432_SOURCE_MODE_BARB**.

e1432_set_srcbuffer_size sets the source buffer size in the source itself. The source contains two buffers which are each set to *size* words.

e1432_set_srcbuffer_init initializes the source buffers as well as the substrate buffers. Two initialization options are specified by the values of *initmode*:

E1432_SRCBUFFER_INIT_EMPTY resets all buffers, substrate and source board, to empty.

E1432_SRCBUFFER_INIT_XFER resets all transfer buffers (substrate) only.

It is important to note that a *e1432_set_srcbuffer_init* call is necessary in order for new values of *mode* and *size* to take effect.

SOURCE DRAM USAGE

If an E1434 has DRAM installed, then the DRAM can be used to allow a larger-than-normal source buffer. This works only when the source buffer is in **E1432_SRCBUFFER_ONESHOT** of **E1432_SRCBUFFER_PERIODIC_A** modes, and allows for a large buffer of data to be loaded and then sent out the source. For E1432 or E1433 modules that have an option 1D4 source board installed, DRAM may also be used for a large source buffer, but only if there are **no** active input channels in the module.

The DRAM is split evenly among the active source channels in the module. However, on an E1434, if channel 2 is active but channel 1 is not active, then some of the DRAM is wasted on an unused buffer for channel 1. Similarly, if channel 4 is active but channel 3 is not active, then some of the DRAM is wasted on an unused buffer for channel 3. This means that the maximum source buffer size will be smaller than if the DRAM were not wasted. To avoid this situation, always use channel 1 or channel 3 when you only need one active source channel.

Each source sample takes up four bytes of DRAM. So, for example, if only source channel 1 is active and the DRAM size is 32 MB, then the source buffer can hold 8388608 samples. At a sample rate of 65536 Hz, this is 128 seconds of output signal.

A call to *e1432_set_srcbuffer_size* should specify the desired source buffer size. There is no special function to turn on the use of DRAM. DRAM is used automatically if it is available and the specified source buffer size is large enough to need it. Other than making a larger source buffer available to the host, the use of DRAM is transparent to the host program.

When using DRAM as a large source buffer, *e1432_write_srcbuffer_data* is no longer restricted to a transfer size of **E1432_SRC_DATA_NUMWORDS_MAX**. Instead, the transfer size can be as large as the source buffer.

The use of DRAM for source buffer works only when *e1432_set_source_mode* is set to **E1432_SOURCE_MODE_ARB**, and not when it is set to **E1432_SOURCE_MODE_BARB**.

RESET VALUES

After a reset, *mode* is set to **E1432_SRCBUFFER_PERIODIC_A**, and *size* is set to 1024.

RETURN VALUES

Each function returns 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_source_mode, e1432_write_srcbuffer_data, e1432_get_srcbuffer_size_limits

NAME

e1432_set_srcparm_mode − Set source parameter mode
e1432_get_srcparm_mode − Get current state of source parameter mode
e1432_update_srcparm − Update source with current parameters

SYNOPSIS

```
SHORTSIZ16 e1432_set_srcparm_mode(E1432ID hw, SHORTSIZ16 ID,
                                  LONGSIZ32 mode)
SHORTSIZ16 e1432_get_srcparm_mode(E1432ID hw, SHORTSIZ16 ID,
                                  LONGSIZ32 *mode)
SHORTSIZ16 e1432_update_srcparm(E1432ID hw, SHORTSIZ16 ID,
                                  LONGSIZ32 updatemode)
```

DESCRIPTION

e1432_set_srcparm_mode sets the source parameter mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_srcparm_mode returns the current value of the source parameter mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode determines the source parameter mode when the source is on. At the start of a measurement with an active source, the source parameters are sent to the source independent of the *mode* setting. The *mode* setting is effective after the start of the measurement. The valid values are:

E1432_SRCPARM_MODE_IMMED for immediate update of any source parameter (default),

E1432_SRCPARM_MODE_DEFR all source parameter changes are deferred until a call to *e1432_update_srcparm*.

The source parameters that can be set on the fly (and controlled by *mode*) are: amp_scale, range, freq, phase (as appropriate for the source mode).

e1432_update_srcparm updates the source with current parameters values using the method determined by the value given in *updatemode*, for a single channel or group of channels *ID*. It is to be called durring a measurement, after source parameter values are changed, when *mode* is **E1432_SRCPARM_MODE_DEFR**.

update_mode determines the method used in updating the source parameters. *update_mode* is ignored if *mode* is **E1432_SRCPARM_MODE_IMMED**. For all source channels, valid values are:

E1432_SRCPARM_UPDATE_IMMED for parameters changed immediately, when *e1432_update_srcparm* is called.

For sine or burst sine source channels, additional valid values are:

E1432_SRCPARM_UPDATE_IMMEDTRGOUT for parameters changed immediately and send a trigger out (trigger out not sent when source is burst mode),

E1432_SRCPARM_UPDATE_XING for parameters changed at next zero crossing,

E1432_SRCPARM_UPDATE_XINGTRGOUT for parameters changed at next zero crossing and send a trigger out (trigger out not sent when source is burst mode),

E1432_SRCPARM_UPDATE_TRGIN for parameters changed at next trigger (not avaliable on the source daughter board channel),

Obviously, these functions is are not useful when talking to input or tach channels. Only source channels have a srcparm mode.

RESET VALUE

After a reset, *mode* is set to **E1432_SRCPARM_MODE_IMMED**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

NAME

e1432_set_sumbus − Set driver for the VXI SUMBUS line
e1432_get_sumbus − Get current value of VXI SUMBUS driver

SYNOPSIS

```
SHORTSIZ16 e1432_set_sumbus(E1432ID hw, SHORTSIZ16 ID,
                            SHORTSIZ16 sumbus)
SHORTSIZ16 e1432_get_sumbus(E1432ID hw, SHORTSIZ16 ID,
                            SHORTSIZ16 *sumbus)
```

DESCRIPTION

e1432_set_sumbus sets the driver for the VXI SUMBUS line, for a single channel or group of channels *ID*, to the value given in *sumbus*.

e1432_get_sumbus returns the current setting of the VXI SUMBUS driver, of a single channel or group of channels *ID*, into a memory location pointed to by *sumbus*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

sumbus determines the driver for the SUMBUS line. This should be one of:

E1432_SUMBUS_OFF, to have nothing drive the line.

E1432_SUMBUS_CALOUT, to connect the internal CALOUT line to the SUMBUS. The CALOUT line can be driven by the optional internal source board.

RESET VALUE

After a reset, *sumbus* is set to **E1432_SUMBUS_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calin, e1432_set_cal_voltage, e1432_set_source_output

NAME

e1432_set_tach_decimate − Set tach decimation count
e1432_get_tach_decimate − Get tach decimation count

SYNOPSIS

```
SHORTSIZ16 e1432_set_tach_decimate(E1432ID hw, SHORTSIZ16 ID,
                                   LONGSIZ32 tach_decimate)
SHORTSIZ16 e1432_get_tach_decimate(E1432ID hw, SHORTSIZ16 ID,
                                   LONGSIZ32 *tach_decimate)
```

DESCRIPTION

e1432_set_tach_decimate sets the tach decimation count, of a single channel or group of channels *ID*, to the value given in *tach_decimate*.

e1432_get_tach_decimate returns the current value of the tach decimation count, of a single channel or group of channels *ID*, into a memory location pointed to by *tach_decimate*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

tach_decimate is the tach decimation count. This is the number of tach edges to **skip** between each tach edge that is kept. A value of zero means to use every tach edge (and is the default).

For input channels and source channels, this parameter is not used.

RESET VALUE

After a reset, the *tach_decimate* is set to 0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_tach_decimate_limits

NAME

e1432_set_tach_holdoff − Set tach holdoff time
e1432_get_tach_holdoff − Get tach holdoff time

SYNOPSIS

```
SHORTSIZ16 e1432_set_tach_holdoff(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 tach_holdoff)
SHORTSIZ16 e1432_get_tach_holdoff(E1432ID hw, SHORTSIZ16 ID,
                                  FLOATSIZ32 *tach_holdoff)
```

DESCRIPTION

e1432_set_tach_holdoff sets the tach holdoff time, of a single channel or group of channels *ID*, to the value given in *tach_holdoff*.

e1432_get_tach_holdoff returns the current value of the tach holdoff time, of a single channel or group of channels *ID*, into a memory location pointed to by *tach_holdoff*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

tach_holdoff is the tach holdoff time, in seconds. This is the amount of time after a valid tach edge, before the tach channel will start looking for another tach edge. A value of zero is silently adjusted to the minimum holdoff time supported by the hardware.

For input channels and source channels, this parameter is not used.

RESET VALUE

After a reset, the *tach_holdoff* defaults to the smallest non-zero value supported by the tach channel. For the Option AYF tachometer board, this is approximately 10 microseconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_tach_holdoff_limits

NAME

e1432_set_tach_irq_number − Set number of tachs per irq
e1432_get_tach_irq_number − Get number of tachs per irq

SYNOPSIS

```
SHORTSIZ16 e1432_set_tach_irq_number(E1432ID hw, SHORTSIZ16 ID,
                                     LONGSIZ16 number)
SHORTSIZ16 e1432_get_tach_irq_number(E1432ID hw, SHORTSIZ16 ID,
                                     LONGSIZ16 *number)
```

DESCRIPTION

e1432_set_tach_irq_number sets the number of tachs present in the internal raw tach buffer before the **E1432_IRQ_TACHS_AVAIL** bit is set in the **E1432_IRQ_STATUS2_REG** register and, if enabled, a tach available interrupt generated. This can be used in the host to regulate the frequency at which the *e1432_get_raw_tachs* or *e1432_send_tachs* functions are called.

e1432_get_tach_irq_number returns the number of tachs per irq.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

number selects the number of tachs in the raw tach buffer necessary to set the tach available bit.

RESET VALUE

After a reset, *number* is set to **64**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_send_tachs, e1432_get_raw_tachs

NAME

e1432_set_tach_max_time − Set tach maximum time between pulses
e1432_get_tach_max_time − Get tach maximum time between pulses

SYNOPSIS

```
SHORTSIZ16 e1432_set_tach_max_time(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 tach_max_time)
SHORTSIZ16 e1432_get_tach_max_time(E1432ID hw, SHORTSIZ16 ID,
                                   FLOATSIZ32 *tach_max_time)
```

DESCRIPTION

e1432_set_tach_max_time sets the maximum time between tachometer pulses of a single channel or group of channels, *ID*.

e1432_get_tach_max_time returns the current value of the tach maximum time, of a single channel or group of channels *ID*, into a memory location pointed to by *tach_max_time*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of tach channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

tach_max_time is the maximum amount of time between valid tach edges before the rpm value of the tach channel is set to zero, indicating that the tach may be disconnected.

For input channels and source channels, this parameter is not used.

RESET VALUE

After a reset, the *tach_max_time* defaults to 30.0 seconds.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_tach_max_time_limits

NAME

e1432_set_tach_ppr − Set tach pulses per revolution
e1432_get_tach_ppr − Get tach pulses per revolution

SYNOPSIS

```
SHORTSIZ16 e1432_set_tach_ppr(E1432ID hw, SHORTSIZ16 ID,
                              FLOATSIZ32 tach_ppr)
SHORTSIZ16 e1432_get_tach_ppr(E1432ID hw, SHORTSIZ16 ID,
                              FLOATSIZ32 *tach_ppr)
```

DESCRIPTION

e1432_set_tach_ppr sets the number of tachometer pulses per revolution, of a single channel or group of channels *ID*, to the value given in *tach_ppr*.

e1432_get_tach_ppr returns the current value of the tach pulses per revolution, of a single channel or group of channels *ID*, into a memory location pointed to by *tach_ppr*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

tach_ppr is the tach pulses per revolution.

For input channels and source channels, this parameter is not used.

Note: The E1432/3/4 tachometer inputs have a bandwidth of 100kHz, representing 8333 RPM at 720 PPR. However, this 100kHz rate will not be usable in many circumstances.

The E1432/3/4 order-tracking algorithm for resampling time data has no theoretical limits on maximum ppr. There are, however, practical limits as a result of available processing power; order-tracking is a very computationally intensive algorithm. Settings of 0.5 to 4 for *tach_ppr* seem optimal for most applications. If the DUT is producing more than 4 PPR, it is possible to reduce the amount of tach processing required with the aid of the *e1432_set_tach_decimate* function.

When the resample mode (as set by *e1432_set_sample_mode*) is set to **E1432_RESAMP_AT_TACH**, the order tracking algorithm used is less computationally intensive. This should increase the practical limit on *tach_ppr* somewhat.

The best way to determine what these practical limitations are is by experimentation. Factors that affect this processing include ppr, channel count, module count, block size, sample rate, amount of DRAM available, and device RPM. System aspects which may have an affect include the host interface hardware, the host, and programming environment.

