LLNL Nanosecond Gated Camera

2.1.2

Generated by Doxygen 1.10.0

Chapter 1

Namespace Index

1.1 Package List

Here are the packages with brief descriptions (if available):

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 nsCamera Namespace Reference

Namespaces

- namespace boards
- namespace CameraAssembler
- namespace comms
- namespace sensors
- namespace utils

Variables

- list __all__ = ["CameraAssembler"]

5.1.1 Detailed Description

```
Created on Tue Mar 22 15:47:43 2016

The Package includes a Camera object and an assembler.

The camera object will be the workhorse of the API.  The assembler is used
to create the Camera object.

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.1.2 Variable Documentation

5.1.2.1 __all__

```
list nsCamera.__all__ = ["CameraAssembler"]  [private]
```

Definition at line 26 of file __init__.py.

5.2 nsCamera.boards Namespace Reference

Namespaces

- namespace LLNL_v1
- namespace LLNL_v4

Variables

- list __all__ = ["LLNL_v1", "LLNL_v4"]

5.2.1 Detailed Description

```
This package is a collection of modules that represent the camera boards
Each board has its own number of ADCs, POTs, and sensors. More devices can be added in
the future. The list of imports will grow as we make more types of boards.

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.2.2 Variable Documentation

5.2.2.1 __all__

```
list nsCamera.boards.__all__ = ["LLNL_v1", "LLNL_v4"]  [private]
```

Definition at line 23 of file __init__.py.

5.3 nsCamera.boards.LLNL_v1 Namespace Reference

Classes

- class llnl_v1

5.3.1 Detailed Description

```
LLNLv1 board definition, including monitors, pots, and other board-specific settings

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.4 nsCamera.boards.LLNL_v4 Namespace Reference

Classes

- class llnl_v4

5.4.1 Detailed Description

```
LLNLv4 board definition, including monitors, DACS, and other board-specific settings

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.5 nsCamera.CameraAssembler Namespace Reference

Classes

- class CameraAssembler

5.5.1 Detailed Description

```
CameraAssembler assembles the separate camera parts into a camera object. This object
controls a combination of three components:

1. board : FPGA board -- LLNL_V1, LLNL_V4
2. comms: communication interface -- GigE, RS422
3. sensor : sensor type -- icarus, icarus2, daedalus

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.6 nsCamera.comms Namespace Reference

Namespaces

- namespace GigE
- namespace RS422

Variables

- list __all__ = ["RS422", "GigE"]

5.6.1 Detailed Description

```
Created on Tue Mar 22 15:47:43 2016

This package is a collection of modules for uniform handling of the nsCamera's
  communication systems

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.6.2 Variable Documentation

5.6.2.1 __all__

```
list nsCamera.comms.__all__ = ["RS422", "GigE"]  [private]
```

Definition at line 25 of file __init__.py.

5.7 nsCamera.comms.GigE Namespace Reference

Classes

- class GigE

5.7.1 Detailed Description

```
Gigabit Ethernet interface for nsCamera.

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.8 nsCamera.comms.RS422 Namespace Reference

Classes

- class RS422

5.8.1 Detailed Description

```
RS422 driver for nsCamera

Author: Brad Funsten (funsten1@llnl.gov)
Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.9 nsCamera.sensors Namespace Reference

Namespaces

- namespace daedalus
- namespace icarus
- namespace icarus2
- namespace sensorBase

Variables

- list __all__ = ["icarus", "icarus2", "daedalus"]

5.9.1 Detailed Description

```
This package is a collection of modules for uniform handling of the various sensors

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.9.2 Variable Documentation

5.9.2.1 __all__

```
list nsCamera.sensors.__all__ = ["icarus", "icarus2", "daedalus"]  [private]
```

Definition at line 23 of file __init__.py.

5.10 nsCamera.sensors.daedalus Namespace Reference

Classes

- class daedalus

5.10.1 Detailed Description

```
Parameters and functions specific to the daedalus three-frame sensor


Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.11 nsCamera.sensors.icarus Namespace Reference

Classes

- class icarus

5.11.1 Detailed Description

```
Parameters and functions specific to the icarus two-frame sensor

***Do not use this file as a template for new code development***

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.12 nsCamera.sensors.icarus2 Namespace Reference

Classes

- class icarus2

5.12.1 Detailed Description

```
Parameters and functions specific to the four-frame icarus2 sensor

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.13 nsCamera.sensors.sensorBase Namespace Reference

Classes

- class sensorBase

5.13.1 Detailed Description

```
Superclass for nsCamera sensors

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.14 nsCamera.utils Namespace Reference

Namespaces

- namespace crc16pure
- namespace FlatField
- namespace misc
- namespace Packet
- namespace Subregister

Variables

- list __all__ = ["SubRegister", "Packet", "FlatField", "misc"]

5.14.1 Detailed Description

```
This package is a collection of utility classes for the CameraAssembler

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.14.2 Variable Documentation

5.14.2.1 __all__

```
list nsCamera.utils.__all__ = ["SubRegister", "Packet", "FlatField", "misc"]  [private]
```

Definition at line 30 of file __init__.py.

5.15 nsCamera.utils.crc16pure Namespace Reference

Functions

- _crc16 (data, crc, table)
- crc16xmodem (data, crc=0)

Variables

- list CRC16_XMODEM_TABLE

5.15.1 Detailed Description

```
Pure python library for calculating CRC16
    NOTE: modified slightly to combine Python 2 and Python 3 versions in single file
```

5.15.2 Function Documentation

5.15.2.1 _crc16()

```
nsCamera.utils.crc16pure._crc16 (
              data,
              crc,
              table )  [protected]
```

```
Calculate CRC16 using the given table.
'data'      - data for calculating CRC, must be a string
'crc'       - initial value
'table'     - table for caclulating CRC (list of 256 integers)
Return calculated value of CRC
```

Definition at line 299 of file crc16pure.py.

```
def _crc16(data, crc, table):
    """Calculate CRC16 using the given table.
    `data`      - data for calculating CRC, must be a string
    `crc`       - initial value
    `table`     - table for caclulating CRC (list of 256 integers)
    Return calculated value of CRC
    """
    for byte in data:
        if sys.version_info > (3,):
            crc = ((crc « 8) & 0xFF00) ^ table[((crc » 8) & 0xFF) ^ byte]
        else:
            crc = ((crc « 8) & 0xFF00) ^ table[((crc » 8) & 0xFF) ^ ord(byte)]

    return crc & 0xFFFF


```

5.15.2.2 crc16xmodem()

```
nsCamera.utils.crc16pure.crc16xmodem (
              data,
              crc = 0 )
```

```
Calculate CRC-CCITT (XModem) variant of CRC16.
'data'      - data for calculating CRC, must be a string
'crc'       - initial value
Return calculated value of CRC
```

Definition at line 315 of file crc16pure.py.

```
def crc16xmodem(data, crc=0):
    """Calculate CRC-CCITT (XModem) variant of CRC16.
    `data`      - data for calculating CRC, must be a string
    `crc`       - initial value
    Return calculated value of CRC
    """
    return _crc16(data, crc, CRC16_XMODEM_TABLE)
```

5.15.3 Variable Documentation

5.15.3.1 CRC16_XMODEM_TABLE

```
list nsCamera.utils.crc16pure.CRC16_XMODEM_TABLE
```

Definition at line 39 of file crc16pure.py.

5.16 nsCamera.utils.FlatField Namespace Reference

Functions

- getFilenames (frame="Frame 1")
- getROIvector (imgfilename, roi)
- tslopes (x, y)
- generateFF (FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"], roi=[0, 0, 512, 1024], directory="", ncores=-1)
- removeFF (filename, directory="", roi=[0, 0, 512, 1024])
- removeFFall (directory="", FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"], roi=[0, 0, 512, 1024])

Variables

- parser = argparse.ArgumentParser()
- action
- dest
- default
- help
- nargs
- args = parser.parse_args()
- list framelist = ["Frame_" + str(frame) for frame in args.frames]
- directory

5.16.1 Detailed Description

```
Functions for batch flat-field image corrections

***Do not use this file as a template for new code development***

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.16.2 Function Documentation

5.16.2.1 generateFF()

```
nsCamera.utils.FlatField.generateFF (
                FRAMES = ["Frame_0", "Frame_1", "Frame_2", "Frame_3"],
                roi = [0, 0, 512, 1024],
                directory = "",
                ncores = -1 )
```

Definition at line 58 of file FlatField.py.

```
00063 ):
00064     # TODO: documentation
00065     # use of ROI here not compatible with use of ROI in removeFF
00066
00067     if directory:
00068         cwd = os.getcwd()
00069         newpath = os.path.join(cwd, directory)
00070         os.chdir(newpath)
00071     if not FRAMES:
00072         print("No framelist provided, defaulting to four frames")
00073         FRAMES = ["Frame_0", "Frame_1", "Frame_2", "Frame_3"]
00074     for f in FRAMES:
00075         files = getFilenames(frame=f)
00076         imgslist = [getROIvector(fn, roi) for fn in files]  # a list of flattened images
00077         imgsarray = np.vstack(imgslist)  # turn the list into an array
00078         npix = np.shape(imgsarray)[1]  # total number of pixels
00079         x = np.median(imgsarray, axis=1)  # median of each image used for flat fielding
00080         y = []
00081         for i in range(npix):
00082             # each member of y represents a pixel, as a list of magnitudes over all the
00083             #   images
00084             y.append(imgsarray[:, i])
00085         # get pixel gain and offset for flatfield ff using Thiel-Sen slopes
00086         ff = []
00087         ff = parallel.Parallel(n_jobs=ncores, verbose=5, pre_dispatch="2 * n_jobs")(
00088             delayed(tslopes)(x, pixel) for pixel in y
00089         )
00090         # x is the dependent variable; here uses median of image as characteristic of
00091         #   noise level
00092         m, c = zip(*ff)  # separate into gain and offset
00093         m = np.array(m)
00094         m[m < 0.1] = 0.1  # handle outliers
00095         m[m > 1000] = 1000  # handle outliers
00096         m = 1.0 / m
00097         m = m.reshape(roi[3] - roi[1], roi[2] - roi[0])  # turn into matrix
00098         c = np.array(c).reshape(roi[3] - roi[1], roi[2] - roi[0])  # turn into matrix
00099
00100         with open("px_gain_%s.txt" % f.replace("Frame_", "f"), "w+") as file:
00101             np.savetxt(file, m)
00102         with open("px_off_%s.txt" % f.replace("Frame_", "f"), "w+") as file:
00103             np.savetxt(file, c)
00104
00105
```

5.16.2.2 getFilenames()

```
nsCamera.utils.FlatField.getFilenames (
                frame = "Frame 1" )
```

```
get a list of tiff filenames in current working director for frame
```

Definition at line 32 of file FlatField.py.

```
00032 def getFilenames(frame="Frame 1"):
00033     """
00034     get a list of tiff filenames in current working director for frame
00035     """
00036     onlyfiles = next(os.walk("./"))[2]
00037     return [k for k in onlyfiles if frame in k and "tif" in k]
00038
00039
```

5.16.2.3 getROIvector()

```
nsCamera.utils.FlatField.getROIvector (
            imgfilename,
            roi )
```

```
return a numpy row vector of version of the image
```

Definition at line 40 of file FlatField.py.

```
def getROIvector(imgfilename, roi):
    """
    return a numpy row vector of version of the image
    """
    img = imread(imgfilename)
    vroi = img[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])].flattenlist()
    return vroi

```

5.16.2.4 removeFF()

```
nsCamera.utils.FlatField.removeFF (
            filename,
            directory = "",
            roi = [0, 0, 512, 1024] )
```

Definition at line 106 of file FlatField.py.

```
def removeFF(filename, directory="", roi=[0, 0, 512, 1024]):
    if directory:
        cwd = os.getcwd()
        newpath = os.path.join(cwd, directory)
        os.chdir(newpath)
    framenum = re.search("Frame_(\d)", filename).group(1)
    gainFilename = "px_gain_f" + framenum + ".txt"
    gainall = np.loadtxt(gainFilename)
    gain = gainall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]
    offFilename = "px_off_f" + framenum + ".txt"
    offsetall = np.loadtxt(offFilename, dtype="uint32")
    offset = offsetall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]

    beforeImageall = imread(filename)
    beforeImage = beforeImageall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]
    imageMed = np.median(beforeImage)

    flat = imageMed * gain + offset
    flat = flat.clip(0)
    fix = beforeImage - flat
    clipped = fix.clip(0)
    fixinit = clipped.astype("uint16")
    fiximg = Image.fromarray(fixinit)

    fixFilename = filename[:-4] + "ff" + filename[-4:]
    fiximg.save(fixFilename)

```

5.16.2.5 removeFFall()

```
nsCamera.utils.FlatField.removeFFall (
            directory = "",
            FRAMES = ["Frame_0", "Frame_1", "Frame_2", "Frame_3"],
            roi = [0, 0, 512, 1024] )
```

Definition at line 133 of file FlatField.py.

```
):
    cwd = os.getcwd()
    if directory:
        newpath = os.path.join(cwd, directory)
    else:
        newpath = cwd
    os.chdir(newpath)
    files = next(os.walk("./"))[2]
    filelist = []
    for frame in FRAMES:
        filelist.extend([k for k in files if frame in k and "tif" in k])
    for fname in filelist:
        removeFF(fname, directory, roi)


```

5.16.2.6 tslopes()

```
nsCamera.utils.FlatField.tslopes (
                x,
                y )
```

```
theilslopes implements a method for robust linear regression.
It computes the slope as the median of all slopes between paired values.
```

Definition at line 49 of file FlatField.py.

```
def tslopes(x, y):
    """
    theilslopes implements a method for robust linear regression.
    It computes the slope as the median of all slopes between paired values.
    """
    val = theilslopes(x, y)
    return [val[0], val[1]]


```

5.16.3 Variable Documentation

5.16.3.1 action

```
nsCamera.utils.FlatField.action
```

Definition at line 157 of file FlatField.py.

5.16.3.2 args

```
nsCamera.utils.FlatField.args = parser.parse_args()
```

Definition at line 167 of file FlatField.py.

5.16.3.3 default

```
nsCamera.utils.FlatField.default
```

Definition at line 157 of file FlatField.py.

5.16.3.4 dest

```
nsCamera.utils.FlatField.dest
```

Definition at line 157 of file FlatField.py.

5.16.3.5 directory

```
nsCamera.utils.FlatField.directory
```

Definition at line 169 of file FlatField.py.

5.16.3.6 framelist

```
nsCamera.utils.FlatField.framelist = ["Frame_" + str(frame) for frame in args.frames]
```

Definition at line 168 of file FlatField.py.

5.16.3.7 help

```
nsCamera.utils.FlatField.help
```

Definition at line 157 of file FlatField.py.

5.16.3.8 nargs

```
nsCamera.utils.FlatField.nargs
```

Definition at line 161 of file FlatField.py.

5.16.3.9 parser

```
nsCamera.utils.FlatField.parser = argparse.ArgumentParser()
```

Definition at line 155 of file FlatField.py.

5.17 nsCamera.utils.misc Namespace Reference

Classes

- class fakeCA

Functions

- makeLogLabels (logtag, label)
- getEnter (text)
- checkCRC (rval)
- str2bytes (astring)
- bytes2str (bytesequence)
- str2nparray (valstring)
- flattenlist (x)
- generateFrames (camassem, data, columns=1)
- loadDumpedData (filename="frames.txt", path=None, filetype="txt", sensor="daedalus", firstframe=None, lastframe=None, width=None, height=None, padToFull=None, firstrow=None, lastrow=None, maxwidth=None, maxheight=None, bytesperpixel=None, interlacing=None, columns=1)
- saveTiffs (self, frames, path=None, filename="Frame", prefix=None, index=None)
- plotFrames (self, frames, index=None)
- partition (self, frames, columns)

5.17.1 Detailed Description

```
Miscellaneous utilities, including batch processing of images acquired using the
  nsCamera. These are functions that don't require a cameraAssembler object to be
  instantiated before use.

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

5.17.2 Function Documentation

5.17.2.1 bytes2str()

```
nsCamera.utils.misc.bytes2str (
             bytesequence )
```

```
Python-version-agnostic converter of bytes to hexadecimal strings

Args:
    bytesequence: sequence of bytes as string (Py2) or bytes (Py3)

Returns:
    hexadecimal string representation of 'bytes' without '0x'
```

Definition at line 123 of file misc.py.

```
def bytes2str(bytesequence):
    """
    Python-version-agnostic converter of bytes to hexadecimal strings

    Args:
        bytesequence: sequence of bytes as string (Py2) or bytes (Py3)

    Returns:
        hexadecimal string representation of 'bytes' without '0x'
    """
    try:
        estring = binascii.b2a_hex(bytesequence)
    except TypeError:
        logging.error(
            "ERROR: bytes2str: Invalid byte sequence: '{bytesequence}'; returning an"
            " empty string".format(bytesequence=bytesequence)
        )
        return ""
    python, _, _, _, _ = sys.version_info
    if python >= 3:
        estring = str(estring)[2:-1]
    return estring


```

5.17.2.2 checkCRC()

```
nsCamera.utils.misc.checkCRC (
            rval )
```

```
Calculate CRC for rval[:-4] and compare with expected CRC in rval[-4:]

Args:
    rval: hexadecimal string

Returns:
    boolean, True if CRCs match, False if they don't match or the input is invalid
```

Definition at line 72 of file misc.py.

```
def checkCRC(rval):
    """
    Calculate CRC for rval[:-4] and compare with expected CRC in rval[-4:]

    Args:
        rval: hexadecimal string

    Returns:
        boolean, True if CRCs match, False if they don't match or the input is invalid
    """
    if not isinstance(rval, str) or len(rval) < 5:
        logging.error("ERROR: checkCRC: Invalid input: {rval}".format(rval=rval))
        return False
    data_crc = int(rval[-4:], base=16)
    CRC_calc = crc16pure.crc16xmodem(str2bytes(rval[:-4]))
    return CRC_calc == data_crc


```

5.17.2.3 flattenlist()

```
nsCamera.utils.misc.flattenlist (
            x )
```

```
Flatten list of lists recursively into single list
```

Definition at line 179 of file misc.py.

```
def flattenlist(x):
    """
    Flatten list of lists recursively into single list
    """
    python, _, _, _, _ = sys.version_info
    try:
        if python >= 3:
            if isinstance(x, collections.abc.Iterable):
                return [a for i in x for a in flattenlist(i)]
            else:
                return [x]
        else:
            if isinstance(x, collections.Iterable):
                return [a for i in x for a in flattenlist(i)]
            else:
                return [x]
    except RecursionError:
        logging.error(
            "ERROR: flattenlist: input '{x}' is pathological and cannot be flattened."
            " Attempting to return the input unchanged"
        )
        return x


```

5.17.2.4 generateFrames()

```
nsCamera.utils.misc.generateFrames (
              camassem,
              data,
              columns = 1 )
```

```
Processes data stream from board into frames and applies sensor-specific parsing.
  Generates padded data for full-size option of setRows.
If used for offline processing, replace the 'self' object with the parameter object
  returned by loadDumpedData().
If the data stream is incomplete (e.g., from an interrupted download), the data is
  padded with zeros to the correct length.

Args:
    self: during normal operation, is the CameraAssembler object. During offline
      operation, is a parameters object as returned by loadDumpedData()
    data: text stream from board or loaded file, or numpy from loaded file
    columns: 1 for single image, 2 for separate hemisphere images

Returns: list of parsed frames
```

Definition at line 261 of file misc.py.

```
def generateFrames(camassem, data, columns=1):
    """
    Processes data stream from board into frames and applies sensor-specific parsing.
      Generates padded data for full-size option of setRows.
    If used for offline processing, replace the 'self' object with the parameter object
      returned by loadDumpedData().
    If the data stream is incomplete (e.g., from an interrupted download), the data is
      padded with zeros to the correct length.

    Args:
        self: during normal operation, is the CameraAssembler object. During offline
          operation, is a parameters object as returned by loadDumpedData()
        data: text stream from board or loaded file, or numpy from loaded file
        columns: 1 for single image, 2 for separate hemisphere images

    Returns: list of parsed frames
    """
    logging.debug("DEBUG" + camassem.logtag + ": generateFrames")
    if isinstance(data[0], str):
        allframes = str2nparray(data)
    else:
        allframes = data
    nframes = camassem.sensor.lastframe - camassem.sensor.firstframe + 1
    frames = [0] * nframes
    framesize = camassem.sensor.width * (
        camassem.sensor.lastrow - camassem.sensor.firstrow + 1
    )
    if hasattr(camassem, "ca"):
        padIt = camassem.ca.padToFull
    else:
        padIt = camassem.padToFull
    if padIt:
        toprows = camassem.sensor.firstrow
        botrows = (camassem.sensor.maxheight - 1) - camassem.sensor.lastrow
        padtop = np.zeros(toprows * camassem.sensor.maxwidth, dtype=int)
        padbot = np.zeros(botrows * camassem.sensor.maxwidth, dtype=int)
        for n in range(nframes):
            thisframe = np.concatenate(
                (padtop, allframes[n * framesize : ((n + 1) * framesize)], padbot)
            )
            frames[n] = thisframe
    else:
        for n in range(nframes):
            frames[n] = allframes[n * framesize : (n + 1) * framesize]
    # self.clearStatus()
    parsed = camassem.sensor.parseReadoff(frames, columns)
    return parsed


```

5.17.2.5 getEnter()

```
nsCamera.utils.misc.getEnter (
               text )
```

```
Wait for enter key to be pressed.

Args:
    text: message asking for keypress
```

Definition at line 58 of file misc.py.

```
00058 def getEnter(text):
00059     """
00060     Wait for enter key to be pressed.
00061 
00062     Args:
00063         text: message asking for keypress
00064     """
00065     python, _, _, _, _ = sys.version_info
00066     if python >= 3:
00067         input(text)
00068     else:
00069         raw_input(text)
00070 
00071 
```

5.17.2.6 loadDumpedData()

```
nsCamera.utils.misc.loadDumpedData (
               filename = "frames.txt",
               path = None,
               filetype = "txt",
               sensor = "daedalus",
               firstframe = None,
               lastframe = None,
               width = None,
               height = None,
               padToFull = None,
               firstrow = None,
               lastrow = None,
               maxwidth = None,
               maxheight = None,
               bytesperpixel = None,
               interlacing = None,
               columns = 1 )
```

```
_summary_

    Output can be piped directly to saveTiffs:
        pars,frames=loadDumpedData(filename="Dump.npy")
        saveTiffs(pars,frames)
Args:
    filename (str, optional): _description_. Defaults to "frames.txt".
    path (_type_, optional): _description_. Defaults to None.
    filetype (str, optional): _description_. Defaults to "txt".
    sensor (str, optional): _description_. Defaults to "daedalus".
    firstframe (_type_, optional): _description_. Defaults to None.
    lastframe (_type_, optional): _description_. Defaults to None.
    width (_type_, optional): _description_. Defaults to None.
    height (_type_, optional): _description_. Defaults to None.
    padToFull (_type_, optional): _description_. Defaults to None.
    firstrow (_type_, optional): _description_. Defaults to None.
    lastrow (_type_, optional): _description_. Defaults to None.
    maxwidth (_type_, optional): _description_. Defaults to None.
    maxheight (_type_, optional): _description_. Defaults to None.
    interlacing (_type_, optional): _description_. Defaults to None.
    columns (int, optional): _description_. Defaults to 1.


Returns:
    Tuple (parameter object, list of data frames)
```

Definition at line 310 of file misc.py.

```
):
    """_summary_

        Output can be piped directly to saveTiffs:
            pars,frames=loadDumpedData(filename="Dump.npy")
            saveTiffs(pars,frames)
    Args:
        filename (str, optional): _description_. Defaults to "frames.txt".
        path (_type_, optional): _description_. Defaults to None.
        filetype (str, optional): _description_. Defaults to "txt".
        sensor (str, optional): _description_. Defaults to "daedalus".
        firstframe (_type_, optional): _description_. Defaults to None.
        lastframe (_type_, optional): _description_. Defaults to None.
        width (_type_, optional): _description_. Defaults to None.
        height (_type_, optional): _description_. Defaults to None.
        padToFull (_type_, optional): _description_. Defaults to None.
        firstrow (_type_, optional): _description_. Defaults to None.
        lastrow (_type_, optional): _description_. Defaults to None.
        maxwidth (_type_, optional): _description_. Defaults to None.
        maxheight (_type_, optional): _description_. Defaults to None.
        interlacing (_type_, optional): _description_. Defaults to None.
        columns (int, optional): _description_. Defaults to 1.


    Returns:
        Tuple (parameter object, list of data frames)
    """
    logging.debug("DEBUG: loadDumpedData")
    if sensor.lower() == "daedalus":
        import nsCamera.sensors.daedalus as snsr
    elif sensor.lower() == "icarus":
        import nsCamera.sensors.icarus as snsr
    elif sensor.lower() == "icarus2":
        import nsCamera.sensors.icarus2 as snsr
    else:
        logging.error(
            "ERROR loadDumpedData: invalid sensor type provided, defaulting to icarus2"
        )
        import nsCamera.sensors.icarus2 as snsr

    def buildEmptyFrames():
        cols = [0] * 512
        frame = np.array([cols] * (lastrow - firstrow + 1))
        return [frame] * (lastframe - firstframe + 1)

    # get defaults from class declarations if not specified as parameter
    if firstframe is None:
        firstframe = snsr.firstframe
    if lastframe is None:
        lastframe = snsr.lastframe
    # TODO: first frame number
    if width is None:
        width = snsr.width
    if height is None:
        height = snsr.height
    if firstrow is None:
        firstrow = snsr.firstrow
    if lastrow is None:
        lastrow = snsr.lastrow
    if maxwidth is None:
        maxwidth = snsr.maxwidth
    if maxheight is None:
        maxheight = snsr.maxheight
    if bytesperpixel is None:
        bytesperpixel = snsr.bytesperpixel
    if interlacing is None:
        interlacing = snsr.interlacing

    parameters = fakeCA(
        sensor,
        firstframe,
        lastframe,
        width,
        height,
        padToFull,
        firstrow,
        lastrow,
        maxwidth,
        maxheight,
        bytesperpixel,
        interlacing,
    )

    if path is None:
        path = os.path.join(os.getcwd())
    datafile = os.path.join(path, filename)
    if filename[-3:].lower() == "txt":
        filetype = "txt"
    elif filename[-3:].lower() == "npy":
        filetype = "npy"
    # TODO: return empty frames if error

    if filetype == "npy":
        expectedlength = (lastframe - firstframe + 1) * (lastrow - firstrow + 1) * width
        try:
            f = np.load(datafile)
            padding = expectedlength - len(f)
            if padding:
                logging.warning(
                    "{logwarn}loadDumpedData: Payload is shorter than expected."
                    " Padding with '0's".format(logwarn=parameters.logwarn)
                )
                f = np.pad(f, (0, padding), "constant", constant_values=(0))
            frames = generateFrames(parameters, f, columns)
            return parameters, frames

        except OSError as err:
            logging.error(
                "{logerr}loadDumpedData: OS error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=err)
            )
            return parameters, buildEmptyFrames()
        except:
            logging.error(
                "{logerr}loadDumpedData: Unexpected error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=str(sys.exc_info()[0]))
            )
            return parameters, buildEmptyFrames()
    # if filetype is not explicitly npy, try loading as text
    else:
        # Payload size as string implied by provided parameters
        expectedlength = (
            4 * (lastframe - firstframe + 1) * (lastrow - firstrow + 1) * width
        )

        try:
            f = open(datafile, "r")
            s = f.read()

            padding = expectedlength - len(s)
            if padding:
                logging.warning(
                    "{logwarn}loadDumpedData: Payload is shorter than expected."
                    " Padding with '0's".format(logwarn=parameters.logwarn)
                )
                s = s.ljust(expectedlength, "0")

            frames = generateFrames(parameters, s)
            return parameters, frames

        except OSError as err:
            logging.error(
                "{logerr}loadDumpedData: OS error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=err)
            )
            return parameters, buildEmptyFrames()
        except ValueError:
            logging.error(
                "{logerr}loadDumpedData: Could not convert data to an integer."
                " Returning empty frames.".format(logerr=parameters.logerr)
            )
            return parameters, buildEmptyFrames()
        except:
            logging.error(
                "{logerr}loadDumpedData: Unexpected error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=str(sys.exc_info()[0]))
            )
            return parameters, buildEmptyFrames()


```

5.17.2.7 makeLogLabels()

```
nsCamera.utils.misc.makeLogLabels (
                logtag,
                label )
```

Definition at line 39 of file misc.py.

```
def makeLogLabels(logtag, label):
    if logtag is None:
        logtag = ""

    logcritbase = "CRITICAL{logtag}: ".format(logtag=logtag)
    logerrbase = "ERROR{logtag}: ".format(logtag=logtag)
    logwarnbase = "WARNING{logtag}: ".format(logtag=logtag)
    loginfobase = "INFO{logtag}: ".format(logtag=logtag)
    logdebugbase = "DEBUG{logtag}: ".format(logtag=logtag)

    logcrit = "{base}{label}".format(base=logcritbase, label=label)
    logerr = "{base}{label}".format(base=logerrbase, label=label)
    logwarn = "{base}{label}".format(base=logwarnbase, label=label)
    loginfo = "{base}{label}".format(base=loginfobase, label=label)
    logdebug = "{base}{label}".format(base=logdebugbase, label=label)

    return logcrit, logerr, logwarn, loginfo, logdebug


```

5.17.2.8 partition()

```
nsCamera.utils.misc.partition (
              self,
              frames,
              columns )
```

```
Extracts interlaced frames and divides images by hemispheres. If interlacing does
  not evenly divide the height, remainder lines will be dropped

Args:
    self: during normal operation, is sensor object. During offline
      operation, is the parameter.sensor object returned by loadDumpedData()
    frames: list of full-sized frames
    columns: 1 for single image, 2 for separate hemisphere images

Returns: list of deinterlaced frames
```

Definition at line 634 of file misc.py.

```
00634 def partition(self, frames, columns):
00635     """
00636     Extracts interlaced frames and divides images by hemispheres. If interlacing does
00637       not evenly divide the height, remainder lines will be dropped
00638 
00639     Args:
00640         self: during normal operation, is sensor object. During offline
00641           operation, is the parameter.sensor object returned by loadDumpedData()
00642         frames: list of full-sized frames
00643         columns: 1 for single image, 2 for separate hemisphere images
00644 
00645     Returns: list of deinterlaced frames
00646     """
00647     logging.debug(
00648         "{logdebug}partition: columns = {columns}, interlacing = {interlacing}".format(
00649             logdebug=self.logdebug, columns=columns, interlacing=self.sensor.interlacing
00650         )
00651     )
00652 
00653     def unshuffle(frames, ifactor):
00654         warntrimmed = False
00655         if self.padToFull:
00656             newheight = self.sensor.maxheight // (ifactor + 1)
00657             if newheight != (self.sensor.maxheight / (ifactor + 1)):
00658                 warntrimmed = True
00659         else:
00660             newheight = self.sensor.height // (ifactor + 1)
00661             if newheight != (self.sensor.height / (ifactor + 1)):
00662                 warntrimmed = True
00663 
00664         if warntrimmed:
00665             logging.warning(
00666                 "{logwarn} partition: interlacing setting requires dropping of lines to"
00667                 " maintain consistent frame sizes ".format(logwarn=self.logwarn)
00668             )
00669         delaced = []
00670         for frame in frames:
00671             for sub in range(ifactor + 1):
00672                 current = np.zeros((newheight, self.sensor.width // columns), dtype=int)
00673                 for line in range(newheight):
00674                     current[line] = frame[(ifactor + 1) * line + sub]
00675                 delaced.append(current)
00676         nframes = self.sensor.lastframe - self.sensor.firstframe + 1
00677         resorted = [None] * len(delaced)
```

```
00678         for sub in range(ifactor + 1):
00679             for idx, frame in enumerate(frames):
00680                 resorted[sub * nframes + idx] = delaced[idx * (ifactor + 1) + sub]
00681         return resorted
00682 
00683     if self.sensor.interlacing[0] != self.sensor.interlacing[1]:
00684         columns = 2  # true even if not explicitly requested by readoff
00685     if columns == 1:
00686         if self.sensor.interlacing == [0, 0]:  # don't do anything
00687             return frames
00688         else:
00689             return unshuffle(frames, self.sensor.interlacing[0])
00690     else:
00691         # reshape frame into the proper shape, then split horizontally
00692         if self.padToFull:
00693             framesab = [
00694                 np.hsplit(frame.reshape(self.sensor.maxheight, -1), 2)
00695                 for frame in frames
00696             ]
00697         else:
00698             framesab = [
00699                 np.hsplit(
00700                     frame.reshape((self.sensor.lastrow - self.sensor.firstrow + 1), -1),
00701                     2,
00702                 )
00703                 for frame in frames
00704             ]
00705         framesa = [hemis[0] for hemis in framesab]
00706         framesb = [hemis[1] for hemis in framesab]
00707     if self.sensor.interlacing == [0, 0]:
00708         return framesa + framesb
00709     else:
00710         return unshuffle(framesa, self.sensor.interlacing[0]) + unshuffle(
00711             framesb, self.sensor.interlacing[1]
00712         )
00713 
00714 
00715 """
00716 Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
00717 LLNL-CODE-838080
00718 
00719 This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
00720 contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
00721 and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
00722 'nsCamera' is distributed under the terms of the MIT license. All new contributions must
00723 be made under this license.
00724 """
```

5.17.2.9 plotFrames()

```
nsCamera.utils.misc.plotFrames (
                self,
                frames,
                index = None )
```

```
Plot frame or list of frames as individual graphs.

Args:
    self: during normal operation, is cameraAssembler object. During offline
      operation, is the parameter object returned by loadDumpedData()
    frames: numpy array or list of numpy arrays
    index: number to start frame numbering

Returns:
    Error string
```

Definition at line 567 of file misc.py.

```
00567 def plotFrames(self, frames, index=None):
00568     """
```

```
    Plot frame or list of frames as individual graphs.

    Args:
        self: during normal operation, is cameraAssembler object. During offline
          operation, is the parameter object returned by loadDumpedData()
        frames: numpy array or list of numpy arrays
        index: number to start frame numbering

    Returns:
        Error string
    """
    # logging.info(self.loginfo + "plotFrames: index = " + str(index))
    logging.info(
        "{loginfo}: plotFrames: index = {index}".format(
            loginfo=self.loginfo, index=index
        )
    )
    err = ""
    if index is None:
        nframe = self.sensor.firstframe
    else:
        nframe = index

    if not isinstance(frames, list):
        frames = [frames]

    # if this is a text string from fast readoff, do the numpy conversion now
    if isinstance(frames[0], str):
        frames = generateFrames(frames)

    framestemp = np.copy(frames)
    for idx, frame in enumerate(framestemp):
        if idx < len(framestemp) / 2:
            interlacing = self.sensor.interlacing[0]
        else:
            interlacing = self.sensor.interlacing[1]
        try:
            if self.padToFull:
                frame = np.reshape(
                    frame, (self.sensor.maxheight // (interlacing + 1), -1)
                )
            else:
                frame = np.reshape(
                    frame,
                    (
                        (self.sensor.lastrow - self.sensor.firstrow + 1)
                        // (interlacing + 1),
                        -1,
                    ),
                )
        except:
            err = "{logerr}plotFrames: unable to plot frame".format(logerr=self.logerr)
            logging.error(err)
            continue
        plt.imshow(frame, cmap="gray")
        name = "Frame %d" % nframe
        plt.title(name)
        plt.show()
        nframe += 1
    return err


#  TODO: separate images for hemispheres with different timing


```

5.17.2.10 saveTiffs()

```
nsCamera.utils.misc.saveTiffs (
                self,
                frames,
                path = None,
                filename = "Frame",
                prefix = None,
                index = None )
```

```
Save numpy array or list of numpy arrays or single array to disk as individual
  tiffs, with frame number appended to filename. If used for standalone, use the
  parameter object returned by loadDumpedData() as the first argument

Args:
    self: during normal operation, is cameraAssembler object. During offline
      operation, is the parameter object returned by loadDumpedData()
    frames: numpy array or list of numpy arrays
    path: save path, defaults to './output'
    filename: defaults to 'Frame' followed by frame number
    prefix: prepended to 'filename', defaults to time/date
      (e.g. '160830-124704_')
    index: number to start frame numbering

Returns:
    Error string
```

Definition at line 487 of file misc.py.

```
):
    """
    Save numpy array or list of numpy arrays or single array to disk as individual
      tiffs, with frame number appended to filename. If used for standalone, use the
      parameter object returned by loadDumpedData() as the first argument

    Args:
        self: during normal operation, is cameraAssembler object. During offline
          operation, is the parameter object returned by loadDumpedData()
        frames: numpy array or list of numpy arrays
        path: save path, defaults to './output'
        filename: defaults to 'Frame' followed by frame number
        prefix: prepended to 'filename', defaults to time/date
          (e.g. '160830-124704_')
        index: number to start frame numbering

    Returns:
        Error string
    """
    # logging.info("INFO" + self.logtag + ": saveTiffs")
    logging.info("{loginfo}: saveTiffs".format(loginfo=self.loginfo))
    err = ""
    if path is None:
        path = os.path.join(os.getcwd(), "output")
    if prefix is None:
        prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
    if not os.path.exists(path):
        os.makedirs(path)
    if index is None:
        firstnum = self.sensor.firstframe
    else:
        firstnum = index

    # if this is a text string from fast readoff, do the numpy conversion now
    if isinstance(frames[0], str):
        frames = generateFrames(frames)

    framestemp = np.copy(frames)
    if np.issubdtype(type(framestemp[0]), np.number):
        # if type(framestemp[0]) is np.uint16:
        # single frame needs to be a list containing one frame
        framestemp = [framestemp]
    for idx, frame in enumerate(framestemp):
        if idx < len(framestemp) / 2:
            interlacing = self.sensor.interlacing[0]
        else:
            interlacing = self.sensor.interlacing[1]
        try:
            if self.padToFull:
                frame = np.reshape(
                    frame, (self.sensor.maxheight // (interlacing + 1), -1)
                )
            else:
                frame = np.reshape(
                    frame,
                    (
                        (self.sensor.lastrow - self.sensor.firstrow + 1)
                        // (interlacing + 1),
00552                         -1,
00553                     ),
00554                 )
00555             frameimg = Image.fromarray(frame, "I;16")
00556             namenum = filename + "_%d" % firstnum
00557             tifpath = os.path.join(path, prefix + namenum + ".tif")
00558             frameimg.save(tifpath)
00559             firstnum += 1
00560         except Exception:
00561             err = "saveTiffs: unable to save images"
00562             # logging.error("ERROR" + self.logtag + ": " + err)
00563             logging.error("{logerr}: {err}".format(logerr=self.logerr, err=err))
00564     return err
00565
00566
```

5.17.2.11 str2bytes()

```
nsCamera.utils.misc.str2bytes (
                astring )
```

```
Python-version-agnostic converter of hexadecimal strings to bytes

Args:
    astring: hexadecimal string without '0x'

Returns:
    byte string equivalent to input string
```

Definition at line 90 of file misc.py.

```
00090 def str2bytes(astring):
00091     """
00092     Python-version-agnostic converter of hexadecimal strings to bytes
00093
00094     Args:
00095         astring: hexadecimal string without '0x'
00096
00097     Returns:
00098         byte string equivalent to input string
00099     """
00100
00101     python, _, _, _, _ = sys.version_info
00102     if python >= 3:
00103         try:
00104             dbytes = binascii.a2b_hex(astring)
00105         except:
00106             logging.error(
00107                 "ERROR: str2bytes: invalid input: '{astring}'; returning zero"
00108                 " byte".format(astring=astring)
00109             )
00110             dbytes = b"\x00"
00111     else:
00112         try:
00113             dbytes = astring.decode("hex")
00114         except:
00115             logging.error(
00116                 "ERROR: str2bytes: invalid input: '{astring}'; returning zero "
00117                 " byte".format(astring=astring)
00118             )
00119             dbytes = b"\x00"
00120     return dbytes
00121
00122
```

5.17.2.12 str2nparray()

```
nsCamera.utils.misc.str2nparray (
              valstring )
```

```
Convert string into array of uint16s

Args:
    valstring: string of hexadecimal characters

Returns:
    numpy array of uint16
```

Definition at line 147 of file misc.py.

```
def str2nparray(valstring):
    """
    Convert string into array of uint16s

    Args:
        valstring: string of hexadecimal characters

    Returns:
        numpy array of uint16
    """
    if not isinstance(valstring, str):
        logging.error(
            "ERROR: str2nparray: Invalid input: {valstring} is not a string. Returning"
            " an empty array".format(valstring=valstring)
        )
        return np.array([])
    stringlen = len(valstring)
    arraylen = int(stringlen / 4)
    outarray = np.empty(int(arraylen), dtype="uint16")

    for i in range(0, arraylen):
        try:
            outarray[i] = int(valstring[4 * i : 4 * i + 4], 16)
        except ValueError:
            logging.error(
                "ERROR: str2nparray: input string does not represent a hexadecimal"
                " integer. Returning an empty array"
            )
            return np.array([])
    return outarray


```

5.18 nsCamera.utils.Packet Namespace Reference

Classes

- class Packet

5.18.1 Detailed Description

```
Packet object for communication with boards

Author: Brad Funsten (funsten1@llnl.gov)
Author: Jeremy Hill (hill35@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
```

5.19 nsCamera.utils.Subregister Namespace Reference

Classes

- class SubRegister

5.19.1 Detailed Description

```
Subregister object represents a subset of a full register

Author: Matthew Dayton (dayton5@llnl.gov)
Author: Jeremy Martin Hill (jerhill@llnl.gov)
```

Chapter 6

Class Documentation

6.1 nsCamera.CameraAssembler.CameraAssembler Class Reference

Public Member Functions

- __init__ (self, boardname="llnl_v4", commname="GigE", sensorname="icarus2", verbose=4, port=None, ip=None, logfile=None, logtag=None, timeout=30)
- initBoard (self)

  Aliases to other objects' methods TODO: properly delegate these methods.
- initPots (self)
- latchPots (self)
- initSensor (self)
- configADCs (self)
- disarm (self)
- startCapture (self, mode)
- readSRAM (self)
- waitForSRAM (self, timeout=None)
- getTimer (self)
- resetTimer (self)
- enableLED (self, status=1)
- setLED (self, LED=1, status=1)
- setPowerSave (self, status=1)
- setPPER (self, pollperiod=None)
- getTemp (self, scale=None)
- getPressure (self, offset=None, sensitivity=None, units=None)
- clearStatus (self)
- checkStatus (self)
- checkStatus2 (self)
- reportStatus (self)
- reportEdgeDetects (self)
- dumpStatus (self)
- checkSensorVoltStat (self)
- setTiming (self, side="AB", sequence=None, delay=0)
- setArbTiming (self, side="AB", sequence=None)

- getTiming (self, side=None, actual=None)
- setManualShutters (self, timing=None)
- setManualTiming (self, timing=None)
- getManualTiming (self)
- getSensTemp (self, scale=None, offset=None, slope=None, dec=1)
- sensorSpecific (self)
- selectOscillator (self, osc=None)
- setInterlacing (self, ifactor=None, side=None)
- setHighFullWell (self, flag=True)
- setZeroDeadTime (self, flag=True, side=None)
- setTriggerDelay (self, delay=0)
- setPhiDelay (self, side=None, delay=0)
- setExtClk (self, dilation=None, frequency=None)
- parseReadoff (self, frames, columns=1)
- sendCMD (self, pkt)
- arm (self, mode=None)
- readFrames (self, waitOnSRAM=None, timeout=0, fast=False, columns=1)
- readoff (self, waitOnSRAM=None, timeout=0, fast=None, columns=1)
- writeSerial (self, cmd, timeout=None)
- readSerial (self, size, timeout=None)
- closeDevice (self)
- saveTiffs (self, frames, path=None, filename="Frame", prefix=None, index=None)
- plotFrames (self, frames, index=None)
- getEnter (self, text)
- checkCRC (self, rval)
- str2bytes (self, astring)
- bytes2str (self, bytesequence)
- str2nparray (self, valstring)
- flattenlist (self, mylist)
- partition (self, frames, columns)
- initialize (self)

  End aliases.
- reinitialize (self)
- reboot (self)
- getBoardInfo (self)
- getRegister (self, regname)
- setRegister (self, regname, regval)
- resolveSubreg (self, srname)
- getSubregister (self, subregname)
- setSubregister (self, subregname, valstring)
- submitMessages (self, messages, errorstring="Error")
- getPot (self, potname, errflag=False)
- setPot (self, potname, value=1.0, errflag=False)
- getPotV (self, potname, errflag=False)
- setPotV (self, potname, voltage, tune=False, accuracy=0.01, iterations=20, approach=0.75, errflag=False)
- getMonV (self, monname, errflag=False)
- readImgs (self, waitOnSRAM=True, mode="Hardware")
- saveFrames (self, frames, path=None, filename="frames", prefix=None)
- saveNumpys (self, frames, path=None, filename="Frame", prefix=None, index=None)
- dumpNumpy (self, datastream, path=None, filename="Dump", prefix=None)

- checkRegSet (self, regname, teststring)
- initPowerCheck (self)
- powerCheck (self, delta=10)
- printBoardInfo (self)
- dumpRegisters (self)
- dumpSubregisters (self)
- mmReadoff (self, waitOnSRAM, variation=None)
- setFrames (self, minframe=None, maxframe=None)
- setRows (self, minrow=0, maxrow=None, padToFull=False)
- abortReadoff (self, flag=True)
- batchAcquire (self, sets=1, trig="Hardware", path=None, filename="Frame", prefix=None, showProgress=0)
- saveHDF (self, frames, path=None, filename="Acquisition", prefix=None)

Public Attributes

- version
- currtime
- oldtime
- trigtime
- waited
- read
- unstringed
- parsedtime
- savetime
- cycle
- boardname
- timeout
- commname
- sensorname

  For regular version.
- verbose
- port
- python
- pyth1
- pyth2
- PY3
- platform
- arch
- FPGAVersion
- FPGANum
- FPGAboardtype
- FPGArad
- FPGAsensor
- FPGAinterfaces
- FPGAinvalid
- iplist
- packageroot
- armed
- senstiming
- sensmanual

- inittime
- padToFull
- abort
- verbmap
- logtag
- logcritbase
- logerrbase
- logwarnbase
- loginfobase
- logdebugbase
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- verblevel
- payloaderror
- sensor
- comms
- board

6.1.1 Detailed Description

```
Code to assemble correct code to manage FPGA, frame grabber, and sensor

Exposed methods:
    initialize() - initializes board registers and pots, sets up sensor
    reinitialize() - initialize board and sensors, restore last known timer settings
    reboot() - perform software reset of board and reinitialize
    getBoardInfo() - parses FPGA_NUM register to retrieve board description
    getRegister(regname) - retrieves contents of named register
    setRegister(regname, string) - sets named register to given value
    getSubregister(subregname) - return substring of register identified in board
      attribute 'subregname'
    setSubregister(subregname, valstring) - replace substring of register identified
      in board attribute 'subregname' with 'valstring'
    submitMessages(messages) - set registers or subregisters based on list of
      destination/payload tuples
    getPot(potname) - returns float (0 < value < 1) corresponding to integer stored
      in pot or monitor 'potname'
    setPot(potname, value) - 0 < value < 1; sets named pot to fixed-point number =
      'value' * (maximum pot value)
    getPotV(potname) - returns voltage setting of 'potname'
    setPotV(potname, voltage) - sets named pot to voltage
    getMonV(monname) - returns voltage read by monitor 'monname' (or monitor
      associated with given potname)
    readImgs() - calls arm() and readoff() functions
    saveFrames(frames) - save image object as one file
    saveNumpys(frames) - save individual frames as numpy data files
    dumpNumpy(datastream) - save datastream string to numpy file
    powerCheck(delta) - check that board power has not failed
    printBoardInfo() - print board information derived from FPGA_NUM register
    dumpRegisters() - return contents of all board registers
    dumpSubregisters() - return contents of all named board subregisters
    setFrames(min, max) - select subset of frames for readoff
    setRows(min, max, padToFull) - select subset of rows for readoff
    abortReadoff() - cancel readoff in wait-for-SRAM loop
    batchAquire() - fast acquire a finite series of images
    loadTextFrames() - load data sets previously saved as text and convert to frames

Includes aliases to board- and sensor- specific methods:
    Board methods
        disarm() - take camera out of waiting-for-trigger state
        clearStatus() - clear contents of status registers
        checkStatus() - print contents of status register as reversed bit string
        checkStatus2() - print contents of status register 2 as reversed bit string
        reportStatus() - print report on contents of status registers
        resetTimer() - reset on-board timer
        getTimer() - read on-board timer
        enableLED(status) - enable (default) or disable (status = 0) on-board LEDs
        setLED(LED#, status) - turn LED on (default) or off (status = 0)
        setPowerSave(status) - turn powersave functionality on (default) or off
          (status = 0)
        getTemp() - read on-board temperature sensor
        getPressure() - read on-board pressure sensor
        dumpStatus() - generate dictionary of status, register, and subregister
          contents
    Sensor methods
        checkSensorVoltStat() - checks that jumper settings match sensor selection
        setTiming(side, sequencetuple, delay) - configure high-speed timing
        setArbTiming(side, sequencelist) - configure arbitrary high-speed timing
          sequence
        getTiming(side) - returns high speed timing settings from registers
        setManualTiming() - configures manual shutter timing
        getManualTiming() - returns manual shutter settings from registers
        selectOscillator(osc) - select timing oscillator
        setInterlacing(ifactor) - sets interlacing factor
        setHighFullWell(flag) - controls High Full Well mode
        setZeroDeadTime(flag, side) - controls Zero Dead Time mode
        setTriggerDelay(delayblocks) - sets trigger delay
    Comms methods
        sendCMD(pkt)- sends packet object via serial port
        arm() - configures software buffers & arms camera
        readFrames() - waits for data ready flag, then downloads image data
        readoff() - waits for data ready flag, then downloads image data
        closeDevice() - disconnect interface and release resources
    Miscellaneous functions (bare functions that can be called as methods)
        saveTiffs(frames) - save individual frames as tiffs
        plotFrames(frames) - plot individual frames as tiffs

Informational class variables:
    version - nsCamera software version
    FPGAVersion - firmware version (date)
    FPGANum - firmware implementation identifier
    FPGAboardtype - FPGA self-identified board type (should match 'boardname')
    FPGArad = Flag indicating radiation-tolerant FPGA build
    FPGAsensor = FPGA self-identified sensor family (should correspond to
      'sensorname')
    FPGAinterfaces = FPGA self-identified interfaces (list should include
      'commname')
    FPGAinvalid = flag indicating invalid FPGA information in register
```

Definition at line 59 of file CameraAssembler.py.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 __init__()

```
nsCamera.CameraAssembler.CameraAssembler.__init__ (
              self,
              boardname = "llnl_v4",
              commname = "GigE",
              sensorname = "icarus2",
              verbose = 4,
              port = None,
              ip = None,
              logfile = None,
              logtag = None,
              timeout = 30 )
```

```
Args:
    boardname: name of FPGA board: llnl_v1, llnl_v4
    commname: name of communication interface: rs422, gige
    sensorname: name of sensor: icarus, icarus2, daedalus
    verbose: optional, sets logging level
        0: print no logging messages
        1: print CRITICAL logging messages (camera will not operate, e.g.,
          unable to connect to board)
        2: print ERROR logging messages (camera will not operate as directed,
          e.g., an attempt to set the timing mode has failed, but the camera
          is still operational)
        3: print WARNING logging messages (camera will operate as directed, but
          perhaps not as expected, e.g., ca.setTiming('A', (9, 8), 1) may be
          programmed correctly, but the actual timing generated by the board
          will be {1} [9, 8, 9, 14, 9, 8, 9])
        4: print INFO logging messages (operational messages from ordinary
          camera operation)
        5. print DEBUG logging messages (detailed messages describing specific
          operations and messages)
    port: optional integer
        When using RS422, this preselects the comport for RS422 and bypasses
          port search
        When using GigE, this preselects the OrangeTree control port for GigE
          (ignored if an ip parameter is not also provided)
    ip: optional string (e.g., '192.168.1.100')
        GigE: bypasses network search and selects particular OrangeTree board -
          required for some operating systems
    logfile: optional string, name of file to divert console output
    timeout: timeout in seconds for connecting using Gigabit Ethernet
```

Definition at line 151 of file CameraAssembler.py.

```
    ):
        """
        Args:
            boardname: name of FPGA board: llnl_v1, llnl_v4
            commname: name of communication interface: rs422, gige
            sensorname: name of sensor: icarus, icarus2, daedalus
            verbose: optional, sets logging level
                0: print no logging messages
                1: print CRITICAL logging messages (camera will not operate, e.g.,
                  unable to connect to board)
                2: print ERROR logging messages (camera will not operate as directed,
                  e.g., an attempt to set the timing mode has failed, but the camera
                  is still operational)
                3: print WARNING logging messages (camera will operate as directed, but
                  perhaps not as expected, e.g., ca.setTiming('A', (9, 8), 1) may be
                  programmed correctly, but the actual timing generated by the board
                  will be {1} [9, 8, 9, 14, 9, 8, 9])
                4: print INFO logging messages (operational messages from ordinary
                  camera operation)
                5. print DEBUG logging messages (detailed messages describing specific
                  operations and messages)
            port: optional integer
                When using RS422, this preselects the comport for RS422 and bypasses
                  port search
                When using GigE, this preselects the OrangeTree control port for GigE
                  (ignored if an ip parameter is not also provided)
            ip: optional string (e.g., '192.168.1.100')
```

```
                GigE: bypasses network search and selects particular OrangeTree board -
                  required for some operating systems
            logfile: optional string, name of file to divert console output
            timeout: timeout in seconds for connecting using Gigabit Ethernet
        """
        self.version = "2.1.2"
        self.currtime = 0
        self.oldtime = 0
        self.trigtime = []
        self.waited = []
        self.read = []
        self.unstringed = []
        self.parsedtime = []
        self.savetime = []
        self.cycle = []
        self.boardname = boardname.lower()
        self.timeout = timeout
        # TODO: parse boardname, etc. in separate method
        if self.boardname in ["llnlv1", "v1", "1", 1]:
            self.boardname = "llnl_v1"
        if self.boardname in ["llnlv4", "v4", "4", 4]:
            self.boardname = "llnl_v4"
        self.commname = commname.lower()
        if self.commname[0] == "g" or self.commname[0] == "e":
            self.commname = "gige"
        if self.commname[0] == "r":
            self.commname = "rs422"
        self.sensorname = sensorname.lower()
        if self.sensorname in ["i1", "ic1", "icarus1"]:
            self.sensorname = "icarus"
        if self.sensorname in ["i2", "ic2"]:
            self.sensorname = "icarus2"
        if self.sensorname == "d":
            self.sensorname = "daedalus"
        self.verbose = int(verbose)
        self.port = port
        self.python, self.pyth1, self.pyth2, _, _ = sys.version_info
        self.PY3 = self.python >= 3
        self.platform = platform.system()
        self.arch, _ = platform.architecture()

        self.FPGAVersion = ""
        self.FPGANum = ""
        # FPGA information here and below populated during initialization using
        #   getBoardInfo
        self.FPGAboardtype = ""
        self.FPGArad = False
        self.FPGAsensor = ""
        self.FPGAinterfaces = []

        # indicates invalid FPGA information in register# (0x80000001 accepted as valid)
        self.FPGAinvalid = False

        self.iplist = None
        self.packageroot = os.path.dirname(inspect.getfile(CameraAssembler))
        self.armed = False

        # only one of these collections (senstiming, sensmanual) should be nonempty at
        #   any given time
        self.senstiming = {}  # preserve HST setting against possible power failure
        self.sensmanual = []  # preserve manual timing
        self.inittime = 0
        self.padToFull = False
        self.abort = False

        self.verbmap = {
            0: 99,
            1: logging.CRITICAL,
            2: logging.ERROR,
            3: logging.WARNING,
            4: logging.INFO,
            5: logging.DEBUG,
        }
        if logtag is None:
            logtag = ""
        self.logtag = logtag

        self.logcritbase = "CRITICAL {logtag}: ".format(logtag=logtag)
        self.logerrbase = "ERROR {logtag}: ".format(logtag=logtag)
        self.logwarnbase = "WARNING {logtag}: ".format(logtag=logtag)
        self.loginfobase = "INFO {logtag}: ".format(logtag=logtag)
        self.logdebugbase = "DEBUG {logtag}: ".format(logtag=logtag)

        self.logcrit = "{lb}[CA]".format(lb=self.logcritbase)
        self.logerr = "{lb}[CA]".format(lb=self.logerrbase)
        self.logwarn = "{lb}[CA]".format(lb=self.logwarnbase)
        self.loginfo = "{lb}[CA]".format(lb=self.loginfobase)
        self.logdebug = "{lb}[CA]".format(lb=self.logdebugbase)

        self.verblevel = self.verbmap.get(verbose, 5)  # defaults to 5 for invalid entry

        if logfile:
            logging.basicConfig(format="%(message)s", filename=logfile)
        else:
            logging.basicConfig(format="%(message)s")
        logging.getLogger().setLevel(self.verblevel)
        logging.getLogger("matplotlib.font_manager").disabled = True
        logging.debug(
            "{logdebug}CameraAssembler: boardname = {boardname}; commname = {commname};"
            " sensorname = {sensorname}; verbose = {verbose}; port = {port}; ip = {ip};"
            " logfile = {logfile}; logtag = {logtag}".format(
                logdebug=self.logdebug,
                boardname=boardname,
                commname=commname,
                sensorname=sensorname,
                verbose=verbose,
                port=port,
                ip=ip,
                logfile=logfile,
                logtag=logtag,
            )
        )

        if ip:
            try:
                iphex = socket.inet_aton(ip)
            except socket.error:
                logging.critical(
                    "{logcrit}CameraAssembler: invalid IP provided".format(
                        logcrit=self.logcrit
                    )
                )
                sys.exit(1)
            ipnum = [0, 0, 0, 0]
            for i in range(4):
                if self.PY3:
                    ipnum[i] = iphex[i]
                else:
                    ipnum[i] = int(iphex[i].encode("hex"), 16)
            self.iplist = ipnum

        self.payloaderror = False

        # code pulled out of __init__ to facilitate reinitialization of the board
        #   without needing to instantiate a new CameraAssembler object
        self.initialize()

```

6.1.3 Member Function Documentation

6.1.3.1 abortReadoff()

```
nsCamera.CameraAssembler.CameraAssembler.abortReadoff (
              self,
              flag = True )
```

```
Simple abort command for readoff in waiting mode--does not interrupt download in
   progress. Requires external threading to function. *WARNING* if not
   intercepted by active readoff command, will terminate next readoff command
   immediately at inception.
Args:
    flag: Sets passive abort flag read by readoff command
Returns:
    boolean: updated setting of flag
```

Definition at line 1989 of file CameraAssembler.py.

```
    def abortReadoff(self, flag=True):
        """
        Simple abort command for readoff in waiting mode--does not interrupt download in
           progress. Requires external threading to function. *WARNING* if not
           intercepted by active readoff command, will terminate next readoff command
           immediately at inception.
        Args:
            flag: Sets passive abort flag read by readoff command
        Returns:
            boolean: updated setting of flag
        """
        logging.info(self.loginfo + "abortReadoff")
        self.abort = flag
        return flag

```

6.1.3.2 arm()

```
nsCamera.CameraAssembler.CameraAssembler.arm (
              self,
              mode = None )
```

Definition at line 452 of file CameraAssembler.py.

```
    def arm(self, mode=None):
        return self.comms.arm(mode)

```

6.1.3.3 batchAcquire()

```
nsCamera.CameraAssembler.CameraAssembler.batchAcquire (
              self,
              sets = 1,
              trig = "Hardware",
              path = None,
              filename = "Frame",
              prefix = None,
              showProgress = 0 )
```

```
Acquire a series of images as fast as possible, then process and save to disk.
*WARNING* This method stores images in RAM, so the number of sets that can be
  acquired in a single call is limited by available memory.

Args:
    sets: Number of acquisitions to perform
    trig: trigger type; 'hardware', 'software', or 'dual'
    path: save path, defaults to './output'
    filename: defaults to 'frames.bin'
    prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')
      DO NOT USE unless providing a varying value (a fixed prefix will cause
      overwriting)
    showProgress: if non-zero, show notice every 'showProgress' acquisitions and
      print total acquisition time

Returns:
    Time taken for acquisition (seconds)
```

Definition at line 2004 of file CameraAssembler.py.

```
    ):
        """
        Acquire a series of images as fast as possible, then process and save to disk.
        *WARNING* This method stores images in RAM, so the number of sets that can be
          acquired in a single call is limited by available memory.

        Args:
            sets: Number of acquisitions to perform
            trig: trigger type; 'hardware', 'software', or 'dual'
            path: save path, defaults to './output'
            filename: defaults to 'frames.bin'
            prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')
              DO NOT USE unless providing a varying value (a fixed prefix will cause
              overwriting)
            showProgress: if non-zero, show notice every 'showProgress' acquisitions and
              print total acquisition time

        Returns:
            Time taken for acquisition (seconds)
        """
        logging.debug(
            self.logdebug
            + "batchAcquire: sets = "
            + str(sets)
            + "; trig = "
            + str(trig)
            + "; path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
            + "; showProgress = "
            + str(showProgress)
        )
        datalist = ["0"] * sets
        timelist = [datetime.now()] * sets
        logging.info(
            self.loginfo
            + "batchAcquire: temporarily disabling warning and information logging "
        )
        logging.getLogger().setLevel(self.verbmap.get(2))
        beforeread = time.time()
        for i in range(sets):
            if showProgress and not (i + 1) % showProgress:
                print(self.loginfo + "batchAcquire: Acquiring set " + str(i + 1))
            self.arm(trig)
            data, datalen, data_err = self.readoff(fast=True)
            datalist[i] = data
            timelist[i] = datetime.now()
        afterread = time.time()
        if showProgress:
            print(
                self.loginfo
                + "batchAcquire: "
                + str(afterread - beforeread)
                + " seconds for "
                + str(sets)
                + " sets"
            )
        setnum = 0
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        for imset, imtime in zip(datalist, timelist):
            setnum = setnum + 1
            if showProgress and not setnum % showProgress:
                print(self.loginfo + "batchAcquire: Saving set " + str(setnum))
            parsed = generateFrames(self, imset)
            if prefix is None:
                setprefix = imtime.strftime("%y%m%d-%H%M%S%f")[:-2] + "_"
            else:
                setprefix = prefix
            self.saveTiffs(parsed, path, filename, prefix=setprefix)
        logging.getLogger().setLevel(self.verblevel)
        logging.info(self.loginfo + "batchAcquire: re-enabling logging")
        return afterread - beforeread

```

6.1.3.4 bytes2str()

```
nsCamera.CameraAssembler.CameraAssembler.bytes2str (
              self,
              bytesequence )
```

Definition at line 486 of file CameraAssembler.py.

```
    def bytes2str(self, bytesequence):
        return bytes2str(bytesequence)

```

6.1.3.5 checkCRC()

```
nsCamera.CameraAssembler.CameraAssembler.checkCRC (
              self,
              rval )
```

Definition at line 480 of file CameraAssembler.py.

```
    def checkCRC(self, rval):
        return checkCRC(rval)

```

6.1.3.6 checkRegSet()

```
nsCamera.CameraAssembler.CameraAssembler.checkRegSet (
              self,
              regname,
              teststring )
```

```
Quick check to confirm that data read from register matches data write

Args:
    regname: register to test
    teststring: value to assign to register, as integer or hexadecimal string
      with or without '0x'

Returns:
    boolean, True if read and write values match
```

Definition at line 1662 of file CameraAssembler.py.

```
    def checkRegSet(self, regname, teststring):
        """
        Quick check to confirm that data read from register matches data write

        Args:
            regname: register to test
            teststring: value to assign to register, as integer or hexadecimal string
              with or without '0x'

        Returns:
            boolean, True if read and write values match
        """
        self.setRegister(regname, teststring)
        # tell board to send data; wait to clear before interrogating register contents
        if regname == "SRAM_CTL":
            time.sleep(2)
            if self.commname == "rs422":
                logging.info(
                    self.loginfo + "skipping 'SRAM_CTL' register check for RS422"
```

```
01681                 )
01682                 return True
01683         else:
01684             time.sleep(0.1)
01685         temp = self.getRegister(regname)
01686         resp = temp[1].upper()
01687         if resp != teststring.upper():
01688             logging.error(
01689                 self.logerr
01690                 + "checkRegSet failure: "
01691                 + regname
01692                 + " ; set: "
01693                 + teststring
01694                 + " ; read: "
01695                 + resp
01696             )
01697             return False
01698         return True
01699
```

6.1.3.7 checkSensorVoltStat()

```
nsCamera.CameraAssembler.CameraAssembler.checkSensorVoltStat (
                self )
```

Definition at line 398 of file CameraAssembler.py.

```
00398     def checkSensorVoltStat(self):
00399         return self.sensor.checkSensorVoltStat()
00400
```

6.1.3.8 checkStatus()

```
nsCamera.CameraAssembler.CameraAssembler.checkStatus (
                self )
```

Definition at line 383 of file CameraAssembler.py.

```
00383     def checkStatus(self):
00384         return self.board.checkStatus()
00385
```

6.1.3.9 checkStatus2()

```
nsCamera.CameraAssembler.CameraAssembler.checkStatus2 (
                self )
```

Definition at line 386 of file CameraAssembler.py.

```
00386     def checkStatus2(self):
00387         return self.board.checkStatus2()
00388
```

6.1.3.10 clearStatus()

```
nsCamera.CameraAssembler.CameraAssembler.clearStatus (
                self )
```

Definition at line 380 of file CameraAssembler.py.

```
00380     def clearStatus(self):
00381         return self.board.clearStatus()
00382
```

6.1.3.11 closeDevice()

```
nsCamera.CameraAssembler.CameraAssembler.closeDevice (
              self )
```

Definition at line 468 of file CameraAssembler.py.

```
    def closeDevice(self):
        return self.comms.closeDevice()

```

6.1.3.12 configADCs()

```
nsCamera.CameraAssembler.CameraAssembler.configADCs (
              self )
```

Definition at line 341 of file CameraAssembler.py.

```
    def configADCs(self):
        return self.board.configADCs()

```

6.1.3.13 disarm()

```
nsCamera.CameraAssembler.CameraAssembler.disarm (
              self )
```

Definition at line 344 of file CameraAssembler.py.

```
    def disarm(self):
        return self.board.disarm()

```

6.1.3.14 dumpNumpy()

```
nsCamera.CameraAssembler.CameraAssembler.dumpNumpy (
              self,
              datastream,
              path = None,
              filename = "Dump",
              prefix = None )
```

```
Datastream is converted directly to numpy array and saved to disk. No attempt to
  parse headers or separate into individual frames is made. The packet header is
  removed before saving

Args:
    datastream: string to be saved
    path: save path, defaults to './output'
    filename: defaults to 'Dump'
    prefix: prepended to 'filename', defaults to time/date
      (e.g. '160830-124704_')

Returns:
    Error string
```

Definition at line 1610 of file CameraAssembler.py.

```
    ):
        """
        Datastream is converted directly to numpy array and saved to disk. No attempt to
          parse headers or separate into individual frames is made. The packet header is
          removed before saving

        Args:
            datastream: string to be saved
            path: save path, defaults to './output'
            filename: defaults to 'Dump'
            prefix: prepended to 'filename', defaults to time/date
              (e.g. '160830-124704_')

        Returns:
            Error string
        """
        logging.info(self.loginfo + "dumpNumpy")
        logging.debug(
            self.logdebug
            + "dumpNumpy: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
        )
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = time.strftime("%y%m%d-%H%M%S_", time.localtime())
        if not os.path.exists(path):
            os.makedirs(path)
        npdata = str2nparray(datastream[36:])
        try:
            nppath = os.path.join(path, prefix + filename + ".npy")
            np.save(nppath, npdata)
        except SystemExit:
            raise
        except KeyboardInterrupt:
            raise
        except Exception:
            err = self.logerr + "dumpNumpy: unable to save data stream"
            logging.error(err)
        return err

```

6.1.3.15 dumpRegisters()

```
nsCamera.CameraAssembler.CameraAssembler.dumpRegisters (
              self )
```

```
*DEPRECATED* use dumpStatus() instead

List contents of all registers in board.registers. *WARNING* some status flags
  will reset when read.

Returns:
    Sorted list: [register name (register address) : register contents as
      hexadecimal string without '0x']
```

Definition at line 1764 of file CameraAssembler.py.

```
    def dumpRegisters(self):
        """
        *DEPRECATED* use dumpStatus() instead

        List contents of all registers in board.registers. *WARNING* some status flags
          will reset when read.

        Returns:
            Sorted list: [register name (register address) : register contents as
              hexadecimal string without '0x']
        """
        dump = {}
        for key in self.board.registers.keys():
            err, rval = self.getRegister(key)
            dump[key] = rval
        reglistmax = int(max(self.board.registers.values()), 16)
        dumplist = [0] * (reglistmax + 1)
        for k, v in dump.items():
            regnum = self.board.registers[k]
            dumplist[int(regnum, 16)] = (
                "(" + regnum + ") {0:<24} {1}".format(k, v.upper())
            )
        reglist = [a for a in dumplist if a]
        return reglist

```

6.1.3.16 dumpStatus()

```
nsCamera.CameraAssembler.CameraAssembler.dumpStatus (
              self )
```

Definition at line 395 of file CameraAssembler.py.

```
    def dumpStatus(self):
        return self.board.dumpStatus()

```

6.1.3.17 dumpSubregisters()

```
nsCamera.CameraAssembler.CameraAssembler.dumpSubregisters (
              self )
```

```
*DEPRECATED* use dumpStatus() instead

List contents of all subregisters in board.channel_lookups and
  board.monitor_lookups.
*WARNING* some registers will reset when read; only the first subregister from
  such a register will return the correct value, the remainder will return zeros

Returns:
    dictionary  {subregister name : subregister contents as binary string
      without initial '0b'}
```

Definition at line 1789 of file CameraAssembler.py.

```
    def dumpSubregisters(self):
        """
        *DEPRECATED* use dumpStatus() instead

        List contents of all subregisters in board.channel_lookups and
          board.monitor_lookups.
        *WARNING* some registers will reset when read; only the first subregister from
          such a register will return the correct value, the remainder will return zeros

        Returns:
            dictionary  {subregister name : subregister contents as binary string
              without initial '0b'}
        """
        dump = {}
        for sub in self.board.subreglist:
            key = sub.name
            err, resp = self.getSubregister(key)
            if err:
                logging.warning(
                    self.logwarn + "dumpSubregisters: unable to read subregister " + key
                )
            val = hex(int(resp, 2))
            dump[key] = val
        return dump

```

6.1.3.18 enableLED()

```
nsCamera.CameraAssembler.CameraAssembler.enableLED (
              self,
              status = 1 )
```

Definition at line 362 of file CameraAssembler.py.

```
    def enableLED(self, status=1):
        return self.board.enableLED(status)

```

6.1.3.19 flattenlist()

```
nsCamera.CameraAssembler.CameraAssembler.flattenlist (
              self,
              mylist )
```

Definition at line 492 of file CameraAssembler.py.

```
    def flattenlist(self, mylist):
        return flattenlist(mylist)

```

6.1.3.20 getBoardInfo()

```
nsCamera.CameraAssembler.CameraAssembler.getBoardInfo (
              self )
```

```
Get board info from FPGA_NUM register. Returns error flag if register contents
  are invalid and tuple (board version number, rad tolerance flag, sensor name)

Returns:
    tuple (errorFlag, (board version, rad tolerance flag, sensor name))
```

