`pysbig`
Python module for SBIG Camera
User's Manual

Release 0.8

Laurent Le Guillou

October 18, 2006

Mercator Telescope, P.O. Box 474, C/A. de Abreu 70, E-38700 Santa Cruz de La Palma, Spain

Legal Notice

CONTENTS

`pysbig` (“`sbig`”) is a Python extension to control SBIG CCD cameras. It has been developed in the framework of the HERMES spectrograph project to control the instrument guiding camera (based on a SBIG ST-1603ME), but may prove useful for other SBIG CCD cameras as well.

The present document is known as HERMES-WP-430-05 in the HERMES project documentation system.

CHAPTER

ONE

Introduction

This chapter introduces the `pysbig` Python extension and outlines the rest of the document.

This manual contains:

Installing pysbig Chapter 2 provides information on compiling and installing the `pysbig` extension from sources.

Overview Chapter 3 gives an overview of the `pysbig` module and provides step-by-step instructions to use it.

Functions Chapter 4 provides a detailed description of the `pysbig` functions.

Issues Chapter 5 discusses the limitations, bugs and issues of the current version of the `pysbig` module.

Where to get information and code

The `pysbig` Python extension has been written in the framework of the Hermes echelle-spectrograph project http://hermes.ster.kuleuven.be. This instrument, developped by a consortium lead by the Institute of Astronomy of the Katholieke Universiteit of Leuven (Belgium), will be installed on the Mercator telescope in La Palma (Canary Islands, Spain) in 2007. The guiding camera of the spectrograph is a SBIG ST-1603ME CCD camera for which we wrote the present software.

The source code can be obtained by contacting the HERMES project coordinators, Hans Van Winckel hans@ster.kuleuven.be and Gert Raskin gert@ster.kuleuven.be, or Laurent Le Guillou llg@lpnhep.in2p3.fr.

Please send comments and corrections to this manual to gert@ster.kuleuven.be or llg@lpnhep.in2p3.fr, or write to Mercator Telescope, Sea-level Office, P.O. Box 474, C/A. de Abreu 70, E-38700 Santa Cruz de La Palma, Canary Islands, Spain.

CHAPTER

TWO

Installation

This chapter explains how to install and test the Python pysbig module.

2.1 Testing the Python installation

The first step is to install Python if you haven't already. Python is available from the Python project page at http://sourceforge.net/projects/python. Click on the link corresponding to your platform, and follow the instructions described there. The Python pysbig module requires version Python 2.3 at a minimum. When installed, starting Python by typing **python** at the shell or double-clicking on the Python interpreter should give a prompt such as:

```
Python 2.3 (#2, Aug 22 2003, 13:47:10) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
```

If you have problems getting this part to work, consider contacting a local support person or emailing python-help@python.org for help. If neither solution works, consider posting on the comp.lang.python newsgroup.

Before proceding to the installation, please check that the numarray extension is present on your platform (to manipulate the produced CCD frames which are given as numarray arrays). The pyfits module is not strictly required, but it may prove useful to store frames as FITS files.

You may check if numarray has been installed on your platform by doing:

```
Python 2.3 (#2, Aug 22 2003, 13:47:10) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> import numarray
>>>
```

If the numarray module is absent you will obtain instead:

```
Python 2.3 (#2, Aug 22 2003, 13:47:10) [C] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> import numarray
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named numarray
>>>
```

In that case, you need to install the numarray extension before trying to install pysbig.

The pysbig module has been developped and tested on Debian GNU/Linux etch. It should work as well on other

Linux/UNIX platforms. For MS Windows platforms, some tests are probably needed.

2.2 Installing the SBIG low level driver

The Python module `pysbig` is based on the underlying SBIG low-level driver distributed by SBIG. The SBIG driver (with the associated firmware) and the SBIG libraries should be properly installed on your system before compiling the Python module. Please refer to the SBIG documentation [1] and to the HERMES-WP-430-04 Hermes Project document [2] when installing the SBIG low level driver.

2.3 Installation of `pysbig` from sources

To be able to compile the module, you will need the Python headers files. In some Linux distributions, they are distributed as a separate package (from the main Python packages) named “`python-dev`” or “`python2.3-dev`” (or something equivalent). On other Unix systems you may need to install the Python source tree before proceeding.

It may be necessary to define or modify your LD_LIBRARY_PATH in order to include the path for the needed dynamic SBIG libraries (‘libsbigcam.so’, ‘libsbigudrv.so’). Please refer to the HERMES-WP-430-04 document [2].