RESET VALUE

After a reset, the *tach_ppr* is set to 1.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_get_tach_ppr_limits, e1432_set_sample_mode, e1432_set_tach_decimate

NAME

e1432_set_trigger − Set all trigger parameters except auto trigger

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger(E1432ID hw, SHORTSIZ16 ID,
                             SHORTSIZ16 chanState,
                             LONGSIZ32 delay, FLOATSIZ32 lowLevel,
                             FLOATSIZ32 highLevel, SHORTSIZ16 slope,
                             SHORTSIZ16 mode)
```

DESCRIPTION

e1432_set_trigger sets all parameters associated with the trigger section of an E1432 or group of E1432s, except the auto trigger mode (see *e1432_set_auto_trigger*).

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

chanState determines if the channel or group of channels can digitally trigger a measurement. **E1432_CHANNEL_ON** enables it to trigger a measurement and sets the channel active, if it is not already in that state. **E1432_CHANNEL_OFF** disables it from triggering the measurement. This parameter may also be set with *e1432_set_trigger_channel* in conjunction with *e1432_set_active*.

delay is the elapsed time, in number of samples, between the occurrence of a trigger and the beginning of the data acquisition. A negative delay indicates a pre-trigger condition, where samples prior to the trigger event are included in the first measurement block. The maximum and minimum amounts of trigger delay are determined by the amount of ram available and the number of active channels in the E1432 modules. This parameter may also be set with *e1432_set_trigger_delay*.

lowLevel and *highLevel* are the value of the two trigger levels. Their value ranges from roughly **-125%** to **+125%**. These trigger levels may also be set with *e1432_set_trigger_level*. When *mode* is set to **E1432_TRIGGER_MODE_LEVEL**, the difference between the two levels controls the amount of noise rejection, 10% is a good value to use here. For the AYF tachometer option the values are in volts and fall between -25.0 and +25.0.

slope selects the edge of the trigger source (i.e. the direction) on which the trigger occurs. **E1432_TRIGGER_SLOPE_POS** sets it to a positive crossing of the highLevel, or to an exit of the bounded zone. **E1432_TRIGGER_SLOPE_NEG** sets it to a negative crossing of the lowLevel, or to an entry into the bounded zone. This parameter may also be set with *e1432_set_trigger_slope*.

mode selects the operating mode of the trigger detection. **E1432_TRIGGER_MODE_LEVEL** selects the positive or negative crossing of a unique trigger level. **E1432_TRIGGER_MODE_BOUND** selects the exit from or entry to a zone bounded by the two trigger levels. This parameter may also be set with *e1432_set_trigger_mode*.

RESET VALUE

After a reset, *chanState* is set to **E1432_CHANNEL_ON**, *delay* is set to **0**, *lowLevel* is set to **-10%** for input channels and **-0.05** volts for tach channels, *highLevel* is set to **0%** for input channels and **0.0** volts for tach channels, *slope* is set to **E1432_TRIGGER_SLOPE_POS**, and *mode* is set to

E1432_TRIGGER_MODE_LEVEL.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_auto_trigger, e1432_set_trigger_channel, e1432_set_trigger_delay, e1432_set_trigger_level, e1432_set_trigger_mode, e1432_set_trigger_slope

NAME

e1432_set_trigger_channel − Enable a channel to generate triggers
e1432_get_trigger_channel − Get current trigger setting

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_channel(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 state)
SHORTSIZ16 e1432_get_trigger_channel(E1432ID hw, SHORTSIZ16 ID,
                                     SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_trigger_channel sets a single channel or group of channels, *ID*, to digitally trigger a measurement, depending on the value given in *state*.

e1432_get_trigger_channel returns the current value of the digital trigger source, of a single channel or group of channels *ID*, into a memory location pointed to by *state*.

This parameter may also be set with *e1432_set_trigger*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state determines if the channel or group of channels can digitally trigger a measurement.

E1432_CHANNEL_OFF disables it from triggering the measurement.

E1432_CHANNEL_ON enables it to trigger a measurement, and also enables a trigger channel on an Option AYF tachometer board to pre-arm the measurement if the pre-arm mode is **E1432_ARM_EXTERNAL**.

E1432_CHANNEL_PRE_ARM enables the channel to pre-arm a measurement, but not trigger the measurement, and is valid only for the trigger channel on an Option AYF tachometer board.

In addition to selecting a channel or group of channels as a source for triggering a measurement, it is possible to set up the following trigger parameters for each of the selected channels: the level may be set with *e1432_set_trigger_level*, the slope may be set with *e1432_set_trigger_slope*, and the mode may be set with *e1432_set_trigger_mode*.

A channel or group of channel which has been set to digitally trigger the measurement will implicitly do that on a local (i.e. module) basis. If it is required that these local trigger conditions translate into system wide (i.e. multiple modules) conditions, it is necessary to configure the module to act upon the system sync/arm/trigger line (see *e1432_set_multi_sync*). This is normally handled automatically by *e1432_init_measure*.

Input channels can be set to trigger the system when the input signal crosses a threshold, or when the input signal exceeds programmed bounds. Source channels can be set to trigger the system at the start of a "burst". Some tach channels can be set to trigger the system when the signal crosses a threshold.

For the Option AYF tachometer, only the second of the two tach channels can be set to trigger in the non-RPM arming modes; the first tach channel is not capable of triggering. In the RPM arming modes, either channel can be used for RPM arming/triggering. If the module's pre-arm mode is set to **E1432_ARM_EXTERNAL**, then the second of the two tach channels can be used to pre-arm the measurement.

RESET VALUE

After a reset, *state* is set to **E1432_CHANNEL_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_multi_sync, e1432_set_trigger, e1432_set_trigger_level, e1432_set_trigger_mode, e1432_set_trigger_slope, e1432_set_pre_arm_mode.

NAME

e1432_set_trigger_delay − Set trigger delay
e1432_get_trigger_delay − Get current trigger delay

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_delay(E1432ID hw, SHORTSIZ16 ID,
                                   LONGSIZ32 delay)
SHORTSIZ16 e1432_get_trigger_delay(E1432ID hw, SHORTSIZ16 ID,
                                   LONGSIZ32 *delay)
```

DESCRIPTION

e1432_set_trigger_delay sets the trigger delay, of a single channel or group of channels *ID*, to the value given in *delay*.

e1432_get_trigger_delay returns the current value of the delay, of a single channel or group of channels *ID*, into a memory location pointed to by *delay*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

delay is the elapsed time, in number of samples, between the occurrence of a trigger and the beginning of the data acquisition. A negative delay indicates a pre-trigger condition, where samples prior to the trigger event are included in the first measurement block. The minimum and maximum values for this parameter vary depending on the amount of ram and the number of active channels in the E1432 modules. This parameter may also be set with *e1432_set_trigger*.

NOTE: When order tracking is enabled with the e1432_set_calc_data, or when doing RPM arming (set by *e1432_set_arm_mode*) there are only three valid trigger delay values:

0	trigger is at the beginning of the block of data
-blocksize/2	trigger is at the middle of the block of data
-blocksize	trigger is at the end of the block of data

and any other values will be flagged as an illegal trigger delay value when the measurement is started.

NOTE: Use of a trigger delay other then 0 with Octave measurements (as set with *e1432_set_octave_mode*) will have indeterminate results.

When the data size is set to **E1432_DATA_SIZE_16**, then the *delay* will be rounded down to an even number.

RESET VALUE

After a reset, *delay* is set to **0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_arm_mode, e1432_set_calc_data, e1432_set_trigger, e1432_get_trigger_delay_limits

NAME

e1432_set_trigger_ext − Set external trigger mode
e1432_get_trigger_ext − Get current external trigger mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_ext(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 ext)
SHORTSIZ16 e1432_get_trigger_ext(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 *ext)
```

DESCRIPTION

Each E1432 has an external trigger input which can be used to pre-arm and/or trigger a measurement. This trigger input is not part of any SCA, and is used only when there is no option 1D4 source board and no option AYF tach/trigger board plugged into the rear connector of the E1432. This trigger input is a TTL input, so the trigger level can't be adjusted for this input.

e1432_set_trigger_ext sets the external trigger mode for one or more E1432 modules.

e1432_get_trigger_ext returns the current external trigger mode, of a single channel or group of channels *ID*, in the memory location pointed to by *ext*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ext specifies the external trigger mode for the modules referred to by *ID*. The valid choices are:

E1432_TRIGGER_EXT_OFF, which disables the external trigger;

E1432_TRIGGER_EXT_POS, which enables the external trigger and uses the positive edge of the trigger to pre-arm or trigger a measurement;

E1432_TRIGGER_EXT_NEG, which enables the external trigger and uses the negative edge of the trigger to pre-arm or trigger a measurement;

E1432_TRIGGER_EXT_PREARM_POS, which enables the external trigger for pre-arm, but not for triggering, and uses the positive edge of the trigger to pre-arm the measurement;

E1432_TRIGGER_EXT_PREARM_NEG, which enables the external trigger for pre-arm, but not for triggering, and uses the negative edge of the trigger to pre-arm the measurement.

The last two "prearm" values are useful only if the pre-arm mode is set to **E1432_ARM_EXTERNAL**, see *e1432_set_pre_arm_mode* for details.

RESET VALUE

After a reset, *ext* is set to **E1432_TRIGGER_EXT_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_trigger_channel, e1432_set_trigger_slope, e1432_set_pre_arm_mode

NAME

e1432_set_trigger_level − Set trigger level
e1432_get_trigger_level − Get current trigger level

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_level(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 whichLevel,
                                   FLOATSIZ32 level)
SHORTSIZ16 e1432_get_trigger_level(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 whichLevel,
                                   FLOATSIZ32 *level)
```

DESCRIPTION

e1432_set_trigger_level sets one of the two trigger levels, of a single channel or group of channels *ID*, to the value given in *level*.

e1432_get_trigger_level returns one of the two current trigger levels, of a single channel or group of channels *ID*, into a memory location pointed to by *level*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

whichLevel selects one of the two trigger levels. **E1432_TRIGGER_LEVEL_LOWER** sets the low level, **E1432_TRIGGER_LEVEL_UPPER** sets the high level.

level is the value of the trigger level, expressed as a percentage of the current input range. Its value can be from roughly **-125%** to **+125%**.

For a tach channel, there is no "full scale", so the trigger level is expressed as an absolute voltage. The Option AYF tachometer board can set the trigger level between +-25 Volts.

For source channels, this function is not used, since it doesn’t make sense to specify a trigger level for a source.

RESET VALUE

After a reset, the lower level is set to **-10%** for input channel and **-0.05** volts for tach channels. The higher level is set to **0%** for input channels and **0.0** volts for tach channels.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_trigger_mode, e1432_set_trigger, e1432_get_trigger_level_limits

NAME

e1432_set_trigger_master − Set trigger master state
e1432_get_trigger_master − Get current trigger master state

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_master(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 state)
SHORTSIZ16 e1432_get_trigger_master(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_trigger_master sets the trigger master state of a module when using one of the RPM arming modes in a multiple module system. These RPM arming mode are **E1432_ARM_RPM_RUNUP, E1432_ARM_RPM_RUNDOWN,** and **E1432_ARM_RPM_DELTA**, and are set by *e1432_set_arm_mode*. This is used to allow the module with the AYF tachometer option to be the master module. This module then sends each trigger point to the host computer, which relays it to all slave modules using the *e1432_send_trigger* function.

e1432_get_trigger_master returns the current trigger master state of a module.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state enables/disables the ability of the module to be a master. The legal values are **E1432_TRIGGER_MASTER_ON** and **E1432_TRIGGER_MASTER_OFF**. At each trigger point on the master module while it is in one of the RPM arming modes, an index of the trigger point in the data FIFO is stored for transmission to the host and the **E1432_IRQ_TRIGGER** bit in the **E1432_IRQ_STATUS2_REG** register is set. This bit can be polled or interrupted upon using *e1432_set_interrupt_mask*. When this bit is set, it is up to the host program to call the *e1432_send_trigger* function, which will read the trigger index from the master module and send it to all of the slave modules. Since the data FIFOs on all modules operate synchronously, this FIFO index will be the exact trigger point in the slave modules as well. This triggering scheme only works when the data mode has been set by the *e1432_set_data_mode* function to be **E1432_DATA_MODE_OVERLAP_BLOCK**, so that multiple triggers can be stored.

If *ID* is a groupID and the *state* is **E1432_TRIGGER_MASTER_ON** the first module associated with the group is picked. If *ID* is a channel ID then the module with that channel is picked.

RESET VALUE

After a reset, *state* is set to **E1432_TRIGGER_MASTER_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_send_trigger, e1432_set_arm_mode, e1432_set_data_mode

NAME

e1432_set_trigger_mode − Set trigger mode (either level or bound)
e1432_get_trigger_mode − Get current trigger mode

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_mode(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 mode)
SHORTSIZ16 e1432_get_trigger_mode(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *mode)
```

DESCRIPTION

e1432_set_trigger_mode sets the trigger mode, of a single channel or group of channels *ID*, to the value given in *mode*.

e1432_get_trigger_mode returns the current value of the mode, of a single channel or group of channels *ID*, into a memory location pointed to by *mode*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

mode selects the area covered by the trigger detection. **E1432_TRIGGER_MODE_LEVEL** selects the positive or negative crossing of a unique trigger level. **E1432_TRIGGER_MODE_BOUND** selects the exit from or entry to a zone bounded by two trigger levels. See *e1432_set_trigger_slope*, for the direction which is effective. If the trigger slope is positive, the zone is defined as either crossing the level upwards, or exiting the zone. If the trigger slope is negative, the zone is defined as either crossing the level downwards, or entering the zone. This parameter may also be set with *e1432_set_trigger*.

For input channels, the **E1432_TRIGGER_MODE_LEVEL** and **E1432_TRIGGER_MODE_BOUND** values are both valid. For tach channels, only **E1432_TRIGGER_MODE_LEVEL** makes sense, so the others are not valid. For source channels, neither of these values makes any sense.

RESET VALUE

After a reset, *mode* is set to **E1432_TRIGGER_MODE_LEVEL**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_trigger, e1432_set_trigger_slope

NAME

e1432_set_trigger_slope − Set slope of trigger
e1432_get_trigger_slope − Get current slope of trigger

SYNOPSIS

```
SHORTSIZ16 e1432_set_trigger_slope(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 slope)
SHORTSIZ16 e1432_get_trigger_slope(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 *slope)
```

DESCRIPTION

e1432_set_trigger_slope sets the trigger slope, of a single channel or group of channels *ID*, to the value given in *slope*.

e1432_get_trigger_slope returns the current value of the slope, of a single channel or group of channels *ID*, into a memory location pointed to by *slope*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

slope selects the edge of the trigger source (i.e. the direction) on which the trigger occurs. **E1432_TRIGGER_SLOPE_POS** sets it to a positive going crossing of the level. **E1432_TRIGGER_SLOPE_NEG** sets it to a negative going crossing of the level. This parameter may also be set with *e1432_set_trigger*. If the trigger mode (see *e1432_set_trigger_mode*) is bound, positive is synonymous of exiting the zone defined by the two trigger levels, and negative is synonymous of entering it.

This function does not apply to source channels, since the concept of a trigger slope makes no sense for a source channel.

RESET VALUE

After a reset, *slope* is set to **E1432_TRIGGER_SLOPE_POS**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_trigger, e1432_set_trigger_mode

NAME

e1432_set_triggers_per_arm − Set number of triggers done for each arm
e1432_get_triggers_per_arm − Get number of triggers done for each arm

SYNOPSIS

```
SHORTSIZ16 e1432_set_triggers_per_arm(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 n)
SHORTSIZ16 e1432_get_triggers_per_arm(E1432ID hw, SHORTSIZ16 ID,
                                      LONGSIZ32 *n)
```

DESCRIPTION

When a measurement is running, an arm must take place before the E1432 is ready to receive a trigger. This function is used to set the number of triggers which are processed before the measurement loop looks for another arm.

e1432_get_triggers_per_arm returns the current number of triggers processed per arm, of a single channel or group of channels *ID*, in the memory location pointed to by *n*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

n specifies the number of triggers processed per arm. If this value is negative, an error occurs. If this number is zero, then an infinite number of triggers will be processed. If this number is positive, then that many triggers will be processed for each arm.

RESET VALUE

After a reset, *n* is set to 1.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_auto_trigger, e1432_set_arm_mode

NAME

e1432_set_try_recover − Control signal trapping

SYNOPSIS

```
void e1432_set_try_recover(LONGSIZ32 state)
```

DESCRIPTION

e1432_set_try_recover controls the signal trapping functions of the E1432 library. If *state* is non zero, bus error signals are trapped by the library and dealt with, usually by terminating. If *state* is zero, the library does not trap the signal, which will allow user software to deal with the signal.

RESET VALUE

Default is 0, bus errors not trapped. This is different than the E1431 library, which defaults to trapping bus errors.

RETURN VALUE

This function does not return a value.

NAME

e1432_set_ttltrg_clock − Select a TTLTRG line for freerun clock
e1432_get_ttltrg_clock − Get current TTLTRG line for freerun clock

SYNOPSIS