Definition at line 643 of file CameraAssembler.py.

```
    def getBoardInfo(self):
        """
        Get board info from FPGA_NUM register. Returns error flag if register contents
          are invalid and tuple (board version number, rad tolerance flag, sensor name)

        Returns:
            tuple (errorFlag, (board version, rad tolerance flag, sensor name))
        """
        invalidFPGANum = False
        interfaces = []

        # TODO: move to new method (combine with parsing from initialize)
        if int(self.FPGANum[0], 16) & 8:
            if self.FPGANum[1] == "1":
                boardtype = "LLNLv1"
            elif self.FPGANum[1] == "4":
                boardtype = "LLNLv4"
            else:
                boardtype = "LLNLv?"
                invalidFPGANum = True
        else:
            boardtype = "SNLrevC"
            logging.warning(
                self.logwarn + "FPGA self-identifies as SNLrevC, which is not"
                " supported by this software "
            )
            invalidFPGANum = True
        self.FPGAboardtype = boardtype

        if int(self.FPGANum[6], 16) & 1:
            rad = True
        else:
            rad = False
        self.FPGArad = rad

        if self.FPGANum[7] == "1":
            sensor = "Icarus"
        elif self.FPGANum[7] == "2":
            sensor = "Daedalus"
        else:
            sensor = "Undefined"
            invalidFPGANum = True
        self.FPGAsensor = sensor

        if int(self.FPGANum[5], 16) & 1:
            interfaces.append("RS422")
        if int(self.FPGANum[5], 16) & 2:
            interfaces.append("GigE")
        self.FPGAinterfaces = interfaces

        if invalidFPGANum:
            if self.FPGANum == "80000001":
                invalidFPGANum = False
            else:
                logging.warning(self.logwarn + "FPGA self-identification is invalid")
        self.FPGAinvalid = invalidFPGANum

        return invalidFPGANum, (boardtype, rad, sensor)

```

6.1.3.21 getEnter()

```
nsCamera.CameraAssembler.CameraAssembler.getEnter (
                self,
                text )
```

Definition at line 477 of file CameraAssembler.py.

```
00477     def getEnter(self, text):
00478         return getEnter(text)
00479
```

6.1.3.22 getManualTiming()

```
nsCamera.CameraAssembler.CameraAssembler.getManualTiming (
                self )
```

Definition at line 416 of file CameraAssembler.py.

```
00416     def getManualTiming(self):
00417         return self.sensor.getManualTiming()
00418
```

6.1.3.23 getMonV()

```
nsCamera.CameraAssembler.CameraAssembler.getMonV (
                self,
                monname,
                errflag = False )
```

```
Reads voltage from monitor named or that associated with the pot named 'monname'

Args:
    monname: name of pot or monitor, e.g., VRST or MON_CH2 found in
      board.subreg_aliases or defined in board.subregisters
    errflag: if True, return tuple with error string

Returns:
    if errflag:
        tuple: (error string, float value of voltage measured by monitor)
    else:
        float value of voltage measured by monitor
```

Definition at line 1396 of file CameraAssembler.py.

```
    def getMonV(self, monname, errflag=False):
        """
        Reads voltage from monitor named or that associated with the pot named 'monname'

        Args:
            monname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of voltage measured by monitor)
            else:
                float value of voltage measured by monitor
        """
        logging.debug(
            self.logdebug
            + "getMonV: monname = "
            + str(monname)
            + "; errflag = "
            + str(errflag)
        )
        monname = monname.upper()
        if monname in self.board.subreg_aliases:
            monname = self.board.subreg_aliases[monname].upper()
        # else:
        for key, value in self.board.monitor_controls.items():
            if value == monname:
                monname = key
        if monname not in self.board.monitor_controls:
            if monname in self.board.subreglist:
                pass  # no change necessary
            else:
                err = (
                    self.logerr + "getMonV: invalid lookup " + monname + ", returning 0"
                )
                logging.error(err)
                if errflag:
                    return err, 0
                return 0
        err, monval = self.getPot(monname, errflag=True)
        logging.debug(self.logdebug + "getMonV: monval = " + str(monval))
        if err:
            logging.error(
                self.logerr + "getMonV: unable to read monitor value for " + monname
            )
        if self.board.ADC5_bipolar:
            if monval >= 0.5:
                monval -= 1  # handle negative measurements (two's complement)
            if errflag:
                return err, 2 * self.board.ADC5_mult * monval * self.board.VREF
            return 2 * self.board.ADC5_mult * monval * self.board.VREF
        else:
            if errflag:
                return err, self.board.ADC5_mult * monval * self.board.VREF
            return self.board.ADC5_mult * monval * self.board.VREF

```

6.1.3.24 getPot()

```
nsCamera.CameraAssembler.CameraAssembler.getPot (
              self,
              potname,
              errflag = False )
```

```
Retrieves value of pot or ADC monitor subregister, scaled to [0,1).

Args:
    potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
      board.subreg_aliases or defined in board.subregisters
    errflag: if True, return tuple with error string

Returns:
    if errflag:
        tuple: (error string, float value of subregister, scaled to [0,1) )
    else:
        float value of subregister, scaled to [0,1)
```

Definition at line 1013 of file CameraAssembler.py.

```
    def getPot(self, potname, errflag=False):
        """
        Retrieves value of pot or ADC monitor subregister, scaled to [0,1).

        Args:
            potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of subregister, scaled to [0,1) )
            else:
                float value of subregister, scaled to [0,1)
        """
        logging.debug(
            "{logdebug}getPot: potname = {potname}; errflag = {errflag}".format(
                logdebug=self.logdebug, potname=potname, errflag=errflag
            )
        )

        potname, potobj, _ = self.resolveSubreg(potname)
        if not potobj:
            err = "{logerr}getPot: invalid lookup: {potname}; returning 0".format(
                logerr=self.logerr, potname=potname
            )

            logging.error(err)
            if errflag:
                return err, 0
            return 0
        err, b_pot_value = self.getSubregister(potname)
        if err:
            err = "{logerr}getPot: unable to read subregister: {potname}".format(
                logerr=self.logerr, potname=potname
            )

        # convert binary string back to decimal
        f_reg_value = 1.0 * int(b_pot_value, 2)
        value = (f_reg_value - potobj.min) / (potobj.max - potobj.min)
        # logging.debug(self.logdebug + "getpot: value = " + str(value))

        logging.debug(
            "{logdebug}getpot: value =  {value}".format(
                logdebug=self.logdebug, value=value
            )
        )

        if errflag:
            return err, value
        return value

```

6.1.3.25 getPotV()

```
nsCamera.CameraAssembler.CameraAssembler.getPotV (
              self,
              potname,
              errflag = False )
```

```
Reads voltage _setting_ (not actual voltage) of specified pot

Args:
    potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
      board.subreg_aliases or defined in board.subregisters
    errflag: if True, return tuple with error string

Returns:
    if errflag:
        tuple: (error string, float value of pot voltage)
    else:
        float value of pot voltage
```

Definition at line 1155 of file CameraAssembler.py.

```
    def getPotV(self, potname, errflag=False):
        """
        Reads voltage _setting_ (not actual voltage) of specified pot

        Args:
            potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of pot voltage)
            else:
                float value of pot voltage
        """
        logging.debug(
            self.logdebug
            + "getPotV: potname = "
            + str(potname)
            + "; errflag = "
            + str(errflag)
        )
        potname, potobj, _ = self.resolveSubreg(potname)
        if not potobj:
            err = (
                self.logerr + "getPotV: invalid lookup: " + potname + " , returning 0 "
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        err, val = self.getPot(potname, errflag=True)
        logging.debug(self.logdebug + "getPotV: val = " + str(val))
        if err:
            logging.error(self.logerr + "getPotV: unable to read pot " + potname)
        minV = potobj.minV
        maxV = potobj.maxV
        if errflag:
            return err, val * (maxV - minV)
        return val * (maxV - minV)
```

6.1.3.26 getPressure()

```
nsCamera.CameraAssembler.CameraAssembler.getPressure (
              self,
              offset = None,
              sensitivity = None,
              units = None )
```

Definition at line 377 of file CameraAssembler.py.

```
    def getPressure(self, offset=None, sensitivity=None, units=None):
        return self.board.getPressure(offset, sensitivity, units)

```

6.1.3.27 getRegister()

```
nsCamera.CameraAssembler.CameraAssembler.getRegister (
                self,
                regname )
```

```
Retrieves contents of named register as hexadecimal string without '0x'

Args:
    regname: name of register as given in ICD

Returns:
    tuple: (error string, register contents as hexadecimal string without '0x')
```

Definition at line 702 of file CameraAssembler.py.

```
    def getRegister(self, regname):
        """
        Retrieves contents of named register as hexadecimal string without '0x'

        Args:
            regname: name of register as given in ICD

        Returns:
            tuple: (error string, register contents as hexadecimal string without '0x')
        """
        # logging.debug(self.logdebug + "getRegister: regname = " + str(regname))
        logging.debug(
            "{logdebug}getRegister: regname = {regname}".format(
                logdebug=self.logdebug, regname=regname
            )
        )

        regname = regname.upper()
        if regname not in self.board.registers:
            err = "{logerr}getRegister: Invalid register name: {regname}; returning"
            " zeros".format(logerr=self.logerr, regname=regname)
            logging.error(err)
            return err, "00000000"
        sendpkt = Packet(cmd="1", addr=self.board.registers[regname])
        err, rval = self.comms.sendCMD(sendpkt)
        if err:
            logging.error(
                "{logerr}getRegister: {regname}; {err}".format(
                    logerr=self.logerr, regname=regname, err=err
                )
            )

        retval = rval[8:16]
        logging.debug(
            "{logdebug}getRegister: retval = {retval}".format(
                logdebug=self.logdebug, retval=retval
            )
        )

        return err, retval

```

6.1.3.28 getSensTemp()

```
nsCamera.CameraAssembler.CameraAssembler.getSensTemp (
              self,
              scale = None,
              offset = None,
              slope = None,
              dec = 1 )
```

Definition at line 419 of file CameraAssembler.py.

```
    def getSensTemp(self, scale=None, offset=None, slope=None, dec=1):
        return self.sensor.getSensTemp(scale, offset, slope, dec)

```

6.1.3.29 getSubregister()

```
nsCamera.CameraAssembler.CameraAssembler.getSubregister (
              self,
              subregname )
```

```
Returns substring of register identified in board attribute 'subregname'

Args:
    subregname: listed in board.subreg_aliases or defined in board.subregisters

Returns:
    tuple: (error string, contents of subregister as binary string without '0b')
```

Definition at line 836 of file CameraAssembler.py.

```
    def getSubregister(self, subregname):
        """
        Returns substring of register identified in board attribute 'subregname'

        Args:
            subregname: listed in board.subreg_aliases or defined in board.subregisters

        Returns:
            tuple: (error string, contents of subregister as binary string without '0b')
        """
        logging.debug(
            "{logdebug}getSubegister: subregname = {subregname}".format(
                logdebug=self.logdebug,
                subregname=subregname,
            )
        )

        subregname, subregobj, _ = self.resolveSubreg(subregname)
        if not subregobj:
            err = "{logerr}getSubregister: invalid lookup: {subregname}; returning"
            " string of zeroes".format(logerr=self.logerr, subregname=subregname)

            logging.error(err)
            return err, "".zfill(8)
        err, resp = self.getRegister(subregobj.register)
        if err:
            logging.error(
                "{logerr}getSubregister: unable to retrieve register setting: \
                {subregname}; returning '0' string".format(
                    logerr=self.logerr, subregname=subregname
                )
            )

            return err, "".zfill(8)
        hex_str = "0x" + resp  # this should be a hexadecimalstring
        b_reg_value = "{0:0=32b}".format(int(hex_str, 16))  # convert to binary string
        # list indexing is reversed from bit string; the last bit of the string is at
        #   index 0 in the list (thus bit 0 is at index 0)
        startindex = 31 - subregobj.start_bit
        retval = b_reg_value[startindex : startindex + subregobj.width]
        logging.debug(
            "{logdebug}getSubregister: retval = {retval}".format(
                logdebug=self.logdebug, retval=retval
            )
        )
        return "", retval

```

6.1.3.30 getTemp()

```
nsCamera.CameraAssembler.CameraAssembler.getTemp (
                self,
                scale = None )
```

Definition at line 374 of file CameraAssembler.py.

```
    def getTemp(self, scale=None):
        return self.board.getTemp(scale)

```

6.1.3.31 getTimer()

```
nsCamera.CameraAssembler.CameraAssembler.getTimer (
                self )
```

Definition at line 356 of file CameraAssembler.py.

```
    def getTimer(self):
        return self.board.getTimer()

```

6.1.3.32 getTiming()

```
nsCamera.CameraAssembler.CameraAssembler.getTiming (
                self,
                side = None,
                actual = None )
```

Definition at line 407 of file CameraAssembler.py.

```
    def getTiming(self, side=None, actual=None):
        return self.sensor.getTiming(side, actual)

```

6.1.3.33 initBoard()

```
nsCamera.CameraAssembler.CameraAssembler.initBoard (
                self )
```

Aliases to other objects' methods TODO: properly delegate these methods.

Definition at line 329 of file CameraAssembler.py.

```
    def initBoard(self):
        return self.board.initBoard()

```

6.1.3.34 initialize()

```
nsCamera.CameraAssembler.CameraAssembler.initialize (
                self )
```

End aliases.

```
Initialize board registers and set pots
```

Definition at line 500 of file CameraAssembler.py.

```
00500     def initialize(self):
00501         """
00502         Initialize board registers and set pots
00503         """
00504         # TODO: automate sensor and board selection from firmware info
00505 
00507 
00508         # get sensor
00509         # TODO: pull sensor, board, comm id out to separate methods
00510         if self.sensorname == "icarus":
00511             import nsCamera.sensors.icarus as snsr
00512         elif self.sensorname == "icarus2":
00513             import nsCamera.sensors.icarus2 as snsr
00514         elif self.sensorname == "daedalus":
00515             import nsCamera.sensors.daedalus as snsr
00516         else:  # catch-all for added sensors to attempt object encapsulation
00517             sensormodname = ".sensors." + self.sensorname
00518             try:
00519                 sensormod = importlib.import_module(sensormodname, "nsCamera")
00520             except ImportError:
00521                 logging.critical(self.logcrit + "invalid sensor name")
00522                 sys.exit(1)
00523             snsr = getattr(sensormod, self.sensorname)
00524         self.sensor = snsr(self)
00525 
00526         # kill existing connections (for reinitialize)
00527         if hasattr(self, "comms"):
00528             self.closeDevice()
00529 
00530         # get communications interface
00531         if self.commname == "rs422":
00532             import nsCamera.comms.RS422 as comms
00533         elif self.commname == "gige":
00534             import nsCamera.comms.GigE as comms
00535         else:
00536             commsmodname = ".comms." + self.commname
00537             try:
00538                 commsmod = importlib.import_module(commsmodname, "nsCamera")
00539             except ImportError:
00540                 logging.critical(self.logcrit + "invalid comms name")
00541                 sys.exit(1)
00542             comms = getattr(commsmod, self.commname)
00543         self.comms = comms(self)
00544 
00545         # get board
00546         if self.boardname == "llnl_v1":
00547             import nsCamera.boards.LLNL_v1 as brd
00548 
00549             self.board = brd.llnl_v1(self)
00550         elif self.boardname == "llnl_v4":
00551             import nsCamera.boards.LLNL_v4 as brd
00552 
00553             self.board = brd.llnl_v4(self)
00554         else:
00555             boardmodname = ".board." + self.boardname
00556             try:
00557                 boardmod = importlib.import_module(boardmodname, "nsCamera")
00558             except ImportError:
00559                 logging.critical(self.logcrit + "invalid board name")
00560                 sys.exit(1)
00561             boardobj = getattr(boardmod, self.boardname)
00562             self.board = boardobj(self)
00563 
00564         # Now that board exists, initialize board-specific aliases for sensors
```

```
        self.sensor.init_board_specific()



        # TODO: make cython the standard version
        # ###############
        # # For cython version
        #
        # # get sensor
        # if self.sensorname == "icarus":
        #     import nsCamera.sensors.icarus as snsr
        #     self.sensor = snsr.icarus(self)
        # elif self.sensorname == "icarus2":
        #     import nsCamera.sensors.icarus2 as snsr
        #     self.sensor = snsr.icarus2(self)
        # elif self.sensorname == "daedalus":
        #     import nsCamera.sensors.daedalus as snsr
        #     self.sensor = snsr.daedalus(self)
        #
        # # kill existing connections (for reinitialize)
        # if hasattr(self, "comms"):
        #     self.closeDevice()
        #
        # # get communications interface
        # if self.commname == "rs422":
        #     import nsCamera.comms.RS422 as comms
        #     self.comms = comms.RS422(self)
        # elif self.commname == "gige":
        #     import nsCamera.comms.GigE as comms
        #     self.comms = comms.GigE(self)
        #
        # # get board
        # if self.boardname == "llnl_v1":
        #     import nsCamera.boards.LLNL_v1 as brd
        #     self.board = brd.llnl_v1(self)
        # elif self.boardname == "llnl_v4":
        #     import nsCamera.boards.LLNL_v4 as brd
        #     self.board = brd.llnl_v4(self)
        # ###############

        err, rval = self.getRegister("FPGA_NUM")
        if err or rval == "":
            err, rval = self.getRegister("FPGA_NUM")
            if err or rval == "":
                logging.critical(
                    self.logcrit + "Initialization failed: unable to communicate with"
                    " board. "
                )
            sys.exit(1)

        self.initBoard()
        self.initPots()
        self.initSensor()
        self.initPowerCheck()
        self.getBoardInfo()
        self.printBoardInfo()

```

6.1.3.35 initPots()

```
nsCamera.CameraAssembler.CameraAssembler.initPots (
                self )
```

Definition at line 332 of file CameraAssembler.py.

```
    def initPots(self):
        return self.board.initPots()

```

6.1.3.36 initPowerCheck()

```
nsCamera.CameraAssembler.CameraAssembler.initPowerCheck (
                self )
```

```
Reset software and board timers for monitoring power status
```

Definition at line 1700 of file CameraAssembler.py.

```
    def initPowerCheck(self):
        """
        Reset software and board timers for monitoring power status
        """
        self.inittime = time.time()
        logging.info(self.loginfo + "resetting timer for power check function")
        self.resetTimer()

```

6.1.3.37 initSensor()

```
nsCamera.CameraAssembler.CameraAssembler.initSensor (
              self )
```

Definition at line 338 of file CameraAssembler.py.

```
    def initSensor(self):
        return self.board.initSensor()

```

6.1.3.38 latchPots()

```
nsCamera.CameraAssembler.CameraAssembler.latchPots (
              self )
```

Definition at line 335 of file CameraAssembler.py.

```
    def latchPots(self):
        return self.board.latchPots()

```

6.1.3.39 mmReadoff()

```
nsCamera.CameraAssembler.CameraAssembler.mmReadoff (
              self,
              waitOnSRAM,
              variation = None )
```

```
Convenience function for parsing frames for use by MicroManager plugin
Args:
    waitOnSRAM: readoff wait flag
    variation: format of frames generated from readoff
        default - return first frame only
        "LastFrame" - return last frame only
        "Average" - provide average of frames as single frame
        "Landscape" - stitch frames together horizontally into single wide frame

Returns:
    ndarray - single image frame
```

Definition at line 1814 of file CameraAssembler.py.

```
    def mmReadoff(self, waitOnSRAM, variation=None):
        """
        Convenience function for parsing frames for use by MicroManager plugin
        Args:
            waitOnSRAM: readoff wait flag
            variation: format of frames generated from readoff
                default - return first frame only
                "LastFrame" - return last frame only
                "Average" - provide average of frames as single frame
                "Landscape" - stitch frames together horizontally into single wide frame

        Returns:
            ndarray - single image frame
        """
        frames, datalen, data_err = self.readoff(waitOnSRAM)
        if variation == "LastFrame":
            return frames[self.sensor.nframes - 1]
        elif variation == "Average":
            return np.sum(frames, axis=0) // self.sensor.nframes
        elif variation == "Landscape":
            shaped = [
                np.reshape(frame, (self.sensor.maxheight, self.sensor.maxwidth))
                for frame in frames
            ]
            return np.concatenate(shaped, axis=1)
        else:
            return frames[0]

```

6.1.3.40 parseReadoff()

```
nsCamera.CameraAssembler.CameraAssembler.parseReadoff (
                self,
                frames,
                columns = 1 )
```

Definition at line 446 of file CameraAssembler.py.

```
    def parseReadoff(self, frames, columns=1):
        return self.sensor.parseReadoff(frames, columns)

```

6.1.3.41 partition()

```
nsCamera.CameraAssembler.CameraAssembler.partition (
                self,
                frames,
                columns )
```

Definition at line 495 of file CameraAssembler.py.

```
    def partition(self, frames, columns):
        return partition(self, frames, columns)

```

6.1.3.42 plotFrames()

```
nsCamera.CameraAssembler.CameraAssembler.plotFrames (
                self,
                frames,
                index = None )
```

Definition at line 474 of file CameraAssembler.py.

```
    def plotFrames(self, frames, index=None):
        return plotFrames(self, frames, index)

```

6.1.3.43 powerCheck()

```
nsCamera.CameraAssembler.CameraAssembler.powerCheck (
              self,
              delta = 10 )
```

```
Check to see if board power has persisted since powerCheck was last initialized.
  Compares time elapsed since initialization against board’s timer. If the
  difference is greater than ’delta,’ flag as False (power has likely failed)

Args:
    delta: difference in seconds permitted between software and board timers

Returns:
    boolean, ’True’ means timer difference is less than ’delta’ parameter;
             ’False’ indicates power failure
```

Definition at line 1708 of file CameraAssembler.py.

```
    def powerCheck(self, delta=10):
        """
        Check to see if board power has persisted since powerCheck was last initialized.
          Compares time elapsed since initialization against board’s timer. If the
          difference is greater than ’delta,’ flag as False (power has likely failed)

        Args:
            delta: difference in seconds permitted between software and board timers

        Returns:
            boolean, ’True’ means timer difference is less than ’delta’ parameter;
                     ’False’ indicates power failure
        """
        elapsed = time.time() - self.inittime
        logging.debug(self.logdebug + "powerCheck: elapsed time = " + str(elapsed))
        difference = abs(elapsed - self.getTimer())
        if difference > delta:
            logging.warning(
                self.logwarn + "powerCheck function has failed; may indicate current "
                "or recent power failure "
            )
        return difference < delta

```

6.1.3.44 printBoardInfo()

```
nsCamera.CameraAssembler.CameraAssembler.printBoardInfo (
              self )
```

Definition at line 1731 of file CameraAssembler.py.

```
    def printBoardInfo(self):
        # TODO: add override option if logging level is above info
        logging.info(
            self.loginfo
            + "Python version: "
            + str(self.python)
            + "."
            + str(self.pyth1)
            + "."
            + str(self.pyth2)
        )
        logging.info(self.loginfo + "nsCamera software version: " + self.version)
        logging.info(self.loginfo + "FPGA firmware version: " + self.FPGAVersion)
        logging.info(self.loginfo + "FPGA implementation: " + self.FPGANum)
        if self.FPGAinvalid:
            logging.info(self.loginfo + "FPGA information unavailable")
        else:
            logging.info(self.loginfo + "Board type: " + self.FPGAboardtype)
            logging.info(self.loginfo + "Rad-Tolerant: " + str(self.FPGArad))
            logging.info(self.loginfo + "Sensor family: " + self.FPGAsensor)
            logging.info(self.loginfo + "Sensor label: " + self.sensor.loglabel)
            logging.info(
                self.loginfo + "Available interfaces: " + ", ".join(self.FPGAinterfaces)
            )
        if self.commname == "gige":
            ci = self.comms.CardInfoP.contents
            ip = ".".join(str(e) for e in [b for b in ci.IPAddr])
            logging.info(
                self.loginfo + "GigE connected to " + ip + ":" + str(self.port)
            )
        elif self.commname == "rs422":
            logging.info(self.loginfo + "RS422 connected to " + self.comms.port)

```

6.1.3.45 readFrames()

```
nsCamera.CameraAssembler.CameraAssembler.readFrames (
                self,
                waitOnSRAM = None,
                timeout = 0,
                fast = False,
                columns = 1 )
```

Definition at line 455 of file CameraAssembler.py.

```
    def readFrames(self, waitOnSRAM=None, timeout=0, fast=False, columns=1):
        frames, _, _ = self.comms.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

```

6.1.3.46 readImgs()

```
nsCamera.CameraAssembler.CameraAssembler.readImgs (
                self,
                waitOnSRAM = True,
                mode = "Hardware" )
```

```
Combines arm() and readoff() functions

Returns:
    tuple (list of numpy arrays, length of downloaded payload, payload error
      flag) returned by readoff
```

Definition at line 1453 of file CameraAssembler.py.

```
    def readImgs(self, waitOnSRAM=True, mode="Hardware"):
        """
        Combines arm() and readoff() functions

        Returns:
            tuple (list of numpy arrays, length of downloaded payload, payload error
              flag) returned by readoff
        """
        logging.info(self.loginfo + "readImgs")
        self.arm(mode)
        return self.readoff(waitOnSRAM)

```

6.1.3.47 readoff()

```
nsCamera.CameraAssembler.CameraAssembler.readoff (
                self,
                waitOnSRAM = None,
                timeout = 0,
                fast = None,
                columns = 1 )
```

Definition at line 459 of file CameraAssembler.py.

```
    def readoff(self, waitOnSRAM=None, timeout=0, fast=None, columns=1):
        return self.comms.readoff(waitOnSRAM, timeout, fast, columns)

```

6.1.3.48 readSerial()

```
nsCamera.CameraAssembler.CameraAssembler.readSerial (
                self,
                size,
                timeout = None )
```

Definition at line 465 of file CameraAssembler.py.

```
    def readSerial(self, size, timeout=None):
        return self.comms.readSerial(size, timeout)

```

6.1.3.49 readSRAM()

```
nsCamera.CameraAssembler.CameraAssembler.readSRAM (
                self )
```

Definition at line 350 of file CameraAssembler.py.

```
    def readSRAM(self):
        return self.board.readSRAM()

```

6.1.3.50 reboot()

```
nsCamera.CameraAssembler.CameraAssembler.reboot (
                self )
```

```
Perform soft reboot on board and reinitialize
```

Definition at line 636 of file CameraAssembler.py.

```
    def reboot(self):
        """
        Perform soft reboot on board and reinitialize
        """
        self.board.softReboot()
        self.reinitialize()

```

6.1.3.51 reinitialize()

```
nsCamera.CameraAssembler.CameraAssembler.reinitialize (
              self )
```

```
Reinitialize board registers and pots, reinitialize sensor timing (if
  previously set)
```

Definition at line 622 of file CameraAssembler.py.

```
    def reinitialize(self):
        """
        Reinitialize board registers and pots, reinitialize sensor timing (if
          previously set)
        """
        logging.info(self.loginfo + "reinitializing")
        self.initialize()

        for side in self.senstiming:
            self.setTiming(side, self.senstiming[side][0], self.senstiming[side][1])

        if self.sensmanual:  # should be mutually exclusive with anything in senstiming
            self.setManualShutters(self.sensmanual)

```

6.1.3.52 reportEdgeDetects()

```
nsCamera.CameraAssembler.CameraAssembler.reportEdgeDetects (
              self )
```

Definition at line 392 of file CameraAssembler.py.

```
    def reportEdgeDetects(self):
        return self.board.reportEdgeDetects()

```

6.1.3.53 reportStatus()

```
nsCamera.CameraAssembler.CameraAssembler.reportStatus (
              self )
```

Definition at line 389 of file CameraAssembler.py.

```
    def reportStatus(self):
        return self.board.reportStatus()

```

6.1.3.54 resetTimer()

```
nsCamera.CameraAssembler.CameraAssembler.resetTimer (
              self )
```

Definition at line 359 of file CameraAssembler.py.

```
    def resetTimer(self):
        return self.board.resetTimer()

```

6.1.3.55 resolveSubreg()

```
nsCamera.CameraAssembler.CameraAssembler.resolveSubreg (
              self,
              srname )
```

```
Resolves subregister name or alias, returns object associated with subregister
  and flag indicating writability

Args:
    srname: name or alias of subregister

Returns:
    tuple(subregister name string, associated object, writable flag)
```

Definition at line 801 of file CameraAssembler.py.

```
    def resolveSubreg(self, srname):
        """
        Resolves subregister name or alias, returns object associated with subregister
          and flag indicating writability

        Args:
            srname: name or alias of subregister

        Returns:
            tuple(subregister name string, associated object, writable flag)
        """
        logging.debug(
            "{logdebug}resolveSubreg: srname = {srname}".format(
                logdebug=self.logdebug,
                srname=srname,
            )
        )
        writable = False
        srname = srname.upper()
        if srname in self.board.subreg_aliases:
            srname = self.board.subreg_aliases[srname].upper()
        if srname in self.board.subreglist:
            srobj = getattr(self.board, srname)
            writable = getattr(self.board, srname).writable
        else:
            # No-object error is handled by calling function
            srobj = None
        logging.debug(
            "{logdebug}resolveSubreg: srobj = {srobj}, writable={writable}".format(
                logdebug=self.logdebug, srobj=srobj, writable=writable
            )
        )

        return srname, srobj, writable

```

6.1.3.56 saveFrames()

```
nsCamera.CameraAssembler.CameraAssembler.saveFrames (
              self,
              frames,
              path = None,
              filename = "frames",
              prefix = None )
```

```
Save list of numpy arrays to disk. If passed an unprocessed text string, saves
  it directly to disk for postprocessing. Use 'prefix=""' for no prefix

Args:
    frames: numpy array or list of numpy arrays OR text string
    path: save path, defaults to './output'
    filename: defaults to 'frames.bin'
    prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')

Returns:
    Error string
```

Definition at line 1465 of file CameraAssembler.py.

```
    def saveFrames(self, frames, path=None, filename="frames", prefix=None):
        """
        Save list of numpy arrays to disk. If passed an unprocessed text string, saves
          it directly to disk for postprocessing. Use 'prefix=""' for no prefix

        Args:
            frames: numpy array or list of numpy arrays OR text string
            path: save path, defaults to './output'
            filename: defaults to 'frames.bin'
            prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')

        Returns:
            Error string
        """
        logging.debug(
            self.logdebug
            + "saveFrames: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
        )
        logging.info(self.loginfo + "saveFrames")
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)

        # TODO catch save file exceptions
        if isinstance(frames[0], str):
            logging.debug(self.logdebug + "saveFrames: saving text frames")
            filename = filename + ".txt"
            savefile = open(os.path.join(path, prefix + filename), "w+")
            savefile.write(frames)
        else:
            logging.debug(self.logdebug + "saveFrames: saving numerical frames")
            filename = filename + ".bin"
            stacked = np.stack(frames)
            try:
                stacked = stacked.reshape(
                    (
                        self.sensor.nframes,
                        self.sensor.height // (self.sensor.interlacing + 1),
                        self.sensor.width,
                    )
                )
            except Exception as e:
                err = self.logerr + "saveFrames: unable to save frames: " + str(e)
                logging.error(err)

            stacked.tofile(os.path.join(path, prefix + filename))
        return err

```

6.1.3.57 saveHDF()

```
nsCamera.CameraAssembler.CameraAssembler.saveHDF (
                self,
                frames,
                path = None,
                filename = "Acquisition",
                prefix = None )
```

Definition at line 2092 of file CameraAssembler.py.

```
    ):
        """ """
        logging.info(self.loginfo + ": saveHDF")
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)

        h5file = os.path.join(path, prefix + filename + ".hdf5")
        with h5py.File(h5file, "w") as f:
            # shotgrp = f.create_group("DATA/SHOT")
            frame_index = 0
            for frame in frames:
                grp = f.create_group("DATA/SHOT/FRAME_0" + str(frame_index))
                data = grp.create_dataset(
                    "DATA", (self.sensor.height, self.sensor.width), data=frame
                )
                frame_index += 1


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.1.3.58 saveNumpys()

```
nsCamera.CameraAssembler.CameraAssembler.saveNumpys (
                self,
                frames,
                path = None,
                filename = "Frame",
                prefix = None,
                index = None )
```

```
Save numpy array or list of numpy arrays to disk as individual numpy data files,
  with frame number appended to filename.

Args:
    frames: numpy array or list of numpy arrays or single numpy array
    path: save path, defaults to './output'
    filename: defaults to 'Frame' followed by frame number
    prefix: prepended to 'filename', defaults to time/date
      (e.g. '160830-124704_')
    index: number to start frame numbering

Returns:
    Error string
```

Definition at line 1522 of file CameraAssembler.py.

```
    ):
        """
        Save numpy array or list of numpy arrays to disk as individual numpy data files,
          with frame number appended to filename.

        Args:
            frames: numpy array or list of numpy arrays or single numpy array
            path: save path, defaults to './output'
            filename: defaults to 'Frame' followed by frame number
            prefix: prepended to 'filename', defaults to time/date
              (e.g. '160830-124704_')
            index: number to start frame numbering

        Returns:
            Error string
        """
        logging.info(self.loginfo + "saveNumpys")
        logging.debug(
            self.logdebug
            + "saveNumpys: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
            + "; index = "
            + str(index)
        )
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)
        if index is None:
            firstnum = self.sensor.firstframe
        else:
            firstnum = index
        if not isinstance(frames, list):
            frames = [frames]

        # if this is a text string from fast readoff, do the numpy conversion now
        if isinstance(frames[0], str):
            frames = generateFrames(frames)

        framestemp = np.copy(frames)

        for idx, frame in enumerate(framestemp):
            if idx < len(framestemp) / 2:
                interlacing = self.sensor.interlacing[0]
            else:
                interlacing = self.sensor.interlacing[1]
            try:
                if self.padToFull:
                    frame = np.reshape(
                        frame, (self.sensor.maxheight // (interlacing + 1), -1)
                    )
                else:
                    frame = np.reshape(
                        frame,
                        (
                            (self.sensor.lastrow - self.sensor.firstrow + 1)
                            // (interlacing + 1),
                            -1,
                        ),
                    )
                namenum = filename + "_%d" % firstnum
                nppath = os.path.join(path, prefix + namenum + ".npy")
                np.save(nppath, frame)
                firstnum += 1
            except SystemExit:
                raise
            except KeyboardInterrupt:
                raise
            except Exception:
                err = self.logerr + "saveNumpys: unable to save arrays"
                logging.error(err)
                continue
        return err
```

```
01609
```

6.1.3.59 saveTiffs()

```
nsCamera.CameraAssembler.CameraAssembler.saveTiffs (
              self,
              frames,
              path = None,
              filename = "Frame",
              prefix = None,
              index = None )
```

Definition at line 471 of file CameraAssembler.py.

```
    def saveTiffs(self, frames, path=None, filename="Frame", prefix=None, index=None):
        return saveTiffs(self, frames, path, filename, prefix, index)

```

6.1.3.60 selectOscillator()

```
nsCamera.CameraAssembler.CameraAssembler.selectOscillator (
              self,
              osc = None )
```

Definition at line 425 of file CameraAssembler.py.

```
    def selectOscillator(self, osc=None):
        return self.sensor.selectOscillator(osc)

```

6.1.3.61 sendCMD()

```
nsCamera.CameraAssembler.CameraAssembler.sendCMD (
              self,
              pkt )
```

Definition at line 449 of file CameraAssembler.py.

```
    def sendCMD(self, pkt):
        return self.comms.sendCMD(pkt)

```

6.1.3.62 sensorSpecific()

```
nsCamera.CameraAssembler.CameraAssembler.sensorSpecific (
              self )
```

Definition at line 422 of file CameraAssembler.py.

```
    def sensorSpecific(self):
        return self.sensor.sensorSpecific()

```

6.1.3.63 setArbTiming()

```
nsCamera.CameraAssembler.CameraAssembler.setArbTiming (
              self,
              side = "AB",
              sequence = None )
```

Definition at line 404 of file CameraAssembler.py.

```
    def setArbTiming(self, side="AB", sequence=None):
        return self.sensor.setArbTiming(side, sequence)

```

6.1.3.64 setExtClk()

```
nsCamera.CameraAssembler.CameraAssembler.setExtClk (
              self,
              dilation = None,
              frequency = None )
```

Definition at line 443 of file CameraAssembler.py.

```
    def setExtClk(self, dilation=None, frequency=None):
        return self.sensor.setExtClk(dilation, frequency)

```

6.1.3.65 setFrames()

```
nsCamera.CameraAssembler.CameraAssembler.setFrames (
              self,
              minframe = None,
              maxframe = None )
```

```
Sets bounds on frames returned by board, inclusive (e.g., 0,3 returns four
frames). If called without parameters, resets to full set of frames.

Args:
    minframe: first frame to read from board
    maxframe: last frame to read from board

Returns:
    Error string
```

Definition at line 1842 of file CameraAssembler.py.

```
    def setFrames(self, minframe=None, maxframe=None):
        """
        Sets bounds on frames returned by board, inclusive (e.g., 0,3 returns four
        frames). If called without parameters, resets to full set of frames.

        Args:
            minframe: first frame to read from board
            maxframe: last frame to read from board

        Returns:
            Error string
        """
        logging.debug(
            self.logdebug
            + "setFrames: minframe = "
            + str(minframe)
            + "; maxframe = "
            + str(maxframe)
        )
        if minframe is None:
            minframe = self.sensor.minframe
        if maxframe is None:
            maxframe = self.sensor.maxframe
        if (
            not isinstance(minframe, int)
            or minframe < self.sensor.minframe
            or minframe > maxframe
            or not isinstance(maxframe, int)
            or maxframe > self.sensor.maxframe
        ):
            err = (
                self.logerr + "setFrames: invalid frame limits submitted. Frame "
                "selection remains unchanged. "
            )
            logging.error(err)
            return err

        initframe = hex(minframe)[2:].zfill(8)
        finframe = hex(maxframe)[2:].zfill(8)
        err1, _ = self.setRegister("FPA_FRAME_INITIAL", initframe)
        err2, _ = self.setRegister("FPA_FRAME_FINAL", finframe)
        self.sensor.firstframe = minframe
        self.sensor.lastframe = maxframe
        self.sensor.nframes = maxframe - minframe + 1
        self.comms.payloadsize = (
            self.sensor.width
            * self.sensor.height
            * self.sensor.nframes
            * self.sensor.bytesperpixel
        )
        plural = ""
        if self.sensor.nframes > 1:
            plural = "s"
        logging.info(
            self.loginfo
            + "Readoff set to "
            + str(self.sensor.nframes)
            + " frame"
            + plural
            + " ("
            + str(minframe)
            + ", "
            + str(maxframe)
            + ")"
        )
        err = err1 + err2
        if err:
            logging.error(
                self.logerr + "setFrames may not have functioned properly: " + err
            )
        return err

```

6.1.3.66 setHighFullWell()

```
nsCamera.CameraAssembler.CameraAssembler.setHighFullWell (
                self,
                flag = True )
```

Definition at line 431 of file CameraAssembler.py.

```
    def setHighFullWell(self, flag=True):
        return self.sensor.setHighFullWell(flag)

```

6.1.3.67 setInterlacing()

```
nsCamera.CameraAssembler.CameraAssembler.setInterlacing (
                self,
                ifactor = None,
                side = None )
```

Definition at line 428 of file CameraAssembler.py.

```
    def setInterlacing(self, ifactor=None, side=None):
        return self.sensor.setInterlacing(ifactor, side)

```

6.1.3.68 setLED()

```
nsCamera.CameraAssembler.CameraAssembler.setLED (
            self,
            LED = 1,
            status = 1 )
```

Definition at line 365 of file CameraAssembler.py.

```
    def setLED(self, LED=1, status=1):
        return self.board.setLED(LED, status)

```

6.1.3.69 setManualShutters()

```
nsCamera.CameraAssembler.CameraAssembler.setManualShutters (
            self,
            timing = None )
```

Definition at line 410 of file CameraAssembler.py.

```
    def setManualShutters(self, timing=None):
        return self.sensor.setManualTiming(timing)

```

6.1.3.70 setManualTiming()

```
nsCamera.CameraAssembler.CameraAssembler.setManualTiming (
            self,
            timing = None )
```

Definition at line 413 of file CameraAssembler.py.

```
    def setManualTiming(self, timing=None):
        return self.sensor.setManualTiming(timing)

```

6.1.3.71 setPhiDelay()

```
nsCamera.CameraAssembler.CameraAssembler.setPhiDelay (
            self,
            side = None,
            delay = 0 )
```

Definition at line 440 of file CameraAssembler.py.

```
    def setPhiDelay(self, side=None, delay=0):
        return self.sensor.setPhiDelay(side, delay)

```

6.1.3.72 setPot()

```
nsCamera.CameraAssembler.CameraAssembler.setPot (
              self,
              potname,
              value = 1.0,
              errflag = False )
```

```
Sets value of pot to value, normalized so that  '1.0' corresponds with the
  fixed point maximum value of pot.

Args:
    potname: common name of pot, e.g., VRST found in board.subreg_aliases or
      defined in board.subregisters
    value: float between 0 and 1
    errflag: if True, return tuple with error string

Returns:
    if errflag:
        tuple: (error string, response packet as string)
    else:
        response packet as string
```

Definition at line 1065 of file CameraAssembler.py.

```
    def setPot(self, potname, value=1.0, errflag=False):
        """
        Sets value of pot to value, normalized so that  '1.0' corresponds with the
          fixed point maximum value of pot.

        Args:
            potname: common name of pot, e.g., VRST found in board.subreg_aliases or
              defined in board.subregisters
            value: float between 0 and 1
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, response packet as string)
            else:
                response packet as string
        """
        logging.debug(
            "{logdebug}setPot: potname = {potname}; value={value} errflag = {errflag}"
            "".format(
                logdebug=self.logdebug, potname=potname, value=value, errflag=errflag
            )
        )

        if value < 0:
            value = 0.0
        if value > 1:
            value = 1.0

        potname, potobj, writable = self.resolveSubreg(potname)
        if not potobj:
            err = "{logerr}setPot: invalid lookup: {potname}; returning '0'".format(
                logerr=self.logerr, potname=potname
            )

            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if not writable:
            err = "{logerr}setPot: not a writable subregister: {potname}; returning '0'"
            "".format(logerr=self.logerr, potname=potname)
            logging.error(err)
            if errflag:
                return err, "0"
            return "0"
        setpoint = int(round(value * potobj.max_value))
        setpointpadded = "{num:{fill}{width}b}".format(
            num=setpoint, fill="0", width=potobj.width
        )
        logging.debug(
            "{logdebug}setpot: setpointpadded =  {setpointpadded}".format(
                logdebug=self.logdebug, setpointpadded=setpointpadded
            )
        )

        err, rval = self.setSubregister(potname, setpointpadded)
        if err:
            logging.error(
                err="{logerr}setPot: unable to confirm setting of subregister:"
                " {potname}".format(logerr=self.logerr, potname=potname)
            )
        ident = potname[3:]
        if ident[0].isdigit():  # numbered pot scheme
            potnumlatch = int(ident) * 2 + 1
            potnumlatchstring = "{num:{fill}{width}x}".format(
                num=potnumlatch, fill="0", width=8
            )
            err1, resp = self.setRegister("POT_CTL", potnumlatchstring)
        else:  # alphabetical DAC scheme
            ident = ident.upper()  # expects single character, e.g. 'A' from 'DACA'
            identnum = ord(ident) - ord("A")  # DACA -> 0
            potnumlatch = int(identnum) * 2 + 1
            potnumlatchstring = "{num:{fill}{width}x}".format(
                num=potnumlatch, fill="0", width=8
            )
            err1, resp = self.setRegister("DAC_CTL", potnumlatchstring)
        if err1:
            # logging.error(self.logerr + "setPot: unable to latch register")

            logging.error(
                err="{logerr}setPot: unable to latch register".format(
                    logerr=self.logerr
                )
            )

        if errflag:
            return err + err1, rval
        return rval

```

6.1.3.73 setPotV()

```
nsCamera.CameraAssembler.CameraAssembler.setPotV (
              self,
              potname,
              voltage,
              tune = False,
              accuracy = 0.01,
              iterations = 20,
              approach = 0.75,
              errflag = False )
```

```
Sets pot to specified voltage. If tune=True, uses monitor to adjust pot to
  correct voltage. Tuning will attempt to tune to closest LSB on pot; if
  'accuracy' > LSB resolution, will only complain if tuning is unable to get
  the voltage within 'accuracy'

Args:
    potname: common name of pot, e.g., VRST found in board.subreg_aliases or
      defined in board.subregisters
    voltage: voltage bound by pot max and min (set in board object)
    tune: if True, iterate with monitor to correct voltage
    accuracy: acceptable error in volts (if None, attempts to find the closest
      possible pot setting and warns if last iteration does not reduce error
      below the resolution of the pot)
    iterations: number of iteration attempts
    approach: approximation parameter (>1 may cause overshoot)
    errflag: if True, return tuple with error string

Returns:
    if errflag:
        tuple: (error string, response string)
    else:
        response string
```

Definition at line 1197 of file CameraAssembler.py.

```
    ):
        """
        Sets pot to specified voltage. If tune=True, uses monitor to adjust pot to
          correct voltage. Tuning will attempt to tune to closest LSB on pot; if
          'accuracy' > LSB resolution, will only complain if tuning is unable to get
          the voltage within 'accuracy'

        Args:
            potname: common name of pot, e.g., VRST found in board.subreg_aliases or
              defined in board.subregisters
            voltage: voltage bound by pot max and min (set in board object)
            tune: if True, iterate with monitor to correct voltage
            accuracy: acceptable error in volts (if None, attempts to find the closest
              possible pot setting and warns if last iteration does not reduce error
              below the resolution of the pot)
            iterations: number of iteration attempts
            approach: approximation parameter (>1 may cause overshoot)
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, response string)
            else:
                response string
        """
        logging.debug(
            self.logdebug
            + "setPotV: potname = "
            + str(potname)
            + "; voltage = "
            + str(voltage)
            + "; tune = "
            + str(tune)
            + "; accuracy = "
            + str(accuracy)
            + "; iterations = "
            + str(iterations)
            + "; approach = "
            + str(approach)
            + "; errflag = "
            + str(errflag)
        )
        potname, potobj, writable = self.resolveSubreg(potname)
        if not potobj:
            err = (
                self.logerr
                + "setPotV: invalid lookup: "
                + potname
                + " , returning zero "
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if not writable:
            err = (
                self.logerr
                + "setPotV: not a writable subregister: "
                + potname
                + "; returning zero"
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if voltage < potobj.minV:
            voltage = potobj.minV
        if voltage > potobj.maxV:
            voltage = potobj.maxV
        setting = (voltage - potobj.minV) / (potobj.maxV - potobj.minV)
        logging.debug(self.logdebug + "setPotV: setting = " + str(setting))
        err, rval = self.setPot(potname, setting, errflag=True)
        time.sleep(0.1)
        # TODO: refactor tuning to separate method
        if tune:
            logging.debug(self.logdebug + "setPotV: beginning tuning")
            if potname not in self.board.monitor_controls.values():
                err = (
                    self.logerr
                    + "setPotV: pot '"
                    + potname
                    + "' does not have a corresponding monitor"
                )
                logging.error(err)
                if errflag:
                    return err, rval
                return rval
            self.setPot(potname, 0.65)
            time.sleep(0.2)
            err1, mon65 = self.getMonV(potname, errflag=True)
            self.setPot(potname, 0.35)
            time.sleep(0.2)
            err2, mon35 = self.getMonV(potname, errflag=True)
            # theoretical voltage range assuming linearity
            potrange = (mon65 - mon35) / 0.3
            stepsize = potrange / (potobj.max_value + 1)
            err += err1 + err2
            if err or potrange < 1:
                err += self.logerr + "setPotV: unable to tune pot " + potname
                if potrange < 1:  # potrange should be on the order of 3.3 or 5 volts
                    err += "; monitor shows insufficient change with pot variation; "
                    "retrying setPotV with tune=False"
                logging.warning(err)
                err, rval = self.setPotV(
                    potname=potname, voltage=voltage, tune=False, errflag=True
                )
                if errflag:
                    return err, rval
                return rval
            potzero = 0.35 - (mon35 / potrange)
            potone = 1.65 - (mon65 / potrange)
            if potzero < 0:
                potzero = 0
            if potone > 1:
                potone = 1

            if accuracy > stepsize:
                mindiff = accuracy
            else:
                mindiff = stepsize
            setting = potzero + (voltage / potone)
            self.setPot(potname, setting)
            lastdiff = 0
            smalladjust = 0
            err3 = ""
            for _ in range(iterations):
                err3i, measured = self.getMonV(potname, errflag=True)
                if err3i:
                    err3 = err3 + err3i + " "
                diff = voltage - measured
                if abs(diff - lastdiff) < stepsize / 2:
                    if (
                        smalladjust > 12
                    ):  # magic number for now; if it doesn't converge after several
                        #   tries, it never will, usually because the setting is pinned
                        #   to 0 or 1 and adjust can't change it
                        logging.warning(
                            self.logwarn
                            + "setPotV: Tuning converged too slowly: pot "
                            + potname
                            + " set to "
                            + str(voltage)
                            + "V, monitor returns "
                            + str(measured)
                            + "V; if this value is incorrect, consider trying "
                            + "tune=False"
                        )
                        logging.debug(self.logdebug + "setPotV: tuning complete")
                        if errflag:
                            return "", rval
                        return rval
                    smalladjust += 1
                if not int(2 * diff / stepsize):
                    # TODO: is this check redundant with the first one?
                    logging.debug(self.logdebug + "setPotV: tuning complete")
                    if errflag:
                        return "", rval
                    return rval
                adjust = approach * (diff / potrange)
                setting += adjust
                if setting > 1:
                    setting = 1
                elif setting < 0:
                    setting = 0
                err1, rval = self.setPot(potname, setting, True)
                lastdiff = diff
                time.sleep(0.2)
            err4, measured = self.getMonV(potname, errflag=True)
            diff = voltage - measured
            # code will try to get to within one stepsize, but will only complain if it
            #   doesn't get within mindiff
            if int(diff / mindiff):
                logging.warning(
                    self.logwarn
                    + "setPotV: pot "
                    + potname
                    + " set to "
                    + str(voltage)
                    + "V, monitor returns "
                    + str(measured)
                    + "V"
                )
            err += err1 + err2 + err3 + err4
        if err:
            logging.error(self.logerr + "setPotV: errors occurred: " + err)
        if errflag:
            return err, rval
        logging.debug(self.logdebug + "setPotV: tuning complete")
        return rval
```

6.1.3.74 setPowerSave()

```
nsCamera.CameraAssembler.CameraAssembler.setPowerSave (
                self,
                status = 1 )
```

Definition at line 368 of file CameraAssembler.py.

```
    def setPowerSave(self, status=1):
        return self.board.setPowerSave(status)
```

6.1.3.75 setPPER()

```
nsCamera.CameraAssembler.CameraAssembler.setPPER (
                self,
                pollperiod = None )
```

Definition at line 371 of file CameraAssembler.py.

```
    def setPPER(self, pollperiod=None):
        return self.board.setPPER(pollperiod)
```

6.1.3.76 setRegister()

```
nsCamera.CameraAssembler.CameraAssembler.setRegister (
              self,
              regname,
              regval )
```

```
Sets named register to given value

Args:
    regname: name of register as given in ICD
    regval: value to assign to register, as integer or hexadecimal string
      with or without '0x'

Returns:
    tuple: (error string, response string)
```

Definition at line 743 of file CameraAssembler.py.

```python
    def setRegister(self, regname, regval):
        """
        Sets named register to given value

        Args:
            regname: name of register as given in ICD
            regval: value to assign to register, as integer or hexadecimal string
              with or without '0x'

        Returns:
            tuple: (error string, response string)
        """
        logging.debug(
            "{logdebug}setRegister: regname = {regname}; regval = {regval}".format(
                logdebug=self.logdebug, regname=regname, regval=regval
            )
        )

        regname = regname.upper()
        if regname not in self.board.registers:
            err = "{logerr}setRegister: Invalid register name: {regname}".format(
                logerr=self.logerr, regname=regname
            )
            logging.error(err)
            return err, "00000000"
        if isinstance(regval, int):
            regval = hex(regval)
        try:
            if regval[0:2] == "0x":
                regval = regval[2:]
        except TypeError:
            err = "{logerr}setRegister: invalid register value parameter".format(
                logerr=self.logerr
            )
            logging.error(err)
            return err, "00000000"
        pkt = Packet(addr=self.board.registers[regname], data=regval)
        err, rval = self.comms.sendCMD(pkt)
        if err:
            logging.error(
                "{logerr}setRegister: {regname}: {err}".format(
                    logerr=self.logerr, regname=regname, err=err
                )
            )
        if len(rval) < 32:
            logging.debug(
                "{logdebug}SetRegister: rval = {rval}".format(
                    logdebug=self.logdebug, rval=rval
                )
            )
        else:
            logging.debug(
                "{logdebug}SetRegister: rval (truncated)= {rval}".format(
                    logdebug=self.logdebug, rval=rval[0:32]
                )
            )
        return err, rval

```

6.1.3.77 setRows()

```
nsCamera.CameraAssembler.CameraAssembler.setRows (
              self,
              minrow = 0,
              maxrow = None,
              padToFull = False )
```

```
Sets bounds on rows returned by board, inclusive (e.g., 0,1023 returns all 1024
  rows). If called without parameters, resets to full image size.

Args:
    minrow: first row to return from board
    maxrow: last row to return from board
    padToFull: if True, generate full size frames, padding collected rows with
      zeroes if necessary
```

Definition at line 1914 of file CameraAssembler.py.

```
    def setRows(self, minrow=0, maxrow=None, padToFull=False):
        """
        Sets bounds on rows returned by board, inclusive (e.g., 0,1023 returns all 1024
          rows). If called without parameters, resets to full image size.

        Args:
            minrow: first row to return from board
            maxrow: last row to return from board
            padToFull: if True, generate full size frames, padding collected rows with
              zeroes if necessary
        """
        logging.debug(
            self.logdebug
            + "setRows: minrow = "
            + str(minrow)
            + "; maxrow = "
            + str(maxrow)
            + "; padToFull = "
            + str(padToFull)
        )
        if maxrow is None:
            maxrow = self.sensor.maxheight - 1
        if (
            not isinstance(minrow, int)
            or minrow < 0
            or minrow > maxrow
            or not isinstance(maxrow, int)
            or maxrow >= self.sensor.maxheight
        ):
            err = (
                self.logerr + "setRows: invalid row arguments submitted. Frame size"
                " remains unchanged. "
            )
            logging.error(err)
            return err

        initrow = hex(minrow)[2:].zfill(8)
        finrow = hex(maxrow)[2:].zfill(8)
        err1, _ = self.setRegister("FPA_ROW_INITIAL", initrow)
        err2, _ = self.setRegister("FPA_ROW_FINAL", finrow)
        self.sensor.firstrow = minrow
        self.sensor.lastrow = maxrow
        self.sensor.height = maxrow - minrow + 1
        self.comms.payloadsize = (
            self.sensor.width
            * self.sensor.height
            * self.sensor.nframes
            * self.sensor.bytesperpixel
        )

        if self.commname == "rs422":
            self.comms.datatimeout = (
                (1.0 * self.sensor.height / self.sensor.maxheight)
                * 5e7
                * self.sensor.nframes
                / self.comms.baud
            )
        self.padToFull = padToFull
        logging.info(
            self.loginfo
            + "Readoff set to "
            + str(self.sensor.height)
            + " rows ("
            + str(minrow)
            + ", "
            + str(maxrow)
            + ")"
        )
        err = err1 + err2
        if err:
            logging.error(
                self.logerr + "setRows may not have functioned properly: " + err
            )
        return err

```

6.1.3.78 setSubregister()

```
nsCamera.CameraAssembler.CameraAssembler.setSubregister (
                self,
                subregname,
                valstring )
```

```
Sets substring of register identified in board attribute 'subregname' to
  valstring if subregister is writable

Args:
    subregname: listed in board.subreg_aliases or defined in board.subregisters
    valstring: integer or binary string with or without '0b'

Returns:
    tuple: (error, packet response string) from setRegister
```

Definition at line 883 of file CameraAssembler.py.

```
    def setSubregister(self, subregname, valstring):
        """
        Sets substring of register identified in board attribute 'subregname' to
          valstring if subregister is writable

        Args:
            subregname: listed in board.subreg_aliases or defined in board.subregisters
            valstring: integer or binary string with or without '0b'

        Returns:
            tuple: (error, packet response string) from setRegister
        """
        logging.debug(
            "{logdebug}setSubegister: subregname = {subregname}; valstring ="
            " {valstring}".format(
                logdebug=self.logdebug, subregname=subregname, valstring=valstring
            )
        )

        subregname, subregobj, writable = self.resolveSubreg(subregname)
        if not subregobj:
            err = "{logerr}getSubregister: invalid lookup: {subregname}".format(
                logerr=self.logerr, subregname=subregname
            )

            logging.error(err)
            return err, "0"
        if not writable:
            err = "{logerr}getSubregister: not a writable subregister: {subregname}"
            "".format(logerr=self.logerr, subregname=subregname)
            logging.error(err)
            return err, "0"
        if isinstance(valstring, int):
            valstring = bin(valstring)[2:]
        try:
            if valstring[0:2] == "0b":
                valstring = valstring[2:]
        except TypeError:
            err = "{logerr}getSubregister: invalid subregister value parameter".format(
                logerr=self.logerr
            )

            logging.error(err)
            return err, "0"
        if len(str(valstring)) > subregobj.width:
            err = "{logerr}getSubregister: ialue string is too long".format(
                logerr=self.logerr
            )

            logging.error(err)
            return err, "0"
        # read current value of register data
        err, resp = self.getRegister(subregobj.register)
        if err:
            logging.error(
                "{logerr}getSubregister: unable to retrieve register setting; setting"
                " of {subregname} likely failed ".format(
                    logerr=self.logerr, subregname=subregname
                )
            )

            return err, "0"
        hex_str = "0x" + resp
        b_reg_value = "{0:0=32b}".format(int(hex_str, 16))  # convert to binary
        # list indexing is reversed from bit string; the last bit of the string is at
        #   index 0 in the list (thus bit 0 is at index 0)
        startindex = 31 - subregobj.start_bit
        valstringpadded = str(valstring).zfill(subregobj.width)
        fullreg = list(b_reg_value)
        fullreg[startindex : startindex + subregobj.width] = valstringpadded
        # convert binary string back to hexadecimal string for writing
        new_reg_value = "".join(fullreg)
        h_reg_value = "{num:{fill}{width}x}".format(
            num=int(new_reg_value, 2), fill="0", width=8
        )
        err, retval = self.setRegister(subregobj.register, h_reg_value)
        # logging.debug(self.logdebug + "retval = " + str(retval))
        if len(retval) < 32:
            logging.debug(
                "{logdebug}setSubregister: retval = {retval}".format(
                    logdebug=self.logdebug, retval=retval
                )
            )
        else:
            logging.debug(
                "{logdebug}setSubregister: retval (truncated) = {retval}".format(
                    logdebug=self.logdebug, retval=retval[0:32]
                )
            )

        return err, retval

```

6.1.3.79 setTiming()

```
nsCamera.CameraAssembler.CameraAssembler.setTiming (
                self,
                side = "AB",
                sequence = None,
                delay = 0 )
```

Definition at line 401 of file CameraAssembler.py.

```
    def setTiming(self, side="AB", sequence=None, delay=0):
        return self.sensor.setTiming(side, sequence, delay)

```

6.1.3.80 setTriggerDelay()

```
nsCamera.CameraAssembler.CameraAssembler.setTriggerDelay (
              self,
              delay = 0 )
```

Definition at line 437 of file CameraAssembler.py.

```
    def setTriggerDelay(self, delay=0):
        return self.sensor.setTriggerDelay(delay)

```

6.1.3.81 setZeroDeadTime()

```
nsCamera.CameraAssembler.CameraAssembler.setZeroDeadTime (
              self,
              flag = True,
              side = None )
```

Definition at line 434 of file CameraAssembler.py.

```
    def setZeroDeadTime(self, flag=True, side=None):
        return self.sensor.setZeroDeadTime(flag, side)

```

6.1.3.82 startCapture()

```
nsCamera.CameraAssembler.CameraAssembler.startCapture (
              self,
              mode )
```

Definition at line 347 of file CameraAssembler.py.

```
    def startCapture(self, mode):
        return self.board.startCapture(mode)

```

6.1.3.83 str2bytes()

```
nsCamera.CameraAssembler.CameraAssembler.str2bytes (
              self,
              astring )
```

Definition at line 483 of file CameraAssembler.py.

```
    def str2bytes(self, astring):
        return str2bytes(astring)

```

6.1.3.84 str2nparray()

```
nsCamera.CameraAssembler.CameraAssembler.str2nparray (
              self,
              valstring )
```

Definition at line 489 of file CameraAssembler.py.

```
    def str2nparray(self, valstring):
        return str2nparray(valstring)

```

6.1.3.85 submitMessages()

```
nsCamera.CameraAssembler.CameraAssembler.submitMessages (
              self,
              messages,
              errorstring = "Error" )
```

```
Serially set multiple register / subregister values

Args:
    messages: list of tuples (register name, integer or hexadecimal string with
      or without '0x') and/or (subregister name, integer or binary string with
      or without '0b')
    errorstring: error message to print in case of failure

Returns:
    tuple (accumulated error string, response string of final message)
```

Definition at line 975 of file CameraAssembler.py.

```
    def submitMessages(self, messages, errorstring="Error"):
        """
        Serially set multiple register / subregister values

        Args:
            messages: list of tuples (register name, integer or hexadecimal string with
              or without '0x') and/or (subregister name, integer or binary string with
              or without '0b')
            errorstring: error message to print in case of failure

        Returns:
            tuple (accumulated error string, response string of final message)
        """
        logging.debug(
            "{logdebug}submitMessages: messages = {messages}; errorstring ="
            " {errorstring}".format(
                logdebug=self.logdebug, messages=messages, errorstring=errorstring
            )
        )

        errs = ""
        err = ""
        rval = ""
        for m in messages:
            if m[0].upper() in self.board.registers:
                err, rval = self.setRegister(m[0].upper(), m[1])
            elif m[0].upper() in self.board.subreglist:
                err, rval = self.setSubregister(m[0].upper(), m[1])
            else:
                err = "{logerr}submitMessages: Invalid register/subregister:"
                " {errorstring}:{m0}; ".format(
                    logerr=self.logerr, errorstring=errorstring, m0=m[0]
                )

                logging.error(err)
            errs = errs + err
        return err, rval

```

6.1.3.86 waitForSRAM()

```
nsCamera.CameraAssembler.CameraAssembler.waitForSRAM (
              self,
              timeout = None )
```

Definition at line 353 of file CameraAssembler.py.

```
    def waitForSRAM(self, timeout=None):
        return self.board.waitForSRAM(timeout)

```

6.1.3.87 writeSerial()

```
nsCamera.CameraAssembler.CameraAssembler.writeSerial (
              self,
              cmd,
              timeout = None )
```

Definition at line 462 of file CameraAssembler.py.

