Linux driver for Physik Instrumente PCI C843 card
Version 0.54

Matt Clark

May 2003

Contents

Chapter 1

Acknowledgements

Physik Instrumente (PI) GmbH and Co. KG supported the development of this driver by donating / lending hardware and software support. They have gone out of their way to support the Linux community. This software provides a high degree of functional compatibility with PI's existing software (especially their Windows driver and DLL).

Chapter 2

License

This software and documentation is cover by GPL and / or LGPL public licenses. This basically means you have access to the source code, you can modify the code, you can distribute the code and you can use the code all free. You cannot charge for this code in any way shape or form. If you distribute a modified version of the code it must be subject to these original license conditions.
The driver released under GPL, since the driver is stand alone and embedded in the kernel the license does not extend to userspace code which accesses the driver.
The library is released under LGPL - this means to can compile the library into your code without any obliging you to use GPL on the resulting application.
You can find all the details at `http://www.gnu.org/licenses`
The GPL license covering the documentation, driver and any utility / demonstration programs:

> Copyright (c) 2003 Matt Clark
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

The LGPL license covering the library:

> Copyright (c) 2003 Matt Clark

> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

The full licenses can be found at `http://www.gnu.org/licenses`.

Chapter 3

Revision history

Revisions 0.10→0.40 development versions.
Package release 0.41 first beta release.
Package release 0.50 addition of `pi_execute()` and related functions.
Package release 0.51 addition of `pi_find_home()` and `pi_vector_xxx()`..
Package release 0.52 driver interrupt handling revised.
Package release 0.53 additional functions added `pi_limit_status()`.
Package release 0.54 additional functions added: `pi_[sg]et_vel_cps()` and `pi_stop_dead()`. Documentation updated for `pi_find_home()`, `pi_vector_xxx()`, `pi_limit_status()`, `pi_[sg]et_vel_cps()` and `pi_stop_dead()`.
Package release 0.60 additional function added `pi_set_pos()`, additional utility programs added `pi_save_pos` and `pi_recall_pos`, makefiles and install scripts updated. Package release 0.61 removal of `pi_reset_board();` in `pi_openbaord();`.

Chapter 4

Known issues

As of version 0.52 the is one known issue:

4.1 Missing interrupts / no asynchronous notification

During testing of the driver it was found that *for one specific computer* that in some circumstances that the interrupts generated by the board were *not routed to the driver*. At the time of writing, following extensive testing on several different machines, it has not been possible to replicate this problem *except on the problem machine*. At present it is thought most likely that this problem is confined to this machine and not actually a driver, kernel or library issue. However, if the problem is observed (see below) please submit a full bug report to the author.
The problem is (so far) specific to one computer.
There is an existing work around for this computer.
The problem *only* affects asynchronous notification and does not affect any other functionality.

4.1.1 Description

On the specific test machine `hong` it was found that asynchronous events were never received by the user program. In the case of the test program `motion_control` this is noted by the *absence* of asynchronous notification messages like so:

```
Asynchronous event:  index=6, axis=1, timestamp=83574637,
event= 311
```

These should be received after any axis event, for instance the end of movement or and axis error. Absence of the messages may indicate the problem.
Examination of the problem revealed that the driver and the kernel never received the interrupt. Examination of the system revealed that the interrupt was

registered correctly and that as far as the kernel was concerned any interrupts would be routed.
It was also found by experiment that by changing the PCI slot or BIOS configuration to get a different interrupt that under some circumstances the interrupt could be routed and did lead to asynchronous notification.
A list of "bad" IRQs were found on the system - according to the kernel no interrupts were ever generated or received on any of these IRQs regardless of the installed hardware or drivers. Any of the "good" IRQs worked as normal.

4.1.2 Workaround

Choose a PCI slot or configure the BIOS to give a good IRQ. Please submit a full bug report with kernel version (uname -a), processor (cat /proc/cpuinfo) and motherboard type and the following information: (after driver install and before, during and after running manual_slm) dmesg, cat /proc/interrupts, cat /proc/pi_stage. Also /sbin/lspci -vv.

Chapter 5

Notes for windows users

If you are familiar with Linux skip this chapter. At the request of PI I have included this to explain some differences between Linux and Windows and how this affects device drivers and more importantly compatibility between Windows and Linux software using the respective drivers.

5.1 Compiler / library / API / GUI differences

The main difference you will notice is the compilers - windows users will probably be familiar with some variant on MS visual C++ with access to a large windows API. Under Linux you will almost certainly be using G++ from the GCC compiler suite and will have access to standard Unix libraries. G++ aims to conform to the ISO 14882 C++ Standard and POSIX standards. You will have no access to any windows API functions and this will probably mean a complete rewrite of your code.
You will probably have to start by removing any idea of a windows based GUI - my suggestion is that you start with a command line based text program and then implement any GUI on top via the internet / mozilla / apache / PHP (my solution) or Tk / Tcl scripting language. You can support X-windows directly but you face a fairly severe learning curve - you may want to start with libsx - a simplified X windows library which is adequate for most GUIs and a lot simpler than full X.

5.2 Device drivers under Linux / Unix

Devices drivers under Linux / Unix are bits of the kernel (operating system) that are accessed via standard library calls that are (more or less) common to all flavours of Unix / Linux. These library calls are basically C IO calls. File and device access under Unix is essentially the same (although file IO is really stream IO which is another layer built on top of the basic IO functions).

Most device drivers are implemented as "loadable kernel modules" - bits of compiled code that can be inserted and removed from a running kernel. They *must* be compiled specifically for the kernel that is running - even a minor difference can result is big trouble. *Never force a module to be loaded if it complains about incompatible versions* - you could crash your computer and lose all your data! Fortunately it is simple to avoid, in a standard installation all you have to do is recompile the driver whenever you update or recompile your kernel. This is a trivial step in the installation process. The kernel will refuse to load an incorrect driver.

5.2.1 How devices drivers appear

In Unix there is a generally philosophy of treating everything like a file (exception network drivers). Device drivers are accessed through special files which are normally found in the directory /dev. To open a device driver you open its entry in /dev like a file, to send data to it you write to it, to get data out of it you read from it etc.
Through this special file (and via the library functions open, read, write, ioctl and close for char special drivers) you communicate with the device driver in the kernel - this device driver then directly communicates with the hardware to get the jobs done.
There are two basic types of device driver file that you will encounter *block special* and *char special*. They support different library functions - ignore block special drivers they are primarily for raw disk drives and other storage media which support the `mount` command.

5.2.2 Major and Minor numbers

Special file entries for drivers come with two numbers associated with them the *major* and *minor* device numbers - you need to know what they do, once you know you can forget them. The major number links the special file with a particular driver - when you open this file the system opens communication with the driver that is registered to that number. The minor number is used as private data by the driver - typically this is used to refer to a particular device. For instance, the major block device number on my computer for IDE devices is 3 and all device drivers for IDE disks have 3 as the major number and then minor numbers that refer to physical disks or disk partitions:

```
file /dev/hda*
/dev/hda:   block special (3/0) ## the actual disk
/dev/hda1:  block special (3/1) ## my /boot partition
...
/dev/hda2:  block special (3/2) ## my / partition
...
/dev/hda3:  block special (3/3) ## my swap partition
```

There may be loads of other entries in /dev/ which don't actually relate to real hardware - just ignore them - it is common practise to have entries like this - if you try to use them you will get an error.

What you need to know about these numbers

The driver this library installs is called `/dev/pi_stage`, the major number will be dynamically allocated at install time by the install script: *this major number may change from install to install and from reboot to reboot.* You don't really need to worry about this so long as you use `/dev/pi_stage` to access the driver and the install script.
The minor number is not used at present and is reserved for selecting between multiple cards (and different stages) if this is ever supported.

5.2.3 Windows DLL and Linux library differences

The are a number of differences between the Windows DLL and Linux library - most of which arise from the differences between the operating systems.

5.2.4 Function names

The first difference you will probably notice between the windows DLL and the Linux library is that the function names have changed. All the Linux functions start with `pi_` and with the exception of the `pi_xxxQMC()` functions all the names are in lower case consistent with Linux programming styles.
The prefix on the names is to differentiate the functions from their windows counterparts (because there are some functional differences) and to prevent namespace pollution (for instance my hardware control program controls 22 separate bits of hardware each of which has an `init_board()` type function - the prefix prevents conflicts).

5.2.5 Types and function calling conventions

The are several defences between Windows and Unix types which result in differences between the libraries the most important are:

No `HRESULT`

Windows API functions usually return `HRESULT`, consequently most DLL functions that return data usually do so via a pointer. `HRESULT` does not exist in Unix so most library functions return either `void` or data.

`sizeof(int)`

ints are 16 bits under windows and *at least* 32 bits under Linux (on some platforms they are 64 bits long). Both 16 bit and 32 bit data and results are passed as ints in the Linux library - the DLL used a combination of 16 bit ints and

pairs of 16 bit ints to pass 16 bit and 32 bit data. Where two 16 bit words are required (as opposed to one 32 bit word) two 32 bit ints are used - no range reduction or range error checking is performed. You shouldn't really notice this when using the library except when you would normally split a 32 bit argument in to 2 16 bit ones (just use the single 32 bit one).

5.2.6 Numeric constants

The numeric constants for the various QFL commands have changed - if you use the symbolic constants this will not affect you - if you are porting code with contains the numeric constants you will have to update them. The Linux library uses a completely new technique for implementing these commands this uses different numeric constants - see chapter 12 for details.

5.2.7 Call by arguments

Several functions now support calling by variable numbers of arguments so that rather than `MoveA`, `MoveA_12`, etc you use `pi_move_abs` with one, two three or four pairs of arguments.

5.2.8 RESET

Calling `pi_reset_board` or `PI_RESET` is not required in the linux driver - if you are porting Windows code that uses these functions consider removing them. See section 11.4.2 for more details.

Chapter 6

Description of software

This software package is constitutes a kernel space device driver and user space library for controlling Physik Instrumente's (PI) PCI C843 (and variations) PCI cards. These card consist of a PCI bridge, a motion control chipset and output amplifiers for controlling PI's range of translation stages.
The software comes in two main bits: the driver and the library. The driver is loaded into the Linux kernel (operating system) and provides a standard interface (char dev) for the system and user. The driver functionally resides in a special file (char dev) called /dev/pi_stage which is accessed (like a file) to control the stages.
The library is a suite of user functions which simplify control of the card, motion control chipset and stages. These functions provide a degree of compatibility with the PI Windows DLL. It is recommended that the library is used to control the card etc. rather than raw IO because this insulates the user from potentially damaging IO errors.