Once you have copied and unpacked the `pysbig` module source tree (with **tar**) from the tarball ‘sbig-X.X.tar.gz’, you can proceed with the compilation and the installation. Choose the destination directory prefix where you want to install the module (‘/usr/local’ is the default), and run:

```
python setup.py install --prefix=<PREFIX>
```

You may need to define or modify the PYTHONPATH environment variable to include ‘¡PREFIX¿/lib/python2.3/site-packages/’. Otherwise Python will not be able to find the `pysbig` module when you will try to load it.

To test if it works, just try to load the module:

```
# python
Python 2.3.5 (#2, Aug 30 2005, 15:50:26)
[GCC 4.0.2 20050821 (prerelease) (Debian 4.0.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sbig
>>> print sbig.__version__
’0.8’
```

You can now enjoy using the `pysbig` extension!

CHAPTER

THREE

Overview

The aim of this chapter is to give you a short overview of the main features of the `pysbig` extension. To do so, we will study an example of a typical Python script based on the `pysbig` extension (This script is available in the `pysbig` source distribution as 'examples/readout.py').

3.1 Initialisation

The script begins with the usual extension imports, including, of course, the import of the `pysbig` extension.

```
#! /usr/bin/env python
import os, sys, time
import sbig                    # this module
import pyfits                  # only for FITS output
```

To send commands to the SBIG CCD camera, the first step is to initialise the SBIG driver with the `open_driver` function. If a problem occurs a `SBIGDriverError` exception will be raised.

```
try:
    sbig.open_driver()
except sbig.SBIGDriverError, msg:
    print >>sys.stderr, msg
    sys.exit(2)
```

Once the driver is opened, you may send a request to the USB bus to identify all the SBIG USB cameras connected to the system with the `query_usb` function.

```
try:
    devices = sbig.query_usb() # return list of USB SBIG cameras found
except sbig.SBIGDriverError, msg:
    print >>sys.stderr, msg
    sys.exit(2)
```

In the current version of the SBIG library, this function is buggy and will only work once; each subsequent call will return an empty list of devices, whenever one or several SBIG cameras are connected to the USB ports. It is necessary to close the driver and open it again (and sometimes even to power cycle the camera) to obtain the correct list of USB connected devices.

```
try:
    sbig.open_device(sbig.DEV_USB)
except sbig.SBIGDriverError, msg:
    print >>sys.stderr, "Unable to open SBIG camera device"
    print >>sys.stderr, "Error: ", msg
    sys.exit(3)
```

Once the device has been opened, the communication link to the camera should be established. It is often needed to repeat several times the call to the `establish_link` function to establish the link (advice of Jan Soldan, co-maintainer of the SBIG low level driver). A typical program using the SBIG camera can initialise the link like this:

```
tries=5
camera_type = sbig.NO_CAMERA
for i in xrange(tries):
    try:
        camera_type = sbig.establish_link()
        break
    except sbig.SBIGDriverError, msg:
        print >>sys.stderr, msg, "Trying again..."
        pass
if (camera_type == sbig.NO_CAMERA):
    print >>sys.stderr, "Cannot establish link with the camera."
    sys.exit(4)
```

3.2 CCD camera details

Once the link with the camera has been established, it is possible to get precise informations about the CCD model (size, gain), and the available readout modes with that model, by using the `get_ccd_info` function.

```
ccd_info = sbig.get_ccd_info()
```

It returns a dictionary with this structure:

```
ccd_info = { "firmware": firmware,
             "type":     cameraType,
             "name":     cameraName,
             "modes":    { mode_id: mode_info, mode_id: mode_info, ... } }
```

where *firmware*, *cameraType* and *cameraName* are identifiers of the firmware and the camera model, and *modes* a dictionary where the keys are the numeric idenfiers of the available CCD readout modes. For each readout mode in that dictionary, the *mode_info* description (i.e. the value in the dictionary) is also a dictionary structure:

```
mode_info = { "width":        width,
              "height":       heigth,
              "gain":         gain,
              "pixel_width":  pixel_width,
              "pixel_height": pixel_height }
```

where *width* and *height* are the frame dimensions (in pixels) corresponding to the binning of that particular readout mode, *gain* the associated CCD gain (in electron/ADU), and *pixel_width*, *pixel_heigth* the physical pixel dimensions on the CCD chip for that binning in $\mu\mathrm{m}$.

With this structure, you may obtain whichever parameter you need for a given readout mode. For example, the CCD width (in pixels) in the read out mode #3 is:

```
width = ccd_info["modes"][3]["width"]
```

Read out parameters are useful to specify the CCD region you want to get during CCD read out (See 3.2).

3.3 Temperature control

In many SBIG cameras, a Peltier junction is available to cool down the CCD and thus diminish the thermal noise. The following part of the script first activates the cooling system, then waits until the CCD temperature (here 15 °C) reaches the set point value. Once the target temperature is reached, the Peltier voltage will be modulated to keep the CCD temperature stable.