```
    def writeSerial(self, cmd, timeout=None):
        return self.comms.writeSerial(cmd, timeout)

```

6.1.4 Member Data Documentation

6.1.4.1 abort

```
nsCamera.CameraAssembler.CameraAssembler.abort
```

Definition at line 252 of file CameraAssembler.py.

6.1.4.2 arch

```
nsCamera.CameraAssembler.CameraAssembler.arch
```

Definition at line 228 of file CameraAssembler.py.

6.1.4.3 armed

```
nsCamera.CameraAssembler.CameraAssembler.armed
```

Definition at line 244 of file CameraAssembler.py.

6.1.4.4 board

```
nsCamera.CameraAssembler.CameraAssembler.board
```

Definition at line 549 of file CameraAssembler.py.

6.1.4.5 boardname

```
nsCamera.CameraAssembler.CameraAssembler.boardname
```

Definition at line 204 of file CameraAssembler.py.

6.1.4.6 commname

```
nsCamera.CameraAssembler.CameraAssembler.commname
```

Definition at line 211 of file CameraAssembler.py.

6.1.4.7 comms

```
nsCamera.CameraAssembler.CameraAssembler.comms
```

Definition at line 543 of file CameraAssembler.py.

6.1.4.8 currtime

```
nsCamera.CameraAssembler.CameraAssembler.currtime
```

Definition at line 195 of file CameraAssembler.py.

6.1.4.9 cycle

```
nsCamera.CameraAssembler.CameraAssembler.cycle
```

Definition at line 203 of file CameraAssembler.py.

6.1.4.10 FPGAboardtype

```
nsCamera.CameraAssembler.CameraAssembler.FPGAboardtype
```

Definition at line 234 of file CameraAssembler.py.

6.1.4.11 FPGAinterfaces

```
nsCamera.CameraAssembler.CameraAssembler.FPGAinterfaces
```

Definition at line 237 of file CameraAssembler.py.

6.1.4.12 FPGAinvalid

```
nsCamera.CameraAssembler.CameraAssembler.FPGAinvalid
```

Definition at line 240 of file CameraAssembler.py.

6.1.4.13 FPGANum

```
nsCamera.CameraAssembler.CameraAssembler.FPGANum
```

Definition at line 231 of file CameraAssembler.py.

6.1.4.14 FPGArad

```
nsCamera.CameraAssembler.CameraAssembler.FPGArad
```

Definition at line 235 of file CameraAssembler.py.

6.1.4.15 FPGAsensor

```
nsCamera.CameraAssembler.CameraAssembler.FPGAsensor
```

Definition at line 236 of file CameraAssembler.py.

6.1.4.16 FPGAVersion

```
nsCamera.CameraAssembler.CameraAssembler.FPGAVersion
```

Definition at line 230 of file CameraAssembler.py.

6.1.4.17 inittime

```
nsCamera.CameraAssembler.CameraAssembler.inittime
```

Definition at line 250 of file CameraAssembler.py.

6.1.4.18 iplist

```
nsCamera.CameraAssembler.CameraAssembler.iplist
```

Definition at line 242 of file CameraAssembler.py.

6.1.4.19 logcrit

```
nsCamera.CameraAssembler.CameraAssembler.logcrit
```

Definition at line 272 of file CameraAssembler.py.

6.1.4.20 logcritbase

`nsCamera.CameraAssembler.CameraAssembler.logcritbase`

Definition at line 266 of file CameraAssembler.py.

6.1.4.21 logdebug

`nsCamera.CameraAssembler.CameraAssembler.logdebug`

Definition at line 276 of file CameraAssembler.py.

6.1.4.22 logdebugbase

`nsCamera.CameraAssembler.CameraAssembler.logdebugbase`

Definition at line 270 of file CameraAssembler.py.

6.1.4.23 logerr

`nsCamera.CameraAssembler.CameraAssembler.logerr`

Definition at line 273 of file CameraAssembler.py.

6.1.4.24 logerrbase

`nsCamera.CameraAssembler.CameraAssembler.logerrbase`

Definition at line 267 of file CameraAssembler.py.

6.1.4.25 loginfo

`nsCamera.CameraAssembler.CameraAssembler.loginfo`

Definition at line 275 of file CameraAssembler.py.

6.1.4.26 loginfobase

`nsCamera.CameraAssembler.CameraAssembler.loginfobase`

Definition at line 269 of file CameraAssembler.py.

6.1.4.27 logtag

```
nsCamera.CameraAssembler.CameraAssembler.logtag
```

Definition at line 264 of file CameraAssembler.py.

6.1.4.28 logwarn

```
nsCamera.CameraAssembler.CameraAssembler.logwarn
```

Definition at line 274 of file CameraAssembler.py.

6.1.4.29 logwarnbase

```
nsCamera.CameraAssembler.CameraAssembler.logwarnbase
```

Definition at line 268 of file CameraAssembler.py.

6.1.4.30 oldtime

```
nsCamera.CameraAssembler.CameraAssembler.oldtime
```

Definition at line 196 of file CameraAssembler.py.

6.1.4.31 packageroot

```
nsCamera.CameraAssembler.CameraAssembler.packageroot
```

Definition at line 243 of file CameraAssembler.py.

6.1.4.32 padToFull

```
nsCamera.CameraAssembler.CameraAssembler.padToFull
```

Definition at line 251 of file CameraAssembler.py.

6.1.4.33 parsedtime

```
nsCamera.CameraAssembler.CameraAssembler.parsedtime
```

Definition at line 201 of file CameraAssembler.py.

6.1.4.34 payloaderror

`nsCamera.CameraAssembler.CameraAssembler.payloaderror`

Definition at line 320 of file CameraAssembler.py.

6.1.4.35 platform

`nsCamera.CameraAssembler.CameraAssembler.platform`

Definition at line 227 of file CameraAssembler.py.

6.1.4.36 port

`nsCamera.CameraAssembler.CameraAssembler.port`

Definition at line 224 of file CameraAssembler.py.

6.1.4.37 PY3

`nsCamera.CameraAssembler.CameraAssembler.PY3`

Definition at line 226 of file CameraAssembler.py.

6.1.4.38 pyth1

`nsCamera.CameraAssembler.CameraAssembler.pyth1`

Definition at line 225 of file CameraAssembler.py.

6.1.4.39 pyth2

`nsCamera.CameraAssembler.CameraAssembler.pyth2`

Definition at line 225 of file CameraAssembler.py.

6.1.4.40 python

`nsCamera.CameraAssembler.CameraAssembler.python`

Definition at line 225 of file CameraAssembler.py.

6.1.4.41 read

`nsCamera.CameraAssembler.CameraAssembler.read`

Definition at line 199 of file CameraAssembler.py.

6.1.4.42 savetime

`nsCamera.CameraAssembler.CameraAssembler.savetime`

Definition at line 202 of file CameraAssembler.py.

6.1.4.43 sensmanual

`nsCamera.CameraAssembler.CameraAssembler.sensmanual`

Definition at line 249 of file CameraAssembler.py.

6.1.4.44 sensor

`nsCamera.CameraAssembler.CameraAssembler.sensor`

Definition at line 524 of file CameraAssembler.py.

6.1.4.45 sensorname

`nsCamera.CameraAssembler.CameraAssembler.sensorname`

For regular version.

Definition at line 216 of file CameraAssembler.py.

6.1.4.46 senstiming

`nsCamera.CameraAssembler.CameraAssembler.senstiming`

Definition at line 248 of file CameraAssembler.py.

6.1.4.47 timeout

`nsCamera.CameraAssembler.CameraAssembler.timeout`

Definition at line 205 of file CameraAssembler.py.

6.1.4.48 trigtime

`nsCamera.CameraAssembler.CameraAssembler.trigtime`

Definition at line 197 of file CameraAssembler.py.

6.1.4.49 unstringed

`nsCamera.CameraAssembler.CameraAssembler.unstringed`

Definition at line 200 of file CameraAssembler.py.

6.1.4.50 verblevel

`nsCamera.CameraAssembler.CameraAssembler.verblevel`

Definition at line 278 of file CameraAssembler.py.

6.1.4.51 verbmap

`nsCamera.CameraAssembler.CameraAssembler.verbmap`

Definition at line 254 of file CameraAssembler.py.

6.1.4.52 verbose

`nsCamera.CameraAssembler.CameraAssembler.verbose`

Definition at line 223 of file CameraAssembler.py.

6.1.4.53 version

`nsCamera.CameraAssembler.CameraAssembler.version`

Definition at line 194 of file CameraAssembler.py.

6.1.4.54 waited

`nsCamera.CameraAssembler.CameraAssembler.waited`

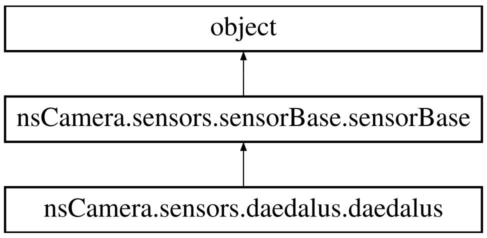
Definition at line 198 of file CameraAssembler.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/CameraAssembler.py

6.2 nsCamera.sensors.daedalus.daedalus Class Reference

Inheritance diagram for nsCamera.sensors.daedalus.daedalus:


object
nsCamera.sensors.sensorBase.sensorBase
nsCamera.sensors.daedalus.daedalus

Public Member Functions

- __init__ (self, ca)
- sensorSpecific (self)
- setInterlacing (self, ifactor=None, side=None)
- setHighFullWell (self, flag)
- setZeroDeadTime (self, flag=True, side=None)
- selectOscillator (self, osc=None)
- setTriggerDelay (self, delay=0)
- setPhiDelay (self, side=None, delay=0)
- setExtClk (self, dilation=None, frequency=None)
- setManualShutters (self, timing=None)
- setManualTiming (self, timing=None)
- getManualTiming (self)
- getSensTemp (self, scale=None, offset=None, slope=None, dec=1)
- parseReadoff (self, frames, columns)
- reportStatusSensor (self, statusbits, statusbits2)

Public Member Functions inherited from nsCamera.sensors.sensorBase.sensorBase

- init_board_specific (self)
- checkSensorVoltStat (self)
- setTiming (self, side="AB", sequence=None, delay=0)
- setArbTiming (self, side="AB", sequence=None)
- getTiming (self, side, actual)
- getSensorStatus (self)

Public Attributes

- ca
- sens_registers
- sens_subregisters
- interlacing
- columns
- HFW
- ZDT

Public Attributes inherited from nsCamera.sensors.sensorBase.sensorBase

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- fpganumID
- sensfam

Static Public Attributes

- str specwarn = ""
- int minframe = 0
- int maxframe = 2
- int maxwidth = 512
- int maxheight = 1024
- int bytesperpixel = 2
- int fpganumID = 2
- str detect = "DAEDALUS_DET"
- str sensfam = "Daedalus"
- str loglabel = "[Daedalus] "
- bool ZDT = False
- bool HFW = False
- int firstframe = 0
- int lastframe = 2
- int nframes = 3
- int width = 512
- int height = 1024
- int firstrow = 0
- int lastrow = 1023
- list interlacing = [0, 0]
- int columns = 1
- bool padToFull = True
- float toffset = -165.76
- float tslope = 81.36

6.2.1 Detailed Description

Definition at line 30 of file daedalus.py.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 __init__()

```
nsCamera.sensors.daedalus.daedalus.__init__ (
                self,
                ca )
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 56 of file daedalus.py.

```
00056     def __init__(self, ca):
00057         self.ca = ca
00058         super(daedalus, self).__init__(ca)
00059 
00060         self.sens_registers = OrderedDict(
00061             {
00062                 "HST_READBACK_A_LO": "018",
00063                 "HST_READBACK_A_HI": "019",
00064                 "HST_READBACK_B_LO": "01A",
00065                 "HST_READBACK_B_HI": "01B",
00066                 "HSTALLWEN_WAIT_TIME": "03F",
00067                 "VRESET_HIGH_VALUE": "04A",
00068                 "FRAME_ORDER_SEL": "04B",
00069                 "EXT_PHI_CLK_SH0_ON": "050",
00070                 "EXT_PHI_CLK_SH0_OFF": "051",
00071                 "EXT_PHI_CLK_SH1_ON": "052",
00072                 "EXT_PHI_CLK_SH1_OFF": "053",
00073                 "EXT_PHI_CLK_SH2_ON": "054",
00074                 "HST_TRIGGER_DELAY_DATA_LO": "120",
00075                 "HST_TRIGGER_DELAY_DATA_HI": "121",
00076                 "HST_PHI_DELAY_DATA": "122",
00077                 "HST_EXT_CLK_HALF_PER": "129",
00078                 "HST_COUNT_TRIG": "130",
00079                 "HST_DELAY_EN": "131",
00080                 "RSL_HFW_MODE_EN": "133",
00081                 "RSL_ZDT_MODE_B_EN": "135",
00082                 "RSL_ZDT_MODE_A_EN": "136",
00083                 "BGTRIMA": "137",
00084                 "BGTRIMB": "138",
00085                 "COLUMN_TEST_EN": "139",
00086                 "RSL_CONFIG_DATA_B0": "140",
00087                 "RSL_CONFIG_DATA_B1": "141",
00088                 "RSL_CONFIG_DATA_B2": "142",
00089                 "RSL_CONFIG_DATA_B3": "143",
00090                 "RSL_CONFIG_DATA_B4": "144",
00091                 "RSL_CONFIG_DATA_B5": "145",
00092                 "RSL_CONFIG_DATA_B6": "146",
00093                 "RSL_CONFIG_DATA_B7": "147",
00094                 "RSL_CONFIG_DATA_B8": "148",
00095                 "RSL_CONFIG_DATA_B9": "149",
00096                 "RSL_CONFIG_DATA_B10": "14A",
00097                 "RSL_CONFIG_DATA_B11": "14B",
00098                 "RSL_CONFIG_DATA_B12": "14C",
00099                 "RSL_CONFIG_DATA_B13": "14D",
00100                 "RSL_CONFIG_DATA_B14": "14E",
00101                 "RSL_CONFIG_DATA_B15": "14F",
00102                 "RSL_CONFIG_DATA_B16": "150",
00103                 "RSL_CONFIG_DATA_B17": "151",
00104                 "RSL_CONFIG_DATA_B18": "152",
00105                 "RSL_CONFIG_DATA_B19": "153",
00106                 "RSL_CONFIG_DATA_B20": "154",
00107                 "RSL_CONFIG_DATA_B21": "155",
00108                 "RSL_CONFIG_DATA_B22": "156",
00109                 "RSL_CONFIG_DATA_B23": "157",
00110                 "RSL_CONFIG_DATA_B24": "158",
00111                 "RSL_CONFIG_DATA_B25": "159",
00112                 "RSL_CONFIG_DATA_B26": "15A",
00113                 "RSL_CONFIG_DATA_B27": "15B",
00114                 "RSL_CONFIG_DATA_B28": "15C",
00115                 "RSL_CONFIG_DATA_B29": "15D",
00116                 "RSL_CONFIG_DATA_B30": "15E",
00117                 "RSL_CONFIG_DATA_B31": "15F",
00118                 "RSL_CONFIG_DATA_A0": "160",
                "RSL_CONFIG_DATA_A1": "161",
                "RSL_CONFIG_DATA_A2": "162",
                "RSL_CONFIG_DATA_A3": "163",
                "RSL_CONFIG_DATA_A4": "164",
                "RSL_CONFIG_DATA_A5": "165",
                "RSL_CONFIG_DATA_A6": "166",
                "RSL_CONFIG_DATA_A7": "167",
                "RSL_CONFIG_DATA_A8": "168",
                "RSL_CONFIG_DATA_A9": "169",
                "RSL_CONFIG_DATA_A10": "16A",
                "RSL_CONFIG_DATA_A11": "16B",
                "RSL_CONFIG_DATA_A12": "16C",
                "RSL_CONFIG_DATA_A13": "16D",
                "RSL_CONFIG_DATA_A14": "16E",
                "RSL_CONFIG_DATA_A15": "16F",
                "RSL_CONFIG_DATA_A16": "170",
                "RSL_CONFIG_DATA_A17": "171",
                "RSL_CONFIG_DATA_A18": "172",
                "RSL_CONFIG_DATA_A19": "173",
                "RSL_CONFIG_DATA_A20": "174",
                "RSL_CONFIG_DATA_A21": "175",
                "RSL_CONFIG_DATA_A22": "176",
                "RSL_CONFIG_DATA_A23": "177",
                "RSL_CONFIG_DATA_A24": "178",
                "RSL_CONFIG_DATA_A25": "179",
                "RSL_CONFIG_DATA_A26": "17A",
                "RSL_CONFIG_DATA_A27": "17B",
                "RSL_CONFIG_DATA_A28": "17C",
                "RSL_CONFIG_DATA_A29": "17D",
                "RSL_CONFIG_DATA_A30": "17E",
                "RSL_CONFIG_DATA_A31": "17F",
            }
        )

        self.sens_subregisters = [

            ("HST_MODE", "HS_TIMING_CTL", 0, 1, True),
            ("SLOWREADOFF_0", "CTRL_REG", 4, 1, True),
            ("SLOWREADOFF_1", "CTRL_REG", 5, 1, True),
            ("MANSHUT_MODE", "CTRL_REG", 8, 1, True),
            ("INTERLACING_EN", "CTRL_REG", 9, 1, True),
            ("HFW", "RSL_HFW_MODE_EN", 0, 1, True),
            ("ZDT_A", "RSL_ZDT_MODE_A_EN", 0, 1, True),
            ("ZDT_B", "RSL_ZDT_MODE_B_EN", 0, 1, True),
            ("HST_DEL_EN", "HST_DELAY_EN", 0, 1, True),
            ("PHI_DELAY_A", "HST_PHI_DELAY_DATA", 9, 10, True),
            ("PHI_DELAY_B", "HST_PHI_DELAY_DATA", 29, 10, True),
            # Assume that daedalus is not to be used with v1 board
            ("VRESET_HIGH", "VRESET_HIGH_VALUE", 15, 16, True),

            ("STAT_SH0RISEUR", "STAT_REG", 3, 1, False),
            ("STAT_SH0FALLUR", "STAT_REG", 4, 1, False),
            ("STAT_RSLNALLWENA", "STAT_REG", 12, 1, False),
            ("STAT_RSLNALLWENB", "STAT_REG", 15, 1, False),
            # ("STAT_CONFIGHSTDONE", "STAT_REG", 16, 1, False),
        ]

```

6.2.3 Member Function Documentation

6.2.3.1 getManualTiming()

```
nsCamera.sensors.daedalus.daedalus.getManualTiming (
              self )
```

```
Read off manual shutter timing settings
Returns:
    list of manual timing intervals
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 659 of file daedalus.py.

```
    def getManualTiming(self):
        """
        Read off manual shutter timing settings
        Returns:
            list of manual timing intervals
        """
        timing = []
        for reg in [
            "EXT_PHI_CLK_SH0_ON",
            "EXT_PHI_CLK_SH0_OFF",
            "EXT_PHI_CLK_SH1_ON",
            "EXT_PHI_CLK_SH1_OFF",
            "EXT_PHI_CLK_SH2_ON",
        ]:
            _, reghex = self.ca.getRegister(reg)
            timing.append(25 * int(reghex, 16))
        return timing

```

6.2.3.2 getSensTemp()

```
nsCamera.sensors.daedalus.daedalus.getSensTemp (
                self,
                scale = None,
                offset = None,
                slope = None,
                dec = 1 )
```

```
Read temperature sensor located on the Daedalus sensor
Args:
    scale: temperature scale to report (defaults to C, options are F and K)
    offset: offset of linear fit of sensor response (defaults to self.toffset)
    slope: slope of linear fit of sensor response (defaults to self.tslope)
    dec: round to 'dec' digits after the decimal point

Returns:
    temperature as float on given scale, rounded to .1 degree
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 677 of file daedalus.py.

```
    def getSensTemp(self, scale=None, offset=None, slope=None, dec=1):
        """
        Read temperature sensor located on the Daedalus sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)
            offset: offset of linear fit of sensor response (defaults to self.toffset)
            slope: slope of linear fit of sensor response (defaults to self.tslope)
            dec: round to 'dec' digits after the decimal point

        Returns:
            temperature as float on given scale, rounded to .1 degree
        """
        err, rval = self.ca.getMonV("MON_TSENSE_OUT", errflag=True)
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0
        if offset is None:
            offset = self.toffset
        if slope is None:
            slope = self.tslope
        ctemp = offset + slope * rval
        if scale == "K":
            temp = round(ctemp + 273.15, dec)
        elif scale == "F":
            temp = round(1.8 * ctemp + 32, dec)
        else:
            temp = round(ctemp, dec)
        return temp

```

6.2.3.3 parseReadoff()

```
nsCamera.sensors.daedalus.daedalus.parseReadoff (
                self,
                frames,
                columns )
```

```
Parses frames from board into images
Args:
    frames: list of data arrays (frames) returned from board
    columns: 1 (full width image) or 2 (hemispheres generate distinct images)
Returns:
    list of data arrays (frames) reordered and deinterlaced
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 710 of file daedalus.py.

```
    def parseReadoff(self, frames, columns):
        """
        Parses frames from board into images
        Args:
            frames: list of data arrays (frames) returned from board
            columns: 1 (full width image) or 2 (hemispheres generate distinct images)
        Returns:
            list of data arrays (frames) reordered and deinterlaced
        """
        logging.debug(self.logdebug + "parseReadoff")
        w = self.width
        if hasattr(self, "ca"):  # TODO: this may no longer be necessary
            padIt = self.ca.padToFull
        else:
            padIt = self.padToFull
        if padIt:
            rows = self.maxheight
        else:
            rows = self.lastrow - self.firstrow + 1
        parsed = []
        for frame in frames:
            current = np.zeros((rows, w), dtype=np.uint16)
            mapped = np.zeros((rows, w), dtype=np.uint16)
            frame = frame.reshape(rows, w)

            for entry in range(int(w / 2)):
                col = 32 * (entry % 8) + entry // 8  # lookup from daedlookup.xls
                for row in range(rows):
                    current[row][col] = frame[row][2 * entry]
                    current[row][col + 256] = frame[row][2 * entry + 1]

            for row in range(rows):
                mapped[row][0:32] = current[row][320:352]
                mapped[row][32:64] = current[row][352:384]
                mapped[row][64:96] = current[row][192:224]
                mapped[row][96:128] = current[row][160:192]
                mapped[row][128:160] = current[row][256:288]
                mapped[row][160:192] = current[row][288:320]
                mapped[row][192:224] = current[row][416:448]
                mapped[row][224:256] = current[row][32:64]
                mapped[row][256:288] = current[row][128:160]
                mapped[row][288:320] = current[row][224:256]
                mapped[row][320:352] = current[row][384:416]
                mapped[row][352:384] = current[row][448:480]
                mapped[row][384:416] = current[row][480:512]
                mapped[row][416:448] = current[row][0:32]
                mapped[row][448:480] = current[row][64:96]
                mapped[row][480:512] = current[row][96:128]
            parsed.append(mapped)

        images = self.ca.partition(parsed, columns)
        flatimages = [flattenlist(x) for x in images]
        return flatimages

```

6.2.3.4 reportStatusSensor()

```
nsCamera.sensors.daedalus.daedalus.reportStatusSensor (
                self,
                statusbits,
                statusbits2 )
```

```
Print status messages from sensor-specific bits of status register or object
  status flags

Args:
    statusbits: result of checkStatus()
    statusbits2: result of checkStatus2()
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 764 of file daedalus.py.

```
    def reportStatusSensor(self, statusbits, statusbits2):
        """
        Print status messages from sensor-specific bits of status register or object
          status flags

        Args:
            statusbits: result of checkStatus()
            statusbits2: result of checkStatus2()
        """
        if int(statusbits[3]):
            print(self.loginfo + "SH0_rise_B_edge detected")
        if int(statusbits[4]):
            print(self.loginfo + "SH0_fall_B_edge detected")
        if int(statusbits[12]):
            print(self.loginfo + "RSLNALLWENB detected")
        if int(statusbits[15]):
            print(self.loginfo + "RSLNALLWENA detected")
        if self.HFW:
            print(self.loginfo + "High Full Well mode active")
        # TODO: handle two hemispheres for ZDT
        elif self.ZDT:
            print(self.loginfo + "Zero Dead Time mode active")
        elif self.interlacing != [0, 0]:
            print(
                "{loginfo}Interlacing active: {interlacing}".format(
                    loginfo=self.loginfo, interlacing=str(self.interlacing)
                )
            )
        if self.ca.sensmanual == []:
            print(
                "{loginfo}High-speed timing: A:{Atiming}, B:{Btiming}".format(
                    loginfo=self.loginfo,
                    Atiming=self.getTiming(side="A", actual=True),
                    Btiming=self.getTiming(side="B", actual=True),
                )
            )
        else:
            print(
                "{loginfo}Manual timing set to {timing}".format(
                    loginfo=self.loginfo, timing=self.getManualTiming()
                )
            )


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.2.3.5 selectOscillator()

```
nsCamera.sensors.daedalus.daedalus.selectOscillator (
                self,
                osc = None )
```

```
Selects oscillator to control sensor timing
Args:
    osc: 500|100|'ring'|external', defaults to 500 MHz

Returns:
    error message as string
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 444 of file daedalus.py.

```
    def selectOscillator(self, osc=None):
        """
        Selects oscillator to control sensor timing
        Args:
            osc: 500|100|'ring'|external', defaults to 500 MHz

        Returns:
            error message as string
        """
        logging.info(self.loginfo + "selectOscillator; osc = " + str(osc))
        if osc is None:
            osc = 500
        osc = str(osc)
        if osc[:3] == "500":
            payload = "00"
        elif osc[:3] == "100":
            payload = "01"
        elif osc.upper()[:3] == "RIN":
            payload = "10"
        elif osc.upper()[:3] in ["EXT"]:
            payload = "11"
        else:
            err = (
                self.logerr + "selectOscillator: invalid parameter supplied. "
                "Oscillator selection is unchanged."
            )
            logging.error(err)
            return err
        self.ca.setSubregister("OSC_SELECT", payload)
```

6.2.3.6 sensorSpecific()

```
nsCamera.sensors.daedalus.daedalus.sensorSpecific (
                self )
```

```
Returns:
    list of tuples, (Sensor-specific register, default setting)
```

Definition at line 180 of file daedalus.py.

```
00180     def sensorSpecific(self):
00181         """
00182         Returns:
00183             list of tuples, (Sensor-specific register, default setting)
00184         """
00185         return [
00186             ("FPA_FRAME_INITIAL", "00000000"),
00187             ("FPA_FRAME_FINAL", "00000002"),
00188             ("FPA_ROW_INITIAL", "00000000"),
00189             ("FPA_ROW_FINAL", "000003FF"),
00190             ("HS_TIMING_DATA_ALO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
00191             ("HS_TIMING_DATA_AHI", "00000000"),
00192             ("HS_TIMING_DATA_BLO", "00006666"),
00193             ("HS_TIMING_DATA_BHI", "00000000"),
00194             ("FRAME_ORDER_SEL", "00000000"),
00195             ("RSL_HFW_MODE_EN", "00000000"),
00196             ("RSL_ZDT_MODE_B_EN", "00000000"),
00197             ("RSL_ZDT_MODE_A_EN", "00000000"),
00198             ("RSL_CONFIG_DATA_B0", "00000000"),
00199             ("RSL_CONFIG_DATA_B1", "00000000"),
00200             ("RSL_CONFIG_DATA_B2", "00000000"),
00201             ("RSL_CONFIG_DATA_B3", "00000000"),
00202             ("RSL_CONFIG_DATA_B4", "00000000"),
00203             ("RSL_CONFIG_DATA_B5", "00000000"),
00204             ("RSL_CONFIG_DATA_B6", "00000000"),
00205             ("RSL_CONFIG_DATA_B7", "00000000"),
00206             ("RSL_CONFIG_DATA_B8", "00000000"),
00207             ("RSL_CONFIG_DATA_B9", "00000000"),
00208             ("RSL_CONFIG_DATA_B10", "00000000"),
00209             ("RSL_CONFIG_DATA_B11", "00000000"),
00210             ("RSL_CONFIG_DATA_B12", "00000000"),
00211             ("RSL_CONFIG_DATA_B13", "00000000"),
00212             ("RSL_CONFIG_DATA_B14", "00000000"),
00213             ("RSL_CONFIG_DATA_B15", "00000000"),
00214             ("RSL_CONFIG_DATA_B16", "00000000"),
00215             ("RSL_CONFIG_DATA_B17", "00000000"),
00216             ("RSL_CONFIG_DATA_B18", "00000000"),
00217             ("RSL_CONFIG_DATA_B19", "00000000"),
00218             ("RSL_CONFIG_DATA_B20", "00000000"),
00219             ("RSL_CONFIG_DATA_B21", "00000000"),
00220             ("RSL_CONFIG_DATA_B22", "00000000"),
00221             ("RSL_CONFIG_DATA_B23", "00000000"),
00222             ("RSL_CONFIG_DATA_B24", "00000000"),
00223             ("RSL_CONFIG_DATA_B25", "00000000"),
00224             ("RSL_CONFIG_DATA_B26", "00000000"),
00225             ("RSL_CONFIG_DATA_B27", "00000000"),
00226             ("RSL_CONFIG_DATA_B28", "00000000"),
00227             ("RSL_CONFIG_DATA_B29", "00000000"),
00228             ("RSL_CONFIG_DATA_B30", "00000000"),
00229             ("RSL_CONFIG_DATA_B31", "00000000"),
00230             ("RSL_CONFIG_DATA_A0", "00000000"),
00231             ("RSL_CONFIG_DATA_A1", "00000000"),
00232             ("RSL_CONFIG_DATA_A2", "00000000"),
00233             ("RSL_CONFIG_DATA_A3", "00000000"),
00234             ("RSL_CONFIG_DATA_A4", "00000000"),
00235             ("RSL_CONFIG_DATA_A5", "00000000"),
00236             ("RSL_CONFIG_DATA_A6", "00000000"),
00237             ("RSL_CONFIG_DATA_A7", "00000000"),
00238             ("RSL_CONFIG_DATA_A8", "00000000"),
00239             ("RSL_CONFIG_DATA_A9", "00000000"),
00240             ("RSL_CONFIG_DATA_A10", "00000000"),
00241             ("RSL_CONFIG_DATA_A11", "00000000"),
00242             ("RSL_CONFIG_DATA_A12", "00000000"),
00243             ("RSL_CONFIG_DATA_A13", "00000000"),
00244             ("RSL_CONFIG_DATA_A14", "00000000"),
00245             ("RSL_CONFIG_DATA_A15", "00000000"),
            ("RSL_CONFIG_DATA_A16", "00000000"),
            ("RSL_CONFIG_DATA_A17", "00000000"),
            ("RSL_CONFIG_DATA_A18", "00000000"),
            ("RSL_CONFIG_DATA_A19", "00000000"),
            ("RSL_CONFIG_DATA_A20", "00000000"),
            ("RSL_CONFIG_DATA_A21", "00000000"),
            ("RSL_CONFIG_DATA_A22", "00000000"),
            ("RSL_CONFIG_DATA_A23", "00000000"),
            ("RSL_CONFIG_DATA_A24", "00000000"),
            ("RSL_CONFIG_DATA_A25", "00000000"),
            ("RSL_CONFIG_DATA_A26", "00000000"),
            ("RSL_CONFIG_DATA_A27", "00000000"),
            ("RSL_CONFIG_DATA_A28", "00000000"),
            ("RSL_CONFIG_DATA_A29", "00000000"),
            ("RSL_CONFIG_DATA_A30", "00000000"),
            ("RSL_CONFIG_DATA_A31", "00000000"),
            ("HST_TRIGGER_DELAY_DATA_LO", "00000000"),
            ("HST_TRIGGER_DELAY_DATA_HI", "00000000"),
            ("HST_PHI_DELAY_DATA", "00000000"),
            ("SLOWREADOFF_0", "0"),
            ("SLOWREADOFF_1", "0"),
        ]

```

6.2.3.7 setExtClk()

```
nsCamera.sensors.daedalus.daedalus.setExtClk (
                self,
                dilation = None,
                frequency = None )
```

```
Override the standard board clock with the external clock.
Args:
    dilation: ratio of base frequency (500 MHz) to desired external clock
      frequency. Default is 25. Overridden if frequency parameter is provided
    frequency: Desired frequency for phi clock.
Returns:
    error message as string
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 556 of file daedalus.py.

```
    def setExtClk(self, dilation=None, frequency=None):
        """
        Override the standard board clock with the external clock.
        Args:
            dilation: ratio of base frequency (500 MHz) to desired external clock
              frequency. Default is 25. Overridden if frequency parameter is provided
            frequency: Desired frequency for phi clock.
        Returns:
            error message as string
        """
        logging.debug(
            self.logdebug
            + "setExtClk; dilation = "
            + str(dilation)
            + "; frequency = "
            + str(frequency)
        )
        if not (isinstance(frequency, int) or isinstance(frequency, float)):
            err = (
                self.logerr
                + "invalid external clock frequency submitted. Clock is not "
                + "operating"
            )
            logging.error(err)
            return err
        self.ca.selectOscillator("external")
        if not dilation:
            dilation = 25
        if not frequency:
            frequency = 5e7 / float(dilation)
        count = 2e7 / float(frequency) - 1  # base phi clock is 20 MHz?
        if count < 0:
            count = 0
            warn = (
                self.logwarn
                + "external clock frequency exceeding maximum. Frequency set to "
                + "maximum (20 MHz)"
            )
            logging.warning(warn)
        if count > 0xFFFFFFFF:
            count = 0xFFFFFFFF
        counthex = hex(int(count))[2:].zfill(8)
        self.ca.setRegister("HST_EXT_CLK_HALF_PER", counthex)

```

6.2.3.8 setHighFullWell()

nsCamera.sensors.daedalus.daedalus.setHighFullWell (
self,
flag)

```
Activates High Full Well mode. All frames are acquired simultaneously. Zero Dead
  Time mode and interlacing will be automatically deactivated and column number
  will be reset to 0. NOTE: after deactivating HFW, the board remains in
  uninterlaced mode (interlacing = 0)

Args:
    flag: True to activate HFW mode, False to deactivate

Returns:
    Error message
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 354 of file daedalus.py.

```
    def setHighFullWell(self, flag):
        """
        Activates High Full Well mode. All frames are acquired simultaneously. Zero Dead
          Time mode and interlacing will be automatically deactivated and column number
          will be reset to 0. NOTE: after deactivating HFW, the board remains in
          uninterlaced mode (interlacing = 0)

        Args:
            flag: True to activate HFW mode, False to deactivate

        Returns:
            Error message
        """
        logging.debug(self.logdebug + "setHighFullWell; flag = " + str(flag))
        err0 = ""
        if flag:
            if self.ZDT:
                logging.warning(
                    self.logwarn + "ZDT mode will be disengaged because of HFW "
                    "setting "
                )
                err0 = self.setZeroDeadTime(False)
            err1, _ = self.ca.setSubregister("HFW", "1")
            self.HFW = False  # preclude HFW deactivation message in setInterlacing
            self.setInterlacing(0)
            self.HFW = True
            logging.info(self.loginfo + "High Full Well mode active")
        else:
            self.HFW = False
            err1, _ = self.ca.setSubregister("HFW", "0")
            logging.info(self.loginfo + "High Full Well mode inactivate")
        err = err0 + err1
        if err:
            logging.error(self.logerr + "HFW option may not be set correctly ")
        return err

```

6.2.3.9 setInterlacing()

```
nsCamera.sensors.daedalus.daedalus.setInterlacing (
              self,
              ifactor = None,
              side = None )
```

```
Sets interlacing factor. NOTE: if called directly when HFW or ZDT mode is active,
this will disengage those modes automatically. If hemispheres have different
factors when the image is acquired, the resulting frames are separated into
half-width images

Args:
    ifactor: number of interlaced lines (generates ifactor + 1 images per frame)
      defaults to 0 (no interlacing)
    side: identify particular hemisphere (A or B) to control. If left blank,
      control both hemispheres

Returns:
    integer: active interlacing factor (unchanged if error)
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 269 of file daedalus.py.

```
    def setInterlacing(self, ifactor=None, side=None):
        """
        Sets interlacing factor. NOTE: if called directly when HFW or ZDT mode is active,
        this will disengage those modes automatically. If hemispheres have different
        factors when the image is acquired, the resulting frames are separated into
        half-width images

        Args:
            ifactor: number of interlaced lines (generates ifactor + 1 images per frame)
              defaults to 0 (no interlacing)
            side: identify particular hemisphere (A or B) to control. If left blank,
              control both hemispheres

        Returns:
            integer: active interlacing factor (unchanged if error)
        """
        logging.debug(self.logdebug + "setInterlacing; ifactor = " + str(ifactor))
        if ifactor is None:
            ifactor = 0
        if (
            not isinstance(ifactor, int)
            or ifactor < 0
            or ifactor > (self.maxheight - 1)
        ):
            err = (
                self.logerr + "invalid interlacing factor submitted. "
                "Interlacing remains unchanged. "
            )
            logging.error(err)
            return self.interlacing
        if self.HFW:
            logging.warning(
                self.logwarn + "HFW mode will be disengaged because of new "
                "interlacing setting "
            )
            self.setHighFullWell(False)
        if self.ZDT:
            logging.warning(
                self.logwarn + "ZDT mode will be disengaged because of new "
                "interlacing setting "
            )
            self.setZeroDeadTime(False)
        if ifactor == 0:
            bitscheme = self.maxheight * [0]
            # deactivating one side shouldn't turn off enable for both sides
            # TODO: is it a problem if sides are set separately, so interlacing is zero
            #    but still enabled?
            if side is None:
                self.ca.setSubregister("INTERLACING_EN", "0")
        else:
            pattern = [0] + ifactor * [1]
            reps = 1 + self.maxheight // (ifactor + 1)
            bitscheme = (reps * pattern)[0 : self.maxheight]
            self.ca.setSubregister("INTERLACING_EN", "1")
        err = ""
        for regnum in range(32):
            regbits = bitscheme[32 * regnum : 32 * (regnum + 1)]
            logging.debug(self.logdebug + "regbits = " + str(regbits))
            # generated pattern is reverse order from placement in register (element 0
            #   of the list is the LSB of the register)
            bitsrev = regbits[::-1]
            s = [str(i) for i in bitsrev]
            b = "".join(s)  # assemble as binary number for processing
            hexval = "%x" % int(b, 2)
            val = hexval.zfill(8)
            err0 = ""
            err1 = ""
            if side is None or side.lower() == "a":
                lname = "RSL_CONFIG_DATA_A" + str(regnum)
                err1, _ = self.ca.setRegister(lname, val)
                self.interlacing[1] = ifactor
            if side is None or side.lower() == "b":
                rname = "RSL_CONFIG_DATA_B" + str(regnum)
                err0, _ = self.ca.setRegister(rname, val)
                self.interlacing[0] = ifactor
            err = err + err0 + err1
        if err:
            logging.error(self.logerr + "interlacing may not be set correctly: " + err)
        logging.info(self.loginfo + "Interlacing set to " + str(self.interlacing))
        if self.interlacing[0] == self.interlacing[1]:
            self.columns = 1
        else:
            self.columns = 2
        return self.interlacing

```

6.2.3.10 setManualShutters()

```
nsCamera.sensors.daedalus.daedalus.setManualShutters (
                self,
                timing = None )
```

Legacy alias for setManualTiming()

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 602 of file daedalus.py.

```
    def setManualShutters(self, timing=None):
        """
        Legacy alias for setManualTiming()
        """
        self.setManualTiming(timing)

```

6.2.3.11 setManualTiming()

```
nsCamera.sensors.daedalus.daedalus.setManualTiming (
                self,
                timing = None )
```

```
Manual shutter timing, five intervals given in nanoseconds, e.g.,
  [100,50,100,50,100] for frame 0 open for 100 ns, an interframe pause of 50 ns,
   frame 1 open for 100 ns, etc. Timing is set for both hemispheres.

The actual timing is rounded down to the nearest multiple of 25 ns. (Each
  count = 25 ns. e.g., a request for 140 ns rounds down to a count of '5',
  which corresponds to 125 ns))
    - Minimum timing is 75 ns
    - Maximum is 25 * 2^30 ns (approximately 27 seconds)

Args:
    timing: 5-element list in nanoseconds

Returns:
    tuple (error string, response string from final message)
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 608 of file daedalus.py.

```
    def setManualTiming(self, timing=None):
        """
        Manual shutter timing, five intervals given in nanoseconds, e.g.,
          [100,50,100,50,100] for frame 0 open for 100 ns, an interframe pause of 50 ns,
           frame 1 open for 100 ns, etc. Timing is set for both hemispheres.

        The actual timing is rounded down to the nearest multiple of 25 ns. (Each
          count = 25 ns. e.g., a request for 140 ns rounds down to a count of '5',
          which corresponds to 125 ns))
            - Minimum timing is 75 ns
            - Maximum is 25 * 2^30 ns (approximately 27 seconds)

        Args:
            timing: 5-element list in nanoseconds

        Returns:
            tuple (error string, response string from final message)
        """
        if timing is None:
            logging.info(
                self.loginfo
                + "No manual timing setting provided, defaulting to (100, 150, 100, "
                " 150, 100, 150, 100) for both hemispheres"
            )
            timing = [(100, 150, 100, 150, 100)]

        logging.info(self.loginfo + "Manual shutter sequence: " + str(timing))
        flattened = flattenlist(timing)
        if (
            len(flattened) != 5
            or not all(isinstance(x, (int, float)) for x in flattened)
            or not all(x >= 25 for x in flattened)
        ):
            err = self.logerr + "Invalid manual shutter timing list: " + str(timing)
            logging.error(err + "; timing settings unchanged")
            return err, "00000000"

        timecounts = [int(a // 25) for a in flattened]
        self.ca.sensmanual = timing
        self.ca.senstiming = {}  # clear HST settings from ca object

        control_messages = [
            ("MANSHUT_MODE", "1"),
            ("EXT_PHI_CLK_SH0_ON", "{0:#0{1}x}".format(timecounts[0], 10)[2:10]),
            ("EXT_PHI_CLK_SH0_OFF", "{0:#0{1}x}".format(timecounts[1], 10)[2:10]),
            ("EXT_PHI_CLK_SH1_ON", "{0:#0{1}x}".format(timecounts[2], 10)[2:10]),
            ("EXT_PHI_CLK_SH1_OFF", "{0:#0{1}x}".format(timecounts[3], 10)[2:10]),
            ("EXT_PHI_CLK_SH2_ON", "{0:#0{1}x}".format(timecounts[4], 10)[2:10]),
        ]
        return self.ca.submitMessages(control_messages, " setManualShutters: ")

```

6.2.3.12 setPhiDelay()

```
nsCamera.sensors.daedalus.daedalus.setPhiDelay (
                self,
                side = None,
                delay = 0 )
```

```
Use phi delay timer. Actual delay is rounded down to multiple of .15 ns, up to a
  maximum delay of 1.5 ns
Args:
    side: hemisphere to delay; if None, delay both hemispheres
    delay: phi delay in ns

Returns:
    String of errors, if any
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 516 of file daedalus.py.

```
    def setPhiDelay(self, side=None, delay=0):
        """
        Use phi delay timer. Actual delay is rounded down to multiple of .15 ns, up to a
          maximum delay of 1.5 ns
        Args:
            side: hemisphere to delay; if None, delay both hemispheres
            delay: phi delay in ns

        Returns:
            String of errors, if any
        """
        logging.debug(self.logdebug + "setPhiDelay; delay = " + str(delay))
        if (
            not (isinstance(delay, int) or isinstance(delay, float))
            or delay < 0
            or delay > 1.5
        ):
            err = (
                self.logerr + "invalid phi delay submitted. Delay remains "
                "unchanged. "
            )
            logging.error(err)
            return err
        delayblocks = int(delay / 0.15)
        if delayblocks < 0:
            delayblocks = 0
        if delayblocks > 10:
            delayblocks = 10
        delayseq = (10 - delayblocks) * [0] + delayblocks * [1]
        seqstr = "".join(str(x) for x in delayseq)
        err1 = ""
        err2 = ""
        if side is None or side.upper() == "A":
            err1, _ = self.ca.setSubregister("PHI_DELAY_A", seqstr)
        if side is None or side.upper() == "B":
            err2, _ = self.ca.setSubregister("PHI_DELAY_B", seqstr)
        delayed = delayblocks * 0.15
        logging.info(self.loginfo + "Actual phi delay = " + str(delayed) + " ns")
        return err1 + err2

```

6.2.3.13 setTriggerDelay()

```
nsCamera.sensors.daedalus.daedalus.setTriggerDelay (
                self,
                delay = 0 )
```

```
Use trigger delay timer. Actual delay is rounded down to multiple of .15 ns, up
  to a maximum delay of 6 ns
```

```
Args:
    delay: trigger delay in ns

Returns:
    String of errors, if any
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 474 of file daedalus.py.

```
    def setTriggerDelay(self, delay=0):
        """
        Use trigger delay timer. Actual delay is rounded down to multiple of .15 ns, up
          to a maximum delay of 6 ns

        Args:
            delay: trigger delay in ns

        Returns:
            String of errors, if any
        """
        logging.debug(self.logdebug + "setTriggerDelay; delay = " + str(delay))
        if (
            not (isinstance(delay, int) or isinstance(delay, float))
            or delay < 0
            or delay > 6
        ):
            err = (
                self.logerr + "invalid trigger delay submitted. Delay remains "
                "unchanged. "
            )
            logging.error(err)
            return err
        delayblocks = int(delay / 0.15)
        if delayblocks < 0:
            delayblocks = 0
        if delayblocks > 40:
            delayblocks = 40
        delayseq = (40 - delayblocks) * [0] + delayblocks * [1]
        seqstr = "".join(str(x) for x in delayseq)
        seqhex = "%x" % int(seqstr, 2)
        logging.debug(self.logdebug + "seqhex = " + str(seqhex))
        highpart = seqhex[-10:-8].zfill(8)
        lowpart = seqhex[-8:].zfill(8)
        self.ca.setSubregister("HST_DEL_EN", "1")
        err0, _ = self.ca.setRegister("HST_TRIGGER_DELAY_DATA_LO", lowpart)
        err1, _ = self.ca.setRegister("HST_TRIGGER_DELAY_DATA_HI", highpart)
        err2, _ = self.ca.setSubregister("HST_MODE", "1")
        delayed = delayblocks * 0.15
        logging.info(self.loginfo + "Actual trigger delay = " + str(delayed) + " ns")
        return err0 + err1 + err2

```

6.2.3.14 setZeroDeadTime()

```
nsCamera.sensors.daedalus.daedalus.setZeroDeadTime (
              self,
              flag = True,
              side = None )
```

```
Activates Zero Dead Time mode. Even rows follow the assigned HST schedule; odd
  rows are acquired while the 'shutter' for the even rows are closed. High Full
  Well mode and interlacing will be automatically deactivated.
*NOTE* after deactivating ZDT, the board reverts to uninterlaced mode
  (interlacing = 0)

Args:
    flag: True to activate ZDT mode, False to deactivate
    side: identify particular hemisphere (A or B) to control. If left blank,
      control both hemispheres

Returns:
    Error message
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 390 of file daedalus.py.

```
    def setZeroDeadTime(self, flag=True, side=None):
        """
        Activates Zero Dead Time mode. Even rows follow the assigned HST schedule; odd
          rows are acquired while the 'shutter' for the even rows are closed. High Full
          Well mode and interlacing will be automatically deactivated.
        *NOTE* after deactivating ZDT, the board reverts to uninterlaced mode
          (interlacing = 0)

        Args:
            flag: True to activate ZDT mode, False to deactivate
            side: identify particular hemisphere (A or B) to control. If left blank,
              control both hemispheres

        Returns:
            Error message
        """
        logging.debug(self.logdebug + "setZeroDeadTime; flag = " + str(flag))
        err0 = ""
        err1 = ""
        err2 = ""
        if flag:
            if self.HFW:
                logging.warning(
                    self.logwarn + "HFW mode will be disengaged because of ZDT "
                    "setting "
                )
                err0 = self.setHighFullWell(False)
            if side is None or side.lower() == "a":
                err2, _ = self.ca.setSubregister("ZDT_A", "1")
                self.interlacing[0] = 1
            if side is None or side.lower() == "b":
                err1, _ = self.ca.setSubregister("ZDT_B", "1")
                self.interlacing[1] = 1
            # self.ZDT = False  # preclude ZDT deactivation message in setInterlacing
            # if self.interlacing != [0, 0]:
            #     self.setInterlacing(0)
            # TODO: need to handle flags when ZDT active for just one side
            self.ZDT = True
            logging.info(
                self.loginfo + "Zero Dead Time mode active; actual interlacing = 1"
            )
        else:
            self.ZDT = False
            if side is None or side.lower() == "a":
                err2, _ = self.ca.setSubregister("ZDT_A", "0")
            if side is None or side.lower() == "b":
                err1, _ = self.ca.setSubregister("ZDT_B", "0")
            self.setInterlacing(0)
            logging.info(self.loginfo + "Zero Dead Time mode inactivate")
        err = err0 + err1 + err2
        if err:
            logging.error(self.logerr + "ZDT option may not be set correctly ")
        return err

```

6.2.4 Member Data Documentation

6.2.4.1 bytesperpixel

```
int nsCamera.sensors.daedalus.daedalus.bytesperpixel = 2  [static]
```

Definition at line 36 of file daedalus.py.

6.2.4.2 ca

```
nsCamera.sensors.daedalus.daedalus.ca
```

Definition at line 57 of file daedalus.py.

6.2.4.3 columns [1/2]

```
int nsCamera.sensors.daedalus.daedalus.columns = 1  [static]
```

Definition at line 51 of file daedalus.py.

6.2.4.4 columns [2/2]

```
nsCamera.sensors.daedalus.daedalus.columns
```

Definition at line 349 of file daedalus.py.

6.2.4.5 detect

```
str nsCamera.sensors.daedalus.daedalus.detect = "DAEDALUS_DET"  [static]
```

Definition at line 38 of file daedalus.py.

6.2.4.6 firstframe

```
int nsCamera.sensors.daedalus.daedalus.firstframe = 0  [static]
```

Definition at line 43 of file daedalus.py.

6.2.4.7 firstrow

```
int nsCamera.sensors.daedalus.daedalus.firstrow = 0  [static]
```

Definition at line 48 of file daedalus.py.

6.2.4.8 fpganumID

```
int nsCamera.sensors.daedalus.daedalus.fpganumID = 2  [static]
```

Definition at line 37 of file daedalus.py.

6.2.4.9 height

```
int nsCamera.sensors.daedalus.daedalus.height = 1024  [static]
```

Definition at line 47 of file daedalus.py.

6.2.4.10 HFW [1/2]

```
bool nsCamera.sensors.daedalus.daedalus.HFW = False  [static]
```

Definition at line 42 of file daedalus.py.

6.2.4.11 HFW [2/2]

```
nsCamera.sensors.daedalus.daedalus.HFW
```

Definition at line 377 of file daedalus.py.

6.2.4.12 interlacing [1/2]

```
list nsCamera.sensors.daedalus.daedalus.interlacing = [0, 0]  [static]
```

Definition at line 50 of file daedalus.py.

6.2.4.13 interlacing [2/2]

```
nsCamera.sensors.daedalus.daedalus.interlacing
```

Definition at line 347 of file daedalus.py.

6.2.4.14 lastframe

```
int nsCamera.sensors.daedalus.daedalus.lastframe = 2  [static]
```

Definition at line 44 of file daedalus.py.

6.2.4.15 lastrow

```
int nsCamera.sensors.daedalus.daedalus.lastrow = 1023  [static]
```

Definition at line 49 of file daedalus.py.

6.2.4.16 loglabel

```
str nsCamera.sensors.daedalus.daedalus.loglabel = "[Daedalus] "  [static]
```

Definition at line 40 of file daedalus.py.

6.2.4.17 maxframe

```
int nsCamera.sensors.daedalus.daedalus.maxframe = 2  [static]
```

Definition at line 33 of file daedalus.py.

6.2.4.18 maxheight

```
int nsCamera.sensors.daedalus.daedalus.maxheight = 1024  [static]
```

Definition at line 35 of file daedalus.py.

6.2.4.19 maxwidth

```
int nsCamera.sensors.daedalus.daedalus.maxwidth = 512  [static]
```

Definition at line 34 of file daedalus.py.

6.2.4.20 minframe

```
int nsCamera.sensors.daedalus.daedalus.minframe = 0  [static]
```

Definition at line 32 of file daedalus.py.

6.2.4.21 nframes

```
int nsCamera.sensors.daedalus.daedalus.nframes = 3  [static]
```

Definition at line 45 of file daedalus.py.

6.2.4.22 padToFull

```
bool nsCamera.sensors.daedalus.daedalus.padToFull = True  [static]
```

Definition at line 52 of file daedalus.py.

6.2.4.23 sens_registers

```
nsCamera.sensors.daedalus.daedalus.sens_registers
```

Definition at line 60 of file daedalus.py.

6.2.4.24 sens_subregisters

```
nsCamera.sensors.daedalus.daedalus.sens_subregisters
```

Definition at line 153 of file daedalus.py.

6.2.4.25 sensfam

```
str nsCamera.sensors.daedalus.daedalus.sensfam = "Daedalus"  [static]
```

Definition at line 39 of file daedalus.py.

6.2.4.26 specwarn

```
str nsCamera.sensors.daedalus.daedalus.specwarn = ""  [static]
```

Definition at line 31 of file daedalus.py.

6.2.4.27 toffset

```
float nsCamera.sensors.daedalus.daedalus.toffset = -165.76  [static]
```

Definition at line 53 of file daedalus.py.

6.2.4.28 tslope

```
float nsCamera.sensors.daedalus.daedalus.tslope = 81.36  [static]
```

Definition at line 54 of file daedalus.py.

6.2.4.29 width

```
int nsCamera.sensors.daedalus.daedalus.width = 512  [static]
```

Definition at line 46 of file daedalus.py.

6.2.4.30 ZDT `[1/2]`

```
bool nsCamera.sensors.daedalus.daedalus.ZDT = False  [static]
```

Definition at line 41 of file daedalus.py.

6.2.4.31 ZDT `[2/2]`

```
nsCamera.sensors.daedalus.daedalus.ZDT
```

Definition at line 427 of file daedalus.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/sensors/daedalus.py

6.3 nsCamera.utils.misc.fakeCA Class Reference

Public Member Functions

- __init__ (self, sensorname="icarus2", firstframe=0, lastframe=3, width=512, height=1024, padToFull=True, firstrow=0, lastrow=1023, maxwidth=512, maxheight=1024, bytesperpixel=2, interlacing=None, columns=1, logtag=None)
- partition (self, parsed, columns)

Public Attributes

- sensorname
- boardname
- padToFull
- logtag
- logcritbase
- logerrbase
- logwarnbase
- loginfobase
- logdebugbase
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- sensor

6.3.1 Detailed Description

```
Fake 'cameraAssembler' object to use as a parameter object in offline functions.
  Returned by generateFrames(), it contains the frame details required to save and
  plot images.
```

Definition at line 203 of file misc.py.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 __init__()

```
nsCamera.utils.misc.fakeCA.__init__ (
              self,
              sensorname = "icarus2",
              firstframe = 0,
              lastframe = 3,
              width = 512,
              height = 1024,
              padToFull = True,
              firstrow = 0,
              lastrow = 1023,
              maxwidth = 512,
              maxheight = 1024,
              bytesperpixel = 2,
              interlacing = None,
              columns = 1,
              logtag = None )
```

Definition at line 210 of file misc.py.

```
    ):
        self.sensorname = sensorname
        self.boardname = None
        self.padToFull = padToFull
        if logtag is None:
            self.logtag = ""
        else:
            self.logtag = logtag

        self.logcritbase = "CRITICAL" + self.logtag + ": "
        self.logerrbase = "ERROR" + self.logtag + ": "
        self.logwarnbase = "WARNING" + self.logtag + ": "
        self.loginfobase = "INFO" + self.logtag + ": "
        self.logdebugbase = "DEBUG" + self.logtag + ": "

        self.logcrit = self.logcritbase + "[FS] "
        self.logerr = self.logerrbase + "[FS] "
        self.logwarn = self.logwarnbase + "[FS] "
        self.loginfo = self.loginfobase + "[FS] "
        self.logdebug = self.logdebugbase + "[FS] "

        if self.sensorname == "icarus":
            import nsCamera.sensors.icarus as snsr
        elif self.sensorname == "icarus2":
            import nsCamera.sensors.icarus2 as snsr
        elif self.sensorname == "daedalus":
            import nsCamera.sensors.daedalus as snsr

        self.sensor = snsr(self)

```

6.3.3 Member Function Documentation

6.3.3.1 partition()

```
nsCamera.utils.misc.fakeCA.partition (
              self,
              parsed,
              columns )
```

Definition at line 256 of file misc.py.

```
    def partition(self, parsed, columns):
        # return nsCamera.utils.misc.partition(self, parsed, columns)
        return partition(self, parsed, columns)


```

6.3.4 Member Data Documentation

6.3.4.1 boardname

```
nsCamera.utils.misc.fakeCA.boardname
```

Definition at line 228 of file misc.py.

6.3.4.2 logcrit

```
nsCamera.utils.misc.fakeCA.logcrit
```

Definition at line 241 of file misc.py.

6.3.4.3 logcritbase

```
nsCamera.utils.misc.fakeCA.logcritbase
```

Definition at line 235 of file misc.py.

6.3.4.4 logdebug

```
nsCamera.utils.misc.fakeCA.logdebug
```

Definition at line 245 of file misc.py.

6.3.4.5 logdebugbase

```
nsCamera.utils.misc.fakeCA.logdebugbase
```

Definition at line 239 of file misc.py.

6.3.4.6 logerr

```
nsCamera.utils.misc.fakeCA.logerr
```

Definition at line 242 of file misc.py.

6.3.4.7 logerrbase

```
nsCamera.utils.misc.fakeCA.logerrbase
```

Definition at line 236 of file misc.py.

6.3.4.8 loginfo

```
nsCamera.utils.misc.fakeCA.loginfo
```

Definition at line 244 of file misc.py.

6.3.4.9 loginfobase

```
nsCamera.utils.misc.fakeCA.loginfobase
```

Definition at line 238 of file misc.py.

6.3.4.10 logtag

```
nsCamera.utils.misc.fakeCA.logtag
```

Definition at line 231 of file misc.py.

6.3.4.11 logwarn

```
nsCamera.utils.misc.fakeCA.logwarn
```

Definition at line 243 of file misc.py.

6.3.4.12 logwarnbase

```
nsCamera.utils.misc.fakeCA.logwarnbase
```

Definition at line 237 of file misc.py.

6.3.4.13 padToFull

```
nsCamera.utils.misc.fakeCA.padToFull
```

Definition at line 229 of file misc.py.

6.3.4.14 sensor

```
nsCamera.utils.misc.fakeCA.sensor
```

Definition at line 254 of file misc.py.

6.3.4.15 sensorname

```
nsCamera.utils.misc.fakeCA.sensorname
```

Definition at line 227 of file misc.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/utils/misc.py

6.4 nsCamera.comms.GigE.GigE Class Reference

Classes

- class ZESTETM1_CARD_INFO

Public Member Functions

- __init__ (self, camassem)
- sendCMD (self, pkt)
- arm (self, mode)
- readFrames (self, waitOnSRAM, timeout=0, fast=False, columns=1)
- readoff (self, waitOnSRAM, timeout=0, fast=False, columns=1)
- writeSerial (self, outstring, timeout)
- readSerial (self, size, timeout=None)
- openDevice (self, timeout=30)
- closeDevice (self)
- getCardIP (self)
- getCardInfo (self)

Public Attributes

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- mode
- writeTimeout
- readTimeout
- payloadsize
- skipError
- ZErrorDict
- dport
- closecard
- CardInfo
- CardInfoP
- ZCountCards
- ZESTETM1_CARD_INFO
- ZOpenConnection
- ZWriteData
- ZReadData
- Connection

Protected Attributes

- _zest

6.4.1 Detailed Description

```
Code to manage Gigabit Ethernet connection to board. Each GigE object manages a
  single OT card; to use multiple cards, instantiate multiple cameraAssembler
  objects, each specifying the unique IPs of the corresponding OT card.

Note: Orange Tree card must be configured before use. See the README for details

Exposed methods:
    arm() - puts camera into wait state for external trigger
    readFrames() - waits for data ready register flag, then copies camera image data
      into numpy arrays
    readoff() - waits for data ready register flag, then copies camera image data
      into numpy arrays; returns payload, payload size, and error message
    sendCMD(pkt) - sends packet object via serial port
    readSerial(size, timeout) - read 'size' bytes from connection
    writeSerial(outstring) - submits string 'outstring' over connection
    closeDevice() - close connections and free resources
    getCardIP() - returns IP address of OT card
    getCardInfo() - prints report of details of OT card and connection
```

Definition at line 27 of file GigE.py.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 __init__()

```
nsCamera.comms.GigE.GigE.__init__ (
                self,
                camassem )
```

```
Args:
    camassem: parent cameraAssembler object
```

Definition at line 49 of file GigE.py.

```
    def __init__(self, camassem):
        """
        Args:
            camassem: parent cameraAssembler object
        """
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[GigE] "
        self.logerr = self.ca.logerrbase + "[GigE] "
        self.logwarn = self.ca.logwarnbase + "[GigE] "
        self.loginfo = self.ca.loginfobase + "[GigE] "
        self.logdebug = self.ca.logdebugbase + "[GigE] "
        logging.info(self.loginfo + "Initializing GigE comms object")
        self.mode = 1
        self.writeTimeout = 10000
        self.readTimeout = 10000
        self.payloadsize = (
            self.ca.sensor.width
            * self.ca.sensor.height
            * self.ca.sensor.nframes
            * self.ca.sensor.bytesperpixel
        )
        logging.debug(
            self.logdebug + "Payload size: " + str(self.payloadsize) + " bytes"
        )
        self.skipError = False

        self.ZErrorDict = {
            0x8000: "Socket Error",
            0x8001: "Internal Error",
            0x8002: "Illegal Status Code",
            0x8003: "Null Parameter",
            0x8004: "Out of Memory",
            0x8005: "Invalid Connection Type",
            0x8006: "Illegal Connection",
            0x8007: "Socket Closed Unexpectedly",
            0x8008: "Timeout",
            0x8009: "Illegal Parameter",
        }

        if self.ca.port:
            logging.debug(
                self.logdebug + "Port supplied to GigE.py: " + str(self.ca.port)
            )
            if isinstance(self.ca.port, int) and 0 < self.ca.port < 65536:
                self.dport = self.ca.port
            else:
                logging.error(
                    self.logerr + "Invalid port number supplied, defaulting to "
                    "20482 "
                )
                self.dport = 20482
        else:
            self.dport = 20482  # default

        self.ca.port = self.dport
        logging.debug(self.logdebug + "Port used by GigE.py: " + str(self.dport))

        logging.debug(self.logdebug + "CPU architecture: " + str(self.ca.arch))
        if self.ca.arch == "64bit":
            arch = "64"
        else:
            arch = "32"

        logging.debug(self.logdebug + "Operating system: " + str(self.ca.platform))
        if self.ca.platform == "Windows":
            lib_name = "ZestETM1.dll"
        elif self.ca.platform == "Linux" or self.ca.platform == "Darwin":
            lib_name = "libZestETM1.so"
        else:
            logging.warning(
                self.logwarn + "System does not self-identify as Linux, Windows, "
                "or Mac. Assuming posix-style libraries "
            )
            lib_name = "libZestETM1.so"

        self.closecard = False

        libpath = os.path.join(self.ca.packageroot, "comms", "ZestETM1", arch, lib_name)
        self._zest = C.CDLL(libpath)

        self.CardInfo = self.ZESTETM1_CARD_INFO()
        self.CardInfoP = C.pointer(self.CardInfo)

        # functions
        self.ZCountCards = self._zest.ZestETM1CountCards
        self.ZCountCards.argtypes = [
            C.POINTER(C.c_ulong),
            C.POINTER(C.POINTER(self.ZESTETM1_CARD_INFO)),
            C.c_int,
        ]

        self.ZOpenConnection = self._zest.ZestETM1OpenConnection
        self.ZOpenConnection.argtypes = [
            C.POINTER(self.ZESTETM1_CARD_INFO),
            C.c_int,
            C.c_ushort,
            C.c_ushort,
            C.POINTER(C.c_void_p),
        ]

        self.ZWriteData = self._zest.ZestETM1WriteData
        self.ZWriteData.argtypes = [
            C.c_void_p,
            C.c_void_p,
            C.c_ulong,
            C.POINTER(C.c_ulong),
            C.c_ulong,
        ]

        self.ZReadData = self._zest.ZestETM1ReadData
        self.ZReadData.argtypes = [
            C.c_void_p,
            C.c_void_p,
            C.c_ulong,
            C.POINTER(C.c_ulong),
            C.c_ulong,
        ]

        self.Connection = C.c_void_p()
        self.openDevice(self.ca.timeout)

```

6.4.3 Member Function Documentation

6.4.3.1 arm()

```
nsCamera.comms.GigE.GigE.arm (
                self,
                mode )
```

```
Puts camera into wait state for trigger. Mode determines source; defaults to
```

```
  'Hardware'

Args:
    mode:   'Software'|'S' activates software, disables hardware triggering
            'Hardware'|'H' activates hardware, disables software triggering
              Hardware is the default

Returns:
    tuple (error, response string)
```

Definition at line 221 of file GigE.py.

```
    def arm(self, mode):
        """
        Puts camera into wait state for trigger. Mode determines source; defaults to
          'Hardware'

        Args:
            mode:   'Software'|'S' activates software, disables hardware triggering
                    'Hardware'|'H' activates hardware, disables software triggering
                      Hardware is the default

        Returns:
            tuple (error, response string)
        """
        if not mode:
            mode = "Hardware"
            logging.info(self.loginfo + "arm")
        logging.debug(self.logdebug + "arming mode: " + str(mode))
        self.ca.clearStatus()
        self.ca.latchPots()
        err, resp = self.ca.startCapture(mode)
        if err:
            logging.error(self.logerr + "unable to arm camera")
        else:
            self.ca.armed = True
            self.skipError = True
        return err, resp
```

6.4.3.2 closeDevice()

```
nsCamera.comms.GigE.GigE.closeDevice (
                self )
```

```
Close connection to Orange Tree card and free resources
```

Definition at line 467 of file GigE.py.

```
    def closeDevice(self):
        """
        Close connection to Orange Tree card and free resources
        """
        logging.debug(self.logdebug + "Closing connection to Orange Tree card")
        self._zest.ZestETM1CloseConnection(self.Connection)
        if self.closecard:
            try:
                self._zest.ZestETM1FreeCards(self.CardInfoP)
            except SystemExit:
                raise
            except KeyboardInterrupt:
                raise
            except Exception:
                logging.error(self.logerr + "Error reported in OT card closure")
        self._zest.ZestETM1Close()
```

6.4.3.3 getCardInfo()

```
nsCamera.comms.GigE.GigE.getCardInfo (
              self )
```

```
Prints status message with information returned by OT card
```

Definition at line 493 of file GigE.py.

```
    def getCardInfo(self):
        """
        Prints status message with information returned by OT card
        """
        ci = self.CardInfoP.contents
        print("GigE Card Status:")
        print("-------------")
        print("IP: " + ".".join(str(e) for e in [b for b in ci.IPAddr]))
        print("ControlPort: " + str(ci.ControlPort))
        print("Timeout: " + str(ci.Timeout))
        print("HTTPPort: " + str(ci.HTTPPort))
        print("MACAddr: " + ".".join(format(e, "02X") for e in [b for b in ci.MACAddr]))
        print("SubNet: " + ".".join(str(e) for e in [b for b in ci.SubNet]))
        print("Gateway: " + ".".join(str(e) for e in [b for b in ci.Gateway]))
        print("SerialNumber: " + str(ci.SerialNumber))
        print("FirmwareVersion: " + str(ci.FirmwareVersion))
        print("HardwareVersion: " + str(ci.HardwareVersion))
        print("-------------")

```

6.4.3.4 getCardIP()

```
nsCamera.comms.GigE.GigE.getCardIP (
              self )
```

```
Query IP address of OT card

Returns: address of OT card as list of bytes
```

Definition at line 484 of file GigE.py.

```
    def getCardIP(self):
        """
        Query IP address of OT card

        Returns: address of OT card as list of bytes
        """
        return self.CardInfo.IPAddr

```

6.4.3.5 openDevice()

```
nsCamera.comms.GigE.GigE.openDevice (
              self,
              timeout = 30 )
```

```
Find Orange Tree card and open a connection; if IP is supplied as parameter for
  the CameraAssembler, bypass network search and connect directly to indicated
  IP address

Args:
    timeout: timeout in seconds for attempting to connect to a card
```

Definition at line 402 of file GigE.py.

```
    def openDevice(self, timeout=30):
        """
        Find Orange Tree card and open a connection; if IP is supplied as parameter for
          the CameraAssembler, bypass network search and connect directly to indicated
          IP address

        Args:
            timeout: timeout in seconds for attempting to connect to a card
        """
        err = self._zest.ZestETM1Init()
        if err:
            logging.critical(self.logcrit + "ZestETM1Init failure")
            sys.exit(1)
        logging.info(self.loginfo + "searching for Orange Tree cards")
        NumCards = C.c_ulong(0)

        if self.ca.iplist:
            ubyte4 = C.c_ubyte * 4
            self.CardInfo.IPAddr = ubyte4(*self.ca.iplist)
            self.CardInfo.ControlPort = C.c_ushort(self.dport)
            self.CardInfo.Timeout = C.c_ulong(self.writeTimeout)
            self.closecard = False
        else:
            wait = 0
            while True:
                logging.debug(
                    self.logdebug + "openDevice: connection wait = " + str(wait)
                )
                err = self.ZCountCards(C.byref(NumCards), C.byref(self.CardInfoP), 1000)
                if err:
                    logging.critical(self.logcrit + "CountCards failure")
                    sys.exit(1)
                if NumCards.value > 0:
                    break
                if wait == timeout:
                    logging.critical(
                        "{}No Orange Tree cards found in {} seconds".format(
                            self.logcrit, timeout
                        )
                    )
                    sys.exit(1)
                elif not wait % 5:
                    logging.info(
                        "{}Still trying to connect after {} seconds...".format(
                            self.loginfo, wait
                        )
                    )
                wait += 1
            logging.info(
                self.loginfo + str(NumCards.value) + " Orange Tree card(s) found"
            )  # TODO: add check for GigE bit in board description
        err = self.ZOpenConnection(
            self.CardInfoP, 0, self.dport, 0, C.byref(self.Connection)
        )
        if err:
            if err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.critical(
                    self.logcrit + "OpenConnection failure: " + self.ZErrorDict[err]
                )
            sys.exit(1)

```

6.4.3.6 readFrames()

```
nsCamera.comms.GigE.GigE.readFrames (
               self,
               waitOnSRAM,
               timeout = 0,
               fast = False,
               columns = 1 )
```

```
Copies image data from board into numpy arrays.

Args:
    waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
      data
    timeout: passed to waitForSRAM; after this many seconds begin copying data
      irrespective of SRAM_READY status; 'zero' means wait indefinitely
      WARNING: If acquisition fails, the SRAM will not contain a current image,
        but the code will copy the data anyway
    fast: if False, parse and convert frames to numpy arrays; if True, return
      unprocessed text stream
    columns: 1 for single image per frame, 2 for separate hemisphere images

Returns:
    list of numpy arrays OR raw text stream
```

Definition at line 248 of file GigE.py.

```
    def readFrames(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays.

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            list of numpy arrays OR raw text stream

        """
        frames, _, _ = self.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

```

6.4.3.7 readoff()

```
nsCamera.comms.GigE.GigE.readoff (
              self,
              waitOnSRAM,
              timeout = 0,
              fast = False,
              columns = 1 )
```

```
Copies image data from board into numpy arrays; returns data, length of data,
and error messages. Use 'readFrames()' unless you require this additional
information

Args:
    waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
      data
    timeout: passed to waitForSRAM; after this many seconds begin copying data
      irrespective of SRAM_READY status; 'zero' means wait indefinitely
      WARNING: If acquisition fails, the SRAM will not contain a current image,
        but the code will copy the data anyway
    fast: if False, parse and convert frames to numpy arrays; if True, return
      unprocessed text stream
    columns: 1 for single image per frame, 2 for separate hemisphere images
```

```
Returns:
    tuple (list of numpy arrays OR raw text stream, length of downloaded payload
      in bytes, payload error flag) since CRC check is handled by TCP/IP,
      payload error flag is always False for GigE
```

Definition at line 270 of file GigE.py.

```
    def readoff(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays; returns data, length of data,
        and error messages. Use 'readFrames()' unless you require this additional
        information

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            tuple (list of numpy arrays OR raw text stream, length of downloaded payload
              in bytes, payload error flag) since CRC check is handled by TCP/IP,
              payload error flag is always False for GigE
        """
        logging.info(self.loginfo + "readoff")
        logging.debug(
            self.logdebug
            + "readoff: waitonSRAM = "
            + str(waitOnSRAM)
            + "; timeout = "
            + str(timeout)
            + "; fast = "
            + str(fast)
        )
        # Wait for data to be ready on board
        # Skip wait only if explicitly tagged 'False' ('None' defaults to True)
        if waitOnSRAM is not False:
            self.ca.waitForSRAM(timeout)
        self.skipError = False
        err, rval = self.ca.readSRAM()
        if err:
            logging.error(self.logerr + "Error detected in readSRAM")
        elif self.ca.boardname == "llnl_v4":
            # self.ca.setSubregister('SWACK','1')
            pass
        # extract the data. Remove header; the FPGA returns a packet without the CRC
        #   suffix
        logging.debug(self.logdebug + "readoff: first 64 chars: " + str(rval[0:64]))
        data = rval[32:]
        if fast:
            return data, len(data) // 2, bool(err)
        else:
            parsed = generateFrames(self.ca, data, columns)
            return parsed, len(data) // 2, bool(err)

```

6.4.3.8 readSerial()

```
nsCamera.comms.GigE.GigE.readSerial (
            self,
            size,
            timeout = None )
```

```
Read bytes from the serial port. Does not verify packets.

Args:
   size: number of bytes to read
   timeout: serial timeout in sec (defaults to self.readTimeout)

Returns:
   tuple (error string, string read from serial port)
```

Definition at line 363 of file GigE.py.

```
    def readSerial(self, size, timeout=None):
        """
        Read bytes from the serial port. Does not verify packets.

        Args:
           size: number of bytes to read
           timeout: serial timeout in sec (defaults to self.readTimeout)

        Returns:
           tuple (error string, string read from serial port)
        """
        logging.debug(
            self.logdebug
            + "readSerial: size = "
            + str(size)
            + "; timeout = "
            + str(timeout)
        )
        if not timeout:
            timeout = self.readTimeout
        inbuff = C.create_string_buffer(size + 1)
        inbuffp = C.pointer(inbuff)
        readlen = C.c_ulong(0)
        err = self.ZReadData(self.Connection, inbuffp, size, C.byref(readlen), timeout)
        if err:
            if self.skipError:
                logging.debug(
                    self.logdebug + "readSerial: skipped error: " + self.ZErrorDict[err]
                )
                self.skipError = False
            elif err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.error(self.logerr + "readSerial error: " + self.ZErrorDict[err])
        return bytes2str(inbuff.raw)[:-2]

```

6.4.3.9 sendCMD()

```
nsCamera.comms.GigE.GigE.sendCMD (
              self,
              pkt )
```

```
Submit packet and verify the response packet.
Packet communications with FPGA omit CRC suffix, so adds fake CRC bytes to
  response

Args:
    pkt: Packet object

Returns:
    tuple (error, response string)
```

Definition at line 170 of file GigE.py.

```
    def sendCMD(self, pkt):
        """
        Submit packet and verify the response packet.
        Packet communications with FPGA omit CRC suffix, so adds fake CRC bytes to
```

```
            response

        Args:
            pkt: Packet object

        Returns:
            tuple (error, response string)
        """

        pktStr = pkt.pktStr()[0:16]
        logging.debug(self.logdebug + "sendCMD packet: " + str(pktStr))
        err = ""
        self.ca.writeSerial(pktStr)
        if (
            hasattr(self.ca, "board")
            and pktStr[4] == "0"
            and pktStr[5:8] == self.ca.board.registers["SRAM_CTL"]
        ):
            bufsize = self.payloadsize + 16
            resptext = self.readSerial(bufsize)

            if len(resptext) < 32:
                logging.debug(self.logdebug + "sendCMD resptext = " + str(resptext))
            else:
                logging.debug(
                    self.logdebug
                    + "sendCMD resptext (truncated) = "
                    + str(resptext)[0:32]
                )

            if len(resptext) < bufsize + 16:
                err += (
                    self.logerr + "sendCMD- packet too small, payload may be incomplete"
                )
                logging.error(err)
        else:
            # workaround for initial setup before board object has been initialized
            resp = self.readSerial(8)
            logging.debug(self.logdebug + "sendCMD response: " + str(resp))
            if len(resp) < 8:
                err += self.logerr + "sendCMD- response too small, returning zeros"
                resptext = "00000000000000000000"
                logging.error(err)
            else:
                resptext = resp + "0000"
        return err, resptext

```

6.4.3.10 writeSerial()

```
nsCamera.comms.GigE.GigE.writeSerial (
                self,
                outstring,
                timeout )
```

```
Transmit string to board

Args:
    outstring: string to write
    timeout: serial timeout in sec (defaults to self.writeTimeout)

Returns:
    integer number of bytes written
```

Definition at line 323 of file GigE.py.