6.1 SMP and non ix86 systems

Linux runs on many different platforms including many different processors and SMP multiprocessor systems. The kernel is designed with a high degree of compatibility between different platforms, however minor differences do exist.
The driver should work with SMP systems BUT it has never been tested under SMP. This means that it has never been compiled for an SMP system and it has never been installed on an SMP. I do not have access to any SMP build platforms and I am not likely to get access to SMP systems. The long and short of it is that installation on SMP systems is not recommended. If you want to have a go please remember this could deadlock your system if it doesn't work (spinlocks are in place and the driver implements single openness so deadlocks and reentrancy problems should be Ok but they have never been tested).
Non ix86 platforms: the driver should (nearly) work out of the box for other processor types. Things to look out for are byte ordering conversions (little-

endian, big endian problems) and minor initialisation problems. I will be happy to help debug any of these for you - if they exist it will be due to ignorance and laziness on my part.

6.2 PCI compliance and hardware limitations

Please note that there are some hardware limitations that are not a function of the driver. In particular the card is *not* (quite) PCI compliant because it does not have a unique vendor / function ID. It looks to the system like a, "Co-processor: Texas Instruments PCI2040 PCI to DSP Bridge Controller" because it uses a TI PCI to DSP Bridge and still uses TI's PCI id.
PCI card are supposed to have unique vendor / function ID numbers - this is how the computer is supposed to find the card and differentiate it from others. This was a major advance over the ISA architecture.
Since the card does not have a unique ID it is technically possible to mistake it for a another device which uses the same ID (ie a TI DSP card). To reduce the chance of trashing your system the driver probes the motion chipset further but this is not failsafe: probing may upset a different card using the same PCI ID (if present) and it only probes 4 bits so there is a 1 in 16 chance that it will mistakenly identify another card (using the TI ID) as a PI card.
Summary: It is unlikely that you will encounter this problem - if you do contact PI and ask them to obtain a unique ID, then contact me for a driver update (it is possible to identify the card by slot location as well).

6.3 Driver limitations

The driver is coded for a single card and single openness.
The driver does not support multiple cards because I only had one card to develop on. Contact me if you require multicard support, I may develop it if I have multiple cards to test with. Linux could support multiple cards (if coded) without the PROM changes required for the windows driver using geographical addressing.
The driver implements single openness, that is the driver only allows access by one application at a time. This is encoded as a matter of policy because of safety concerns.

6.4 Performance

If you were expecting a huge leap in performance from an ISA card system to the PCI card system forget it. The card uses a PMD navigator motion control chips set (`http://www.pmdcorp.com/`) to control the system. This chipset is the primary performance limiter in the PCI card. Compared with the C842 ISA card the performance is up 10 times (commands / second) but this is way off the potential performance gains that migrating from ISA to PCI could give.

The card can allow poll based IO or interrupt based IO. This driver uses poll based IO as the interrupt lag (this appears to be on the card rather than in the system) costs too much in performance terms. This means that around 20k/s instructions can be sent to the card but at a cost of 100% CPU usage (waiting for the motion chipset to be ready after each 16 bit word transfer). Interrupt driven IO was at least 10× slower because of the lag. I believe that the winNT driver uses interrupt driven IO but all other windows flavours use poll driven IO.
The driver implements a 2 stage poll process, it polls quickly (blocking) for a preset number of times before switching to a slow (non-blocking) number of times. In normal use it should always complete in the first set, abnormally it can complete in the second set. If it fails the second set then there is a serious error and it is possible that the card has crashed, this will cause and exception and report an error in the kernel logs. In this case you may have to reboot the system. This has only ever been necessary during testing to set the number of fast and slow polls for optimum performance. It is possible (but very unlikely) that on an non-X86 architecture or a future go-faster computer that this number will have to be revised.
The slow polling method can be reached if the preset number of times is too low or if your computer is considerably faster than the development one. If you ever observe the following message, "`PI: Warning: Driver error: looped out - please contact driver author`", in the logs then this is likely to be the problem. Contact me and I will revise the number of polls.
If you observe the message, "`PI: Driver error: fatal: you must reboot to recover - contact driver author`", then the board has crashed. This should never happen and it indicates a serious error. Please make a full bug report.
These messages may be withdrawn in later versions of the code.

6.5 Asynchronous notification of events

The PMD chipset can generate interrupts indicating events occurring on the axis. The driver can route these to userspace programs using signals, this involves the scheduler running and means that there is a potential time lag for notification of the order of 10ms or more. Faster responses are possible using blocking IO and a custom kernel (with HZ set higher and lock-breaking patches applied). I will consider these as feature requests if there is demand, they involve a lot of work but should guarantee sub ms response for realtime processes. In the meantime I will be polling from user space (upto 20kHz) for time critical measurements. Most of the kernel modifications required here are expected to be in k2.6 (lock-breaking and HZ are already in 2.5 see `http://www.tech9.net/rml/linux/`).

6.5.1 Ringbuffer overflow

The driver implements a small circular ring buffer to capture events to. If the events are not read then the buffer will fill up and the head and tail of the buffer will collide. In this case the driver deletes the oldest entry to make space for the new. If the buffer is drained correctly then this should never happen in ordinary use, however, if the events come in a fast burst (before the user code can empty the buffer) it is possible that the buffer will fill up. In this case the driver prints the following message in the logs "`PI: ring buffer collision`". If this occurs check your user code to make sure you collect the events you are registered for or to make sure you haven't requested events you don't want. If you are sure your code is operating correctly then it is possible the buffer is too small, in this case contact the author to increase the buffer size. The current buffer size is 32 events.
This feature can involve a huge number of kernel messages and may slow the system down if events are generated quickly and not collected. It may be withdrawn or changed in future releases.

Chapter 7

Obtaining and installing the software

Linux drivers are usually supplied as source code which is then compiled on your system and inserted into the kernel (operating system). Kernels on different computers vary so you must recompile the code for each computer you use it on. The software comes with scripts to simplify building and installing the software. This section covers getting and installing the source code, chapters 8 and 9 cover compiling the driver and library respectively.

7.1 Getting the source code

This is a dummy section, no home has been decided for this software package. It should be available from PI and EEE, Nottingham.

7.2 Installing the software ready to build

Instructions for build the individual bits of the software are detailed in chapters 8 and 9. This section covers installing the source code ready to build and installing the system software necessary to build the driver.
The software comes bundled up as a tar.gz package. Save this to a sensible working directory and unpack the source code (version numbers might be different) using the tar program (> is the prompt):

```
> tar -zxvf pi_driver_v0.52.tar.gz
pi_driver/
pi_driver/manual.aux
....
```

This will create a directory pi_driver with all the source code in it, it is from this install directory that all software building is done.

It is important that the software is built for the system it is used on. If you upgrade you system or kernel or rebuild your kernel you must rebuild the driver and the library. Never be tempted to use or force the use of a mismatching driver. Never use a mismatching driver and library. Always use the same compiler for building the kernel and driver.
Using a mismatched kernel and driver could trash your system - you have been warned.

7.3 Before building....

Before building you may have to install additional system components. In order to build any of the software you must have the gcc compiler suite installed. The driver was written using gcc version 3.xx, it should compile with late versions of gcc 2.xx without problem. The compiler suite is probably available on your installation as standard, if not consult your system documentation about installing it or find it at `http://www.gnu.org/software/gcc/gcc.html`.
You must have the relevant kernel sources installed. These are used by the driver during compilation. The kernel sources must match the kernel you are running, if not you will get a version mismatch which will (hopefully) prevent you from loading the kernel (if not it could trash your system).
The driver is compatible with 2.4.x kernels only, it has not been tested on any 2.5 kernel and is not backwards compatible with 2.2.x.

Chapter 8

The kernel driver

8.1 Building the driver

In the directory "`driver`" in the install directory build the driver using the make command:

```
> make


gcc -c -O2 -I /usr/src/linux-2.4/include -D__KERNEL__ -DMODULE
-DSLM_DEBUG pi_stage.c
```

The most likely error here is that the compiler cannot find the kernel sources. The kernel sources are specified in the `Makefile` by the line

```
# Edit this path to point at your kernel sources
INCLUDES= -I /usr/src/linux-2.4/include
```

you may need to edit the `Makefile` to point it to the correct location for your kernel headers.
Assuming the build process completes (there should be no errors or warnings) it will create a file `pi_stage.o`, this is a loadable kernel module (the driver) ready to be inserted into the kernel.

8.2 Installing the driver

Once the build process is complete you are ready to install the kernel driver. The tools included here only allow for installing the driver and removing the driver from the system. They do not configure it to be automatically installed on boot. If you wish to do this consult your local documentation.
To install the driver use the `install_pi_stage` script (or `make install`):

```
> ./install_pi_stage
Installing device driver
Determining device driver major
Checking the number=254
************** Making Binary device
****************************** Setting permissions
/dev/pi_stage ****************************** Installed successfully
```

You must be logged in as root (superuser) to do this (use 'su' to become root). If the script detects any errors it will abort. If it is successful it will create a file /dev/pi_stage, this is the device driver. It will also create a /proc filesystem /proc/pi_stage. This is a special file which can be read to see the status of the driver, in version 0.52 this files looks like this:

```
> cat /proc/pi_stage
PI_STAGE: PI PCI stage controller driver v0.52 March 2003
(c)mc 2003

Interrupt status
Interrupts rec'd (all) 0
Interrupts rec'd (io) 0
Interrupts rec'd (axis) 0
Interrupts rec'd (board) 0

Profileing info
2-1 0
3-2 0

Last error status:
No errors

Ring buffer status
Async events rec'd 0
Event head 0
Event tail 0
Null proc entry 369
```

Most of this is debugging information of no relevance to the user, however, it does ensure that the driver is running in the kernel.
The driver creates kernel messages you can see these in /var/log/messages or more conveniently by using the dmesg command:

```
> dmesg
...
...
```

```
PI PCI stage controller v0.52 March 2003 (c)mc

PI: stage contoller card found- probing for additional info
PI: init_modules:  interrupt not claimed - in open now
PI: reset performed
PI: Board probed, version 2140, 22
PI: Board has 4 axis capability
PI: init_module:  device major=254
PI: init_module:  installing /proc
PI: installation complete
PI: enabling board
```

If things go wrong valuable debugging information will go here.