When the temperature regulation is activated, a sudden change in the Peltier voltage may increase the noise if it happens during a frame readout. Therefore, in order to minimize the readout noise, it is possible to automatically "freeze" the temperature regulation process during each CCD readout (see line 51).

```
34  # Enable temperature regulation at 15 C
35  setpoint = 15.0
36  sbig.set_temperature_regulation(sbig.REGULATION_ON, setpoint_celsius=setpoint)
37
38  # Wait until the temperature reach the setpoint value (+/- 1 C)
39  while True:
40      temp_status = sbig.query_temperature_status()
41      print "ccd_celsius = ", temp_status['ccd_celsius']
42      print "setpoint = ", temp_status['setpoint_celsius']
43      if abs(temp_status['ccd_celsius'] -
44             temp_status['setpoint_celsius']) < 1.0:
45          break
46      time.sleep(1)
47
48  # Regulation will be "frozen" during readout to minimize readout noise.
49  # (Freeze will start at the first readout_line command and stop
50  # when the end_readout command will be sent.)
51  sbig.set_temperature_regulation(sbig.REGULATION_ENABLE_AUTOFREEZE)
```

3.4 Exposure

After all these initialisation steps, it is time to take an image with the camera. The `start_exposure` function will opens the shutter for the specified duration. It may be needed to indicate which CCD you are using, as it may be the main CCD of the camera or the guiding CCD (if the SBIG model you are using has one, of course).

The `start_exposure` function returns immediatly after opening the shutter. Therefore you have to check yourself if the exposure is finished or not, by using the `query_command_status` which return the status of the last command sent to the camera. The status is `STATUS_INTEGRATING` during the exposure, and `STATUS_COMPLETE` when the integration is achieved.

Once the exposure is done, a call to the `end_exposure` is needed before proceeding with the readout.

```
52  sbig.start_exposure(ccd=sbig.CCD_IMAGING, exposure=10.0)
53
54  # Waiting until the end of the exposure (by checking status)
55  # sbig.query_command_status() gives status of the last command
56
57  while (sbig.query_command_status() == sbig.STATUS_INTEGRATING):
58      time.sleep(0.1)
59
60  # ending exposure (can also be used to end it prematurely)
61  sbig.end_exposure(ccd=sbig.CCD_IMAGING)
```

3.5 CCD frame readout

Once the exposure is done, it is time to read out the CCD to get the frame. Depending of the SBIG camera model, several readout modes may be available, with different binning and therefore, different apparent CCD dimensions. For a given mode, you may obtain the relevant information by using `get_ccd_info` (See 3.2 and 4.2). The readout process should be initiated by the `start_readout` function, it is necessary to specify the CCD you will use (the main one or the guiding one), the readout mode and the dimension of the window you want to read. Then, you may read the frame row by row with `readout_line`, or in one shot with `readout_lines`. These functions return respectively a 1-dimensional and 2-dimensional `numarray` array. To prepare the camera for the next exposure, you need to terminate the readout with `end_readout`.

```
mode=0  # readout mode (binning 1x1 i.e. not binned)
fullheight = ccd_info['modes'][mode]['height']
fullwidth = ccd_info['modes'][mode]['width']
height=fullheight
width=fullwidth
top=0
left=0
sbig.start_readout(ccd=sbig.CCD_IMAGING, mode=mode,
                   top=top, left=left,
                   height=height, width=width)

# You may read row by row with readout_line(...),
# or a block with readout_lines(...)
# image is a numarray 2D-array.
image = sbig.readout_lines(ccd=sbig.CCD_IMAGING, mode=mode,
                           start=left, length=width, lines=height)
sbig.end_readout(ccd=sbig.CCD_IMAGING)
```

The parameters *ccd*, *mode*, *top*, *left*, *height*, *width*, *start*, *length* and *lines* should keep coherent values through the call to `start_readout` and subsequent calls to `readout_lines` or `readout_line`.

If you plan to read out a window instead of a full CCD frame, please read section 5.2 first, as the current `pysbig` version has some limitations.

3.6 Saving the frame as a FITS file

When the readout is achieved, you may do whatever postprocessing you want on the resulting frame stored as a `numarray` array. If you would like to save it into a FITS file (FITS is the astronomical standard file format, see [7]), you will need the `pyfits` extension to manipulate FITS files. Here is a way to do it:

```
filename = "out.fits"
fitsobj = pyfits.HDUList()
hdu = pyfits.PrimaryHDU()
# scale the data to Int16 with user specified bscale/bzero
# no other operation should manipulate the data after this
# hdu.data = image
hdu.data = image*1.0  # seems needed for the next operation
hdu.scale('Int16', '', bzero=32768)
fitsobj.append(hdu)
try:
    os.unlink(filename)
except OSError:
    pass
fitsobj.writeto(filename)
```

You may notice that the data array has been scaled to signed integers. This is due to the fact that pixels in the frame have values between 0 and 65535 (unsigned integer), but unsigned integer FITS image are not allowed by the FITS standard [7]. The way to solve this is to scale the pixel values between -32768 and +32767, and to set the two FITS keywords `BSCALE=1.0` and `BZERO=32768`. That way, any software reading the FITS file will use these keywords to translate the pixel values stored into the original ones (See [7] and [6] for details).