```
    def writeSerial(self, outstring, timeout):
        """
        Transmit string to board

        Args:
            outstring: string to write
            timeout: serial timeout in sec (defaults to self.writeTimeout)

        Returns:
            integer number of bytes written
        """
        logging.debug(
            self.logdebug
            + "writeSerial: outstring = "
            + str(outstring)
            + "; timeout = "
            + str(timeout)
        )
        if not timeout:
            timeout = self.writeTimeout
        outstring = str2bytes(outstring)
        outbuff = C.create_string_buffer(outstring)
        outbuffp = C.pointer(outbuff)
        outbufflen = len(outstring)
        writelen = C.c_ulong(0)
        err = self.ZWriteData(
            self.Connection, outbuffp, outbufflen, C.byref(writelen), timeout
        )
        if err:
            if err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.error(
                    self.logerr + "writeSerial error: " + self.ZErrorDict[err]
                )
        logging.debug(self.logdebug + "writeSerial: writelen = " + str(writelen))
        return writelen

```

6.4.4 Member Data Documentation

6.4.4.1 _zest

`nsCamera.comms.GigE.GigE._zest [protected]`

Definition at line 127 of file GigE.py.

6.4.4.2 ca

`nsCamera.comms.GigE.GigE.ca`

Definition at line 54 of file GigE.py.

6.4.4.3 CardInfo

`nsCamera.comms.GigE.GigE.CardInfo`

Definition at line 129 of file GigE.py.

6.4.4.4 CardInfoP

`nsCamera.comms.GigE.GigE.CardInfoP`

Definition at line 130 of file GigE.py.

6.4.4.5 closecard

`nsCamera.comms.GigE.GigE.closecard`

Definition at line 124 of file GigE.py.

6.4.4.6 Connection

`nsCamera.comms.GigE.GigE.Connection`

Definition at line 167 of file GigE.py.

6.4.4.7 dport

`nsCamera.comms.GigE.GigE.dport`

Definition at line 93 of file GigE.py.

6.4.4.8 logcrit

`nsCamera.comms.GigE.GigE.logcrit`

Definition at line 55 of file GigE.py.

6.4.4.9 logdebug

`nsCamera.comms.GigE.GigE.logdebug`

Definition at line 59 of file GigE.py.

6.4.4.10 logerr

`nsCamera.comms.GigE.GigE.logerr`

Definition at line 56 of file GigE.py.

6.4.4.11 loginfo

`nsCamera.comms.GigE.GigE.loginfo`

Definition at line 58 of file GigE.py.

6.4.4.12 logwarn

`nsCamera.comms.GigE.GigE.logwarn`

Definition at line 57 of file GigE.py.

6.4.4.13 mode

`nsCamera.comms.GigE.GigE.mode`

Definition at line 61 of file GigE.py.

6.4.4.14 payloadsize

`nsCamera.comms.GigE.GigE.payloadsize`

Definition at line 64 of file GigE.py.

6.4.4.15 readTimeout

`nsCamera.comms.GigE.GigE.readTimeout`

Definition at line 63 of file GigE.py.

6.4.4.16 skipError

`nsCamera.comms.GigE.GigE.skipError`

Definition at line 73 of file GigE.py.

6.4.4.17 writeTimeout

`nsCamera.comms.GigE.GigE.writeTimeout`

Definition at line 62 of file GigE.py.

6.4.4.18 ZCountCards

`nsCamera.comms.GigE.GigE.ZCountCards`

Definition at line 133 of file GigE.py.

6.4.4.19 ZErrorDict

`nsCamera.comms.GigE.GigE.ZErrorDict`

Definition at line 75 of file GigE.py.

6.4.4.20 ZESTETM1_CARD_INFO

`nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO`

Definition at line 136 of file GigE.py.

6.4.4.21 ZOpenConnection

`nsCamera.comms.GigE.GigE.ZOpenConnection`

Definition at line 140 of file GigE.py.

6.4.4.22 ZReadData

`nsCamera.comms.GigE.GigE.ZReadData`

Definition at line 158 of file GigE.py.

6.4.4.23 ZWriteData

`nsCamera.comms.GigE.GigE.ZWriteData`

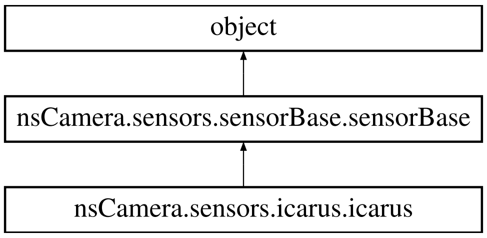
Definition at line 149 of file GigE.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/comms/GigE.py

6.5 nsCamera.sensors.icarus.icarus Class Reference

Inheritance diagram for nsCamera.sensors.icarus.icarus:


object
nsCamera.sensors.sensorBase.sensorBase
nsCamera.sensors.icarus.icarus

Public Member Functions

- __init__ (self, ca)
- checkSensorVoltStat (self)
- sensorSpecific (self)

Public Member Functions inherited from nsCamera.sensors.sensorBase.sensorBase

- init_board_specific (self)
- setInterlacing (self, ifactor)
- setHighFullWell (self, flag)
- setZeroDeadTime (self, flag)
- setTriggerDelay (self, delay)
- setPhiDelay (self, delay)
- setExtClk (self, delay)
- setTiming (self, side="AB", sequence=None, delay=0)
- setArbTiming (self, side="AB", sequence=None)
- getTiming (self, side, actual)
- setManualShutters (self, timing=None)
- setManualTiming (self, timing=None)
- getManualTiming (self)
- getSensTemp (self, scale=None, offset=None, slope=None, dec=None)
- selectOscillator (self, osc=None)
- parseReadoff (self, frames, columns)
- getSensorStatus (self)
- reportStatusSensor (self, statusbits, statusbits2)

Public Attributes

- ca
- sens_registers
- sens_subregisters

Public Attributes inherited from nsCamera.sensors.sensorBase.sensorBase

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- fpganumID
- sensfam

Static Public Attributes

- str specwarn = " and the use of the Icarus model 1 sensor"
- int minframe = 1
- int maxframe = 2
- int maxwidth = 512
- int maxheight = 1024
- int bytesperpixel = 2
- int icarustype = 1
- int fpganumID = 1
- str detect = "ICARUS_DET"
- str sensfam = "Icarus"
- str loglabel = "[Icarus1] "
- int firstframe = 1
- int lastframe = 2
- int nframes = 2
- int width = 512
- int height = 1024
- int firstrow = 0
- int lastrow = 1023
- list interlacing = [0, 0]
- int columns = 1
- bool padToFull = True

6.5.1 Detailed Description

Definition at line 27 of file icarus.py.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 __init__()

```
nsCamera.sensors.icarus.icarus.__init__ (
                self,
                ca )
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 53 of file icarus.py.

```
00053     def __init__(self, ca):
00054         self.ca = ca
00055         super(icarus, self).__init__(ca)
00056
00057         self.sens_registers = OrderedDict(
00058             {
00059                 "VRESET_WAIT_TIME": "03E",
00060                 "ICARUS_VER_SEL": "041",
00061                 "VRESET_HIGH_VALUE": "04A",
00062                 "MISC_SENSOR_CTL": "04C",
00063                 "MANUAL_SHUTTERS_MODE": "050",
00064                 "W0_INTEGRATION": "051",
00065                 "W0_INTERFRAME": "052",
00066                 "W1_INTEGRATION": "053",
00067                 "W1_INTERFRAME": "054",
00068                 "W2_INTEGRATION": "055",
00069                 "W2_INTERFRAME": "056",
00070                 "W3_INTEGRATION": "057",
00071                 "W0_INTEGRATION_B": "058",
00072                 "W0_INTERFRAME_B": "059",
00073                 "W1_INTEGRATION_B": "05A",
00074                 "W1_INTERFRAME_B": "05B",
00075                 "W2_INTEGRATION_B": "05C",
00076                 "W2_INTERFRAME_B": "05D",
00077                 "W3_INTEGRATION_B": "05E",
00078                 "TIME_ROW_DCD": "05F",
00079             }
00080         )
00081
00082         self.sens_subregisters = [
00083
00085             ("MANSHUT_MODE", "MANUAL_SHUTTERS_MODE", 0, 1, True),
00086             ("REVREAD", "CTRL_REG", 4, 1, True),
00087             ("PDBIAS_LOW", "CTRL_REG", 6, 1, True),
00088             ("ROWDCD_CTL", "CTRL_REG", 7, 1, True),
00089             ("ACCUMULATION_CTL", "MISC_SENSOR_CTL", 0, 1, True),
00090             ("HST_TST_ANRST_EN", "MISC_SENSOR_CTL", 1, 1, True),
00091             ("HST_TST_BNRST_EN", "MISC_SENSOR_CTL", 2, 1, True),
00092             ("HST_TST_ANRST_IN", "MISC_SENSOR_CTL", 3, 1, True),
00093             ("HST_TST_BNRST_IN", "MISC_SENSOR_CTL", 4, 1, True),
00094             ("HST_PXL_RST_EN", "MISC_SENSOR_CTL", 5, 1, True),
00095             ("HST_CONT_MODE", "MISC_SENSOR_CTL", 6, 1, True),
00096             ("COL_DCD_EN", "MISC_SENSOR_CTL", 7, 1, True),
00097             ("COL_READOUT_EN", "MISC_SENSOR_CTL", 8, 1, True),
00098             ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True),
00099
00102             ("STAT_W3TOPAEDGE1", "STAT_REG", 3, 1, False),
00103             ("STAT_W3TOPBEDGE1", "STAT_REG", 4, 1, False),
00104             ("STAT_HST_ALL_W_EN_DETECTED", "STAT_REG", 12, 1, False),
00105             ("PDBIAS_UNREADY", "STAT_REG2", 5, 1, False),
00106         ]
00107
00108         if self.ca.boardname == "llnl_v1":
00109             self.sens_subregisters.append(
00110                 ("VRESET_HIGH", "VRESET_HIGH_VALUE", 7, 8, True)
00111             )
00112         else:
00113             self.sens_subregisters.extend(
00114                 [
00115                     ("VRESET_HIGH", "VRESET_HIGH_VALUE", 15, 16, True),
00116                     ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True),
00117                 ]
00118             )
00119             self.sens_registers.update({"DELAY_ASSERTION_ROWDCD_EN": "04F"})
00120
```

6.5.3 Member Function Documentation

6.5.3.1 checkSensorVoltStat()

```
nsCamera.sensors.icarus.icarus.checkSensorVoltStat (
              self )
```

```
Checks register tied to sensor select jumpers to confirm match with sensor
  object

Returns:
    boolean, True if jumpers select for Icarus sensor
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 121 of file icarus.py.

```
    def checkSensorVoltStat(self):
        """
        Checks register tied to sensor select jumpers to confirm match with sensor
          object

        Returns:
            boolean, True if jumpers select for Icarus sensor
        """
        logging.debug(self.logdebug + "checkSensorVoltStat")
        err, status = self.ca.getSubregister("ICARUS_DET")
        if err:
            logging.error(self.logerr + "unable to confirm sensor status")
            return False
        if not int(status):
            logging.error(self.logerr + "Icarus sensor not detected")
            return False
        return True

```

6.5.3.2 sensorSpecific()

```
nsCamera.sensors.icarus.icarus.sensorSpecific (
              self )
```

```
Returns:
    list of tuples, (Sensor-specific register, default setting)
```

Definition at line 139 of file icarus.py.

```
    def sensorSpecific(self):
        """
        Returns:
            list of tuples, (Sensor-specific register, default setting)
        """
        icarussettings = [
            ("ICARUS_VER_SEL", "00000001"),
            ("FPA_FRAME_INITIAL", "00000001"),
            ("FPA_FRAME_FINAL", "00000002"),
            ("FPA_ROW_INITIAL", "00000000"),
            ("FPA_ROW_FINAL", "000003FF"),
            ("VRESET_WAIT_TIME", "000927C0"),
            ("HS_TIMING_DATA_BHI", "00000000"),
            ("HS_TIMING_DATA_BLO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
            ("HS_TIMING_DATA_AHI", "00000000"),
            ("HS_TIMING_DATA_ALO", "00006666"),
        ]
        if self.ca.boardname == "llnl_v1":
            icarussettings.append(
                ("VRESET_HIGH_VALUE", "000000D5")  # 3.3 V (FF = 3.96)
            )
        else:
            icarussettings.append(("VRESET_HIGH_VALUE", "0000FFFF"))
        return icarussettings


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.5.4 Member Data Documentation

6.5.4.1 bytesperpixel

```
int nsCamera.sensors.icarus.icarus.bytesperpixel = 2  [static]
```

Definition at line 36 of file icarus.py.

6.5.4.2 ca

```
nsCamera.sensors.icarus.icarus.ca
```

Definition at line 54 of file icarus.py.

6.5.4.3 columns

```
int nsCamera.sensors.icarus.icarus.columns = 1  [static]
```

Definition at line 50 of file icarus.py.

6.5.4.4 detect

```
str nsCamera.sensors.icarus.icarus.detect = "ICARUS_DET"  [static]
```

Definition at line 39 of file icarus.py.

6.5.4.5 firstframe

```
int nsCamera.sensors.icarus.icarus.firstframe = 1  [static]
```

Definition at line 42 of file icarus.py.

6.5.4.6 firstrow

```
int nsCamera.sensors.icarus.icarus.firstrow = 0  [static]
```

Definition at line 47 of file icarus.py.

6.5.4.7 fpganumID

```
int nsCamera.sensors.icarus.icarus.fpganumID = 1  [static]
```

Definition at line 38 of file icarus.py.

6.5.4.8 height

```
int nsCamera.sensors.icarus.icarus.height = 1024  [static]
```

Definition at line 46 of file icarus.py.

6.5.4.9 icarustype

```
int nsCamera.sensors.icarus.icarus.icarustype = 1  [static]
```

Definition at line 37 of file icarus.py.

6.5.4.10 interlacing

```
list nsCamera.sensors.icarus.icarus.interlacing = [0, 0]  [static]
```

Definition at line 49 of file icarus.py.

6.5.4.11 lastframe

```
int nsCamera.sensors.icarus.icarus.lastframe = 2  [static]
```

Definition at line 43 of file icarus.py.

6.5.4.12 lastrow

```
int nsCamera.sensors.icarus.icarus.lastrow = 1023  [static]
```

Definition at line 48 of file icarus.py.

6.5.4.13 loglabel

```
str nsCamera.sensors.icarus.icarus.loglabel = "[Icarus1] "  [static]
```

Definition at line 41 of file icarus.py.

6.5.4.14 maxframe

```
int nsCamera.sensors.icarus.icarus.maxframe = 2  [static]
```

Definition at line 30 of file icarus.py.

6.5.4.15 maxheight

```
int nsCamera.sensors.icarus.icarus.maxheight = 1024  [static]
```

Definition at line 35 of file icarus.py.

6.5.4.16 maxwidth

```
int nsCamera.sensors.icarus.icarus.maxwidth = 512  [static]
```

Definition at line 34 of file icarus.py.

6.5.4.17 minframe

```
int nsCamera.sensors.icarus.icarus.minframe = 1  [static]
```

Definition at line 29 of file icarus.py.

6.5.4.18 nframes

```
int nsCamera.sensors.icarus.icarus.nframes = 2  [static]
```

Definition at line 44 of file icarus.py.

6.5.4.19 padToFull

```
bool nsCamera.sensors.icarus.icarus.padToFull = True  [static]
```

Definition at line 51 of file icarus.py.

6.5.4.20 sens_registers

```
nsCamera.sensors.icarus.icarus.sens_registers
```

Definition at line 57 of file icarus.py.

6.5.4.21 sens_subregisters

```
nsCamera.sensors.icarus.icarus.sens_subregisters
```

Definition at line 82 of file icarus.py.

6.5.4.22 sensfam

```
str nsCamera.sensors.icarus.icarus.sensfam = "Icarus"  [static]
```

Definition at line 40 of file icarus.py.

6.5.4.23 specwarn

```
str nsCamera.sensors.icarus.icarus.specwarn = " and the use of the Icarus model 1 sensor"  [static]
```

Definition at line 28 of file icarus.py.

6.5.4.24 width

```
int nsCamera.sensors.icarus.icarus.width = 512  [static]
```

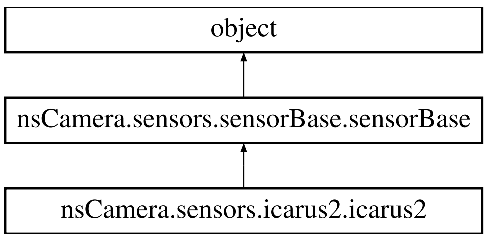
Definition at line 45 of file icarus.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/sensors/icarus.py

6.6 nsCamera.sensors.icarus2.icarus2 Class Reference

Inheritance diagram for nsCamera.sensors.icarus2.icarus2:


object
nsCamera.sensors.sensorBase.sensorBase
nsCamera.sensors.icarus2.icarus2

Public Member Functions

- __init__ (self, ca)
- sensorSpecific (self)

Public Member Functions inherited from nsCamera.sensors.sensorBase.sensorBase

- init_board_specific (self)
- checkSensorVoltStat (self)
- setInterlacing (self, ifactor)
- setHighFullWell (self, flag)
- setZeroDeadTime (self, flag)
- setTriggerDelay (self, delay)
- setPhiDelay (self, delay)
- setExtClk (self, delay)
- setTiming (self, side="AB", sequence=None, delay=0)
- setArbTiming (self, side="AB", sequence=None)
- getTiming (self, side, actual)
- setManualShutters (self, timing=None)
- setManualTiming (self, timing=None)
- getManualTiming (self)
- getSensTemp (self, scale=None, offset=None, slope=None, dec=None)
- selectOscillator (self, osc=None)
- parseReadoff (self, frames, columns)
- getSensorStatus (self)
- reportStatusSensor (self, statusbits, statusbits2)

Public Attributes

- ca
- sens_registers
- sens_subregisters

Public Attributes inherited from nsCamera.sensors.sensorBase.sensorBase

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- fpganumID
- sensfam

Static Public Attributes

- str specwarn = ""
- int minframe = 0
- int maxframe = 3
- int maxwidth = 512
- int maxheight = 1024
- int bytesperpixel = 2
- int icarustype = 0
- int fpganumID = 1
- str detect = "ICARUS_DET"
- str sensfam = "Icarus"
- str loglabel = "[Icarus2] "
- int firstframe = 0
- int lastframe = 3
- int nframes = 4
- int width = 512
- int height = 1024
- int firstrow = 0
- int lastrow = 1023
- list interlacing = [0, 0]
- int columns = 1
- bool padToFull = True

6.6.1 Detailed Description

Definition at line 24 of file icarus2.py.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 __init__()

```
nsCamera.sensors.icarus2.icarus2.__init__ (
                self,
                ca )
```

Reimplemented from nsCamera.sensors.sensorBase.sensorBase.

Definition at line 47 of file icarus2.py.

```
    def __init__(self, ca):
        self.ca = ca
        super(icarus2, self).__init__(ca)

        self.sens_registers = OrderedDict(
            {
                "VRESET_WAIT_TIME": "03E",
                "ICARUS_VER_SEL": "041",
                "MISC_SENSOR_CTL": "04C",
                "MANUAL_SHUTTERS_MODE": "050",
                "W0_INTEGRATION": "051",
                "W0_INTERFRAME": "052",
                "W1_INTEGRATION": "053",
                "W1_INTERFRAME": "054",
                "W2_INTEGRATION": "055",
                "W2_INTERFRAME": "056",
                "W3_INTEGRATION": "057",
                "W0_INTEGRATION_B": "058",
                "W0_INTERFRAME_B": "059",
                "W1_INTEGRATION_B": "05A",
                "W1_INTERFRAME_B": "05B",
                "W2_INTEGRATION_B": "05C",
                "W2_INTERFRAME_B": "05D",
                "W3_INTEGRATION_B": "05E",
                "TIME_ROW_DCD": "05F",
            }
        )

        self.sens_subregisters = [

            ("MANSHUT_MODE", "MANUAL_SHUTTERS_MODE", 0, 1, True),
            ("REVREAD", "CTRL_REG", 4, 1, True),
            ("PDBIAS_LOW", "CTRL_REG", 6, 1, True),
            ("ROWDCD_CTL", "CTRL_REG", 7, 1, True),
            ("ACCUMULATION_CTL", "MISC_SENSOR_CTL", 0, 1, True),
            ("HST_TST_ANRST_EN", "MISC_SENSOR_CTL", 1, 1, True),
            ("HST_TST_BNRST_EN", "MISC_SENSOR_CTL", 2, 1, True),
            ("HST_TST_ANRST_IN", "MISC_SENSOR_CTL", 3, 1, True),
            ("HST_TST_BNRST_IN", "MISC_SENSOR_CTL", 4, 1, True),
            ("HST_PXL_RST_EN", "MISC_SENSOR_CTL", 5, 1, True),
            ("HST_CONT_MODE", "MISC_SENSOR_CTL", 6, 1, True),
            ("COL_DCD_EN", "MISC_SENSOR_CTL", 7, 1, True),
            ("COL_READOUT_EN", "MISC_SENSOR_CTL", 8, 1, True),

            ("STAT_W3TOPAEDGE1", "STAT_REG", 3, 1, False),
            ("STAT_W3TOPBEDGE1", "STAT_REG", 4, 1, False),
            ("STAT_HST_ALL_W_EN_DETECTED", "STAT_REG", 12, 1, False),
            ("PDBIAS_UNREADY", "STAT_REG2", 5, 1, False),
        ]

        if self.ca.boardname == "llnl_v4":
            self.sens_subregisters.append(
                ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True)
            )
            self.sens_registers.update({"DELAY_ASSERTION_ROWDCD_EN": "04F"})

```

6.6.3 Member Function Documentation

6.6.3.1 sensorSpecific()

```
nsCamera.sensors.icarus2.icarus2.sensorSpecific (
              self )
```

```
Returns:
    list of tuples, (Sensor-specific register, default setting)
```

Definition at line 107 of file icarus2.py.

```
    def sensorSpecific(self):
        """
        Returns:
            list of tuples, (Sensor-specific register, default setting)
        """
        return [
            ("ICARUS_VER_SEL", "00000000"),
            ("FPA_FRAME_INITIAL", "00000000"),
            ("FPA_FRAME_FINAL", "00000003"),
            ("FPA_ROW_INITIAL", "00000000"),
            ("FPA_ROW_FINAL", "000003FF"),
            ("HS_TIMING_DATA_BHI", "00000000"),
            ("HS_TIMING_DATA_BLO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
            ("HS_TIMING_DATA_AHI", "00000000"),
            ("HS_TIMING_DATA_ALO", "00006666"),
        ]
```

```


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.6.4 Member Data Documentation

6.6.4.1 bytesperpixel

`int nsCamera.sensors.icarus2.icarus2.bytesperpixel = 2  [static]`

Definition at line 30 of file icarus2.py.

6.6.4.2 ca

`nsCamera.sensors.icarus2.icarus2.ca`

Definition at line 48 of file icarus2.py.

6.6.4.3 columns

`int nsCamera.sensors.icarus2.icarus2.columns = 1  [static]`

Definition at line 44 of file icarus2.py.

6.6.4.4 detect

`str nsCamera.sensors.icarus2.icarus2.detect = "ICARUS_DET"  [static]`

Definition at line 33 of file icarus2.py.

6.6.4.5 firstframe

`int nsCamera.sensors.icarus2.icarus2.firstframe = 0  [static]`

Definition at line 36 of file icarus2.py.

6.6.4.6 firstrow

```
int nsCamera.sensors.icarus2.icarus2.firstrow = 0  [static]
```

Definition at line 41 of file icarus2.py.

6.6.4.7 fpganumID

```
int nsCamera.sensors.icarus2.icarus2.fpganumID = 1  [static]
```

Definition at line 32 of file icarus2.py.

6.6.4.8 height

```
int nsCamera.sensors.icarus2.icarus2.height = 1024  [static]
```

Definition at line 40 of file icarus2.py.

6.6.4.9 icarustype

```
int nsCamera.sensors.icarus2.icarus2.icarustype = 0  [static]
```

Definition at line 31 of file icarus2.py.

6.6.4.10 interlacing

```
list nsCamera.sensors.icarus2.icarus2.interlacing = [0, 0]  [static]
```

Definition at line 43 of file icarus2.py.

6.6.4.11 lastframe

```
int nsCamera.sensors.icarus2.icarus2.lastframe = 3  [static]
```

Definition at line 37 of file icarus2.py.

6.6.4.12 lastrow

```
int nsCamera.sensors.icarus2.icarus2.lastrow = 1023  [static]
```

Definition at line 42 of file icarus2.py.

6.6.4.13 loglabel

```
str nsCamera.sensors.icarus2.icarus2.loglabel = "[Icarus2] "  [static]
```

Definition at line 35 of file icarus2.py.

6.6.4.14 maxframe

```
int nsCamera.sensors.icarus2.icarus2.maxframe = 3  [static]
```

Definition at line 27 of file icarus2.py.

6.6.4.15 maxheight

```
int nsCamera.sensors.icarus2.icarus2.maxheight = 1024  [static]
```

Definition at line 29 of file icarus2.py.

6.6.4.16 maxwidth

```
int nsCamera.sensors.icarus2.icarus2.maxwidth = 512  [static]
```

Definition at line 28 of file icarus2.py.

6.6.4.17 minframe

```
int nsCamera.sensors.icarus2.icarus2.minframe = 0  [static]
```

Definition at line 26 of file icarus2.py.

6.6.4.18 nframes

```
int nsCamera.sensors.icarus2.icarus2.nframes = 4  [static]
```

Definition at line 38 of file icarus2.py.

6.6.4.19 padToFull

```
bool nsCamera.sensors.icarus2.icarus2.padToFull = True  [static]
```

Definition at line 45 of file icarus2.py.

6.6.4.20 sens_registers

```
nsCamera.sensors.icarus2.icarus2.sens_registers
```

Definition at line 51 of file icarus2.py.

6.6.4.21 sens_subregisters

```
nsCamera.sensors.icarus2.icarus2.sens_subregisters
```

Definition at line 75 of file icarus2.py.

6.6.4.22 sensfam

```
str nsCamera.sensors.icarus2.icarus2.sensfam = "Icarus"  [static]
```

Definition at line 34 of file icarus2.py.

6.6.4.23 specwarn

```
str nsCamera.sensors.icarus2.icarus2.specwarn = ""  [static]
```

Definition at line 25 of file icarus2.py.

6.6.4.24 width

```
int nsCamera.sensors.icarus2.icarus2.width = 512  [static]
```

Definition at line 39 of file icarus2.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/sensors/icarus2.py

6.7 nsCamera.boards.LLNL_v1.llnl_v1 Class Reference

Public Member Functions

- __init__ (self, camassem)
- initBoard (self)
- initPots (self)
- latchPots (self)
- initSensor (self)
- configADCs (self)
- softReboot (self)
- disarm (self)
- startCapture (self, mode="Hardware")
- readSRAM (self)
- waitForSRAM (self, timeout)
- getTimer (self)
- resetTimer (self)
- enableLED (self, status)
- setLED (self, LED, status)
- setPowerSave (self, status)
- setPPER (self, pollperiod)
- getTemp (self, scale=None, offset=None, slope=None)
- getPressure (self, offset, sensitivity, units)
- clearStatus (self)
- checkStatus (self)
- checkStatus2 (self)
- reportStatus (self)
- reportEdgeDetects (self)
- dumpStatus (self)

Public Attributes

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- VREF
- ADC5_mult
- ADC5_bipolar
- rs422_baud
- rs422_cmd_wait
- icarus_subreg_aliases
- icarus_monitor_controls
- daedalus_subreg_aliases
- daedalus_monitor_controls
- subreglist

Static Public Attributes

- registers
- list subregisters

6.7.1 Detailed Description

```
Livermore LLNL v1.0 board

Compatible communication protocols: RS422, GigE
Compatible sensors: icarus, icarus2, daedalus
```

Definition at line 28 of file LLNL_v1.py.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 __init__()

```
nsCamera.boards.LLNL_v1.llnl_v1.__init__ (
                self,
                camassem )
```

Definition at line 192 of file LLNL_v1.py.

```
    def __init__(self, camassem):
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[LLNL_v1] "
        self.logerr = self.ca.logerrbase + "[LLNL_v1] "
        self.logwarn = self.ca.logwarnbase + "[LLNL_v1] "
        self.loginfo = self.ca.loginfobase + "[LLNL_v1] "
        self.logdebug = self.ca.logdebugbase + "[LLNL_v1] "
        logging.info(self.loginfo + "initializing board object")
        self.VREF = 2.5  # default
        self.ADC5_mult = 2  # i.e., monmax = 2 * VREF
        # False => monitor range runs 0 to monmax, True => +/- monmax
        self.ADC5_bipolar = True
        self.rs422_baud = 921600
        self.rs422_cmd_wait = 0.3

        fpgaNum_pkt = Packet(cmd="1", addr=self.registers["FPGA_NUM"])
        fpgaRev_pkt = Packet(cmd="1", addr=self.registers["FPGA_REV"])

        _, _ = self.ca.sendCMD(fpgaNum_pkt)  # dummy duplicate call
        err, rval = self.ca.sendCMD(fpgaNum_pkt)
        self.ca.FPGANum = rval[8:16]

        err, rval = self.ca.sendCMD(fpgaRev_pkt)
        self.ca.FPGAVersion = rval[8:16]

        # map channels to signal names for abstraction at the camera assembler level;
        #   each requires a corresponding entry in 'subregisters'

        self.icarus_subreg_aliases = OrderedDict(
            {
                "COL_BOT_IBIAS_IN": "POT1",
                "HST_A_PDELAY": "POT2",
                "HST_B_NDELAY": "POT3",
                "HST_RO_IBIAS": "POT4",
                "HST_OSC_VREF_IN": "POT5",
                "HST_B_PDELAY": "POT6",
                "HST_OSC_CTL": "POT7",
                "HST_A_NDELAY": "POT8",
                "COL_TOP_IBIAS_IN": "POT9",
                "HST_OSC_R_BIAS": "POT10",
                "VAB": "POT11",
                "HST_RO_NC_IBIAS": "POT12",
                "VRST": "POT13",
                "MON_HST_A_PDELAY": "MON_CH2",
                "MON_HST_B_NDELAY": "MON_CH3",
                "MON_HST_RO_IBIAS": "MON_CH4",
                "MON_HST_OSC_VREF_IN": "MON_CH5",
                "MON_HST_B_PDELAY": "MON_CH6",
                "MON_HST_OSC_CTL": "MON_CH7",
                "MON_HST_A_NDELAY": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.icarus_monitor_controls = OrderedDict(
            {
                "MON_CH2": "POT2",
                "MON_CH3": "POT3",
                "MON_CH4": "POT4",
                "MON_CH5": "POT5",
                "MON_CH6": "POT6",
                "MON_CH7": "POT7",
                "MON_CH8": "POT8",
                # Note: VRST is not measured across the pot; it will read a voltage
                #   approximately 1 Volt lower than pot13's actual output
                "MON_VRST": "POT13",
            }
        )

        self.daedalus_subreg_aliases = OrderedDict(
            {
                "HST_OSC_CTL": "POT4",
                "HST_RO_NC_IBIAS": "POT5",
                "HST_OSC_VREF_IN": "POT6",
                "VAB": "POT11",
                "MON_TSENSEOUT": "MON_CH2",
                "MON_BGREF": "MON_CH3",
                "MON_HST_OSC_CTL": "MON_CH4",
                "MON_HST_RO_NC_IBIAS": "MON_CH5",
                "MON_HST_OSC_VREF_IN": "MON_CH6",
                "MON_COL_TST_IN": "MON_CH7",
                "MON_HST_OSC_PBIAS_PAD": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.daedalus_monitor_controls = OrderedDict(
            {
                "MON_CH4": "POT4",
                "MON_CH5": "POT5",
                "MON_CH6": "POT6",
                # Note: VRST is not measured across the pot; it will read a voltage
                #   lower than pot13's actual output
                "MON_VRST": "POT13",
            }
        )

        self.subreglist = []
        for s in self.subregisters:
            self.subreglist.append(s[0].upper())
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)

        # set voltage ranges for all pots
        for n in range(1, 13):
            potname = "POT" + str(n)
            potobj = getattr(self, potname)
            potobj.minV = 0
            potobj.maxV = 3.3
            # resolution is approximately .0129 V / LSB
            potobj.resolution = (1.0 * potobj.maxV - potobj.minV) / potobj.max_value
        self.POT13.minV = 0
        self.POT13.maxV = 3.96
        # POT13 resolution is approximately .0155 V / LSB
        self.POT13.resolution = (
            1.0 * self.POT13.maxV - self.POT13.minV
        ) / self.POT13.max_value

```

6.7.3 Member Function Documentation

6.7.3.1 checkStatus()

```
nsCamera.boards.LLNL_v1.llnl_v1.checkStatus (
              self )
```

```
Check status register, convert to reverse-order bit stream (i.e., bit 0 is
  statusbits[0])

Returns:
    bit string (no ’0b’) in reversed order
```

Definition at line 729 of file LLNL_v1.py.

```
    def checkStatus(self):
        """
        Check status register, convert to reverse-order bit stream (i.e., bit 0 is
          statusbits[0])

        Returns:
            bit string (no ’0b’) in reversed order
        """
        logging.debug(self.logdebug + "checkStatus")
        err, rval = self.ca.getRegister("STAT_REG")
        if not rval:
            logging.error(
                self.logerr + "Unable to check status register (zeroes returned)"
            )
            rval = "0"
        rvalbits = bin(int(rval, 16))[2:].zfill(32)
        statusbits = rvalbits[::-1]
        return statusbits  # TODO: add error handling

```

6.7.3.2 checkStatus2()

```
nsCamera.boards.LLNL_v1.llnl_v1.checkStatus2 (
              self )
```

```
Check second status register, convert to reverse-order bit stream (i.e., bit 0
  is statusbits[0])

Returns: bit string (no ’0b’) in reversed order
```

Definition at line 748 of file LLNL_v1.py.

```
    def checkStatus2(self):
        """
        Check second status register, convert to reverse-order bit stream (i.e., bit 0
          is statusbits[0])

        Returns: bit string (no ’0b’) in reversed order
        """
        logging.debug(self.logdebug + "checkStatus2")
        err, rval = self.ca.getRegister("STAT_REG2")
        if not rval:
            logging.error(
                self.logerr + "Unable to check status register 2 (zeroes returned)"
            )
            rval = "0"
        rvalbits = bin(int(rval, 16))[2:].zfill(5)
        statusbits = rvalbits[::-1]
        return statusbits  # TODO: add error handling

```

6.7.3.3 clearStatus()

```
nsCamera.boards.LLNL_v1.llnl_v1.clearStatus (
              self )
```

```
Check status registers to clear them

Returns:
    error string
```

Definition at line 714 of file LLNL_v1.py.

```
    def clearStatus(self):
        """
        Check status registers to clear them

        Returns:
            error string
        """
        logging.debug(self.logdebug + "clearStatus")
        err1, rval = self.ca.getRegister("STAT_REG_SRC")
        err2, rval = self.ca.getRegister("STAT_REG2_SRC")
        err = err1 + err2
        if err:
            logging.error(self.logerr + "clearStatus failed")
        return err

```

6.7.3.4 configADCs()

```
nsCamera.boards.LLNL_v1.llnl_v1.configADCs (
              self )
```

```
Sets default ADC configuration (does not latch settings)

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 437 of file LLNL_v1.py.

```
    def configADCs(self):
        """
        Sets default ADC configuration (does not latch settings)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "configADCs")

        control_messages = [
            # just in case ADC_RESET was set (pull all ADCs out # of reset)
            ("ADC_RESET", "00000000"),
            # workaround for uncertain behavior after previous readoff
            ("ADC1_CONFIG_DATA", "FFFFFFFF"),
            ("ADC2_CONFIG_DATA", "FFFFFFFF"),
            ("ADC3_CONFIG_DATA", "FFFFFFFF"),
            ("ADC4_CONFIG_DATA", "FFFFFFFF"),
            ("ADC_CTL", "FFFFFFFF"),
            ("ADC1_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC2_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC3_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC4_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC5_CONFIG_DATA", "81A883FF"),  # int Vref 2.50V
        ]
        return self.ca.submitMessages(control_messages, " configADCs: ")

```

6.7.3.5 disarm()

```
nsCamera.boards.LLNL_v1.llnl_v1.disarm (
              self )
```

```
Takes camera out of trigger wait state. Has no effect if camera is not already
  in wait state.

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 475 of file LLNL_v1.py.

```
    def disarm(self):
        """
        Takes camera out of trigger wait state. Has no effect if camera is not already
          in wait state.

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "disarm")
        self.ca.clearStatus()
        self.ca.armed = False
        control_messages = [
            ("HW_TRIG_EN", "0"),
            ("SW_TRIG_EN", "0"),
        ]
        return self.ca.submitMessages(control_messages, " disarm: ")

```

6.7.3.6 dumpStatus()

```
nsCamera.boards.LLNL_v1.llnl_v1.dumpStatus (
              self )
```

```
Create dictionary of status values, DAC settings, monitor values, and register
  values

WARNING: the behavior of self-resetting subregisters may be difficult to predict
  and may generate contradictory results

Returns:
    dictionary of system diagnostic values
```

Definition at line 824 of file LLNL_v1.py.

```
    def dumpStatus(self):
        """
        Create dictionary of status values, DAC settings, monitor values, and register
          values

        WARNING: the behavior of self-resetting subregisters may be difficult to predict
          and may generate contradictory results

        Returns:
            dictionary of system diagnostic values
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()
        temp = self.ca.getTemp()

        statDict = OrderedDict(
            {
                "Temperature reading": "{0:1.2f}".format(temp) + " C",
                "Sensor read complete": str(statusbits[0]),
                "Coarse trigger detected": str(statusbits[1]),
                "Fine trigger detected": str(statusbits[2]),
                "Sensor readout in progress": str(statusbits[5]),
                "Sensor readout complete": str(statusbits[6]),
                "SRAM readout started": str(statusbits[7]),
                "SRAM readout complete": str(statusbits[8]),
                "High-speed timing configured": str(statusbits[9]),
                "All ADCs configured": str(statusbits[10]),
                "All pots configured": str(statusbits[11]),
                "HST_All_W_En detected": str(statusbits[12]),
                "Timer has reset": str(statusbits[13]),
                "Camera is Armed": str(statusbits[14]),
                "FPA_IF_TO": str(statusbits2[0]),
                "SRAM_RO_TO": str(statusbits2[1]),
                "PixelRd Timeout Error": str(statusbits2[2]),
                "UART_TX_TO_RST": str(statusbits2[3]),
                "UART_RX_TO_RST": str(statusbits2[4]),
            }
        )

        POTDict = OrderedDict()
        MonDict = OrderedDict()
        for entry in self.subreg_aliases:
            if self.subreg_aliases[entry][0] == "P":
                val = str(round(self.ca.getPotV(entry), 3)) + " V"
                POTDict["POT_" + entry] = val
            else:
                val = str(round(self.ca.getMonV(entry), 3)) + " V"
                MonDict[entry] = val

        regDict = OrderedDict()
        for key in self.registers.keys():
            err, rval = self.ca.getRegister(key)
            regDict[key] = rval

        dumpDict = OrderedDict()
        for x in [statDict, MonDict, POTDict, regDict]:
            dumpDict.update(x)
        return dumpDict


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.7.3.7 enableLED()

```
nsCamera.boards.LLNL_v1.llnl_v1.enableLED (
              self,
              status )
```

```
Enable/disable on-board LEDs

Args:
    status: 0 for disabled, 1 for enabled

Returns:
    tuple: (error string, response string from setSubregister()
```

Definition at line 603 of file LLNL_v1.py.

```
    def enableLED(self, status):
        """
        Enable/disable on-board LEDs

        Args:
            status: 0 for disabled, 1 for enabled

        Returns:
            tuple: (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "enableLED")
        if status:
            status = 1
        return self.ca.setSubregister("LED_EN", str(status))

```

6.7.3.8 getPressure()

```
nsCamera.boards.LLNL_v1.llnl_v1.getPressure (
                self,
                offset,
                sensitivity,
                units )
```

```
Read pressure sensor

Currently unimplemented

Returns:
    0 as float
```

Definition at line 700 of file LLNL_v1.py.

```
    def getPressure(self, offset, sensitivity, units):
        """
        Read pressure sensor

        Currently unimplemented

        Returns:
            0 as float
        """
        logging.warning(
            "WARNING: [LLNL_v1] 'getPressure' is not implemented on the LLNLv1 board"
        )
        return 0.0

```

6.7.3.9 getTemp()

```
nsCamera.boards.LLNL_v1.llnl_v1.getTemp (
                self,
                scale = None,
                offset = None,
                slope = None )
```

```
Read temperature sensor
Args:
    scale: temperature scale to report (defaults to C, options are F and K)
    offset: unused
    slope: unused
Returns:
    temperature as float on given scale
```

Definition at line 672 of file LLNL_v1.py.

```
    def getTemp(self, scale=None, offset=None, slope=None):
        """
        Read temperature sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)
            offset: unused
            slope: unused
        Returns:
            temperature as float on given scale
        """
        logging.debug(self.logdebug + "getTemp: scale = " + str(scale))
        err, rval = self.ca.getRegister("TEMP_SENSE_DATA")
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0

        ctemp = int(rval[-3:], 16) / 16.0
        if scale == "K":
            temp = ctemp + 273.15
        elif scale == "F":
            temp = 1.8 * ctemp + 32
        else:
            temp = ctemp
        return temp

```

6.7.3.10 getTimer()

```
nsCamera.boards.LLNL_v1.llnl_v1.getTimer (
                self )
```

```
Read value of on-board timer

Returns:
    timer value as integer
```

Definition at line 575 of file LLNL_v1.py.

```
    def getTimer(self):
        """
        Read value of on-board timer

        Returns:
            timer value as integer
        """
        logging.info(self.loginfo + "getTimer")
        err, rval = self.ca.getRegister("TIMER_VALUE")
        if err:
            logging.error(
                self.logerr + "unable to retrieve timer information (getTimer), "
                'returning "0" '
            )
            return 0
        return int(rval, 16)

```

6.7.3.11 initBoard()

```
nsCamera.boards.LLNL_v1.llnl_v1.initBoard (
                self )
```

```
Register and reset board, set up firmware for sensor

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 315 of file LLNL_v1.py.

```
    def initBoard(self):
        """
        Register and reset board, set up firmware for sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initBoard LLNLv1")
        control_messages = [("LED_EN", "1")]

        self.clearStatus()
        self.configADCs()

        err, resp = self.ca.getSubregister("ADC5_VREF3")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_VREF3'")
        if int(resp, 2):  # check to see if Vref is 3 or 2.5 volts
            vrefmax = 3.0
        else:
            vrefmax = 2.5
        err, resp = self.ca.getSubregister("ADC5_VREF")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_VREF'")
        self.VREF = vrefmax * int(resp, 2) / 1024.0
        err, multmask = self.ca.getSubregister("ADC5_MULT")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_MULT'")
        if multmask[0] and multmask[1] and multmask[3] and multmask[5]:
            self.ADC5_mult = 2
        elif not (multmask[0] or multmask[1] or multmask[3] or multmask[5]):
            self.ADC5_mult = 4
        else:
            logging.error(self.logerr + "inconsistent mode settings on ADC5")
        return self.ca.submitMessages(control_messages, " initBoard: ")

```

6.7.3.12 initPots()

```
nsCamera.boards.LLNL_v1.llnl_v1.initPots (
                self )
```

```
Configure default pot settings before image acquisition

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 350 of file LLNL_v1.py.

```
    def initPots(self):
        """
        Configure default pot settings before image acquisition

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initPots")
        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            err0, _ = self.ca.setPot("HST_A_PDELAY", 0, errflag=True)
            err1, _ = self.ca.setPotV("HST_B_NDELAY", 3.3, errflag=True)
            err2, _ = self.ca.setPotV("HST_RO_IBIAS", 2.5, tune=True, errflag=True)
            err3, _ = self.ca.setPotV("HST_OSC_VREF_IN", 2.9, tune=True, errflag=True)
            err4, _ = self.ca.setPot("HST_B_PDELAY", 0, errflag=True)
            err5, _ = self.ca.setPotV("HST_OSC_CTL", 1.45, tune=True, errflag=True)
            err6, _ = self.ca.setPotV("HST_A_NDELAY", 3.3, errflag=True)
            err7, _ = self.ca.setPotV("VAB", 0.5, errflag=True)
            err8, _ = self.ca.setPotV("HST_RO_NC_IBIAS", 2.5, errflag=True)
            err9, _ = self.ca.setPotV("VRST", 0.3, tune=True, errflag=True)
            err = err0 + err1 + err2 + err3 + err4 + err5 + err6 + err7 + err8 + err9
        else:  # Daedalus
            err0, _ = self.ca.setPotV("HST_OSC_CTL", 1.0, tune=True, errflag=True)
            err1, _ = self.ca.setPotV("HST_RO_NC_IBIAS", 1.0, errflag=True)
            err2, _ = self.ca.setPotV("HST_OSC_VREF_IN", 1.0, tune=True, errflag=True)
            err3, _ = self.ca.setPotV("VAB", 0.5, errflag=True)
            err = err0 + err1 + err2 + err3
        return err, ""

```

6.7.3.13 initSensor()

```
nsCamera.boards.LLNL_v1.llnl_v1.initSensor (
                self )
```

```
Register sensor, set default timing settings

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 404 of file LLNL_v1.py.

```
    def initSensor(self):
        """
        Register sensor, set default timing settings

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initSensor")
        if int(self.ca.FPGANum[7]) != self.ca.sensor.fpganumID:
            logging.error(
                self.logerr + "unable to confirm sensor compatibility with FPGA"
            )
        self.registers.update(self.ca.sensor.sens_registers)
        self.subregisters.extend(self.ca.sensor.sens_subregisters)
        for s in self.ca.sensor.sens_subregisters:
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)
            self.subreglist.append(s[0])
        self.ca.checkSensorVoltStat()
        control_messages = self.ca.sensorSpecific() + [
            # ring w/caps=01, relax=00, ring w/o caps = 02
            ("OSC_SELECT", "00"),
            ("FPGA_DIVCLK_EN_ADDR", "00000001"),  # TODO Make this a subregister
        ]
        return self.ca.submitMessages(control_messages, " initSensor: ")

```

6.7.3.14 latchPots()

```
nsCamera.boards.LLNL_v1.llnl_v1.latchPots (
                self )
```

```
Latch pot settings into sensor

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 378 of file LLNL_v1.py.

```
    def latchPots(self):
        """
        Latch pot settings into sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "latchPots")

        control_messages = [
            ("POT_CTL", "00000003"),  # latches register settings for pot 1
            ("POT_CTL", "00000005"),
            ("POT_CTL", "00000007"),
            ("POT_CTL", "00000009"),
            ("POT_CTL", "0000000B"),
            ("POT_CTL", "0000000D"),
            ("POT_CTL", "0000000F"),
            ("POT_CTL", "00000011"),
            ("POT_CTL", "00000013"),
            ("POT_CTL", "00000015"),
            ("POT_CTL", "00000017"),
            ("POT_CTL", "00000019"),
            ("POT_CTL", "0000001B"),
        ]
        return self.ca.submitMessages(control_messages, " latchPots: ")

```

6.7.3.15 readSRAM()

```
nsCamera.boards.LLNL_v1.llnl_v1.readSRAM (
              self )
```

```
Start readoff of SRAM

Returns:
    tuple (error string, response string from register set)
```

Definition at line 529 of file LLNL_v1.py.

```
    def readSRAM(self):
        """
        Start readoff of SRAM

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "readSRAM")
        control_messages = [("READ_SRAM", "1")]
        return self.ca.submitMessages(control_messages, " readSRAM: ")

```

6.7.3.16 reportEdgeDetects()

```
nsCamera.boards.LLNL_v1.llnl_v1.reportEdgeDetects (
              self )
```

```
Unimplemented
```

Definition at line 815 of file LLNL_v1.py.

```
    def reportEdgeDetects(self):
        """
        Unimplemented
        """
        logging.warning(
            self.logwarn + "'reportEdgeDetects' is not implemented on the LLNLv1 "
            "board "
        )

```

6.7.3.17 reportStatus()

```
nsCamera.boards.LLNL_v1.llnl_v1.reportStatus (
                self )
```

```
Check contents of status register, print relevant messages
```

Definition at line 766 of file LLNL_v1.py.

```
    def reportStatus(self):
        """
        Check contents of status register, print relevant messages
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()
        logging.info(self.loginfo + "Status report:")
        if int(statusbits[0]):
            print(self.loginfo + "Sensor read complete")
        if int(statusbits[1]):
            print(self.loginfo + "Coarse trigger detected")
        if int(statusbits[2]):
            print(self.loginfo + "Fine trigger detected")
        if int(statusbits[5]):
            print(self.loginfo + "Sensor readout in progress")
        if int(statusbits[6]):
            print(self.loginfo + "Sensor readout complete")
        if int(statusbits[7]):
            print(self.loginfo + "SRAM readout started")
        if int(statusbits[8]):
            print(self.loginfo + "SRAM readout complete")
        if int(statusbits[9]):
            print(self.loginfo + "High-speed timing configuration started")
        if int(statusbits[10]):
            print(self.loginfo + "All ADCs configured")
        if int(statusbits[11]):
            print(self.loginfo + "All pots configured")
        if int(statusbits[13]):
            print(self.loginfo + "Timer has reset")
        if int(statusbits[14]):
            print(self.loginfo + "Camera is Armed")
        self.ca.sensor.reportStatusSensor(statusbits, statusbits2)
        temp = int(statusbits[27:15:-1], 2) / 16.0
        logging.info(
            self.loginfo + "Temperature reading: " + "{0:1.2f}".format(temp) + " C"
        )
        # press = int(statusbits[:27:-1], 2)
        # logging.info(self.loginfo + "Pressure reading: " + "{0:1.2f}".format(press))
        if int(statusbits2[0]):
            print(self.loginfo + "FPA_IF_TO")
        if int(statusbits2[1]):
            print(self.loginfo + "SRAM_RO_TO")
        if int(statusbits2[2]):
            print(self.loginfo + "PixelRd Timeout Error")
        if int(statusbits2[3]):
            print(self.loginfo + "UART_TX_TO_RST")
        if int(statusbits2[4]):
            print(self.loginfo + "UART_RX_TO_RST")

```

6.7.3.18 resetTimer()

```
nsCamera.boards.LLNL_v1.llnl_v1.resetTimer (
              self )
```

```
Reset on-board timer

Returns:
    tuple (error string, response string from register set)
```

Definition at line 592 of file LLNL_v1.py.

```
    def resetTimer(self):
        """
        Reset on-board timer

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "resetTimer")
        control_messages = [("RESET_TIMER", "1"), ("RESET_TIMER", "0")]
        return self.ca.submitMessages(control_messages, " resetTimer: ")
```

6.7.3.19 setLED()

```
nsCamera.boards.LLNL_v1.llnl_v1.setLED (
              self,
              LED,
              status )
```

```
Illuminate on-board LED

Args:
    LED: LED number (1-8)
    status: 0 is off, 1 is on

Returns:
    tuple: (error string, response string from setSubregister()
```

Definition at line 618 of file LLNL_v1.py.

```
    def setLED(self, LED, status):
        """
        Illuminate on-board LED

        Args:
            LED: LED number (1-8)
            status: 0 is off, 1 is on

        Returns:
            tuple: (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "setLED")
        key = "LED" + str(LED)
        return self.ca.setSubregister(key, str(status))
```

6.7.3.20 setPowerSave()

```
nsCamera.boards.LLNL_v1.llnl_v1.setPowerSave (
              self,
              status )
```

```
Select powersave option

Args:
    status: setting for powersave option (1 is enabled)

Returns:
    tuple (error string, response string from setSubregister()
```

Definition at line 633 of file LLNL_v1.py.

```
    def setPowerSave(self, status):
        """
        Select powersave option

        Args:
            status: setting for powersave option (1 is enabled)

        Returns:
            tuple (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "setPowerSave")
        if status:
            status = 1
        return self.ca.setSubregister("POWERSAVE", str(status))

```

6.7.3.21 setPPER()

```
nsCamera.boards.LLNL_v1.llnl_v1.setPPER (
              self,
              pollperiod )
```

```
Set polling period for ADCs.
Args:
    pollperiod: milliseconds, between 1 and 255, defaults to 50

Returns:
    tuple (error string, response string from setSubregister OR invalid time
      setting string)
```

Definition at line 648 of file LLNL_v1.py.

```
    def setPPER(self, pollperiod):
        """
        Set polling period for ADCs.
        Args:
            pollperiod: milliseconds, between 1 and 255, defaults to 50

        Returns:
            tuple (error string, response string from setSubregister OR invalid time
              setting string)
        """
        logging.debug(self.logdebug + "setPPER: time = " + str(pollperiod))
        if pollperiod is None:
            pollperiod = 50
        if not isinstance(pollperiod, int) or pollperiod < 1 or pollperiod > 255:
            err = (
                self.logerr + "invalid poll period submitted. Setting remains "
                "unchanged. "
            )
            logging.error(err)
            return err, str(pollperiod)
        else:
            binset = bin(pollperiod)[2:].zfill(8)
            return self.ca.setSubregister("PPER", binset)

```

6.7.3.22 softReboot()

```
nsCamera.boards.LLNL_v1.llnl_v1.softReboot (
              self )
```

```
Perform software reboot of board. WARNING: board reboot will likely prevent
  correct communication reponses and therefore will generate an error message

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 463 of file LLNL_v1.py.

```
    def softReboot(self):
        """
        Perform software reboot of board. WARNING: board reboot will likely prevent
          correct communication reponses and therefore will generate an error message

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "reboot")
        control_messages = [("RESET", "1")]
        return self.ca.submitMessages(control_messages, " disarm: ")

```

6.7.3.23 startCapture()

```
nsCamera.boards.LLNL_v1.llnl_v1.startCapture (
              self,
              mode = "Hardware" )
```

```
Selects trigger mode and enables board for image capture

Args:
    mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
      insensitive)

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 492 of file LLNL_v1.py.

```
    def startCapture(self, mode="Hardware"):
        """
        Selects trigger mode and enables board for image capture

        Args:
            mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
              insensitive)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "startCapture")
        if self.ca.sensmanual:
            timingReg = "MANSHUT_MODE"
        else:
            timingReg = "HST_MODE"

        if mode.upper()[0] == "S":  # SOFTWARE
            trigmess = [
                ("HW_TRIG_EN", "0"),
                ("SW_TRIG_EN", "1"),
                ("SW_TRIG_START", "1"),
            ]
        else:  # HARDWARE
            trigmess = [
                ("SW_TRIG_EN", "0"),
                ("HW_TRIG_EN", "1"),
            ]

        control_messages = [
            ("ADC_CTL", "0000001F"),  # configure all ADCs
            (timingReg, "1"),
        ]

        control_messages.extend(trigmess)
        return self.ca.submitMessages(control_messages, " startCapture: ")

```

6.7.3.24 waitForSRAM()

```
nsCamera.boards.LLNL_v1.llnl_v1.waitForSRAM (
                self,
                timeout )
```

```
Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
  timeout = None or zero means wait indefinitely

Args:
    timeout - time in seconds before readoff proceeds automatically without
      waiting for SRAM_READY flag

Returns:
    error string
```

Definition at line 540 of file LLNL_v1.py.

```
    def waitForSRAM(self, timeout):
        """
        Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
          timeout = None or zero means wait indefinitely

        Args:
            timeout - time in seconds before readoff proceeds automatically without
              waiting for SRAM_READY flag

        Returns:
            error string
        """
        logging.info(self.loginfo + "waitForSRAM")
        waiting = True
        starttime = time.time()
        err = ""
        while waiting:
            err, status = self.ca.getSubregister("SRAM_READY")
            if err:
                logging.error(
                    self.logerr + "error in register read: " + err + " (waitForSRAM)"
                )
            if int(status):
                waiting = False
                logging.info(self.loginfo + "SRAM ready")
            if self.ca.abort:
                waiting = False
                logging.info(self.loginfo + "readoff aborted by user")
                self.ca.abort = False
            if timeout and time.time() - starttime > timeout:
                err += self.logerr + "SRAM timeout; proceeding with download attempt"
                logging.error(err)
                return err
        return err

```

6.7.4 Member Data Documentation

6.7.4.1 ADC5_bipolar

`nsCamera.boards.LLNL_v1.llnl_v1.ADC5_bipolar`

Definition at line 203 of file LLNL_v1.py.

6.7.4.2 ADC5_mult

`nsCamera.boards.LLNL_v1.llnl_v1.ADC5_mult`

Definition at line 201 of file LLNL_v1.py.

6.7.4.3 ca

`nsCamera.boards.LLNL_v1.llnl_v1.ca`

Definition at line 193 of file LLNL_v1.py.

6.7.4.4 daedalus_monitor_controls

`nsCamera.boards.LLNL_v1.llnl_v1.daedalus_monitor_controls`

Definition at line 276 of file LLNL_v1.py.

6.7.4.5 daedalus_subreg_aliases

`nsCamera.boards.LLNL_v1.llnl_v1.daedalus_subreg_aliases`

Definition at line 260 of file LLNL_v1.py.

6.7.4.6 icarus_monitor_controls

`nsCamera.boards.LLNL_v1.llnl_v1.icarus_monitor_controls`

Definition at line 245 of file LLNL_v1.py.

6.7.4.7 icarus_subreg_aliases

`nsCamera.boards.LLNL_v1.llnl_v1.icarus_subreg_aliases`

Definition at line 220 of file LLNL_v1.py.

6.7.4.8 logcrit

```
nsCamera.boards.LLNL_v1.llnl_v1.logcrit
```

Definition at line 194 of file LLNL_v1.py.

6.7.4.9 logdebug

```
nsCamera.boards.LLNL_v1.llnl_v1.logdebug
```

Definition at line 198 of file LLNL_v1.py.

6.7.4.10 logerr

```
nsCamera.boards.LLNL_v1.llnl_v1.logerr
```

Definition at line 195 of file LLNL_v1.py.

6.7.4.11 loginfo

```
nsCamera.boards.LLNL_v1.llnl_v1.loginfo
```

Definition at line 197 of file LLNL_v1.py.

6.7.4.12 logwarn

```
nsCamera.boards.LLNL_v1.llnl_v1.logwarn
```

Definition at line 196 of file LLNL_v1.py.

6.7.4.13 registers

```
nsCamera.boards.LLNL_v1.llnl_v1.registers  [static]
```

Definition at line 37 of file LLNL_v1.py.

6.7.4.14 rs422_baud

```
nsCamera.boards.LLNL_v1.llnl_v1.rs422_baud
```

Definition at line 204 of file LLNL_v1.py.

6.7.4.15 rs422_cmd_wait

`nsCamera.boards.LLNL_v1.llnl_v1.rs422_cmd_wait`

Definition at line 205 of file LLNL_v1.py.

6.7.4.16 subregisters

`list nsCamera.boards.LLNL_v1.llnl_v1.subregisters [static]`

Definition at line 98 of file LLNL_v1.py.

6.7.4.17 subreglist

`nsCamera.boards.LLNL_v1.llnl_v1.subreglist`

Definition at line 287 of file LLNL_v1.py.

6.7.4.18 VREF

`nsCamera.boards.LLNL_v1.llnl_v1.VREF`

Definition at line 200 of file LLNL_v1.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/boards/LLNL_v1.py

6.8 nsCamera.boards.LLNL_v4.llnl_v4 Class Reference

Public Member Functions

- __init__ (self, camassem)
- initBoard (self)
- initPots (self)
- latchPots (self)
- initSensor (self)
- configADCs (self)
- softReboot (self)
- disarm (self)
- startCapture (self, mode="Hardware")
- readSRAM (self)
- waitForSRAM (self, timeout)
- getTimer (self)

- resetTimer (self)
- enableLED (self, status)
- setLED (self, LED, status)
- setPowerSave (self, status)
- setPPER (self, pollperiod)
- getTemp (self, scale=None)
- getPressure (self, offset=None, sensitivity=None, units=None)
- clearStatus (self)
- checkStatus (self)
- checkStatus2 (self)
- reportStatus (self)
- reportEdgeDetects (self)
- dumpStatus (self)

Public Attributes

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- VREF
- ADC5_mult
- ADC5_bipolar
- rs422_baud
- rs422_cmd_wait
- defoff
- defsens
- icarus_subreg_aliases
- icarus_monitor_controls
- daedalus_subreg_aliases
- daedalus_monitor_controls
- subreglist

Static Public Attributes

- registers
- list subregisters

6.8.1 Detailed Description

```
Livermore LLNL v4.0 board

Compatible communication protocols: RS422, GigE
Compatible sensors: icarus, icarus2, daedalus
```

Definition at line 29 of file LLNL_v4.py.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 __init__()

```
nsCamera.boards.LLNL_v4.llnl_v4.__init__ (
                self,
                camassem )
```

Definition at line 181 of file LLNL_v4.py.

```
00181     def __init__(self, camassem):
00182         self.ca = camassem
00183         self.logcrit = self.ca.logcritbase + "[LLNL_v4] "
00184         self.logerr = self.ca.logerrbase + "[LLNL_v4] "
00185         self.logwarn = self.ca.logwarnbase + "[LLNL_v4] "
00186         self.loginfo = self.ca.loginfobase + "[LLNL_v4] "
00187         self.logdebug = self.ca.logdebugbase + "[LLNL_v4] "
00188         logging.info(self.loginfo + "Initializing board object")
00189         self.VREF = 3.3  # must be supplied externally for ADC128S102
00190         self.ADC5_mult = 1
00191 
00192         # ADC128S102; False => monitor range runs 0 to monmax, True => +/- monmax
00193         self.ADC5_bipolar = False
00194         self.rs422_baud = 921600
00195         self.rs422_cmd_wait = 0.3
00196 
00197         fpgaNum_pkt = Packet(cmd="1", addr=self.registers["FPGA_NUM"])
00198         fpgaRev_pkt = Packet(cmd="1", addr=self.registers["FPGA_REV"])
00199 
00200         _, _ = self.ca.sendCMD(fpgaNum_pkt)  # dummy duplicate call
00201         err, rval = self.ca.sendCMD(fpgaNum_pkt)
00202         self.ca.FPGANum = rval[8:16]
00203 
00204         err, rval = self.ca.sendCMD(fpgaRev_pkt)
00205         self.ca.FPGAVersion = rval[8:16]
00206 
00207         self.defoff = 34.5  # default pressure sensor offset
00208         self.defsens = 92.5  # default pressure sensor sensitivity
00209 
00210         # TODO: move to sensor scripts?
00211         # map channels to signal names for abstraction at the camera assembler level;
00212         #   each requires a corresponding entry in 'subregisters'
00213         self.icarus_subreg_aliases = OrderedDict(
00214             {
00215                 "HST_A_PDELAY": "DACA",
00216                 "HST_A_NDELAY": "DACB",
00217                 "HST_B_PDELAY": "DACC",
00218                 "HST_B_NDELAY": "DACD",
00219                 "HST_RO_IBIAS": "DACE",
00220                 "HST_RO_NC_IBIAS": "DACE",
00221                 "HST_OSC_CTL": "DACF",
00222                 "VAB": "DACG",
00223                 "VRST": "DACH",
00224                 "MON_PRES_MINUS": "MON_CH1",
00225                 "MON_PRES_PLUS": "MON_CH2",
00226                 "MON_TEMP": "MON_CH3",
00227                 "MON_COL_TOP_IBIAS_IN": "MON_CH4",
00228                 "MON_HST_OSC_R_BIAS": "MON_CH5",
00229                 "MON_VAB": "MON_CH6",
00230                 "MON_HST_RO_IBIAS": "MON_CH7",
00231                 "MON_HST_RO_NC_IBIAS": "MON_CH7",
00232                 "MON_VRST": "MON_CH8",
00233                 "MON_COL_BOT_IBIAS_IN": "MON_CH9",
00234                 "MON_HST_A_PDELAY": "MON_CH10",
00235                 "MON_HST_B_NDELAY": "MON_CH11",
00236                 "DOSIMETER": "MON_CH12",
00237                 "MON_HST_OSC_VREF_IN": "MON_CH13",
00238                 "MON_HST_B_PDELAY": "MON_CH14",
00239                 "MON_HST_OSC_CTL": "MON_CH15",
00240                 "MON_HST_A_NDELAY": "MON_CH16",
00241                 "MON_CHA": "MON_CH10",
00242                 "MON_CHB": "MON_CH16",
00243                 "MON_CHC": "MON_CH14",
00244                 "MON_CHD": "MON_CH11",
00245                 "MON_CHE": "MON_CH7",
00246                 "MON_CHF": "MON_CH15",
                "MON_CHG": "MON_CH6",
                "MON_CHH": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.icarus_monitor_controls = OrderedDict(
            {
                "MON_CH10": "DACA",
                "MON_CH16": "DACB",
                "MON_CH14": "DACC",
                "MON_CH11": "DACD",
                "MON_CH7": "DACE",
                "MON_CH15": "DACF",
                "MON_CH6": "DACG",
                "MON_CH8": "DACH",
            }
        )
        self.daedalus_subreg_aliases = OrderedDict(
            {
                "HST_OSC_VREF_IN": "DACC",
                "HST_OSC_CTL": "DACE",
                "COL_TST_IN": "DACF",
                "VAB": "DACG",
                "VRST": "DACH",
                "MON_PRES_MINUS": "MON_CH1",
                "MON_PRES_PLUS": "MON_CH2",
                "MON_TEMP": "MON_CH3",
                "MON_VAB": "MON_CH6",
                "MON_HST_OSC_CTL": "MON_CH7",
                "MON_TSENSE_OUT": "MON_CH10",
                "MON_BGREF": "MON_CH11",
                "DOSIMETER": "MON_CH12",
                "MON_HST_RO_NC_IBIAS": "MON_CH13",
                "MON_HST_OSC_VREF_IN": "MON_CH14",
                "MON_COL_TST_IN": "MON_CH15",
                "MON_HST_OSC_PBIAS_PAD": "MON_CH16",
                "MON_CHC": "MON_CH14",
                "MON_CHE": "MON_CH7",
                "MON_CHF": "MON_CH15",
                "MON_CHG": "MON_CH6",
                "MON_CHH": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.daedalus_monitor_controls = OrderedDict(
            {
                "MON_CH14": "DACC",
                "MON_CH7": "DACE",
                "MON_CH15": "DACF",
                "MON_CH6": "DACG",
                "MON_CH8": "DACH",
            }
        )
        self.subreglist = []
        for s in self.subregisters:
            self.subreglist.append(s[0].upper())
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)

        # set voltage ranges for all DACs - WARNING: actual output voltage limited to
        #   external supply (3.3 V)
        # setpot('potx', n) will generate 3.3 V for all n > .66
        for n in range(0, 8):
            potname = "DAC" + string.ascii_uppercase[n]
            potobj = getattr(self, potname)
            potobj.minV = 0
            potobj.maxV = 5  #
            potobj.resolution = (
                1.0 * potobj.maxV - potobj.minV
            ) / potobj.max_value  # 76 uV / LSB

```

6.8.3 Member Function Documentation

6.8.3.1 checkStatus()

```
nsCamera.boards.LLNL_v4.llnl_v4.checkStatus (
              self )
```

```
Check status register, convert to reverse-order bit stream (i.e., bit 0 is
  statusbits[0])

Returns:
    bit string (no '0b') in reversed order
```

Definition at line 706 of file LLNL_v4.py.

```
    def checkStatus(self):
        """
        Check status register, convert to reverse-order bit stream (i.e., bit 0 is
          statusbits[0])

        Returns:
            bit string (no '0b') in reversed order
        """
        err, rval = self.ca.getRegister("STAT_REG")
        rvalbits = bin(int(rval, 16))[2:].zfill(32)
        statusbits = rvalbits[::-1]
        return statusbits

```

6.8.3.2 checkStatus2()

```
nsCamera.boards.LLNL_v4.llnl_v4.checkStatus2 (
              self )
```

```
Check second status register, convert to reverse-order bit stream (i.e., bit 0
  is statusbits[0])

Returns: bit string (no '0b') in reversed order
```

Definition at line 719 of file LLNL_v4.py.

```
    def checkStatus2(self):
        """
        Check second status register, convert to reverse-order bit stream (i.e., bit 0
          is statusbits[0])

        Returns: bit string (no '0b') in reversed order
        """
        err, rval = self.ca.getRegister("STAT_REG2")
        rvalbits = bin(int(rval, 16))[2:].zfill(6)
        statusbits = rvalbits[::-1]
        return statusbits

```

6.8.3.3 clearStatus()

```
nsCamera.boards.LLNL_v4.llnl_v4.clearStatus (
              self )
```

```
Check status registers to clear them

Returns:
    error string
```

Definition at line 692 of file LLNL_v4.py.

```
    def clearStatus(self):
        """
        Check status registers to clear them

        Returns:
            error string
        """
        err1, rval = self.ca.getRegister("STAT_REG_SRC")
        err2, rval = self.ca.getRegister("STAT_REG2_SRC")
        err = err1 + err2
        if err:
            logging.error(self.logerr + "clearStatus failed")
        return err
```

6.8.3.4 configADCs()

```
nsCamera.boards.LLNL_v4.llnl_v4.configADCs (
              self )
```

```
Sets default ADC configuration (does not latch settings)

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 402 of file LLNL_v4.py.

```
    def configADCs(self):
        """
        Sets default ADC configuration (does not latch settings)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "configADCs")

        control_messages = [
            # just in case ADC_RESET was set on any of the ADCs (pull all ADCs out of
            #   reset)
            ("ADC_RESET", "00000000"),
            # workaround for uncertain behavior after previous readoff
            ("ADC1_CONFIG_DATA", "FFFFFFFF"),
            ("ADC2_CONFIG_DATA", "FFFFFFFF"),
            ("ADC3_CONFIG_DATA", "FFFFFFFF"),
            ("ADC4_CONFIG_DATA", "FFFFFFFF"),
            ("ADC_CTL", "FFFFFFFF"),
            ("ADC1_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC2_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC3_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC4_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
        ]
        return self.ca.submitMessages(control_messages, " configADCs: ")
```

6.8.3.5 disarm()

```
nsCamera.boards.LLNL_v4.llnl_v4.disarm (
              self )
```

```
Takes camera out of trigger wait state. Has no effect if camera is not in wait
  state.

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 440 of file LLNL_v4.py.

```
    def disarm(self):
        """
        Takes camera out of trigger wait state. Has no effect if camera is not in wait
          state.

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "disarm")
        self.ca.clearStatus()
        self.ca.armed = False
        control_messages = [
            ("HW_TRIG_EN", "0"),
            ("SW_TRIG_EN", "0"),
        ]
        self.ca.comms.skipError = False
        return self.ca.submitMessages(control_messages, " disarm: ")

```

6.8.3.6 dumpStatus()

```
nsCamera.boards.LLNL_v4.llnl_v4.dumpStatus (
              self )
```

```
Create dictionary of status values, DAC settings, monitor values, and register
  values.

Returns:
    dictionary of system diagnostic values
```

Definition at line 821 of file LLNL_v4.py.

```
    def dumpStatus(self):
        """
        Create dictionary of status values, DAC settings, monitor values, and register
          values.

        Returns:
            dictionary of system diagnostic values
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()

        temp = int(statusbits[23:16:-1], 2) * 3.3 * 1000 / 4096
        press = int(statusbits[:23:-1], 2) * 3.3 * 1000 / 4096

        statDict = OrderedDict(
            {
                "Temperature sensor reading (Deg C)": "{0:1.2f}".format(temp),
                "Pressure reading (Torr)": str(round(self.ca.getPressure(), 3)),
                "Pressure sensor reading (mV)": "{0:1.2f}".format(press),
                "Sensor read complete": str(statusbits[0]),
                "Coarse trigger detected": str(statusbits[1]),
                "Fine trigger detected": str(statusbits[2]),
                "Sensor readout in progress": str(statusbits[5]),
                "Sensor readout complete": str(statusbits[6]),
                "SRAM readout started": str(statusbits[7]),
                "SRAM readout complete": str(statusbits[8]),
                "High-speed timing configured": str(statusbits[9]),
                "All ADCs configured": str(statusbits[10]),
                "All DACs configured": str(statusbits[11]),
                "Timer has reset": str(statusbits[13]),
                "Camera is Armed": str(statusbits[14]),
                "FPA_IF_TO": str(statusbits2[0]),
                "SRAM_RO_TO": str(statusbits2[1]),
                "PixelRd Timeout Error": str(statusbits2[2]),
                "UART_TX_TO_RST": str(statusbits2[3]),
                "UART_RX_TO_RST": str(statusbits2[4]),
                "PDBIAS Unready": str(statusbits2[5]),
            }
        )

        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            senslabs = {
                3: "W3_Top_A_Edge1 detected",
                4: "W3_Top_B_Edge1 detected",
                12: "HST_All_W_En detected",
            }
        else:
            senslabs = {
                3: "RSLROWOUTA",
                4: "RSLROWOUTB",
                12: "RSLNALLWENB",
                15: "RSLNALLWENA",
                16: "Config HST is done",
            }
        sensDict = {senslabs[x]: str(statusbits[x]) for x in senslabs.keys()}

        DACDict = OrderedDict()
        MonDict = OrderedDict()
        for entry in self.subreg_aliases:
            if self.subreg_aliases[entry][0] == "D":
                val = str(round(self.ca.getPotV(entry), 3))
                DACDict["DAC_" + entry] = val
            else:
                val = str(round(self.ca.getMonV(entry), 3))
                MonDict[entry] = val

        regDict = OrderedDict()
        for key in self.registers.keys():
            # Load in all registers except for the read-clear status registers.
            if key == "STAT_REG_SRC" or key == "STAT_REG2_SRC":
                pass
            else:
                err, rval = self.ca.getRegister(key)
                regDict[key] = rval

        dumpDict = OrderedDict()
        for x in [statDict, sensDict, MonDict, DACDict, regDict]:
            dumpDict.update(x)
        return dumpDict


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.8.3.7 enableLED()

```
nsCamera.boards.LLNL_v4.llnl_v4.enableLED (
                self,
                status )
```

```
Dummy function; feature is not implemented on LLNL_V4 board

Returns:
    tuple: dummy of (error string, response string from setSubregister())
```

Definition at line 571 of file LLNL_v4.py.

```
    def enableLED(self, status):
        """
        Dummy function; feature is not implemented on LLNL_V4 board

        Returns:
            tuple: dummy of (error string, response string from setSubregister())
        """
        del status
        return "", "0"

```

6.8.3.8 getPressure()

```
nsCamera.boards.LLNL_v4.llnl_v4.getPressure (
                self,
                offset = None,
                sensitivity = None,
                units = None )
```

```
Read pressure sensor. Uses default offset and sensitivity defined in init
  function unless alternatives are specified. NOTE: to reset defaults, reassign
  board.defoff and board.defsens explicitly

Args:
    offset: non-default offset in mv/V
    sensitivity: non-default sensitivity in mV/V/span
    units: units to report pressure (defaults to Torr, options are psi, bar,
      inHg, atm)

Returns:
    Pressure as float in chosen units, defaults to torr
```

Definition at line 653 of file LLNL_v4.py.