8.3 Removing the driver

The driver will be automatically remove on shutdown or reboot. To remove it before hand use the `remove_pi_stage` script:

```
> ./remove_pi_stage
```

This removes the driver and the files `/dev/pi_stage` and `/proc/pi_stage`.

8.4 Installing and removing while preserving the stage position

Please refer to section 10.2 for information on how to install and remove the driver while preserving the stage positions by saving the position to disk.

Chapter 9

The userspace library

9.1 Building the library

In the directory "`library`" in the install directory build the library using the make command:

```
> make
g++ -c pi_user.cc
ar crv libpi_user.a pi_user.o
a - pi_user.o
g++ -O2 -Wall -c -o motion_control.o motion_control.cc
g++ motion_control.o -o motion_control -lpi_user -lncurses
-L .
```

This builds the library `libpi_user.a`. [1]

9.2 Installing the library and header files

You can install the library and header files in `/usr/local/lib` and `/usr/local/include` using the `make install` command:

```
make install
cp libpi_user.a /usr/local/lib
cp pi_user.h /usr/local/include
```

You will need to be logged in as root to do this.

[1] **Note:** Building the library requires the `pi_driver.h` header from the driver sources and expects to find it in "`../driver`", if you relocate the library source code or change the install process this will break the make process if this header can't be found in the usual place.

The compiler should pick these files up in these locations, however, if it doesn't you may have to relocate these files or point the compiler to them using the `-I` and `-L` flags, see your local documentation.

9.3 Compiling user programs with the library

To use the library you must include the header `pi_stage.h` in your source code and link to the library at compile time. Add the following line to you source code:

```
#include <pi_user.h>
```

Compile your program (`mycode.cc`) linking to the library thus:

```
> g++ mycode.cc -lpi_user
```

9.4 Porting windows code

The user library and kernel driver provide a fully featured interface for controlling PI stages via the C843 card. Differences between the DLL and Linux library exist because of function differences between the two systems and because of style differences between Windows and Unix programming (Windows code looks ugly). The library is written for Unix programmers but nonetheless provides a high degree of functional compatibility with the Windows DLL.

9.4.1 Comparison with the PI windows DLL

All function and variable names in the Linux library are prefixed with `pi_` or `PI_` to reduce namespace pollution[2].
Most of the windows DLL functions return `HRESULT`, this is meaningless in Linux so functions either return `void` or something useful. This means that the arguments are often changed between windows DLL and Linux library counterparts. Details can be found in chapter 11.
Some of the functions have no meaning in the Linux driver and are not supported (for instance `SelectBoard()`, the Linux driver only supports one board and would implement multiboard access differently anyway).
Currently command parsing functions (for instance `translate`) are not supported. These do not offer any additional functionality and are inefficient in comparison with the `pi_setQMC()` interface. This should not present a significant problem for converting programs from windows to Linux.

[2]For instance one of my applications uses 5 different hardware control systems, each one has a function `init_board()` prefixing with a unique identifier aids program development

Chapter 10

Utility / Example programs

The software also includes a utility /example program which demonstrates the use of the library. This isn't the greatest bit of software and if has grown rather inelegantly from simple test program to its present form.

10.1 motion_control

This is automatically built when the library is built.
It is an interactive terminal based program which controls up to four axis. The program is run from the command line thus:

```
> ./motion_control
```

If it cannot contact the device driver (because it is not installed) you will get this message: (install the driver before using).

```
Matt's PI stage demo 0.32 13 Feb 2003
About to auto init everything and switch to screen mode

Opening the board
Failed (is the device driver installed?)
Press any key to quit
```

Assuming that the driver is Ok you will see the follow instruction screen:

```
Matt's PI stage demo 0.32 13 Feb 2003
About to auto init everything and switch to screen mode

Opening the board   Ok
The number of axis supported 4
Axis 0: sense 0   Axis 1: sense 0   Axis 2: sense 1   Axis 3: sense 1
   Commands:
   To set axis type a# where # is the axis number
```

```
To move abs type m#####<enter> where ##### is the position
To move rel type s#####<enter> where ##### is the shift
To set velocity  v#####<enter> where ##### is counts/cycle
To move off limits      r
To set motor Output     o
To enable brakes        b
To set motor PWM        p
To set motor DAC        d
To clear errors         e
To quit type q


                  Press any key to continue
```

After this you will see the working screen:

```
Axis 0            Axis 1            Axis 2            Axis 3
Lim OK            Lim OK            Lim OK            Lim OK
V=50000           V=50000           V=50000           V=50000
V=0               V=0               V=0               V=0
P=0               P=0               P=0               P=0
E=0               E=0               E=0               E=0
BRK=1             BRK=1             BRK=1             BRK=1
AMP=1             AMP=1             AMP=1             AMP=1
MTR=1             MTR=1             MTR=1             MTR=1
ENC=0             ENC=0             ENC=0             ENC=0
AB=10             AB=11             AB=00             AB=11
Analog            Analog            Analog            Analog
M 300             M 1300            M 2300            M 3300
  Ok                Ok                Ok                Ok


  0                 0                 0                 0


Command   Axis 0
```

where V(1) is the commanded velocity, V(2) is the actual velocity (counts / second extracted from counts / cycle hence the low resolution), P is the position (counts), E is the position error, BRK is the analogue output to the brakes, AMP is the analogue output to the amplifiers, MTR is the motor enable signal, ENC is the encoder source signal, AB is the encoder status, Analog / PWM+/- is the output mode from the chipset (analog or PWM) and M is the event status (hex).
The status of each axis is updated continuously. If the limit switches are activated the motor will be disabled and `Lim OK` will change to `On +ve` or `On -ve`,

use the 'r' command to move off the limit switches.
Asynchronous events are captured and display above the command line. The program is set up to capture "motion complete" interrupts.

10.2 Saving and recalling the origin

As long as the driver is loaded the position of the stages is remembered by the chipset in the card so that subsequent programs will see the same origin. However, once the driver has been unloaded the card is stateless and the origin will probably be lost. This will also occur when a machine is rebooted.
Two utility functions `pi_save_pos` and `pi_recall_pos` are included which will save the current position to disk and recall it again. These utilities are built automatically with the user library and installed by default in `/usr/local/bin`. They save the position to a file in `/etc/pi_stages`.
Assuming the stages are not moved (by hand or by an other driver such as the windows driver) saving the position before unloading the driver and then recalling it after loading but before use should statefully restore the position.
Two scripts are included that preserve the last known position of the stages between driver loads and unloads these are: `install_pi_stage_recall` and `remove_pi_stage_save`. They can be found with the `install_pi_stage` and `remove_pi_stage` scripts (note: the save and recall scripts use these scripts and must be in the same directory as them to work). These scripts can only be used once the user library has been compiled and installed.
These are designed to be a sensible way of preserving the reference position between machine reboots, obviously they cannot work if the stages have been moved while the driver has been unloaded.
Expect small drifts in position while the stages are turned off and expect vertical stages to have shifted due to gravity.

Note:

The directory `/etc/pi_stage` will have the default permissions for root. This should allow root to read and write files here but no one else. This is usually fine as you have to be root (superuser) to install or remove the driver. If you wish to use `pi_save_pos` and `pi_recall_pos` without being root you will have to change the permissions of this directory (and possibly the files within it).

Caution!

Watch out for error messages. If any errors are reported then it is possible that the origin has not been recalled successfully and the stages may move in an unpredictable if you then rely on the origin being correct.
The format of the file may change between releases so you cannot rely on this feature to work during the transition between different versions of the software (although it probably will).

Chapter 11

Library reference

11.1 Windows DLL to Linux library conversion

Windows DLL function	Linux libpi_user function
autodetect	int pi_auto_set_limit(PI_AXIS);
AutoFindEdge	void pi_find_home(PI_AXIS);
Axis_Installed	int pi_axis_installed(void);
get_activity_status	int pi_get_activity_status(PI_AXIS);
get_event_status	int pi_get_event_status(PI_AXIS);
get_signal_status	int pi_get_signal_status(PI_AXIS);
get_pos	int pi_get_pos(PI_AXIS);
get_pos4	void pi_get_pos4(int *);
getQMC	int pi_getQMC(int, PI_AXIS);
getQMCA	int pi_getQMC(int, PI_AXIS, int);
	int pi_getQMCA(int, PI_AXIS, int);
get_RefSignal	not supported in this version
get_v	int pi_get_v(PI_AXIS); (deprecated)
	int pi_get_vel_cps(PI_AXIS); (preferred)
InitBoard	BOOL pi_init_board(void) (called by pi_openboard)
Init_LS	void pi_init_ls(PI_AXIS, int);
InitAxis	BOOL pi_init_axis(PI_AXIS);
MoveA	void pi_move_abs(PI_AXIS, int);
MoveA_12	void pi_move_abs(PI_AXIS, int, PI_AXIS, int);
MoveA_123	void pi_move_abs(PI_AXIS, int, ...);
Also MoveA_1234	void pi_move_abs(PI_AXIS, int, ...);
MoveR	void pi_move_rel(PI_AXIS, int);
MoveR_12	void pi_move_rel(PI_AXIS, int, PI_AXIS, int);
MoveR_123	void pi_move_rel(PI_AXIS, int, ...);
Also MoveR_1234	void pi_move_rel(PI_AXIS, int, ...);
moving	BOOL pi_moving(PI_AXIS);
OpenBoard	BOOL pi_openboard(void); // use this one
	BOOL pi_openboard(char *);
	BOOL pi_closeboard(void);
SelectBoard	not supported
setQMC	void pi_setQMC(int, PI_AXIS, int);
setQMCA	void pi_setQMC(int, PI_AXIS, int, int);
	void pi_setQMCA(int, PI_AXIS, int, int);
VectorA	void pi_vector_abs(PI_AXIS, int, PI_AXIS, int, int);
VectorB	void pi_vector_rel(PI_AXIS, int, PI_AXIS, int, int);
WaitStop	pi_wait_stop(PI_AXIS axis);
execute	pi_execute
execute2	pi_execute
GeneralCommandParser	not supported
translate	not supported
translate_error	not supported