```
SHORTSIZ16 e1432_set_ttltrg_clock(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 ttltrg)
SHORTSIZ16 e1432_get_ttltrg_clock(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *ttltrg)
```

DESCRIPTION

e1432_set_ttltrg_clock sets the VXI TTLTRG line used for the freerunning clock. If a measurement is in progress while calling this function, the measurement is aborted. The freerunning clock can be shared by several E1432 modules, to ensure that they sample data at the same time.

e1432_get_ttltrg_clock returns the currently selected VXI TTLTRG line for the freerunning clock in the memory location pointed to by *ttltrg*.

An alternative way to specify TTLTRG line for the freerunning clock is to use the *e1432_set_ttltrg_lines* function. That function restricts the choice of TTLTRG line to those compatible with the E1431 8-Channel VXI Input.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ttltrg selects a VXI ttltrg line to be used for the freerunning sample clock.

E1432_TTLTRG_0 selects TTLTRG0 for the clock. **E1432_TTLTRG_1** selects TTLTRG1 for the clock.
E1432_TTLTRG_2 selects TTLTRG2 for the clock. **E1432_TTLTRG_3** selects TTLTRG3 for the clock.
E1432_TTLTRG_4 selects TTLTRG4 for the clock. **E1432_TTLTRG_5** selects TTLTRG5 for the clock.
E1432_TTLTRG_6 selects TTLTRG6 for the clock. **E1432_TTLTRG_7** selects TTLTRG7 for the clock.

The mere selection of the TTLTRG line by this function does not necessarily cause the clock to get driven onto that TTLTRG line. Normally, the clock is driven onto the TTLTRG line only when running a multi-module measurement. This is taken care of automatically during *e1432_init_measure* if *e1432_set_auto_group_meas* is on.

When *e1432_set_auto_group_meas* is off, the TTLTRG line is driven only if one of the following conditions is met:

1. *e1432_set_clock_master* is **E1432_MASTER_CLOCK_ON**.

2. *e1432_set_multi_sync* is **E1432_SYSTEM_SYNC_ON**.

3. *e1432_set_clock_source* is **E1432_CLOCK_SOURCE_VXI** or **E1432_CLOCK_VXI_DEC_3**.

RESET VALUE

After a reset, *ttltrg* is set to **E1432_TTLTRG_1**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_gclock, e1432_set_ttltrg_satrg, e1432_set_ttltrg_trigger, e1432_set_ttltrg_lines

NAME

e1432_set_ttltrg_gclock − Select a TTLTRG line for gated clock
e1432_get_ttltrg_gclock − Get current TTLTRG line for gated clock

SYNOPSIS

```
SHORTSIZ16 e1432_set_ttltrg_gclock(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 ttltrg)
SHORTSIZ16 e1432_get_ttltrg_gclock(E1432ID hw, SHORTSIZ16 ID,
                                   SHORTSIZ16 *ttltrg)
```

DESCRIPTION

Gated clock lines are not normally used by the E1432 module, so this function is not normally used.

e1432_set_ttltrg_gclock sets the VXI TTLTRG line used for the gated clock. If a measurement is in progress while calling this function, the measurement is aborted. The gated clock is normally turned on only when data is actively being collected for a block of data that will be sent to the host. Other VXI devices could use this clock and know that they are sampling data at the same time as the E1432.

e1432_get_ttltrg_gclock returns the currently selected VXI TTLTRG line for the gated clock in the memory location pointed to by *ttltrg*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ttltrg selects a VXI ttltrg line to be used for the gated sample clock.

E1432_TTLTRG_0 selects TTLTRG0 for the clock. **E1432_TTLTRG_1** selects TTLTRG1 for the clock.
E1432_TTLTRG_2 selects TTLTRG2 for the clock. **E1432_TTLTRG_3** selects TTLTRG3 for the clock.
E1432_TTLTRG_4 selects TTLTRG4 for the clock. **E1432_TTLTRG_5** selects TTLTRG5 for the clock.
E1432_TTLTRG_6 selects TTLTRG6 for the clock. **E1432_TTLTRG_7** selects TTLTRG7 for the clock.

The mere selection of the TTLTRG line by this function does not necessarily cause the clock to get driven onto that TTLTRG line. The TTLTRG line is driven only if one of the following conditions is met:

1. *e1432_set_multi_sync* is **E1432_SYSTEM_SYNC_VXD** or **E1432_SYSTEM_SYNC_VXD_MIN**.

2. *e1432_set_clock_source* is **E1432_CLOCK_SOURCE_VXI** or **E1432_CLOCK_VXI_DEC_3**.

RESET VALUE

After a reset, *ttltrg* is set to **E1432_TTLTRG_1**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_clock, e1432_set_ttltrg_satrg, e1432_set_ttltrg_trigger, e1432_set_ttltrg_lines

NAME

e1432_set_ttltrg_lines − Select a pair of sync/clock lines
e1432_get_ttltrg_lines − Get current selection of sync/clock lines

SYNOPSIS

```
SHORTSIZ16 e1432_set_ttltrg_lines(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 ttltrg)
SHORTSIZ16 e1432_get_ttltrg_lines(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *ttltrg)
```

DESCRIPTION

e1432_set_ttltrg_lines sets the system VXI TTLTRG lines, of a single channel or group of channels *ID*, to the value given in *ttltrg*. If a measurement is in progress while calling this function, the measurement is aborted.

e1432_get_ttltrg_lines returns the currently selected VXI TTLTRG lines, of a single channel or group of channels *ID*, into a memory location pointed to by *ttltrg*.

An alternative way to specify TTLTRG lines is to use the *e1432_set_ttltrg_clock* and *e1432_set_ttltrg_satrg* functions. Those functions allow arbitrary choices for the TTLTRG lines. However, *e1432_set_ttltrg_lines* ensures that the pair of lines chosen is compatible with the E1431 8-Channel VXI Input module.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ttltrg selects a pair of VXI ttltrg lines to be used for the system lines. In the following list, the first line is the one used for the Sync/Arm/Trigger line (i.e. controlling measurement loop transitions: booting, synchronizing and settling, arming, triggering and measuring), and the second line is the one used for a free-running system clock.

E1432_TTLTRG_01 selects TTLTRG0 for SYNC, and TTLTRG1 for clock. **E1432_TTLTRG_23** selects TTLTRG2 for SYNC, and TTLTRG3 for clock. **E1432_TTLTRG_45** selects TTLTRG4 for SYNC, and TTLTRG5 for clock. **E1432_TTLTRG_67** selects TTLTRG6 for SYNC, and TTLTRG7 for clock.

RESET VALUE

After a reset, *ttltrg* is set to **E1432_TTLTRG_01**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_clock, e1432_set_ttltrg_gclock, e1432_set_ttltrg_satrg, e1432_set_ttltrg_trigger

NAME

e1432_set_ttltrg_satrg − Select a TTLTRG line for sync/arm/trigger
e1432_get_ttltrg_satrg − Get current TTLTRG line for sync/arm/trigger

SYNOPSIS

```
SHORTSIZ16 e1432_set_ttltrg_satrg(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 ttltrg)
SHORTSIZ16 e1432_get_ttltrg_satrg(E1432ID hw, SHORTSIZ16 ID,
                                  SHORTSIZ16 *ttltrg)
```

DESCRIPTION

e1432_set_ttltrg_satrg sets the VXI TTLTRG line used for sync/arm/trigger between E1432 modules. If a measurement is in progress while calling this function, the measurement is aborted. The sync/arm/trigger can be shared by several E1432 modules, to ensure that they sync and trigger at the same time.

e1432_get_ttltrg_satrg returns the currently selected VXI TTLTRG line for sync/arm/trigger in the memory location pointed to by *ttltrg*.

An alternative way to specify TTLTRG line for sync/arm/trigger is to use the *e1432_set_ttltrg_lines* function. That function restricts the choice of TTLTRG line to those compatible with the E1431 8-Channel VXI Input.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ttltrg selects a VXI ttltrg line to be used for sync/arm/trigger.

E1432_TTLTRG_0 selects TTLTRG0 for satrg. **E1432_TTLTRG_1** selects TTLTRG1 for satrg. **E1432_TTLTRG_2** selects TTLTRG2 for satrg. **E1432_TTLTRG_3** selects TTLTRG3 for satrg. **E1432_TTLTRG_4** selects TTLTRG4 for satrg. **E1432_TTLTRG_5** selects TTLTRG5 for satrg. **E1432_TTLTRG_6** selects TTLTRG6 for satrg. **E1432_TTLTRG_7** selects TTLTRG7 for satrg.

The mere selection of the TTLTRG line by this function does not necessarily cause the sync/arm/trigger to get driven onto that TTLTRG line. Normally, the sync/arm/trigger is driven onto the TTLTRG line only when running a multi-module measurement. This is taken care of automatically during *e1432_init_measure* if *e1432_set_auto_group_meas* is on.

When *e1432_set_auto_group_meas* is off, the TTLTRG line is driven only if one of the following conditions is met:

1. *e1432_set_clock_master* is **E1432_MASTER_CLOCK_ON**.

2. *e1432_set_multi_sync* is **E1432_SYSTEM_SYNC_ON**.

3. *e1432_set_clock_source* is **E1432_CLOCK_SOURCE_VXI** or **E1432_CLOCK_VXI_DEC_3**.

RESET VALUE

After a reset, *ttltrg* is set to **E1432_TTLTRG_0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_clock, e1432_set_ttltrg_gclock, e1432_set_ttltrg_trigger, e1432_set_ttltrg_lines

NAME

e1432_set_ttltrg_trigger − Select a TTLTRG line for once-per-loop trigger
e1432_get_ttltrg_trigger − Get current TTLTRG line for once-per-loop trigger

SYNOPSIS

```
SHORTSIZ16 e1432_set_ttltrg_trigger(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 ttltrg)
SHORTSIZ16 e1432_get_ttltrg_trigger(E1432ID hw, SHORTSIZ16 ID,
                                    SHORTSIZ16 *ttltrg)
```

DESCRIPTION

The one-pulse-per-loop trigger is not normally used by the E1432 module, so this function is not normally used.

e1432_set_ttltrg_trigger sets the VXI TTLTRG line used for a one-pulse-per-loop trigger. If a measurement is in progress while calling this function, the measurement is aborted. The one-pulse-per-loop trigger is asserted at the start of each block of data acquired from an E1432. It is not asserted to synchronise the modules at the start of a measurement, nor does it get asserted before the arm state as the sync/arm/trigger line does. This line is therefore not compatible with the E1431 8-channel input module.

e1432_get_ttltrg_trigger returns the currently selected VXI TTLTRG line for the one-pulse-per-loop trigger in the memory location pointed to by *ttltrg*.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

ttltrg selects a VXI ttltrg line to be used for the one-pulse-per-loop trigger.

E1432_TTLTRG_0 selects TTLTRG0 for the trigger. **E1432_TTLTRG_1** selects TTLTRG1 for the trigger. **E1432_TTLTRG_2** selects TTLTRG2 for the trigger. **E1432_TTLTRG_3** selects TTLTRG3 for the trigger. **E1432_TTLTRG_4** selects TTLTRG4 for the trigger. **E1432_TTLTRG_5** selects TTLTRG5 for the trigger. **E1432_TTLTRG_6** selects TTLTRG6 for the trigger. **E1432_TTLTRG_7** selects TTLTRG7 for the trigger.

The mere selection of the TTLTRG line by this function does not necessarily cause the trigger to get driven onto that TTLTRG line. The TTLTRG line is driven only if one of the following conditions is met:

1. *e1432_set_multi_sync* is **E1432_SYSTEM_SYNC_VXD** or **E1432_SYSTEM_SYNC_VXD_MIN**.

2. *e1432_set_clock_source* is **E1432_CLOCK_SOURCE_VXI** or **E1432_CLOCK_VXI_DEC_3**.

RESET VALUE

After a reset, *ttltrg* is set to **E1432_TTLTRG_0**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_ttltrg_clock, e1432_set_ttltrg_satrg, e1432_set_ttltrg_gclock, e1432_set_ttltrg_lines

NAME

e1432_set_user_data − Set user data parameters.

SYNOPSIS

```
SHORTSIZ16 e1432_set_user_data(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 frame_length,
                               SHORTSIZ16 word_length,
                               SHORTSIZ16 sub_length,
                               SHORTSIZ16 sub_pos)
```

DESCRIPTION

e1432_set_user_data sets up the necessary parameters for handling user data from SCAs. *e1432_set_user_data* is applicable only when *e1432_sca_dsp_download* has been used to download alternative DSP programs to the SCAs which interleave user data bits/words with time data bits/words.

After a successful call to *e1432_set_user_data*, **E1432_DATA_USER1** becomes a valid parameter for *e1432_set_calc_data*, **E1432_ENABLE_TYPE_USER1** becomes a valid parameter for *e1432_set_enable*, and **E1432_USER1_DATA** becomes a valid parameter for *e1432_read_raw_data*.

These parameter are "global" parameters. They apply to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

frame_length specifies how many user data points constitute a "frame". Data returned by *e1432_read_raw_data* is frame aligned. *frame_length* also becomes the block length for **E1432_USER1_DATA**. *frame_length* must be greater than or equal to 1.

word_length is the length, in bits of a user data point. *word_length* must be less than or equal to 32, the length of the words returned by *e1432_read_raw_data*. A *word_length* of 0 turns off the user data mode. Otherwise, *word_length* must be greater than or equal to 2. *word_length* must be 32 for interleaved 16 bit time data, 16 bit user data.

sub_length is the number of bits in the lower bits of an SCA data word that are user data bits. *sub_length* must be greater than or equal to 2 and less than or equal to 16. *sub_length* must be 16 for interleaved 16 bit time data, 16 bit user data. *sub_length* should divide evenly into *word_length* with a quotient greater than or equal to 1.

sub_pos is the position of the user bits in the SCA data word. It should be 0 for interleaved 16 bit time data, 16 bit user data. For *sub_length* less than 16, it should be 8, to account for the status bits being placed in the lower 8 bits of the SCA data word.

RESET VALUE

After a reset, the user data mode is not in effect (*word_length* is 0).

RETURN VALUE

Return 0 if successful, **ERR1432_HARDWARE_INCAPABLE** when the parameters cannot be accommodated.

SEE ALSO

e1432_dsp_exec_query, e1432_sca_dsp_download

NAME

e1432_set_user_window − Download arbitary FFT window type

SYNOPSIS