```
    def getPressure(self, offset=None, sensitivity=None, units=None):
        """
        Read pressure sensor. Uses default offset and sensitivity defined in init
          function unless alternatives are specified. NOTE: to reset defaults, reassign
          board.defoff and board.defsens explicitly

        Args:
            offset: non-default offset in mv/V
            sensitivity: non-default sensitivity in mV/V/span
            units: units to report pressure (defaults to Torr, options are psi, bar,
              inHg, atm)

        Returns:
            Pressure as float in chosen units, defaults to torr
        """
        if offset is None:
            offset = self.defoff
        if sensitivity is None:
            sensitivity = self.defsens
        if units is None:
            units = "torr"
        pplus = self.ca.getMonV("MON_PRES_PLUS")
        pminus = self.ca.getMonV("MON_PRES_MINUS")
        delta = 1000 * (pplus - pminus)
        ratio = sensitivity / 30  # nominal is 21/30
        psi = (delta - offset) / ratio
        if units.lower() == "psi":
            press = psi
        elif units.lower() == "bar":
            press = psi / 14.504
        elif units.lower() == "atm":
            press = psi / 14.695
        elif units.lower() == "inHg":
            press = psi * 2.036
        else:
            press = 51.715 * psi  # default to Torr

        return press

```

6.8.3.9 getTemp()

```
nsCamera.boards.LLNL_v4.llnl_v4.getTemp (
               self,
               scale = None )
```

```
Read temperature sensor
Args:
    scale: temperature scale to report (defaults to C, options are F and K)

Returns:
    temperature as float on given scale
```

Definition at line 628 of file LLNL_v4.py.

```
    def getTemp(self, scale=None):
        """
        Read temperature sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)

        Returns:
            temperature as float on given scale
        """
        err, rval = self.ca.getMonV("MON_TEMP", errflag=True)
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0
        ctemp = rval * 1000 - 273.15
        if scale == "K":
            temp = ctemp + 273.15
        elif scale == "F":
            temp = 1.8 * ctemp + 32
        else:
            temp = ctemp
        return temp

```

6.8.3.10 getTimer()

```
nsCamera.boards.LLNL_v4.llnl_v4.getTimer (
               self )
```

```
Read value of on-board timer

Returns:
    timer value as integer
```

Definition at line 543 of file LLNL_v4.py.

```
    def getTimer(self):
        """
        Read value of on-board timer

        Returns:
            timer value as integer
        """
        logging.info(self.loginfo + "getTimer")
        err, rval = self.ca.getRegister("TIMER_VALUE")
        if err:
            logging.error(
                self.logerr + "unable to retrieve timer information (getTimer), "
                'returning "0" '
            )
            return 0
        return int(rval, 16)

```

6.8.3.11 initBoard()

```
nsCamera.boards.LLNL_v4.llnl_v4.initBoard (
                self )
```

```
Register and reset board, set up firmware for sensor

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 325 of file LLNL_v4.py.

```
    def initBoard(self):
        """
        Register and reset board, set up firmware for sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initBoard LLNLv4")
        control_messages = []
        self.clearStatus()
        self.configADCs()
        return self.ca.submitMessages(control_messages, " initBoard: ")

```

6.8.3.12 initPots()

```
nsCamera.boards.LLNL_v4.llnl_v4.initPots (
                self )
```

```
Dummy function; initial DAC values are set by firmware at startup

Returns:
    tuple (empty string, empty string)
```

Definition at line 338 of file LLNL_v4.py.

```
    def initPots(self):
        """
        Dummy function; initial DAC values are set by firmware at startup

        Returns:
            tuple (empty string, empty string)
        """
        logging.debug(self.logdebug + "InitPots")
        return "", ""

```

6.8.3.13 initSensor()

```
nsCamera.boards.LLNL_v4.llnl_v4.initSensor (
                self )
```

```
Register sensor, set default timing settings

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 368 of file LLNL_v4.py.

```
    def initSensor(self):
        """
        Register sensor, set default timing settings

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initSensor")
        if int(self.ca.FPGANum[7]) != self.ca.sensor.fpganumID:
            logging.warning(
                self.logwarn + "unable to confirm sensor compatibility with FPGA"
            )
        self.registers.update(self.ca.sensor.sens_registers)
        self.subregisters.extend(self.ca.sensor.sens_subregisters)
        for s in self.ca.sensor.sens_subregisters:
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)
            self.subreglist.append(s[0])
        # TODO: self.ca.checkSensorVoltStat() # SENSOR_VOLT_STAT and SENSOR_VOLT_CTL are
        #   deactivated for v4 icarus and daedalus firmware for now, is this permanent?
        control_messages = self.ca.sensorSpecific() + [
            # ring w/caps=01, relax=00, ring w/o caps = 02
            ("OSC_SELECT", "00"),
            ("FPA_DIVCLK_EN_ADDR", "00000001"),
        ]
        return self.ca.submitMessages(control_messages, " initSensor: ")

```

6.8.3.14 latchPots()

```
nsCamera.boards.LLNL_v4.llnl_v4.latchPots (
                self )
```

```
Latch DAC settings into sensor

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 348 of file LLNL_v4.py.

```
    def latchPots(self):
        """
        Latch DAC settings into sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "latchPots")
        control_messages = [
            ("DAC_CTL", "00000001"),  # latches register settings for DACA
            ("DAC_CTL", "00000003"),
            ("DAC_CTL", "00000005"),
            ("DAC_CTL", "00000007"),
            ("DAC_CTL", "00000009"),
            ("DAC_CTL", "0000000B"),
            ("DAC_CTL", "0000000D"),
            ("DAC_CTL", "0000000F"),
        ]
        return self.ca.submitMessages(control_messages, " latchPots: ")

```

6.8.3.15 readSRAM()

```
nsCamera.boards.LLNL_v4.llnl_v4.readSRAM (
              self )
```

```
Start readoff of SRAM

Returns:
    tuple (error string, response string from register set)
```

Definition at line 495 of file LLNL_v4.py.

```
    def readSRAM(self):
        """
        Start readoff of SRAM

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "readSRAM")
        control_messages = [("READ_SRAM", "1")]
        return self.ca.submitMessages(control_messages, " readSRAM: ")

```

6.8.3.16 reportEdgeDetects()

```
nsCamera.boards.LLNL_v4.llnl_v4.reportEdgeDetects (
              self )
```

```
Report edge detects
```

Definition at line 788 of file LLNL_v4.py.

```
    def reportEdgeDetects(self):
        """
        Report edge detects
        """
        err, rval = self.ca.getRegister("STAT_EDGE_DETECTS")
        # shift to left to fake missing edge detect
        edgebits = bin(int(rval, 16) « 1)[2:].zfill(32)
        # reverse to get order matching assignment
        bitsrev = edgebits[::-1]
        detdict = {}
        bitidx = 0
        for frame in range(4):
            for vert in ("TOP", "BOT"):
                for edge in range(1, 3):
                    for hor in ("A", "B"):
                        detname = (
                            "W"
                            + str(frame)
```

```
00806                             + "_"
00807                             + vert
00808                             + "_"
00809                             + hor
00810                             + "_EDGE"
00811                             + str(edge)
00812                         )
00813                         detdict[detname] = bitsrev[bitidx]
00814                         bitidx += 1
00815         # remove faked detect
00816         del detdict["W0_TOP_A_EDGE1"]
00817         logging.info(self.loginfo + "Edge detect report:")
00818         for key, val in detdict.items():
00819             logging.info(self.loginfo + key + ": " + val)
00820
```

6.8.3.17 reportStatus()

```
nsCamera.boards.LLNL_v4.llnl_v4.reportStatus (
                self )
```

```
Check contents of status register, print relevant messages
```

Definition at line 731 of file LLNL_v4.py.

```
00731     def reportStatus(self):
00732         """
00733         Check contents of status register, print relevant messages
00734         """
00735         statusbits = self.checkStatus()
00736         statusbits2 = self.checkStatus2()
00737
00738         logging.info(self.loginfo + "Status report:")
00739         if int(statusbits[0]):
00740             print(self.loginfo + "Sensor read complete")
00741         if int(statusbits[1]):
00742             print(self.loginfo + "Coarse trigger detected")
00743         if int(statusbits[2]):
00744             print(self.loginfo + "Fine trigger detected")
00745         if int(statusbits[5]):
00746             print(self.loginfo + "Sensor readout in progress")
00747         if int(statusbits[6]):
00748             print(self.loginfo + "Sensor readout complete")
00749         if int(statusbits[7]):
00750             print(self.loginfo + "SRAM readout started")
00751         if int(statusbits[8]):
00752             print(self.loginfo + "SRAM readout complete")
00753         if int(statusbits[9]):
00754             print(self.loginfo + "High-speed timing configuration started")
00755         if int(statusbits[10]):
00756             print(self.loginfo + "All ADCs configured")
00757         if int(statusbits[11]):
00758             print(self.loginfo + "All DACs configured")
00759         if int(statusbits[13]):
00760             print(self.loginfo + "Timer has reset")
00761         if int(statusbits[14]):
00762             print(self.loginfo + "Camera is Armed")
00763         if int(statusbits[16]):
00764             print(self.loginfo + "High-speed timing configuration complete")
00765         self.ca.sensor.reportStatusSensor(statusbits, statusbits2)
00766         temp = int(statusbits[23:16:-1], 2) * 3.3 * 1000 / 4096
00767         logging.info(
00768             self.loginfo + "Temperature reading: " + "{0:1.2f}".format(temp) + " C"
00769         )
00770         press = int(statusbits[:23:-1], 2) * 3.3 * 1000 / 4096
00771         logging.info(
00772             self.loginfo
00773             + "Pressure sensor reading: "
00774             + "{0:1.2f}".format(press)
00775             + " mV"
00776         )
00777         if int(statusbits2[0]):
00778             print(self.loginfo + "FPA_IF_TO")
```

```
        if int(statusbits2[1]):
            print(self.loginfo + "SRAM_RO_TO")
        if int(statusbits2[2]):
            print(self.loginfo + "PixelRd Timeout Error")
        if int(statusbits2[3]):
            print(self.loginfo + "UART_TX_TO_RST")
        if int(statusbits2[4]):
            print(self.loginfo + "UART_RX_TO_RST")

```

6.8.3.18 resetTimer()

```
nsCamera.boards.LLNL_v4.llnl_v4.resetTimer (
                self )
```

```
Reset on-board timer

Returns:
    tuple (error string, response string from register set)
```

Definition at line 560 of file LLNL_v4.py.

```
    def resetTimer(self):
        """
        Reset on-board timer

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "resetTimer")
        control_messages = [("RESET_TIMER", "1"), ("RESET_TIMER", "0")]
        return self.ca.submitMessages(control_messages, " resetTimer: ")

```

6.8.3.19 setLED()

```
nsCamera.boards.LLNL_v4.llnl_v4.setLED (
                self,
                LED,
                status )
```

```
Dummy function; feature is not implemented on LLNL_V4 board

Returns:
    tuple: dummy of (error string, response string from setSubregister())
```

Definition at line 581 of file LLNL_v4.py.

```
    def setLED(self, LED, status):
        """
        Dummy function; feature is not implemented on LLNL_V4 board

        Returns:
            tuple: dummy of (error string, response string from setSubregister())
        """
        del LED, status
        return "", "0"

```

6.8.3.20 setPowerSave()

```
nsCamera.boards.LLNL_v4.llnl_v4.setPowerSave (
                self,
                status )
```

```
Select powersave option

Args:
    status: setting for powersave option (1 is enabled)

Returns:
    tuple (error string, response string from setSubregister())
```

Definition at line 591 of file LLNL_v4.py.

```
    def setPowerSave(self, status):
        """
        Select powersave option

        Args:
            status: setting for powersave option (1 is enabled)

        Returns:
            tuple (error string, response string from setSubregister())
        """
        if status:
            status = 1
        return self.ca.setSubregister("POWERSAVE", str(status))

```

6.8.3.21 setPPER()

```
nsCamera.boards.LLNL_v4.llnl_v4.setPPER (
                self,
                pollperiod )
```

```
Set polling period for ADCs.
Args:
    pollperiod: milliseconds, between 1 and 255; defaults to 50

Returns:
    tuple (error string, response string from setSubregister() OR invalid time
      setting string)
```

Definition at line 605 of file LLNL_v4.py.

```
    def setPPER(self, pollperiod):
        """
        Set polling period for ADCs.
        Args:
            pollperiod: milliseconds, between 1 and 255; defaults to 50

        Returns:
            tuple (error string, response string from setSubregister() OR invalid time
              setting string)
        """
        if pollperiod is None:
            pollperiod = 50
        if not isinstance(pollperiod, int) or pollperiod < 1 or pollperiod > 255:
            err = (
                self.logerr + "invalid poll period submitted. Setting remains "
                "unchanged. "
            )
            logging.error(err)
            return err, str(pollperiod)
        else:
            binset = bin(pollperiod)[2:].zfill(8)
            return self.ca.setSubregister("PPER", binset)

```

6.8.3.22 softReboot()

```
nsCamera.boards.LLNL_v4.llnl_v4.softReboot (
              self )
```

```
Perform software reboot of board. WARNING: board reboot will likely prevent
  correct response and therefore will generate an error message

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 428 of file LLNL_v4.py.

```
    def softReboot(self):
        """
        Perform software reboot of board. WARNING: board reboot will likely prevent
          correct response and therefore will generate an error message

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "reboot")
        control_messages = [("RESET", "0")]
        return self.ca.submitMessages(control_messages, " disarm: ")

```

6.8.3.23 startCapture()

```
nsCamera.boards.LLNL_v4.llnl_v4.startCapture (
              self,
              mode = "Hardware" )
```

```
Selects trigger mode and enables board for image capture

Args:
    mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
      insensitive)

Returns:
    tuple (error string, response string) from final control message
```

Definition at line 458 of file LLNL_v4.py.

```
    def startCapture(self, mode="Hardware"):
        """
        Selects trigger mode and enables board for image capture

        Args:
            mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
              insensitive)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "startCapture")
        if self.ca.sensmanual:
            timingReg = "MANSHUT_MODE"
        else:
            timingReg = "HST_MODE"

        if mode.upper()[0] == "S":  # SOFTWARE
            trigmess = [
                ("HW_TRIG_EN", "0"),
                ("SW_TRIG_EN", "1"),
                ("SW_TRIG_START", "1"),
00480             ]
00481         else:  # HARDWARE
00482             trigmess = [
00483                 ("SW_TRIG_EN", "0"),
00484                 ("HW_TRIG_EN", "1"),
00485             ]
00486 
00487         control_messages = [
00488             ("ADC_CTL", "0000000F"),  # configure all ADCs
00489             (timingReg, "1"),
00490         ]
00491 
00492         control_messages.extend(trigmess)
00493         return self.ca.submitMessages(control_messages, " startCapture: ")
00494 
```

6.8.3.24 waitForSRAM()

```
nsCamera.boards.LLNL_v4.llnl_v4.waitForSRAM (
                self,
                timeout )
```

```
Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
  timeout = None or zero means wait indefinitely

Args:
    timeout - time in seconds before readoff proceeds automatically without
      waiting for SRAM_READY flag

Returns:
    error string
```

Definition at line 506 of file LLNL_v4.py.

```
00506     def waitForSRAM(self, timeout):
00507         """
00508         Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
00509           timeout = None or zero means wait indefinitely
00510 
00511         Args:
00512             timeout - time in seconds before readoff proceeds automatically without
00513               waiting for SRAM_READY flag
00514 
00515         Returns:
00516             error string
00517         """
00518         logging.info(self.loginfo + "waitForSRAM, timeout = " + str(timeout))
00519         waiting = True
00520         starttime = time.time()
00521         err = ""
00522         while waiting:
00523             err, status = self.ca.getSubregister("SRAM_READY")
00524             if err:
00525                 err = self.logerr + "error in register read: " + err + " (waitForSRAM)"
00526                 logging.error(err)
00527             if int(status):
00528                 waiting = False
00529                 logging.info(self.loginfo + "SRAM ready")
00530             if self.ca.abort:
00531                 waiting = False
00532                 logging.info(self.loginfo + "readoff aborted by user")
00533                 self.ca.abort = False
00534             if timeout and time.time() - starttime > timeout:
00535                 err += self.logerr + "SRAM timeout; proceeding with download attempt"
00536                 logging.error(err)
00537                 return err
00538             # Slow down for debugging (avoid thousands of messages)
00539             if self.ca.verbose >= 5:
00540                 time.sleep(0.5)
00541         return err
00542 
```

6.8.4 Member Data Documentation

6.8.4.1 ADC5_bipolar

`nsCamera.boards.LLNL_v4.llnl_v4.ADC5_bipolar`

Definition at line 193 of file LLNL_v4.py.

6.8.4.2 ADC5_mult

`nsCamera.boards.LLNL_v4.llnl_v4.ADC5_mult`

Definition at line 190 of file LLNL_v4.py.

6.8.4.3 ca

`nsCamera.boards.LLNL_v4.llnl_v4.ca`

Definition at line 182 of file LLNL_v4.py.

6.8.4.4 daedalus_monitor_controls

`nsCamera.boards.LLNL_v4.llnl_v4.daedalus_monitor_controls`

Definition at line 291 of file LLNL_v4.py.

6.8.4.5 daedalus_subreg_aliases

`nsCamera.boards.LLNL_v4.llnl_v4.daedalus_subreg_aliases`

Definition at line 264 of file LLNL_v4.py.

6.8.4.6 defoff

`nsCamera.boards.LLNL_v4.llnl_v4.defoff`

Definition at line 207 of file LLNL_v4.py.

6.8.4.7 defsens

`nsCamera.boards.LLNL_v4.llnl_v4.defsens`

Definition at line 208 of file LLNL_v4.py.

6.8.4.8 icarus_monitor_controls

```
nsCamera.boards.LLNL_v4.llnl_v4.icarus_monitor_controls
```

Definition at line 252 of file LLNL_v4.py.

6.8.4.9 icarus_subreg_aliases

```
nsCamera.boards.LLNL_v4.llnl_v4.icarus_subreg_aliases
```

Definition at line 213 of file LLNL_v4.py.

6.8.4.10 logcrit

```
nsCamera.boards.LLNL_v4.llnl_v4.logcrit
```

Definition at line 183 of file LLNL_v4.py.

6.8.4.11 logdebug

```
nsCamera.boards.LLNL_v4.llnl_v4.logdebug
```

Definition at line 187 of file LLNL_v4.py.

6.8.4.12 logerr

```
nsCamera.boards.LLNL_v4.llnl_v4.logerr
```

Definition at line 184 of file LLNL_v4.py.

6.8.4.13 loginfo

```
nsCamera.boards.LLNL_v4.llnl_v4.loginfo
```

Definition at line 186 of file LLNL_v4.py.

6.8.4.14 logwarn

```
nsCamera.boards.LLNL_v4.llnl_v4.logwarn
```

Definition at line 185 of file LLNL_v4.py.

6.8.4.15 registers

`nsCamera.boards.LLNL_v4.llnl_v4.registers [static]`

Definition at line 38 of file LLNL_v4.py.

6.8.4.16 rs422_baud

`nsCamera.boards.LLNL_v4.llnl_v4.rs422_baud`

Definition at line 194 of file LLNL_v4.py.

6.8.4.17 rs422_cmd_wait

`nsCamera.boards.LLNL_v4.llnl_v4.rs422_cmd_wait`

Definition at line 195 of file LLNL_v4.py.

6.8.4.18 subregisters

`list nsCamera.boards.LLNL_v4.llnl_v4.subregisters [static]`

Definition at line 101 of file LLNL_v4.py.

6.8.4.19 subreglist

`nsCamera.boards.LLNL_v4.llnl_v4.subreglist`

Definition at line 300 of file LLNL_v4.py.

6.8.4.20 VREF

`nsCamera.boards.LLNL_v4.llnl_v4.VREF`

Definition at line 189 of file LLNL_v4.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/boards/LLNL_v4.py

6.9 nsCamera.utils.Packet.Packet Class Reference

Public Member Functions

- __init__ (self, preamble="aaaa", cmd="0", addr="", data="00000000", seqID="", payload_length="", payload="", crc="")
- pktStr (self)
- calculateCRC (self)
- checkCRC (self)
- checkReadPacket (self, resppkt)
- checkResponsePacket (self, resppkt)
- checkResponseString (self, respstr)

Public Attributes

- PY3
- preamble
- cmd
- addr
- data
- seqID
- payload_length
- payload
- crc
- type

6.9.1 Detailed Description

```
Packet object for communication with boards. See ICD for details.

Single Command/Response packet:
+----------+-----------+-----------+------------+-----------+
| 16 bits  |  4 bits   |  12 bits  |   32 bits  |  16 bits  |
| Preamble |  Command  |  Address  |    Data    |   CRC16   |
+----------+-----------+-----------+------------+-----------+

Read Burst Response packet:

+----------+-----------+---------------+---------------+
| 16 bits  |  4 bits   |    4 bits     |    16 bits    %
| Preamble |  Command  |  Sub-command  |  Sequence ID  %
+----------+-----------+---------------+---------------+
                                        +------------------+------------+-----------+
                                        %      16 bits     |  Variable  |  16 bits  |
                                        %  Payload Length  |  Payload   |   CRC16   |
                                        +------------------+------------+-----------+
```

Definition at line 29 of file Packet.py.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 __init__()

```
nsCamera.utils.Packet.Packet.__init__ (
                self,
                preamble = "aaaa",
                cmd = "0",
                addr = "",
                data = "00000000",
                seqID = "",
                payload_length = "",
                payload = "",
                crc = "" )
```

Definition at line 52 of file Packet.py.

```
00063     ):
00064         self.PY3 = sys.version_info > (3,)
00065         self.preamble = preamble  # 16 bit packet preamble
00066         self.cmd = str(cmd)  # 4 bit command packet
00067         self.addr = addr.zfill(3)  # 12 bit address packet
00068         self.data = data.zfill(8)  # 32 bit data packet
00069         # 16 bit sequence ID packet (only Read Burst)
00070         self.seqID = seqID
00071         # 16 bit payload packet (only Read Burst)
00072         self.payload_length = payload_length
00073         # variable payload packet (only Read Burst) for now it’s 16 bits
00074         self.payload = payload
00075         # 16 bit CRC-CCIT (XModem) packet
00076         self.crc = crc
00077         self.type = ""
00078         if self.crc == "":  # check if packet to be sent needs crc appended
00079             self.crc = self.calculateCRC()
00080
```

6.9.3 Member Function Documentation

6.9.3.1 calculateCRC()

```
nsCamera.utils.Packet.Packet.calculateCRC (
                self )
```

```
Calculate CRC-CCIT (XModem) (2 bytes) from 8 byte packet for send and rcv

Returns:
    CRC as hexadecimal string without ’0x’
```

Definition at line 108 of file Packet.py.

```
00108     def calculateCRC(self):
00109         """
00110         Calculate CRC-CCIT (XModem) (2 bytes) from 8 byte packet for send and rcv
00111
00112         Returns:
00113             CRC as hexadecimal string without ’0x’
00114         """
00115         preamble = self.preamble
00116         crc = self.crc
00117         self.crc = ""
00118         self.preamble = ""
00119
```

```
        CRC_dec = crc16pure.crc16xmodem(str2bytes(self.pktStr()))
        # input = int type decimal, output = hex string with 0x at the beginning
        CRC_hex_0x = "0x%0.4X" % CRC_dec
        # make all hex letters lower case for comparison
        CRC_hex = CRC_hex_0x.lower()
        # input = hex string with 0x at the beginning, output = hex str with 0x removed
        CRC_hex = CRC_hex[2:]
        self.preamble = preamble
        self.crc = crc
        return CRC_hex

```

6.9.3.2 checkCRC()

```
nsCamera.utils.Packet.Packet.checkCRC (
              self )
```

```
Returns: boolean, True if CRC check passes
```

Definition at line 131 of file Packet.py.

```
    def checkCRC(self):
        """
        Returns: boolean, True if CRC check passes
        """
        return self.calculateCRC() == self.crc

```

6.9.3.3 checkReadPacket()

```
nsCamera.utils.Packet.Packet.checkReadPacket (
              self,
              resppkt )
```

```
Confirm that Read Single occurred without error
Args:
    resppkt: response packet

Returns:
    tuple (error string, response packet as string)
```

Definition at line 137 of file Packet.py.

```
    def checkReadPacket(self, resppkt):
        """
        Confirm that Read Single occurred without error
        Args:
            resppkt: response packet

        Returns:
            tuple (error string, response packet as string)
        """
        err = ""
        if int(resppkt.cmd.upper(), 16) - int(self.cmd.upper(), 16) != 0x8:
            err = "invalid command; "
        if resppkt.addr.upper() != self.addr.upper():
            err += "invalid address; "
        if resppkt.crc.upper() != resppkt.calculateCRC().upper():
            err += "invalid CRC; "
        return err, resppkt.pktStr()

```

6.9.3.4 checkResponsePacket()

```
nsCamera.utils.Packet.Packet.checkResponsePacket (
                self,
                resppkt )
```

```
Confirm that Write Single occurred without error
Args:
    resppkt: response packet

Returns:
    tuple (error string, response packet as string)
```

Definition at line 155 of file Packet.py.

```
    def checkResponsePacket(self, resppkt):
        """
        Confirm that Write Single occurred without error
        Args:
            resppkt: response packet

        Returns:
            tuple (error string, response packet as string)
        """
        err = ""
        if int(resppkt.data, 16) & 1:
            err += "Checksum error; "
        if int(resppkt.data, 16) & 2:
            err += "Invalid command / command not executed; "
        err1, rval = self.checkReadPacket(resppkt)
        err += err1
        return err, rval

```

6.9.3.5 checkResponseString()

```
nsCamera.utils.Packet.Packet.checkResponseString (
                self,
                respstr )
```

```
Checks response string for error indicators
Args:
    respstr: packet as hexadecimal string

Returns:
    tuple (error string, response packet string)
```

Definition at line 173 of file Packet.py.

```
    def checkResponseString(self, respstr):
        """
        Checks response string for error indicators
        Args:
            respstr: packet as hexadecimal string

        Returns:
            tuple (error string, response packet string)
        """
        respstring = respstr.decode(encoding="UTF-8")
        resppkt = Packet(
            preamble=respstring[0:4],
            cmd=respstring[4],
            addr=respstring[5:8],
            data=respstring[8:16],
        )

        if resppkt.cmd == "8":
            # verify response to write command
            err, rval = self.checkResponsePacket(resppkt)
        elif resppkt.cmd == "9":
            err, rval = self.checkReadPacket(resppkt)  # verify response to read command
        else:
            err = "Packet command invalid; "
            rval = ""
        return err, rval


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.9.3.6 pktStr()

```
nsCamera.utils.Packet.Packet.pktStr (
                self )
```

```
Generate hexadecimal string form of packet

Returns:
    packet as hexadecimal string without '0x'
```

Definition at line 81 of file Packet.py.

```
    def pktStr(self):
        """
        Generate hexadecimal string form of packet

        Returns:
            packet as hexadecimal string without '0x'
        """
        if self.seqID != "":
            # Read burst response
            packetparts = [
                self.preamble,
                self.cmd,
                self.seqID,
                self.payload_length,
                self.payload,
                self.crc,
            ]
        else:
            # Single Command/Response response
            packetparts = [self.preamble, self.cmd, self.addr, self.data, self.crc]
        stringparts = [
            part.decode("ascii") if isinstance(part, bytes) else part
            for part in packetparts
        ]
        out = "".join(stringparts)
        return out

```

6.9.4 Member Data Documentation

6.9.4.1 addr

```
nsCamera.utils.Packet.Packet.addr
```

Definition at line 67 of file Packet.py.

6.9.4.2 cmd

`nsCamera.utils.Packet.Packet.cmd`

Definition at line 66 of file Packet.py.

6.9.4.3 crc

`nsCamera.utils.Packet.Packet.crc`

Definition at line 76 of file Packet.py.

6.9.4.4 data

`nsCamera.utils.Packet.Packet.data`

Definition at line 68 of file Packet.py.

6.9.4.5 payload

`nsCamera.utils.Packet.Packet.payload`

Definition at line 74 of file Packet.py.

6.9.4.6 payload_length

`nsCamera.utils.Packet.Packet.payload_length`

Definition at line 72 of file Packet.py.

6.9.4.7 preamble

`nsCamera.utils.Packet.Packet.preamble`

Definition at line 65 of file Packet.py.

6.9.4.8 PY3

`nsCamera.utils.Packet.Packet.PY3`

Definition at line 64 of file Packet.py.

6.9.4.9 seqID

```
nsCamera.utils.Packet.Packet.seqID
```

Definition at line 70 of file Packet.py.

6.9.4.10 type

```
nsCamera.utils.Packet.Packet.type
```

Definition at line 77 of file Packet.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/utils/Packet.py

6.10 nsCamera.comms.RS422.RS422 Class Reference

Public Member Functions

- __init__ (self, camassem, baud=921600, par="O", stop=1)
- serialClose (self)
- sendCMD (self, pkt)
- arm (self, mode)
- readFrames (self, waitOnSRAM, timeout=0, fast=False, columns=1)
- readoff (self, waitOnSRAM, timeout, fast, columns=1)
- writeSerial (self, outstring, timeout)
- readSerial (self, size, timeout=None)
- closeDevice (self)

Public Attributes

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- mode
- baud
- par
- stop
- read_timeout
- write_timeout
- datatimeout
- PY3
- skipError
- port
- payloadsize

Protected Attributes

- _ser

6.10.1 Detailed Description

```
Code to manage RS422 connection. Will automatically query available COM interfaces
  until a board is found. Use the 'port=x' parameter in cameraAssembler call to
  specify a particular COM interface.

Exposed methods:
    arm() - Puts camera into wait state for external trigger
    readFrames() - waits for data ready register flag, then copies camera image data
      into numpy arrays
    readoff() - waits for data ready register flag, then copies camera image data
      into numpy arrays; returns payload, payload size, and error message
    sendCMD(pkt) - sends packet object via serial port
    readSerial(size, timeout) - read 'size' bytes from serial port
    writeSerial(cmd) - submits string 'cmd' (assumes string is preformed packet)
    closeDevice() - close serial connections
```

Definition at line 30 of file RS422.py.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 __init__()

```
nsCamera.comms.RS422.RS422.__init__ (
              self,
              camassem,
              baud = 921600,
              par = "O",
              stop = 1 )
```

```
Args:
    camassem: parent cameraAssembler object
    baud: bits per second
    par: parity type
    stop: number of stop bits
```

Definition at line 48 of file RS422.py.

```
    def __init__(self, camassem, baud=921600, par="O", stop=1):
        """
        Args:
            camassem: parent cameraAssembler object
            baud: bits per second
            par: parity type
            stop: number of stop bits
        """
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[RS422] "
        self.logerr = self.ca.logerrbase + "[RS422] "
        self.logwarn = self.ca.logwarnbase + "[RS422] "
        self.loginfo = self.ca.loginfobase + "[RS422] "
        self.logdebug = self.ca.logdebugbase + "[RS422] "
        logging.info(self.loginfo + "initializing RS422 comms object")
        logging.debug(
            self.logdebug
            + "Init: baud = "
            + str(baud)
            + "; par = "
            + str(par)
            + "; stop = "
            + str(stop)
        )
        self.mode = 0
        self.baud = baud  # Baud rate (bits/second)
        self.par = par  # Parity bit
        self.stop = stop  # Number of stop bits
        self.read_timeout = 1  # default timeout for ordinary packets
        self.write_timeout = 1
        # TODO: make datatimeout a cameraAssembler parameter
        self.datatimeout = 60  # timeout for data read
        logging.debug(
            self.logdebug + "Data timeout = " + str(self.datatimeout) + " seconds"
        )
        self.PY3 = sys.version_info > (3,)
        self.skipError = False
        port = ""
        ports = list(serial.tools.list_ports.comports())
        logging.debug(self.logdebug + "Comports: " + str(ports))
        for p, desc, add in ports:
            if self.ca.port is None or p == "COM" + str(self.ca.port):
                logging.info(self.loginfo + "found comm port " + p)
                try:
                    with serial.Serial(
                        p,
                        self.baud,
                        parity=self.par,
                        timeout=0.01,
                        write_timeout=0.01,
                    ) as ser:
                        ser.write(str2bytes("aaaa1000000000001a84"))
                        time.sleep(1)
                        s = ser.read(10)
                        resp = bytes2str(s)
                        logging.debug(self.logdebug + "Init response: " + str(resp))
                        if (
                            resp[0:5].lower() == "aaaa9"
                        ):  # TODO: add check for RS422 bit in board description
                            boardid = resp[8:10]
                            if boardid == "00":
                                logging.critical(
                                    self.logcrit + "SNLrevC board detected - not "
                                    "compatible with nsCamera >= 2.0"
                                )
                                sys.exit(1)
                            elif boardid == "81":
                                logging.info(self.loginfo + "LLNLv1 board detected")
                            elif boardid == "84":
                                logging.info(self.loginfo + "LLNLv4 board detected")
                            else:
                                logging.info(
                                    self.loginfo + "unidentified board detected"
                                )
                            logging.info(self.loginfo + "connected to " + p)
                            port = p
                            ser.reset_input_buffer()
                            ser.reset_output_buffer()
                            break
                except Exception as e:
                    logging.error(self.logerr + "port identification: " + str(e))
        if port == "":
            if self.ca.port:
                logging.critical(
                    self.logcrit + "No usable board found at port " + str(self.ca.port)
                )
                sys.exit(1)
            else:
                logging.critical(self.logcrit + "No usable board found")
                sys.exit(1)
        self.port = port  # COM port to use for RS422 link
        self.ca.port = port[3:]  # re-extract port number from com name

        self._ser = serial.Serial(  # Class RS422
            port=self.port,
            baudrate=self.baud,
            parity=self.par,
00145             stopbits=self.stop,
00146             timeout=self.read_timeout,  # timeout for serial read
00147             bytesize=serial.EIGHTBITS,
00148         )
00149         self.payloadsize = (
00150             self.ca.sensor.width
00151             * self.ca.sensor.height
00152             * self.ca.sensor.nframes
00153             * self.ca.sensor.bytesperpixel
00154         )
00155         logging.debug(
00156             self.logdebug + "Payload size: " + str(self.payloadsize) + " bytes"
00157         )
00158         self._ser.flushInput()
00159         if not self._ser.is_open:
00160             logging.critical(self.logcrit + "Unable to open serial connection")
00161             sys.exit(1)
00162
```

6.10.3 Member Function Documentation

6.10.3.1 arm()

```
nsCamera.comms.RS422.RS422.arm (
                self,
                mode )
```

```
Puts camera into wait state for trigger. Mode determines source; defaults to
  'Hardware'

Args:
    mode:   'Software'|'S' activates software, disables hardware triggering
            'Hardware'|'H' activates hardware, disables software triggering
              Hardware is the default

Returns:
    tuple (error, response string)
```

Definition at line 319 of file RS422.py.

```
00319     def arm(self, mode):
00320         """
00321         Puts camera into wait state for trigger. Mode determines source; defaults to
00322           'Hardware'
00323
00324         Args:
00325             mode:   'Software'|'S' activates software, disables hardware triggering
00326                     'Hardware'|'H' activates hardware, disables software triggering
00327                       Hardware is the default
00328
00329         Returns:
00330             tuple (error, response string)
00331         """
00332         if not mode:
00333             mode = "Hardware"
00334         logging.info(self.loginfo + "arm")
00335         logging.debug(self.logdebug + "arming mode: " + str(mode))
00336         self.ca.clearStatus()
00337         self.ca.latchPots()
00338         err, resp = self.ca.startCapture(mode)
00339         if err:
00340             logging.error(self.logerr + "unable to arm camera")
00341         else:
00342             self.ca.armed = True
00343             self.skipError = True
00344         return err, resp
00345
```

6.10.3.2 closeDevice()

```
nsCamera.comms.RS422.RS422.closeDevice (
              self )
```

```
Close primary serial interface
```

Definition at line 496 of file RS422.py.

```
    def closeDevice(self):
        """
        Close primary serial interface
        """
        logging.debug(self.logdebug + "Closing RS422 connection")
        self._ser.close()


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.10.3.3 readFrames()

```
nsCamera.comms.RS422.RS422.readFrames (
              self,
              waitOnSRAM,
              timeout = 0,
              fast = False,
              columns = 1 )
```

```
Copies image data from board into numpy arrays.

Args:
    waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
      data
    timeout: passed to waitForSRAM; after this many seconds begin copying data
      irrespective of SRAM_READY status; 'zero' means wait indefinitely
      WARNING: If acquisition fails, the SRAM will not contain a current image,
        but the code will copy the data anyway
    fast: if False, parse and convert frames to numpy arrays; if True, return
      unprocessed text stream
    columns: 1 for single image per frame, 2 for separate hemisphere images

Returns:
    list of numpy arrays OR raw text stream
```

Definition at line 346 of file RS422.py.

```
    def readFrames(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays.

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
```

```
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            list of numpy arrays OR raw text stream

        """
        frames, _, _ = self.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

```

6.10.3.4 readoff()

```
nsCamera.comms.RS422.RS422.readoff (
                self,
                waitOnSRAM,
                timeout,
                fast,
                columns = 1 )
```

```
Copies image data from board into numpy arrays; returns data, length of data,
and error messages. Use 'readFrames()' unless you require this additional
information

Args:
    waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
      data
    timeout: passed to waitForSRAM; after this many seconds begin copying data
      irrespective of SRAM_READY status; 'zero' means wait indefinitely
      WARNING: If acquisition fails, the SRAM will not contain a current image,
        but the code will copy the data anyway
    fast: if False, parse and convert frames to numpy arrays; if True, return
      unprocessed text stream
    columns: 1 for single image per frame, 2 for separate hemisphere images

Returns:
    tuple (list of numpy arrays OR raw text stream, length of downloaded payload
      in bytes, payload error flag)
    NOTE: This reduces readoff by <1 second, so will have no noticeable impact
      when using RS422
```

Definition at line 368 of file RS422.py.

```
    def readoff(self, waitOnSRAM, timeout, fast, columns=1):
        """
        Copies image data from board into numpy arrays; returns data, length of data,
        and error messages. Use 'readFrames()' unless you require this additional
        information

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            tuple (list of numpy arrays OR raw text stream, length of downloaded payload
              in bytes, payload error flag)
            NOTE: This reduces readoff by <1 second, so will have no noticeable impact
              when using RS422
        """
        logging.info(self.loginfo + "readoff")
        logging.debug(
            self.logdebug
            + "readoff: waitonSRAM = "
            + str(waitOnSRAM)
            + "; timeout = "
            + str(timeout)
            + "; fast = "
            + str(fast)
        )
        errortemp = False

        # Wait for data to be ready on board, turns off error messaging
        # Skip wait only if explicitly tagged 'False' ('None' defaults to True)
        if waitOnSRAM is not False:
            logging.getLogger().setLevel(logging.CRITICAL)
            self.ca.waitForSRAM(timeout)
            logging.getLogger().setLevel(self.ca.verblevel)

        # Retrieve data
        err, rval = self.ca.readSRAM()
        if err:
            logging.error(self.logerr + "Error detected in readSRAM")
        time.sleep(0.3)
        logging.debug(self.logdebug + "readoff: first 64 chars: " + str(rval[0:64]))
        # extract only the read burst data. Remove header & CRC footer
        read_burst_data = rval[36:-4]

        # Payload size as string implied by provided parameters
        expectedlength = (
            4
            * (self.ca.sensor.lastframe - self.ca.sensor.firstframe + 1)
            * (self.ca.sensor.lastrow - self.ca.sensor.firstrow + 1)
            * self.ca.sensor.width
        )
        padding = expectedlength - len(read_burst_data)
        if padding:
            logging.warning(
                "{logwarn}readoff: Payload is shorter than expected."
                " Padding with '0's".format(logwarn=self.logwarn)
            )
            read_burst_data = read_burst_data.ljust(expectedlength, "0")

        if fast:
            return read_burst_data, len(read_burst_data) // 2, errortemp
        else:
            parsed = generateFrames(self.ca, read_burst_data, columns)
            return parsed, len(read_burst_data) // 2, errortemp
```

6.10.3.5 readSerial()

```
nsCamera.comms.RS422.RS422.readSerial (
                self,
                size,
                timeout = None )
```

```
Read bytes from the serial port. Does not verify packets.

Args:
   size: number of bytes to read
   timeout: serial timeout in sec

Returns:
   tuple (error string, string read from serial port)
```

Definition at line 465 of file RS422.py.

```
00465     def readSerial(self, size, timeout=None):
00466         """
00467         Read bytes from the serial port. Does not verify packets.
00468 
00469         Args:
00470            size: number of bytes to read
00471            timeout: serial timeout in sec
00472 
00473         Returns:
00474            tuple (error string, string read from serial port)
00475         """
00476         logging.debug(
00477             self.logdebug
00478             + "readSerial: size = "
00479             + str(size)
00480             + "; timeout = "
00481             + str(timeout)
00482         )
00483         err = ""
00484         if timeout:
00485             self._ser.timeout = timeout
00486         else:
00487             self._ser.timeout = self.read_timeout
00488         resp = self._ser.read(size)
00489         if len(resp) < 10:  # bytes
00490             err += (
00491                 self.logerr + "readSerial : packet too small: '" + bytes2str(resp) + "'"
00492             )
00493             logging.error(err)
00494         return err, bytes2str(resp)
00495 
```

6.10.3.6 sendCMD()

```
nsCamera.comms.RS422.RS422.sendCMD (
                self,
                pkt )
```

```
Submit packet and verify response packet. Recognizes readoff packet and adjusts
read size and timeout appropriately

Args:
    pkt: Packet object

Returns:
    tuple (error, response string)
```

Definition at line 170 of file RS422.py.

```
00170     def sendCMD(self, pkt):
00171         """
00172         Submit packet and verify response packet. Recognizes readoff packet and adjusts
00173         read size and timeout appropriately
00174 
00175         Args:
00176             pkt: Packet object
00177 
00178         Returns:
00179             tuple (error, response string)
00180         """
00181         pktStr = pkt.pktStr()
00182         logging.debug(self.logdebug + "sendCMD packet: " + str(pktStr))
00183         self._ser.flushInput()
00184         time.sleep(0.01)  # wait 10 ms in between flushing input and output buffers
00185         self._ser.flushOutput()
00186         self.ca.writeSerial(pktStr)
00187         err0 = ""
00188         err = ""
00189         resp = ""
00190         tries = 3  # TODO: make a function parameter?
00191 
```

```
        if (
            hasattr(self.ca, "board")
            and pktStr[4] == "0"
            and pktStr[5:8] == self.ca.board.registers["SRAM_CTL"]
        ):
            # download data payload
            logging.info(
                self.loginfo + "Payload size (bytes) = " + str(self.payloadsize)
            )
            crcresp0 = ""
            crcresp1 = ""
            smallresp = ""
            emptyResponse = False
            wrongSize = False
            # TODO: refactor payload error management to another method
            for i in range(tries):
                err, resp = self.readSerial(
                    self.payloadsize + 20, timeout=self.datatimeout
                )
                if err:
                    logging.error(
                        self.logerr + "sendCMD: read payload failed " + pktStr + err
                    )
                    self.ca.payloaderror = True
                else:
                    if not len(resp):
                        err0 = self.logerr + "sendCMD: empty response from board"
                        logging.error(err0)
                        emptyResponse = True
                        self.ca.payloaderror = True
                    elif len(resp) != 2 * (self.payloadsize + 20):
                        err0 = (
                            self.logerr
                            + "sendCMD: incorrect response; expected "
                            + str(self.payloadsize + 20)
                            + " bytes, received "
                            + str(len(resp) // 2)
                        )
                        logging.error(err0)
                        wrongSize = True
                        smallresp = resp
                        self.ca.payloaderror = True
                    elif not checkCRC(resp[4:20]):
                        err0 = (
                            self.logerr
                            + "sendCMD: "
                            + pktStr
                            + " - payload preface CRC fail"
                        )
                        logging.error(err0)
                        self.ca.payloaderror = True
                        crcresp1 = resp
                    elif not checkCRC(resp[24:]):
                        err0 = (
                            self.logerr + "sendCMD: " + pktStr + " - payload CRC fail"
                        )
                        logging.error(err0)
                        self.ca.payloaderror = True
                        crcresp0 = resp
                    err += err0
                time.sleep(5)
                if self.ca.payloaderror:
                    # keep best results over multiple tries; e.g., if first try is
                    #   bad CRC and second try is an incomplete payload, use the
                    #   first payload
                    if i == tries - 1:
                        if crcresp0:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire "
                                "CRC-confirmed payload after "
                                + str(tries)
                                + " attempts. Continuing with unconfirmed payload"
                            )
                            resp = crcresp0
                        elif crcresp1:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire "
                                "CRC-confirmed readoff after "
                                + str(tries)
                                + " attempts. Continuing with unconfirmed payload"
                            )
                            resp = crcresp1
                        elif wrongSize:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire complete "
                                "payload after "
                                + str(tries)
                                + " attempts. Dumping datastream to file."
                            )
                            resp = smallresp
                            self.ca.dumpNumpy(resp)
                        elif emptyResponse:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire any "
                                "payload after " + str(tries) + " attempts."
                            )
                    else:
                        logging.info(
                            self.loginfo + "Retrying download, attempt #" + str(i + 1)
                        )
                        err = ""
                        err0 = ""
                        self.ca.payloaderror = False
                        self.ca.writeSerial(pktStr)
                else:
                    logging.info(self.loginfo + "Download successful")
                    if self.ca.boardname == "llnl_v4":
                        # self.ca.setSubregister('SWACK','1')
                        pass
                    break

        else:
            # non-payload messages and workaround for initial setup before board object
            #   has been initialized
            time.sleep(0.03)
            self._ser.timeout = 0.02
            err, resp = self.readSerial(10)
            logging.debug(self.logdebug + "sendCMD response: " + str(resp))
            if err:
                logging.error(
                    self.logerr + "sendCMD: readSerial failed (regular packet) " + err
                )
            elif not checkCRC(resp[4:20]):
                err = self.logerr + "sendCMD- regular packet CRC fail: " + resp
                logging.error(err)
        return err, resp

```

6.10.3.7 serialClose()

nsCamera.comms.RS422.RS422.serialClose (
self)

Close serial interface

Definition at line 163 of file RS422.py.

```
00163     def serialClose(self):
00164         """
00165         Close serial interface
00166         """
00167         logging.debug(self.logdebug + "serialclose")
00168         self._ser.close()  # close serial interface COM port
00169
```

6.10.3.8 writeSerial()

nsCamera.comms.RS422.RS422.writeSerial (
self,
outstring,
timeout)

```
Transmit string to board

Args:
    outstring: string to write
    timeout: serial timeout in sec
Returns:
    integer length of string written to serial port
```

Definition at line 440 of file RS422.py.

```
    def writeSerial(self, outstring, timeout):
        """
        Transmit string to board

        Args:
            outstring: string to write
            timeout: serial timeout in sec
        Returns:
            integer length of string written to serial port
        """
        logging.debug(
            self.logdebug
            + "writeSerial: outstring = "
            + str(outstring)
            + "; timeout = "
            + str(timeout)
        )
        if timeout:
            self._ser.timeout = timeout
        else:
            self._ser.timeout = self.write_timeout
        lengthwritten = self._ser.write(str2bytes(outstring))
        self._ser.timeout = self.read_timeout  # reset if changed above
        return lengthwritten

```

6.10.4 Member Data Documentation

6.10.4.1 _ser

```
nsCamera.comms.RS422.RS422._ser  [protected]
```

Definition at line 141 of file RS422.py.

6.10.4.2 baud

```
nsCamera.comms.RS422.RS422.baud
```

Definition at line 73 of file RS422.py.

6.10.4.3 ca

```
nsCamera.comms.RS422.RS422.ca
```

Definition at line 56 of file RS422.py.

6.10.4.4 datatimeout

`nsCamera.comms.RS422.RS422.datatimeout`

Definition at line 79 of file RS422.py.

6.10.4.5 logcrit

`nsCamera.comms.RS422.RS422.logcrit`

Definition at line 57 of file RS422.py.

6.10.4.6 logdebug

`nsCamera.comms.RS422.RS422.logdebug`

Definition at line 61 of file RS422.py.

6.10.4.7 logerr

`nsCamera.comms.RS422.RS422.logerr`

Definition at line 58 of file RS422.py.

6.10.4.8 loginfo

`nsCamera.comms.RS422.RS422.loginfo`

Definition at line 60 of file RS422.py.

6.10.4.9 logwarn

`nsCamera.comms.RS422.RS422.logwarn`

Definition at line 59 of file RS422.py.

6.10.4.10 mode

`nsCamera.comms.RS422.RS422.mode`

Definition at line 72 of file RS422.py.

6.10.4.11 par

```
nsCamera.comms.RS422.RS422.par
```

Definition at line 74 of file RS422.py.

6.10.4.12 payloadsize

```
nsCamera.comms.RS422.RS422.payloadsize
```

Definition at line 149 of file RS422.py.

6.10.4.13 port

```
nsCamera.comms.RS422.RS422.port
```

Definition at line 138 of file RS422.py.

6.10.4.14 PY3

```
nsCamera.comms.RS422.RS422.PY3
```

Definition at line 83 of file RS422.py.

6.10.4.15 read_timeout

```
nsCamera.comms.RS422.RS422.read_timeout
```

Definition at line 76 of file RS422.py.

6.10.4.16 skipError

```
nsCamera.comms.RS422.RS422.skipError
```

Definition at line 84 of file RS422.py.

6.10.4.17 stop

```
nsCamera.comms.RS422.RS422.stop
```

Definition at line 75 of file RS422.py.

6.10.4.18 write_timeout

```
nsCamera.comms.RS422.RS422.write_timeout
```

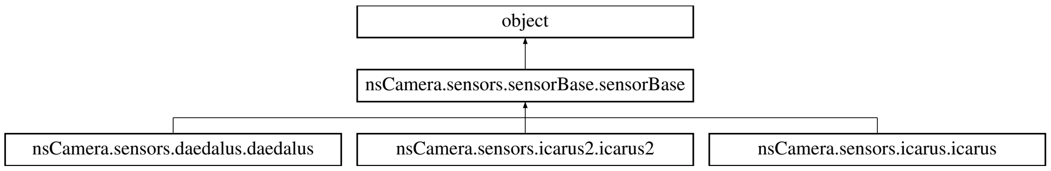
Definition at line 77 of file RS422.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/comms/RS422.py

6.11 nsCamera.sensors.sensorBase.sensorBase Class Reference

Inheritance diagram for nsCamera.sensors.sensorBase.sensorBase:


object
nsCamera.sensors.sensorBase.sensorBase
nsCamera.sensors.daedalus.daedalus
nsCamera.sensors.icarus2.icarus2
nsCamera.sensors.icarus.icarus

Public Member Functions

- __init__ (self, camassem)
- init_board_specific (self)
- checkSensorVoltStat (self)
- setInterlacing (self, ifactor)
- setHighFullWell (self, flag)
- setZeroDeadTime (self, flag)
- setTriggerDelay (self, delay)
- setPhiDelay (self, delay)
- setExtClk (self, delay)
- setTiming (self, side="AB", sequence=None, delay=0)
- setArbTiming (self, side="AB", sequence=None)
- getTiming (self, side, actual)
- setManualShutters (self, timing=None)
- setManualTiming (self, timing=None)
- getManualTiming (self)
- getSensTemp (self, scale=None, offset=None, slope=None, dec=None)
- selectOscillator (self, osc=None)
- parseReadoff (self, frames, columns)
- getSensorStatus (self)
- reportStatusSensor (self, statusbits, statusbits2)

Public Attributes

- ca
- logcrit
- logerr
- logwarn
- loginfo
- logdebug
- fpganumID
- sensfam

6.11.1 Detailed Description

```
Base class for sensors. 'Virtual' methods below default to Icarus behavior.
daedalus.py overrides these methods as necessary
```

Definition at line 25 of file sensorBase.py.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 __init__()

```
nsCamera.sensors.sensorBase.sensorBase.__init__ (
                self,
                camassem )
```

Reimplemented in nsCamera.sensors.daedalus.daedalus, nsCamera.sensors.icarus.icarus, and nsCamera.sensors.icarus2.icarus2.

Definition at line 31 of file sensorBase.py.

```
    def __init__(self, camassem):
        self.ca = camassem
        # skip board settings if no board object exists
        if hasattr(self.ca, "board"):
            self.init_board_specific()

        (
            self.logcrit,
            self.logerr,
            self.logwarn,
            self.loginfo,
            self.logdebug,
        ) = makeLogLabels(self.ca.logtag, self.loglabel)

        # skip assignment if no comms object exists
        if hasattr(self.ca, "comms"):
            self.ca.comms.payloadsize = (
                self.width * self.height * self.nframes * self.bytesperpixel
            )

        logging.info(self.loginfo + "Initializing sensor object")

```

6.11.3 Member Function Documentation

6.11.3.1 checkSensorVoltStat()

```
nsCamera.sensors.sensorBase.sensorBase.checkSensorVoltStat (
              self )
```

```
Checks register tied to sensor select jumpers to confirm match with sensor
object

Returns:
    boolean, True if jumpers select for Icarus sensor
```

Reimplemented in nsCamera.sensors.icarus.icarus.

Definition at line 64 of file sensorBase.py.

```
    def checkSensorVoltStat(self):
        """
        Checks register tied to sensor select jumpers to confirm match with sensor
        object

        Returns:
            boolean, True if jumpers select for Icarus sensor
        """
        logging.debug(self.logdebug + "checkSensorVoltStat")
        err, status = self.ca.getSubregister(self.detect)
        if err:
            logging.error(self.logerr + "Unable to confirm sensor status")
            return False
        if not int(status):
            logging.error(self.logerr + self.sensfam + " sensor not detected")
            return False
        return True

```

6.11.3.2 getManualTiming()

```
nsCamera.sensors.sensorBase.sensorBase.getManualTiming (
              self )
```

```
Read off manual shutter timing settings
Overridden in daedalus.py
Returns:
    list of 2 lists of timing from A and B sides, respectively
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 611 of file sensorBase.py.

```
    def getManualTiming(self):
        """
        Read off manual shutter timing settings
        Overridden in daedalus.py
        Returns:
            list of 2 lists of timing from A and B sides, respectively
        """
        aside = []
        bside = []
        for reg in [
            "W0_INTEGRATION",
            "W0_INTERFRAME",
            "W1_INTEGRATION",
            "W1_INTERFRAME",
            "W2_INTEGRATION",
            "W2_INTERFRAME",
            "W3_INTEGRATION",
        ]:
            _, reghex = self.ca.getRegister(reg)
            aside.append(25 * int(reghex, 16))
        for reg in [
            "W0_INTEGRATION_B",
            "W0_INTERFRAME_B",
            "W1_INTEGRATION_B",
            "W1_INTERFRAME_B",
            "W2_INTEGRATION_B",
            "W2_INTERFRAME_B",
            "W3_INTEGRATION_B",
        ]:
            _, reghex = self.ca.getRegister(reg)
            bside.append(25 * int(reghex, 16))
        return [aside, bside]

```

6.11.3.3 getSensorStatus()

```
nsCamera.sensors.sensorBase.sensorBase.getSensorStatus (
                self )
```

```
Wrapper for reportSensorStatus so that the user doesn't have to query statusbits
```

Definition at line 691 of file sensorBase.py.

```
00691     def getSensorStatus(self):
00692         """
00693         Wrapper for reportSensorStatus so that the user doesn't have to query statusbits
00694         """
00695         sb1 = self.ca.board.checkstatus()
00696         sb2 = self.ca.board.checkstatus2()
00697         self.reportStatusSensor(sb1, sb2)
00698
```

6.11.3.4 getSensTemp()

```
nsCamera.sensors.sensorBase.sensorBase.getSensTemp (
                self,
                scale = None,
                offset = None,
                slope = None,
                dec = None )
```

```
Virtual method (Temperature sensor is not present on Icarus sensors). Returns 0.
Overridden by Daedalus method
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 644 of file sensorBase.py.

```
00644     def getSensTemp(self, scale=None, offset=None, slope=None, dec=None):
00645         """
00646         Virtual method (Temperature sensor is not present on Icarus sensors). Returns 0.
00647         Overridden by Daedalus method
00648         """
00649         return 0
00650
```

6.11.3.5 getTiming()

```
nsCamera.sensors.sensorBase.sensorBase.getTiming (
              self,
              side,
              actual )
```

```
actual = True: returns actual high speed intervals that will be generated by the
            FPGA as list
         False: Returns high speed timing settings as set by setTiming. Assumes
            that timing was set via the setTiming method--it will not accurately
            report arbitrary timings set by direct register sets or manual
            shutter control

Args:
    side: Hemisphere ’A’ or ’B’
    actual: False: return HST settings
            True: calculate and return actual HST behavior

Returns:
    actual= True: list of shutter intervals;
                icarus: [delay, open0, closed0, open1, closed1, open2, closed2,
                          open3]
                daedalus: [delay, open0, closed0, open1, closed1, open2]
            False: tuple (hemisphere label,
                          ’open shutter’ in ns,
                          ’closed shutter’ in ns,
                          initial delay in ns)
```

Definition at line 434 of file sensorBase.py.

```
    def getTiming(self, side, actual):
        """
        actual = True: returns actual high speed intervals that will be generated by the
                    FPGA as list
                 False: Returns high speed timing settings as set by setTiming. Assumes
                    that timing was set via the setTiming method--it will not accurately
                    report arbitrary timings set by direct register sets or manual
                    shutter control

        Args:
            side: Hemisphere ’A’ or ’B’
            actual: False: return HST settings
                    True: calculate and return actual HST behavior

        Returns:
            actual= True: list of shutter intervals;
                        icarus: [delay, open0, closed0, open1, closed1, open2, closed2,
                                  open3]
                        daedalus: [delay, open0, closed0, open1, closed1, open2]
                    False: tuple (hemisphere label,
                                  ’open shutter’ in ns,
                                  ’closed shutter’ in ns,
                                  initial delay in ns)

        """
        logging.info("{}getTiming".format(self.loginfo))
        if side is None:
            side = "A"
        logging.info(self.loginfo + "get timing, side " + side.upper())
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            logging.error(
                self.logerr
                + "Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            return "", 0, 0, 0
        err, lowpart = self.ca.getRegister(lowreg)
        err1, highpart = self.ca.getRegister(highreg)
        if err or err1:
            logging.error(
                self.logerr + "Unable to retrieve timing setting (getTiming), "
                "returning zeroes "
            )
            return side.upper(), 0, 0, 0
        full40hex = highpart[-2:] + lowpart.zfill(8)
        full40bin = "{0:0=40b}".format(int(full40hex, 16))
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        if actual:
            if full40bin == "0" * 40:  # all-zero timing
                if self.fpganumID == 2:
                    times = [0] * 6
                else:
                    times = [0] * 8
            else:
                full160 = 4 * full40bin
                gblist = [[k, len(list(g))] for k, g in itertools.groupby(full160)]
                if self.fpganumID == 2:
                    times = [int(x[1]) for x in gblist[:-7:-1]]
                else:
                    times = [int(x[1]) for x in gblist[:-9:-1]]
                times[0] = times[0] - 1
            if self.ca.sensorname == "icarus":
                # get timing for frames 1 and 2, keep delay as offset
                # TODO: should this give a 'correct' offset from frame 0?
                times12 = [times[0]] + times[3:6]
                return times12
            return times
        else:
            if full40bin == "0" * 40:  # all-zero timing
                timeon, timeoff, delay = (0, 0, 0)
            else:
                gblist = [[k, len(list(g))] for k, g in itertools.groupby(full40bin)]
                delay = gblist[-1][1] - 1
                timeon = gblist[-2][1]

                if self.ca.sensorname == "icarus":
                    if len(gblist) == 2:  # 39,1 corner case
                        timeoff = 1
                    elif len(gblist) == 3:  # sequence fits only once
                        timeoff = 40 - timeon
                    else:
                        timeoff = gblist[-3][1]
                else:
                    if len(gblist) < self.nframes:  # sequence fits only once
                        timeoff = 40 - timeon
                    else:
                        # TODO: confirm '-3' works for daedalus
                        timeoff = gblist[-3][1]
            return side.upper(), timeon, timeoff, delay

```

6.11.3.6 init_board_specific()

```
nsCamera.sensors.sensorBase.sensorBase.init_board_specific (
            self )
```

```
Initialize aliases and subregisters specific to the current board and sensor.
```

Definition at line 53 of file sensorBase.py.

```
    def init_board_specific(self):
        """Initialize aliases and subregisters specific to the current board and sensor."""

        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            self.ca.board.subreg_aliases = self.ca.board.icarus_subreg_aliases
            self.ca.board.monitor_controls = self.ca.board.icarus_monitor_controls
        else:
            self.ca.board.subreg_aliases = self.ca.board.daedalus_subreg_aliases
            self.ca.board.monitor_controls = self.ca.board.daedalus_monitor_controls

```

6.11.3.7 parseReadoff()

```
nsCamera.sensors.sensorBase.sensorBase.parseReadoff (
              self,
              frames,
              columns )
```

```
Virtual method (Order parsing is unnecessary for Icarus, continue to hemisphere
  parsing.)
Overridden by Daedalus method
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 683 of file sensorBase.py.

```
    def parseReadoff(self, frames, columns):
        """
        Virtual method (Order parsing is unnecessary for Icarus, continue to hemisphere
          parsing.)
        Overridden by Daedalus method
        """
        return self.ca.partition(frames, columns)

```

6.11.3.8 reportStatusSensor()

```
nsCamera.sensors.sensorBase.sensorBase.reportStatusSensor (
              self,
              statusbits,
              statusbits2 )
```

```
Print status messages from sensor-specific bits of status register, default for
  Icarus family sensors
Args:
    statusbits: result of checkStatus()
    statusbits2: result of checkStatus2()
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 699 of file sensorBase.py.

```
    def reportStatusSensor(self, statusbits, statusbits2):
        """
        Print status messages from sensor-specific bits of status register, default for
          Icarus family sensors
        Args:
            statusbits: result of checkStatus()
            statusbits2: result of checkStatus2()
        """
        if int(statusbits[3]):
            print(self.loginfo + "W3_Top_A_Edge1 detected")
        if int(statusbits[4]):
            print(self.loginfo + "W3_Top_B_Edge1 detected")
        if int(statusbits[12]):
            print(self.loginfo + "HST_All_W_En detected")
        if self.ca.boardname == "llnl_v4" and int(statusbits2[5]):
            print(self.loginfo + "PDBIAS Unready")


# TODO: add function to control TIME_ROW_DCD delay

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.11.3.9 selectOscillator()

```
nsCamera.sensors.sensorBase.sensorBase.selectOscillator (
              self,
              osc = None )
```

```
Selects oscillator to control sensor timing
Overridden in daedalus.py
Args:
    osc: 'relaxation'|'ring'|'ringnoosc'|'external', defaults to relaxation

Returns:
    error message as string
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 651 of file sensorBase.py.

```
    def selectOscillator(self, osc=None):
        """
        Selects oscillator to control sensor timing
        Overridden in daedalus.py
        Args:
            osc: 'relaxation'|'ring'|'ringnoosc'|'external', defaults to relaxation

        Returns:
            error message as string
        """
        logging.info(self.loginfo + "selectOscillator; osc = " + str(osc))
        if osc is None:
            osc = "rel"
        osc = str(osc)
        if osc.upper()[:3] == "REL":
            payload = "00"
        elif osc.upper()[:3] == "RIN":
            if "NO" in osc.upper() or "0" in osc:
                payload = "10"
            else:
                payload = "01"
        elif osc.lower()[:3] in ["ext", "phi"]:
            payload = "11"
        else:
            err = (
                self.logerr + "selectOscillator: invalid parameter supplied. "
                "Oscillator selection is unchanged."
            )
            logging.error(err)
            return err
        self.ca.setSubregister("OSC_SELECT", payload)

```

6.11.3.10 setArbTiming()

```
nsCamera.sensors.sensorBase.sensorBase.setArbTiming (
              self,
              side = "AB",
              sequence = None )
```

```
Set arbitrary high-speed timing sequence.
Args:
    side: Hemisphere 'A', 'B', 'AB'
    sequence: list of arbitrary timing intervals, beginning with initial delay.
      The conventional timing (3,2) with delay = 0 would be represented by
      [0,3,2,3,2,3,2,3] on icarus devices, [0,3,2,3,2,3] on daedalus. If used
      for interlacing or ZDT, you should populate the entire 40-bit register,
      e.g., [0,3,2,3,2,3,2,3,2,3,2,3,2,3,2,3,2]

    *NOTE* Icarus sensors generally cannot use 1 ns timing, so should use at
      least 2 ns for frames 2 and 3 integration and interframe times (an initial
      delay of only 1 ns is acceptable)

    *NOTE* although the Icarus model 1 only images the middle two frames, timing
      entries must be provided for all four frames; to implement frame 1 open
      for X ns, shutter closed for Y ns, and frame 2 open for Z ns, use the
      sequence [0,1,1,X,Y,Z,1,1]

    *WARNING* arbitrary timings will not be restored after a board power cycle

Returns:
    list: Actual timing results
```

Definition at line 308 of file sensorBase.py.

```
    def setArbTiming(self, side="AB", sequence=None):
        """
        Set arbitrary high-speed timing sequence.
        Args:
            side: Hemisphere 'A', 'B', 'AB'
            sequence: list of arbitrary timing intervals, beginning with initial delay.
              The conventional timing (3,2) with delay = 0 would be represented by
              [0,3,2,3,2,3,2,3] on icarus devices, [0,3,2,3,2,3] on daedalus. If used
              for interlacing or ZDT, you should populate the entire 40-bit register,
              e.g., [0,3,2,3,2,3,2,3,2,3,2,3,2,3,2,3,2]

            *NOTE* Icarus sensors generally cannot use 1 ns timing, so should use at
              least 2 ns for frames 2 and 3 integration and interframe times (an initial
              delay of only 1 ns is acceptable)

            *NOTE* although the Icarus model 1 only images the middle two frames, timing
              entries must be provided for all four frames; to implement frame 1 open
              for X ns, shutter closed for Y ns, and frame 2 open for Z ns, use the
              sequence [0,1,1,X,Y,Z,1,1]

            *WARNING* arbitrary timings will not be restored after a board power cycle

        Returns:
            list: Actual timing results
        """
        logging.info(
            "{}setArbTiming; side={}, sequence={}".format(self.loginfo, side, sequence)
        )
        if sequence is None:
            if self.sensfam == "Daedalus":
                sequence = [0, 2, 3, 4, 5, 6]
            else:
                sequence = [0, 2, 3, 4, 5, 6, 7, 8]
        logging.info(
            self.loginfo + "HST side " + side.upper() + " (arbitrary): " + str(sequence)
        )
        if side.upper() == "AB":
            err1, _ = self.setArbTiming(side="A", sequence=sequence)
            err2, actual = self.setArbTiming(side="B", sequence=sequence)
            return err1 + err2, actual
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            err = (
                self.logerr
                + "Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            logging.error("{}setArbTiming: {}".format(self.logerr, err))
            return err, "0000000000"

        full40 = [0] * 40
        bitlist = []
        flag = 0  # similar to setTiming, but starts with delay

        for a in sequence:
            add = [flag] * a
            bitlist += add
            if flag:
                flag = 0
            else:
                flag = 1

        logging.debug(self.logdebug + "bitlist = " + str(bitlist))
        reversedlist = bitlist[39::-1]
        full40[-(len(reversedlist) + 1) : -1] = reversedlist
        full40bin = "".join(str(x) for x in full40)
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        full40hex = "%x" % int(full40bin, 2)
        logging.debug(self.logdebug + "full40hex = " + str(full40hex))
        highpart = full40hex[-10:-8].zfill(8)
        lowpart = full40hex[-8:].zfill(8)
        self.ca.setRegister(lowreg, lowpart)
        self.ca.setRegister(highreg, highpart)
        # deactivates manual shutter mode if previously engaged
        self.ca.setSubregister("MANSHUT_MODE", "0")
        self.ca.setSubregister("HST_MODE", "1")
        actual = self.getTiming(side, actual=True)
        f0delay = sequence[1] + sequence[2]

        if self.ca.sensorname == "icarus":
            if actual != sequence[:1] + sequence[3:6]:
                logging.warning(
                    self.logwarn + "Due to sequence length and use of the Icarus model "
                    "1 sensor, the actual timing sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0] + f0delay)
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
            else:
                logging.warning(
                    self.logwarn + "Due to use of the Icarus model 1 sensor, the actual"
                    " timing sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0] + f0delay)
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        else:
            if actual != sequence:
                logging.warning(
                    self.logwarn + "Due to sequence length, actual timing sequence "
                    "for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0])
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        return "", actual

```

6.11.3.11 setExtClk()

```
nsCamera.sensors.sensorBase.sensorBase.setExtClk (
               self,
               delay )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 136 of file sensorBase.py.

```
    def setExtClk(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "External Phi Clock is not supported by Icarus sensors. "
            )

```

6.11.3.12 setHighFullWell()

```
nsCamera.sensors.sensorBase.sensorBase.setHighFullWell (
              self,
              flag )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 96 of file sensorBase.py.

```
    def setHighFullWell(self, flag):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if flag:
            logging.warning(
                self.logwarn + "HighFullWell mode is not supported by Icarus sensors. "
            )

```

6.11.3.13 setInterlacing()

```
nsCamera.sensors.sensorBase.sensorBase.setInterlacing (
              self,
              ifactor )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py

Returns:
    integer 0
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 82 of file sensorBase.py.

```
    def setInterlacing(self, ifactor):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py

        Returns:
            integer 0
        """
        if ifactor:
            logging.warning(
                self.logwarn + "Interlacing is not supported by Icarus sensors. "
            )
        return 0

```

6.11.3.14 setManualShutters()

```
nsCamera.sensors.sensorBase.sensorBase.setManualShutters (
              self,
              timing = None )
```

```
Legacy alias for setManualTiming()
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 531 of file sensorBase.py.

```
    def setManualShutters(self, timing=None):
        """
        Legacy alias for setManualTiming()
        """
        self.setManualTiming(timing)

```

6.11.3.15 setManualTiming()

```
nsCamera.sensors.sensorBase.sensorBase.setManualTiming (
              self,
              timing = None )
```

```
Manual shutter timing, seven intervals to assign to both hemispheres, e.g.,
  [(100,150,100,150,100,150,100)] for frame 0 open for 100 ns, an interframe
  pause of 50 ns,frame 1 open for 100 ns, etc.
Provide two sets of seven intervals, e.g., [(100,150,100,150,100,150,100),
  (200,250,200,250,200,250,200)] to program the A and B hemispheres
  independently

Overridden in daedalus.py

The timing list is flattened before processing; the suggested tuple structure is
  just for clarity (first tuple is A, second is B) and is optional.