Additional functions	Description
pi_get_limit_status(PI_AXIS)	returns the limit status for the axis
pi_get_vel_cps(PI_AXIS)	returns the velocity in ∼ counts per second
pi_set_vel_cps(PI_AXIS)	returns the velocity in ∼ counts per second
pi_stop_dead(PI_AXIS)	stops the axis by target pos to actual pos
pi_stop_dead(void)	stops all axis
pi_set_pos(PI_AXIS, int pos)	Calls PI_SET_ACTUAL_POSITION rewrites the reference position

11.2 Function library

11.2.1 IO primitives

These functions are not intended for user use. They are IO primitives intended to simplify the library and improve readability. Please don't use these functions, they are not safe for user code and reserved for future changes. They are documented here for completeness only.

```
inline int pi_io_c(PI_CMD_ONLY);
inline int pi_io_cw(PI_CMD_WRITE1, PI_DATA);
inline int pi_io_cww(PI_CMD_WRITE2, PI_DATA, PI_DATA);
inline int pi_io_cwww(PI_CMD_WRITE3, PI_DATA, PI_DATA, PI_DATA);
inline int pi_io_cr(PI_CMD_READ1, PI_DATA *);
inline int pi_io_crr(PI_CMD_READ2, PI_DATA *);
inline int pi_io_cwr(PI_CMD_WRITE_READ1, PI_DATA, PI_DATA *);
inline int pi_io_cwrr(PI_CMD_WRITE_READ2, PI_DATA, PI_DATA *);
```

11.3 IO user functions

These are the main bank of functions for user IO, in all cases the first argument is the command, the second the axis and subsequent ones the arguments which are all 32 bit signed integers (ints). They are the preferred way of performing low level IO and considered safe.
Most users will probably use these functions sparingly, preferring to use the functions in the following sections.
The functions `pi_getQMCA` and `pi_setQMCA` are identical to their non 'A' counterparts and provide for compatibility with the DLL.

```
int pi_getQMC(int, PI_AXIS);
int pi_getQMC(int, PI_AXIS, int);
int pi_getQMCA(int, PI_AXIS, int);
void pi_setQMC(int, PI_AXIS, int);
void pi_setQMC(int, PI_AXIS, int, int);
void pi_setQMCA(int, PI_AXIS, int, int);
```

The following commands are defined for this library:

```
// send a command with no data IO
#define PI_CLEAR_INTERRUPT              (PI_CMD_ONLY)0xac
#define PI_NO_OPERATION                 (PI_CMD_ONLY)0x00
#define PI_RESET                        (PI_CMD_ONLY)0x39
#define PI_UPDATE                       (PI_CMD_ONLY)0x1a
// commands that get something but require an argument
#define PI_READ_ANALOG                  (PI_CMD_WRITE_READ1)0xef
#define PI_READ_BUFFER                  (PI_CMD_WRITE_READ2)0xc9
#define PI_READ_IO                      (PI_CMD_WRITE_READ1)0x83
#define PI_GET_TRACE_VARIABLE           (PI_CMD_WRITE_READ1)0xb7
#define PI_GET_BREAKPOINT               (PI_CMD_WRITE_READ1)0xd5
#define PI_GET_BREAKPOINT_VALUE         (PI_CMD_WRITE_READ2)0xd7
#define PI_GET_BUFFER_FUNCTION          (PI_CMD_WRITE_READ1)0xcb
#define PI_GET_BUFFER_LENGTH            (PI_CMD_WRITE_READ2)0xc3
#define PI_GET_BUFFER_READ_INDEX        (PI_CMD_WRITE_READ2)0xc7
#define PI_GET_BUFFER_START             (PI_CMD_WRITE_READ2)0xc1
#define PI_GET_BUFFER_WRITE_INDEX       (PI_CMD_WRITE_READ2)0xc5

// commands that get something
#define PI_GET_ACCELERATION             (PI_CMD_READ2)0x4c
#define PI_GET_ACTIVITY_STATUS          (PI_CMD_READ1)0xa6
#define PI_GET_ACTUAL_POSITION          (PI_CMD_READ2)0x37
#define PI_GET_ACTUAL_VELOCITY          (PI_CMD_READ2)0xad
#define PI_GET_AUTO_STOP_MODE           (PI_CMD_READ1)0xd3
#define PI_GET_AXIS_MODE                (PI_CMD_READ1)0x88
#define PI_GET_AXIS_OUT_SOURCE          (PI_CMD_READ1)0xee
#define PI_GET_CAPTURE_SOURCE           (PI_CMD_READ2)0xd9
#define PI_GET_CAPTURE_VALUE            (PI_CMD_READ2)0x36
#define PI_GET_CHECKSUM                 (PI_CMD_READ2)0xf8
#define PI_GET_COMMANDED_ACCELERATION   (PI_CMD_READ2)0xa7
#define PI_GET_COMMANDED_POSITION       (PI_CMD_READ2)0x1d
#define PI_GET_COMMANDED_VELOCITY       (PI_CMD_READ2)0x1e
#define PI_GET_CURRENT_MOTOR_COMMAND    (PI_CMD_READ1)0x3a
#define PI_GET_DECELERATION             (PI_CMD_READ2)0x92
#define PI_GET_DERIVATIVE               (PI_CMD_READ1)0x9b
#define PI_GET_DERIVATIVE_TIME          (PI_CMD_READ1)0x9d
#define PI_GET_DIAGNOSTIC_PORT_MODE     (PI_CMD_READ1)0x8a
#define PI_GET_ENCODER_MODULUS          (PI_CMD_READ1)0x8d
#define PI_GET_ENCODER_SOURCE           (PI_CMD_READ1)0xdb
#define PI_GET_EVENT_STATUS             (PI_CMD_READ1)0x31
#define PI_GET_GEAR_MASTER              (PI_CMD_READ1)0xaf
#define PI_GET_GEAR_RATIO               (PI_CMD_READ2)0x59
#define PI_GET_HOST_IO_ERROR            (PI_CMD_READ1)0xa5
#define PI_GET_INTEGRAL                 (PI_CMD_READ2)0x9a
#define PI_GET_INTEGRATION_LIMIT        (PI_CMD_READ2)0x96
#define PI_GET_INTERRUPT_AXIS           (PI_CMD_READ1)0xe1
```

```
#define PI_GET_INTERRUPT_MASK           (PI_CMD_READ1)0x56
#define PI_GET_JERK                     (PI_CMD_READ2)0x58
#define PI_GET_KAFF                     (PI_CMD_READ1)0x94
#define PI_GET_KD                       (PI_CMD_READ1)0x52
#define PI_GET_KI                       (PI_CMD_READ1)0x51
#define PI_GET_KOUT                     (PI_CMD_READ1)0x9f
#define PI_GET_KP                       (PI_CMD_READ1)0x50
#define PI_GET_KVFF                     (PI_CMD_READ1)0x54
#define PI_GET_LIMIT_SWITCH_MODE        (PI_CMD_READ1)0x81
#define PI_GET_MOTION_COMPLETE_MODE     (PI_CMD_READ1)0xec
#define PI_GET_MOTOR_BIAS               (PI_CMD_READ1)0x2d
#define PI_GET_MOTOR_COMMAND            (PI_CMD_READ1)0x69
#define PI_GET_MOTOR_LIMIT              (PI_CMD_READ1)0x07
#define PI_GET_MOTOR_MODE               (PI_CMD_READ1)0xdd
#define PI_GET_OUTPUT_MODE              (PI_CMD_READ1)0x6e
#define PI_GET_POSITION                 (PI_CMD_READ2)0x4a
#define PI_GET_POSITION_ERROR           (PI_CMD_READ2)0x99
#define PI_GET_POSITION_ERROR_LIMIT     (PI_CMD_READ2)0x98
#define PI_GET_PROFILE_MODE             (PI_CMD_READ2)0xa1
#define PI_GET_SAMPLE_TIME              (PI_CMD_READ1)0x61
#define PI_GET_SERIAL_PORT_MODE         (PI_CMD_READ1)0x8c
#define PI_GET_SETTLE_TIME              (PI_CMD_READ1)0xab
#define PI_GET_SETTLE_WINDOW            (PI_CMD_READ1)0xbd
#define PI_GET_SIGNAL_SENSE             (PI_CMD_READ1)0xa3
#define PI_GET_SIGNAL_STATUS            (PI_CMD_READ1)0xa4
#define PI_GET_STOP_MODE                (PI_CMD_READ1)0xd1
#define PI_GET_TIME                     (PI_CMD_READ2)0x3e
#define PI_GET_TRACE_COUNT              (PI_CMD_READ2)0xbb
#define PI_GET_TRACE_MODE               (PI_CMD_READ1)0xb1
#define PI_GET_TRACE_PERIOD             (PI_CMD_READ1)0xb9
#define PI_GET_TRACE_START              (PI_CMD_READ1)0xb3
#define PI_GET_TRACE_STATUS             (PI_CMD_READ1)0xba
#define PI_GET_TRACE_STOP               (PI_CMD_READ1)0xb5
#define PI_GET_TRACKING_WINDOW          (PI_CMD_READ1)0xa9
#define PI_GET_VELOCITY                 (PI_CMD_READ2)0x4b
#define PI_GET_VERSION                  (PI_CMD_READ2)0x8f

// commands that set something
#define PI_ADJUST_ACTUAL_POSITION       (PI_CMD_WRITE2)0xf5
#define PI_CLEAR_POSITION_ERROR         (PI_CMD_WRITE1)0x47
#define PI_MULTI_UPDATE                 (PI_CMD_WRITE1)0x5b
#define PI_RESET_EVENT_STATUS           (PI_CMD_WRITE1)0x34
#define PI_SET_ACCELERATION             (PI_CMD_WRITE2)0x90
#define PI_SET_ACTUAL_POSITION          (PI_CMD_WRITE2)0x4d
#define PI_SET_AUTO_STOP_MODE           (PI_CMD_WRITE1)0xd2
#define PI_SET_AXIS_MODE                (PI_CMD_WRITE1)0x87
```

```
#define PI_SET_AXIS_OUT_SOURCE          (PI_CMD_WRITE1)0xed
#define PI_SET_BREAKPOINT               (PI_CMD_WRITE1)0xd4
#define PI_SET_BREAKPOINT_VALUE         (PI_CMD_WRITE3)0xd6
#define PI_SET_BUFFER_FUNCTION          (PI_CMD_WRITE2)0xca
#define PI_SET_BUFFER_LENGTH            (PI_CMD_WRITE3)0xc2
#define PI_SET_BUFFER_READ_INDEX        (PI_CMD_WRITE3)0xc6
#define PI_SET_BUFFER_START             (PI_CMD_WRITE3)0xc0
#define PI_SET_BUFFER_WRITE_INDEX       (PI_CMD_WRITE3)0xc4
#define PI_SET_CAPTURE_SOURCE           (PI_CMD_WRITE1)0xd8
#define PI_SET_DECELERATION             (PI_CMD_WRITE2)0x91
#define PI_SET_DERIVATIVE_TIME          (PI_CMD_WRITE1)0x9c
#define PI_SET_DIAGNOSTIC_PORT_MODE     (PI_CMD_WRITE1)0x89
#define PI_SET_ENCODER_MODULUS          (PI_CMD_WRITE1)0x8e
#define PI_SET_ENCODER_SOURCE           (PI_CMD_WRITE1)0xda
#define PI_SET_GEAR_MASTER              (PI_CMD_WRITE1)0xae
#define PI_SET_GEAR_RATIO               (PI_CMD_WRITE2)0x14
#define PI_SET_INTEGRATION_LIMIT        (PI_CMD_WRITE2)0x95
#define PI_SET_INTERRUPT_MASK           (PI_CMD_WRITE1)0x2f
#define PI_SET_JERK                     (PI_CMD_WRITE2)0x13
#define PI_SET_KAFF                     (PI_CMD_WRITE1)0x93
#define PI_SET_KD                       (PI_CMD_WRITE1)0x27
#define PI_SET_KI                       (PI_CMD_WRITE1)0x26
#define PI_SET_KOUT                     (PI_CMD_WRITE1)0x9e
#define PI_SET_KP                       (PI_CMD_WRITE1)0x25
#define PI_SET_KVFF                     (PI_CMD_WRITE1)0x2b
#define PI_SET_LIMIT_SWITCH_MODE        (PI_CMD_WRITE1)0x80
#define PI_SET_MOTION_COMPLETE_MODE     (PI_CMD_WRITE1)0xeb
#define PI_SET_MOTOR_BIAS               (PI_CMD_WRITE1)0x0f
#define PI_SET_MOTOR_COMMAND            (PI_CMD_WRITE1)0x77
#define PI_SET_MOTOR_LIMIT              (PI_CMD_WRITE1)0x06
#define PI_SET_MOTOR_MODE               (PI_CMD_WRITE1)0xdc
#define PI_SET_OUTPUT_MODE              (PI_CMD_WRITE1)0xe0
#define PI_SET_POSITION                 (PI_CMD_WRITE2)0x10
#define PI_SET_POSITION_ERROR_LIMIT     (PI_CMD_WRITE2)0x97
#define PI_SET_PROFILE_MODE             (PI_CMD_WRITE1)0xa0
#define PI_SET_SAMPLE_TIME              (PI_CMD_WRITE1)0x38
#define PI_SET_SERIAL_PORT_MODE         (PI_CMD_WRITE1)0x8b
#define PI_SET_SETTLE_TIME              (PI_CMD_WRITE1)0xaa
#define PI_SET_SETTLE_WINDOW            (PI_CMD_WRITE1)0xbc
#define PI_SET_SIGNAL_SENSE             (PI_CMD_WRITE1)0xa2
#define PI_SET_STOP_MODE                (PI_CMD_WRITE1)0xd0
#define PI_SET_TRACE_MODE               (PI_CMD_WRITE1)0xb0
#define PI_SET_TRACE_PERIOD             (PI_CMD_WRITE1)0xb8
#define PI_SET_TRACE_START              (PI_CMD_WRITE1)0xb2
#define PI_SET_TRACE_STOP               (PI_CMD_WRITE1)0xb4
#define PI_SET_TRACE_VARIABLE           (PI_CMD_WRITE2)0xb6
```

```
#define PI_SET_TRACKING_WINDOW          (PI_CMD_WRITE1)0xa8
#define PI_SET_VELOCITY                 (PI_CMD_WRITE2)0x11
#define PI_WRITE_BUFFER                 (PI_CMD_WRITE3)0xc8
#define PI_WRITE_IO                     (PI_CMD_WRITE2)0x82
```