3.7 End of the program

When you do not intend to use the SBIG driver anymore (typically at the end of a program), you need to close the device, and to close the driver before the end of the Python script.

```
sbig.close_device()
sbig.close_driver()
```

3.8 Exceptions

The low level SBIG library returns error codes when an error occurs. In the `pysbig` Python extension, these error codes have been replaced by the Python exception named `SBIGDriverError`. This exception is raised each time the low level SBIG driver returns an error; the associated exception message gives details about what happens. Depending of what you want to do, you may catch these exceptions and manage the problem from within your code, or let the exception reach the calling function.

CHAPTER

FOUR

Functions

4.1 Initialisation and driver control

To initialise the driver and establish the link with the SBIG camera, a few operations are needed. The driver and the camera device should be opened, and the communication established before any exposure can be taken. When the CCD camera is no more in use (typically when the program ends), the device and the driver should be closed. A typical program will do something like this:

```
>>> import sbig
>>> sbig.open_driver()
>>> sbig.open_device(sbig.DEV_USB)
>>> sbig.establish_link()
>>> ...
>>> ... # Taking many exposures
>>> ...
>>> sbig.close_device()
>>> sbig.close_driver()
```

The initialisation functions are described here, as well as a few other functions that may prove useful during the initialisation.

open_driver()

open_driver initialises the SBIG driver. It returns None. If the SBIG driver cannot be found on the system, a SBIGDriverError exception will be raised.

close_driver()

close_driver closes the SBIG driver. It returns None. If the SBIG driver has not been opened first (with open_driver), a SBIGDriverError exception will be raised.

open_device(*device=DEV_USB*, *lpt_base_address=0x378*, *ip_address=0*)

open_driver initialises the SBIG driver. It returns None. If the SBIG driver cannot be found on the system, a SBIGDriverError exception will be raised.

The *device* parameter should be one of the constants given in table 4.1.

Depending of the camera device (USB, LPT, Ethernet), some extra parameters may be needed: the I/O port address *lpt_base_address* for the parallel port (0x378 for LPT1, 0x278 for LPT2), or the IP address *ip_address* for the ethernet port.

If the device cannot be opened (no device, permission denied, etc.), a SBIGDriverError exception will be raised.

close_device()

close_device closes the opened camera device. It returns None. If the camera device has not been opened first (with open_device), a SBIGDriverError exception will be raised.

SBIG Constant	device
`DEV_NONE`	No device
`DEV_LPT1`	1st parallel port (LPT1)
`DEV_LPT2`	2nd parallel port (LPT2)
`DEV_LPT3`	3rd parallel port (LPT3)
`DEV_ETH`	Ethernet port
`DEV_USB1`	1st USB port
`DEV_USB2`	2nd USB port
`DEV_USB3`	3rd USB port
`DEV_USB4`	4th USB port
`DEV_USB`	Generic USB port

Table 4.1: *SBIG device constants*

`establish_link``()`

Once the device has been opened, it is necessary to establish the connection with it, with the `establish_link` function. It returns a constant representing the camera model (for instance, `sbig.ST402_CAMERA=16` for ST402-based cameras like the ST-1603ME one). If the link cannot be established, a `SBIGDriverError` exception will be raised.

The current version of the SBIG library has a bug: the `establish_link` function can fail, and it may be needed to call it several times to establish the connection with the device. A typical program may do something like this:

```
tries=5
print "Establish the link with the camera..."
camera_type = sbig.NO_CAMERA
for i in xrange(tries):
    try:
        camera_type = sbig.establish_link()
        break
    except sbig.SBIGDriverError, msg:
        print >>sys.stderr, msg, "Trying again..."
        pass

if (camera_type == sbig.NO_CAMERA):
    print >>sys.stderr, "Cannot establish link with the camera."
    sys.exit(2)
```

`get_link_status``()`

`get_link_status` Returns the status of the link established with the camera. The returned object is a dictionary with this structure:

```
status = { "established":      linkEstablished,
           "lpt_base_address": baseAddress,
           "type":             cameraType,
           "total":            comTotal,
           "failed":           comFailed }
```

where *linkEstablished* is an integer which is 1 if the communication link is still established with the CCD camera, and 0 otherwise; *baseAddress* is the memory address associated with the LPT port (if you are using the parallel port, otherwise this parameter is meaningless); *cameraType* is the camera identifier (see `establish_link` above). The two last parameters *comTotal* and *comFailed* are the total number of communications with the camera, and the number of failed communications respectively. These informations may help to identify a cable problem for instance.