The actual timing is rounded down to the nearest multiple of 25 ns. (Each
  count = 25 ns. e.g., a request for 140 ns rounds down to a count of ’5’,
  which corresponds to 125 ns))
    - Minimum timing is 75 ns
    - Maximum is 25 * 2^30 ns (approximately 27 seconds)

Args:
    timing: 7- or 14-element list (substructure optional) in nanoseconds

Returns:
    tuple (error string, response string from final message)
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 537 of file sensorBase.py.

```
    def setManualTiming(self, timing=None):
        """
        Manual shutter timing, seven intervals to assign to both hemispheres, e.g.,
          [(100,150,100,150,100,150,100)] for frame 0 open for 100 ns, an interframe
          pause of 50 ns,frame 1 open for 100 ns, etc.
        Provide two sets of seven intervals, e.g., [(100,150,100,150,100,150,100),
          (200,250,200,250,200,250,200)] to program the A and B hemispheres
          independently

        Overridden in daedalus.py

        The timing list is flattened before processing; the suggested tuple structure is
          just for clarity (first tuple is A, second is B) and is optional.

        The actual timing is rounded down to the nearest multiple of 25 ns. (Each
          count = 25 ns. e.g., a request for 140 ns rounds down to a count of '5',
          which corresponds to 125 ns))
            - Minimum timing is 75 ns
            - Maximum is 25 * 2^30 ns (approximately 27 seconds)

        Args:
            timing: 7- or 14-element list (substructure optional) in nanoseconds

        Returns:
            tuple (error string, response string from final message)
        """
        if timing is None:
            logging.info(
                self.loginfo
                + "No manual timing setting provided, defaulting to (100, 150, 100, "
                " 150, 100, 150, 100) for both hemispheres"
            )
            timing = [
                (100, 150, 100, 150, 100, 150, 100),
                (100, 150, 100, 150, 100, 150, 100),
            ]
        logging.info(self.loginfo + "Manual shutter sequence: " + str(timing))
        flattened = flattenlist(timing)
        if len(flattened) == 7:
            flattened = 2 * flattened
        if (
            len(flattened) != 14
            or not all(isinstance(x, numbers.Real) for x in flattened)
            or not all(x >= 75 for x in flattened)
            or not all(x <= 26843545600 for x in flattened)
        ):
            err = self.logerr + "Invalid manual shutter timing list: " + str(timing)
            logging.error(err + "; timing settings unchanged")
            return err, "00000000"

        timecounts = [int(a // 25) for a in flattened]
        self.ca.sensmanual = timing
        self.ca.senstiming = {}  # clear HST settings from ca object

        control_messages = [
            ("W0_INTEGRATION", "{0:#0{1}x}".format(timecounts[0], 10)[2:10]),
            ("W0_INTERFRAME", "{0:#0{1}x}".format(timecounts[1], 10)[2:10]),
            ("W1_INTEGRATION", "{0:#0{1}x}".format(timecounts[2], 10)[2:10]),
            ("W1_INTERFRAME", "{0:#0{1}x}".format(timecounts[3], 10)[2:10]),
            ("W2_INTEGRATION", "{0:#0{1}x}".format(timecounts[4], 10)[2:10]),
            ("W2_INTERFRAME", "{0:#0{1}x}".format(timecounts[5], 10)[2:10]),
            ("W3_INTEGRATION", "{0:#0{1}x}".format(timecounts[6], 10)[2:10]),
            ("W0_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[7], 10)[2:10]),
            ("W0_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[8], 10)[2:10]),
            ("W1_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[9], 10)[2:10]),
            ("W1_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[10], 10)[2:10]),
            ("W2_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[11], 10)[2:10]),
            ("W2_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[12], 10)[2:10]),
            ("W3_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[13], 10)[2:10]),
            ("HST_MODE", "0"),
            ("MANSHUT_MODE", "1"),
        ]
        return self.ca.submitMessages(control_messages, " setManualShutters: ")

```

6.11.3.16 setPhiDelay()

```
nsCamera.sensors.sensorBase.sensorBase.setPhiDelay (
              self,
              delay )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 126 of file sensorBase.py.

```
    def setPhiDelay(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "Phi Delay is not supported by Icarus sensors. "
            )

```

6.11.3.17 setTiming()

```
nsCamera.sensors.sensorBase.sensorBase.setTiming (
               self,
               side = "AB",
               sequence = None,
               delay = 0 )
```

```
Sets timing registers based on 'sequence.' Requesting (0,0) timing will clear the
  timing register.
*WARNING* if the entire sequence does not fit into the 40-bit register space,
  then the actual timings generated may differ from those requested. If the
  timing sequence fits only once into the register space (i.e., for a single
  frame, open + closed > 20 ns), then the actual timing will be (n, 40-n),
  irrespective of the setting of second parameter, e.g. (35,1) will actually
  result in (35,5) timing.
*NOTE* Icarus sensors generally cannot use 1 ns timing, so all values for these
  devices (besides the delay) should be at least 2 ns

Args:
    side: Hemisphere 'A', 'B', 'AB'
    sequence: two-element tuple of timing durations in ns, e.g., '(5,2)'
    delay: initial delay in ns (1 ns delay is acceptable)

Returns:
    tuple (error string, 10-character hexadecimal representation of timing
      sequence)
```

Definition at line 149 of file sensorBase.py.

```
    def setTiming(self, side="AB", sequence=None, delay=0):
        """
        Sets timing registers based on 'sequence.' Requesting (0,0) timing will clear the
          timing register.
        *WARNING* if the entire sequence does not fit into the 40-bit register space,
          then the actual timings generated may differ from those requested. If the
          timing sequence fits only once into the register space (i.e., for a single
          frame, open + closed > 20 ns), then the actual timing will be (n, 40-n),
          irrespective of the setting of second parameter, e.g. (35,1) will actually
          result in (35,5) timing.
        *NOTE* Icarus sensors generally cannot use 1 ns timing, so all values for these
          devices (besides the delay) should be at least 2 ns

        Args:
            side: Hemisphere 'A', 'B', 'AB'
            sequence: two-element tuple of timing durations in ns, e.g., '(5,2)'
            delay: initial delay in ns (1 ns delay is acceptable)

        Returns:
            tuple (error string, 10-character hexadecimal representation of timing
              sequence)
        """
        logging.info(
            "{}setTiming; side={}, sequence={}, delay={}".format(
                self.loginfo, side, sequence, delay
            )
        )
        if sequence is None:
            sequence = (3, 2)
        if delay is None:
            delay = 0
        logging.info(
            self.loginfo
            + "HST side "
            + side.upper()
            + ": "
            + str(sequence)
            + "; delay = "
            + str(delay)
        )
        err = ""
        if len(sequence) != 2:
            err = (
                self.logerr
                + "Invalid sequence setting for side: "
                + side
                + "; timing settings are unchanged"
            )
            logging.error(err)
            return err, "0000000000"
        if side.upper() == "AB":
            err1, _ = self.setTiming(side="A", sequence=sequence, delay=delay)
            err2, full40hex = self.setTiming(side="B", sequence=sequence, delay=delay)
            return err1 + err2, full40hex
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            err = (
                self.logerr
                + "setTiming: Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            logging.error(err)
            return err, "0000000000"
        if (sequence[0] + sequence[1]) + delay > 40:
            err = (
                self.logerr
                + "setTiming:  Timing sequence is too long to be implemented; "
                "timing settings unchanged "
            )
            logging.error(err)
            return err, "0000000000"

        self.ca.senstiming[side.upper()] = (sequence, delay)
        self.ca.sensmanual = []  # clear manual settings from ca

        full40 = [0] * 40
        bitlist = []
        flag = 1
        sequence = sequence[:2]  # TODO: is this redundant?
        for a in sequence:
            add = [flag] * a
            bitlist += add
            if flag:
                flag = 0
            else:
                flag = 1
        # automatically truncates sequence to 39 characters
        logging.debug(self.logdebug + "bitlist = " + str(bitlist))
        if bitlist:  # skip this if timing is [0,0]
            reversedlist = bitlist[39::-1]
            trunclist = reversedlist[:]
            while trunclist[0] == 0:
                trunclist.pop(0)
            # fullrepeat counts open/closed cycles, doesn't include final frame
            fullrepeats = (40 - len(trunclist) - delay) // len(reversedlist)
            logging.debug(self.logdebug + "fullrepeats = " + str(fullrepeats))
            # Pattern from sequence repeated to fit inside 40 bits
            repeated = trunclist + reversedlist * fullrepeats
            full40[-(len(repeated) + delay + 1) : -(delay + 1)] = repeated
        else:
            logging.warning(self.logwarn + "setTiming: all-zero timing supplied")
            fullrepeats = self.nframes
        full40bin = "".join(str(x) for x in full40)
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        full40hex = "%x" % int(full40bin, 2)
        logging.debug(self.logdebug + "full40hex = " + str(full40hex))
        highpart = full40hex[-10:-8].zfill(8)
        lowpart = full40hex[-8:].zfill(8)
        err0, _ = self.ca.setRegister(lowreg, lowpart)
        err1, _ = self.ca.setRegister(highreg, highpart)
        err2, _ = self.ca.setSubregister("MANSHUT_MODE", "0")
        err3, _ = self.ca.setSubregister("HST_MODE", "1")
        err = err0 + err1 + err2 + err3
        if err:
            logging.error(
                self.logerr + "setTiming: Timing may not have been set correctly"
            )
        if fullrepeats < self.nframes - 1:
            actual = self.getTiming(side, actual=True)
            if self.fpganumID == 2:
                expected = [delay] + 2 * list(sequence) + [sequence[0]]
            else:
                expected = [delay] + 3 * list(sequence) + [sequence[0]]
            if actual != expected:
                logging.warning(
                    self.logwarn
                    + "setTiming: Due to sequence length"
                    + self.specwarn
                    + ", the actual timing "
                    "sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0])
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        elif self.ca.sensorname == "icarus":
            f0delay = sequence[0] + sequence[1]
            logging.warning(
                self.logwarn + "setTiming: Due to use of the Icarus model 1 sensor, the"
                " initial delay for side "
                + side
                + " will actually be "
                + str(delay + f0delay)
                + " nanoseconds"
            )
        return err, full40hex

```

6.11.3.18 setTriggerDelay()

```
nsCamera.sensors.sensorBase.sensorBase.setTriggerDelay (
               self,
               delay )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 116 of file sensorBase.py.

```
    def setTriggerDelay(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "Trigger Delay is not supported by Icarus sensors. "
            )

```

6.11.3.19 setZeroDeadTime()

```
nsCamera.sensors.sensorBase.sensorBase.setZeroDeadTime (
              self,
              flag )
```

```
Virtual function; feature is not implemented on Icarus
Overridden in daedalus.py
```

Reimplemented in nsCamera.sensors.daedalus.daedalus.

Definition at line 106 of file sensorBase.py.

```
    def setZeroDeadTime(self, flag):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if flag:
            logging.warning(
                self.logwarn + "ZeroDeadTime mode is not supported by Icarus sensors. "
            )

```

6.11.4 Member Data Documentation

6.11.4.1 ca

```
nsCamera.sensors.sensorBase.sensorBase.ca
```

Definition at line 32 of file sensorBase.py.

6.11.4.2 fpganumID

```
nsCamera.sensors.sensorBase.sensorBase.fpganumID
```

Definition at line 274 of file sensorBase.py.

6.11.4.3 logcrit

```
nsCamera.sensors.sensorBase.sensorBase.logcrit
```

Definition at line 38 of file sensorBase.py.

6.11.4.4 logdebug

```
nsCamera.sensors.sensorBase.sensorBase.logdebug
```

Definition at line 42 of file sensorBase.py.

6.11.4.5 logerr

```
nsCamera.sensors.sensorBase.sensorBase.logerr
```

Definition at line 39 of file sensorBase.py.

6.11.4.6 loginfo

```
nsCamera.sensors.sensorBase.sensorBase.loginfo
```

Definition at line 41 of file sensorBase.py.

6.11.4.7 logwarn

```
nsCamera.sensors.sensorBase.sensorBase.logwarn
```

Definition at line 40 of file sensorBase.py.

6.11.4.8 sensfam

```
nsCamera.sensors.sensorBase.sensorBase.sensfam
```

Definition at line 337 of file sensorBase.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/sensors/sensorBase.py

6.12 nsCamera.utils.Subregister.SubRegister Class Reference

Public Member Functions

- __init__ (self, board, name, register, start_bit=31, width=8, writable=False, value=255, minV=0, maxV=5)

Public Attributes

- name
- register
- addr
- start_bit
- width
- value
- max_value
- min
- max
- writable
- minV
- maxV
- resolution

6.12.1 Detailed Description

```
Represents a subset of a 32-bit register [31..0] starting at 'start_bit' consisting
  of 'width' bits. Consistent with the ICD usage, start_bit is MSB e.g., for [7..0],
  the start_bit is '7'.
```

Definition at line 21 of file Subregister.py.

6.12.2 Constructor & Destructor Documentation

6.12.2.1 __init__()

```
nsCamera.utils.Subregister.SubRegister.__init__ (
              self,
              board,
              name,
              register,
              start_bit = 31,
              width = 8,
              writable = False,
              value = 255,
              minV = 0,
              maxV = 5 )
```

Definition at line 28 of file Subregister.py.

```
    ):
        self.name = name
        self.register = register
        self.addr = board.registers[register]
        self.start_bit = start_bit
        self.width = width
        self.value = value
        self.max_value = 2**width - 1  # used to normalize the input values to 1
        self.min = 0
        self.max = self.max_value
        self.writable = writable
        self.minV = minV
        self.maxV = maxV
        # resolution should be reset after init if actual min and max are different
        self.resolution = (1.0 * maxV - minV) / self.max_value


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

6.12.3 Member Data Documentation

6.12.3.1 addr

```
nsCamera.utils.Subregister.SubRegister.addr
```

Definition at line 42 of file Subregister.py.

6.12.3.2 max

```
nsCamera.utils.Subregister.SubRegister.max
```

Definition at line 48 of file Subregister.py.

6.12.3.3 max_value

```
nsCamera.utils.Subregister.SubRegister.max_value
```

Definition at line 46 of file Subregister.py.

6.12.3.4 maxV

```
nsCamera.utils.Subregister.SubRegister.maxV
```

Definition at line 51 of file Subregister.py.

6.12.3.5 min

```
nsCamera.utils.Subregister.SubRegister.min
```

Definition at line 47 of file Subregister.py.

6.12.3.6 minV

```
nsCamera.utils.Subregister.SubRegister.minV
```

Definition at line 50 of file Subregister.py.

6.12.3.7 name

```
nsCamera.utils.Subregister.SubRegister.name
```

Definition at line 40 of file Subregister.py.

6.12.3.8 register

```
nsCamera.utils.Subregister.SubRegister.register
```

Definition at line 41 of file Subregister.py.

6.12.3.9 resolution

`nsCamera.utils.Subregister.SubRegister.resolution`

Definition at line 53 of file Subregister.py.

6.12.3.10 start_bit

`nsCamera.utils.Subregister.SubRegister.start_bit`

Definition at line 43 of file Subregister.py.

6.12.3.11 value

`nsCamera.utils.Subregister.SubRegister.value`

Definition at line 45 of file Subregister.py.

6.12.3.12 width

`nsCamera.utils.Subregister.SubRegister.width`

Definition at line 44 of file Subregister.py.

6.12.3.13 writable

`nsCamera.utils.Subregister.SubRegister.writable`

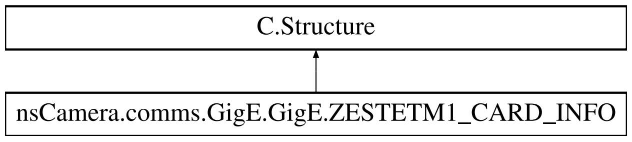
Definition at line 49 of file Subregister.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/utils/Subregister.py

6.13 nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO Class Reference

Inheritance diagram for nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO:



Static Public Attributes

- int ubyte4 = C.c_ubyte ∗ 4
- int ubyte6 = C.c_ubyte ∗ 6

Static Protected Attributes

- list _fields_

6.13.1 Detailed Description

Definition at line 512 of file GigE.py.

6.13.2 Member Data Documentation

6.13.2.1 _fields_

```
list nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO._fields_  [static], [protected]
```

Initial value:

```
=  [
            ("IPAddr", ubyte4),
            ("ControlPort", C.c_ushort),
            ("Timeout", C.c_ulong),
            ("HTTPPort", C.c_ushort),
            ("MACAddr", ubyte6),
            ("SubNet", ubyte4),
            ("Gateway", ubyte4),
            ("SerialNumber", C.c_ulong),
            ("FirmwareVersion", C.c_ulong),
            ("HardwareVersion", C.c_ulong),
        ]
```

Definition at line 515 of file GigE.py.

6.13.2.2 ubyte4

```
int nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO.ubyte4 = C.c_ubyte ∗ 4  [static]
```

Definition at line 513 of file GigE.py.

6.13.2.3 ubyte6