The types `PI_CMD_ONLY`, `PI_CMD_WRITE_READ1`, `PI_CMD_WRITE_READ2`, `PI_CMD_READ1`, `PI_CMD_READ2`, `PI_CMD_WRITE1`, `PI_CMD_WRITE2` and `PI_CMD_WRITE3` are all `int` and are used in this way as an aid memoir.

11.4 Utility Functions

11.4.1 Opening and initialising the board

Opening the board

The open board function comes in two version. Usually you will use the one with no argument, however, if you wish to access the device driver through another name (other than `/dev/pi_stage`) you may use the second version and specify the location of the device driver.

```
BOOL pi_openboard(void);
BOOL pi_openboard(char *);
```

Before any access to the board it must be opened using `pi_openboard` – failure to do this will (probably) result in a segmentation fault.
This function also initialised the board and resets it to default values by calling `pi_init_board();`.

11.4.2 Note on `pi_reset_board();` and `PI_RESET`

As of revision 0.60 this function no longer calls `pi_reset_board();`. This is thought to be unnecessary (although I think the Windows library does this by default). The `pi_reset_board();` is a special function that is executed by the kernel driver and should only need to be called at the installation of the driver (or if the card crashes but this should never happen). It has been removed so that the save and recall origin functions can work (`pi_reset_board();` sets the origin to 0 so that if it is in the call to `pi_init_board();` the origin cannot be statefully remembered by the card between opens).
This is experimental but not thought to have any consequences. As far as the user is concerned all it means is that in the event of a card crash (which never occurs) `pi_reset_board()` will need to be call explicitly whereas prior to version 0.60 just reopening the board would reset the card.
Calling `PI_RESET` through `pi_setQMC` or `pi_execute` just calls `pi_reset_board();`.
It is not necessary unless the card has crashed (which doesn't happen).
Return value: True on success, FALSE of failure.

Closing the board

`BOOL pi_closeboard(void);`
Return value: True on success, FALSE of failure.

Initialising the board

The board is automatically initialised when opened, no further action is needed. For documentation purposes `pi_openboard();` calls:
`BOOL pi_init_board(void);`
but you don't have to.
See section 11.4.2 to learn about `pi_reset_board();` which is no longer called since version 0.60.

11.4.3 Initialising Axis and limit switches

Getting the number of axis supported

To determine the number of axis supported use:
`int pi_axis_installed(void);`
Return value: the number of axis supported.
To initialise the axis use:
`BOOL pi_init_axis(PI_AXIS);`
This must be done before using the axis and is *not* done automatically.

Setting the limit switch sense

To set the limit switch sense automagically use:
`int pi_auto_set_limit(PI_AXIS);`
Return value: the limit switch sense (`PI_LIMIT_ZERO` or `PI_LIMIT_ONE`) on success or `PI_LIMIT_FAIL` on failure.
This function relies on the fact that both limit switches cannot be activated at the same time so it is safe to guess the sense and test it provided both read the same, if the limit switches disagree then one of them must be activated and the sense cannot be automatically determined. If you have a new type of stage where both switches can be activated at the same time then this function is unsafe. You can use this function to determine the usual state of the limit switches.
You will need to know what sense the limit switches are in the unfortunate event that you are on the limit switches when initialising. Without knowing the correct sense you can not move the stages safely and the automatic limit switch recover function may mistake the +ve limit for the -ve one and move the wrong way.
`void pi_set_limit(PI_AXIS,int);`
is the manual version of the limit switch sensing function, the second argument is either (`PI_LIMIT_ZERO` or`PI_LIMIT_ONE`).

Enabling or disabling the limit switches

`void pi_init_ls(PI_AXIS ,int);`
this function can enable or disable limit switch sensing. If you have no limit switches (say it is a circular stage) use it to turn limit switching off. Usually you do not need to turn it on, this is done automatically by `pi_openboard();` via `pi_init_board();`. It takes `PI_LIMIT_ON` and `PI_LIMIT_OFF`.

Moving off of the limit switches

`void pi_move_off_limit_switch(PI_AXIS);`
If the limit switch sense is correct this function will always move away from the activate switch. As of version 0.52 this function should move as far as required to get of the limit switch with only one call.

11.4.4 Finding the home position

With suitably equipped stages there is an encoder signal that can be used to define a "home" position roughly at the centre of the stages. The behaviour of this function is untested with stages that lack this functionality (by design the stages will move to seek this signal but will move in a random direction until they encounter a limit switch).
`void pi_find_home_(PI_AXIS);`
This finds the home (centre) position for the axis provided it is equipped with the `ENCODER_INDEX` signal. It will find the transition edge of this signal always moving in a +ve direction. Under tests this edge was found with a repeatability within $\pm 2\mu$m.
Warning This function has been tested with a PI stage M511.DD only. You are advised to examine the performance of this function with other stages under test conditions prior to use.
Warning The axis will move under its own control while this function is being used - you must take care to make sure it will not cause damage while seeking its home position.
Warning If the stage is not equipped with the `ENCODER_INDEX` signal it may move in a random direction until it hits a limit switch.
Warning The stage moves with the predetermined velocity - this function does not return until the home has been found - if the velocity is very low or 0 then it will take a long time (or never return). Set the velocity *before calling it.* A velocity of 0 is set following a motion error! This is a true of version 0.52 and may be changed in following versions.