`query_command_status``()`

`query_command_status` is used to monitor the status of the last command sent to the SBIG driver. The status returned is one of the constants of table 4.2.

SBIG Status Constant	Meaning
`STATUS_IDLE`	Camera ready
`STATUS_IN_PROGRESS`	Last command is beeing executed
`STATUS_INTEGRATING`	Integration (exposure) in progress
`STATUS_COMPLETE`	Integration (exposure) achieved

Table 4.2: *SBIG status constants*

For instance, `query_command_status` is useful to check if an exposure is achieved or not (see 3.4 for an example).

`query_usb``()`

`query_usb` queries the USB bus and detect up to 4 USB SBIG cameras. It returns a tuple: each element of the tuple is a dictionary associated with the corresponding USB camera. The keys are `'device'` for the camera device (See `DEV_*` constants in table 4.1), `'serial'` for the serial number, `'type'` for the camera model id (See `*_CAMERA` constants), and `'name'` for the model name.

```
>>> import sbig
>>> sbig.open_driver()
>>> devices = sbig.query_usb()
>>> print devices
({'device': 32514, 'serial': '051100736',
  'type': 16, 'name': 'SBIG ST-1602 CCD Camera'},)
>>> print "%d USB camera(s) detected." % len(devices)
1 USB camera(s) detected.
>>> print "camera #0 name:", devices[0]['name']
camera #0 name: SBIG ST-1602 CCD Camera
>>> print "camera #0 device:", devices[0]['device']
camera #0 device: 32514
>>> print sbig.DEV_USB1
32514
```

There is a bug in the SBIG library: this function will return the correct tuple the first time you use it after you turn the power on, and will always return an empty tuple `()` after.

Anyway, you may always call the function `open_device` with *device* set to the generic USB device constant `DEV_USB`: the driver will open the first camera found on the USB bus. That way, a call to the `query_usb` function is not needed. In fact this function is only useful if you have several USB cameras connected on the same computer.

`get_driver_handle``()`

`set_driver_handle``(`*handle*`)`

Functions `get_driver_handle` and `set_driver_handle` are useful if you plan to manage several SBIG cameras on various ports at the same time. If your program will only talk with one camera, you may ignore these two functions.

For more details about these two functions, please see the SBIG library documentation [1].

`get_driver_info``(`*request=*`0)`

`get_driver_info` returns the version and capabilities of the driver, as a dictionary. Keys are `'version'` for the driver version, and `'name'` for the driver name. If *request* is `0` (the default), the function returns informations about the high level SBIG driver; if *request* is set to `1`, data about the low level LPT or USB driver are returned instead. See [1] for other requests.

```
>>> sbig.get_driver_info()
{'version': '04.43', 'name': 'libsbigudrv Ver 4.43-LINUX',
 'max_request': 1}
```

4.2 Exposure and Readout

This section describe the `pysbig` functions needed to manage CCD exposure and readout. All these functions may be used as well for the main CCD of the camera, for the internal tracking CCD (if the camera has one) or for an external tracking CCD connected to the SBIG camera. When using one of these functions, the choice of the CCD is made through the *ccd* parameter, which can take 3 values: `CCD_IMAGING` for the main imaging CCD, `CCD_TRACKING` for the internal tracking CCD, and constantCCD_EXT_TRACKING for an external tracking CCD.

When taking frames with the CCD, you need to start the exposure, wait, end it, initialise the readout by defining the readout mode and the CCD area to be read, read the frame row by row or in one shot, and finally end the readout. A typical program will do something like this:

```
>>> ...
>>> sbig.start_exposure(sbig.CCD_IMAGING, 10.0)
>>> time.sleep(10.0)  # There is a better way to do this, see section 3.5
>>> sbig.end_exposure(sbig.CCD_IMAGING)
>>> sbig.start_readout(sbig.CCD_IMAGING, mode, top, left, height, width)
>>> image = sbig.readout_lines(sbig.CCD_IMAGING, mode,
                               start=left, length=width, lines=height)
>>> sbig.end_readout(sbig.CCD_IMAGING)
>>> ...
```

Each SBIG camera model has several readout modes, with different binning (1×1, 3×3, and so on). The available modes may differ from one model to another. To know the available readout modes of your SBIG camera and the corresponding CCD dimensions in each mode, see the `get_ccd_info` function below.