```
int nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO.ubyte6 = C.c_ubyte ∗ 6  [static]
```

Definition at line 514 of file GigE.py.

The documentation for this class was generated from the following file:

- C:/Users/hill35/git/camera_python/nsCamera/comms/GigE.py

Chapter 7

File Documentation

7.1 C:/Users/hill35/git/camera_python/nsCamera/__init__.py File Reference

Namespaces

- namespace nsCamera

Variables

- list nsCamera.__all__ = ["CameraAssembler"]

7.2 __init__.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 22 15:47:43 2016

The Package includes a Camera object and an assembler.

The camera object will be the workhorse of the API.  The assembler is used
to create the Camera object.

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from nsCamera import CameraAssembler

__all__ = ["CameraAssembler"]

"""
```

```
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.3 C:/Users/hill35/git/camera_python/nsCamera/boards/__init__.py File Reference

Namespaces

- namespace nsCamera
- namespace nsCamera.boards

Variables

- list nsCamera.boards.__all__ = ["LLNL_v1", "LLNL_v4"]

7.4 __init__.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
This package is a collection of modules that represent the camera boards
Each board has its own number of ADCs, POTs, and sensors. More devices can be added in
the future. The list of imports will grow as we make more types of boards.

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""
from .LLNL_v1 import llnl_v1
from .LLNL_v4 import llnl_v4

__all__ = ["LLNL_v1", "LLNL_v4"]

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.5 C:/Users/hill35/git/camera_python/nsCamera/comms/__init__.py File Reference

Namespaces

- namespace nsCamera
- namespace nsCamera.comms

Variables

- list nsCamera.comms.__all__ = ["RS422", "GigE"]

7.6 __init__.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 22 15:47:43 2016

This package is a collection of modules for uniform handling of the nsCamera's
  communication systems

Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from .GigE import GigE
from .RS422 import RS422

__all__ = ["RS422", "GigE"]

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.7 C:/Users/hill35/git/camera_python/nsCamera/sensors/__init__.py File Reference

Namespaces

- namespace nsCamera
- namespace nsCamera.sensors

Variables

- list nsCamera.sensors.__all__ = ["icarus", "icarus2", "daedalus"]

7.8 __init__.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
This package is a collection of modules for uniform handling of the various sensors

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from .daedalus import daedalus
from .icarus import icarus
from .icarus2 import icarus2

__all__ = ["icarus", "icarus2", "daedalus"]

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.9 C:/Users/hill35/git/camera_python/nsCamera/utils/__init__.py File Reference

Namespaces

- namespace nsCamera
- namespace nsCamera.utils

Variables

- list nsCamera.utils.__all__ = ["SubRegister", "Packet", "FlatField", "misc"]

7.10 __init__.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
This package is a collection of utility classes for the CameraAssembler

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from .Packet import Packet
from .Subregister import SubRegister

try:
    from misc import *
    from .crc16pure import *
    from .FlatField import *

except:
    pass

__all__ = ["SubRegister", "Packet", "FlatField", "misc"]

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.11 C:/Users/hill35/git/camera_python/nsCamera/boards/LLNL_v1.py File Reference

Classes

- class nsCamera.boards.LLNL_v1.llnl_v1

Namespaces

- namespace nsCamera
- namespace nsCamera.boards
- namespace nsCamera.boards.LLNL_v1

7.12 LLNL_v1.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
LLNLv1 board definition, including monitors, pots, and other board-specific settings

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import logging
import time
from collections import OrderedDict

from nsCamera.utils.Packet import Packet
from nsCamera.utils.Subregister import SubRegister


class llnl_v1:
    """
    Livermore LLNL v1.0 board

    Compatible communication protocols: RS422, GigE
    Compatible sensors: icarus, icarus2, daedalus
    """

    # FPGA register map - use '.upper()' on keys to ensure uppercase lookup
    registers = OrderedDict(
        {
            "FPGA_NUM": "000",
            "FPGA_REV": "001",
            "HS_TIMING_CTL": "010",
            "HS_TIMING_DATA_ALO": "013",
            "HS_TIMING_DATA_AHI": "014",
            "HS_TIMING_DATA_BLO": "015",
            "HS_TIMING_DATA_BHI": "016",
            "SW_TRIGGER_CONTROL": "017",
            "STAT_REG": "024",
            "CTRL_REG": "025",
            "POT_CTL": "026",
            "POT_REG4_TO_1": "027",
            "POT_REG8_TO_5": "028",
            "POT_REG12_TO_9": "029",
            "POT_REG13": "02A",
            "LED_GP": "02B",
            "SW_RESET": "02D",
            "HST_SETTINGS": "02E",
            "STAT_REG_SRC": "02F",
            "STAT_REG2": "030",
            "STAT_REG2_SRC": "031",
            "ADC_BYTECOUNTER": "032",
            "RBP_PIXEL_CNTR": "033",
            "DIAG_MAX_CNT_0": "034",
            "DIAG_MAX_CNT_1": "035",
            "DIAG_CNTR_VAL_0": "036",
            "DIAG_CNTR_VAL_1": "037",
            "TRIGGER_CTL": "03A",
            "SRAM_CTL": "03B",
            "TIMER_CTL": "03C",
            "TIMER_VALUE": "03D",
            "HSTALLWEN_WAIT_TIME": "03F",
            "FPA_ROW_INITIAL": "042",
            "FPA_ROW_FINAL": "043",
            "FPA_FRAME_INITIAL": "044",
            "FPA_FRAME_FINAL": "045",
            "FPA_DIVCLK_EN_ADDR": "046",
            "FPA_OSCILLATOR_SEL_ADDR": "047",
            "FRAME_ORDER_SEL": "04B",
            "SENSOR_VOLT_STAT": "082",
            "SENSOR_VOLT_CTL": "083",
            "ADC_CTL": "090",
            "ADC1_CONFIG_DATA": "091",
            "ADC2_CONFIG_DATA": "092",
            "ADC3_CONFIG_DATA": "093",
            "ADC4_CONFIG_DATA": "094",
            "ADC5_CONFIG_DATA": "095",
            "ADC5_DATA_1": "096",
            "ADC5_DATA_2": "097",
            "ADC5_DATA_3": "098",
            "ADC5_DATA_4": "099",
            "ADC5_PPER": "09A",
            "ADC_STANDBY": "09B",  # board version <= AD
            "ADC_RESET": "09B",  # board version > AD
            "TEMP_SENSE_PPER": "0A0",
            "TEMP_SENSE_DATA": "0A1",
        }
    )

    subregisters = [

        ("HST_MODE", "HS_TIMING_CTL", 0, 1, True),
        ("SW_TRIG_START", "SW_TRIGGER_CONTROL", 0, 1, True),
        ("LED_EN", "CTRL_REG", 1, 1, True),
        ("COLQUENCHEN", "CTRL_REG", 2, 1, True),
        ("POWERSAVE", "CTRL_REG", 3, 1, True),
        ("POT1", "POT_REG4_TO_1", 7, 8, True),
        ("POT2", "POT_REG4_TO_1", 15, 8, True),
        ("POT3", "POT_REG4_TO_1", 23, 8, True),
        ("POT4", "POT_REG4_TO_1", 31, 8, True),
        ("POT5", "POT_REG8_TO_5", 7, 8, True),
        ("POT6", "POT_REG8_TO_5", 15, 8, True),
        ("POT7", "POT_REG8_TO_5", 23, 8, True),
        ("POT8", "POT_REG8_TO_5", 31, 8, True),
        ("POT9", "POT_REG12_TO_9", 7, 8, True),
        ("POT10", "POT_REG12_TO_9", 15, 8, True),
        ("POT11", "POT_REG12_TO_9", 23, 8, True),
        ("POT12", "POT_REG12_TO_9", 31, 8, True),
        ("POT13", "POT_REG13", 7, 8, True),
        ("LED1", "LED_GP", 0, 1, True),
        ("LED2", "LED_GP", 1, 1, True),
        ("LED3", "LED_GP", 2, 1, True),
        ("LED4", "LED_GP", 3, 1, True),
        ("LED5", "LED_GP", 4, 1, True),
        ("LED6", "LED_GP", 5, 1, True),
        ("LED7", "LED_GP", 6, 1, True),
        ("LED8", "LED_GP", 7, 1, True),
        ("RESET", "SW_RESET", 0, 1, True),
        ("HST_SW_CTL_EN", "HST_SETTINGS", 0, 1, True),
        ("SW_HSTALLWEN", "HST_SETTINGS", 1, 1, True),
        ("MAXERR_FIT", "DIAG_MAX_CNT_0", 31, 16, True),
        ("MAXERR_SRT", "DIAG_MAX_CNT_0", 7, 8, True),
        ("MAXERR_UTTR", "DIAG_MAX_CNT_1", 31, 16, True),
        ("MAXERR_URTR", "DIAG_MAX_CNT_1", 15, 16, True),
        ("HW_TRIG_EN", "TRIGGER_CTL", 0, 1, True),
        ("SW_TRIG_EN", "TRIGGER_CTL", 2, 1, True),
        ("READ_SRAM", "SRAM_CTL", 0, 1, True),
        ("RESET_TIMER", "TIMER_CTL", 0, 1, True),
        ("OSC_SELECT", "FPA_OSCILLATOR_SEL_ADDR", 1, 2, True),
        ("ADC5_VREF", "ADC5_CONFIG_DATA", 9, 10, True),
        ("ADC5_VREF3", "ADC5_CONFIG_DATA", 13, 1, True),
        ("ADC5_INT", "ADC5_CONFIG_DATA", 15, 1, True),
        ("ADC5_MULT", "ADC5_CONFIG_DATA", 24, 6, True),
        ("PPER", "ADC5_PPER", 7, 8, True),

        ("SRAM_READY", "STAT_REG", 0, 1, False),
        ("STAT_COARSE", "STAT_REG", 1, 1, False),
        ("STAT_FINE", "STAT_REG", 2, 1, False),
        ("STAT_SENSREADIP", "STAT_REG", 5, 1, False),
        ("STAT_SENSREADDONE", "STAT_REG", 6, 1, False),
        ("STAT_SRAMREADSTART", "STAT_REG", 7, 1, False),
        ("STAT_SRAMREADDONE", "STAT_REG", 8, 1, False),
        ("STAT_HSTCONFIGSTART", "STAT_REG", 9, 1, False),
        ("STAT_ADCSCONFIGURED", "STAT_REG", 10, 1, False),
        ("STAT_POTSCONFIGURED", "STAT_REG", 11, 1, False),
        ("STAT_TIMERCOUNTERRESET", "STAT_REG", 13, 1, False),
        ("STAT_ARMED", "STAT_REG", 14, 1, False),
        ("STAT_TEMP", "STAT_REG", 27, 11, False),
        ("STAT_PRESS", "STAT_REG", 31, 4, False),
        ("FPA_IF_TO", "STAT_REG2", 0, 1, False),
        ("SRAM_RO_TO", "STAT_REG2", 1, 1, False),
        ("PIXELRD_TOUT_ERR", "STAT_REG2", 2, 1, False),
        ("UART_TX_TO_RST", "STAT_REG2", 3, 1, False),
        ("UART_RX_TO_RST", "STAT_REG2", 4, 1, False),
        ("SENSOR_POSN", "SENSOR_VOLT_STAT", 0, 1, False),
        ("SENSOR_NEGP", "SENSOR_VOLT_STAT", 1, 1, False),
        ("ICARUS_DET", "SENSOR_VOLT_STAT", 2, 1, False),
        ("DAEDALUS_DET", "SENSOR_VOLT_STAT", 3, 1, False),
        ("HORUS_DET", "SENSOR_VOLT_STAT", 4, 1, False),
        ("SENSOR_POWER", "SENSOR_VOLT_STAT", 5, 1, False),
        ("FIT_COUNT", "DIAG_CNTR_VAL_0", 31, 16, False),
        ("SRT_COUNT", "DIAG_CNTR_VAL_0", 7, 8, False),
        ("UTTR_COUNT", "DIAG_CNTR_VAL_1", 31, 16, False),
        ("URTR_COUNT", "DIAG_CNTR_VAL_1", 15, 16, False),
        # monitor ADC channels defined here - the poll period will need to be set during
        #   camera initialization (x98)
        ("MON_CH2", "ADC5_DATA_1", 15, 16, False),
        ("MON_CH3", "ADC5_DATA_1", 31, 16, False),
        ("MON_CH4", "ADC5_DATA_2", 15, 16, False),
        ("MON_CH5", "ADC5_DATA_2", 31, 16, False),
        ("MON_CH6", "ADC5_DATA_3", 15, 16, False),
        ("MON_CH7", "ADC5_DATA_3", 31, 16, False),
        ("MON_CH8", "ADC5_DATA_4", 15, 16, False),
        ("MON_VRST", "ADC5_DATA_4", 31, 16, False),
    ]

    def __init__(self, camassem):
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[LLNL_v1] "
        self.logerr = self.ca.logerrbase + "[LLNL_v1] "
        self.logwarn = self.ca.logwarnbase + "[LLNL_v1] "
        self.loginfo = self.ca.loginfobase + "[LLNL_v1] "
        self.logdebug = self.ca.logdebugbase + "[LLNL_v1] "
        logging.info(self.loginfo + "initializing board object")
        self.VREF = 2.5  # default
        self.ADC5_mult = 2  # i.e., monmax = 2 * VREF
        # False => monitor range runs 0 to monmax, True => +/- monmax
        self.ADC5_bipolar = True
        self.rs422_baud = 921600
        self.rs422_cmd_wait = 0.3

        fpgaNum_pkt = Packet(cmd="1", addr=self.registers["FPGA_NUM"])
        fpgaRev_pkt = Packet(cmd="1", addr=self.registers["FPGA_REV"])

        _, _ = self.ca.sendCMD(fpgaNum_pkt)  # dummy duplicate call
        err, rval = self.ca.sendCMD(fpgaNum_pkt)
        self.ca.FPGANum = rval[8:16]

        err, rval = self.ca.sendCMD(fpgaRev_pkt)
        self.ca.FPGAVersion = rval[8:16]

        # map channels to signal names for abstraction at the camera assembler level;
        #   each requires a corresponding entry in 'subregisters'

        self.icarus_subreg_aliases = OrderedDict(
            {
                "COL_BOT_IBIAS_IN": "POT1",
                "HST_A_PDELAY": "POT2",
                "HST_B_NDELAY": "POT3",
                "HST_RO_IBIAS": "POT4",
                "HST_OSC_VREF_IN": "POT5",
                "HST_B_PDELAY": "POT6",
                "HST_OSC_CTL": "POT7",
                "HST_A_NDELAY": "POT8",
                "COL_TOP_IBIAS_IN": "POT9",
                "HST_OSC_R_BIAS": "POT10",
                "VAB": "POT11",
                "HST_RO_NC_IBIAS": "POT12",
                "VRST": "POT13",
                "MON_HST_A_PDELAY": "MON_CH2",
                "MON_HST_B_NDELAY": "MON_CH3",
                "MON_HST_RO_IBIAS": "MON_CH4",
                "MON_HST_OSC_VREF_IN": "MON_CH5",
                "MON_HST_B_PDELAY": "MON_CH6",
                "MON_HST_OSC_CTL": "MON_CH7",
                "MON_HST_A_NDELAY": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.icarus_monitor_controls = OrderedDict(
            {
                "MON_CH2": "POT2",
                "MON_CH3": "POT3",
                "MON_CH4": "POT4",
                "MON_CH5": "POT5",
                "MON_CH6": "POT6",
                "MON_CH7": "POT7",
                "MON_CH8": "POT8",
                # Note: VRST is not measured across the pot; it will read a voltage
                #   approximately 1 Volt lower than pot13's actual output
                "MON_VRST": "POT13",
            }
        )

        self.daedalus_subreg_aliases = OrderedDict(
            {
                "HST_OSC_CTL": "POT4",
                "HST_RO_NC_IBIAS": "POT5",
                "HST_OSC_VREF_IN": "POT6",
                "VAB": "POT11",
                "MON_TSENSEOUT": "MON_CH2",
                "MON_BGREF": "MON_CH3",
                "MON_HST_OSC_CTL": "MON_CH4",
                "MON_HST_RO_NC_IBIAS": "MON_CH5",
                "MON_HST_OSC_VREF_IN": "MON_CH6",
                "MON_COL_TST_IN": "MON_CH7",
                "MON_HST_OSC_PBIAS_PAD": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.daedalus_monitor_controls = OrderedDict(
            {
                "MON_CH4": "POT4",
                "MON_CH5": "POT5",
                "MON_CH6": "POT6",
                # Note: VRST is not measured across the pot; it will read a voltage
                #   lower than pot13's actual output
                "MON_VRST": "POT13",
            }
        )

        self.subreglist = []
        for s in self.subregisters:
            self.subreglist.append(s[0].upper())
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)

        # set voltage ranges for all pots
        for n in range(1, 13):
            potname = "POT" + str(n)
            potobj = getattr(self, potname)
            potobj.minV = 0
            potobj.maxV = 3.3
            # resolution is approximately .0129 V / LSB
            potobj.resolution = (1.0 * potobj.maxV - potobj.minV) / potobj.max_value
        self.POT13.minV = 0
        self.POT13.maxV = 3.96
        # POT13 resolution is approximately .0155 V / LSB
        self.POT13.resolution = (
            1.0 * self.POT13.maxV - self.POT13.minV
        ) / self.POT13.max_value

    def initBoard(self):
        """
        Register and reset board, set up firmware for sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initBoard LLNLv1")
        control_messages = [("LED_EN", "1")]

        self.clearStatus()
        self.configADCs()

        err, resp = self.ca.getSubregister("ADC5_VREF3")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_VREF3'")
        if int(resp, 2):  # check to see if Vref is 3 or 2.5 volts
            vrefmax = 3.0
        else:
            vrefmax = 2.5
        err, resp = self.ca.getSubregister("ADC5_VREF")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_VREF'")
        self.VREF = vrefmax * int(resp, 2) / 1024.0
        err, multmask = self.ca.getSubregister("ADC5_MULT")
        if err:
            logging.error(self.logerr + "unable to read 'ADC5_MULT'")
        if multmask[0] and multmask[1] and multmask[3] and multmask[5]:
            self.ADC5_mult = 2
        elif not (multmask[0] or multmask[1] or multmask[3] or multmask[5]):
            self.ADC5_mult = 4
        else:
            logging.error(self.logerr + "inconsistent mode settings on ADC5")
        return self.ca.submitMessages(control_messages, " initBoard: ")

    def initPots(self):
        """
        Configure default pot settings before image acquisition

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initPots")
        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            err0, _ = self.ca.setPot("HST_A_PDELAY", 0, errflag=True)
            err1, _ = self.ca.setPotV("HST_B_NDELAY", 3.3, errflag=True)
            err2, _ = self.ca.setPotV("HST_RO_IBIAS", 2.5, tune=True, errflag=True)
            err3, _ = self.ca.setPotV("HST_OSC_VREF_IN", 2.9, tune=True, errflag=True)
            err4, _ = self.ca.setPot("HST_B_PDELAY", 0, errflag=True)
            err5, _ = self.ca.setPotV("HST_OSC_CTL", 1.45, tune=True, errflag=True)
            err6, _ = self.ca.setPotV("HST_A_NDELAY", 3.3, errflag=True)
            err7, _ = self.ca.setPotV("VAB", 0.5, errflag=True)
            err8, _ = self.ca.setPotV("HST_RO_NC_IBIAS", 2.5, errflag=True)
            err9, _ = self.ca.setPotV("VRST", 0.3, tune=True, errflag=True)
            err = err0 + err1 + err2 + err3 + err4 + err5 + err6 + err7 + err8 + err9
        else:  # Daedalus
            err0, _ = self.ca.setPotV("HST_OSC_CTL", 1.0, tune=True, errflag=True)
            err1, _ = self.ca.setPotV("HST_RO_NC_IBIAS", 1.0, errflag=True)
            err2, _ = self.ca.setPotV("HST_OSC_VREF_IN", 1.0, tune=True, errflag=True)
            err3, _ = self.ca.setPotV("VAB", 0.5, errflag=True)
            err = err0 + err1 + err2 + err3
        return err, ""

    def latchPots(self):
        """
        Latch pot settings into sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "latchPots")

        control_messages = [
            ("POT_CTL", "00000003"),  # latches register settings for pot 1
            ("POT_CTL", "00000005"),
            ("POT_CTL", "00000007"),
            ("POT_CTL", "00000009"),
            ("POT_CTL", "0000000B"),
            ("POT_CTL", "0000000D"),
            ("POT_CTL", "0000000F"),
            ("POT_CTL", "00000011"),
            ("POT_CTL", "00000013"),
            ("POT_CTL", "00000015"),
            ("POT_CTL", "00000017"),
            ("POT_CTL", "00000019"),
            ("POT_CTL", "0000001B"),
        ]
        return self.ca.submitMessages(control_messages, " latchPots: ")

    def initSensor(self):
        """
        Register sensor, set default timing settings

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initSensor")
        if int(self.ca.FPGANum[7]) != self.ca.sensor.fpganumID:
            logging.error(
                self.logerr + "unable to confirm sensor compatibility with FPGA"
            )
        self.registers.update(self.ca.sensor.sens_registers)
        self.subregisters.extend(self.ca.sensor.sens_subregisters)
        for s in self.ca.sensor.sens_subregisters:
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)
            self.subreglist.append(s[0])
        self.ca.checkSensorVoltStat()
        control_messages = self.ca.sensorSpecific() + [
            # ring w/caps=01, relax=00, ring w/o caps = 02
            ("OSC_SELECT", "00"),
            ("FPA_DIVCLK_EN_ADDR", "00000001"),  # TODO Make this a subregister
        ]
        return self.ca.submitMessages(control_messages, " initSensor: ")

    def configADCs(self):
        """
        Sets default ADC configuration (does not latch settings)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "configADCs")

        control_messages = [
            # just in case ADC_RESET was set (pull all ADCs out # of reset)
            ("ADC_RESET", "00000000"),
            # workaround for uncertain behavior after previous readoff
            ("ADC1_CONFIG_DATA", "FFFFFFFF"),
            ("ADC2_CONFIG_DATA", "FFFFFFFF"),
            ("ADC3_CONFIG_DATA", "FFFFFFFF"),
            ("ADC4_CONFIG_DATA", "FFFFFFFF"),
            ("ADC_CTL", "FFFFFFFF"),
            ("ADC1_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC2_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC3_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC4_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC5_CONFIG_DATA", "81A883FF"),  # int Vref 2.50V
        ]
        return self.ca.submitMessages(control_messages, " configADCs: ")

    def softReboot(self):
        """
        Perform software reboot of board. WARNING: board reboot will likely prevent
          correct communication reponses and therefore will generate an error message

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "reboot")
        control_messages = [("RESET", "1")]
        return self.ca.submitMessages(control_messages, " disarm: ")

    def disarm(self):
        """
        Takes camera out of trigger wait state. Has no effect if camera is not already
          in wait state.

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "disarm")
        self.ca.clearStatus()
        self.ca.armed = False
        control_messages = [
            ("HW_TRIG_EN", "0"),
            ("SW_TRIG_EN", "0"),
        ]
        return self.ca.submitMessages(control_messages, " disarm: ")

    def startCapture(self, mode="Hardware"):
        """
        Selects trigger mode and enables board for image capture

        Args:
            mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
              insensitive)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "startCapture")
        if self.ca.sensmanual:
            timingReg = "MANSHUT_MODE"
        else:
            timingReg = "HST_MODE"

        if mode.upper()[0] == "S":  # SOFTWARE
            trigmess = [
                ("HW_TRIG_EN", "0"),
                ("SW_TRIG_EN", "1"),
                ("SW_TRIG_START", "1"),
            ]
        else:  # HARDWARE
            trigmess = [
                ("SW_TRIG_EN", "0"),
                ("HW_TRIG_EN", "1"),
            ]

        control_messages = [
            ("ADC_CTL", "0000001F"),  # configure all ADCs
            (timingReg, "1"),
        ]

        control_messages.extend(trigmess)
        return self.ca.submitMessages(control_messages, " startCapture: ")

    def readSRAM(self):
        """
        Start readoff of SRAM

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "readSRAM")
        control_messages = [("READ_SRAM", "1")]
        return self.ca.submitMessages(control_messages, " readSRAM: ")

    def waitForSRAM(self, timeout):
        """
        Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
          timeout = None or zero means wait indefinitely

        Args:
            timeout - time in seconds before readoff proceeds automatically without
              waiting for SRAM_READY flag

        Returns:
            error string
        """
        logging.info(self.loginfo + "waitForSRAM")
        waiting = True
        starttime = time.time()
        err = ""
        while waiting:
            err, status = self.ca.getSubregister("SRAM_READY")
            if err:
                logging.error(
                    self.logerr + "error in register read: " + err + " (waitForSRAM)"
                )
            if int(status):
                waiting = False
                logging.info(self.loginfo + "SRAM ready")
            if self.ca.abort:
                waiting = False
                logging.info(self.loginfo + "readoff aborted by user")
                self.ca.abort = False
            if timeout and time.time() - starttime > timeout:
                err += self.logerr + "SRAM timeout; proceeding with download attempt"
                logging.error(err)
                return err
        return err

    def getTimer(self):
        """
        Read value of on-board timer

        Returns:
            timer value as integer
        """
        logging.info(self.loginfo + "getTimer")
        err, rval = self.ca.getRegister("TIMER_VALUE")
        if err:
            logging.error(
                self.logerr + "unable to retrieve timer information (getTimer), "
                'returning "0" '
            )
            return 0
        return int(rval, 16)

    def resetTimer(self):
        """
        Reset on-board timer

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "resetTimer")
        control_messages = [("RESET_TIMER", "1"), ("RESET_TIMER", "0")]
        return self.ca.submitMessages(control_messages, " resetTimer: ")

    def enableLED(self, status):
        """
        Enable/disable on-board LEDs

        Args:
            status: 0 for disabled, 1 for enabled

        Returns:
            tuple: (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "enableLED")
        if status:
            status = 1
        return self.ca.setSubregister("LED_EN", str(status))

    def setLED(self, LED, status):
        """
        Illuminate on-board LED

        Args:
            LED: LED number (1-8)
            status: 0 is off, 1 is on

        Returns:
            tuple: (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "setLED")
        key = "LED" + str(LED)
        return self.ca.setSubregister(key, str(status))

    def setPowerSave(self, status):
        """
        Select powersave option

        Args:
            status: setting for powersave option (1 is enabled)

        Returns:
            tuple (error string, response string from setSubregister()
        """
        logging.info(self.loginfo + "setPowerSave")
        if status:
            status = 1
        return self.ca.setSubregister("POWERSAVE", str(status))

    def setPPER(self, pollperiod):
        """
        Set polling period for ADCs.
        Args:
            pollperiod: milliseconds, between 1 and 255, defaults to 50

        Returns:
            tuple (error string, response string from setSubregister OR invalid time
              setting string)
        """
        logging.debug(self.logdebug + "setPPER: time = " + str(pollperiod))
        if pollperiod is None:
            pollperiod = 50
        if not isinstance(pollperiod, int) or pollperiod < 1 or pollperiod > 255:
            err = (
                self.logerr + "invalid poll period submitted. Setting remains "
                "unchanged. "
            )
            logging.error(err)
            return err, str(pollperiod)
        else:
            binset = bin(pollperiod)[2:].zfill(8)
            return self.ca.setSubregister("PPER", binset)

    def getTemp(self, scale=None, offset=None, slope=None):
        """
        Read temperature sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)
            offset: unused
            slope: unused
        Returns:
            temperature as float on given scale
        """
        logging.debug(self.logdebug + "getTemp: scale = " + str(scale))
        err, rval = self.ca.getRegister("TEMP_SENSE_DATA")
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0

        ctemp = int(rval[-3:], 16) / 16.0
        if scale == "K":
            temp = ctemp + 273.15
        elif scale == "F":
            temp = 1.8 * ctemp + 32
        else:
            temp = ctemp
        return temp

    def getPressure(self, offset, sensitivity, units):
        """
        Read pressure sensor

        Currently unimplemented

        Returns:
            0 as float
        """
        logging.warning(
            "WARNING: [LLNL_v1] 'getPressure' is not implemented on the LLNLv1 board"
        )
        return 0.0

    def clearStatus(self):
        """
        Check status registers to clear them

        Returns:
            error string
        """
        logging.debug(self.logdebug + "clearStatus")
        err1, rval = self.ca.getRegister("STAT_REG_SRC")
        err2, rval = self.ca.getRegister("STAT_REG2_SRC")
        err = err1 + err2
        if err:
            logging.error(self.logerr + "clearStatus failed")
        return err

    def checkStatus(self):
        """
        Check status register, convert to reverse-order bit stream (i.e., bit 0 is
          statusbits[0])

        Returns:
            bit string (no '0b') in reversed order
        """
        logging.debug(self.logdebug + "checkStatus")
        err, rval = self.ca.getRegister("STAT_REG")
        if not rval:
            logging.error(
                self.logerr + "Unable to check status register (zeroes returned)"
            )
            rval = "0"
        rvalbits = bin(int(rval, 16))[2:].zfill(32)
        statusbits = rvalbits[::-1]
        return statusbits  # TODO: add error handling

    def checkStatus2(self):
        """
        Check second status register, convert to reverse-order bit stream (i.e., bit 0
          is statusbits[0])

        Returns: bit string (no '0b') in reversed order
        """
        logging.debug(self.logdebug + "checkStatus2")
        err, rval = self.ca.getRegister("STAT_REG2")
        if not rval:
            logging.error(
                self.logerr + "Unable to check status register 2 (zeroes returned)"
            )
            rval = "0"
        rvalbits = bin(int(rval, 16))[2:].zfill(5)
        statusbits = rvalbits[::-1]
        return statusbits  # TODO: add error handling

    def reportStatus(self):
        """
        Check contents of status register, print relevant messages
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()
        logging.info(self.loginfo + "Status report:")
        if int(statusbits[0]):
            print(self.loginfo + "Sensor read complete")
        if int(statusbits[1]):
            print(self.loginfo + "Coarse trigger detected")
        if int(statusbits[2]):
            print(self.loginfo + "Fine trigger detected")
        if int(statusbits[5]):
            print(self.loginfo + "Sensor readout in progress")
        if int(statusbits[6]):
            print(self.loginfo + "Sensor readout complete")
        if int(statusbits[7]):
            print(self.loginfo + "SRAM readout started")
        if int(statusbits[8]):
            print(self.loginfo + "SRAM readout complete")
        if int(statusbits[9]):
            print(self.loginfo + "High-speed timing configuration started")
        if int(statusbits[10]):
            print(self.loginfo + "All ADCs configured")
        if int(statusbits[11]):
            print(self.loginfo + "All pots configured")
        if int(statusbits[13]):
            print(self.loginfo + "Timer has reset")
        if int(statusbits[14]):
            print(self.loginfo + "Camera is Armed")
        self.ca.sensor.reportStatusSensor(statusbits, statusbits2)
        temp = int(statusbits[27:15:-1], 2) / 16.0
        logging.info(
            self.loginfo + "Temperature reading: " + "{0:1.2f}".format(temp) + " C"
        )
        # press = int(statusbits[:27:-1], 2)
        # logging.info(self.loginfo + "Pressure reading: " + "{0:1.2f}".format(press))
        if int(statusbits2[0]):
            print(self.loginfo + "FPA_IF_TO")
        if int(statusbits2[1]):
            print(self.loginfo + "SRAM_RO_TO")
        if int(statusbits2[2]):
            print(self.loginfo + "PixelRd Timeout Error")
        if int(statusbits2[3]):
            print(self.loginfo + "UART_TX_TO_RST")
        if int(statusbits2[4]):
            print(self.loginfo + "UART_RX_TO_RST")

    def reportEdgeDetects(self):
        """
        Unimplemented
        """
        logging.warning(
            self.logwarn + "'reportEdgeDetects' is not implemented on the LLNLv1 "
            "board "
        )

    def dumpStatus(self):
        """
        Create dictionary of status values, DAC settings, monitor values, and register
          values

        WARNING: the behavior of self-resetting subregisters may be difficult to predict
          and may generate contradictory results

        Returns:
            dictionary of system diagnostic values
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()
        temp = self.ca.getTemp()

        statDict = OrderedDict(
            {
                "Temperature reading": "{0:1.2f}".format(temp) + " C",
                "Sensor read complete": str(statusbits[0]),
                "Coarse trigger detected": str(statusbits[1]),
                "Fine trigger detected": str(statusbits[2]),
                "Sensor readout in progress": str(statusbits[5]),
                "Sensor readout complete": str(statusbits[6]),
                "SRAM readout started": str(statusbits[7]),
                "SRAM readout complete": str(statusbits[8]),
                "High-speed timing configured": str(statusbits[9]),
                "All ADCs configured": str(statusbits[10]),
                "All pots configured": str(statusbits[11]),
                "HST_All_W_En detected": str(statusbits[12]),
                "Timer has reset": str(statusbits[13]),
                "Camera is Armed": str(statusbits[14]),
                "FPA_IF_TO": str(statusbits2[0]),
                "SRAM_RO_TO": str(statusbits2[1]),
                "PixelRd Timeout Error": str(statusbits2[2]),
                "UART_TX_TO_RST": str(statusbits2[3]),
                "UART_RX_TO_RST": str(statusbits2[4]),
            }
        )

        POTDict = OrderedDict()
        MonDict = OrderedDict()
        for entry in self.subreg_aliases:
            if self.subreg_aliases[entry][0] == "P":
                val = str(round(self.ca.getPotV(entry), 3)) + " V"
                POTDict["POT_" + entry] = val
            else:
                val = str(round(self.ca.getMonV(entry), 3)) + " V"
                MonDict[entry] = val

        regDict = OrderedDict()
        for key in self.registers.keys():
            err, rval = self.ca.getRegister(key)
            regDict[key] = rval

        dumpDict = OrderedDict()
        for x in [statDict, MonDict, POTDict, regDict]:
            dumpDict.update(x)
        return dumpDict


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.13 C:/Users/hill35/git/camera_python/nsCamera/boards/LLNL_v4.py File Reference

Classes

- class nsCamera.boards.LLNL_v4.llnl_v4

Namespaces

- namespace nsCamera
- namespace nsCamera.boards
- namespace nsCamera.boards.LLNL_v4

7.14 LLNL_v4.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
LLNLv4 board definition, including monitors, DACS, and other board-specific settings

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import logging
import string
import time
from collections import OrderedDict

from nsCamera.utils.Packet import Packet
from nsCamera.utils.Subregister import SubRegister


class llnl_v4:
    """
    Livermore LLNL v4.0 board

    Compatible communication protocols: RS422, GigE
    Compatible sensors: icarus, icarus2, daedalus
    """

    # FPGA register map - use '.upper()' on keys to ensure uppercase lookup
    registers = OrderedDict(
        {
            "FPGA_NUM": "000",
            "FPGA_REV": "001",
            "HS_TIMING_CTL": "010",
            "HS_TIMING_DATA_ALO": "013",
            "HS_TIMING_DATA_AHI": "014",
            "HS_TIMING_DATA_BLO": "015",
            "HS_TIMING_DATA_BHI": "016",
            "SW_TRIGGER_CONTROL": "017",
            "SW_COARSE_CONTROL": "01C",
            "STAT_REG": "024",
            "CTRL_REG": "025",
            "DAC_CTL": "026",
            "DAC_REG_A_AND_B": "027",
            "DAC_REG_C_AND_D": "028",
            "DAC_REG_E_AND_F": "029",
            "DAC_REG_G_AND_H": "02A",
            "SW_RESET": "02D",
            "HST_SETTINGS": "02E",
            "STAT_REG_SRC": "02F",
            "STAT_REG2": "030",
            "STAT_REG2_SRC": "031",
            "ADC_BYTECOUNTER": "032",
            "RBP_PIXEL_CNTR": "033",
            "DIAG_MAX_CNT_0": "034",
            "DIAG_MAX_CNT_1": "035",
            "DIAG_CNTR_VAL_0": "036",
            "DIAG_CNTR_VAL_1": "037",
            "STAT_EDGE_DETECTS": "038",
            "TRIGGER_CTL": "03A",
            "SRAM_CTL": "03B",
            "TIMER_CTL": "03C",
            "TIMER_VALUE": "03D",
            "HSTALLWEN_WAIT_TIME": "03F",
            "FPA_ROW_INITIAL": "042",
            "FPA_ROW_FINAL": "043",
            "FPA_FRAME_INITIAL": "044",
            "FPA_FRAME_FINAL": "045",
            "FPA_DIVCLK_EN_ADDR": "046",
            "FPA_OSCILLATOR_SEL_ADDR": "047",
            "SUSPEND_TIME": "04D",
            "FPA_INTERFACE_STATE": "04E",
            "DELAY_READOFF": "04F",
            "STAT_REG_SEC": "060",
            "ADC_CTL": "090",
            "ADC1_CONFIG_DATA": "091",
            "ADC2_CONFIG_DATA": "092",
            "ADC3_CONFIG_DATA": "093",
            "ADC4_CONFIG_DATA": "094",
            "ADC5_DATA_1": "095",
            "ADC5_DATA_2": "096",
            "ADC5_DATA_3": "097",
            "ADC5_DATA_4": "098",
            "ADC6_DATA_1": "099",
            "ADC6_DATA_2": "09A",
            "ADC6_DATA_3": "09B",
            "ADC6_DATA_4": "09C",
            "ADC_PPER": "09D",
            "ADC_RESET": "09E",
        }
    )

    subregisters = [

        ("HST_MODE", "HS_TIMING_CTL", 0, 1, True),
        ("SW_TRIG_START", "SW_TRIGGER_CONTROL", 0, 1, True),
        ("SW_COARSE_TRIGGER", "SW_COARSE_CONTROL", 0, 1, True),
        ("LED_EN", "CTRL_REG", 1, 1, True),
        ("COLQUENCHEN", "CTRL_REG", 2, 1, True),
        ("POWERSAVE", "CTRL_REG", 3, 1, True),
        ("PDBIAS_LOW", "CTRL_REG", 6, 1, True),
        # ("SWACK", "CTRL_REG", 10, 1, True),
        ("DACA", "DAC_REG_A_AND_B", 31, 16, True),
        ("DACB", "DAC_REG_A_AND_B", 15, 16, True),
        ("DACC", "DAC_REG_C_AND_D", 31, 16, True),
        ("DACD", "DAC_REG_C_AND_D", 15, 16, True),
        ("DACE", "DAC_REG_E_AND_F", 31, 16, True),
        ("DACF", "DAC_REG_E_AND_F", 15, 16, True),
        ("DACG", "DAC_REG_G_AND_H", 31, 16, True),
        ("DACH", "DAC_REG_G_AND_H", 15, 16, True),
        ("RESET", "SW_RESET", 0, 1, True),
        ("HST_SW_CTL_EN", "HST_SETTINGS", 0, 1, True),
        ("SW_HSTALLWEN", "HST_SETTINGS", 1, 1, True),
        ("MAXERR_FIT", "DIAG_MAX_CNT_0", 31, 16, True),
        ("MAXERR_SRT", "DIAG_MAX_CNT_0", 7, 8, True),
        ("MAXERR_UTTR", "DIAG_MAX_CNT_1", 31, 16, True),
        ("MAXERR_URTR", "DIAG_MAX_CNT_1", 15, 16, True),
        ("HW_TRIG_EN", "TRIGGER_CTL", 0, 1, True),
        ("SW_TRIG_EN", "TRIGGER_CTL", 2, 1, True),
        ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True),
        ("READ_SRAM", "SRAM_CTL", 0, 1, True),
        ("RESET_TIMER", "TIMER_CTL", 0, 1, True),
        ("OSC_SELECT", "FPA_OSCILLATOR_SEL_ADDR", 1, 2, True),
        ("PPER", "ADC_PPER", 7, 8, True),

        ("SRAM_READY", "STAT_REG", 0, 1, False),
        ("STAT_COARSE", "STAT_REG", 1, 1, False),
        ("STAT_FINE", "STAT_REG", 2, 1, False),
        ("STAT_SENSREADIP", "STAT_REG", 5, 1, False),
        ("STAT_SENSREADDONE", "STAT_REG", 6, 1, False),
        ("STAT_SRAMREADSTART", "STAT_REG", 7, 1, False),
        ("STAT_SRAMREADDONE", "STAT_REG", 8, 1, False),
        ("STAT_HSTCONFIGSTART", "STAT_REG", 9, 1, False),
        ("STAT_ADCSCONFIGURED", "STAT_REG", 10, 1, False),
        ("STAT_DACSCONFIGURED", "STAT_REG", 11, 1, False),
        ("STAT_TIMERCOUNTERRESET", "STAT_REG", 13, 1, False),
        ("STAT_HSTCONFIGDONE", "STAT_REG", 16, 1, False),
        ("STAT_ARMED", "STAT_REG", 14, 1, False),
        ("STAT_TEMP", "STAT_REG", 23, 7, False),
        ("STAT_PRESS", "STAT_REG", 31, 8, False),
        ("FPA_IF_TO", "STAT_REG2", 0, 1, False),
        ("SRAM_RO_TO", "STAT_REG2", 1, 1, False),
        ("PIXELRD_TOUT_ERR", "STAT_REG2", 2, 1, False),
        ("UART_TX_TO_RST", "STAT_REG2", 3, 1, False),
        ("UART_RX_TO_RST", "STAT_REG2", 4, 1, False),
        ("FIT_COUNT", "DIAG_CNTR_VAL_0", 31, 16, False),
        ("SRT_COUNT", "DIAG_CNTR_VAL_0", 7, 8, False),
        ("UTTR_COUNT", "DIAG_CNTR_VAL_1", 31, 16, False),
        ("URTR_COUNT", "DIAG_CNTR_VAL_1", 15, 16, False),
        # monitor ADC channels defined here - the poll period will need to be set
        #   during camera initialization (x98)
        ("MON_CH1", "ADC5_DATA_1", 11, 12, False),
        ("MON_CH2", "ADC5_DATA_1", 23, 12, False),
        ("MON_CH3", "ADC5_DATA_2", 11, 12, False),
        ("MON_CH4", "ADC5_DATA_2", 23, 12, False),
        ("MON_CH5", "ADC5_DATA_3", 11, 12, False),
        ("MON_CH6", "ADC5_DATA_3", 23, 12, False),
        ("MON_CH7", "ADC5_DATA_4", 11, 12, False),
        ("MON_CH8", "ADC5_DATA_4", 23, 12, False),
        ("MON_CH9", "ADC6_DATA_1", 11, 12, False),
        ("MON_CH10", "ADC6_DATA_1", 23, 12, False),
        ("MON_CH11", "ADC6_DATA_2", 11, 12, False),
        ("MON_CH12", "ADC6_DATA_2", 23, 12, False),
        ("MON_CH13", "ADC6_DATA_3", 11, 12, False),
        ("MON_CH14", "ADC6_DATA_3", 23, 12, False),
        ("MON_CH15", "ADC6_DATA_4", 11, 12, False),
        ("MON_CH16", "ADC6_DATA_4", 23, 12, False),
    ]

    def __init__(self, camassem):
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[LLNL_v4] "
        self.logerr = self.ca.logerrbase + "[LLNL_v4] "
        self.logwarn = self.ca.logwarnbase + "[LLNL_v4] "
        self.loginfo = self.ca.loginfobase + "[LLNL_v4] "
        self.logdebug = self.ca.logdebugbase + "[LLNL_v4] "
        logging.info(self.loginfo + "Initializing board object")
        self.VREF = 3.3  # must be supplied externally for ADC128S102
        self.ADC5_mult = 1

        # ADC128S102; False => monitor range runs 0 to monmax, True => +/- monmax
        self.ADC5_bipolar = False
        self.rs422_baud = 921600
        self.rs422_cmd_wait = 0.3

        fpgaNum_pkt = Packet(cmd="1", addr=self.registers["FPGA_NUM"])
        fpgaRev_pkt = Packet(cmd="1", addr=self.registers["FPGA_REV"])

        _, _ = self.ca.sendCMD(fpgaNum_pkt)  # dummy duplicate call
        err, rval = self.ca.sendCMD(fpgaNum_pkt)
        self.ca.FPGANum = rval[8:16]

        err, rval = self.ca.sendCMD(fpgaRev_pkt)
        self.ca.FPGAVersion = rval[8:16]

        self.defoff = 34.5  # default pressure sensor offset
        self.defsens = 92.5  # default pressure sensor sensitivity

        # TODO: move to sensor scripts?
        # map channels to signal names for abstraction at the camera assembler level;
        #   each requires a corresponding entry in 'subregisters'
        self.icarus_subreg_aliases = OrderedDict(
            {
                "HST_A_PDELAY": "DACA",
                "HST_A_NDELAY": "DACB",
                "HST_B_PDELAY": "DACC",
                "HST_B_NDELAY": "DACD",
                "HST_RO_IBIAS": "DACE",
                "HST_RO_NC_IBIAS": "DACE",
                "HST_OSC_CTL": "DACF",
                "VAB": "DACG",
                "VRST": "DACH",
                "MON_PRES_MINUS": "MON_CH1",
                "MON_PRES_PLUS": "MON_CH2",
                "MON_TEMP": "MON_CH3",
                "MON_COL_TOP_IBIAS_IN": "MON_CH4",
                "MON_HST_OSC_R_BIAS": "MON_CH5",
                "MON_VAB": "MON_CH6",
                "MON_HST_RO_IBIAS": "MON_CH7",
                "MON_HST_RO_NC_IBIAS": "MON_CH7",
                "MON_VRST": "MON_CH8",
                "MON_COL_BOT_IBIAS_IN": "MON_CH9",
                "MON_HST_A_PDELAY": "MON_CH10",
                "MON_HST_B_NDELAY": "MON_CH11",
                "DOSIMETER": "MON_CH12",
                "MON_HST_OSC_VREF_IN": "MON_CH13",
                "MON_HST_B_PDELAY": "MON_CH14",
                "MON_HST_OSC_CTL": "MON_CH15",
                "MON_HST_A_NDELAY": "MON_CH16",
                "MON_CHA": "MON_CH10",
                "MON_CHB": "MON_CH16",
                "MON_CHC": "MON_CH14",
                "MON_CHD": "MON_CH11",
                "MON_CHE": "MON_CH7",
                "MON_CHF": "MON_CH15",
                "MON_CHG": "MON_CH6",
                "MON_CHH": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.icarus_monitor_controls = OrderedDict(
            {
                "MON_CH10": "DACA",
                "MON_CH16": "DACB",
                "MON_CH14": "DACC",
                "MON_CH11": "DACD",
                "MON_CH7": "DACE",
                "MON_CH15": "DACF",
                "MON_CH6": "DACG",
                "MON_CH8": "DACH",
            }
        )
        self.daedalus_subreg_aliases = OrderedDict(
            {
                "HST_OSC_VREF_IN": "DACC",
                "HST_OSC_CTL": "DACE",
                "COL_TST_IN": "DACF",
                "VAB": "DACG",
                "VRST": "DACH",
                "MON_PRES_MINUS": "MON_CH1",
                "MON_PRES_PLUS": "MON_CH2",
                "MON_TEMP": "MON_CH3",
                "MON_VAB": "MON_CH6",
                "MON_HST_OSC_CTL": "MON_CH7",
                "MON_TSENSE_OUT": "MON_CH10",
                "MON_BGREF": "MON_CH11",
                "DOSIMETER": "MON_CH12",
                "MON_HST_RO_NC_IBIAS": "MON_CH13",
                "MON_HST_OSC_VREF_IN": "MON_CH14",
                "MON_COL_TST_IN": "MON_CH15",
                "MON_HST_OSC_PBIAS_PAD": "MON_CH16",
                "MON_CHC": "MON_CH14",
                "MON_CHE": "MON_CH7",
                "MON_CHF": "MON_CH15",
                "MON_CHG": "MON_CH6",
                "MON_CHH": "MON_CH8",
            }
        )
        # Read-only; identifies controls corresponding to monitors
        self.daedalus_monitor_controls = OrderedDict(
            {
                "MON_CH14": "DACC",
                "MON_CH7": "DACE",
                "MON_CH15": "DACF",
                "MON_CH6": "DACG",
                "MON_CH8": "DACH",
            }
        )
        self.subreglist = []
        for s in self.subregisters:
            self.subreglist.append(s[0].upper())
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)

        # set voltage ranges for all DACs - WARNING: actual output voltage limited to
        #   external supply (3.3 V)
        # setpot('potx', n) will generate 3.3 V for all n > .66
        for n in range(0, 8):
            potname = "DAC" + string.ascii_uppercase[n]
            potobj = getattr(self, potname)
            potobj.minV = 0
            potobj.maxV = 5  #
            potobj.resolution = (
                1.0 * potobj.maxV - potobj.minV
            ) / potobj.max_value  # 76 uV / LSB

    def initBoard(self):
        """
        Register and reset board, set up firmware for sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initBoard LLNLv4")
        control_messages = []
        self.clearStatus()
        self.configADCs()
        return self.ca.submitMessages(control_messages, " initBoard: ")

    def initPots(self):
        """
        Dummy function; initial DAC values are set by firmware at startup

        Returns:
            tuple (empty string, empty string)
        """
        logging.debug(self.logdebug + "InitPots")
        return "", ""

    def latchPots(self):
        """
        Latch DAC settings into sensor

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "latchPots")
        control_messages = [
            ("DAC_CTL", "00000001"),  # latches register settings for DACA
            ("DAC_CTL", "00000003"),
            ("DAC_CTL", "00000005"),
            ("DAC_CTL", "00000007"),
            ("DAC_CTL", "00000009"),
            ("DAC_CTL", "0000000B"),
            ("DAC_CTL", "0000000D"),
            ("DAC_CTL", "0000000F"),
        ]
        return self.ca.submitMessages(control_messages, " latchPots: ")

    def initSensor(self):
        """
        Register sensor, set default timing settings

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "initSensor")
        if int(self.ca.FPGANum[7]) != self.ca.sensor.fpganumID:
            logging.warning(
                self.logwarn + "unable to confirm sensor compatibility with FPGA"
            )
        self.registers.update(self.ca.sensor.sens_registers)
        self.subregisters.extend(self.ca.sensor.sens_subregisters)
        for s in self.ca.sensor.sens_subregisters:
            sr = SubRegister(
                self,
                name=s[0].upper(),
                register=s[1].upper(),
                start_bit=s[2],
                width=s[3],
                writable=s[4],
            )
            setattr(self, s[0].upper(), sr)
            self.subreglist.append(s[0])
        # TODO: self.ca.checkSensorVoltStat() # SENSOR_VOLT_STAT and SENSOR_VOLT_CTL are
        #   deactivated for v4 icarus and daedalus firmware for now, is this permanent?
        control_messages = self.ca.sensorSpecific() + [
            # ring w/caps=01, relax=00, ring w/o caps = 02
            ("OSC_SELECT", "00"),
            ("FPA_DIVCLK_EN_ADDR", "00000001"),
        ]
        return self.ca.submitMessages(control_messages, " initSensor: ")

    def configADCs(self):
        """
        Sets default ADC configuration (does not latch settings)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "configADCs")

        control_messages = [
            # just in case ADC_RESET was set on any of the ADCs (pull all ADCs out of
            #   reset)
            ("ADC_RESET", "00000000"),
            # workaround for uncertain behavior after previous readoff
            ("ADC1_CONFIG_DATA", "FFFFFFFF"),
            ("ADC2_CONFIG_DATA", "FFFFFFFF"),
            ("ADC3_CONFIG_DATA", "FFFFFFFF"),
            ("ADC4_CONFIG_DATA", "FFFFFFFF"),
            ("ADC_CTL", "FFFFFFFF"),
            ("ADC1_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC2_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC3_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
            ("ADC4_CONFIG_DATA", "81A801FF"),  # ext Vref 1.25V
        ]
        return self.ca.submitMessages(control_messages, " configADCs: ")

    def softReboot(self):
        """
        Perform software reboot of board. WARNING: board reboot will likely prevent
          correct response and therefore will generate an error message

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "reboot")
        control_messages = [("RESET", "0")]
        return self.ca.submitMessages(control_messages, " disarm: ")

    def disarm(self):
        """
        Takes camera out of trigger wait state. Has no effect if camera is not in wait
          state.

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "disarm")
        self.ca.clearStatus()
        self.ca.armed = False
        control_messages = [
            ("HW_TRIG_EN", "0"),
            ("SW_TRIG_EN", "0"),
        ]
        self.ca.comms.skipError = False
        return self.ca.submitMessages(control_messages, " disarm: ")

    def startCapture(self, mode="Hardware"):
        """
        Selects trigger mode and enables board for image capture

        Args:
            mode: trigger mode ("hardware"|"software"|"dual|"h"|"s"|"d" , is case-
              insensitive)

        Returns:
            tuple (error string, response string) from final control message
        """
        logging.info(self.loginfo + "startCapture")
        if self.ca.sensmanual:
            timingReg = "MANSHUT_MODE"
        else:
            timingReg = "HST_MODE"

        if mode.upper()[0] == "S":  # SOFTWARE
            trigmess = [
                ("HW_TRIG_EN", "0"),
                ("SW_TRIG_EN", "1"),
                ("SW_TRIG_START", "1"),
            ]
        else:  # HARDWARE
            trigmess = [
                ("SW_TRIG_EN", "0"),
                ("HW_TRIG_EN", "1"),
            ]

        control_messages = [
            ("ADC_CTL", "0000000F"),  # configure all ADCs
            (timingReg, "1"),
        ]

        control_messages.extend(trigmess)
        return self.ca.submitMessages(control_messages, " startCapture: ")

    def readSRAM(self):
        """
        Start readoff of SRAM

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "readSRAM")
        control_messages = [("READ_SRAM", "1")]
        return self.ca.submitMessages(control_messages, " readSRAM: ")

    def waitForSRAM(self, timeout):
        """
        Wait until subreg 'SRAM_READY' flag is true or timeout is exceeded;
          timeout = None or zero means wait indefinitely

        Args:
            timeout - time in seconds before readoff proceeds automatically without
              waiting for SRAM_READY flag

        Returns:
            error string
        """
        logging.info(self.loginfo + "waitForSRAM, timeout = " + str(timeout))
        waiting = True
        starttime = time.time()
        err = ""
        while waiting:
            err, status = self.ca.getSubregister("SRAM_READY")
            if err:
                err = self.logerr + "error in register read: " + err + " (waitForSRAM)"
                logging.error(err)
            if int(status):
                waiting = False
                logging.info(self.loginfo + "SRAM ready")
            if self.ca.abort:
                waiting = False
                logging.info(self.loginfo + "readoff aborted by user")
                self.ca.abort = False
            if timeout and time.time() - starttime > timeout:
                err += self.logerr + "SRAM timeout; proceeding with download attempt"
                logging.error(err)
                return err
            # Slow down for debugging (avoid thousands of messages)
            if self.ca.verbose >= 5:
                time.sleep(0.5)
        return err

    def getTimer(self):
        """
        Read value of on-board timer

        Returns:
            timer value as integer
        """
        logging.info(self.loginfo + "getTimer")
        err, rval = self.ca.getRegister("TIMER_VALUE")
        if err:
            logging.error(
                self.logerr + "unable to retrieve timer information (getTimer), "
                'returning "0" '
            )
            return 0
        return int(rval, 16)

    def resetTimer(self):
        """
        Reset on-board timer

        Returns:
            tuple (error string, response string from register set)
        """
        logging.info(self.loginfo + "resetTimer")
        control_messages = [("RESET_TIMER", "1"), ("RESET_TIMER", "0")]
        return self.ca.submitMessages(control_messages, " resetTimer: ")

    def enableLED(self, status):
        """
        Dummy function; feature is not implemented on LLNL_V4 board

        Returns:
            tuple: dummy of (error string, response string from setSubregister())
        """
        del status
        return "", "0"

    def setLED(self, LED, status):
        """
        Dummy function; feature is not implemented on LLNL_V4 board

        Returns:
            tuple: dummy of (error string, response string from setSubregister())
        """
        del LED, status
        return "", "0"

    def setPowerSave(self, status):
        """
        Select powersave option

        Args:
            status: setting for powersave option (1 is enabled)

        Returns:
            tuple (error string, response string from setSubregister())
        """
        if status:
            status = 1
        return self.ca.setSubregister("POWERSAVE", str(status))

    def setPPER(self, pollperiod):
        """
        Set polling period for ADCs.
        Args:
            pollperiod: milliseconds, between 1 and 255; defaults to 50

        Returns:
            tuple (error string, response string from setSubregister() OR invalid time
              setting string)
        """
        if pollperiod is None:
            pollperiod = 50
        if not isinstance(pollperiod, int) or pollperiod < 1 or pollperiod > 255:
            err = (
                self.logerr + "invalid poll period submitted. Setting remains "
                "unchanged. "
            )
            logging.error(err)
            return err, str(pollperiod)
        else:
            binset = bin(pollperiod)[2:].zfill(8)
            return self.ca.setSubregister("PPER", binset)

    def getTemp(self, scale=None):
        """
        Read temperature sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)

        Returns:
            temperature as float on given scale
        """
        err, rval = self.ca.getMonV("MON_TEMP", errflag=True)
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0
        ctemp = rval * 1000 - 273.15
        if scale == "K":
            temp = ctemp + 273.15
        elif scale == "F":
            temp = 1.8 * ctemp + 32
        else:
            temp = ctemp
        return temp

    def getPressure(self, offset=None, sensitivity=None, units=None):
        """
        Read pressure sensor. Uses default offset and sensitivity defined in init
          function unless alternatives are specified. NOTE: to reset defaults, reassign
          board.defoff and board.defsens explicitly

        Args:
            offset: non-default offset in mv/V
            sensitivity: non-default sensitivity in mV/V/span
            units: units to report pressure (defaults to Torr, options are psi, bar,
              inHg, atm)

        Returns:
            Pressure as float in chosen units, defaults to torr
        """
        if offset is None:
            offset = self.defoff
        if sensitivity is None:
            sensitivity = self.defsens
        if units is None:
            units = "torr"
        pplus = self.ca.getMonV("MON_PRES_PLUS")
        pminus = self.ca.getMonV("MON_PRES_MINUS")
        delta = 1000 * (pplus - pminus)
        ratio = sensitivity / 30  # nominal is 21/30
        psi = (delta - offset) / ratio
        if units.lower() == "psi":
            press = psi
        elif units.lower() == "bar":
            press = psi / 14.504
        elif units.lower() == "atm":
            press = psi / 14.695
        elif units.lower() == "inHg":
            press = psi * 2.036
        else:
            press = 51.715 * psi  # default to Torr

        return press

    def clearStatus(self):
        """
        Check status registers to clear them

        Returns:
            error string
        """
        err1, rval = self.ca.getRegister("STAT_REG_SRC")
        err2, rval = self.ca.getRegister("STAT_REG2_SRC")
        err = err1 + err2
        if err:
            logging.error(self.logerr + "clearStatus failed")
        return err

    def checkStatus(self):
        """
        Check status register, convert to reverse-order bit stream (i.e., bit 0 is
          statusbits[0])

        Returns:
            bit string (no '0b') in reversed order
        """
        err, rval = self.ca.getRegister("STAT_REG")
        rvalbits = bin(int(rval, 16))[2:].zfill(32)
        statusbits = rvalbits[::-1]
        return statusbits

    def checkStatus2(self):
        """
        Check second status register, convert to reverse-order bit stream (i.e., bit 0
          is statusbits[0])

        Returns: bit string (no '0b') in reversed order
        """
        err, rval = self.ca.getRegister("STAT_REG2")
        rvalbits = bin(int(rval, 16))[2:].zfill(6)
        statusbits = rvalbits[::-1]
        return statusbits

    def reportStatus(self):
        """
        Check contents of status register, print relevant messages
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()

        logging.info(self.loginfo + "Status report:")
        if int(statusbits[0]):
            print(self.loginfo + "Sensor read complete")
        if int(statusbits[1]):
            print(self.loginfo + "Coarse trigger detected")
        if int(statusbits[2]):
            print(self.loginfo + "Fine trigger detected")
        if int(statusbits[5]):
            print(self.loginfo + "Sensor readout in progress")
        if int(statusbits[6]):
            print(self.loginfo + "Sensor readout complete")
        if int(statusbits[7]):
            print(self.loginfo + "SRAM readout started")
        if int(statusbits[8]):
            print(self.loginfo + "SRAM readout complete")
        if int(statusbits[9]):
            print(self.loginfo + "High-speed timing configuration started")
        if int(statusbits[10]):
            print(self.loginfo + "All ADCs configured")
        if int(statusbits[11]):
            print(self.loginfo + "All DACs configured")
        if int(statusbits[13]):
            print(self.loginfo + "Timer has reset")
        if int(statusbits[14]):
            print(self.loginfo + "Camera is Armed")
        if int(statusbits[16]):
            print(self.loginfo + "High-speed timing configuration complete")
        self.ca.sensor.reportStatusSensor(statusbits, statusbits2)
        temp = int(statusbits[23:16:-1], 2) * 3.3 * 1000 / 4096
        logging.info(
            self.loginfo + "Temperature reading: " + "{0:1.2f}".format(temp) + " C"
        )
        press = int(statusbits[:23:-1], 2) * 3.3 * 1000 / 4096
        logging.info(
            self.loginfo
            + "Pressure sensor reading: "
            + "{0:1.2f}".format(press)
            + " mV"
        )
        if int(statusbits2[0]):
            print(self.loginfo + "FPA_IF_TO")
        if int(statusbits2[1]):
            print(self.loginfo + "SRAM_RO_TO")
        if int(statusbits2[2]):
            print(self.loginfo + "PixelRd Timeout Error")
        if int(statusbits2[3]):
            print(self.loginfo + "UART_TX_TO_RST")
        if int(statusbits2[4]):
            print(self.loginfo + "UART_RX_TO_RST")

    def reportEdgeDetects(self):
        """
        Report edge detects
        """
        err, rval = self.ca.getRegister("STAT_EDGE_DETECTS")
        # shift to left to fake missing edge detect
        edgebits = bin(int(rval, 16) « 1)[2:].zfill(32)
        # reverse to get order matching assignment
        bitsrev = edgebits[::-1]
        detdict = {}
        bitidx = 0
        for frame in range(4):
            for vert in ("TOP", "BOT"):
                for edge in range(1, 3):
                    for hor in ("A", "B"):
                        detname = (
                            "W"
                            + str(frame)
                            + "_"
                            + vert
                            + "_"
                            + hor
                            + "_EDGE"
                            + str(edge)
                        )
                        detdict[detname] = bitsrev[bitidx]
                        bitidx += 1
        # remove faked detect
        del detdict["W0_TOP_A_EDGE1"]
        logging.info(self.loginfo + "Edge detect report:")
        for key, val in detdict.items():
            logging.info(self.loginfo + key + ": " + val)

    def dumpStatus(self):
        """
        Create dictionary of status values, DAC settings, monitor values, and register
          values.

        Returns:
            dictionary of system diagnostic values
        """
        statusbits = self.checkStatus()
        statusbits2 = self.checkStatus2()

        temp = int(statusbits[23:16:-1], 2) * 3.3 * 1000 / 4096
        press = int(statusbits[:23:-1], 2) * 3.3 * 1000 / 4096

        statDict = OrderedDict(
            {
                "Temperature sensor reading (Deg C)": "{0:1.2f}".format(temp),
                "Pressure reading (Torr)": str(round(self.ca.getPressure(), 3)),
                "Pressure sensor reading (mV)": "{0:1.2f}".format(press),
                "Sensor read complete": str(statusbits[0]),
                "Coarse trigger detected": str(statusbits[1]),
                "Fine trigger detected": str(statusbits[2]),
                "Sensor readout in progress": str(statusbits[5]),
                "Sensor readout complete": str(statusbits[6]),
                "SRAM readout started": str(statusbits[7]),
                "SRAM readout complete": str(statusbits[8]),
                "High-speed timing configured": str(statusbits[9]),
                "All ADCs configured": str(statusbits[10]),
                "All DACs configured": str(statusbits[11]),
                "Timer has reset": str(statusbits[13]),
                "Camera is Armed": str(statusbits[14]),
                "FPA_IF_TO": str(statusbits2[0]),
                "SRAM_RO_TO": str(statusbits2[1]),
                "PixelRd Timeout Error": str(statusbits2[2]),
                "UART_TX_TO_RST": str(statusbits2[3]),
                "UART_RX_TO_RST": str(statusbits2[4]),
                "PDBIAS Unready": str(statusbits2[5]),
            }
        )

        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            senslabs = {
                3: "W3_Top_A_Edge1 detected",
                4: "W3_Top_B_Edge1 detected",
                12: "HST_All_W_En detected",
            }
        else:
            senslabs = {
                3: "RSLROWOUTA",
                4: "RSLROWOUTB",
                12: "RSLNALLWENB",
                15: "RSLNALLWENA",
                16: "Config HST is done",
            }
        sensDict = {senslabs[x]: str(statusbits[x]) for x in senslabs.keys()}

        DACDict = OrderedDict()
        MonDict = OrderedDict()
        for entry in self.subreg_aliases:
            if self.subreg_aliases[entry][0] == "D":
                val = str(round(self.ca.getPotV(entry), 3))
                DACDict["DAC_" + entry] = val
            else:
                val = str(round(self.ca.getMonV(entry), 3))
                MonDict[entry] = val

        regDict = OrderedDict()
        for key in self.registers.keys():
            # Load in all registers except for the read-clear status registers.
            if key == "STAT_REG_SRC" or key == "STAT_REG2_SRC":
                pass
            else:
                err, rval = self.ca.getRegister(key)
                regDict[key] = rval

        dumpDict = OrderedDict()
        for x in [statDict, sensDict, MonDict, DACDict, regDict]:
            dumpDict.update(x)
        return dumpDict


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.15 C:/Users/hill35/git/camera_python/nsCamera/CameraAssembler.py File Reference

Classes

- class nsCamera.CameraAssembler.CameraAssembler

Namespaces

- namespace nsCamera
- namespace nsCamera.CameraAssembler

7.16 CameraAssembler.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
CameraAssembler assembles the separate camera parts into a camera object. This object
controls a combination of three components:

1. board : FPGA board -- LLNL_V1, LLNL_V4
2. comms: communication interface -- GigE, RS422
3. sensor : sensor type -- icarus, icarus2, daedalus

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from __future__ import absolute_import

import importlib
import inspect
import logging
import os
import platform
import socket
import sys
import time
import h5py
from datetime import datetime

import numpy as np

from nsCamera.utils.misc import (
    bytes2str,
    checkCRC,
    flattenlist,
    generateFrames,
    getEnter,
    partition,
    plotFrames,
    saveTiffs,
    str2bytes,
    str2nparray,
)

from nsCamera.utils.Packet import Packet

# TODO: move to Sphinx documentation
# TODO: add pytest and tox scripts


class CameraAssembler:
    """
    Code to assemble correct code to manage FPGA, frame grabber, and sensor

    Exposed methods:
        initialize() - initializes board registers and pots, sets up sensor
        reinitialize() - initialize board and sensors, restore last known timer settings
        reboot() - perform software reset of board and reinitialize
        getBoardInfo() - parses FPGA_NUM register to retrieve board description
        getRegister(regname) - retrieves contents of named register
        setRegister(regname, string) - sets named register to given value
        getSubregister(subregname) - return substring of register identified in board
          attribute 'subregname'
        setSubregister(subregname, valstring) - replace substring of register identified
          in board attribute 'subregname' with 'valstring'
        submitMessages(messages) - set registers or subregisters based on list of
          destination/payload tuples
        getPot(potname) - returns float (0 < value < 1) corresponding to integer stored
```

```
          in pot or monitor 'potname'
        setPot(potname, value) - 0 < value < 1; sets named pot to fixed-point number =
          'value' * (maximum pot value)
        getPotV(potname) - returns voltage setting of 'potname'
        setPotV(potname, voltage) - sets named pot to voltage
        getMonV(monname) - returns voltage read by monitor 'monname' (or monitor
          associated with given potname)
        readImgs() - calls arm() and readoff() functions
        saveFrames(frames) - save image object as one file
        saveNumpys(frames) - save individual frames as numpy data files
        dumpNumpy(datastream) - save datastream string to numpy file
        powerCheck(delta) - check that board power has not failed
        printBoardInfo() - print board information derived from FPGA_NUM register
        dumpRegisters() - return contents of all board registers
        dumpSubregisters() - return contents of all named board subregisters
        setFrames(min, max) - select subset of frames for readoff
        setRows(min, max, padToFull) - select subset of rows for readoff
        abortReadoff() - cancel readoff in wait-for-SRAM loop
        batchAquire() - fast acquire a finite series of images
        loadTextFrames() - load data sets previously saved as text and convert to frames

    Includes aliases to board- and sensor- specific methods:
        Board methods
            disarm() - take camera out of waiting-for-trigger state
            clearStatus() - clear contents of status registers
            checkStatus() - print contents of status register as reversed bit string
            checkStatus2() - print contents of status register 2 as reversed bit string
            reportStatus() - print report on contents of status registers
            resetTimer() - reset on-board timer
            getTimer() - read on-board timer
            enableLED(status) - enable (default) or disable (status = 0) on-board LEDs
            setLED(LED#, status) - turn LED on (default) or off (status = 0)
            setPowerSave(status) - turn powersave functionality on (default) or off
              (status = 0)
            getTemp() - read on-board temperature sensor
            getPressure() - read on-board pressure sensor
            dumpStatus() - generate dictionary of status, register, and subregister
              contents
        Sensor methods
            checkSensorVoltStat() - checks that jumper settings match sensor selection
            setTiming(side, sequencetuple, delay) - configure high-speed timing
            setArbTiming(side, sequencelist) - configure arbitrary high-speed timing
              sequence
            getTiming(side) - returns high speed timing settings from registers
            setManualTiming() - configures manual shutter timing
            getManualTiming() - returns manual shutter settings from registers
            selectOscillator(osc) - select timing oscillator
            setInterlacing(ifactor) - sets interlacing factor
            setHighFullWell(flag) - controls High Full Well mode
            setZeroDeadTime(flag, side) - controls Zero Dead Time mode
            setTriggerDelay(delayblocks) - sets trigger delay
        Comms methods
            sendCMD(pkt)- sends packet object via serial port
            arm() - configures software buffers & arms camera
            readFrames() - waits for data ready flag, then downloads image data
            readoff() - waits for data ready flag, then downloads image data
            closeDevice() - disconnect interface and release resources
        Miscellaneous functions (bare functions that can be called as methods)
            saveTiffs(frames) - save individual frames as tiffs
            plotFrames(frames) - plot individual frames as tiffs

    Informational class variables:
        version - nsCamera software version
        FPGAVersion - firmware version (date)
        FPGANum - firmware implementation identifier
        FPGAboardtype - FPGA self-identified board type (should match 'boardname')
        FPGArad = Flag indicating radiation-tolerant FPGA build
        FPGAsensor = FPGA self-identified sensor family (should correspond to
          'sensorname')
        FPGAinterfaces = FPGA self-identified interfaces (list should include
          'commname')
        FPGAinvalid = flag indicating invalid FPGA information in register
    """

    def __init__(
        self,
        boardname="llnl_v4",
        commname="GigE",
        sensorname="icarus2",
        verbose=4,
        port=None,
        ip=None,
        logfile=None,
        logtag=None,
        timeout=30,
    ):
        """
        Args:
            boardname: name of FPGA board: llnl_v1, llnl_v4
            commname: name of communication interface: rs422, gige
            sensorname: name of sensor: icarus, icarus2, daedalus
            verbose: optional, sets logging level
                0: print no logging messages
                1: print CRITICAL logging messages (camera will not operate, e.g.,
                  unable to connect to board)
                2: print ERROR logging messages (camera will not operate as directed,
                  e.g., an attempt to set the timing mode has failed, but the camera
                  is still operational)
                3: print WARNING logging messages (camera will operate as directed, but
                  perhaps not as expected, e.g., ca.setTiming('A', (9, 8), 1) may be
                  programmed correctly, but the actual timing generated by the board
                  will be {1} [9, 8, 9, 14, 9, 8, 9])
                4: print INFO logging messages (operational messages from ordinary
                  camera operation)
                5. print DEBUG logging messages (detailed messages describing specific
                  operations and messages)
            port: optional integer
                When using RS422, this preselects the comport for RS422 and bypasses
                  port search
                When using GigE, this preselects the OrangeTree control port for GigE
                  (ignored if an ip parameter is not also provided)
            ip: optional string (e.g., '192.168.1.100')
                GigE: bypasses network search and selects particular OrangeTree board -
                  required for some operating systems
            logfile: optional string, name of file to divert console output
            timeout: timeout in seconds for connecting using Gigabit Ethernet
        """
        self.version = "2.1.2"
        self.currtime = 0
        self.oldtime = 0
        self.trigtime = []
        self.waited = []
        self.read = []
        self.unstringed = []
        self.parsedtime = []
        self.savetime = []
        self.cycle = []
        self.boardname = boardname.lower()
        self.timeout = timeout
        # TODO: parse boardname, etc. in separate method
        if self.boardname in ["llnlv1", "v1", "1", 1]:
            self.boardname = "llnl_v1"
        if self.boardname in ["llnlv4", "v4", "4", 4]:
            self.boardname = "llnl_v4"
        self.commname = commname.lower()
        if self.commname[0] == "g" or self.commname[0] == "e":
            self.commname = "gige"
        if self.commname[0] == "r":
            self.commname = "rs422"
        self.sensorname = sensorname.lower()
        if self.sensorname in ["i1", "ic1", "icarus1"]:
            self.sensorname = "icarus"
        if self.sensorname in ["i2", "ic2"]:
            self.sensorname = "icarus2"
        if self.sensorname == "d":
            self.sensorname = "daedalus"
        self.verbose = int(verbose)
        self.port = port
        self.python, self.pyth1, self.pyth2, _, _ = sys.version_info
        self.PY3 = self.python >= 3
        self.platform = platform.system()
        self.arch, _ = platform.architecture()

        self.FPGAVersion = ""
        self.FPGANum = ""
        # FPGA information here and below populated during initialization using
        #   getBoardInfo
        self.FPGAboardtype = ""
        self.FPGArad = False
        self.FPGAsensor = ""
        self.FPGAinterfaces = []

        # indicates invalid FPGA information in register# (0x80000001 accepted as valid)
        self.FPGAinvalid = False

        self.iplist = None
        self.packageroot = os.path.dirname(inspect.getfile(CameraAssembler))
        self.armed = False

        # only one of these collections (senstiming, sensmanual) should be nonempty at
        #   any given time
        self.senstiming = {}  # preserve HST setting against possible power failure
        self.sensmanual = []  # preserve manual timing
        self.inittime = 0
        self.padToFull = False
        self.abort = False

        self.verbmap = {
            0: 99,
            1: logging.CRITICAL,
            2: logging.ERROR,
            3: logging.WARNING,
            4: logging.INFO,
            5: logging.DEBUG,
        }
        if logtag is None:
            logtag = ""
        self.logtag = logtag

        self.logcritbase = "CRITICAL {logtag}: ".format(logtag=logtag)
        self.logerrbase = "ERROR {logtag}: ".format(logtag=logtag)
        self.logwarnbase = "WARNING {logtag}: ".format(logtag=logtag)
        self.loginfobase = "INFO {logtag}: ".format(logtag=logtag)
        self.logdebugbase = "DEBUG {logtag}: ".format(logtag=logtag)

        self.logcrit = "{lb}[CA]".format(lb=self.logcritbase)
        self.logerr = "{lb}[CA]".format(lb=self.logerrbase)
        self.logwarn = "{lb}[CA]".format(lb=self.logwarnbase)
        self.loginfo = "{lb}[CA]".format(lb=self.loginfobase)
        self.logdebug = "{lb}[CA]".format(lb=self.logdebugbase)

        self.verblevel = self.verbmap.get(verbose, 5)  # defaults to 5 for invalid entry

        if logfile:
            logging.basicConfig(format="%(message)s", filename=logfile)
        else:
            logging.basicConfig(format="%(message)s")
        logging.getLogger().setLevel(self.verblevel)
        logging.getLogger("matplotlib.font_manager").disabled = True
        logging.debug(
            "{logdebug}CameraAssembler: boardname = {boardname}; commname = {commname};"
            " sensorname = {sensorname}; verbose = {verbose}; port = {port}; ip = {ip};"
            " logfile = {logfile}; logtag = {logtag}".format(
                logdebug=self.logdebug,
                boardname=boardname,
                commname=commname,
                sensorname=sensorname,
                verbose=verbose,
                port=port,
                ip=ip,
                logfile=logfile,
                logtag=logtag,
            )
        )

        if ip:
            try:
                iphex = socket.inet_aton(ip)
            except socket.error:
                logging.critical(
                    "{logcrit}CameraAssembler: invalid IP provided".format(
                        logcrit=self.logcrit
                    )
                )
                sys.exit(1)
            ipnum = [0, 0, 0, 0]
            for i in range(4):
                if self.PY3:
                    ipnum[i] = iphex[i]
                else:
                    ipnum[i] = int(iphex[i].encode("hex"), 16)
            self.iplist = ipnum

        self.payloaderror = False

        # code pulled out of __init__ to facilitate reinitialization of the board
        #   without needing to instantiate a new CameraAssembler object
        self.initialize()



    def initBoard(self):
        return self.board.initBoard()

    def initPots(self):
        return self.board.initPots()

    def latchPots(self):
        return self.board.latchPots()

    def initSensor(self):
        return self.board.initSensor()

    def configADCs(self):
        return self.board.configADCs()

    def disarm(self):
        return self.board.disarm()

    def startCapture(self, mode):
        return self.board.startCapture(mode)

    def readSRAM(self):
        return self.board.readSRAM()

    def waitForSRAM(self, timeout=None):
        return self.board.waitForSRAM(timeout)

    def getTimer(self):
        return self.board.getTimer()

    def resetTimer(self):
        return self.board.resetTimer()

    def enableLED(self, status=1):
        return self.board.enableLED(status)

    def setLED(self, LED=1, status=1):
        return self.board.setLED(LED, status)

    def setPowerSave(self, status=1):
        return self.board.setPowerSave(status)

    def setPPER(self, pollperiod=None):
        return self.board.setPPER(pollperiod)

    def getTemp(self, scale=None):
        return self.board.getTemp(scale)

    def getPressure(self, offset=None, sensitivity=None, units=None):
        return self.board.getPressure(offset, sensitivity, units)

    def clearStatus(self):
        return self.board.clearStatus()

    def checkStatus(self):
        return self.board.checkStatus()

    def checkStatus2(self):
        return self.board.checkStatus2()

    def reportStatus(self):
        return self.board.reportStatus()

    def reportEdgeDetects(self):
        return self.board.reportEdgeDetects()

    def dumpStatus(self):
        return self.board.dumpStatus()

    def checkSensorVoltStat(self):
        return self.sensor.checkSensorVoltStat()

    def setTiming(self, side="AB", sequence=None, delay=0):
        return self.sensor.setTiming(side, sequence, delay)

    def setArbTiming(self, side="AB", sequence=None):
        return self.sensor.setArbTiming(side, sequence)

    def getTiming(self, side=None, actual=None):
        return self.sensor.getTiming(side, actual)

    def setManualShutters(self, timing=None):
        return self.sensor.setManualTiming(timing)

    def setManualTiming(self, timing=None):
        return self.sensor.setManualTiming(timing)

    def getManualTiming(self):
        return self.sensor.getManualTiming()

    def getSensTemp(self, scale=None, offset=None, slope=None, dec=1):
        return self.sensor.getSensTemp(scale, offset, slope, dec)

    def sensorSpecific(self):
        return self.sensor.sensorSpecific()

    def selectOscillator(self, osc=None):
        return self.sensor.selectOscillator(osc)

    def setInterlacing(self, ifactor=None, side=None):
        return self.sensor.setInterlacing(ifactor, side)

    def setHighFullWell(self, flag=True):
        return self.sensor.setHighFullWell(flag)

    def setZeroDeadTime(self, flag=True, side=None):
        return self.sensor.setZeroDeadTime(flag, side)

    def setTriggerDelay(self, delay=0):
        return self.sensor.setTriggerDelay(delay)

    def setPhiDelay(self, side=None, delay=0):
        return self.sensor.setPhiDelay(side, delay)

    def setExtClk(self, dilation=None, frequency=None):
        return self.sensor.setExtClk(dilation, frequency)

    def parseReadoff(self, frames, columns=1):
        return self.sensor.parseReadoff(frames, columns)

    def sendCMD(self, pkt):
        return self.comms.sendCMD(pkt)

    def arm(self, mode=None):
        return self.comms.arm(mode)

    def readFrames(self, waitOnSRAM=None, timeout=0, fast=False, columns=1):
        frames, _, _ = self.comms.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

    def readoff(self, waitOnSRAM=None, timeout=0, fast=None, columns=1):
        return self.comms.readoff(waitOnSRAM, timeout, fast, columns)

    def writeSerial(self, cmd, timeout=None):
        return self.comms.writeSerial(cmd, timeout)

    def readSerial(self, size, timeout=None):
        return self.comms.readSerial(size, timeout)

    def closeDevice(self):
        return self.comms.closeDevice()

    def saveTiffs(self, frames, path=None, filename="Frame", prefix=None, index=None):
        return saveTiffs(self, frames, path, filename, prefix, index)

    def plotFrames(self, frames, index=None):
        return plotFrames(self, frames, index)

    def getEnter(self, text):
        return getEnter(text)

    def checkCRC(self, rval):
        return checkCRC(rval)

    def str2bytes(self, astring):
        return str2bytes(astring)

    def bytes2str(self, bytesequence):
        return bytes2str(bytesequence)

    def str2nparray(self, valstring):
        return str2nparray(valstring)

    def flattenlist(self, mylist):
        return flattenlist(mylist)

    def partition(self, frames, columns):
        return partition(self, frames, columns)



    def initialize(self):
        """
        Initialize board registers and set pots
        """
        # TODO: automate sensor and board selection from firmware info


        # get sensor
        # TODO: pull sensor, board, comm id out to separate methods
        if self.sensorname == "icarus":
            import nsCamera.sensors.icarus as snsr
        elif self.sensorname == "icarus2":
            import nsCamera.sensors.icarus2 as snsr
        elif self.sensorname == "daedalus":
            import nsCamera.sensors.daedalus as snsr
        else:  # catch-all for added sensors to attempt object encapsulation
            sensormodname = ".sensors." + self.sensorname
            try:
                sensormod = importlib.import_module(sensormodname, "nsCamera")
            except ImportError:
                logging.critical(self.logcrit + "invalid sensor name")
                sys.exit(1)
            snsr = getattr(sensormod, self.sensorname)
        self.sensor = snsr(self)

        # kill existing connections (for reinitialize)
        if hasattr(self, "comms"):
            self.closeDevice()

        # get communications interface
        if self.commname == "rs422":
            import nsCamera.comms.RS422 as comms
        elif self.commname == "gige":
            import nsCamera.comms.GigE as comms
        else:
            commsmodname = ".comms." + self.commname
            try:
                commsmod = importlib.import_module(commsmodname, "nsCamera")
            except ImportError:
                logging.critical(self.logcrit + "invalid comms name")
                sys.exit(1)
            comms = getattr(commsmod, self.commname)
        self.comms = comms(self)

        # get board
        if self.boardname == "llnl_v1":
            import nsCamera.boards.LLNL_v1 as brd

            self.board = brd.llnl_v1(self)
        elif self.boardname == "llnl_v4":
            import nsCamera.boards.LLNL_v4 as brd

            self.board = brd.llnl_v4(self)
        else:
            boardmodname = ".board." + self.boardname
            try:
                boardmod = importlib.import_module(boardmodname, "nsCamera")
            except ImportError:
                logging.critical(self.logcrit + "invalid board name")
                sys.exit(1)
            boardobj = getattr(boardmod, self.boardname)
            self.board = boardobj(self)

        # Now that board exists, initialize board-specific aliases for sensors
        self.sensor.init_board_specific()



        # TODO: make cython the standard version
        # ###############
        # # For cython version
        #
        # # get sensor
        # if self.sensorname == "icarus":
        #     import nsCamera.sensors.icarus as snsr
        #     self.sensor = snsr.icarus(self)
        # elif self.sensorname == "icarus2":
        #     import nsCamera.sensors.icarus2 as snsr
        #     self.sensor = snsr.icarus2(self)
        # elif self.sensorname == "daedalus":
        #     import nsCamera.sensors.daedalus as snsr
        #     self.sensor = snsr.daedalus(self)
        #
        # # kill existing connections (for reinitialize)
        # if hasattr(self, "comms"):
        #     self.closeDevice()
        #
        # # get communications interface
        # if self.commname == "rs422":
        #     import nsCamera.comms.RS422 as comms
        #     self.comms = comms.RS422(self)
        # elif self.commname == "gige":
        #     import nsCamera.comms.GigE as comms
        #     self.comms = comms.GigE(self)
        #
        # # get board
        # if self.boardname == "llnl_v1":
        #     import nsCamera.boards.LLNL_v1 as brd
        #     self.board = brd.llnl_v1(self)
        # elif self.boardname == "llnl_v4":
        #     import nsCamera.boards.LLNL_v4 as brd
        #     self.board = brd.llnl_v4(self)
        # ###############

        err, rval = self.getRegister("FPGA_NUM")
        if err or rval == "":
            err, rval = self.getRegister("FPGA_NUM")
            if err or rval == "":
                logging.critical(
                    self.logcrit + "Initialization failed: unable to communicate with"
                    " board. "
                )
            sys.exit(1)

        self.initBoard()
        self.initPots()
        self.initSensor()
        self.initPowerCheck()
        self.getBoardInfo()
        self.printBoardInfo()

    def reinitialize(self):
        """
        Reinitialize board registers and pots, reinitialize sensor timing (if
          previously set)
        """
        logging.info(self.loginfo + "reinitializing")
        self.initialize()

        for side in self.senstiming:
            self.setTiming(side, self.senstiming[side][0], self.senstiming[side][1])

        if self.sensmanual:  # should be mutually exclusive with anything in senstiming
            self.setManualShutters(self.sensmanual)

    def reboot(self):
        """
        Perform soft reboot on board and reinitialize
        """
        self.board.softReboot()
        self.reinitialize()

    def getBoardInfo(self):
        """
        Get board info from FPGA_NUM register. Returns error flag if register contents
```

```
          are invalid and tuple (board version number, rad tolerance flag, sensor name)

        Returns:
            tuple (errorFlag, (board version, rad tolerance flag, sensor name))
        """
        invalidFPGANum = False
        interfaces = []

        # TODO: move to new method (combine with parsing from initialize)
        if int(self.FPGANum[0], 16) & 8:
            if self.FPGANum[1] == "1":
                boardtype = "LLNLv1"
            elif self.FPGANum[1] == "4":
                boardtype = "LLNLv4"
            else:
                boardtype = "LLNLv?"
                invalidFPGANum = True
        else:
            boardtype = "SNLrevC"
            logging.warning(
                self.logwarn + "FPGA self-identifies as SNLrevC, which is not"
                " supported by this software "
            )
            invalidFPGANum = True
        self.FPGAboardtype = boardtype

        if int(self.FPGANum[6], 16) & 1:
            rad = True
        else:
            rad = False
        self.FPGArad = rad

        if self.FPGANum[7] == "1":
            sensor = "Icarus"
        elif self.FPGANum[7] == "2":
            sensor = "Daedalus"
        else:
            sensor = "Undefined"
            invalidFPGANum = True
        self.FPGAsensor = sensor

        if int(self.FPGANum[5], 16) & 1:
            interfaces.append("RS422")
        if int(self.FPGANum[5], 16) & 2:
            interfaces.append("GigE")
        self.FPGAinterfaces = interfaces

        if invalidFPGANum:
            if self.FPGANum == "80000001":
                invalidFPGANum = False
            else:
                logging.warning(self.logwarn + "FPGA self-identification is invalid")
        self.FPGAinvalid = invalidFPGANum

        return invalidFPGANum, (boardtype, rad, sensor)

    def getRegister(self, regname):
        """
        Retrieves contents of named register as hexadecimal string without '0x'

        Args:
            regname: name of register as given in ICD

        Returns:
            tuple: (error string, register contents as hexadecimal string without '0x')
        """
        # logging.debug(self.logdebug + "getRegister: regname = " + str(regname))
        logging.debug(
            "{logdebug}getRegister: regname = {regname}".format(
                logdebug=self.logdebug, regname=regname
            )
        )

        regname = regname.upper()
        if regname not in self.board.registers:
            err = "{logerr}getRegister: Invalid register name: {regname}; returning"
            " zeros".format(logerr=self.logerr, regname=regname)
            logging.error(err)
            return err, "00000000"
        sendpkt = Packet(cmd="1", addr=self.board.registers[regname])
        err, rval = self.comms.sendCMD(sendpkt)
        if err:
            logging.error(
                "{logerr}getRegister: {regname}; {err}".format(
                    logerr=self.logerr, regname=regname, err=err
                )
            )

        retval = rval[8:16]
        logging.debug(
            "{logdebug}getRegister: retval = {retval}".format(
                logdebug=self.logdebug, retval=retval
            )
        )

        return err, retval

    def setRegister(self, regname, regval):
        """
        Sets named register to given value

        Args:
            regname: name of register as given in ICD
            regval: value to assign to register, as integer or hexadecimal string
              with or without ’0x’

        Returns:
            tuple: (error string, response string)
        """
        logging.debug(
            "{logdebug}setRegister: regname = {regname}; regval = {regval}".format(
                logdebug=self.logdebug, regname=regname, regval=regval
            )
        )

        regname = regname.upper()
        if regname not in self.board.registers:
            err = "{logerr}setRegister: Invalid register name: {regname}".format(
                logerr=self.logerr, regname=regname
            )
            logging.error(err)
            return err, "00000000"
        if isinstance(regval, int):
            regval = hex(regval)
        try:
            if regval[0:2] == "0x":
                regval = regval[2:]
        except TypeError:
            err = "{logerr}setRegister: invalid register value parameter".format(
                logerr=self.logerr
            )
            logging.error(err)
            return err, "00000000"
        pkt = Packet(addr=self.board.registers[regname], data=regval)
        err, rval = self.comms.sendCMD(pkt)
        if err:
            logging.error(
                "{logerr}setRegister: {regname}: {err}".format(
                    logerr=self.logerr, regname=regname, err=err
                )
            )
        if len(rval) < 32:
            logging.debug(
                "{logdebug}SetRegister: rval = {rval}".format(
                    logdebug=self.logdebug, rval=rval
                )
            )
        else:
            logging.debug(
                "{logdebug}SetRegister: rval (truncated)= {rval}".format(
                    logdebug=self.logdebug, rval=rval[0:32]
                )
            )
        return err, rval

    def resolveSubreg(self, srname):
        """
        Resolves subregister name or alias, returns object associated with subregister
          and flag indicating writability

        Args:
            srname: name or alias of subregister

        Returns:
            tuple(subregister name string, associated object, writable flag)
        """
        logging.debug(
            "{logdebug}resolveSubreg: srname = {srname}".format(
                logdebug=self.logdebug,
                srname=srname,
            )
        )
        writable = False
        srname = srname.upper()
        if srname in self.board.subreg_aliases:
            srname = self.board.subreg_aliases[srname].upper()
        if srname in self.board.subreglist:
            srobj = getattr(self.board, srname)
            writable = getattr(self.board, srname).writable
        else:
            # No-object error is handled by calling function
            srobj = None
        logging.debug(
            "{logdebug}resolveSubreg: srobj = {srobj}, writable={writable}".format(
                logdebug=self.logdebug, srobj=srobj, writable=writable
            )
        )

        return srname, srobj, writable

    def getSubregister(self, subregname):
        """
        Returns substring of register identified in board attribute ’subregname’

        Args:
            subregname: listed in board.subreg_aliases or defined in board.subregisters

        Returns:
            tuple: (error string, contents of subregister as binary string without ’0b’)
        """
        logging.debug(
            "{logdebug}getSubegister: subregname = {subregname}".format(
                logdebug=self.logdebug,
                subregname=subregname,
            )
        )

        subregname, subregobj, _ = self.resolveSubreg(subregname)
        if not subregobj:
            err = "{logerr}getSubregister: invalid lookup: {subregname}; returning"
            " string of zeroes".format(logerr=self.logerr, subregname=subregname)

            logging.error(err)
            return err, "".zfill(8)
        err, resp = self.getRegister(subregobj.register)
        if err:
            logging.error(
                "{logerr}getSubregister: unable to retrieve register setting: \
                {subregname}; returning ’0’ string".format(
                    logerr=self.logerr, subregname=subregname
                )
            )

            return err, "".zfill(8)
        hex_str = "0x" + resp  # this should be a hexadecimalstring
        b_reg_value = "{0:0=32b}".format(int(hex_str, 16))  # convert to binary string
        # list indexing is reversed from bit string; the last bit of the string is at
        #   index 0 in the list (thus bit 0 is at index 0)
        startindex = 31 - subregobj.start_bit
        retval = b_reg_value[startindex : startindex + subregobj.width]
        logging.debug(
            "{logdebug}getSubregister: retval = {retval}".format(
                logdebug=self.logdebug, retval=retval
            )
        )
        return "", retval

    def setSubregister(self, subregname, valstring):
        """
        Sets substring of register identified in board attribute ’subregname’ to
          valstring if subregister is writable

        Args:
            subregname: listed in board.subreg_aliases or defined in board.subregisters
            valstring: integer or binary string with or without '0b'

        Returns:
            tuple: (error, packet response string) from setRegister
        """
        logging.debug(
            "{logdebug}setSubegister: subregname = {subregname}; valstring ="
            " {valstring}".format(
                logdebug=self.logdebug, subregname=subregname, valstring=valstring
            )
        )

        subregname, subregobj, writable = self.resolveSubreg(subregname)
        if not subregobj:
            err = "{logerr}getSubregister: invalid lookup: {subregname}".format(
                logerr=self.logerr, subregname=subregname
            )

            logging.error(err)
            return err, "0"
        if not writable:
            err = "{logerr}getSubregister: not a writable subregister: {subregname}"
            "".format(logerr=self.logerr, subregname=subregname)
            logging.error(err)
            return err, "0"
        if isinstance(valstring, int):
            valstring = bin(valstring)[2:]
        try:
            if valstring[0:2] == "0b":
                valstring = valstring[2:]
        except TypeError:
            err = "{logerr}getSubregister: invalid subregister value parameter".format(
                logerr=self.logerr
            )

            logging.error(err)
            return err, "0"
        if len(str(valstring)) > subregobj.width:
            err = "{logerr}getSubregister: ialue string is too long".format(
                logerr=self.logerr
            )

            logging.error(err)
            return err, "0"
        # read current value of register data
        err, resp = self.getRegister(subregobj.register)
        if err:
            logging.error(
                "{logerr}getSubregister: unable to retrieve register setting; setting"
                " of {subregname} likely failed ".format(
                    logerr=self.logerr, subregname=subregname
                )
            )

            return err, "0"
        hex_str = "0x" + resp
        b_reg_value = "{0:0=32b}".format(int(hex_str, 16))  # convert to binary
        # list indexing is reversed from bit string; the last bit of the string is at
        #   index 0 in the list (thus bit 0 is at index 0)
        startindex = 31 - subregobj.start_bit
        valstringpadded = str(valstring).zfill(subregobj.width)
        fullreg = list(b_reg_value)
        fullreg[startindex : startindex + subregobj.width] = valstringpadded
        # convert binary string back to hexadecimal string for writing
        new_reg_value = "".join(fullreg)
        h_reg_value = "{num:{fill}{width}x}".format(
            num=int(new_reg_value, 2), fill="0", width=8
        )
        err, retval = self.setRegister(subregobj.register, h_reg_value)
        # logging.debug(self.logdebug + "retval = " + str(retval))
        if len(retval) < 32:
            logging.debug(
                "{logdebug}setSubregister: retval = {retval}".format(
                    logdebug=self.logdebug, retval=retval
                )
            )
        else:
            logging.debug(
                "{logdebug}setSubregister: retval (truncated) = {retval}".format(
                    logdebug=self.logdebug, retval=retval[0:32]
                )
            )

        return err, retval

    def submitMessages(self, messages, errorstring="Error"):
        """
        Serially set multiple register / subregister values

        Args:
            messages: list of tuples (register name, integer or hexadecimal string with
              or without '0x') and/or (subregister name, integer or binary string with
              or without '0b')
            errorstring: error message to print in case of failure

        Returns:
            tuple (accumulated error string, response string of final message)
        """
        logging.debug(
            "{logdebug}submitMessages: messages = {messages}; errorstring ="
            " {errorstring}".format(
                logdebug=self.logdebug, messages=messages, errorstring=errorstring
            )
        )

        errs = ""
        err = ""
        rval = ""
        for m in messages:
            if m[0].upper() in self.board.registers:
                err, rval = self.setRegister(m[0].upper(), m[1])
            elif m[0].upper() in self.board.subreglist:
                err, rval = self.setSubregister(m[0].upper(), m[1])
            else:
                err = "{logerr}submitMessages: Invalid register/subregister:"
                " {errorstring}:{m0}; ".format(
                    logerr=self.logerr, errorstring=errorstring, m0=m[0]
                )

                logging.error(err)
            errs = errs + err
        return err, rval

    def getPot(self, potname, errflag=False):
        """
        Retrieves value of pot or ADC monitor subregister, scaled to [0,1).

        Args:
            potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of subregister, scaled to [0,1) )
            else:
                float value of subregister, scaled to [0,1)
        """
        logging.debug(
            "{logdebug}getPot: potname = {potname}; errflag = {errflag}".format(
                logdebug=self.logdebug, potname=potname, errflag=errflag
            )
        )

        potname, potobj, _ = self.resolveSubreg(potname)
        if not potobj:
            err = "{logerr}getPot: invalid lookup: {potname}; returning 0".format(
                logerr=self.logerr, potname=potname
            )

            logging.error(err)
            if errflag:
                return err, 0
            return 0
        err, b_pot_value = self.getSubregister(potname)
        if err:
            err = "{logerr}getPot: unable to read subregister: {potname}".format(
                logerr=self.logerr, potname=potname
            )

        # convert binary string back to decimal
        f_reg_value = 1.0 * int(b_pot_value, 2)
        value = (f_reg_value - potobj.min) / (potobj.max - potobj.min)
        # logging.debug(self.logdebug + "getpot: value = " + str(value))

        logging.debug(
            "{logdebug}getpot: value =  {value}".format(
                logdebug=self.logdebug, value=value
            )
        )

        if errflag:
            return err, value
        return value

    def setPot(self, potname, value=1.0, errflag=False):
        """
        Sets value of pot to value, normalized so that  '1.0' corresponds with the
          fixed point maximum value of pot.

        Args:
            potname: common name of pot, e.g., VRST found in board.subreg_aliases or
              defined in board.subregisters
            value: float between 0 and 1
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, response packet as string)
            else:
                response packet as string
        """
        logging.debug(
            "{logdebug}setPot: potname = {potname}; value={value} errflag = {errflag}"
            "".format(
                logdebug=self.logdebug, potname=potname, value=value, errflag=errflag
            )
        )

        if value < 0:
            value = 0.0
        if value > 1:
            value = 1.0

        potname, potobj, writable = self.resolveSubreg(potname)
        if not potobj:
            err = "{logerr}setPot: invalid lookup: {potname}; returning '0'".format(
                logerr=self.logerr, potname=potname
            )

            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if not writable:
            err = "{logerr}setPot: not a writable subregister: {potname}; returning '0'"
            "".format(logerr=self.logerr, potname=potname)
            logging.error(err)
            if errflag:
                return err, "0"
            return "0"
        setpoint = int(round(value * potobj.max_value))
        setpointpadded = "{num:{fill}{width}b}".format(
            num=setpoint, fill="0", width=potobj.width
        )
        logging.debug(
            "{logdebug}setpot: setpointpadded =  {setpointpadded}".format(
                logdebug=self.logdebug, setpointpadded=setpointpadded
            )
        )

        err, rval = self.setSubregister(potname, setpointpadded)
        if err:
            logging.error(
                err="{logerr}setPot: unable to confirm setting of subregister:"
                " {potname}".format(logerr=self.logerr, potname=potname)
            )
        ident = potname[3:]
        if ident[0].isdigit():  # numbered pot scheme
            potnumlatch = int(ident) * 2 + 1
            potnumlatchstring = "{num:{fill}{width}x}".format(
                num=potnumlatch, fill="0", width=8
            )
            err1, resp = self.setRegister("POT_CTL", potnumlatchstring)
        else:  # alphabetical DAC scheme
            ident = ident.upper()  # expects single character, e.g. 'A' from 'DACA'
            identnum = ord(ident) - ord("A")  # DACA -> 0
            potnumlatch = int(identnum) * 2 + 1
            potnumlatchstring = "{num:{fill}{width}x}".format(
                num=potnumlatch, fill="0", width=8
            )
            err1, resp = self.setRegister("DAC_CTL", potnumlatchstring)
        if err1:
            # logging.error(self.logerr + "setPot: unable to latch register")

            logging.error(
                err="{logerr}setPot: unable to latch register".format(
                    logerr=self.logerr
                )
            )

        if errflag:
            return err + err1, rval
        return rval

    def getPotV(self, potname, errflag=False):
        """
        Reads voltage _setting_ (not actual voltage) of specified pot

        Args:
            potname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of pot voltage)
            else:
                float value of pot voltage
        """
        logging.debug(
            self.logdebug
            + "getPotV: potname = "
            + str(potname)
            + "; errflag = "
            + str(errflag)
        )
        potname, potobj, _ = self.resolveSubreg(potname)
        if not potobj:
            err = (
                self.logerr + "getPotV: invalid lookup: " + potname + " , returning 0 "
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        err, val = self.getPot(potname, errflag=True)
        logging.debug(self.logdebug + "getPotV: val = " + str(val))
        if err:
            logging.error(self.logerr + "getPotV: unable to read pot " + potname)
        minV = potobj.minV
        maxV = potobj.maxV
        if errflag:
            return err, val * (maxV - minV)
        return val * (maxV - minV)

    # TODO: optimize tuning speed for DACs
    def setPotV(
        self,
        potname,
        voltage,
        tune=False,
        accuracy=0.01,
        iterations=20,
        approach=0.75,
        errflag=False,
    ):
        """
        Sets pot to specified voltage. If tune=True, uses monitor to adjust pot to
          correct voltage. Tuning will attempt to tune to closest LSB on pot; if
          'accuracy' > LSB resolution, will only complain if tuning is unable to get
          the voltage within 'accuracy'

        Args:
            potname: common name of pot, e.g., VRST found in board.subreg_aliases or
              defined in board.subregisters
            voltage: voltage bound by pot max and min (set in board object)
            tune: if True, iterate with monitor to correct voltage
            accuracy: acceptable error in volts (if None, attempts to find the closest
              possible pot setting and warns if last iteration does not reduce error
              below the resolution of the pot)
            iterations: number of iteration attempts
            approach: approximation parameter (>1 may cause overshoot)
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, response string)
            else:
                response string
        """
        logging.debug(
            self.logdebug
            + "setPotV: potname = "
            + str(potname)
            + "; voltage = "
            + str(voltage)
            + "; tune = "
            + str(tune)
            + "; accuracy = "
            + str(accuracy)
            + "; iterations = "
            + str(iterations)
            + "; approach = "
            + str(approach)
            + "; errflag = "
            + str(errflag)
        )
        potname, potobj, writable = self.resolveSubreg(potname)
        if not potobj:
            err = (
                self.logerr
                + "setPotV: invalid lookup: "
                + potname
                + " , returning zero "
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if not writable:
            err = (
                self.logerr
                + "setPotV: not a writable subregister: "
                + potname
                + "; returning zero"
            )
            logging.error(err)
            if errflag:
                return err, 0
            return 0
        if voltage < potobj.minV:
            voltage = potobj.minV
        if voltage > potobj.maxV:
            voltage = potobj.maxV
        setting = (voltage - potobj.minV) / (potobj.maxV - potobj.minV)
        logging.debug(self.logdebug + "setPotV: setting = " + str(setting))
        err, rval = self.setPot(potname, setting, errflag=True)
        time.sleep(0.1)
        # TODO: refactor tuning to separate method
        if tune:
            logging.debug(self.logdebug + "setPotV: beginning tuning")
            if potname not in self.board.monitor_controls.values():
                err = (
                    self.logerr
                    + "setPotV: pot '"
                    + potname
                    + "' does not have a corresponding monitor"
                )
                logging.error(err)
                if errflag:
                    return err, rval
                return rval
            self.setPot(potname, 0.65)
            time.sleep(0.2)
            err1, mon65 = self.getMonV(potname, errflag=True)
            self.setPot(potname, 0.35)
            time.sleep(0.2)
            err2, mon35 = self.getMonV(potname, errflag=True)
            # theoretical voltage range assuming linearity
            potrange = (mon65 - mon35) / 0.3
            stepsize = potrange / (potobj.max_value + 1)
            err += err1 + err2
            if err or potrange < 1:
                err += self.logerr + "setPotV: unable to tune pot " + potname
                if potrange < 1:  # potrange should be on the order of 3.3 or 5 volts
                    err += "; monitor shows insufficient change with pot variation; "
                    "retrying setPotV with tune=False"
                logging.warning(err)
                err, rval = self.setPotV(
                    potname=potname, voltage=voltage, tune=False, errflag=True
                )
                if errflag:
                    return err, rval
                return rval
            potzero = 0.35 - (mon35 / potrange)
            potone = 1.65 - (mon65 / potrange)
            if potzero < 0:
                potzero = 0
            if potone > 1:
                potone = 1

            if accuracy > stepsize:
                mindiff = accuracy
            else:
                mindiff = stepsize
            setting = potzero + (voltage / potone)
            self.setPot(potname, setting)
            lastdiff = 0
            smalladjust = 0
            err3 = ""
            for _ in range(iterations):
                err3i, measured = self.getMonV(potname, errflag=True)
                if err3i:
                    err3 = err3 + err3i + " "
                diff = voltage - measured
                if abs(diff - lastdiff) < stepsize / 2:
                    if (
                        smalladjust > 12
                    ):  # magic number for now; if it doesn't converge after several
                        #   tries, it never will, usually because the setting is pinned
                        #   to 0 or 1 and adjust can't change it
                        logging.warning(
                            self.logwarn
                            + "setPotV: Tuning converged too slowly: pot "
                            + potname
                            + " set to "
                            + str(voltage)
                            + "V, monitor returns "
                            + str(measured)
                            + "V; if this value is incorrect, consider trying "
                            + "tune=False"
                        )
                        logging.debug(self.logdebug + "setPotV: tuning complete")
                        if errflag:
                            return "", rval
                        return rval
                    smalladjust += 1
                if not int(2 * diff / stepsize):
                    # TODO: is this check redundant with the first one?
                    logging.debug(self.logdebug + "setPotV: tuning complete")
                    if errflag:
                        return "", rval
                    return rval
                adjust = approach * (diff / potrange)
                setting += adjust
                if setting > 1:
                    setting = 1
                elif setting < 0:
                    setting = 0
                err1, rval = self.setPot(potname, setting, True)
                lastdiff = diff
                time.sleep(0.2)
            err4, measured = self.getMonV(potname, errflag=True)
            diff = voltage - measured
            # code will try to get to within one stepsize, but will only complain if it
            #   doesn't get within mindiff
            if int(diff / mindiff):
                logging.warning(
                    self.logwarn
                    + "setPotV: pot "
                    + potname
                    + " set to "
                    + str(voltage)
                    + "V, monitor returns "
                    + str(measured)
                    + "V"
                )
            err += err1 + err2 + err3 + err4
        if err:
            logging.error(self.logerr + "setPotV: errors occurred: " + err)
        if errflag:
            return err, rval
        logging.debug(self.logdebug + "setPotV: tuning complete")
        return rval

    def getMonV(self, monname, errflag=False):
        """
        Reads voltage from monitor named or that associated with the pot named 'monname'

        Args:
            monname: name of pot or monitor, e.g., VRST or MON_CH2 found in
              board.subreg_aliases or defined in board.subregisters
            errflag: if True, return tuple with error string

        Returns:
            if errflag:
                tuple: (error string, float value of voltage measured by monitor)
            else:
                float value of voltage measured by monitor
        """
        logging.debug(
            self.logdebug
            + "getMonV: monname = "
            + str(monname)
            + "; errflag = "
            + str(errflag)
        )
        monname = monname.upper()
        if monname in self.board.subreg_aliases:
            monname = self.board.subreg_aliases[monname].upper()
        # else:
        for key, value in self.board.monitor_controls.items():
            if value == monname:
                monname = key
        if monname not in self.board.monitor_controls:
            if monname in self.board.subreglist:
                pass  # no change necessary
            else:
                err = (
                    self.logerr + "getMonV: invalid lookup " + monname + ", returning 0"
                )
                logging.error(err)
                if errflag:
                    return err, 0
                return 0
        err, monval = self.getPot(monname, errflag=True)
        logging.debug(self.logdebug + "getMonV: monval = " + str(monval))
        if err:
            logging.error(
                self.logerr + "getMonV: unable to read monitor value for " + monname
            )
        if self.board.ADC5_bipolar:
            if monval >= 0.5:
                monval -= 1  # handle negative measurements (two's complement)
            if errflag:
                return err, 2 * self.board.ADC5_mult * monval * self.board.VREF
            return 2 * self.board.ADC5_mult * monval * self.board.VREF
        else:
            if errflag:
                return err, self.board.ADC5_mult * monval * self.board.VREF
            return self.board.ADC5_mult * monval * self.board.VREF

    def readImgs(self, waitOnSRAM=True, mode="Hardware"):
        """
        Combines arm() and readoff() functions

        Returns:
            tuple (list of numpy arrays, length of downloaded payload, payload error
              flag) returned by readoff
        """
        logging.info(self.loginfo + "readImgs")
        self.arm(mode)
        return self.readoff(waitOnSRAM)

    def saveFrames(self, frames, path=None, filename="frames", prefix=None):
        """
        Save list of numpy arrays to disk. If passed an unprocessed text string, saves
          it directly to disk for postprocessing. Use 'prefix=""' for no prefix

        Args:
            frames: numpy array or list of numpy arrays OR text string
            path: save path, defaults to './output'
            filename: defaults to 'frames.bin'
            prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')

        Returns:
            Error string
        """
        logging.debug(
            self.logdebug
            + "saveFrames: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
        )
        logging.info(self.loginfo + "saveFrames")
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)

        # TODO catch save file exceptions
        if isinstance(frames[0], str):
            logging.debug(self.logdebug + "saveFrames: saving text frames")
            filename = filename + ".txt"
            savefile = open(os.path.join(path, prefix + filename), "w+")
            savefile.write(frames)
        else:
            logging.debug(self.logdebug + "saveFrames: saving numerical frames")
            filename = filename + ".bin"
            stacked = np.stack(frames)
            try:
                stacked = stacked.reshape(
                    (
                        self.sensor.nframes,
                        self.sensor.height // (self.sensor.interlacing + 1),
                        self.sensor.width,
                    )
                )
            except Exception as e:
                err = self.logerr + "saveFrames: unable to save frames: " + str(e)
                logging.error(err)

            stacked.tofile(os.path.join(path, prefix + filename))
        return err

    def saveNumpys(
        self,
        frames,
        path=None,
        filename="Frame",
        prefix=None,
        index=None,
    ):
        """
        Save numpy array or list of numpy arrays to disk as individual numpy data files,
          with frame number appended to filename.

        Args:
            frames: numpy array or list of numpy arrays or single numpy array
            path: save path, defaults to './output'
            filename: defaults to 'Frame' followed by frame number
            prefix: prepended to 'filename', defaults to time/date
              (e.g. '160830-124704_')
            index: number to start frame numbering

        Returns:
            Error string
        """
        logging.info(self.loginfo + "saveNumpys")
        logging.debug(
            self.logdebug
            + "saveNumpys: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
            + "; index = "
            + str(index)
        )
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)
        if index is None:
            firstnum = self.sensor.firstframe
        else:
            firstnum = index
        if not isinstance(frames, list):
            frames = [frames]

        # if this is a text string from fast readoff, do the numpy conversion now
        if isinstance(frames[0], str):
            frames = generateFrames(frames)

        framestemp = np.copy(frames)

        for idx, frame in enumerate(framestemp):
            if idx < len(framestemp) / 2:
                interlacing = self.sensor.interlacing[0]
            else:
                interlacing = self.sensor.interlacing[1]
            try:
                if self.padToFull:
                    frame = np.reshape(
                        frame, (self.sensor.maxheight // (interlacing + 1), -1)
                    )
                else:
                    frame = np.reshape(
                        frame,
                        (
                            (self.sensor.lastrow - self.sensor.firstrow + 1)
                            // (interlacing + 1),
                            -1,
                        ),
                    )
                namenum = filename + "_%d" % firstnum
                nppath = os.path.join(path, prefix + namenum + ".npy")
                np.save(nppath, frame)
                firstnum += 1
            except SystemExit:
                raise
            except KeyboardInterrupt:
                raise
            except Exception:
                err = self.logerr + "saveNumpys: unable to save arrays"
                logging.error(err)
                continue
        return err

    def dumpNumpy(
        self,
        datastream,
        path=None,
        filename="Dump",
        prefix=None,
    ):
        """
        Datastream is converted directly to numpy array and saved to disk. No attempt to
          parse headers or separate into individual frames is made. The packet header is
          removed before saving

        Args:
            datastream: string to be saved
            path: save path, defaults to './output'
            filename: defaults to 'Dump'
            prefix: prepended to 'filename', defaults to time/date
              (e.g. '160830-124704_')

        Returns:
            Error string
        """
        logging.info(self.loginfo + "dumpNumpy")
        logging.debug(
            self.logdebug
            + "dumpNumpy: path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
        )
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = time.strftime("%y%m%d-%H%M%S_", time.localtime())
        if not os.path.exists(path):
            os.makedirs(path)
        npdata = str2nparray(datastream[36:])
        try:
            nppath = os.path.join(path, prefix + filename + ".npy")
            np.save(nppath, npdata)
        except SystemExit:
            raise
        except KeyboardInterrupt:
            raise
        except Exception:
            err = self.logerr + "dumpNumpy: unable to save data stream"
            logging.error(err)
        return err

    def checkRegSet(self, regname, teststring):
        """
        Quick check to confirm that data read from register matches data write

        Args:
            regname: register to test
            teststring: value to assign to register, as integer or hexadecimal string
              with or without '0x'

        Returns:
            boolean, True if read and write values match
        """
        self.setRegister(regname, teststring)
        # tell board to send data; wait to clear before interrogating register contents
        if regname == "SRAM_CTL":
            time.sleep(2)
            if self.commname == "rs422":
                logging.info(
                    self.loginfo + "skipping 'SRAM_CTL' register check for RS422"
                )
                return True
        else:
            time.sleep(0.1)
        temp = self.getRegister(regname)
        resp = temp[1].upper()
        if resp != teststring.upper():
            logging.error(
                self.logerr
                + "checkRegSet failure: "
                + regname
                + " ; set: "
                + teststring
                + " ; read: "
                + resp
            )
            return False
        return True

    def initPowerCheck(self):
        """
        Reset software and board timers for monitoring power status
        """
        self.inittime = time.time()
        logging.info(self.loginfo + "resetting timer for power check function")
        self.resetTimer()

    def powerCheck(self, delta=10):
        """
        Check to see if board power has persisted since powerCheck was last initialized.
          Compares time elapsed since initialization against board's timer. If the
          difference is greater than 'delta,' flag as False (power has likely failed)

        Args:
            delta: difference in seconds permitted between software and board timers

        Returns:
            boolean, 'True' means timer difference is less than 'delta' parameter;
                     'False' indicates power failure
        """
        elapsed = time.time() - self.inittime
        logging.debug(self.logdebug + "powerCheck: elapsed time = " + str(elapsed))
        difference = abs(elapsed - self.getTimer())
        if difference > delta:
            logging.warning(
                self.logwarn + "powerCheck function has failed; may indicate current "
                "or recent power failure "
            )
        return difference < delta

    def printBoardInfo(self):
        # TODO: add override option if logging level is above info
        logging.info(
            self.loginfo
            + "Python version: "
            + str(self.python)
            + "."
            + str(self.pyth1)
            + "."
            + str(self.pyth2)
        )
        logging.info(self.loginfo + "nsCamera software version: " + self.version)
        logging.info(self.loginfo + "FPGA firmware version: " + self.FPGAVersion)
        logging.info(self.loginfo + "FPGA implementation: " + self.FPGANum)
        if self.FPGAinvalid:
            logging.info(self.loginfo + "FPGA information unavailable")
        else:
            logging.info(self.loginfo + "Board type: " + self.FPGAboardtype)
            logging.info(self.loginfo + "Rad-Tolerant: " + str(self.FPGArad))
            logging.info(self.loginfo + "Sensor family: " + self.FPGAsensor)
            logging.info(self.loginfo + "Sensor label: " + self.sensor.loglabel)
            logging.info(
                self.loginfo + "Available interfaces: " + ", ".join(self.FPGAinterfaces)
            )
        if self.commname == "gige":
            ci = self.comms.CardInfoP.contents
            ip = ".".join(str(e) for e in [b for b in ci.IPAddr])
            logging.info(
                self.loginfo + "GigE connected to " + ip + ":" + str(self.port)
            )
        elif self.commname == "rs422":
            logging.info(self.loginfo + "RS422 connected to " + self.comms.port)

    def dumpRegisters(self):
        """
        *DEPRECATED* use dumpStatus() instead

        List contents of all registers in board.registers. *WARNING* some status flags
          will reset when read.

        Returns:
            Sorted list: [register name (register address) : register contents as
              hexadecimal string without '0x']
        """
        dump = {}
        for key in self.board.registers.keys():
            err, rval = self.getRegister(key)
            dump[key] = rval
        reglistmax = int(max(self.board.registers.values()), 16)
        dumplist = [0] * (reglistmax + 1)
        for k, v in dump.items():
            regnum = self.board.registers[k]
            dumplist[int(regnum, 16)] = (
                "(" + regnum + ") {0:<24} {1}".format(k, v.upper())
            )
        reglist = [a for a in dumplist if a]
        return reglist

    def dumpSubregisters(self):
        """
        *DEPRECATED* use dumpStatus() instead

        List contents of all subregisters in board.channel_lookups and
          board.monitor_lookups.
        *WARNING* some registers will reset when read; only the first subregister from
          such a register will return the correct value, the remainder will return zeros

        Returns:
            dictionary  {subregister name : subregister contents as binary string
              without initial '0b'}
        """
        dump = {}
        for sub in self.board.subreglist:
            key = sub.name
            err, resp = self.getSubregister(key)
            if err:
                logging.warning(
                    self.logwarn + "dumpSubregisters: unable to read subregister " + key
                )
            val = hex(int(resp, 2))
            dump[key] = val
        return dump

    def mmReadoff(self, waitOnSRAM, variation=None):
        """
        Convenience function for parsing frames for use by MicroManager plugin
        Args:
            waitOnSRAM: readoff wait flag
            variation: format of frames generated from readoff
                default - return first frame only
                "LastFrame" - return last frame only
                "Average" - provide average of frames as single frame
                "Landscape" - stitch frames together horizontally into single wide frame

        Returns:
            ndarray - single image frame
        """
        frames, datalen, data_err = self.readoff(waitOnSRAM)
        if variation == "LastFrame":
            return frames[self.sensor.nframes - 1]
        elif variation == "Average":
            return np.sum(frames, axis=0) // self.sensor.nframes
        elif variation == "Landscape":
            shaped = [
                np.reshape(frame, (self.sensor.maxheight, self.sensor.maxwidth))
                for frame in frames
            ]
            return np.concatenate(shaped, axis=1)
        else:
            return frames[0]

    def setFrames(self, minframe=None, maxframe=None):
        """
        Sets bounds on frames returned by board, inclusive (e.g., 0,3 returns four
        frames). If called without parameters, resets to full set of frames.

        Args:
            minframe: first frame to read from board
            maxframe: last frame to read from board

        Returns:
            Error string
        """
        logging.debug(
            self.logdebug
            + "setFrames: minframe = "
            + str(minframe)
            + "; maxframe = "
            + str(maxframe)
        )
        if minframe is None:
            minframe = self.sensor.minframe
        if maxframe is None:
            maxframe = self.sensor.maxframe
        if (
            not isinstance(minframe, int)
            or minframe < self.sensor.minframe
            or minframe > maxframe
            or not isinstance(maxframe, int)
            or maxframe > self.sensor.maxframe
        ):
            err = (
                self.logerr + "setFrames: invalid frame limits submitted. Frame "
                "selection remains unchanged. "
            )
            logging.error(err)
            return err

        initframe = hex(minframe)[2:].zfill(8)
        finframe = hex(maxframe)[2:].zfill(8)
        err1, _ = self.setRegister("FPA_FRAME_INITIAL", initframe)
        err2, _ = self.setRegister("FPA_FRAME_FINAL", finframe)
        self.sensor.firstframe = minframe
        self.sensor.lastframe = maxframe
        self.sensor.nframes = maxframe - minframe + 1
        self.comms.payloadsize = (
            self.sensor.width
            * self.sensor.height
            * self.sensor.nframes
            * self.sensor.bytesperpixel
        )
        plural = ""
        if self.sensor.nframes > 1:
            plural = "s"
        logging.info(
            self.loginfo
            + "Readoff set to "
            + str(self.sensor.nframes)
            + " frame"
            + plural
            + " ("
            + str(minframe)
            + ", "
            + str(maxframe)
            + ")"
        )
        err = err1 + err2
        if err:
            logging.error(
                self.logerr + "setFrames may not have functioned properly: " + err
            )
        return err

    def setRows(self, minrow=0, maxrow=None, padToFull=False):
        """
        Sets bounds on rows returned by board, inclusive (e.g., 0,1023 returns all 1024
          rows). If called without parameters, resets to full image size.

        Args:
            minrow: first row to return from board
            maxrow: last row to return from board
            padToFull: if True, generate full size frames, padding collected rows with
              zeroes if necessary
        """
        logging.debug(
            self.logdebug
            + "setRows: minrow = "
            + str(minrow)
            + "; maxrow = "
            + str(maxrow)
            + "; padToFull = "
            + str(padToFull)
        )
        if maxrow is None:
            maxrow = self.sensor.maxheight - 1
        if (
            not isinstance(minrow, int)
            or minrow < 0
            or minrow > maxrow
            or not isinstance(maxrow, int)
            or maxrow >= self.sensor.maxheight
        ):
            err = (
                self.logerr + "setRows: invalid row arguments submitted. Frame size"
                " remains unchanged. "
            )
            logging.error(err)
            return err

        initrow = hex(minrow)[2:].zfill(8)
        finrow = hex(maxrow)[2:].zfill(8)
        err1, _ = self.setRegister("FPA_ROW_INITIAL", initrow)
        err2, _ = self.setRegister("FPA_ROW_FINAL", finrow)
        self.sensor.firstrow = minrow
        self.sensor.lastrow = maxrow
        self.sensor.height = maxrow - minrow + 1
        self.comms.payloadsize = (
            self.sensor.width
            * self.sensor.height
            * self.sensor.nframes
            * self.sensor.bytesperpixel
        )

        if self.commname == "rs422":
            self.comms.datatimeout = (
                (1.0 * self.sensor.height / self.sensor.maxheight)
                * 5e7
                * self.sensor.nframes
                / self.comms.baud
            )
        self.padToFull = padToFull
        logging.info(
            self.loginfo
            + "Readoff set to "
            + str(self.sensor.height)
            + " rows ("
            + str(minrow)
            + ", "
            + str(maxrow)
            + ")"
        )
        err = err1 + err2
        if err:
            logging.error(
                self.logerr + "setRows may not have functioned properly: " + err
            )
        return err

    def abortReadoff(self, flag=True):
        """
        Simple abort command for readoff in waiting mode--does not interrupt download in
           progress. Requires external threading to function. *WARNING* if not
           intercepted by active readoff command, will terminate next readoff command
           immediately at inception.
        Args:
            flag: Sets passive abort flag read by readoff command
        Returns:
            boolean: updated setting of flag
        """
        logging.info(self.loginfo + "abortReadoff")
        self.abort = flag
        return flag

    def batchAcquire(
        self,
        sets=1,
        trig="Hardware",
        path=None,
        filename="Frame",
        prefix=None,
        showProgress=0,
    ):
        """
        Acquire a series of images as fast as possible, then process and save to disk.
        *WARNING* This method stores images in RAM, so the number of sets that can be
          acquired in a single call is limited by available memory.

        Args:
            sets: Number of acquisitions to perform
            trig: trigger type; 'hardware', 'software', or 'dual'
            path: save path, defaults to './output'
            filename: defaults to 'frames.bin'
            prefix: prepended to filename, defaults to time/date (e.g. '160830-124704_')
              DO NOT USE unless providing a varying value (a fixed prefix will cause
              overwriting)
            showProgress: if non-zero, show notice every 'showProgress' acquisitions and
              print total acquisition time

        Returns:
            Time taken for acquisition (seconds)
        """
        logging.debug(
            self.logdebug
            + "batchAcquire: sets = "
            + str(sets)
            + "; trig = "
            + str(trig)
            + "; path = "
            + str(path)
            + "; filename = "
            + str(filename)
            + "; prefix = "
            + str(prefix)
            + "; showProgress = "
            + str(showProgress)
        )
        datalist = ["0"] * sets
        timelist = [datetime.now()] * sets
        logging.info(
            self.loginfo
            + "batchAcquire: temporarily disabling warning and information logging "
        )
        logging.getLogger().setLevel(self.verbmap.get(2))
        beforeread = time.time()
        for i in range(sets):
            if showProgress and not (i + 1) % showProgress:
                print(self.loginfo + "batchAcquire: Acquiring set " + str(i + 1))
            self.arm(trig)
            data, datalen, data_err = self.readoff(fast=True)
            datalist[i] = data
            timelist[i] = datetime.now()
        afterread = time.time()
        if showProgress:
            print(
                self.loginfo
                + "batchAcquire: "
                + str(afterread - beforeread)
                + " seconds for "
                + str(sets)
                + " sets"
            )
        setnum = 0
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        for imset, imtime in zip(datalist, timelist):
            setnum = setnum + 1
            if showProgress and not setnum % showProgress:
                print(self.loginfo + "batchAcquire: Saving set " + str(setnum))
            parsed = generateFrames(self, imset)
            if prefix is None:
                setprefix = imtime.strftime("%y%m%d-%H%M%S%f")[:-2] + "_"
            else:
                setprefix = prefix
            self.saveTiffs(parsed, path, filename, prefix=setprefix)
        logging.getLogger().setLevel(self.verblevel)
        logging.info(self.loginfo + "batchAcquire: re-enabling logging")
        return afterread - beforeread

    # TODO: should this be just a flag for readoff instead of a distinct method?
    # TODO: make sure this handles single frames (list made already?), text frames
    # TODO: add documentation
    def saveHDF(
        self,
        frames,
        path=None,
        filename="Acquisition",
        prefix=None,
    ):
        """ """
        logging.info(self.loginfo + ": saveHDF")
        err = ""
        if path is None:
            path = os.path.join(os.getcwd(), "output")
        if prefix is None:
            prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
        if not os.path.exists(path):
            os.makedirs(path)

        h5file = os.path.join(path, prefix + filename + ".hdf5")
        with h5py.File(h5file, "w") as f:
            # shotgrp = f.create_group("DATA/SHOT")
            frame_index = 0
            for frame in frames:
                grp = f.create_group("DATA/SHOT/FRAME_0" + str(frame_index))
                data = grp.create_dataset(
                    "DATA", (self.sensor.height, self.sensor.width), data=frame
                )
                frame_index += 1


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.17 C:/Users/hill35/git/camera_python/nsCamera/comms/GigE.py File Reference

Classes

- class nsCamera.comms.GigE.GigE
- class nsCamera.comms.GigE.GigE.ZESTETM1_CARD_INFO

Namespaces

- namespace nsCamera
- namespace nsCamera.comms
- namespace nsCamera.comms.GigE

7.18 GigE.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Gigabit Ethernet interface for nsCamera.

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
```

```
"""

import ctypes as C
import logging
import os.path
import sys

from nsCamera.utils.misc import generateFrames, str2bytes, bytes2str


class GigE:
    """
    Code to manage Gigabit Ethernet connection to board. Each GigE object manages a
      single OT card; to use multiple cards, instantiate multiple cameraAssembler
      objects, each specifying the unique IPs of the corresponding OT card.

    Note: Orange Tree card must be configured before use. See the README for details

    Exposed methods:
        arm() - puts camera into wait state for external trigger
        readFrames() - waits for data ready register flag, then copies camera image data
          into numpy arrays
        readoff() - waits for data ready register flag, then copies camera image data
          into numpy arrays; returns payload, payload size, and error message
        sendCMD(pkt) - sends packet object via serial port
        readSerial(size, timeout) - read 'size' bytes from connection
        writeSerial(outstring) - submits string 'outstring' over connection
        closeDevice() - close connections and free resources
        getCardIP() - returns IP address of OT card
        getCardInfo() - prints report of details of OT card and connection
    """

    def __init__(self, camassem):
        """
        Args:
            camassem: parent cameraAssembler object
        """
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[GigE] "
        self.logerr = self.ca.logerrbase + "[GigE] "
        self.logwarn = self.ca.logwarnbase + "[GigE] "
        self.loginfo = self.ca.loginfobase + "[GigE] "
        self.logdebug = self.ca.logdebugbase + "[GigE] "
        logging.info(self.loginfo + "Initializing GigE comms object")
        self.mode = 1
        self.writeTimeout = 10000
        self.readTimeout = 10000
        self.payloadsize = (
            self.ca.sensor.width
            * self.ca.sensor.height
            * self.ca.sensor.nframes
            * self.ca.sensor.bytesperpixel
        )
        logging.debug(
            self.logdebug + "Payload size: " + str(self.payloadsize) + " bytes"
        )
        self.skipError = False

        self.ZErrorDict = {
            0x8000: "Socket Error",
            0x8001: "Internal Error",
            0x8002: "Illegal Status Code",
            0x8003: "Null Parameter",
            0x8004: "Out of Memory",
            0x8005: "Invalid Connection Type",
            0x8006: "Illegal Connection",
            0x8007: "Socket Closed Unexpectedly",
            0x8008: "Timeout",
            0x8009: "Illegal Parameter",
        }

        if self.ca.port:
            logging.debug(
                self.logdebug + "Port supplied to GigE.py: " + str(self.ca.port)
            )
            if isinstance(self.ca.port, int) and 0 < self.ca.port < 65536:
                self.dport = self.ca.port
            else:
                logging.error(
                    self.logerr + "Invalid port number supplied, defaulting to "
                    "20482 "
                )
                self.dport = 20482
        else:
            self.dport = 20482  # default

        self.ca.port = self.dport
        logging.debug(self.logdebug + "Port used by GigE.py: " + str(self.dport))

        logging.debug(self.logdebug + "CPU architecture: " + str(self.ca.arch))
        if self.ca.arch == "64bit":
            arch = "64"
        else:
            arch = "32"

        logging.debug(self.logdebug + "Operating system: " + str(self.ca.platform))
        if self.ca.platform == "Windows":
            lib_name = "ZestETM1.dll"
        elif self.ca.platform == "Linux" or self.ca.platform == "Darwin":
            lib_name = "libZestETM1.so"
        else:
            logging.warning(
                self.logwarn + "System does not self-identify as Linux, Windows, "
                "or Mac. Assuming posix-style libraries "
            )
            lib_name = "libZestETM1.so"

        self.closecard = False

        libpath = os.path.join(self.ca.packageroot, "comms", "ZestETM1", arch, lib_name)
        self._zest = C.CDLL(libpath)

        self.CardInfo = self.ZESTETM1_CARD_INFOZESTETM1_CARD_INFO()
        self.CardInfoP = C.pointer(self.CardInfo)

        # functions
        self.ZCountCards = self._zest.ZestETM1CountCards
        self.ZCountCards.argtypes = [
            C.POINTER(C.c_ulong),
            C.POINTER(C.POINTER(self.ZESTETM1_CARD_INFOZESTETM1_CARD_INFO)),
            C.c_int,
        ]

        self.ZOpenConnection = self._zest.ZestETM1OpenConnection
        self.ZOpenConnection.argtypes = [
            C.POINTER(self.ZESTETM1_CARD_INFOZESTETM1_CARD_INFO),
            C.c_int,
            C.c_ushort,
            C.c_ushort,
            C.POINTER(C.c_void_p),
        ]

        self.ZWriteData = self._zest.ZestETM1WriteData
        self.ZWriteData.argtypes = [
            C.c_void_p,
            C.c_void_p,
            C.c_ulong,
            C.POINTER(C.c_ulong),
            C.c_ulong,
        ]

        self.ZReadData = self._zest.ZestETM1ReadData
        self.ZReadData.argtypes = [
            C.c_void_p,
            C.c_void_p,
            C.c_ulong,
            C.POINTER(C.c_ulong),
            C.c_ulong,
        ]

        self.Connection = C.c_void_p()
        self.openDevice(self.ca.timeout)

    def sendCMD(self, pkt):
        """
        Submit packet and verify the response packet.
        Packet communications with FPGA omit CRC suffix, so adds fake CRC bytes to
          response

        Args:
            pkt: Packet object

        Returns:
            tuple (error, response string)
        """

        pktStr = pkt.pktStr()[0:16]
        logging.debug(self.logdebug + "sendCMD packet: " + str(pktStr))
        err = ""
        self.ca.writeSerial(pktStr)
        if (
            hasattr(self.ca, "board")
            and pktStr[4] == "0"
            and pktStr[5:8] == self.ca.board.registers["SRAM_CTL"]
        ):
            bufsize = self.payloadsize + 16
            resptext = self.readSerial(bufsize)

            if len(resptext) < 32:
                logging.debug(self.logdebug + "sendCMD resptext = " + str(resptext))
            else:
                logging.debug(
                    self.logdebug
                    + "sendCMD resptext (truncated) = "
                    + str(resptext)[0:32]
                )

            if len(resptext) < bufsize + 16:
                err += (
                    self.logerr + "sendCMD- packet too small, payload may be incomplete"
                )
                logging.error(err)
        else:
            # workaround for initial setup before board object has been initialized
            resp = self.readSerial(8)
            logging.debug(self.logdebug + "sendCMD response: " + str(resp))
            if len(resp) < 8:
                err += self.logerr + "sendCMD- response too small, returning zeros"
                resptext = "00000000000000000000"
                logging.error(err)
            else:
                resptext = resp + "0000"
        return err, resptext

    def arm(self, mode):
        """
        Puts camera into wait state for trigger. Mode determines source; defaults to
          'Hardware'

        Args:
            mode:   'Software'|'S' activates software, disables hardware triggering
                    'Hardware'|'H' activates hardware, disables software triggering
                      Hardware is the default

        Returns:
            tuple (error, response string)
        """
        if not mode:
            mode = "Hardware"
            logging.info(self.loginfo + "arm")
        logging.debug(self.logdebug + "arming mode: " + str(mode))
        self.ca.clearStatus()
        self.ca.latchPots()
        err, resp = self.ca.startCapture(mode)
        if err:
            logging.error(self.logerr + "unable to arm camera")
        else:
            self.ca.armed = True
            self.skipError = True
        return err, resp

    def readFrames(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays.

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
```

```
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            list of numpy arrays OR raw text stream

        """
        frames, _, _ = self.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

    def readoff(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays; returns data, length of data,
        and error messages. Use 'readFrames()' unless you require this additional
        information

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            tuple (list of numpy arrays OR raw text stream, length of downloaded payload
              in bytes, payload error flag) since CRC check is handled by TCP/IP,
              payload error flag is always False for GigE
        """
        logging.info(self.loginfo + "readoff")
        logging.debug(
            self.logdebug
            + "readoff: waitonSRAM = "
            + str(waitOnSRAM)
            + "; timeout = "
            + str(timeout)
            + "; fast = "
            + str(fast)
        )
        # Wait for data to be ready on board
        # Skip wait only if explicitly tagged 'False' ('None' defaults to True)
        if waitOnSRAM is not False:
            self.ca.waitForSRAM(timeout)
        self.skipError = False
        err, rval = self.ca.readSRAM()
        if err:
            logging.error(self.logerr + "Error detected in readSRAM")
        elif self.ca.boardname == "llnl_v4":
            # self.ca.setSubregister('SWACK','1')
            pass
        # extract the data. Remove header; the FPGA returns a packet without the CRC
        #   suffix
        logging.debug(self.logdebug + "readoff: first 64 chars: " + str(rval[0:64]))
        data = rval[32:]
        if fast:
            return data, len(data) // 2, bool(err)
        else:
            parsed = generateFrames(self.ca, data, columns)
            return parsed, len(data) // 2, bool(err)

    def writeSerial(self, outstring, timeout):
        """
        Transmit string to board

        Args:
            outstring: string to write
            timeout: serial timeout in sec (defaults to self.writeTimeout)

        Returns:
            integer number of bytes written
        """
        logging.debug(
            self.logdebug
            + "writeSerial: outstring = "
            + str(outstring)
            + "; timeout = "
            + str(timeout)
        )
        if not timeout:
            timeout = self.writeTimeout
        outstring = str2bytes(outstring)
        outbuff = C.create_string_buffer(outstring)
        outbuffp = C.pointer(outbuff)
        outbufflen = len(outstring)
        writelen = C.c_ulong(0)
        err = self.ZWriteData(
            self.Connection, outbuffp, outbufflen, C.byref(writelen), timeout
        )
        if err:
            if err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.error(
                    self.logerr + "writeSerial error: " + self.ZErrorDict[err]
                )
        logging.debug(self.logdebug + "writeSerial: writelen = " + str(writelen))
        return writelen

    def readSerial(self, size, timeout=None):
        """
        Read bytes from the serial port. Does not verify packets.

        Args:
           size: number of bytes to read
           timeout: serial timeout in sec (defaults to self.readTimeout)

        Returns:
           tuple (error string, string read from serial port)
        """
        logging.debug(
            self.logdebug
            + "readSerial: size = "
            + str(size)
            + "; timeout = "
            + str(timeout)
        )
        if not timeout:
            timeout = self.readTimeout
        inbuff = C.create_string_buffer(size + 1)
        inbuffp = C.pointer(inbuff)
        readlen = C.c_ulong(0)
        err = self.ZReadData(self.Connection, inbuffp, size, C.byref(readlen), timeout)
        if err:
            if self.skipError:
                logging.debug(
                    self.logdebug + "readSerial: skipped error: " + self.ZErrorDict[err]
                )
                self.skipError = False
            elif err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.error(self.logerr + "readSerial error: " + self.ZErrorDict[err])
        return bytes2str(inbuff.raw)[:-2]

    # TODO: check for valid timeout, probably in init in CameraAssembler
    def openDevice(self, timeout=30):
        """
        Find Orange Tree card and open a connection; if IP is supplied as parameter for
          the CameraAssembler, bypass network search and connect directly to indicated
          IP address

        Args:
            timeout: timeout in seconds for attempting to connect to a card
        """
        err = self._zest.ZestETM1Init()
        if err:
            logging.critical(self.logcrit + "ZestETM1Init failure")
            sys.exit(1)
        logging.info(self.loginfo + "searching for Orange Tree cards")
        NumCards = C.c_ulong(0)

        if self.ca.iplist:
            ubyte4 = C.c_ubyte * 4
            self.CardInfo.IPAddr = ubyte4(*self.ca.iplist)
            self.CardInfo.ControlPort = C.c_ushort(self.dport)
            self.CardInfo.Timeout = C.c_ulong(self.writeTimeout)
            self.closecard = False
        else:
            wait = 0
            while True:
                logging.debug(
                    self.logdebug + "openDevice: connection wait = " + str(wait)
                )
                err = self.ZCountCards(C.byref(NumCards), C.byref(self.CardInfoP), 1000)
                if err:
                    logging.critical(self.logcrit + "CountCards failure")
                    sys.exit(1)
                if NumCards.value > 0:
                    break
                if wait == timeout:
                    logging.critical(
                        "{}No Orange Tree cards found in {} seconds".format(
                            self.logcrit, timeout
                        )
                    )
                    sys.exit(1)
                elif not wait % 5:
                    logging.info(
                        "{}Still trying to connect after {} seconds...".format(
                            self.loginfo, wait
                        )
                    )
                wait += 1
            logging.info(
                self.loginfo + str(NumCards.value) + " Orange Tree card(s) found"
            )  # TODO: add check for GigE bit in board description
        err = self.ZOpenConnection(
            self.CardInfoP, 0, self.dport, 0, C.byref(self.Connection)
        )
        if err:
            if err == 0x4000:
                logging.warning(
                    self.logerr + "OT Card emitted an undefined warning message"
                )
            else:
                logging.critical(
                    self.logcrit + "OpenConnection failure: " + self.ZErrorDict[err]
                )
            sys.exit(1)

    def closeDevice(self):
        """
        Close connection to Orange Tree card and free resources
        """
        logging.debug(self.logdebug + "Closing connection to Orange Tree card")
        self._zest.ZestETM1CloseConnection(self.Connection)
        if self.closecard:
            try:
                self._zest.ZestETM1FreeCards(self.CardInfoP)
            except SystemExit:
                raise
            except KeyboardInterrupt:
                raise
            except Exception:
                logging.error(self.logerr + "Error reported in OT card closure")
        self._zest.ZestETM1Close()

    def getCardIP(self):
        """
        Query IP address of OT card

        Returns: address of OT card as list of bytes
        """
        return self.CardInfo.IPAddr

    # TODO: use logging.info, with override option?
    def getCardInfo(self):
        """
        Prints status message with information returned by OT card
        """
        ci = self.CardInfoP.contents
        print("GigE Card Status:")
        print("-------------")
        print("IP: " + ".".join(str(e) for e in [b for b in ci.IPAddr]))
        print("ControlPort: " + str(ci.ControlPort))
        print("Timeout: " + str(ci.Timeout))
        print("HTTPPort: " + str(ci.HTTPPort))
        print("MACAddr: " + ".".join(format(e, "02X") for e in [b for b in ci.MACAddr]))
        print("SubNet: " + ".".join(str(e) for e in [b for b in ci.SubNet]))
        print("Gateway: " + ".".join(str(e) for e in [b for b in ci.Gateway]))
        print("SerialNumber: " + str(ci.SerialNumber))
        print("FirmwareVersion: " + str(ci.FirmwareVersion))
        print("HardwareVersion: " + str(ci.HardwareVersion))
        print("-------------")

    class ZESTETM1_CARD_INFO(C.Structure):
        ubyte4 = C.c_ubyte * 4
        ubyte6 = C.c_ubyte * 6
        _fields_ = [
            ("IPAddr", ubyte4),
            ("ControlPort", C.c_ushort),
            ("Timeout", C.c_ulong),
            ("HTTPPort", C.c_ushort),
            ("MACAddr", ubyte6),
            ("SubNet", ubyte4),
            ("Gateway", ubyte4),
            ("SerialNumber", C.c_ulong),
            ("FirmwareVersion", C.c_ulong),
            ("HardwareVersion", C.c_ulong),
        ]


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.19 C:/Users/hill35/git/camera_python/nsCamera/comms/RS422.py File Reference

Classes

- class nsCamera.comms.RS422.RS422

Namespaces

- namespace nsCamera
- namespace nsCamera.comms
- namespace nsCamera.comms.RS422

7.20 RS422.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
RS422 driver for nsCamera

Author: Brad Funsten (funsten1@llnl.gov)
Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080
```

```

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import logging
import sys
import time  # to time the script

import serial
import serial.tools.list_ports  # for RS422 serial link setup

from nsCamera.utils.misc import generateFrames, str2bytes, bytes2str, checkCRC


class RS422:
    """
    Code to manage RS422 connection. Will automatically query available COM interfaces
      until a board is found. Use the 'port=x' parameter in cameraAssembler call to
      specify a particular COM interface.

    Exposed methods:
        arm() - Puts camera into wait state for external trigger
        readFrames() - waits for data ready register flag, then copies camera image data
          into numpy arrays
        readoff() - waits for data ready register flag, then copies camera image data
          into numpy arrays; returns payload, payload size, and error message
        sendCMD(pkt) - sends packet object via serial port
        readSerial(size, timeout) - read 'size' bytes from serial port
        writeSerial(cmd) - submits string 'cmd' (assumes string is preformed packet)
        closeDevice() - close serial connections
    """

    def __init__(self, camassem, baud=921600, par="O", stop=1):
        """
        Args:
            camassem: parent cameraAssembler object
            baud: bits per second
            par: parity type
            stop: number of stop bits
        """
        self.ca = camassem
        self.logcrit = self.ca.logcritbase + "[RS422] "
        self.logerr = self.ca.logerrbase + "[RS422] "
        self.logwarn = self.ca.logwarnbase + "[RS422] "
        self.loginfo = self.ca.loginfobase + "[RS422] "
        self.logdebug = self.ca.logdebugbase + "[RS422] "
        logging.info(self.loginfo + "initializing RS422 comms object")
        logging.debug(
            self.logdebug
            + "Init: baud = "
            + str(baud)
            + "; par = "
            + str(par)
            + "; stop = "
            + str(stop)
        )
        self.mode = 0
        self.baud = baud  # Baud rate (bits/second)
        self.par = par  # Parity bit
        self.stop = stop  # Number of stop bits
        self.read_timeout = 1  # default timeout for ordinary packets
        self.write_timeout = 1
        # TODO: make datatimeout a cameraAssembler parameter
        self.datatimeout = 60  # timeout for data read
        logging.debug(
            self.logdebug + "Data timeout = " + str(self.datatimeout) + " seconds"
        )
        self.PY3 = sys.version_info > (3,)
        self.skipError = False
        port = ""
        ports = list(serial.tools.list_ports.comports())
        logging.debug(self.logdebug + "Comports: " + str(ports))
        for p, desc, add in ports:
            if self.ca.port is None or p == "COM" + str(self.ca.port):
                logging.info(self.loginfo + "found comm port " + p)
                try:
                    with serial.Serial(
                        p,
                        self.baud,
                        parity=self.par,
                        timeout=0.01,
                        write_timeout=0.01,
                    ) as ser:
                        ser.write(str2bytes("aaaa1000000000001a84"))
                        time.sleep(1)
                        s = ser.read(10)
                        resp = bytes2str(s)
                        logging.debug(self.logdebug + "Init response: " + str(resp))
                        if (
                            resp[0:5].lower() == "aaaa9"
                        ):  # TODO: add check for RS422 bit in board description
                            boardid = resp[8:10]
                            if boardid == "00":
                                logging.critical(
                                    self.logcrit + "SNLrevC board detected - not "
                                    "compatible with nsCamera >= 2.0"
                                )
                                sys.exit(1)
                            elif boardid == "81":
                                logging.info(self.loginfo + "LLNLv1 board detected")
                            elif boardid == "84":
                                logging.info(self.loginfo + "LLNLv4 board detected")
                            else:
                                logging.info(
                                    self.loginfo + "unidentified board detected"
                                )
                            logging.info(self.loginfo + "connected to " + p)
                            port = p
                            ser.reset_input_buffer()
                            ser.reset_output_buffer()
                            break
                except Exception as e:
                    logging.error(self.logerr + "port identification: " + str(e))
        if port == "":
            if self.ca.port:
                logging.critical(
                    self.logcrit + "No usable board found at port " + str(self.ca.port)
                )
                sys.exit(1)
            else:
                logging.critical(self.logcrit + "No usable board found")
                sys.exit(1)
        self.port = port  # COM port to use for RS422 link
        self.ca.port = port[3:]  # re-extract port number from com name

        self._ser = serial.Serial(  # Class RS422
            port=self.port,
            baudrate=self.baud,
            parity=self.par,
            stopbits=self.stop,
            timeout=self.read_timeout,  # timeout for serial read
            bytesize=serial.EIGHTBITS,
        )
        self.payloadsize = (
            self.ca.sensor.width
            * self.ca.sensor.height
            * self.ca.sensor.nframes
            * self.ca.sensor.bytesperpixel
        )
        logging.debug(
            self.logdebug + "Payload size: " + str(self.payloadsize) + " bytes"
        )
        self._ser.flushInput()
        if not self._ser.is_open:
            logging.critical(self.logcrit + "Unable to open serial connection")
            sys.exit(1)

    def serialClose(self):
        """
        Close serial interface
        """
        logging.debug(self.logdebug + "serialclose")
        self._ser.close()  # close serial interface COM port

    def sendCMD(self, pkt):
        """
        Submit packet and verify response packet. Recognizes readoff packet and adjusts
        read size and timeout appropriately

        Args:
            pkt: Packet object

        Returns:
            tuple (error, response string)
        """
        pktStr = pkt.pktStr()
        logging.debug(self.logdebug + "sendCMD packet: " + str(pktStr))
        self._ser.flushInput()
        time.sleep(0.01)  # wait 10 ms in between flushing input and output buffers
        self._ser.flushOutput()
        self.ca.writeSerial(pktStr)
        err0 = ""
        err = ""
        resp = ""
        tries = 3  # TODO: make a function parameter?

        if (
            hasattr(self.ca, "board")
            and pktStr[4] == "0"
            and pktStr[5:8] == self.ca.board.registers["SRAM_CTL"]
        ):
            # download data payload
            logging.info(
                self.loginfo + "Payload size (bytes) = " + str(self.payloadsize)
            )
            crcresp0 = ""
            crcresp1 = ""
            smallresp = ""
            emptyResponse = False
            wrongSize = False
            # TODO: refactor payload error management to another method
            for i in range(tries):
                err, resp = self.readSerial(
                    self.payloadsize + 20, timeout=self.datatimeout
                )
                if err:
                    logging.error(
                        self.logerr + "sendCMD: read payload failed " + pktStr + err
                    )
                    self.ca.payloaderror = True
                else:
                    if not len(resp):
                        err0 = self.logerr + "sendCMD: empty response from board"
                        logging.error(err0)
                        emptyResponse = True
                        self.ca.payloaderror = True
                    elif len(resp) != 2 * (self.payloadsize + 20):
                        err0 = (
                            self.logerr
                            + "sendCMD: incorrect response; expected "
                            + str(self.payloadsize + 20)
                            + " bytes, received "
                            + str(len(resp) // 2)
                        )
                        logging.error(err0)
                        wrongSize = True
                        smallresp = resp
                        self.ca.payloaderror = True
                    elif not checkCRC(resp[4:20]):
                        err0 = (
                            self.logerr
                            + "sendCMD: "
                            + pktStr
                            + " - payload preface CRC fail"
                        )
                        logging.error(err0)
                        self.ca.payloaderror = True
                        crcresp1 = resp
                    elif not checkCRC(resp[24:]):
                        err0 = (
                            self.logerr + "sendCMD: " + pktStr + " - payload CRC fail"
                        )
                        logging.error(err0)
                        self.ca.payloaderror = True
                        crcresp0 = resp
                    err += err0
                time.sleep(5)
                if self.ca.payloaderror:
                    # keep best results over multiple tries; e.g., if first try is
                    #   bad CRC and second try is an incomplete payload, use the
                    #   first payload
                    if i == tries - 1:
                        if crcresp0:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire "
                                "CRC-confirmed payload after "
                                + str(tries)
                                + " attempts. Continuing with unconfirmed payload"
                            )
                            resp = crcresp0
                        elif crcresp1:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire "
                                "CRC-confirmed readoff after "
                                + str(tries)
                                + " attempts. Continuing with unconfirmed payload"
                            )
                            resp = crcresp1
                        elif wrongSize:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire complete "
                                "payload after "
                                + str(tries)
                                + " attempts. Dumping datastream to file."
                            )
                            resp = smallresp
                            self.ca.dumpNumpy(resp)
                        elif emptyResponse:
                            logging.error(
                                self.logerr + "sendCMD: Unable to acquire any "
                                "payload after " + str(tries) + " attempts."
                            )
                    else:
                        logging.info(
                            self.loginfo + "Retrying download, attempt #" + str(i + 1)
                        )
                        err = ""
                        err0 = ""
                        self.ca.payloaderror = False
                        self.ca.writeSerial(pktStr)
                else:
                    logging.info(self.loginfo + "Download successful")
                    if self.ca.boardname == "llnl_v4":
                        # self.ca.setSubregister('SWACK','1')
                        pass
                    break

        else:
            # non-payload messages and workaround for initial setup before board object
            #   has been initialized
            time.sleep(0.03)
            self._ser.timeout = 0.02
            err, resp = self.readSerial(10)
            logging.debug(self.logdebug + "sendCMD response: " + str(resp))
            if err:
                logging.error(
                    self.logerr + "sendCMD: readSerial failed (regular packet) " + err
                )
            elif not checkCRC(resp[4:20]):
                err = self.logerr + "sendCMD- regular packet CRC fail: " + resp
                logging.error(err)
        return err, resp

    def arm(self, mode):
        """
        Puts camera into wait state for trigger. Mode determines source; defaults to
          'Hardware'

        Args:
            mode:   'Software'|'S' activates software, disables hardware triggering
                    'Hardware'|'H' activates hardware, disables software triggering
                      Hardware is the default

        Returns:
            tuple (error, response string)
        """
        if not mode:
            mode = "Hardware"
        logging.info(self.loginfo + "arm")
        logging.debug(self.logdebug + "arming mode: " + str(mode))
        self.ca.clearStatus()
        self.ca.latchPots()
        err, resp = self.ca.startCapture(mode)
        if err:
            logging.error(self.logerr + "unable to arm camera")
        else:
            self.ca.armed = True
            self.skipError = True
        return err, resp

    def readFrames(self, waitOnSRAM, timeout=0, fast=False, columns=1):
        """
        Copies image data from board into numpy arrays.

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            list of numpy arrays OR raw text stream

        """
        frames, _, _ = self.readoff(waitOnSRAM, timeout, fast, columns)
        return frames

    def readoff(self, waitOnSRAM, timeout, fast, columns=1):
        """
        Copies image data from board into numpy arrays; returns data, length of data,
        and error messages. Use 'readFrames()' unless you require this additional
        information

        Args:
            waitOnSRAM: if True, wait until SRAM_READY flag is asserted to begin copying
              data
            timeout: passed to waitForSRAM; after this many seconds begin copying data
              irrespective of SRAM_READY status; 'zero' means wait indefinitely
              WARNING: If acquisition fails, the SRAM will not contain a current image,
                but the code will copy the data anyway
            fast: if False, parse and convert frames to numpy arrays; if True, return
              unprocessed text stream
            columns: 1 for single image per frame, 2 for separate hemisphere images

        Returns:
            tuple (list of numpy arrays OR raw text stream, length of downloaded payload
              in bytes, payload error flag)
            NOTE: This reduces readoff by <1 second, so will have no noticeable impact
              when using RS422
        """
        logging.info(self.loginfo + "readoff")
        logging.debug(
            self.logdebug
            + "readoff: waitonSRAM = "
            + str(waitOnSRAM)
            + "; timeout = "
            + str(timeout)
            + "; fast = "
            + str(fast)
        )
        errortemp = False

        # Wait for data to be ready on board, turns off error messaging
        # Skip wait only if explicitly tagged 'False' ('None' defaults to True)
        if waitOnSRAM is not False:
            logging.getLogger().setLevel(logging.CRITICAL)
            self.ca.waitForSRAM(timeout)
            logging.getLogger().setLevel(self.ca.verblevel)

        # Retrieve data
        err, rval = self.ca.readSRAM()
        if err:
            logging.error(self.logerr + "Error detected in readSRAM")
        time.sleep(0.3)
        logging.debug(self.logdebug + "readoff: first 64 chars: " + str(rval[0:64]))
        # extract only the read burst data. Remove header & CRC footer
        read_burst_data = rval[36:-4]

        # Payload size as string implied by provided parameters
        expectedlength = (
            4
            * (self.ca.sensor.lastframe - self.ca.sensor.firstframe + 1)
            * (self.ca.sensor.lastrow - self.ca.sensor.firstrow + 1)
            * self.ca.sensor.width
        )
        padding = expectedlength - len(read_burst_data)
        if padding:
            logging.warning(
                "{logwarn}readoff: Payload is shorter than expected."
                " Padding with '0's".format(logwarn=self.logwarn)
            )
            read_burst_data = read_burst_data.ljust(expectedlength, "0")

        if fast:
            return read_burst_data, len(read_burst_data) // 2, errortemp
        else:
            parsed = generateFrames(self.ca, read_burst_data, columns)
            return parsed, len(read_burst_data) // 2, errortemp

    def writeSerial(self, outstring, timeout):
        """
        Transmit string to board

        Args:
            outstring: string to write
            timeout: serial timeout in sec
        Returns:
            integer length of string written to serial port
        """
        logging.debug(
            self.logdebug
            + "writeSerial: outstring = "
            + str(outstring)
            + "; timeout = "
            + str(timeout)
        )
        if timeout:
            self._ser.timeout = timeout
        else:
            self._ser.timeout = self.write_timeout
        lengthwritten = self._ser.write(str2bytes(outstring))
        self._ser.timeout = self.read_timeout  # reset if changed above
        return lengthwritten

    def readSerial(self, size, timeout=None):
        """
        Read bytes from the serial port. Does not verify packets.

        Args:
           size: number of bytes to read
           timeout: serial timeout in sec

        Returns:
           tuple (error string, string read from serial port)
        """
        logging.debug(
            self.logdebug
            + "readSerial: size = "
            + str(size)
            + "; timeout = "
            + str(timeout)
        )
        err = ""
        if timeout:
            self._ser.timeout = timeout
        else:
            self._ser.timeout = self.read_timeout
        resp = self._ser.read(size)
        if len(resp) < 10:  # bytes
            err += (
                self.logerr + "readSerial : packet too small: '" + bytes2str(resp) + "'"
            )
            logging.error(err)
        return err, bytes2str(resp)

    def closeDevice(self):
        """
        Close primary serial interface
        """
        logging.debug(self.logdebug + "Closing RS422 connection")
        self._ser.close()


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.21 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/Data.c File Reference

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include "ZestETM1.h"
#include "Private.h"
```