11.4.5 Controlling the analogue motor and brake amplifiers

```
void pi_set_motor(PI_AXIS ,int);
void pi_set_brake(PI_AXIS ,int);
```

They take PI_MOTOR_ON, PI_MOTOR_OFF, PI_BRAKE_ON and PI_BRAKE_OFF as arguments. They can also control all axis at once using PI_MOTORS_ON, PI_MOTORS_OFF, PI_BRAKES_ON and PI_BRAKES_OFF as arguments (in this case the axis argument is ignored).
The motor and brake status can be inspected using:
`int pi_motor_and_brake_status(void);`
Return value: bit field indicating the status of the motor and brakes.
`int pi_get_motor(PI_AXIS);`
Return value: PI_MOTOR_ON or PI_MOTOR_OFF
`int pi_get_brake(PI_AXIS);`
Return value: PI_BRAKE_ON or PI_BRAKE_OFF

Waiting for axis to stop moving

To wait for the axes to stop moving you can use: `void pi_wait_stop(PI_AXIS);` This polls the card every 50ms and returns when the axis has stopped moving. For higher timing resolution poll `pi_moving()`.

11.4.6 Reading and writing the digital IOs

PI maps port 0 (zero) of the PMD motion chipset IO space to a digital output and a digital input. Reading this port returns the 8 bit value of the input, writing this port sets the output. Since the IO space is 16 bits and the IOs are 8 bit the top 8 bits are ignored.
The following functions are recommended for controlling the digital IOs that PI offers on their board. They offer stateful control (you can control each line independently without prior knowledge of the setting), however the output state is currently set to 0x00 when the library is initialised (this should be statefully handled in the kernel driver rather than the library so that settings are preserved - I will consider coding this as a feature request if asked, otherwise you will have to wait for me to get around to it).

```
// functions to handle the DIO
// turn a DO line on
void            pi_DO_up(unsigned char bit);
// turn a DO line off
void            pi_DO_down(unsigned char output);
// set the output port to a particular value
void            pi_DO_set(unsigned char data);
// convert from bit line to hex
unsigned char   pi_n2b(int n);
// read the input port
int             pi_get_DI();
```

Individual output bits can be set with `void pi_DO_up(unsigned char bit);` and `void pi_DO_down(unsigned char output);`. The utility function `pi_n2b();` is provided to convert from bit position to binary, hence to turn the third IO line

on you can use either `pi_DO_up(0x04)` or `pi_DO_up(pi_n2b(3));`. The whole output byte can be set with `pi_DO_set();`. To read the digital input you can use `pi_get_DI();`.
Please note that of version 0.50 May 2003 none of these functions has been tested and validated (just coded).

11.4.7 Stopping the stages quickly

`pi_stop_dead(PI_AXIS)` stops the axis by setting the target position to the current position. This is usually a safe way to abort any movement. Calling this function without and argument stops all the axis installed. This was introduced in version 0.54 and is untested.

11.4.8 Setting the apparent or reference position

```
void pi_set_pos(PI_AXIS, int pos);
```
Calls `PI_SET_ACTUAL_POSITION` which shifts the reference position so that the current report position reads `pos`. This should only be called when the stage is stationary as it may halt any movement. See the PMD programmers reference for an explanation of this (if you want).
This is useful for setting the home position of the stages. It is used with `pi_get_pos()` in the utility programs `pi_save_pos` and `pi_recall_pos`.

11.5 Functions which move the axis

These functions command the axis to move either to absolute positions (`_abs` versions) (relative to the encoder origin (which is zero on reset)) or relative positions (to the current location) (`_rel` versions). They are provided in 4 versions which start 1-4 axis simultaneously.

```
void pi_move_abs(PI_AXIS, int);
void pi_move_abs(PI_AXIS, int, PI_AXIS, int);
void pi_move_abs(PI_AXIS, int, PI_AXIS, int, PI_AXIS, int);
void pi_move_abs(PI_AXIS, int, PI_AXIS, int, PI_AXIS, int, PI_AXIS, int);
void pi_move_rel(PI_AXIS, int position);
void pi_move_rel(PI_AXIS, int, PI_AXIS, int);
void pi_move_rel(PI_AXIS, int, PI_AXIS, int, PI_AXIS, int);
void pi_move_rel(PI_AXIS, int, PI_AXIS, int, PI_AXIS, int, PI_AXIS, int);
```

11.5.1 Vector movement

```
void pi_vector_abs(PI_AXIS axis1, int p1, PI_AXIS axis2, int p2, int
velocity);
void pi_vector_rel(PI_AXIS axis1, int p1, PI_AXIS axis2, int p2, int
velocity);
```

These functions move two stages along a vector, they are supplied to emulate `VectorA(...)` and `VectorB(...)` in the Windows DLL. They should provide fairly good emulation with the following restriction: they do not restore the velocity of the stages after use, they return immediately after setting the stages in motion.

Beware!

The velocity is in *raw chipset units* of *counts per cycle*. Please note that this may change in future releases.

Take care!

You should also be aware that these functions, like their Windows counterparts, are very simple. They merely calculate the target velocity for each stage based on the velocity requested and the length of movement. The function then set the stage velocities and call `pi_move_rel(PI_AXIS, int, PI_AXIS, int);` (with the distances adjusted in the case of `pi_vector_abs`).
Thus the assumption is that both stages are identical and that the stage controller is set to give identical velocity profiles. If either of these assumptions are not correct then the movement may not follow a straight line. In fact the movement is not guaranteed to follow a straight line in any event. In practical use (with unloaded stages) it seems to work fairly well.

11.6 Functions which get information from the card

11.6.1 Determine if the axis is moving

```
BOOL pi_moving(PI_AXIS);
```
Return value: `TRUE` if moving, `FALSE` if stationary.

11.6.2 Finding the position of the stage

```
int pi_get_pos(PI_AXIS);
int pi_get_pos_err(PI_AXIS);
```
Return value: position (or position error) of the axis.

11.6.3 Finding the velocity of the stage (deprecated)

```
int pi_get_v(PI_AXIS);
```
Return value: the velocity is counts per second. This is approximate and low resolution as this is converted from counts per cycle.
This function is deprecated, please use `int pi_get_vel_cps(PI_AXIS);` instead.

11.6.4 Finding the velocity of the stage

`int pi_get_vel_cps(PI_AXIS);`
Return value: the velocity is counts per second. This is approximate and low resolution as this is converted from counts per cycle of the chipset.

11.6.5 Setting the velocity of the stage

`int pi_set_vel_cps(PI_AXIS);`
Sets the velocity in counts per second, it is approximate and low resolution as this is converted to counts per cycle of the chipset.

11.6.6 `pi_get_limit_status`

This function comes in two forms, with or without the axis.
`int pi_get_limit_status(PI_AXIS axis)` returns the limit status of the axis. It returns `PI_LIMIT_OK` if both limit switches are good and returns `PI_ON_POSITIVE_LIMIT` or `PI_ON_NEGATIVE_LIMIT` if it is on the +ve or -ve limit switches respectively.
`int pi_get_limit_status()` returns the limit status of the complete system. It returns `PI_LIMIT_OK` if all the limit switch's are good or returns `PI_LIMIT_BAD` is any of the limit switches are activated.

11.6.7 Inspect the motion controller registers

```
int pi_get_activity_status(PI_AXIS);
int pi_get_event_status(PI_AXIS);
int pi_get_signal_status(PI_AXIS);
```

Return value: the value of the register.
The following constants are defined in `pi_user.h`:

```
// bits in the event status register
#define PI_MOTION_COMPLETE              0x0001
#define PI_WRAP_AROUND                 0x0002
#define PI_BREAKPOINT1                  0x0004
#define PI_CAPTURE_RECEIVED             0x0008
#define PI_MOTION_ERROR                 0x0010
#define PI_EVENT_POS_LIMIT              0x0020
#define PI_EVENT_NEG_LIMIT              0x0040
#define PI_INSTRUCTION_ERR              0x0080
#define PI_COMMUNICATION_ERR            0x0800
#define PI_BREAKPOINT2                  0x4000

// bits in the signal status and signal sense registers
#define PI_ENCODER_A                    0x0001
#define PI_ENCODER_B                    0x0002
#define PI_ENCODER_INDEX                0x0004
```

```
#define PI_ENCODER_HOME                 0x0008
#define PI_SIGNAL_POS_LIMIT             0x0010
#define PI_SIGNAL_NEG_LIMIT             0x0020
#define PI_AXIS_IN                      0x0040
#define PI_HALL_A                       0x0080
#define PI_HALL_B                       0x0100
#define PI_HALL_C                       0x0200
#define PI_AXIS_OUT                     0x0400
// signal sense register only
#define PI_STEP_OUTPUT                  0x0800
#define PI_MOTOR_OUTPUT                 0x1000

//bits in the activity status register
#define PI_PHASING_INITD                0x0001  // (not PI boards)
#define PI_MAX_VELOCITY                 0x0002  // 1= max velocity
#define PI_TRACKING                     0x0004  // 1= in tracking
#define PI_PROFILE_MODE_0               0x0008  // 000=trap
#define PI_PROFILE_MODE_1               0x0010  // 001=v contour
#define PI_PROFILE_MODE_2               0x0020  // 010=s-curve, 011=e-gear
#define PI_RESERVED                     0x0040
#define PI_AXIS_SETTLED                 0x0080  // 1= settled
#define PI_MOTOR_STATUS                 0x0100  // 1= motor on
#define PI_POSITION_CAPTURE             0x0200  // 1= captured
#define PI_IN_MOTION                    0x0400  // 1= moving
#define PI_ACT_POS_LIMIT                0x0800  // +ve limit
#define PI_ACT_NEG_LIMIT                0x1000  // -ve limit
```

11.7 Asynchronous data capture

Note: It is harder to use the documentation for asynchronous data capture than it is to actually use asynchronous data capture. If you find the documentation difficult please try the example below and then refer back to the documentation. The card can generate interrupts on certain (programmed events). These events are captured by the driver which interrogates the card and saves the event. These events can be routed to the user program using the following functions:
`int pi_enable_async_capture(void (*)(int));`
`int pi_read_async_events(struct pi_stage_event_t *, int);`

`pi_enable_async_capture();` takes a function of type `void (*)(int)` as its argument (the event handler) this argument is a function that the user creates. This function is called whenever a `SIGIO` signal is raised. `pi_enable_async_capture();` sets the driver to route `SIGIO` signals to the user program. If you are not using any other asynchronous IO you never need to worry about the mechanics of this process, however, if you do use other asynchronous IO (possibly in an X windows program) then you will need to determine which file pointer generated

the event in which case please look at the library source code (you may want to contact the library author for advice or a feature request). The `int` argument to the event handler can be ignored, technically the event handler is a signal handler using the POSIX signal handling functions, for more information see `man sigaction`.
The events can be read using the `pi_read_async_events();` function, the first argument is a pointer to type `struct pi_stage_event_t` which is defined in the header files, the second argument is the maximum number of events that you can receive in one go.
The struct is currently defined as:

```
struct pi_stage_event_t{
        int index;
        char axis;
        unsigned long ts_high,ts_low;
        __u32 event;
        };
```

where `index` is the event index, several events of different axis that occurred together will share the same index otherwise this just counts up, `axis` is the axis that caused the event, `ts_high` and `ts_low` are time stamp information using the processor clock register, these are intended for debugging purposes (look up rdtsc for more information) and `event` is the contents of the event staus register after the event.
`index`, `ts_high` and `ts_low` are all likely to change in future versions.