`start_exposure``(`*ccd, exposure, antiblooming=*`ABG_LOW7`*, shutter=*`SC_OPEN_SHUTTER`)

`start_exposure` initiates an exposure of duration *exposure* in seconds. The control is immediatly given back to the caller. Therefore the caller has to check whether the exposure is achieved or not by using the `query_command_status` function (see 4.1 and 3.4 for an example).

By default, the shutter will open at the beginning of the exposure and close at the end of the specified duration. If you want a different shutter behavior (for biases or dark frames for instance), you can set the *shutter* parameter to one of the values of table 4.3.

SBIG Shutter Constant	Meaning
`SC_LEAVE_SHUTTER`	Leave shutter alone
`SC_OPEN_SHUTTER`	Open shutter for exposure and close for readout
`SC_CLOSE_SHUTTER`	Close shutter during exposure and readout

Table 4.3: *SBIG shutter constants*

For details about "antiblooming", please see [1].

`end_exposure``(`*ccd*`)`

`end_exposure` has to be used when the exposure is complete in order to prepare the CCD for readout. It may also be used to terminate the exposure prematurely.

start_readout(*ccd, mode, top, left, height, width*)

`start_readout` initiates the CCD readout; it defines the area you intend to readout in subsequent `readout_line` or `readout_lines` calls. The area is defined by the *top*, *left*, *height* and *width* parameters (see figure 4.1). These parameters should be compatible with the readout mode you chose (see function `get_ccd_info` below).

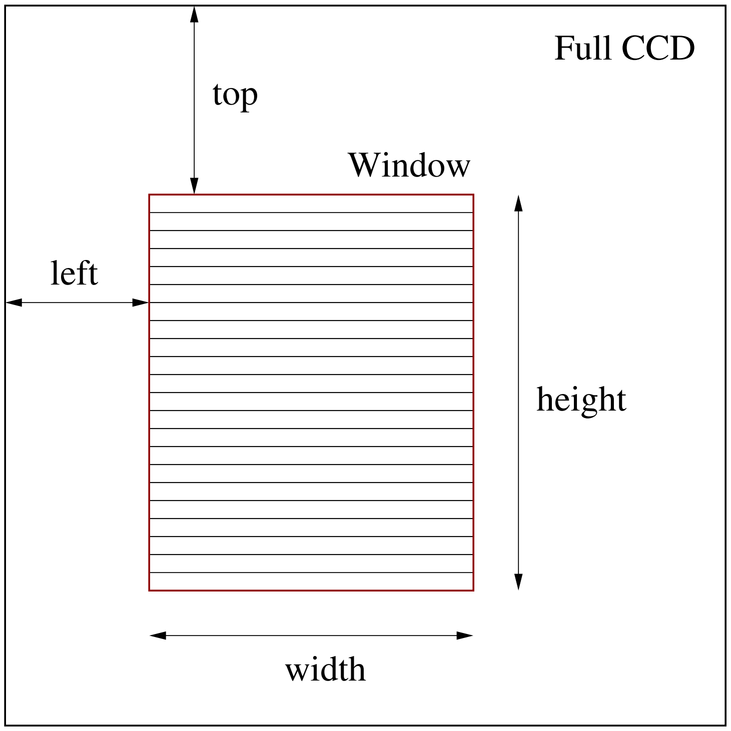

Figure 4.1: *CCD Window readout parameters*

readout_line(*ccd, mode, start, length*)

`readout_line` reads one row of the CCD. In a typical readout, the *start* parameter should be equal to the *left* parameter used for the last call to `start_readout`, and *length* should be equal to *width*. `readout_line` returns a 1-dimensional `numarray` array of size *length*.

This function reads only one row of the area you defined. To read several rows, you need to repeat it (typically *height* times). You may also prefer to use `readout_lines` (see below).

readout_lines(*ccd, mode, start, length, lines*)

`readout_lines` reads several CCD rows at once. In a typical CCD readout, *start* should be equal to the *left* parameter used for the last call to `start_readout`, *length* should be equal to *width*, and if you want to read the full area at once, *lines* should be equal to *height*. `readout_lines` returns a 2-dimensional `numarray` array of shape (*length*, *lines*). This array is perfectly suited to be saved into a FITS file with the `pyfits` extension (See 3.6 for an example).

dump_lines(*ccd, mode, lines*)

`dump_lines` discards entire rows during readout. This function is useful to speed up the readout when doing a partial CCD readout (window), by discarding the rows above the CCD area you would like to read.

end_readout(*ccd*)

`end_readout` prepares the CCD for idle state after readout. That way the CCD is ready for the next exposure.

get_ccd_info(*request=`CCD_IMAGING`*)

Depending of the *request* parameter, `get_ccd_info` may return different type of information about the camera CCD (the imaging CCD or the tracking one).