```
SHORTSIZ16 e1432_set_user_window(E1432ID hw, SHORTSIZ16 ID,
                                 FLOATSIZ64 *buf, FLOATSIZ64 scale)
```

DESCRIPTION

e1432_set_user_window downloads an arbitrary FFT window type. This window is applied to input data as part of the FFT. The FFT can be turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** or **E1432_DATA_ORDER** parameter. Windowing is used to reduce leakage effects caused by input frequency components that are not multiples of (effective_clock_freq)/(input blocksize).

e1432_set_user_window also sets the window type to **E1432_WINDOW_USER1**. Any call to *e1432_set_window* that changes the window type away from **E1432_WINDOW_USER1** will erase the user window.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

buf is an array of size **E1432_WINDOW_SIZE_MAX** defining the arbitrary window. Note that this array must be the full size even if the blocksize used is smaller that **E1432_WINDOW_SIZE_MAX**.

scale is the scale factor for the user window, used to scale the FFTed data properly. The FFTed data is internally multiplied by:

scale / blocksize.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calc_data, e1432_set_window.

NAME

e1432_set_window − Set FFT window type
e1432_get_window − Get FFT window type

SYNOPSIS

```
SHORTSIZ16 e1432_set_window(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 window)
SHORTSIZ16 e1432_get_window(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 *window)
```

DESCRIPTION

e1432_set_window sets the FFT window type. This window is applied to input data as part of the FFT. The FFT can be turned on using *e1432_set_calc_data* with the **E1432_DATA_FREQ** or **E1432_DATA_ORDER** parameter. Windowing is used to reduce leakage effects caused by input frequency components that are not multiples of (effective_clock_freq)/(input blocksize).

e1432_get_window returns the current FFT window type.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

The window that is applied is already scaled for "narrow-band" measurements. For "wide-band" measurements, the user must multiply the data by a factor that depends on which window is used (see below).

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

window selects one of the following window types:

E1432_WINDOW_UNIFORM selects the Uniform window, which is equivalent to no windowing at all.

E1432_WINDOW_HANNING selects the Hann window, which is a simple raised cosine. This window has good frequency resolution and reasonably good side-lobe roll-off, but poor main-lobe flatness and relatively large side-lobe peaks. This window is often a good window to use when measuring broad-band noise. The data will be scaled for narrow-band measurements; to convert to wide-band, multiply the data by the square root of 2/3, which is approximately 0.81649658092772603.

E1432_WINDOW_FLATTOP selects a Flat-top window. This window is a good window to use when making amplitude or phase measurements of relatively pure tones. The maximum side-lobe level is about -95.1 dB, and the maximum main-lobe error is about plus-or-minus 0.00312487654556 dB. The data will be scaled for narrow-band measurements; to convert to wide-band, multiply the data by 0.51150334640807393.

There is a setting of **E1432_WINDOW_USER1**, but you shouldn't set it using this function, use *e1432_set_user_window* function instead, which will load your own window and then will automatically set the window type to **E1432_WINDOW_USER1**. Once you have loaded your USER window, changing window type with this function *e1432_set_window* will erase your USER window.

RESET VALUE

After a reset, *window* is set to **E1432_WINDOW_UNIFORM**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_calc_data, e1432_set_user_window

NAME

e1432_set_zoom - Set zoom state
e1432_get_zoom - Get zoom state

SYNOPSIS

```
SHORTSIZ16 e1432_set_zoom(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 state)
SHORTSIZ16 e1432_get_zoom(E1432ID hw, SHORTSIZ16 ID, SHORTSIZ16 *state)
```

DESCRIPTION

e1432_set_zoom sets the zoom state, of a module. If zoom is on, the inputs are digitally mixed (i.e. multiplied by) by a complex sine wave whose frequency is set by *e1432_set_center_freq*. This results in each time sample becoming a real/imaginary pair, doubling the amount of time data. When a FFT is preformed on this complex time data, the resulting spectrum is centered around the chosen center frequency. By choosing a small span with the *e1432_set_span* function, it is then possible to "zoom in" or increase FFT resolution around a narrow band: center frequency +/- span/2.

There are span, sampling frequency, and center frequency limitations in the zoom mode due to the extra processing required. These limitations are module dependent.

In the E1432 the max center frequency is limited to one fifth of the normal top span of the module for sampling frequencies of 51.2KHz and lower; i.e.

max center frequency = sampling clock frequency / (2.56 * 5)

For sampling frequencies above 51.2KHz the front ends of the E1432 modules divide down the clock to the ADC by a factor of two; so the max center frequency is also limited by an extra factor of two to prevent aliasing:

max center frequency = sampling clock frequency / (2.56 * 10)

For the default sampling frequency of 51.2Khz, the max center frequency frequency in zoom is limited to 4KHz with any energy above this frequency attenuated by the decimation filters. There are only four spans available in the zoomed mode instead of the eight available in the normal mode. The top span available io one half the max center frequency. Spans are centered on the center frequency and progress lower by a factor of four rather than the normal factor of two. For the default sampling frequency of 51.2Khz the available zoomed spans are 2KHz, 500Hz, 125Hz, and 31.25Hz.

In the E1433 the max sampling frequency as set by *e1432_set_clock_freq* is limited to 65536 Hz. The max center frequency is:

max center frequency = sampling clock frequency / 2.56

For the max sampling frequency of 65536 Hz, the max center frequency in zoom is thus limited to 25.6Khz with any energy above this limit attenuated by the decimation filters. The max span is limited to one half of the max center frequency, or 12.8KHz in the above case. There are 15 lower spans available, each a factor of 2 lower than the previous higher one.

When a FFT is being done on the zoomed data, the measurement blocksize is restricted to a minimum of 32 and a maximum of 4096, and must be a power of two. Energy in frequencies above the max center frequency is attenuated by the decimation filters. If the center frequency is lower than span/2, negative frequency points are included in the FFT. These frequency point only duplicate information available in the positive frequency range of a FFT. For these reasons it is customary to limit the center frequency to:

span/2 <= center frequency <= max center frequency - span/2 This limit will maximize the non-redundant, unattenuated frequency points in the FFT.

e1432_get_zoom returns the current value of the state, of a single channel or group of channels *ID*, into a memory location pointed to by *state*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

state can be either E1432_ZOOM_ON or E1432_ZOOM_OFF

RESET VALUE

After a reset state is set to E1432_ZOOM_OFF.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_center_freq, e1432_set_span, e1432_set_clock_freq

NAME

e1432_set_weighting − Set input weighting filter
e1432_get_weighting − Get current input weighting filter

SYNOPSIS

```
SHORTSIZ16 e1432_set_weighting(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 weighting)
SHORTSIZ16 e1432_get_weighting(E1432ID hw, SHORTSIZ16 ID,
                               SHORTSIZ16 *weighting)
```

DESCRIPTION

e1432_set_weighting sets the input weighting filter of a single channel or group of channels *ID*, to the value given in *weighting*.

e1432_get_weighting returns the current input weighting filter, of a single channel or group of channels *ID*, into a memory location pointed to by *weighting*.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

weighting must be one of the following: **E1432_WEIGHTING_OFF** to turn the weighting filter off, **E1432_WEIGHTING_A** for the A weighting filter, **E1432_WEIGHTING_B** for the B weighting filter, **E1432_WEIGHTING_C** for the C weighting filter.

Weighting filter settings other than **E1432_WEIGHTING_OFF** are only available at clock frequencies of 51200 and 65536 Hz.

Weighting filters are currently available only on the E1433. Yhey are applied by a time-domain filter, so they affect both time and frequency domain data from the E1433 module.

The weighting filters are applied prior to Peak and RMS computations. Therefore, the values returned by *e1432_get_current_value*, as well as the peak and rms value entries in the trailer (see *e1432_set_append_status*) have the weighting applied.

If this parameter is changed while a measurement is running, it will not have any effect until the start of the next measurement.

RESET VALUE

After a reset, the *weighting* is set to **E1432_WEIGHTING_OFF**.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_clock_freq, e1432_set_peak_mode, e1432_set_rms_mode, e1432_set_peak_decay_time, e1432_set_rms_avg_time, e1432_set_rms_decay_time, e1432_get_current_value, e1432_set_append_status.

NAME

e1432_set_xfer_size − Set local bus transfer size
e1432_get_xfer_size − Get current local bus transfer size

SYNOPSIS

```
SHORTSIZ16 e1432_set_xfer_size(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 xfer_size)
SHORTSIZ16 e1432_get_xfer_size(E1432ID hw, SHORTSIZ16 ID,
                               LONGSIZ32 *xfer_size)
```

DESCRIPTION

e1432_set_xfer_size sets the local bus transfer size, of a single channel or group of channels *ID*, to the value given in *xfer_size*.

e1432_get_xfer_size returns the current value of the local bus transfer size, of a single channel or group of channels *ID*, into a memory location pointed to by *xfer_size*.

When the data port is set to VME (see *e1432_set_data_port*), then *xfer_size* is not used. When the data port is set to local bus, or local bus eavesdrop, this *xfer_size* is the number of data points in each block sent to the local bus.

The default value for *xfer_size* is zero, which means to use the current *blocksize* instead.

If zoom is on (see *e1432_set_zoom*), each point is complex. This means the number of values sent over the local bus is twice the *xfer_size*, or twice the *blocksize* if *xfer_size* is zero.

This parameter is a "global" parameter. It applies to an entire E1432 module rather than to one of its channels. The *ID* parameter is used only to identify which module the function applies to, and all channels in that module will report the same value for this parameter.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel.

xfer_size selects the number of sample points in a block. The minimum legal value is 0 (which means to use the *blocksize* instead); the maximum depends on how much RAM is available and how many channels are active in a module. This parameter should not include the size of the appended status data, as defined in *e1432_set_append_status*.

When the data size is set to **E1432_DATA_SIZE_16**, the *xfer_size* will be rounded down to an even number (but a non-zero *xfer_size* of less than four will get rounded up to four).

RESET VALUE

After a reset, the measurement *xfer_size* is set to 0.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_set_append_status, e1432_set_data_port, e1432_set_data_size, e1432_get_xfer_size_limits, e1432_get_fifo_size_current_max, e1432_set_zoom

NAME

e1432_src_get_rev − Get revision information for source board hardware and rom

e1432_src_get_fwrev − Get revision information for file containing source board rom binary

SYNOPSIS

```
SHORTSIZ16 e1432_src_get_rev(E1432ID hw,SHORTSIZ16 src_chan,
                  LONGSIZ32 *romid, LONGSIZ32 *romdate, LONGSIZ32 *bdid,
                  char *bddate)
SHORTSIZ16 e1432_src_get_fwrev(LONGSIZ32 *dptr, LONGSIZ32 *fwid,
                               LONGSIZ32 *fwdate, LONGSIZ32 numwords)
```

DESCRIPTION

e1432_src_get_rev returns revision information for source board hardware and rom, of a single channel *src_chan*, into memory locations pointed to by *romid*,*romdate*,*bdid* and *bddate*.

e1432_src_get_fwrev returns revision information for source board firmware file loaded into an array, pointed to by *dptr*, of size *numwords*, into memory locations pointed to by *fwid* and *fwdate*.

romid (or *fwid*) is the revision id of firmware in the rom (or rom file) (one word).

romdate (or *fwdate*) is the revision date of firmware in the rom (or file), when displayed as hex (one word, yyyymmdd).

bdid is the revision id of the source board hardware (one word).

bddate is the revision date of the source board hardware (12 ascii characters).

src_chan is the ID of a single channel.

dptr points to the location the binary rom file is stored in memory.

numwords is the size of the file in 4 byte words.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_src_prog_romimage(3), srcutil(1)

NAME

e1432_src_prog_romimage − Programs source board flash rom.

SYNOPSIS

```
SHORTSIZ16
e1432_src_prog_romimage(E1432ID hw,SHORTSIZ16 src_chan, LONGSIZ32 *dptr,
                        LONGSIZ32 numwords)
```

DESCRIPTION

e1432_src_prog_romimage programs the flash rom associated with source channel *src_chan*, from binary file loaded into array, pointed to by *dptr*, of size *numwords*.

src_chan is the ID of a single channel.

dptr points to the location the binary rom file is stored in memory.

numwords is the size of the file in 4 byte words.

The binary file in the array is validated, the system area of the rom is erased, then it is programmed, 4096 words at a time. Prints to stdout describe what is going on.

CAUTION: Interrupting the erase and reprogramming of the source flash rom will create a currupt rom which is useless. A recover service procedure is in arbsrc/romfix.txt.

The cal factors and hardware revision information are not affected by this process.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_src_get_fwrev(3), e1432_src_get_rev(3), srcutil(1)

NAME

e1432_src_rxfr − Raw multiblock write to source

SYNOPSIS

```
SHORTSIZ16 e1432_src_rxfr(E1432ID hw, SHORTSIZ16 src_chan, LONGSIZ32 numwords,
                LONGSIZ32 *dptr, LONGSIZ32 mode)
```

DESCRIPTION

e1432_src_rxfr calls *e1432_write_srcbuffer_data*, with multiple transfer blocks if needed, to write *numwords* of *dptr* data to source, without any protocol wrapper.

src_chan is the ID of a single channel.

numwords is the size of the file in 4 byte words.

dptr points to the location in memory of the data to be transferred.

mode is the *e1432_write_srcbuffer_data* write mode.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_write_srcbuffer_data(3)

NAME

e1432_trace_level − Enable/disable call tracing printout

SYNOPSIS

```
void e1432_trace_level(SHORTSIZ16 level)
```

DESCRIPTION

Debugging is easier if you are able to trace the progression of calls made to and within the E1432 library. *e1432_trace_level* causes several different levels of call tracing to be displayed.

Level 0 is no tracing

Level 1 displays first level calls, meaning the major E1432_XXX type calls. Some E1432_XXX functions call other E1432_XXX functions. This becomes apparent once you start this type of debugging.

Level 2 and Level 3 display increasingly smaller functions and would probably not be of interest to most users of this the E1432 library.

level is an integer whose value is either **0** (no tracing), or a tracing level desired. **1** will probably do for most users.

RESET VALUE

After a reset, *trace_level* is set to **0**, resulting in no call tracing.

RETURN VALUE

This function does not return a value.

SEE ALSO

e1432_debug_level

NAME

e1432_trigger_measure − Manually trigger an E1432

SYNOPSIS

```
SHORTSIZ16 e1432_trigger_measure(E1432ID hw, SHORTSIZ16 ID,
                                 SHORTSIZ16 wait_after)
```

DESCRIPTION

e1432_trigger_measure moves all modules in the group from the **TRIGGER** state to the **MEASURE** state.

This function performs a "manual trigger", and does not need to be called when either one of the module in the group is set to "auto-trigger", or any of the channels in the group is set as an active trigger source. See the "Measurement setup and control" section earlier in this manual, for a detailed description of the measurement states.

This function waits for all modules to be in the **TRIGGER** state, before proceeding further, and returns an error if this state is not reached after a limited time. After the call to *e1432_trigger_measure* completes successfully, the measurement moves to the **CONVERT** state, and will stay here until a block is acquired.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

ID is either the ID of a group of channels that was obtained with a call to *e1432_create_channel_group*, or the ID of a single channel. If the measurement involves more than one module, it is mandatory that a *group ID* be used, rather than a *channel ID*.

wait_after determines whether this function will wait for the module to actually move beyond the **TRIGGER** state. If zero, the function does not wait; if non-zero, the function waits.

RESET VALUE

Not applicable.

RETURN VALUE

Return 0 if successful, a (negative) error number otherwise.

SEE ALSO

e1432_arm_measure, e1432_init_measure

NAME

e1432_write_srcbuffer_data − Write arbitrary data to source
e1432_check_src_arbrdy − Check data buffer full/empty status
e1432_get_src_arbstates − Get data buffer full/empty status

SYNOPSIS