11.7.1 Asynchronous data capture: example

```
#include <stdio.h>
#include <stdlib.h>
#include <pi_user.h>

void print_async_event(int);
main(){
int i,axis,sense;

// open and step the the board
        pi_openboard();
        axis=pi_axis_installed();
        for(i=0;i<axis;i++){
                pi_init_axis(i);
                sense=pi_auto_set_limit(i);
                }

// enable async events and link to print_async_event
        pi_enable_async_capture((void (*)(int))print_async_event);
```

```
// make the board send "motion complete" messages for all axis
        for(i=0;i<axis;i++){
                pi_setQMC(PI_SET_INTERRUPT_MASK,i,0x0001);
                }
        // this is where you normal code goes
        ....
        ..
        }

void print_async_event(int unused){
int n,i;
// structure fro receiving event data
struct pi_stage_event_t data[32];
// get the event data
        n=pi_read_async_events(data, 32);
// if there were no events we don't get anything so return, false alarm
        if(n==0) return;
        for(i=0;i<n;i++){
         printf("Async event %d: index=%d, axis=%d, timestamp=%ud, event= %x", \\
                        i,data[i].index,data[i].axis,data[i].ts_low,data[i].event);
                }
        }
```

This program (if completed) would print a message every time one of the axis stops moving. These event are received asynchronously via the POSIX signal handling interface.

Chapter 12

pi_execute(); PI compatibility functions

12.1 Introduction

PI's windows DLL library (and other products) define a set of functions that support the proprietary QFL library interface. A limited set of these functions are included here to provide compatibility with PI's existing library of products. Most of the commands defined in the QFL library interface are simply PMD motion chipset commands, a limited number (around 15) are very simple combinations of commands which are emulated by PI's library (and the `pi_execute` library) and finally a very limited number (< 5) are more complex commands which require more than 2 lines of C to code. Since the QFL interface does not provide any advantage over using the lower level (`pi_user` library I do not personally recommend it. Since PI requested this as a feature here it is, I do not endorse it (although it should work fine).

12.1.1 Implementing QFL function without the `pi_execute` library

If you want to implement the functionality of any QFL command but don't want to handle the report string (it is irritating converting to ASCII and back to a number) then use the pi_execute library source code to work out what the function does and implement it with the pi_user library.
The first place to look is the pi_execute.h header, this defines all the QFL constants in such a way as to make it easy to see what they do, for instance:

```
// commands that get something
#define PI_TY   PI_E_CR + PI_GET_QMC + PI_GET_VELOCITY
#define PI_VE   PI_E_CR + PI_GET_QMC + PI_GET_VERSION
```

```
// commands that set something (for execute)
#define PI_MUP  PI_E_CW + PI_SET_QMC + PI_MULTI_UPDATE
#define PI_SA   PI_E_CW + PI_SET_QMC + PI_SET_ACCELERATION

// more complex commands that are emulated by the library
#define PI_AB   PI_E_C + PI_LIB_FUN + 0x00      // calls fAB NOT PI_READ_ANALOG 0xef
#define PI_BN   PI_E_C + PI_LIB_FUN + 0x01      // brake on calls fBN
#define PI_BF   PI_E_C + PI_LIB_FUN + 0x02      // brake off calls fBF
```

Command that have `PI_GET_QMC` in their bit mask just call `pi_getQMC();`, commands that have `PI_SET_QMC` in their bit mask just call `pi_setQMC();`. The symbolic constant at the beginning indicates the amount of data that is passed. Commands that have `PI_LIB_FUN` are emulated by the library, usually by one or two lines of C-code which is defined in the source file pi_execute.cc which should be simple to read.

12.2 Building the `pi_execute` library

The `pi_execute` is automatically built and installed when you build and install the other library components.

12.3 Compiling with and using the `pi_execute` library

The `pi_execute` library is not linked automatically when you use the `pi_user` library. To link against it you must add the following option to the compile line `-lpi_execute` (see section 9.3 on page 24).

12.3.1 Dependencies

The `pi_execute` library is built on top of the `pi_user` library, however the `pi_user` can be used completely independently. Linking to the `pi_execute` library unnecessarily injures a small memory overhead of a few kbytes.
There is no function supplied by the `pi_execute` library which cannot be more efficiently and elegantly implemented with the `pi_user` library alone.

12.4 `pi_execute()`

`void pi_execute(PI_AXIS axis, PI_E_CMD cmd, int var, char * report);`
is pretty much a direct replacement for the PI DLL library function `execute();`.
It returns any argument in the report string in a similar format to `execute();`.
If the axis or variable argument is irrelevant it is ignored (to be compatible).

`#define PI_REPORT_SIZE 128` this constant is defined as the maximum size of report strings. Use it to define storage for the report string thus:
`char report[PI_REPORT_SIZE];`.

12.4.1 Differences between `pi_execute()` and `execute();`

The symbolic constants are different - if you use the constant from the DLL or other PI libraries then the behaviour is undefined. This is because the code is implemented in an entirely different way to provide faster, neater, simpler and smaller code.

Possible report string differences

There are some minor differences with the return strings (I think, since I don't have a working version of the DLL I have to guess at this).
The Linux version always returns arguments as signed ints which are 32 bits long and formatted in the report string appropriately (the DLL can return arguments as signed and unsigned ints (16 bits) or as sign longs (32 bits)). Since all ints under Linux are at least 32 bits long this distinction makes no sense in Linux and some minor formatting issues may result in the report string (all hex numbers will be in the format `0x00000000`).
The QFL library contains several repeated commands, these commands are functionally identical. In the Linux version these repeated commands are treated as aliases. If you use one of these aliases then the return string will report the name of the real command *not the alias*. The following commands are known to be aliased:

```
#define PI_TX   PI_GIP  // alias for GIP
#define PI_TA   PI_GA   // alias for GA
```