If *request* is `CCD_IMAGING` or `CCD_TRACKING`, `get_ccd_info` returns details about the CCD, and a list of all available readout modes. These informations are given as a dictionary with the following structure:

```
ccd_info = { "firmware": firmware,
             "type":     cameraType,
             "name":     cameraName,
             "modes":    modes = { mode_id: mode_info, mode_id: mode_info, ... } }
```

where *firmware*, *cameraType* and *cameraName* are identifiers of the firmware and the camera model, and *modes* a dictionary where the keys are the numeric idenfiers of the available CCD readout modes. For each readout mode in that dictionary, the *mode_info* description (i.e. the value in the dictionary) is also a dictionary structure:

```
mode_info = { "width":        width,
              "height":       heigth,
              "gain":         gain,
              "pixel_width":  pixel_width,
              "pixel_height": pixel_height }
```

where *width* and *height* are the frame dimensions (in pixels) corresponding to the binning of that particular readout mode, *gain* the associated CCD gain (in electron/ADU), and *pixel_width*, *pixel_heigth* the physical pixel dimensions on the CCD chip for that binning in $\mu\mathrm{m}$.

With this data structure, you may obtain whichever parameter you need for a given readout mode. For example, the CCD dimensions (in pixels) in the read out mode #0 are:

```
ccd_info = sbig.get_ccd_info()
mode=0  # readout mode (binning 1x1 i.e. not binned)
fullheight = ccd_info['modes'][mode]['height']
fullwidth = ccd_info['modes'][mode]['width']
gain = ccd_info['modes'][mode]['gain']
```

The *request* parameter may also take other values to get very specific informations. See [1] and source file 'sbig.c' for details.

`read_offset`(*ccd*)

`read_offset` returns the CCD's offset. The offset is adjusted at the factory and cannot be modified.

4.3 Temperature regulation

The commands in this section are used to control or monitor the CCD's temperature regulation. SBIG parallel port cameras have two temperature semsors, one for the CCD and the other for the ambient temperature; on USB based SBIG cameras, only the CCD sensor is present, and reading the ambient temperature will always return 25^{o}C.

`query_temperature_status`()

`query_temperature_status` is useful to monitor the CCD's temperature and the temperature regulation. It returns a dictionary with the following structure:

```
status = { "enabled":          enabled,
           "frozen":           frozen,
           "setpoint_ad":      setpoint_ad,
           "setpoint_celsius": setpoint_celsius,
           "ccd_ad":           ccd_ad,
           "ccd_celsius":      ccd_celsius,
           "ambient_ad":       ambient_ad,
           "ambient_celsius":  ambient_celsius,
           "power":            power }
```

In this structure, *enabled* is 1 if the temperature regulation is on and 0 otherwise; *frozen* is 1 if the regulation will be automatically frozen during CCD readout (see below and in section 3.3 for an example) and 0 if not. *setpoint_** is the CCD setpoint value, *ccd_** is the measured CCD temperature, and *ambient_** the ambient temperature (meaningless for USB based cameras). For these 3 last parameters, **_celsius* is the corresponding value in celsius, and **_ad* the same value expressed in internal A/D units. For more details about the A/D conversion see [1]. *power* is the electric power applied to the cooler, expressed in the range 0–255.

`set_temperature_regulation``(`*regulation, setpoint_celsius, setpoint_ad, power*`)`

`set_temperature_regulation` is used to enable or disable the CCD temperature regulation, and to set the associated parameters, like the temperature set point or the cooler power.

If *regulation* is `REGULATION_ON`, the cooler is activated, and the regulation will start if it is not already started. You need to provide the CCD temperature setpoint, either in Celsius (by specifying the *setpoint_celsius* parameter) or in internal A/D units (with the *setpoint_ad* parameter). If you do not provide one of these two parameters, a `ValueError` exception will be raised.

If *regulation* is `REGULATION_OFF`, the cooler is turned off.

If *regulation* is `REGULATION_FREEZE`, the regulation will be frozen.

If *regulation* is `REGULATION_UNFREEZE`, the regulation will be unfrozen.

If *regulation* is `REGULATION_ENABLE_AUTOFREEZE`, the regulation will be automatically frozen during each CCD readout, to avoid changes of the cooler power which may increase the readout noise during readout. The regulation will be unfrozen after each readout.

If *regulation* is `REGULATION_DISABLE_AUTOFREEZE`, the automatic regulation freeze during readout will be disabled.

If *regulation* is `REGULATION_OVERRIDE`, you can override the temperature regulation process and set yourself the cooler power with the *power* parameter. The *power* should be in the range 0–255. If you do not provide the *power* parameter, a `ValueError` exception will be raised.