```
SHORTSIZ16 e1432_write_srcbuffer_data(E1432ID hw, SHORTSIZ16 chanID,
                                      LONGSIZ32 *dataPtr,
                                      LONGSIZ32 numwords,
                                      SHORTSIZ16 mode)

SHORTSIZ16 e1432_check_src_arbrdy(E1432ID hw, SHORTSIZ16 chanID,
                                      SHORTSIZ16 mode);

SHORTSIZ16 e1432_get_src_arbstates(E1432ID hw, SHORTSIZ16  chanID,
        LONGSIZ32 *xfrbuf_wds, LONGSIZ32 *srcbuf_state_AB,
        LONGSIZ32 *srcbuf_state_A, LONGSIZ32 *srcbuf_state_B)
```

DESCRIPTION

e1432_write_srcbuffer_data transfers arbitrary data from the host to the module's transfer buffer where it is then transferred to the source board in the module. The source buffer internally consists of two buffers which alternate in continuous mode. Returns 0 if successful or a (negative) error number if an error.

NOTE: Under some circumstances, the data pointed to by *dataPtr* may be byte swapped after the call to *e1432_write_srcbuffer_data*. This may pose a problem for the application if it reuses that data.

hw must be the result of a successful call to *e1432_assign_channel_numbers*, and specifies the group of hardware to talk to.

chanID is a channel ID. Group IDs are not yet supported.

dataPtr points to the data to be transferred. Only the most significant 24 bits of this data are used, the bottom eight bits are completely ignored.

The data format must be 32bit signed fraction (2's complement), normalized to full scale. The maximum value is (2^31)-1 = 2147483647 = 0x7FFFFFFF = 32 bit fractional .999999999. The minimum value is -(2^31) = -2147483648 = 0x80000000 = 32 bit fractional -1.

Once normalized so that full scale equals 1, floating-point data can be converted as follows:

```
/* Floats typically have only 23 bits of mantissa, while doubles
   typically have 53 bits.  So use double to ensure that we
   preserve all 24 bits that the source will use. */

double data[DATALENGTH];  /* normalized data */
long arbdata[DATALENGTH]; /* data to be transferred */
double x;

for (i = 0; i < DATALENGTH; i++)
{
    x = data[i];
    if (x > 8388607.0 / 8388608.0)
        x = 8388607.0 / 8388608.0;
    else if (x < -1.0)
        x = -1.0;
    arbdata[i] = floor(8388608.0 * x + 0.5) << 8;
}
```

For a data value of 1, the output voltage is = *e1432_set_range* * *e1432_set_amp_scale*.

numwords is the size of this data array. Normally this must be less than or equal to **E1432_SRC_DATA_NUMWORDS_MAX**, which is 4096.

If all of these are true:

1. The module has no active inputs
2. The module has DRAM
3. The source channel is in source mode **E1432_SOURCE_MODE_ARB**

then *numwords* is not limited to **E1432_SRC_DATA_NUMWORDS_MAX**, and can be as large as the source data buffer size. For more information on the use of DRAM for source buffers, see the manual page for *e1432_set_srcbuffer_mode*.

mode determines the transfer mode and may have the following values:

In **E1432_SRC_DATA_MODE_WAITA**, **E1432_SRC_DATA_MODE_WAITB**, and **E1432_SRC_DATA_MODE_WAITAB** modes, *e1432_write_srcbuffer_data* does not return until the data is both transferred to the module's transfer buffer and from there to the source buffer (A, B, or A concatenated with B), if possible.

In **E1432_SRC_DATA_MODE_A**, **E1432_SRC_DATA_MODE_B**, and **E1432_SRC_DATA_MODE_AB** modes, *e1432_write_srcbuffer_data* returns as soon as data is transferred to the module's transfer buffer. Transfer to the source board occurs in the background. **E1432_SRC_DATA_MODE_AB** must be used for e1432_set_srcbuffer_mode set to **E1432_SRCBUFFER_CONTINUOUS**.

e1432_check_src_arbrdy should be used to determine when more data may be transferred.

e1432_check_src_arbrdy queries whether the module's transfer buffer is ready to accept more data. The returned value is set to 1 if more data may be transferred and 0 if there is no room currently available for more data to be transferred. If an error, it returns a (negative) error number.

e1432_get_src_arbstates can be used to determine which buffers data may be transferred to, for special

case buffer management. Generally this is not needed, as *e1432_check_src_arbrdy* is an easier way to determine if the source is ready for more data. This returns 0 if successful or a (negative) error number if not.

xfrbuf_wds points to the number of words remaining in the transfer buffer. **xfrbuf_wds* will be zero for a successfully completed data transfer.

srcbuf_state_A points to the state of the A buffer.

srcbuf_state_B points to the state of the B buffer.

srcbuf_state_AB points to the state of the buffer of A concatenated with B. If buffers A and B are in the same state, then that is the state of AB. If buffers and and B are in different states, then the state of AB is reported as **E1432_SRCBUF_RDY**.

These buffer states can have the following values:

E1432_SRCBUF_AVAIL, the buffer is ready to be started to accept data. This is the state immediately after using *e1432_set_srcbuffer_init*.

E1432_SRCBUF_RDY, the buffer is ready to continue to accept data. This is the state after sending at least some data to the source, but the source is not yet full.

E1432_SRCBUF_FULL, the buffer is full and not ready to accept data. When the srcbuffer mode is set to **E1432_SRCBUFFER_PERIODIC_A**, then the B buffer is not used, so the status of the B buffer will be **E1432_SRCBUF_FULL**.

The source must be set to **E1432_SOURCE_MODE_ARB** mode or **E1432_SOURCE_MODE_BARB** using *e1432_set_source_mode* in order for these functions to operate properly.

RESET VALUE

Not applicable.

RETURN VALUE

e1432_write_srcbuffer_data and *e1432_get_src_arbstates* return 0 if successful, or a (negative) error number otherwise.

e1432_check_src_arbrdy returns 1 if more data may be transferred, a 0 if there is no room currently available for more data to be transferred, or a (negative) error number.

SEE ALSO

e1432_set_range, e1432_set_amp_scale, e1432_set_srcbuffer_init, e1432_set_srcbuffer_mode, e1432_set_srcbuffer_size

NAME

e1432_bob − Description of E1432 break-out box options

DESCRIPTION

When a smart break-out box (BoB) is attached to an E1432 or E1433 module, the BoB provides additional input modes that are not available without a BoB. For example, a Charge BoB provides a charge input which allows us to use **E1432_INPUT_MODE_CHARGE**. A Mike BoB provides a microphone input which allows us to use **E1432_INPUT_MODE_MIC** or **E1432_INPUT_MODE_MIC_200V**.

E1432_INPUT_MODE_VOLT works the same whether a BoB is attached or not. All BoBs allow the module to program the low-side to be floating or grounded. (If no smart BoB is present, we accept the e1432_set_input_low function to program floating/grounded, but we don't actually do anything except remember the value.) All BoBs also provide a programmable ICP current source, allowing **E1432_INPUT_MODE_ICP** to be used.

For either **E1432_INPUT_MODE_VOLT** or **E1432_INPUT_MODE_ICP**, the ranges that are available are the same as when no BoB is attached: 0.1 to 20 volts for the E1432, and 0.005 to 10 volts for the E1433. The default range setting is 10 volts. The range setting is controlled by the *e1432_set_range* function.

For a Charge BoB, the charge input ranges are specified in picocoulombs, and are controlled by the *e1432_set_range_charge* function. The charge amp has several programmable gain settings, allowing a very wide set of range settings. For the E1432, the charge range settings available are 0.1 to 50000 picocoulombs. For the E1433, the charge range settings available are 0.005 to 50000 picocoulombs, although ranges below 0.1 are probably not very useful because of noise and offsets. The default charge range setting is 50000 picocoulombs. The Charge BoB also provides a 2 kHz low-pass filter which can be switched on and off, for the charge input only. This filter can be controlled with the *e1432_set_filter_freq* function.

For a Mike BoB, the microphone input measures volts, and is controlled by the *e1432_set_range_mike* function. The microphone input allows the input ranges to extend beyond the ranges that are normally available. The microphone ranges allowed are 0.01 to 50 volts for the E1432, and 0.0005 to 50 volts for the E1433. The default microphone range setting is 10 volts.

Both the Charge BoB and the Mike BoB have an option 402 version of the BoB, which provides an additional high-pass filter which can be enabled or disabled programmatically. This high-pass filter works only when the Charge BoB is in Charge mode, or the Mike BoB is in Microphone mode (**E1432_INPUT_MODE_MIC** or **E1432_INPUT_MODE_MIC_200V**). This high-pass filter is controlled by the *e1432_set_coupling* function, and is in addition to any other AC coupling on the input. The high-pass filter is a third order Butterworth filter. On the Charge BoB, the cutoff frequency is about 10 Hz. On the Microphone BoB, the cutoff frequency is about 22.4 Hz.

SEE ALSO

e1432_set_coupling, e1432_set_filter_freq, e1432_set_input_mode, e1432_set_range, e1432_set_range_charge, e1432_set_range_mike

NAME

e1432_e1431_diff − Compares the E1431 and E1432 Host Interface Libraries

DESCRIPTION

This manual page documents some of the basic differences between the E1431 and E1432 Host Interface libraries. This should be helpful for anyone porting an application that worked with the E1431.

Most functions in the E1431 library have a corresponding function in the E1432 library which performs the same action. However, all functions, variables, typedefs, and defines start with e1432 instead of e1431. Many functions were added to the E1432 library that have no corresponding function in the E1431 library.

Almost all E1432 functions have a first parameter of type E1432ID, which is not present in the E1431 functions, which is used to identify what set of hardware to talk to. This is in addition to the channel or group ID. See the manual page on e1432_assign_channel_numbers for more details.

In the E1431 library, the only type of channel that must be dealt with is an input channel. The channels are logically numbered 1 through M, where M is the number of input channels.

In the E1432 library, it is more complicated, since an E1432 can have different types of channels. The three types of channels currently available are input, source and tach/trigger channels. Channel numbers for each type of channel are numbered starting from one, so there will be input channels 1 through M, source channels 1 through N, and tach/trigger channels 1 through P, where M is the number of input channels, N is the number of source channels, and P is the number of tach/trigger channels. See the manual pages e1432_assign_channel_numbers, e1432_create_channel_group, and e1432_id for more details.

Error numbers start at -1300 for E1432, rather than -1200 for E1431, so that they don't overlap.

Error numbers are negative rather than positive. This actually works out the same because the E1431 library always returnes the negative of the error number.

Obviously, low-level register and register bit definitions change substantially. In particular, bits in the status register that are used to indicate the reason for an E1431 VME interrupt have been moved to a different register.

Changes to Functions

e1432_create_channel_group does not inactivate other channels within the modules that the channels are in. It also does not preset the channels in the new group. It also turns on only input channels, while source and tach channels are turned off.

e1432_set_auto_arm defaults to **E1432_AUTO_ARM**, rather than **E1432_MANUAL_ARM**.

e1432_set_auto_trigger defaults to **E1432_AUTO_TRIGGER**, rather than **E1432_MANUAL_TRIGGER**.

e1432_reset_lbus defaults to having the local bus NOT in reset, while E1431 defaults to having the local bus in reset.

e1432_set_try_recover defaults to not trapping bus errors.

e1432_read_raw_data, e1432_read_float32_data, and e1432_read_float64_data, all take an extra parameter called *which*.

e1432_check_overloads checks only active channels in a group, when given a group ID. e1431_check_overloads checks all channels in the group, whether active or not. e1432_check_overloads

replaces the E1431 "adc" parameter with a "half" parameter. This means that the E1432 function *can't* distinguish ADC overloads from other differential overloads, but it also means that the E1432 function *can* tell when a channel is underranged.

The E1431 has the ability to set input ground path to floating or grounded, using function *e1431_set_input_low*. The E1432 can't set this programmatically; instead this is set with a switch on the breakout box that connects to the inputs. So, there is no *e1432_set_input_low* function, and the *e1432_set_analog_input* function has one fewer parameters.

The E1431 has the ability to replay data through its digital filters. The E1432 does not have this ability, so all functions relating to this were removed: e1431_replay_data, e1431_replay_data_wanted, e1431_get_replay_data_size, e1431_set_replay_data_size, and e1431_init_replay.

SEE ALSO

e1432_assign_channel_numbers, e1432_check_overloads, e1432_create_channel_group, e1432_id(5), e1432_set_auto_arm, e1432_set_auto_trigger, e1432_set_try_recover, e1432_read_raw_data, e1432_read_float32_data, e1432_read_float64_data, e1432_set_analog_input

NAME

e1432_id − Description of channel and group IDs

DESCRIPTION

Most functions in the E1432 host interface library take an ID parameter which specifies what channel or group of channels the function should apply to. The ID can either be a channel ID or a group ID. If a group ID is used, then the function is applied to each channel contained in the group.

Channel IDs

Channel IDs are logical IDs which are created by a call to *e1432_assign_channel_numbers*. The *e1432_assign_channel_numbers* function must be called exactly once, following the call to *e1432_init_io_driver*, in order to declare to the library the logical addresses of the E1432 modules that will be used.

This function checks the existence of an E1432 module at each of the logical addresses given in a list of logical addresses, and allocates logical channel identifiers for each channel in all of the E1432s. Input channels, source channels, and tach/trigger channels are kept logically separated. Channel numbers for each type of channel are numbered starting from one, so there will be input channels 1 through M, source channels 1 through N, and tach/trigger channels 1 through P, where M is the number of input channels, N is the number of source channels, and P is the number of tach/trigger channels.

As an example, suppose two logical addresses 100 and 101 are passed to the function, and the logical address 100 has two 4-channel input SCAs and a 2-channel tach/trigger board, while logical address 101 has three 4-channel input SCAs and a 1-channel source board. In this case, input channel IDs 1 through 8 are assigned to the eight input channels at logical address 100, while input channel IDs 9 through 20 are assigned to the twelve input channels at logical address 101. Tach/trigger channel IDs number 1 and 2 are assigned to the two tach/trigger channels at logical address 100, and Source channel ID number 1 is assigned to the source channel at logical address 101.

To use the ID of an input channel, the input channel number is given as an argument to the **E1432_INPUT_CHAN(ID)** macro. (For backwards compatibility with the E1431, the macro currently does nothing.) To use the ID of a source channel, the source channel number is given as an argument to the **E1432_SOURCE_CHAN(ID)** macro. To use the ID of a tach/trigger channel, the tach/trigger channel number is given as an argument to the **E1432_TACH_CHAN(ID)** macro. A channel ID is always positive.

For example, to set the range of the third input channel to 10 volts, the source code would look something like:

```
status = e1432_set_range(hwid, E1432_INPUT_CHAN(3), 10.0);
```

Group IDs

Group IDs are logical IDs which are created by a call to *e1432_create_channel_group*. This function can be called multiple times to create multiple groups, and each group can contain any combination of channels, including mixtures of different types of channels. The channel groups can overlap as well.

This function creates and initializes a channel group. A channel group allows you to issue commands to several E1432 channels at once, simplifying system setup. The state of an individual E1432 channel that is in more than one channel group, is determined by the most recent operation performed on any group to which this channel belongs.

If successful, this function returns the ID of the group that was created, which is then used to reference the channel group in most other functions in this library. A group ID is always negative.

As a side effect, this function makes all input channels in the channel group active, and all source and tach channels in the channel group inactive. Unlike the E1431 library, this function does not inactivate other channels within the modules that the channels are in. Also unlike the E1431 library, this function does not preset the channels in the new group.

As an example, to create a group consisting of the first three input channels and the eighth and ninth input channels, the code would like something like this:

```
SHORTSIZ16 chan_list[5];
SHORTSIZ16 input_group;

chan_list[0] = E1432_INPUT_CHAN(1);
chan_list[1] = E1432_INPUT_CHAN(2);
chan_list[2] = E1432_INPUT_CHAN(3);
chan_list[3] = E1432_INPUT_CHAN(8);
chan_list[4] = E1432_INPUT_CHAN(9);
input_group = e1432_create_channel_group(hw, 5, chan_list);
```

To create a group consisting of the first two source channels, the code would look something like this:

```
SHORTSIZ16 chan_list[2];
SHORTSIZ16 source_group;

chan_list[0] = E1432_SOURCE_CHAN(1);
chan_list[1] = E1432_SOURCE_CHAN(2);
source_group = e1432_create_channel_group(hw, 2, chan_list);
```

Channel Parameters vs. Module Parameters

Some parameters, such as range or coupling, apply to specific channels. When a channel ID is given to a function that sets a channel-specific parameter, only that channel is set to the new value. When a group ID is given to a function that sets a channel-specific parameter, all channels in the group are set to the new value.

Some parameters, such as clock frequency or data transfer mode, apply globally to a module. When a channel ID is given to a function that applies to a whole module, the channel ID is used to determine which module. The parameter is then changed for that module. When a group ID is given to a function that applies to a whole module, the function is applied to each module that contains a channel in the group.

Starting and stopping a measurement is somewhat like setting a module-specific parameter. Starting a measurement starts each active channel in each module that has a channel in the group.

SEE ALSO

e1432_assign_channel_numbers, e1432_create_channel_group, e1432_parm(5)

NAME

e1432_inst − Installing the E1432 distribution

DESCRIPTION

Getting the Latest Distribution Using FTP

The E1432 distribution is shipped on a DAT tape with the E1432 module. This distribution includes the E1432 Host Interface library, associated examples, and manual pages.

The latest version of the E1432 distribution can be obtained via anonymous FTP, at:

`ftp://hpls01.lsid.hp.com/dsp/products/e1432/software/wk_sta/E1432.X.XX.XX.Z`

Revision Numbering

The revision string for each release is of the form *a.NN.NN*, where "a" is a letter and the "N"s are numbers. If the letter is an "A", the release is an official release. If the letter is an "X", the release is an interim release which may provide new features or fix bugs from the previous official release. For each release, the numerical part of the revision string increases.

For example, the first official release was **A.00.00**. The first interim release after that was **X.00.01**. The second interim release was **X.00.02**. Eventually, the features and fixes from the interim releases get rolled into an official release, whose label starts with "A".

For any revision, the revision number of the `sema.bin` or `lib1432.sl` file can be found by typing "`what sema.bin`" or "`what lib1432.sl`". (If the `what` command is not available, `ident` might work.) Both of these probably only work only on unix machines. The latest revision number is

Installing the Distribution

Each file in the FTP directory is a complete release, and will replace all files in the `/opt/e1432` and `/opt/vxipnp/hpux/hpe1432` directories. The files in the FTP directory are compressed "update" files. Similarly, the DAT tape shipped with an E1432 module contains an "update" file. If you are installing onto an HP-UX system, you would unpack one of these files by doing:

```
cd /                                # Start at root directory
uncompress E1432.X.XX.XX.Z          # Uncompress the update file
/etc/update -s /E1432.X.XX.XX '*'   # Unpack the update file
```

Alternatively, if you are not on HP-UX, you can use tar to unpack the update file (an "update" file is really just a tar file with a few extra files added). In this case, do this:

```
cd /                                # Start at root directory
uncompress E1432.X.XX.XX.Z          # Uncompress the update file
tar -xvf E1432.X.XX.XX              # Unpack the update file
rmdir E1432 E1432-VXIPNP            # Remove extraneous directories
rm -f /system/INDEX                 # Remove extraneous file
rm -f /system/INFO                  # Remove extraneous file
rm -f /system/CDFinfo               # Remove extraneous file
```

See the HP-UX reference manual for information about this command. You will have to be root to install the E1432 distribution.

The E1432 distribution is normally installed in the `/opt/e1432` and `/opt/vxipnp/hpux/hpe1432` directories. Under the `/opt/e1432` directory there are several subdirectories, containing include files (in

include), host interface libraries (in lib), utility programs (in bin), demo programs (in demo), example code (in example), manual pages (in man), and source code to the Host Interface library (in hostlib).

Older versions of the E1432 distribution (versions A.02.00 and older) were installed in /usr/e1432 rather than /opt/e1432. If you have one of these older distributions, it may be necessary for you to delete the old distribution by hand. In addition, if you have existing code that looks for the E1432 distribution in the old place, it may be necessary for you to create a symbolic link pointing to the new distribution, by doing this:

```
ln -s /opt/e1432 /usr/e1432
```

Printing the E1432 Function Reference

The E1432 distribution includes manual pages for the E1432 Host Interface library. These manual pages can be examined on-line, using the *ptman* command that is shipped in /opt/e1432/bin. For example, you can read the manual page for the *e1432_init_io_library* function by typing:

```
ptman e1432_init_io_library
```

The distribution also includes several nicely formatted sets of these manual pages, in various formats for different printers or viewers. All manuals come with an index at the end which can be a useful list of the functions that are available. All of these manuals except the PDF version are shipped compressed, to save space (the PDF format is not compressible). They can be uncompressed with the UNIX *uncompress* command. The formatted manual files that are available are:

man.pcl.Z — This is a PCL version of the manual, that can be printed on any PCL printer (such as an HP laserjet or HP deskjet printer). Typically, this manual can be printed by typing:

```
uncompress < /opt/e1432/man/man.pcl.Z | lp -oraw
```

man.pdf — This is a PDF version of the manual. This file may not be present in all distributions.

man.ps.Z — This is a postscript version of the manual, that can be printed on any postscript printer. Typically, this manual can be printed by typing:

```
uncompress < /opt/e1432/man/man.ps.Z | lp -opostscript
```

man.txt.Z — This is a plain text version of the manual. While not as nice as the postscript or PCL manuals, it can be printed on any line printer. However, this manual assumes 66 lines per page, which can be annoying if your printer is configured for only 60 lines per page.

SEE ALSO

ptman(1)

NAME

e1432_intr − Description of E1432 interrupt behavior

DESCRIPTION

E1432 Interrupt Setup

The E1432 VXI module can be programmed to interrupt a host computer using the VME interrupt lines. VME provides seven such lines, and the E1432 module can be told to use any one of them (see *e1432_set_interrupt_priority*).

The E1432 can interrupt the host computer in response to different events. The user can specify a *mask* of events on which to interrupt. This mask is created by ORing together the various conditions that the user wants. The following table, copied from the *e1432_set_interrupt_mask* manual page, shows the conditions that can cause an interrupt:

Interrupt Mask Bit Definitions	
Define (in e1432.h)	Description
E1432_IRQ_BLOCK_READY	Scan of data ready in FIFO
E1432_IRQ_MEAS_ERROR	FIFO overflow
E1432_IRQ_MEAS_STATE_CHANGE	Measurment state machine changed state
E1432_IRQ_MEAS_WARNING	Measurement warning
E1432_IRQ_OVERLOAD_CHANGE	Overload/underrange status changed
E1432_IRQ_SRC_STATUS	Source channel interrupt
E1432_IRQ_TACHS_AVAIL	Raw tach times ready for transfer to other modules
E1432_IRQ_TRIGGER	Trigger ready for transfer to other modules

The **E1432_IRQ_SRC_STATUS** interrupt is used for source channel overload, overread, and shutdown. When the source is in arb data mode, this interrupt is also used for the "ready for arb data" interrupt.

The **E1432_IRQ_MEAS_ERROR** currently is used only for a FIFO overflow. This will interrupt as soon as the FIFO overflows, but note that the FIFO still has useful data in it which can still be read by the *e1432_read_xxx_data* functions. *e1432_block_available* will not indicate that a FIFO overflow has occurred until all of the remaining data is read out of the FIFO.

E1432 Interrupt Handling

To make the E1432 module do the interrupt, both a *mask* and a *VME Interrupt line* must be specified, by calling *e1432_set_interrupt_mask* and *e1432_set_interrupt_priority* respectively. Once the mask and line have been set, and an interrupt occurs, the cause of the interrupt can be obtained by reading the **E1432_IRQ_STATUS2_REG** register (using *e1432_read_register*). The bit positions of the interrupt mask and status registers match so the defines can be used to set and check IRQ bits.

Once it has done this interrupt, the module will not do any more VME interrupts until re-enabled with *e1432_reenable_interrupt*. Normally, the last thing a host computer's interrupt handler should do is call *e1432_reenable_interrupt*.

Events that would have caused an interrupt, but which are blocked because *e1432_reenable_interrupt* has not yet been called, will be saved. After *e1432_reenable_interrupt* is called, these saved events will cause an interrupt, so that there is no way for the host to "miss" an interrupt. However, the module will only do one VME interrupt for all of the saved events, so that the host computer will not get flooded with too many interrupts.

For things like "E1432_IRQ_BLOCK_READY", which are not events but are actually states, the module will do an interrupt after *e1432_reenable_interrupt* only if the state is still present. This allows the host

computer's interrupt handler to potentially read multiple scans from an E1432 module, and not get flooded with block ready interrupts after the fact.

Host Interrupt Setup

The E1432 Host Interface library normally uses the SICL I/O library to communicate with the E1432 hardware. To receive VME interrupts, a variety of SICL setup calls must be made. The "examples" directory of the E1432 distribution contains an example of setting up SICL to receive interrupts from an E1432 module.

This is a summary of how to set up SICL to receive an E1432 interrupt:

1. Query SICL for which VME interrupt lines are available, using *ivxibusstatus* and *ivxirminfo*.
2. Tell the E1432 module to use the VME interrupt line found in step one, using *e1432_set_interrupt_priority*.
3. Set up an interrupt handler routine, using *ionintr* and *isetintr*. The interrupt handler routine will get called when the interrupt occurs.
4. Set up interrupt mask in the E1432 module, using *e1432_set_interrupt_mask*.

Host Interrupt Handling

When the E1432 asserts the VME interrupt line, SICL will cause the specified interrupt handler to get called. Typically the interrupt handler routine will read data from the module, and then re-enable E1432 interrupts with *e1432_reenable_interrupt*. The call to *e1432_reenable_interrupt* must be done unless the host is not interested in any more interrupts.

Inside the interrupt handler, almost any E1432 Host Interface library function can be called. This works because the Host Interface library disables interrupts around critical sections of code, ensuring that communication with the E1432 module stays consistent. Things that are *not* valid in the handler are:

1. Calling *e1432_delete_channel_group* to delete a group that is simultaneously being used by non-interrupt-handler code.
2. Calling one of the read data functions (*e1432_read_raw_data*, *e1432_read_float32_data*, or *e1432_read_float64_data*), if the non-interrupt-handler code is also calling one of these functions.
3. Calling *e1432_assign_channel_numbers* to reset the list of channels that are available to the E1432 library.

As is always the case with interrupt handlers, it is easy to introduce bugs into your program, and generally hard to track down these bugs. Be careful when writing this function.

SEE ALSO

e1432_set_interrupt_priority, e1432_set_interrupt_mask, e1432_reenable_interrupt

NAME

e1432_multimain − Multiple mainframe information

DESCRIPTION

E1482B MXI cards can be used to connect multiple VXI mainframes together, so that modules in both mainframes appear to be in one large logical mainframe. This can be done either with an embedded controller (such as a V/743), or with a MXI cable connected to a host computer.

However, this multi-mainframe setup can be complicated, and it places some restrictions on what measurements can be performed. Before delving into multi-mainframe measurements, you should consider whether it would be sufficient to simply run separate measurements in two separate mainframes.

There are two separate parts to getting a multi-mainframe measurement working with E143x modules. The first part is getting the VXI/MXI configuration into a valid state, which involves setting lots of switches on the MXI cards, and getting the logical addresses of all the modules into the correct ranges. Getting this correct is mostly independent of the E143x cards, other than just ensuring that the E143x cards have the appropriate logical addresses for the mainframe that they are in.

The second part is getting the E143x modules set up correctly. Mostly this involves writing a multi-mainframe-aware host application, but it also involves several SICL configuration files.

VXI/MXI Configuration for Multi-Mainframe Measurements

Before even worrying about E143x configuration and setup, the VXI/MXI system must be configured properly.

E1482B Revisions

Very old E1482B MXI cards have a problem with multi-mainframe configurations. The MXI card in the root mainframe must be revision G2 or later, and the other MXI cards must be revision G or later. The revision can be found on a metallic label on the ‘‘bottom’’ side of the MXI card. If your MXI cards are too old, you may get occasional bus errors in your application. A bus error will typically cause the application to crash, but will not crash the operating system of the host computer. As the number of E143x modules gets larger, your chances for bus errors increase because more data transfers will take place during a measurement.

E1482B Switch Settings

There are a large number of jumpers and switches on an E1482B MXI card. It is important get these all set correctly. Incorrect switch settings can cause anything from the MXI card failing to respond, to bus errors in an application, to module hangs when talking to modules in that mainframe. I believe it is even possible to damage the MXI card or the VXI mainframe backplane if the S1 and S8 switches (which control VXI Slot 0 operation) are set incorrectly.

The MXI card switch settings are well documented in the E1482B manual. This manual has nice pictures of what the switch settings should be, both for the ‘‘root’’ mainframe and the ‘‘non-root’’ mainframes, and for either an embedded controller or a host computer connected by MXI.

The next few paragraphs attempts to document most of these switch settings, but it’s generally a lot easier and less error-prone to use the pictures in the E1482B manual.

The MXI card is the slot 0 controller for all non-root mainframes. The MXI card is also the slot 0 controller for the root mainframe if an external computer is used. The default shipped configuration for the MXI card is mostly correct for these mainframes where the MXI card is the slot 0 controller. There are two things that must change:

* the logical address (see separate section below)
* possibly the VME and INTX terminators

The six VME and four INTX terminators need to be removed for MXI modules that are not at the ends of the daisy chain. For systems that use an embedded controller, this would be needed only when there are three or more mainframes (with two mainframes, both MXI cards are at the ends of the daisy chain). For systems that use MXI to connect to an external controller, the "daisy chain" includes the external controller itself, so this is an issue when there are two or more mainframes.

If an embedded controller is used in the root mainframe, then the MXI module is **not** the slot 0 controller for the root mainframe. This requires several switch settings to be changed. Table 2-1 ‘‘Configuration Settings’’ in the E1482B manual lists the changes from the default settings:

* the logical address (see separate section below)
* not the slot-0 controller (S1, S8)
* MXI System Controller enabled (S4)
* set the VME timeout to 200 usec (W6)
* set VME BTO chain position to 1 extender, non-slot0 (W7)
* set MXI bus timeout to 100 usec (W8)
* do not source CLK10 (W9, W10)

Logical Addresses

The logical address windows required by MXI are explained in the E1482B manual on pages 2-34 through 2-36. The next few paragraphs attempt to summarize those rules, but you may want to refer to the E1482B manual as well. The rules apply to all modules in a mainframe.

Each mainframe gets a window of logical addresses. No windows can overlap. The window size (number of possible logical addresses in a mainframe) must be a power of two. The starting address of the window must be zero, or an address which is a multiple of the window size.

Logical address zero is taken by the system controller, which is either an embedded controller or a host computer connected by MXI cable. If it is a host computer connected by a MXI cable, then the logical address windows of all mainframes must not include logical address zero (since zero is in the host computer, not in any mainframe).

For very-large-channel-count systems, it is best to allocate 32 logical addresses to each mainframe. The first mainframe would have logical addresses 0-31 (or 16-31 if there is an external computer at logical address 0). The second mainframe would have logical addresses 32-63, the third would have 64-95, the fourth 96-127, and so on. This ensures that each mainframe has enough logical addresses, and that the system will not run out of logical addresses too quickly.

The logical address of the MXI card itself need not be within the block of 32 addresses assigned to a mainframe. One good method is to put the first MXI card a logical address 1, the next MXI card at logical address 2, and so on.

Dynamic Configuration of Logical Addresses

The MXI cards do not support dynamic configuration, and so must be set to fixed logical addresses. Typically the addresses chosen would be 1 for the first MXI card, 2 for the second, and so on.

The E1432 card does support dynamic configuration. Dynamic configuration means that you put the E1432 cards all at logical address 255, and the system resource manager automatically figures out an appropriate logical address and tells the module to use that.

For small systems, dynamic configuration is probably not very useful. For large systems, dynamic configuration is quite useful.

There are limits to dynamic configuration. Typically, dynamic configuration does not work on the first card in a mainframe, but works on all the other cards. The first card must be set to a fixed appropriate logical address for that mainframe. The other cards in the mainframe will typically get assigned logical addresses increasing from the address of the first card.

E1482B Connections

If more than two mainframes are needed, daisy-chain them all. The first mainframe is the root mainframe. Each mainframe after the first is a non-root mainframe.

The MXI cables themselves are directional. One end of the cable is enclosed and has only single connectors, while the other end of the cable is open and has dual connectors. The enclosed end must be nearest to the root mainframe.

The E1482B manual mentions a ‘‘Level 2 Extender’’ configuration when using four or more mainframes. Do not use that setup. Use a simple daisy-chain setup.

Embedded Controller VME Bus Timeout

This applies only to systems using embedded controllers. Certain embedded controllers try to detect VME bus timeout, and this conflicts with the MXI cards which also try to detect VME bus timeout. These embedded controllers must be disabled from detecting the VME bus timeout, typically by adjusting a jumper or switch on the back of the module.

This is an issue for the E1405B Command Module, the E1406A Command Module, the E1499 V/382, and the RADI-EPC7 embedded controllers. The E1482B manual shows how to adjust these modules to not detect VME bus timeout. However, note that the E1405B and E1499 are not supported with the E143x module, and the E1406B and RADI-EPS7 are not recommended by HP.

This is not an issue for the V/743 embedded controller, because the V/743 VME bus timeout is much longer than the timeout used by the MXI card.

SICL Versions

SICL is an I/O library used on HP-UX. This low-level I/O library is used by the E143x host interface library when communicating with the VXI system.

Systems that use the E143x Plug&Play libraries do not directly use SICL, and instead use the VISA library defined by the Plug&Play standard. However, on HP-UX the VISA library internally uses SICL, so in the end SICL is used on all HP-UX systems.

Multi-mainframe systems generally involve a larger number of E143x modules than single-mainframe systems. This larger number of modules has a tendency to find problems in the SICL memory mapping code that runs in the HP-UX machine. It is important to use the latest version of SICL, which is somewhat less error-prone. As of July 1997, the latest officially released versions are C_03_09 for HP-UX 9.05, and E.01.01 for HP-UX 10.20. E.01.01 on 10.20 seems to be more reliable than C_03_09 on 9.05. There is also a pre-release unsupported E.01.04 version of SICL for 10.20 available at `ftp://hpls01.lsid.hp.com/dsp/temp/libsicl.sl`.

VXI and SICL Limits

The A24 memory space holds a total of 16 MBytes of memory, though sometimes half of that memory space is wasted due to limits on what the E1482B MXI card can map into the host computer.

The E143x modules each use 256 KBytes of A24 memory space. Assuming 16 MBytes of A24 memory is usable, that limits the number of E143x modules to **64**. E1432 modules with serial number prefixes below 3647 use 1 MByte of A24 space instead of 256 KBytes, so only 16, and sometimes only 8, of these older modules would work. These older E1432 modules can be upgraded in the field by HP to use only 256 KBytes of A24 space.

The limit of 64 E143x modules (including E1434) is an absolute maximum for a VXI system. The limit will be smaller if A24 space is used by other modules. Other modules that use A24 space include the E1562 SCSI Disk module and the E1485 Signal Processor module.

The E1482B MXI card allows a daisychain of at most eight devices. If an external computer is used, it counts as one device, so at most seven VXI mainframes can be used in this case. Also, the total cable length of a MXI daisychain can be at most 20 meters.

For systems that use a very large number of E143x modules, the E1482B MXI card can place additional limitations on how many E143x modules can be present in each mainframe. The E1482B MXI card limits the amount of A24 memory used by each mainframe to a power of two bytes. This typically means that you must use exactly a total of eight E143x (or E1562) modules in each mainframe, or there will be additional A24 memory allocated to the mainframe that is wasted. This waste is not a problem unless you are building a system that needs most of the total A24 memory space.

For versions X.03.15 and earlier of the non-Plug&Play E143x host interface library, there was an additional restriction. When using MXI to connect to an *external computer*, the the maximum number of E143x modules was only 8. This was due to limitations in the SICL I/O library, which have been worked around in versions X.03.16 and later.

For users of the Plug&Play E143x library, using MXI to connect to an external computer, the number of E143x modules seems to be limited to 8. The limit can be increased to around 13 E143x modules, but only when using at least pre-release version E.01.04 of SICL and VISA on HP-UX 10.20.

As of August 1997, this E.01.04 version of SICL and VISA is not officially released, but a patched version is available at `ftp://hpls01.lsid.hp.com/dsp/temp/libsicl.sl` and `ftp://hpls01.lsid.hp.com/dsp/temp/libvisa.sl`. With these patched versions of SICL and VISA, the limit is around 13 E143x modules.

Convincing SICL to use all of A24 Space

The default configuration of SICL on HP-UX will not even try to use all of A24 space. Because of this, the maximum number of E143x modules in a system can be severely restricted unless the configuration is changed.

When using MXI to connect to an external computer, this limitation will prevent using more than 24 E143x modules. If any E1432 modules are old enough that they use 1MB of A24 space, then this limitation will prevent the use of more than 4 E1432 modules.

To fix this, and convince SICL to use all of A24 space, two lines in the iproc.cf configuration file must be modified. On HP-UX 9.05, this file is at /usr/pil/etc/iproc.cf. On HP-UX 10.20, this file is at /etc/opt/sicl/iproc.cf.

Change:

```
boot ivxirm -p -I vxi
```

to:

```
boot ivxirm -p -M -I vxi
```

and change:

```
sysreset vxi ivxirm -t &
```

to:

```
sysreset vxi ivxirm -M -t &
```

After making this change, the iproc daemon must be killed and restarted.

E143x Configuration for Multi-Mainframe Measurements

Once the VXI/MXI system is properly configured, E143x modules must be configured and set up correctly.

Old E143x Modules

Older E1432 (but not E1433 or E1434) modules have a bug in the VXI interface which shows itself in one specific configuration. The configuration with the problem is a multi-mainframe system where an embedded V/743 controls the first mainframe, and MXI cables connect the first mainframe to subsequent mainframes. Data read from E1432 modules in the non-root mainframes is occasionally corrupted.

HP LSD has a new VXI interface ROM that can be installed in old E1432 modules which fixes this problem.

In addition, older E1432, E1433, and E1434 modules have a different bug in the VXI interface which can cause problems with the G5 version of the E1482B MXI card. This bug is generally quite rare, but can cause "Data Handshake Timeout" or errors due to incorrect data transfer to the host computer.

HP LSD has a new VXI interface ROM that can be installed in older E143x modules which fixes this problem.

VXI TTLTRG direction

The VXI bus provides eight TTLTRG lines, which are open collector lines connected to all modules in a mainframe. The E143x modules use two of these eight lines, one for a common sample clock, and the other for a common sync and trigger line. This allows multiple E143x modules to sample simultaneously, and allow trigger events in one E143x module to get propagated to all modules.

The E1482B MXI cards connect the VXI TTLTRG lines between VXI mainframes in only one direction at a time. Either direction is supported by MXI, and either direction can be made to work with the E143x modules. However, we only document one direction, and we encourage everyone to use this one direction, in order to make things less confusing and to simplify support of multi-mainframe systems.

We always document that the root mainframe must generate the TTLTRG lines, and all non-root mainframes must therefore receive the TTLTRG lines.

Because the TTLTRG lines are driven in only one direction, there are limits on what kinds of measurements

can be done with a multi-mainframe system.

We refer to the mainframe generating TTLTRG as the ‘‘master’’ mainframe, and all other mainframes as ‘‘slave’’ mainframes. If you follow our default of having the root mainframe generate TTLTRG, then the root mainframe is the same as the master mainframe.

Controlling TTLTRG Direction

The direction of the TTLTRG lines is controlled by two SICL configuration files. On HP-UX 9.05, these files are `/usr/pil/etc/vxi16/ttltrig.cf` and `/usr/pil/etc/vxi16/oride.cf`. On HP-UX 10.20, the files are `/etc/opt/sicl/vxi16/oride.cf` and `/etc/opt/sicl/vxi16/ttltrig.cf`.

If the `oride.cf` file is used, it overrides the settings in `ttltrig.cf`. Because the settings in `oride.cf` are less supported and harder to explain, we recommend that `oride.cf` **not** be used for setting the TTLTRG direction.

Instead, use `ttltrig.cf` to set the TTLTRG direction. This file is a little easier to understand - you specify a logical address for each TTLTRG line, and the mainframe that contains that logical address will be the one that generates TTLTRG.

The default for `ttltrig.cf` is that logical address 0 generates TTLTRG. For embedded controller systems this works without modification, because the embedded controller is in the root mainframe and we want the root mainframe to generate the TTLTRG lines. For systems with an external controller, the logical addresses in `ttltrig.cf` should be changed to the address of any module in the root mainframe.

Internal E143x TTLTRG driver

When you call *e1432_create_channel_group*, the last channel in this list becomes a TTLTRG driver for the group of modules. This TTLTRG driver is purely an internal thing, not something the user needs to worry about, and normally is hidden from the user.

However, you have to make sure that the TTLTRG driver ends up in the master mainframe so it can successfully drive the TTLTRG line for all modules in all mainframes.

For this reason, when you call *e1432_assign_channel_numbers*, you should list the logical addresses in reverse order. That way, the highest channel ID (which will become the TTLTRG driver when you do *e1432_create_channel_group*) will end up in the master mainframe.

This is opposite of what people intuitively try to do. It means that higher-numbered channels are in the root mainframe and lower-numbered channels are in the non-root mainframe. However, this setup does work and is the one that is tested by HP.

Measurement Arm

Because of synchronization problems caused by the one-directional nature of the TTLTRG lines, you must use manual arm, not auto arm or RPM arm, for multi-mainframe measurements. By manual arm, we mean that the host computer must do the arm using *e1432_arm_measure* (or equivalent, see ‘‘Measurement Trigger’’ below).

Although we haven’t tested it, the *e1432_set_mmf_delay* function should help work around this limitation to manual arm, if auto arm or RPM arm really are necessary for a particular multi-mainframe application.

Measurement Trigger

If you use manual trigger in addition to manual arm, you can simply use *e1432_arm_measure* for the arm and *e1432_trigger_measure* for the trigger.

However, if you want some other trigger (external trigger, input trigger, source trigger, tach trigger), then you must use the special multi-mainframe versions of *e1432_arm_measure*. Instead of just using *e1432_arm_measure*, you must call:

```
e1432_arm_measure_master_setup(hw, master_id);
e1432_arm_measure(hw, global_id, 0);
e1432_arm_measure_slave_finish(hw, slave_id);
e1432_arm_measure_master_finish(hw, master_id);
```

The manual pages for these function goes into a little more detail about the IDs that should be used.

Only E143x modules in the master mainframe can trigger a multi-mainframe E143x measurement. This is true for any trigger mode except manual trigger, where the trigger does not come from a module at all.

Phase Performance

Phase specs will be degraded by the delay that the MXI cables add to the sample clock. A customer-written calibration to correct this is certainly possible, but we have not tested this. The delay may be insignificant for many low-frequency applications, since the phase error scales with frequency.

A system with the two MXI cards, and a 1 Meter cable, shows a 75 nsec sample clock delay to the non-root mainframe. This corresponds to a -0.69 degree phase error at 25.6 kHz.

A 4 Meter cable adds about 17 nsec more delay, for a total of 92 nsec clock delay in the non-root mainframe. This corresponds to a -0.85 degree phase error at 25.6 kHz.

The MXI cable adds about 1.7 nsec per foot of cable.

Each daisy-chained mainframe adds another increment of delay, but only for the additional cabling length.

In theory, it should be possible to use the SMB Trig In input on the MXI card with an external clock source to drive all the mainframes simultaneously. This should eliminate most of the mainframe-to-mainframe phase errors. This might be useful when better phase accuracy is desired, but the application does not want to deal with mainframe-to-mainframe calibration and correction.

For more information about using the MXI Trig In connector, see the end of the manual page for *e1432_set_clock_source*, and the manual page for *e1432_set_auto_group_meas*.

Local Bus

The VXI local bus does not cross the MXI cable. The local bus only connects adjacent VXI modules. It is possible to set up measurements that use the local bus, but each mainframe will need to have its own module that reads the E143x local bus data (such as an E1562 disk module).

SEE ALSO

e1432_init_measure, e1432_arm_measure, e1432_arm_measure_master_finish, e1432_set_mmf_delay, e1432_trigger_measure

NAME

e1432_octave − Overview of E1432 Octave functionality

DESCRIPTION

1D1 Option

This option must be installed in a particular module in order for Octave measurements to run in it. Without this option, Octave parameters can still be set up but the *e1432_init_measure*() call will fail with error -1621, "Required option not installed".

The 1D1 option will normally be ordered when a module is ordered from the factory. However, provisions have been made to allow users to upgrade their units in the field. To do this, the user must supply the serial number of the module which is to receive the upgrade. The *progopt* program can be used to obtain the exact serial string, by running "progopt -S". The "-L" option may be necessary if the module is not located at the default logical address. The *progopt* program is also used to install the option, which is in the form of a codeword from the factory, by running "progopt -A 12345678", where "12345678" represents the codeword. Running "progopt -R" will verify that the codeword was written. Yet to be implemented is the process of ordering the 1D1 option upgrade and receiving the codeword. For now, this will be handled on a case by case basis.

Octave API functions

e1432_set_octave_mode(), *e1432_get_octave_mode*()
E1432_OCTAVE_MODE_OFF No Octave processing
E1432_OCTAVE_MODE_FULL Full Octave processing
E1432_OCTAVE_MODE_THIRD Third Octave processing

e1432_set_octave_avg_mode(), *e1432_get_octave_avg_mode*()
E1432_OCTAVE_AVG_MODE_EXP Exponential averaging
E1432_OCTAVE_AVG_MODE_LIN Linear averaging

e1432_set_octave_hold_mode(), *e1432_get_octave_hold_mode*()
E1432_OCTAVE_HOLD_MODE_OFF No hold mode
E1432_OCTAVE_HOLD_MODE_MAX Maximum value held
E1432_OCTAVE_HOLD_MODE_MIN Minimum value held

e1432_set_octave_start_freq(), *e1432_get_octave_start_freq*(),
e1432_set_octave_stop_freq(), *e1432_get_octave_stop_freq*()
Start/stop frequency band
octave_start_freq: 3.15 Hz minimum
octave_stop_freq: 20 KHz maximum

e1432_set_octave_int_time(), *e1432_get_octave_int_time*()
Integration/average time for Linear averaging.
octave_int_time: .00953125 to 10,000 seconds

e1432_set_octave_time_const(), *e1432_get_octave_time_const*()
Time constant time step for Exponential averaging.
octave_time_const: .0078125 to 1.0 seconds

e1432_set_octave_time_step(), *e1432_get_octave_time_step*()
Update time step/interval.
octave_time_step: .00953125 to 10,000 seconds

e1432_octave_ctl()

E1432_OCTAVE_CTL_STOP Stop/Pause Octave processing
E1432_OCTAVE_CTL_RESTART Restart Octave processing
E1432_OCTAVE_CTL_CONTINUE Continue paused Octave average

e1432_get_octave_blocksize()
The number of Octave samples per block is returned.

e1432_get_current_data()
Gets the most recent Octave data.

Enhanced existing functions

e1432_set_enable()
Enable type **E1432_ENABLE_TYPE_OCTAVE** added.

e1432_read_float64_data(), *e1432_read_float32_data*()
Data type **E1432_OCTAVE_DATA** added.

API considerations

There are a several constraints and/or unavailable features that exist when Octave processing is active.

Most trigger types are supported: auto, external, tachometer, source, and RPM arm. However, input triggering is not supported. Also, trigger delays (other than 0) are not supported.

Decimation filtering and associated features (such as order tracking) are unavailable when Octave processing is active.

The clock frequency, as set by *e1432_set_clock_freq*, must be 65536.

Peak and RMS values are available with Octave processing, without the need to turn on Peak/RMS processing. However, only "filtered", and not "block" processing is done.

Octave data is in mean-squared form and scaled as such. A square root must be taken to convert it into RMS data.

Because Octave processing is not independent from the time data processing, the time parameters, such as blocksize, may still determine the update rate, if they result in an update rate longer than the Octave update rate. Note that this is true, even, when *e1432_set_enable*() is used to turn **E1432_ENABLE_TYPE_TIME** data off.

It may be nececessary to turn **E1432_ENABLE_TYPE_TIME** data off for long Octave update rates. See the documentation for *e1432_set_octave_int_time*(), *e1432_set_octave_time_step*().

Octave data is not currently supported in "eavesdrop" mode (while time data is being throughput to the local bus) but the *e1432_get_current_data*() function may be used as an alternative.

Octave currently works with all types of data_mode, as set by *e1432_set_data_mode*, and data_size, as set by *e1432_set_data_size*.

When reading multiple data types with *e1432_read_float64_data*, etc, E1432_OCTAVE_DATA is currently the last in the order of data to be read.

Octave downloadable

Currently, the Octave downloadable is in the form of a file named soct.bin and it is kept in the same directory as the substrate downloadable, sema.bin. It does not replace sema.bin, but rather enhances functionality to include octave measurements. It is automatically downloaded when octave measurements are selected, using the *e1432_set_octave_mode*(). See the documentation for these functions for more details. Normally, a user has no need to be concerned with this file.

SEE ALSO

progopt(1), e1432_set_octave_mode(3), e1432_set_octave_avg_mode(3), e1432_set_octave_hold_mode(3), e1432_set_octave_start_freq(3), e1432_set_octave_stop_freq(3), e1432_set_octave_int_time(3), e1432_set_octave_time_const(3), e1432_set_octave_time_step(3), e1432_octave_ctl(3), e1432_get_octave_blocksize(3), e1432_get_current_data(3), e1432_set_trigger_delay(3)

NAME

e1432_parm − Description of E1432 parameters

DESCRIPTION

Some parameters, such as range or coupling, apply to specific channels. When a channel ID is given to a function that sets a channel-specific parameter, only that channel is set to the new value.

Some parameters, such as clock frequency or data transfer mode, apply globally to a module. When a channel ID is used to change a parameter that applies to a whole module, the channel ID is used to determine which module. The parameter is then changed for that module.

Starting and stopping a measurement is somewhat like setting a global parameter. Starting a measurement starts each active channel in each module that has a channel in the group.

After firmware is installed, and after a call to e1432_preset, all of the parameters (both channel-specific and global) in an E1432 module are set to their default values. For channel-specific parameters, the default value may depend on the type of channel. Some channel-specific parameters apply only to a specific type of channel. For example, tach holdoff applies only to tach channels. Setting such a parameter for a channel that doesn't make sense will result in an error.

At the start of a measurement, the E1432 firmware sets up all hardware parameters, and ensures that the input hardware is settled before starting to take data. The firmware also ensures that any digital filters have time to settle. This ensures that all data read from the module will be valid.

However, after a measurement starts, E1432 parameters can still be changed. The effect of this change varies, depending on the parameter. For some parameters, changing the value immediately aborts the measurement. For other parameters, the measurement is not aborted, but the changed parameter value is saved and not used until a new measurement is started. For still other parameters, the parameter change takes place immediately, and the data coming from the module may contain glitches or other effects from changing the parameter.

At this time, there is no way to tell the module to wait for settling when changing a parameter in the middle of a measurement. The only way to wait for settling is to stop and re-start the measurement. At this time, there is no way to disable the settling that takes place at the start of a measurement.

This section shows which parameters are global parameters, which are channel-specific, and what types of channels the channel-specific parameters apply to. Default values are shown for all of these parameters. In addition, each parameter is categorized as "abort", "wait", "immediate", or "glitch" depending on the behavior when this parameter is changed during a running measurement. Those with "abort" cause the measurement to abort. Those with "wait" don't take effect until the start of the next measurement. Those with "immediate" take effect immediately. Those with "glitch" take effect immediately, and may cause glitches in the data that is read back, or on the source output if the parameter is applied to a source channel.

Global Parameters		
Parameter	Default Value	Changes
append_status	Off	Immediate
arm_channel	None	Immediate
arm_mode	Auto Arm	Immediate
arm_time_interval	1 Sec	Immediate
auto_group_meas	On	Wait
auto_trigger	Auto Trigger	Abort
avg_mode	None	Wait
avg_number	10	Wait
avg_update	10	Wait
avg_weight	1	Immediate
blocksize	1024	Abort
cal_dac	0	Immediate
cal_voltage	0 Volts	Immediate
calin	Grounded	Immediate
center_freq	12.8 kHz	Immediate
clock_freq	51.2 kHz	Abort
clock_master	Off	Abort
clock_source	Internal	Abort
data_mode	Block Mode	Abort
data_port	VME	Abort
data_size	16 Bit Integer	Abort
decimation_output	Single Pass	Wait
decimation_oversample	Off	Wait
decimation_undersamp	1	Wait

Global Parameters (continued)		
Parameter	Default Value	Changes
delta_order	0.1	Wait
fifo_size	0 (Use All DRAM)	Wait
filter_settling_time	64 samples	Wait
internal_debug	0	Immediate
interrupt_mask	0	Immediate
interrupt_priority	None	Immediate
lbus_mode	Pipe	Immediate
lbus_reset	Off (Not Reset)	Immediate
max_order	10	Wait
meas_time_length	0 (Run Forever)	Immediate
mmf_delay	0	Immediate
multi_sync	Off	Abort
overlap	0	Wait
peak_decay_time	1.5 Sec	Wait
peak_mode	Off	Wait
pre_arm_mode	Auto Arm	Immediate
ramp	Off	Immediate
rms_avg_time	1 Sec	Wait
rms_decay_time	0 Sec	Wait
rms_mode	Off	Wait
span	20 kHz	Immediate
sumbus	Off	Immediate
trigger_delay	0	Wait
trigger_ext	Off	Immediate
trigger_master	Off	Immediate
triggers_per_arm	1	Immediate
ttltrg_clock	TTLTRG1	Abort
ttltrg_gclock	TTLTRG1	Abort
ttltrg_satrg	TTLTRG0	Abort
ttltrg_trigger	TTLTRG0	Abort
window	Uniform	Glitch
xfer_size	0 (Use blocksize)	Wait
zoom	Off	Wait

E1432 51.2 kHz Input Parameters		
Parameter	Default Value	Changes
active	On	Abort
anti_alias_digital(*)	On	Abort
auto_range_mode	Up/Down	Immediate
calc_data	Time	Wait
coupling	DC	Glitch
enable	On	Immediate
filter_freq	200 kHz	Immediate
input_high	Normal	Glitch
input_low	Floating	Glitch
input_mode(*)	Volt	Glitch
range	10 Volts	Glitch
range_charge	50000 picoCoulombs	Glitch
range_mike	10 Volts	Glitch
trigger_channel	Off	Immediate
trigger_level lower	-10%	Immediate
trigger_level upper	0%	Immediate
trigger_mode	Level	Immediate
trigger_slope	Positive	Immediate

E1433 196 kHz Input Parameters		
Parameter	Default Value	Changes
active	On	Abort
anti_alias_digital	On	Abort
auto_range_mode	Up/Down	Immediate
calc_data	Time	Wait
coupling	DC	Abort
coupling_freq	1 Hz	Abort
enable	On	Immediate
filter_freq	200 kHz	Immediate
input_high	Normal	Glitch
input_low	Floating	Glitch
input_mode(*)	Volt	Glitch
input_offset	0	Glitch
range	10 Volts	Glitch
range_charge	50000 picoCoulombs	Glitch
range_mike	10 Volts	Glitch
trigger_channel	Off	Abort
trigger_level lower	-10%	Glitch
trigger_level upper	0%	Glitch
trigger_mode	Level	Abort
trigger_slope	Positive	Abort
weighting	Off	Wait

(*) Input mode is listed as channel-specific, but it actually applies to all channels within an SCA unless there is a smart break-out box attached to the SCA. Similarly, anti_alias_digital applies to all channels within an SCA for the E1432 51.2 kHz input.

For the following table, there is an additional column which applies only when using the E1434 65 kHz arbitrary source. The E1434 channels come in pairs, and some parameters are shared ("coupled") between each pair of channels. If they are coupled, then setting the parameter for one channel of the pair automatically sets it for the other channel of the pair. In addition, a few parameters apply only to the first of the pair

of channels, due to hardware constraints. The last column in the table specifies if the parameter is coupled. "Yes" means the parameter is coupled, "No" means that there is no coupling and the parameter can be set independantly for the two channels, and "1 Chan" means that it only makes sense to set this parameter for the first channel, and it is ignored if set for the second channel.

E1434 65 kHz Arbitrary Source Option 1D4 Single-channel Source Parameters			
Parameter	Default Value	Changes	E1434 Coupled
active	Off	Abort	No
amp_scale	1.0	Immediate	No
anti_alias_digital	On	Wait	1 Chan
duty_cycle	0.5	Wait	Yes
filter_freq	25.6 kHz	Wait	1 Chan
ramp_rate	1 Second	Wait	Yes
range	0.041567 Volt	Glitch	No
sine_freq	1 kHz	Immediate	No
sine_phase	0 Degrees	Glitch	No
source_blocksize	0 (Use input blocksize)	Wait	Yes
source_centerfreq	250 Hz	Wait	No
source_cola	Off	Wait	1 Chan
source_mode	Sine	Abort	Yes
source_output	Normal	Abort	No
source_seed	3	Wait	Yes(*)
source_span	0 (Use input span)	Wait	Yes
source_sum	Off	Wait	1 Chan
srcbuffer_init	Empty	Wait	Yes
srcbuffer_mode	Periodic	Wait	Yes
srcbuffer_size	1024	Wait	Yes
srcparm_mode	Immediate	Immediate	Yes
trigger_channel	Off	Wait	Yes

(*) The single shared value of source seed generates unique sequences on the two E1434 channels.

Option AYF Tachometer Parameters		
Parameter	Default Value	Changes
active	On	Wait
input_high	Normal	Immediate
pre_arm_rpm	600 RPM	Immediate
rpm_high	6000 RPM	Immediate
rpm_interval	25 RPM	Immediate
rpm_low	600 RPM	Immediate
rpm_smoothing	0	Immediate
tach_decimate	0	Immediate
tach_holdoff	10 Microseconds	Immediate
tach_max_time	30 Seconds	Immediate
tach_ppr	1	Immediate
trigger_channel	Off	Wait
trigger_level lower	-0.05 Volts	Immediate
trigger_level upper	0 Volts	Immediate
trigger_slope	Positive	Immediate

SEE ALSO

e1432_id(5)