12.4.2 Simple pass through commands

These commands are simply handed off to the PMD chipset via pi_[gs]etQMC(...);
The definition indicates how many reads and writes the command takes to enable elementary error checking and make abuse of the command structure non-fatal.

```
// send commands with no data (defined for execute)
#define PI_CI   PI_E_C + PI_CLEAR_INTERRUPT
#define PI_NO   PI_E_C + PI_NO_OPERATION
#define PI_RT   PI_E_C + PI_RESET
#define PI_UP   PI_E_C + PI_UPDATE

// commands that get something but require an argument (defined for execute)
#define PI_GB   PI_E_WR + PI_GET_QMC + PI_GET_BREAKPOINT
#define PI_GBV  PI_E_WR + PI_GET_QMC + PI_GET_BREAKPOINT_VALUE
```

```
// command that get something (defined for execute)
#define PI_GA   PI_E_CR + PI_GET_QMC + PI_GET_ACCELERATION
#define PI_GAS  PI_E_CR + PI_GET_QMC + PI_GET_ACTIVITY_STATUS
#define PI_TP   PI_E_CR + PI_GET_QMC + PI_GET_ACTUAL_POSITION
#define PI_TV   PI_E_CR + PI_GET_QMC + PI_GET_ACTUAL_VELOCITY
#define PI_GSM  PI_E_CR + PI_GET_QMC + PI_GET_AUTO_STOP_MODE
#define PI_GAM  PI_E_CR + PI_GET_QMC + PI_GET_AXIS_MODE
#define PI_GIP  PI_E_CR + PI_GET_QMC + PI_GET_CAPTURE_VALUE
#define PI_GPP  PI_E_CR + PI_GET_QMC + PI_GET_COMMANDED_POSITION
#define PI_GPV  PI_E_CR + PI_GET_QMC + PI_GET_COMMANDED_VELOCITY
#define PI_TD   PI_E_CR + PI_GET_QMC + PI_GET_DECELERATION
#define PI_GES  PI_E_CR + PI_GET_QMC + PI_GET_EVENT_STATUS
#define PI_GGM  PI_E_CR + PI_GET_QMC + PI_GET_GEAR_MASTER
#define PI_GGR  PI_E_CR + PI_GET_QMC + PI_GET_GEAR_RATIO
#define PI_GSI  PI_E_CR + PI_GET_QMC + PI_GET_INTEGRAL
#define PI_GL   PI_E_CR + PI_GET_QMC + PI_GET_INTEGRATION_LIMIT
#define PI_GIA  PI_E_CR + PI_GET_QMC + PI_GET_INTERRUPT_AXIS
#define PI_GIM  PI_E_CR + PI_GET_QMC + PI_GET_INTERRUPT_MASK
#define PI_GJ   PI_E_CR + PI_GET_QMC + PI_GET_JERK
#define PI_GD   PI_E_CR + PI_GET_QMC + PI_GET_KD
#define PI_GI   PI_E_CR + PI_GET_QMC + PI_GET_KI
#define PI_GP   PI_E_CR + PI_GET_QMC + PI_GET_KP
#define PI_GF   PI_E_CR + PI_GET_QMC + PI_GET_KVFF
#define PI_GLM  PI_E_CR + PI_GET_QMC + PI_GET_LIMIT_SWITCH_MODE
#define PI_GMM  PI_E_CR + PI_GET_QMC + PI_GET_MOTOR_MODE
#define PI_GOM  PI_E_CR + PI_GET_QMC + PI_GET_OUTPUT_MODE
#define PI_TT   PI_E_CR + PI_GET_QMC + PI_GET_POSITION
#define PI_TF   PI_E_CR + PI_GET_QMC + PI_GET_POSITION_ERROR
#define PI_GPE  PI_E_CR + PI_GET_QMC + PI_GET_POSITION_ERROR_LIMIT
#define PI_GPM  PI_E_CR + PI_GET_QMC + PI_GET_PROFILE_MODE
#define PI_GT   PI_E_CR + PI_GET_QMC + PI_GET_SAMPLE_TIME
#define PI_GSS  PI_E_CR + PI_GET_QMC + PI_GET_SIGNAL_SENSE
#define PI_GTI  PI_E_CR + PI_GET_QMC + PI_GET_TIME
#define PI_TY   PI_E_CR + PI_GET_QMC + PI_GET_VELOCITY
#define PI_VE   PI_E_CR + PI_GET_QMC + PI_GET_VERSION

// commands that set something (for execute)
#define PI_MUP  PI_E_CW + PI_SET_QMC + PI_MULTI_UPDATE
#define PI_SA   PI_E_CW + PI_SET_QMC + PI_SET_ACCELERATION
#define PI_SSM  PI_E_CW + PI_SET_QMC + PI_SET_AUTO_STOP_MODE
#define PI_SAM  PI_E_CW + PI_SET_QMC + PI_SET_AXIS_MODE
#define PI_SCS  PI_E_CW + PI_SET_QMC + PI_SET_CAPTURE_SOURCE
#define PI_SD   PI_E_CW + PI_SET_QMC + PI_SET_DECELERATION
#define PI_SEM  PI_E_CW + PI_SET_QMC + PI_SET_ENCODER_MODULUS
#define PI_SGM  PI_E_CW + PI_SET_QMC + PI_SET_GEAR_MASTER
#define PI_SGR  PI_E_CW + PI_SET_QMC + PI_SET_GEAR_RATIO
```

```
#define PI_SIM  PI_E_CW + PI_SET_QMC + PI_SET_INTERRUPT_MASK
#define PI_DL   PI_E_CW + PI_SET_QMC + PI_SET_INTEGRATION_LIMIT
#define PI_SJ   PI_E_CW + PI_SET_QMC + PI_SET_JERK
#define PI_DP   PI_E_CW + PI_SET_QMC + PI_SET_KP
#define PI_DI   PI_E_CW + PI_SET_QMC + PI_SET_KI
#define PI_DD   PI_E_CW + PI_SET_QMC + PI_SET_KD
#define PI_DF   PI_E_CW + PI_SET_QMC + PI_SET_KVFF
#define PI_RES  PI_E_C + PI_SET_QMC + PI_RESET_EVENT_STATUS
#define PI_SB   PI_E_CW + PI_SET_QMC + PI_SET_BREAKPOINT
#define PI_SBV  PI_E_CW + PI_SET_QMC + PI_SET_BREAKPOINT_VALUE
#define PI_SLM  PI_E_CW + PI_SET_QMC + PI_SET_LIMIT_SWITCH_MODE
#define PI_SMO  PI_E_CW + PI_SET_QMC + PI_SET_MOTOR_COMMAND
#define PI_SMM  PI_E_CW + PI_SET_QMC + PI_SET_MOTOR_MODE
#define PI_SOM  PI_E_CW + PI_SET_QMC + PI_SET_OUTPUT_MODE
#define PI_SP   PI_E_CW + PI_SET_QMC + PI_SET_POSITION
#define PI_SPE  PI_E_CW + PI_SET_QMC + PI_SET_POSITION_ERROR_LIMIT
#define PI_SPM  PI_E_CW + PI_SET_QMC + PI_SET_PROFILE_MODE
#define PI_SST  PI_E_CW + PI_SET_QMC + PI_SET_SAMPLE_TIME
#define PI_SSS  PI_E_CW + PI_SET_QMC + PI_SET_SIGNAL_SENSE
#define PI_SV   PI_E_CW + PI_SET_QMC + PI_SET_VELOCITY
```

12.4.3 Simple emulated commands

These commands in the QFL command set are emulated by the library with very little code.

```
// calls pi_setQMC(PI_SET_STOP_MODE, axis, 1); pi_setQMC(PI_UPDATE, axis, 0);
#define PI_AB   PI_E_C + PI_LIB_FUN + 0x00

// calls pi_set_brake(axis,...);
#define PI_BN   PI_E_C + PI_LIB_FUN + 0x01
#define PI_BF   PI_E_C + PI_LIB_FUN + 0x02

// calls pi_DO_up(pi_n2b(...)); / pi_DO_down(pi_n2b(...));
#define PI_CN   PI_E_CW + PI_LIB_FUN + 0x03
#define PI_CF   PI_E_CW + PI_LIB_FUN + 0x04

// calls pi_setQMC(PI_SET_ACTUAL_POSITION,axis,0);
#define PI_DH   PI_E_C + PI_LIB_FUN + 0x05

// sprintf(report,"Linux V 0.5 May 2003");
#define PI_DLL  PI_E_C + PI_LIB_FUN + 0x06

// pi_setQMC(PI_SET_POSITION,axis,0); pi_setQMC(PI_UPDATE,axis);
#define PI_GH   PI_E_C + PI_LIB_FUN + 0x09
```

```
// pi_move_rel(axis,var); / pi_move_abs(axis,var);
#define PI_MR   PI_E_CW + PI_LIB_FUN + 0x10
#define PI_MA   PI_E_CW + PI_LIB_FUN + 0x11

// not yet supported - will call pi_move_off_limits();
#define PI_REC  PI_E_C + PI_LIB_FUN + 0x12

// pi_setQMC(PI_SET_STOP_MODE,axis,2); pi_setQMC(PI_UPDATE,axis);
#define PI_ST   PI_E_C + PI_LIB_FUN + 0x14

// returns pi_get_DI();
#define PI_TC   PI_E_CR + PI_LIB_FUN + 0x15

// returns pi_getQMC(PI_GET_ACTUAL_POSITION,axis)-pi_getQMC(PI_GET_POSITION,axis);
#define PI_TE   PI_E_CR + PI_LIB_FUN + 0x16

// usleep(1000*var);
#define PI_WA   PI_E_C + PI_LIB_FUN + 0x17

// temp=pi_n2b(var); while(!(pi_get_DI() & temp)) usleep(1000);
#define PI_WC   PI_E_C + PI_LIB_FUN + 0x18

// pi_wait_stop(axis);
#define PI_WS   PI_E_C + PI_LIB_FUN + 0x19      // calls sWS waits for axis to stop

// pi_wait_stop(axis); pi_DO_up(0x01);
#define PI_WT   PI_E_CW + PI_LIB_FUN + 0x20     // calls fWT (waitstop then setQMCA(WRITE_IO
```

12.4.4 Complex commands

These command take more than 2 lines of code to emulate.

```
#define PI_RET  PI_E_C + PI_LIB_FUN + 0x13      // calls fRET which calls:
                                                // setQMC(RESET_EVENT_STATUS, ax, 0);
                                                // gets and sets the velocity and position
                                                // setQMC(SET_MOTOR_MODE, ax, 1);
                                                // setQMC(UPDATE, ax, 0);
```

12.4.5 FEN, FEP and AutoFindEdge

These functions are not implemented, however, the function `pi_find_home()` will find the hardware defined home position with high accuracy.

```
// not supported yet
// Use pi_find_home(); or request a feature
```

```
#define PI_FEP  PI_E_C + PI_LIB_FUN + 0x07
#define PI_FEN  PI_E_C + PI_LIB_FUN + 0x08
```

12.4.6 Comment on selected commands

RET

RET is defined in the DLL / QFL section as "return to working conditions". Analysis of the code indicates that it is flawed and prone to an irritating bug. This bug is essentially reproduced in the Linux library for compatibility. RET causes the "Event Status Register" (ESR) for the axis to be reset in the PMD chipset. This would normally follow some sort of motion error that would cause an ESR exception. The PMD chipset usually stops the motion dead in the event of a potentially damaging exception. It can do this by (1) setting the velocity to 0 and (2) turning the motor off.
RET attempts to recover from this by (1) reseting the ESR, clearing the exception, (2) setting the desired position to the current position to prevent unwanted motion after recovery, (3) reseting the velocity to the preexception value and (4) turning the motors on. The bug here is that the preexception velocity is not statefully stored by the chipset, the driver or the library. It *can* be stored iff the driver object method "SetVelocity" is exclusively used to set the velocity but this is not the case throughout the driver and this is not documented. Thus the restored velocity can be an unknown following a call to RET.
The Linux driver does not attempt to fix this bug, instead is tries to harmlessly emulate it. Following a call to PI_RET / RET the velocity will be the same as it was before the call to RET (ie the original velocity or 0 depending on the state of the chip set). I recommend that whatever the driver that the velocity is statefully reset after calls to RET.

WA

The manual says, "Suspend command execution for n milliseconds. This command can be used in compound command structures to define absolute delay times", remember all the caveats you ever heard about real time processing - it all applies here. In the Linux driver this is merely implemented with a call to the standard library function `usleep()`. This is usually accurate to around $\frac{\pm 1}{HZ}$ in an unloaded system (ie 10 ms on an x86 box running a stock 2.4.xx kernel. The Windows library uses a similar technique.

WC

Sits in a loop until the digital input channel goes high, fits a usleep(1000) in the loop to delay / yield execution. You can write a much better version yourself.

WS

Just calls pi_wait_stop();, note that this cannot "be interrupted by the keyboard by pressing CNTRL and SHIFT at the same time" like the windows driver can. You can however stop / kill a program locked in a wait here by the usual methods (see man kill).

WT

Seems to be largely useless this function, waits for motion to stop then turns digital channel 1 (bit 0) on.

12.5 Direct emulation of the QFL command set

The `pi_execute` library defines all the QFL commands with a `PI_` prefix to prevent namespace pollution (to reduce the chance that the defined constants will conflict with another library). The facility to drop this prefix is supplied by defining the constant `PI_EXECUTE_DEFINES` before the header file is included. This then defines the 90 or so constants that make up the QFL command set exactly as PI do. For example:

```
#define PI_EXECUTE_DEFINES
#include <pi_execute.h>
```