4.4 Control parameters

`get_driver_control``(`*parameter*`)`

`set_driver_control``(`*parameter, value*`)`

These two functions are useful to get or modify the internal control parameters of the SBIG driver. Some driver options may be enabled or disabled that way. Be careful when modifying one of the driver control parameters as this may produce unexpected results. The available control parameters are listed in table 4.4.

SBIG Control Parameter	Meaning	Default
`DCP_USB_FIFO_ENABLE`	Enable FIFO for USB cameras	`True`
`DCP_CALL_JOURNAL_ENABLE`	Broadcast driver API calls (debug)	`False`
`DCP_IVTOH_RATIO`	Dump row speed (do not modify)	`5`
`DCP_USB_FIFO_SIZE`	Size of the USB FIFO	`16384`
`DCP_USB_DRIVER`	(unknown)	
`DCP_KAI_RELGAIN`	(unknown)	
`DCP_USB_PIXEL_DL_ENABLE`	Enable downloading of pixels data (debug)	`True`
`DCP_HIGH_THROUGHPUT`	Enable fastest transfer (high noise)	`False`
`DCP_VDD_OPTIMIZED`	Lower CCD voltage for short exposures	`True`
`DCP_AUTO_AD_GAIN`	Autoset of A/D gain is USB cameras	`True`

Table 4.4: *SBIG control parameter constants*

For more details please refer to [1].

4.5 External devices: filter wheel, mirror tip-tilt, etc.

On some SBIG cameras, it is possible to control external devices, like the tip-tilt of a secondary mirror for instance. Some SBIG models also include a filter wheel. The library distributed by SBIG includes specific functions to control these associated components. For the moment, the corresponding functions have not not yet been implemented in `pysbig`. They will be added in future versions.

CHAPTER

FIVE

Issues

5.1 Array support

The current version of the `pysbig` module is using `numarray` arrays to store the CCD data obtained during a readout [4]. The `readout_line` function returns a 1D `numarray` array, while `readout_lines` returns a 2D `numarray` array.

The Python array support is evolving quite fast, and it is scheduled in the near future to phase out the two conflicting array packages `Numeric` and `numarray`, to replace them by a unified array module, `numpy` [5]. For the moment (September 2006), the `numpy` module is not mature enough, and important packages like `pyfits` (FITS file support, see [6]) are still based upon `numarray`. It is why we chose to write the `pysbig` module with a `numarray` interface. However, it will soon be needed to replace the `numarray` interface by its `numpy` equivalent, to follow the evolution of the Python language. As the authors of `pyfits` also plan to switch their modules to `numpy`, the `pysbig` module will need to be updated when `pyfits` will be, to keep compatibility.[1]

5.2 Known bugs

Here are some known bugs:

- `query_usb` will returns the list of the cameras found on the USB bus only once. After it will always return an empty tuple (See 4.1). A power off seems to be needed to get again the correct tuple.
- The window readout procedure described in [1] does not work and will produce `RX Timeout` errors. The data will also be corrupted. Even if you plan to do a window readout, use the parameters for a full CCD readout, and always read the entire rows. To speed up the readout you may just dump the rows at the top of your window with `dump_lines` before using `readout_lines`. It is not necessary to read or dump the remaining rows at the bottom. Please refer to the 'readout-window.py' sample code to see how to readout a CCD window.

5.3 Not implemented

The functions of the section 3.4 of [1] related to the control of external devices (external relay, AO tip tilt, filter wheel) have not yet been implemented. A few other features are also missing.

[1]The authors of `pyfits` are planning to keep some backward compatibility with `numarray` for a while, so updating the `pysbig` module will not be so urgent. But it should be done, as the backward compatibility will not be kept for long.

Appendix

BIBLIOGRAPHY

[1] Santa Barbara Instrument Group, January 11, 2005, “SBIG Universal Driver/Library”

[2] Le Guillou L., HERMES-WP-430-04, 2006, “SBIG CCD camera: Linux driver setup”.

[3] Guido van Rossum, Python Software Foundation, 29 March 2006, Release 2.4.3, “Extending and Embedding the Python Interpreter”, http://docs.python.org/ext/ext.html

[4] Perry Greenfield *et al.*, Space Telescope Science Institute, Release 1.5, “numarray User’s Manual”, chapter 14: “C extension API”, http://www.stsci.edu/resources/software_hardware/numarray

[5] “Numpy: Numerical Python”, http://sourceforge.net/projects/numpy/

[6] Space Telescope Science Institute, July 2005, “PyFITS User’s Manual” http://www.stsci.edu/resources/software_hardware/pyfits

[7] IAU FITS Working Group, December 9, 2005, FITS Standard, Version 2.1b, “Definition of the Flexible Image Transport System (FITS)”, http://fits.gsfc.nasa.gov/iaufwg/

INDEX