Classes

- struct ZESTETM1_WRITE_REG_CMD
- struct ZESTETM1_WRITE_REG_RESPONSE
- struct ZESTETM1_READ_REG_CMD
- struct ZESTETM1_READ_REG_RESPONSE
- struct ZESTETM1_MAILBOX_INT_CMD
- struct ZESTETM1_MAILBOX_INT_RESPONSE

Macros

- #define _CRT_SECURE_NO_WARNINGS
- #define ZESTETM1_COMMAND_SPI 0xee
- #define ZESTETM1_COMMAND_WRITE_REG 0xf6
- #define ZESTETM1_COMMAND_READ_REG 0xf7
- #define ZESTETM1_COMMAND_MAILBOX_INT 0xf8

Functions

- ZESTETM1_STATUS ZestETM1_OpenConnection (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION_TYPE Type, uint16_t Port, uint16_t LocalPort, ZESTETM1_CONNECTION ∗Connection)
- ZESTETM1_STATUS ZestETM1_CloseConnection (ZESTETM1_CONNECTION Connection)
- static ZESTETM1_STATUS ZestETM1_WriteData (ZESTETM1_CONNECTION Connection, void ∗Buffer, uint32_t Length, unsigned long ∗Written, uint32_t Timeout)
- static ZESTETM1_STATUS ZestETM1_ReadData (ZESTETM1_CONNECTION Connection, void ∗Buffer, uint32_t Length, unsigned long ∗Read, uint32_t Timeout)
- ZESTETM1_STATUS ZestETM1_SendCommand (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION Connection, void ∗WriteBuffer, uint32_t WriteLen, void ∗ReadBuffer, uint32_t ReadLen, int WaitForAck)
- ZESTETM1_STATUS ZestETM1_SPIReadWrite (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION Connection, int Device, int WordLen, uint32_t ∗WriteData, uint32_t ∗ReadData, uint32_t Length, int ReleaseCS, int WaitForAck)
- ZESTETM1_STATUS ZestETM1OpenConnection (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION_TYPE Type, uint16_t Port, uint16_t LocalPort, ZESTETM1_CONNECTION ∗Connection)
- ZESTETM1_STATUS ZestETM1CloseConnection (ZESTETM1_CONNECTION Connection)
- ZESTETM1_STATUS ZestETM1WriteData (ZESTETM1_CONNECTION Connection, void ∗Buffer, unsigned long Length, unsigned long ∗Written, unsigned long Timeout)
- ZESTETM1_STATUS ZestETM1ReadData (ZESTETM1_CONNECTION Connection, void ∗Buffer, unsigned long Length, unsigned long ∗Read, unsigned long Timeout)
- ZESTETM1_STATUS ZestETM1SPIReadWrite (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_SPI_RATE Rate, int WordLen, void ∗WriteData, void ∗ReadData, unsigned long Length, int ReleaseCS)
- ZESTETM1_STATUS ZestETM1WriteRegister (ZESTETM1_CARD_INFO ∗CardInfo, unsigned long Addr, unsigned short Data)
- ZESTETM1_STATUS ZestETM1ReadRegister (ZESTETM1_CARD_INFO ∗CardInfo, unsigned long Addr, unsigned short ∗Data)
- ZESTETM1_STATUS ZestETM1SetInterrupt (ZESTETM1_CARD_INFO ∗CardInfo)

7.21.1 Class Documentation

7.21.1.1 struct ZESTETM1_WRITE_REG_CMD

Definition at line 58 of file Data.c.

Class Members

uint8_t	Addr	
uint8_t	Command	
uint16_t	Data	

7.21.1.2 struct ZESTETM1_WRITE_REG_RESPONSE

Definition at line 64 of file Data.c.

Class Members

uint8_t	Command	

Class Members

uint8_t	Dummy1[2]	
uint8_t	Status	

7.21.1.3 struct ZESTETM1_READ_REG_CMD

Definition at line 70 of file Data.c.

Class Members

uint8_t	Addr	
uint8_t	Command	
uint8_t	Dummy[2]	

7.21.1.4 struct ZESTETM1_READ_REG_RESPONSE

Definition at line 76 of file Data.c.

Class Members

uint8_t	Command	
uint8_t	Status	
uint16_t	Value	

7.21.1.5 struct ZESTETM1_MAILBOX_INT_CMD

Definition at line 82 of file Data.c.

Class Members

uint8_t	Command	
uint8_t	Dummy[3]	

7.21.1.6 struct ZESTETM1_MAILBOX_INT_RESPONSE

Definition at line 87 of file Data.c.

Class Members

uint8_t	Command	

Class Members

uint16_t	Dummy	
uint8_t	Status	

7.21.2 Macro Definition Documentation

7.21.2.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 45 of file Data.c.

7.21.2.2 ZESTETM1_COMMAND_MAILBOX_INT

```
#define ZESTETM1_COMMAND_MAILBOX_INT 0xf8
```

Definition at line 96 of file Data.c.

7.21.2.3 ZESTETM1_COMMAND_READ_REG

```
#define ZESTETM1_COMMAND_READ_REG 0xf7
```

Definition at line 95 of file Data.c.

7.21.2.4 ZESTETM1_COMMAND_SPI

```
#define ZESTETM1_COMMAND_SPI 0xee
```

Definition at line 93 of file Data.c.

7.21.2.5 ZESTETM1_COMMAND_WRITE_REG

```
#define ZESTETM1_COMMAND_WRITE_REG 0xf6
```

Definition at line 94 of file Data.c.

7.21.3 Function Documentation

7.21.3.1 ZestETM1_CloseConnection()

```
ZESTETM1_STATUS ZestETM1_CloseConnection (
              ZESTETM1_CONNECTION Connection )
```

Definition at line 225 of file Data.c.

```
00226 {
00227     ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
00228 
00229     if (Conn==NULL)
00230     {
00231         return ZESTETM1_NULL_PARAMETER;
00232     }
00233     if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
00234     {
00235         return ZESTETM1_ILLEGAL_CONNECTION;
00236     }
00237 
00238     // Cleanup
00239     closesocket(Conn->Socket);
00240     Conn->Magic = 0;
00241     free(Conn);
00242 
00243     return ZESTETM1_SUCCESS;
00244 }
```

7.21.3.2 ZestETM1_OpenConnection()

```
ZESTETM1_STATUS ZestETM1_OpenConnection (
              ZESTETM1_CARD_INFO * CardInfo,
              ZESTETM1_CONNECTION_TYPE Type,
              uint16_t Port,
              uint16_t LocalPort,
              ZESTETM1_CONNECTION * Connection )
```

Definition at line 101 of file Data.c.

```
00106 {
00107     ZESTETM1_CONNECTION_STRUCT *NewStruct;
00108     SOCKET Socket = -1;
00109     char AddrBuffer[32];
00110     char PortBuffer[32];
00111 
00112     if (Connection==NULL || CardInfo==NULL)
00113     {
00114         return ZESTETM1_NULL_PARAMETER;
00115     }
00116 
00117     // Allocate data structure
00118     NewStruct = malloc(sizeof(ZESTETM1_CONNECTION_STRUCT));
00119     if (NewStruct==NULL)
00120     {
00121         return ZESTETM1_OUT_OF_MEMORY;
00122     }
00123 
00124     // Build target addresses
00125     sprintf(AddrBuffer, "%d.%d.%d.%d", CardInfo->IPAddr[0], CardInfo->IPAddr[1],
00126                     CardInfo->IPAddr[2], CardInfo->IPAddr[3]);
00127     sprintf(PortBuffer, "%d", Port);
00128 
00129     if (Type==ZESTETM1_TYPE_UDP)
00130     {
00131         // Open UDP connection
00132         struct sockaddr_in SourceIP;
00133         int SourceLen = (int)sizeof(struct sockaddr_in);
00134         Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
00135         if (Socket<0)
```

```
            return ZESTETM1_SOCKET_ERROR;
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = htons(LocalPort);
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);

        NewStruct->Target.sin_family = AF_INET;
        NewStruct->Target.sin_addr.s_addr = inet_addr(AddrBuffer);
        NewStruct->Target.sin_port = htons(atoi(PortBuffer));
    }
    else if (Type==ZESTETM1_TYPE_TCP)
    {
        // Open TCP connection
        struct addrinfo *AddrResult = NULL,
                        *Ptr = NULL,
                        Hints;
        int Result;
        struct sockaddr_in SourceIP;
        int SourceLen = (int)sizeof(struct sockaddr_in);

        memset(&Hints, 0, sizeof(Hints));
        Hints.ai_family = AF_UNSPEC;
        Hints.ai_socktype = SOCK_STREAM;
        Hints.ai_protocol = IPPROTO_TCP;

        // Resolve the server address and port
        Result = getaddrinfo(AddrBuffer, PortBuffer, &Hints, &AddrResult);
        if (Result!=0)
        {
            free(NewStruct);
            return ZESTETM1_SOCKET_ERROR;
        }

        // Attempt to connect to an address until one succeeds
        for (Ptr=AddrResult; Ptr!=NULL; Ptr=Ptr->ai_next)
        {
            // Create a SOCKET for connecting to server
            Socket = socket(Ptr->ai_family, Ptr->ai_socktype,
                            Ptr->ai_protocol);
            if (Socket<0)
            {
                freeaddrinfo(AddrResult);
                free(NewStruct);
                return ZESTETM1_SOCKET_ERROR;
            }

            // Connect to ZestETM1
            Result = connect(Socket, Ptr->ai_addr, (int)Ptr->ai_addrlen);
            if (Result<0)
            {
                closesocket(Socket);
                Socket = -1;
                continue;
            }
            break;
        }

        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);
        freeaddrinfo(AddrResult);
    }
    else
    {
        free(NewStruct);
        return ZESTETM1_INVALID_CONNECTION_TYPE;
    }

    if (Socket==-1)
    {
        free(NewStruct);
        return ZESTETM1_SOCKET_ERROR;
    }

    NewStruct->Magic = ZESTETM1_CONNECTION_HANDLE_MAGIC;
    NewStruct->Type = Type;
    NewStruct->Port = Port;
    NewStruct->LocalPort = LocalPort;
    NewStruct->Socket = Socket;
    NewStruct->CardInfo = CardInfo;
    *Connection = NewStruct;

    return ZESTETM1_SUCCESS;
}
```

7.21.3.3 ZestETM1_ReadData()

```
static ZESTETM1_STATUS ZestETM1_ReadData (
            ZESTETM1_CONNECTION Connection,
            void * Buffer,
            uint32_t Length,
            unsigned long * Read,
            uint32_t Timeout )  [static]
```

Definition at line 358 of file Data.c.

```
{
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
    fd_set ReadFDS;
    struct timeval Time;
    uint32_t BufferPtr = 0;
    int Result;
    struct sockaddr_in Target;
    int TargetLen = (int)sizeof(struct sockaddr_in);

    if (Conn==NULL)
    {
        if (Read!=NULL) *Read = 0;
        return ZESTETM1_NULL_PARAMETER;
    }
    if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
    {
        if (Read!=NULL) *Read = 0;
        return ZESTETM1_ILLEGAL_CONNECTION;
    }

    do
    {
        int OK = 1;

        // Wait for socket to become ready
        {
            int CurTime = 0;
            while (CurTime!=Timeout)
            {
                int T = (Timeout-CurTime)<1000 ? Timeout-CurTime : 1000;
                FD_ZERO(&ReadFDS);
                FD_SET(Conn->Socket, &ReadFDS);
                Time.tv_sec = T/1000;
                Time.tv_usec = (T%1000)*1000;
                Result = select((int)Conn->Socket+1, &ReadFDS, NULL, NULL, &Time);
                if (Result>0) break;
                CurTime+=T;
            }
        }
        if (Result<0 || !FD_ISSET(Conn->Socket, &ReadFDS))
        {
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_TIMEOUT;//FIXME: Any other errors?
        }

        if (Conn->Type==ZESTETM1_TYPE_UDP)
        {
            Result = recvfrom(Conn->Socket, (char *)Buffer+BufferPtr,
                              Length-BufferPtr, 0,
                              (struct sockaddr *)&Target, &TargetLen);
            if (Conn->LocalPort!=0 && Target.sin_port!=htons(Conn->LocalPort))
                OK = 0;
        }
        else if (Conn->Type==ZESTETM1_TYPE_TCP)
        {
            Result = recv(Conn->Socket, (char *)Buffer+BufferPtr,
                          Length-BufferPtr, 0);
        }
        else
        {
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_ILLEGAL_CONNECTION;
        }

        // Update counters
        if (Result>0 && OK==1)
        {
            BufferPtr += Result;
        }
        else if (Result==0)
        {
            // Connection closed
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_SOCKET_CLOSED;
        }
        else if (Result<0)
        {
            // Socket error
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_SOCKET_ERROR;
        }
    } while (Result>0 && BufferPtr<Length);

    if (Read!=NULL) *Read = BufferPtr;
    return ZESTETM1_SUCCESS;
}
```

7.21.3.4 ZestETM1_SendCommand()

```
ZESTETM1_STATUS ZestETM1_SendCommand (
               ZESTETM1_CARD_INFO * CardInfo,
               ZESTETM1_CONNECTION Connection,
               void * WriteBuffer,
               uint32_t WriteLen,
               void * ReadBuffer,
               uint32_t ReadLen,
               int WaitForAck )
```

Definition at line 453 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    unsigned long Written;
    unsigned long Received;

    // Send/receive data
    Result = ZestETM1_WriteData(Connection, WriteBuffer, WriteLen, &Written,
                               CardInfo->Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        return Result;
    }
    if (Written!=WriteLen)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }
    *((uint8_t *)ReadBuffer) = 0;
    if (WaitForAck==1)
    {
        Result = ZestETM1_ReadData(Connection, ReadBuffer, ReadLen, &Received,
                                  CardInfo->Timeout);
        if (Result!=ZESTETM1_SUCCESS)
        {
            return Result;
        }
        if (Received!=ReadLen)
        {
            return ZESTETM1_INTERNAL_ERROR;
        }
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.5 ZestETM1_SPIReadWrite()

```
ZESTETM1_STATUS ZestETM1_SPIReadWrite (
            ZESTETM1_CARD_INFO * CardInfo,
            ZESTETM1_CONNECTION Connection,
            int Device,
            int WordLen,
            uint32_t * WriteData,
            uint32_t * ReadData,
            uint32_t Length,
            int ReleaseCS,
            int WaitForAck )
```

Definition at line 495 of file Data.c.

```
{
    uint8_t Buffer[65536];
    uint32_t *BufPtr;
    uint32_t i;
    ZESTETM1_STATUS Result;

    // Build command
    Buffer[0] = ZESTETM1_COMMAND_SPI;// Command byte
    Buffer[1] = Device;              // SPI device
    Buffer[2] = WordLen;             // Word length
    Buffer[3] = ReleaseCS;           // Release CS on completion
    BufPtr = (uint32_t *)(Buffer+4);

    BufPtr[0] = WriteData==NULL ? 0 : ZESTETM1_REVERSE(Length);    // Num words
    BufPtr[1] = ReadData==NULL ? 0 : ZESTETM1_REVERSE(Length);
    if (WriteData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            BufPtr[2+i] = ZESTETM1_REVERSE(WriteData[i]);
        }
    }
    else
        memset(BufPtr+2, 0, Length*4);

    // Send command
    Result = ZestETM1_SendCommand(CardInfo, Connection, Buffer,
                                  WriteData==NULL ? 12 : 12+Length*4,
                                  Buffer, ReadData==NULL ? 4 : 4+Length*4,
                                  WaitForAck);
    if (WaitForAck==0)
        return Result;

    // Extract results
    if (Buffer[0]!=ZESTETM1_COMMAND_SPI || Buffer[1]!=0)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }

    BufPtr = (uint32_t *)(Buffer+4);
    if (ReadData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            ReadData[i] = ZESTETM1_REVERSE(BufPtr[i]);
        }
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.6 ZestETM1_WriteData()

```
static ZESTETM1_STATUS ZestETM1_WriteData (
            ZESTETM1_CONNECTION Connection,
            void * Buffer,
            uint32_t Length,
            unsigned long * Written,
            uint32_t Timeout )  [static]
```

Definition at line 249 of file Data.c.

```
{
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
    fd_set WriteFDS;
    struct timeval Time;
    uint32_t BufferPtr = 0;
    int Result;
    int TargetLen = (int)sizeof(struct sockaddr_in);
    unsigned int MaxSize;

    if (Conn==NULL)
    {
        if (Written!=NULL) *Written = 0;
        return ZESTETM1_NULL_PARAMETER;
    }
    if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
    {
        if (Written!=NULL) *Written = 0;
        return ZESTETM1_ILLEGAL_CONNECTION;
    }

    if (Conn->Type==ZESTETM1_TYPE_UDP)
    {
        // Get maximum transfer size
#if defined(MSVC) || defined(WINGCC)
        int MaxSizeLen = sizeof(MaxSize);
        getsockopt(Conn->Socket, SOL_SOCKET, SO_MAX_MSG_SIZE,
                   (char *)&MaxSize, &MaxSizeLen);
#else
        //FIXME: Linux doesn't support SO_MAX_MSG_SIZE
        MaxSize = 65507;
#endif
    }

    do
    {
        uint32_t Bytes = Length-BufferPtr;

        if (Conn->Type==ZESTETM1_TYPE_UDP && Bytes>MaxSize)
            Bytes = MaxSize;

        // Wait for socket to become ready
        {
            int CurTime = 0;
            while (CurTime!=Timeout)
            {
                int T = (Timeout-CurTime)<1000 ? Timeout-CurTime : 1000;
                FD_ZERO(&WriteFDS);
                FD_SET(Conn->Socket, &WriteFDS);
                Time.tv_sec = T/1000;
                Time.tv_usec = (T%1000)*1000;
                Result = select((int)Conn->Socket+1, NULL, &WriteFDS, NULL, &Time);
                if (Result>0) break;
                CurTime+=T;
            }
        }
        if (Result<0 || !FD_ISSET(Conn->Socket, &WriteFDS))
        {
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_TIMEOUT;//FIXME: Any other errors?
        }

        if (Conn->Type==ZESTETM1_TYPE_UDP)
        {
            Result = sendto(Conn->Socket, (char *)Buffer+BufferPtr,
                            Bytes, 0,
                            (struct sockaddr *)&Conn->Target, TargetLen);
        }
        else if (Conn->Type==ZESTETM1_TYPE_TCP)
        {
            Result = send(Conn->Socket, (char *)Buffer+BufferPtr,
                          Bytes, 0);
        }
        else
        {
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_ILLEGAL_CONNECTION;
        }

        // Update counters
        if (Result>0)
        {
            BufferPtr += Result;
        }
        else if (Result==0)
        {
            // Connection closed
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_SOCKET_CLOSED;
        }
        else
        {
            // Socket error
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_SOCKET_ERROR;
        }
    } while (Result>0 && BufferPtr<Length);

    if (Written!=NULL) *Written = BufferPtr;
    return ZESTETM1_SUCCESS;
}
```

7.21.3.7 ZestETM1CloseConnection()

```
ZESTETM1_STATUS ZestETM1CloseConnection (
            ZESTETM1_CONNECTION Connection )
```

Definition at line 575 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1CloseConnection", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.8 ZestETM1OpenConnection()

```
ZESTETM1_STATUS ZestETM1OpenConnection (
            ZESTETM1_CARD_INFO * CardInfo,
            ZESTETM1_CONNECTION_TYPE Type,
            uint16_t Port,
            uint16_t LocalPort,
            ZESTETM1_CONNECTION * Connection )
```

Definition at line 555 of file Data.c.

```
{
    ZESTETM1_STATUS Result;

    Result = ZestETM1_OpenConnection(CardInfo, Type, Port, LocalPort, Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1OpenConnection", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.9 ZestETM1ReadData()

```
ZESTETM1_STATUS ZestETM1ReadData (
            ZESTETM1_CONNECTION Connection,
            void * Buffer,
            unsigned long Length,
            unsigned long * Read,
            unsigned long Timeout )
```

Definition at line 613 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_ReadData(Connection, Buffer, Length, Read, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1ReadData", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.10 ZestETM1ReadRegister()

```
ZESTETM1_STATUS ZestETM1ReadRegister (
            ZESTETM1_CARD_INFO * CardInfo,
            unsigned long Addr,
            unsigned short * Data )
```

Definition at line 728 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_READ_REG_CMD Cmd;
    ZESTETM1_READ_REG_RESPONSE Response;

    if (CardInfo==NULL || Data==NULL)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    // Read register from the device
    Cmd.Command = ZESTETM1_COMMAND_READ_REG;
    Cmd.Addr = (uint8_t)Addr;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_INTERNAL_ERROR);
    }
    *Data = ((Response.Value»8)&0xff) | ((Response.Value&0xff)«8);

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.11 ZestETM1SetInterrupt()

```
ZESTETM1_STATUS ZestETM1SetInterrupt (
              ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 780 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_MAILBOX_INT_CMD Cmd;
    ZESTETM1_MAILBOX_INT_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_NULL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    // Send command to set interrupt
    Cmd.Command = ZESTETM1_COMMAND_MAILBOX_INT;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.12 ZestETM1SPIReadWrite()

```
ZESTETM1_STATUS ZestETM1SPIReadWrite (
              ZESTETM1_CARD_INFO * CardInfo,
              ZESTETM1_SPI_RATE Rate,
              int WordLen,
              void * WriteData,
              void * ReadData,
              unsigned long Length,
              int ReleaseCS )
```

Definition at line 635 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    unsigned long RateVal = Rate==ZESTETM1_SPI_RATE_35 ? ZESTETM1_RATE_40MHz :
                            Rate==ZESTETM1_SPI_RATE_17_5 ? ZESTETM1_RATE_20MHz : ZESTETM1_RATE_10MHz;

    if (CardInfo==NULL || (WriteData==NULL && ReadData==NULL))
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_NULL_PARAMETER);
    }
    if (WordLen<1 || WordLen>32 || Length>16384)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_SPIReadWrite(CardInfo, Connection, ZESTETM1_USER_DEVICE_ID|RateVal, WordLen,
      (uint32_t *)WriteData, (uint32_t *)ReadData, Length, ReleaseCS, 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.13 ZestETM1WriteData()

```
ZESTETM1_STATUS ZestETM1WriteData (
            ZESTETM1_CONNECTION Connection,
            void * Buffer,
            unsigned long Length,
            unsigned long * Written,
            unsigned long Timeout )
```

Definition at line 592 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_WriteData(Connection, Buffer, Length, Written, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1WriteData", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.21.3.14 ZestETM1WriteRegister()

```
ZESTETM1_STATUS ZestETM1WriteRegister (
            ZESTETM1_CARD_INFO * CardInfo,
            unsigned long Addr,
            unsigned short Data )
```

Definition at line 678 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_WRITE_REG_CMD Cmd;
    ZESTETM1_WRITE_REG_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    // Write register to the device
    //FIXME: Do we want to be able to copy this value to flash?
    Cmd.Command = ZESTETM1_COMMAND_WRITE_REG;
    Cmd.Addr = (uint8_t)Addr;
    Cmd.Data = ((Data»8)&0xff) | (Data&0xff);
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.22 Data.c

Go to the documentation of this file.

```
// File:      Data.c
//
// Purpose:
//    ZestETM1 Host Library
//    Data transfer functions
//
// Version: 1.00
// Date: 11/10/12

// Copyright (C) 2012 Orange Tree Technologies Ltd. All rights reserved.
// Orange Tree Technologies grants the purchaser of a ZestETM1 the right to use and
// modify this source code in any form in designs that target the ZestETM1.
// Orange Tree Technologies prohibits the use of this source code or any modification of
// it in any form in designs that target any other hardware unless the purchaser of the
// ZestETM1 has purchased the appropriate licence from Orange Tree Technologies.
// Contact Orange Tree Technologies if you want to purchase such a licence.

```

```
//*****************************************************************************************
//**
//**  Disclaimer: LIMITED WARRANTY AND DISCLAIMER. These designs are
//**              provided to you "as is". Orange Tree Technologies and its licensors
//**              make and you receive no warranties or conditions, express, implied,
//**              statutory or otherwise, and Orange Tree Technologies specifically
//**              disclaims any implied warranties of merchantability, non-infringement,
//**              or fitness for a particular purpose. Orange Tree Technologies does not
//**              warrant that the functions contained in these designs will meet your
//**              requirements, or that the operation of these designs will be
//**              uninterrupted or error free, or that defects in the Designs will be
//**              corrected. Furthermore, Orange Tree Technologies does not warrant or
//**              make any representations regarding use or the results of the use of the
//**              designs in terms of correctness, accuracy, reliability, or otherwise.
//**
//**              LIMITATION OF LIABILITY. In no event will Orange Tree Technologies
//**              or its licensors be liable for any loss of data, lost profits, cost or
//**              procurement of substitute goods or services, or for any special,
//**              incidental, consequential, or indirect damages arising from the use or
//**              operation of the designs or accompanying documentation, however caused
//**              and on any theory of liability. This limitation will apply even if
//**              Orange Tree Technologies has been advised of the possibility of such
//**              damage. This limitation shall apply notwithstanding the failure of the
//**              essential purpose of any limited remedies herein.
//**
//*****************************************************************************************

#define _CRT_SECURE_NO_WARNINGS
#ifdef WINGCC
#define __USE_W32_SOCKETS
#endif

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include "ZestETM1.h"
#include "Private.h"

// Read/write register command structures
typedef struct
{
    uint8_t Command;
    uint8_t Addr;
    uint16_t Data;
} ZESTETM1_WRITE_REG_CMD;
typedef struct
{
    uint8_t Command;
    uint8_t Status;
    uint8_t Dummy1[2];
} ZESTETM1_WRITE_REG_RESPONSE;
typedef struct
{
    uint8_t Command;
    uint8_t Addr;
    uint8_t Dummy[2];
} ZESTETM1_READ_REG_CMD;
typedef struct
{
    uint8_t Command;
    uint8_t Status;
    uint16_t Value;
} ZESTETM1_READ_REG_RESPONSE;
typedef struct
{
    uint8_t Command;
    uint8_t Dummy[3];
} ZESTETM1_MAILBOX_INT_CMD;
typedef struct
{
    uint8_t Command;
    uint8_t Status;
    uint16_t Dummy;
} ZESTETM1_MAILBOX_INT_RESPONSE;
#define ZESTETM1_COMMAND_SPI 0xee
#define ZESTETM1_COMMAND_WRITE_REG 0xf6
#define ZESTETM1_COMMAND_READ_REG 0xf7
#define ZESTETM1_COMMAND_MAILBOX_INT 0xf8

/**********************************************************************
```

```
* Open a connection to a ZestETM1 for data transfer (internal version) *
*********************************************************************/
ZESTETM1_STATUS ZestETM1_OpenConnection(ZESTETM1_CARD_INFO *CardInfo,
                                        ZESTETM1_CONNECTION_TYPE Type,
                                        uint16_t Port,
                                        uint16_t LocalPort,
                                        ZESTETM1_CONNECTION *Connection)
{
    ZESTETM1_CONNECTION_STRUCT *NewStruct;
    SOCKET Socket = -1;
    char AddrBuffer[32];
    char PortBuffer[32];

    if (Connection==NULL || CardInfo==NULL)
    {
        return ZESTETM1_NULL_PARAMETER;
    }

    // Allocate data structure
    NewStruct = malloc(sizeof(ZESTETM1_CONNECTION_STRUCT));
    if (NewStruct==NULL)
    {
        return ZESTETM1_OUT_OF_MEMORY;
    }

    // Build target addresses
    sprintf(AddrBuffer, "%d.%d.%d.%d", CardInfo->IPAddr[0], CardInfo->IPAddr[1],
                    CardInfo->IPAddr[2], CardInfo->IPAddr[3]);
    sprintf(PortBuffer, "%d", Port);

    if (Type==ZESTETM1_TYPE_UDP)
    {
        // Open UDP connection
        struct sockaddr_in SourceIP;
        int SourceLen = (int)sizeof(struct sockaddr_in);
        Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (Socket<0)
            return ZESTETM1_SOCKET_ERROR;
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = htons(LocalPort);
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);

        NewStruct->Target.sin_family = AF_INET;
        NewStruct->Target.sin_addr.s_addr = inet_addr(AddrBuffer);
        NewStruct->Target.sin_port = htons(atoi(PortBuffer));
    }
    else if (Type==ZESTETM1_TYPE_TCP)
    {
        // Open TCP connection
        struct addrinfo *AddrResult = NULL,
                        *Ptr = NULL,
                        Hints;
        int Result;
        struct sockaddr_in SourceIP;
        int SourceLen = (int)sizeof(struct sockaddr_in);

        memset(&Hints, 0, sizeof(Hints));
        Hints.ai_family = AF_UNSPEC;
        Hints.ai_socktype = SOCK_STREAM;
        Hints.ai_protocol = IPPROTO_TCP;

        // Resolve the server address and port
        Result = getaddrinfo(AddrBuffer, PortBuffer, &Hints, &AddrResult);
        if (Result!=0)
        {
            free(NewStruct);
            return ZESTETM1_SOCKET_ERROR;
        }

        // Attempt to connect to an address until one succeeds
        for (Ptr=AddrResult; Ptr!=NULL; Ptr=Ptr->ai_next)
        {
            // Create a SOCKET for connecting to server
            Socket = socket(Ptr->ai_family, Ptr->ai_socktype,
                            Ptr->ai_protocol);
            if (Socket<0)
            {
                freeaddrinfo(AddrResult);
                free(NewStruct);
                return ZESTETM1_SOCKET_ERROR;
            }

            // Connect to ZestETM1
            Result = connect(Socket, Ptr->ai_addr, (int)Ptr->ai_addrlen);
            if (Result<0)
            {
                closesocket(Socket);
                Socket = -1;
                continue;
            }
            break;
        }

        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);
        freeaddrinfo(AddrResult);
    }
    else
    {
        free(NewStruct);
        return ZESTETM1_INVALID_CONNECTION_TYPE;
    }

    if (Socket==-1)
    {
        free(NewStruct);
        return ZESTETM1_SOCKET_ERROR;
    }

    NewStruct->Magic = ZESTETM1_CONNECTION_HANDLE_MAGIC;
    NewStruct->Type = Type;
    NewStruct->Port = Port;
    NewStruct->LocalPort = LocalPort;
    NewStruct->Socket = Socket;
    NewStruct->CardInfo = CardInfo;
    *Connection = NewStruct;

    return ZESTETM1_SUCCESS;
}

/*****************************************************
* Close a connection to a ZestETM1 (internal version *
*****************************************************/
ZESTETM1_STATUS ZestETM1_CloseConnection(ZESTETM1_CONNECTION Connection)
{
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;

    if (Conn==NULL)
    {
        return ZESTETM1_NULL_PARAMETER;
    }
    if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
    {
        return ZESTETM1_ILLEGAL_CONNECTION;
    }

    // Cleanup
    closesocket(Conn->Socket);
    Conn->Magic = 0;
    free(Conn);

    return ZESTETM1_SUCCESS;
}

/*******************************************************
* Write data to ZestETM1 connection (internal version) *
*******************************************************/
static ZESTETM1_STATUS ZestETM1_WriteData(ZESTETM1_CONNECTION Connection,
                                        void *Buffer,
                                        uint32_t Length,
                                        unsigned long *Written,
                                        uint32_t Timeout)
{
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
    fd_set WriteFDS;
    struct timeval Time;
    uint32_t BufferPtr = 0;
    int Result;
    int TargetLen = (int)sizeof(struct sockaddr_in);
    unsigned int MaxSize;

    if (Conn==NULL)
    {
        if (Written!=NULL) *Written = 0;
        return ZESTETM1_NULL_PARAMETER;
    }
    if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
    {
        if (Written!=NULL) *Written = 0;
        return ZESTETM1_ILLEGAL_CONNECTION;
    }

    if (Conn->Type==ZESTETM1_TYPE_UDP)
    {
        // Get maximum transfer size
#if defined(MSVC) || defined(WINGCC)
        int MaxSizeLen = sizeof(MaxSize);
        getsockopt(Conn->Socket, SOL_SOCKET, SO_MAX_MSG_SIZE,
                   (char *)&MaxSize, &MaxSizeLen);
#else
        //FIXME: Linux doesn't support SO_MAX_MSG_SIZE
        MaxSize = 65507;
#endif
    }

    do
    {
        uint32_t Bytes = Length-BufferPtr;

        if (Conn->Type==ZESTETM1_TYPE_UDP && Bytes>MaxSize)
            Bytes = MaxSize;

        // Wait for socket to become ready
        {
            int CurTime = 0;
            while (CurTime!=Timeout)
            {
                int T = (Timeout-CurTime)<1000 ? Timeout-CurTime : 1000;
                FD_ZERO(&WriteFDS);
                FD_SET(Conn->Socket, &WriteFDS);
                Time.tv_sec = T/1000;
                Time.tv_usec = (T%1000)*1000;
                Result = select((int)Conn->Socket+1, NULL, &WriteFDS, NULL, &Time);
                if (Result>0) break;
                CurTime+=T;
            }
        }
        if (Result<0 || !FD_ISSET(Conn->Socket, &WriteFDS))
        {
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_TIMEOUT;//FIXME: Any other errors?
        }

        if (Conn->Type==ZESTETM1_TYPE_UDP)
        {
            Result = sendto(Conn->Socket, (char *)Buffer+BufferPtr,
                            Bytes, 0,
                            (struct sockaddr *)&Conn->Target, TargetLen);
        }
        else if (Conn->Type==ZESTETM1_TYPE_TCP)
        {
            Result = send(Conn->Socket, (char *)Buffer+BufferPtr,
                          Bytes, 0);
        }
        else
        {
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_ILLEGAL_CONNECTION;
        }

        // Update counters
        if (Result>0)
        {
            BufferPtr += Result;
        }
        else if (Result==0)
        {
            // Connection closed
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_SOCKET_CLOSED;
        }
        else
        {
            // Socket error
            if (Written!=NULL) *Written = BufferPtr;
            return ZESTETM1_SOCKET_ERROR;
        }
    } while (Result>0 && BufferPtr<Length);

    if (Written!=NULL) *Written = BufferPtr;
    return ZESTETM1_SUCCESS;
}

/*********************************************************
* Read data from a ZestETM1 connection (internal version) *
*********************************************************/
static ZESTETM1_STATUS ZestETM1_ReadData(ZESTETM1_CONNECTION Connection,
                                       void *Buffer,
                                       uint32_t Length,
                                       unsigned long *Read,
                                       uint32_t Timeout)
{
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
    fd_set ReadFDS;
    struct timeval Time;
    uint32_t BufferPtr = 0;
    int Result;
    struct sockaddr_in Target;
    int TargetLen = (int)sizeof(struct sockaddr_in);

    if (Conn==NULL)
    {
        if (Read!=NULL) *Read = 0;
        return ZESTETM1_NULL_PARAMETER;
    }
    if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
    {
        if (Read!=NULL) *Read = 0;
        return ZESTETM1_ILLEGAL_CONNECTION;
    }

    do
    {
        int OK = 1;

        // Wait for socket to become ready
        {
            int CurTime = 0;
            while (CurTime!=Timeout)
            {
                int T = (Timeout-CurTime)<1000 ? Timeout-CurTime : 1000;
                FD_ZERO(&ReadFDS);
                FD_SET(Conn->Socket, &ReadFDS);
                Time.tv_sec = T/1000;
                Time.tv_usec = (T%1000)*1000;
                Result = select((int)Conn->Socket+1, &ReadFDS, NULL, NULL, &Time);
                if (Result>0) break;
                CurTime+=T;
            }
        }
        if (Result<0 || !FD_ISSET(Conn->Socket, &ReadFDS))
        {
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_TIMEOUT;//FIXME: Any other errors?
        }

        if (Conn->Type==ZESTETM1_TYPE_UDP)
        {
            Result = recvfrom(Conn->Socket, (char *)Buffer+BufferPtr,
                              Length-BufferPtr, 0,
                              (struct sockaddr *)&Target, &TargetLen);
            if (Conn->LocalPort!=0 && Target.sin_port!=htons(Conn->LocalPort))
                OK = 0;
        }
        else if (Conn->Type==ZESTETM1_TYPE_TCP)
        {
            Result = recv(Conn->Socket, (char *)Buffer+BufferPtr,
                          Length-BufferPtr, 0);
        }
        else
        {
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_ILLEGAL_CONNECTION;
        }

        // Update counters
        if (Result>0 && OK==1)
        {
            BufferPtr += Result;
        }
        else if (Result==0)
        {
            // Connection closed
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_SOCKET_CLOSED;
        }
        else if (Result<0)
        {
            // Socket error
            if (Read!=NULL) *Read = BufferPtr;
            return ZESTETM1_SOCKET_ERROR;
        }
    } while (Result>0 && BufferPtr<Length);

    if (Read!=NULL) *Read = BufferPtr;
    return ZESTETM1_SUCCESS;
}

/****************************************************
* Send a control command to GigEx and get response *
****************************************************/
ZESTETM1_STATUS ZestETM1_SendCommand(ZESTETM1_CARD_INFO *CardInfo,
                                   ZESTETM1_CONNECTION Connection,
                                   void *WriteBuffer, uint32_t WriteLen,
                                   void *ReadBuffer, uint32_t ReadLen,
                                   int WaitForAck)
{
    ZESTETM1_STATUS Result;
    unsigned long Written;
    unsigned long Received;

    // Send/receive data
    Result = ZestETM1_WriteData(Connection, WriteBuffer, WriteLen, &Written,
                               CardInfo->Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        return Result;
    }
    if (Written!=WriteLen)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }
    *((uint8_t *)ReadBuffer) = 0;
    if (WaitForAck==1)
    {
        Result = ZestETM1_ReadData(Connection, ReadBuffer, ReadLen, &Received,
                                  CardInfo->Timeout);
        if (Result!=ZESTETM1_SUCCESS)
        {
            return Result;
        }
        if (Received!=ReadLen)
        {
            return ZESTETM1_INTERNAL_ERROR;
        }
    }

    return ZESTETM1_SUCCESS;
}

/**********************************************
* Send data to/from SPI port on GigEx device *
**********************************************/
ZESTETM1_STATUS ZestETM1_SPIReadWrite(ZESTETM1_CARD_INFO *CardInfo,
                                    ZESTETM1_CONNECTION Connection,
                                    int Device,
                                    int WordLen, uint32_t *WriteData,
                                    uint32_t *ReadData, uint32_t Length,
                                    int ReleaseCS, int WaitForAck)
{
    uint8_t Buffer[65536];
    uint32_t *BufPtr;
    uint32_t i;
    ZESTETM1_STATUS Result;

    // Build command
    Buffer[0] = ZESTETM1_COMMAND_SPI;// Command byte
    Buffer[1] = Device;             // SPI device
    Buffer[2] = WordLen;            // Word length
    Buffer[3] = ReleaseCS;          // Release CS on completion
    BufPtr = (uint32_t *)(Buffer+4);

    BufPtr[0] = WriteData==NULL ? 0 : ZESTETM1_REVERSE(Length);    // Num words
    BufPtr[1] = ReadData==NULL ? 0 : ZESTETM1_REVERSE(Length);
    if (WriteData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            BufPtr[2+i] = ZESTETM1_REVERSE(WriteData[i]);
        }
    }
    else
        memset(BufPtr+2, 0, Length*4);

    // Send command
    Result = ZestETM1_SendCommand(CardInfo, Connection, Buffer,
                                  WriteData==NULL ? 12 : 12+Length*4,
                                  Buffer, ReadData==NULL ? 4 : 4+Length*4,
                                  WaitForAck);
    if (WaitForAck==0)
        return Result;

    // Extract results
    if (Buffer[0]!=ZESTETM1_COMMAND_SPI || Buffer[1]!=0)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }

    BufPtr = (uint32_t *)(Buffer+4);
    if (ReadData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            ReadData[i] = ZESTETM1_REVERSE(BufPtr[i]);
        }
    }

    return ZESTETM1_SUCCESS;
}

/****************************************************
* Open a connection to a ZestETM1 for data transfer *
****************************************************/
ZESTETM1_STATUS ZestETM1OpenConnection(ZESTETM1_CARD_INFO *CardInfo,
                                       ZESTETM1_CONNECTION_TYPE Type,
                                       uint16_t Port,
                                       uint16_t LocalPort,
                                       ZESTETM1_CONNECTION *Connection)
{
    ZESTETM1_STATUS Result;

    Result = ZestETM1_OpenConnection(CardInfo, Type, Port, LocalPort, Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1OpenConnection", Result);
    }

    return ZESTETM1_SUCCESS;
}

/**********************************
* Close a connection to a ZestETM1 *
**********************************/
ZESTETM1_STATUS ZestETM1CloseConnection(ZESTETM1_CONNECTION Connection)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1CloseConnection", Result);
    }

    return ZESTETM1_SUCCESS;
}

/**********************************
* Write data to ZestETM1 connection *
**********************************/
ZESTETM1_STATUS ZestETM1WriteData(ZESTETM1_CONNECTION Connection,
                                void *Buffer,
                                unsigned long Length,
                                unsigned long *Written,
                                unsigned long Timeout)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_WriteData(Connection, Buffer, Length, Written, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1WriteData", Result);
    }

    return ZESTETM1_SUCCESS;
}

/**************************************
* Read data from a ZestETM1 connection *
**************************************/
ZESTETM1_STATUS ZestETM1ReadData(ZESTETM1_CONNECTION Connection,
                               void *Buffer,
                               unsigned long Length,
                               unsigned long *Read,
                               unsigned long Timeout)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_ReadData(Connection, Buffer, Length, Read, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1ReadData", Result);
    }

    return ZESTETM1_SUCCESS;
}


/*********************************
* Read data from master SPI port *
*********************************/
ZESTETM1_STATUS ZestETM1SPIReadWrite(ZESTETM1_CARD_INFO *CardInfo, ZESTETM1_SPI_RATE Rate,
                                     int WordLen, void *WriteData,
                                     void *ReadData, unsigned long Length,
                                     int ReleaseCS)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    unsigned long RateVal = Rate==ZESTETM1_SPI_RATE_35 ? ZESTETM1_RATE_40MHz :
                            Rate==ZESTETM1_SPI_RATE_17_5 ? ZESTETM1_RATE_20MHz : ZESTETM1_RATE_10MHz;

    if (CardInfo==NULL || (WriteData==NULL && ReadData==NULL))
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_NULL_PARAMETER);
    }
    if (WordLen<1 || WordLen>32 || Length>16384)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_SPIReadWrite(CardInfo, Connection, ZESTETM1_USER_DEVICE_ID|RateVal, WordLen,
(uint32_t *)WriteData, (uint32_t *)ReadData, Length, ReleaseCS, 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    return ZESTETM1_SUCCESS;
}

/**************************************
* Read/Write user interface registers *
**************************************/
ZESTETM1_STATUS ZestETM1WriteRegister(ZESTETM1_CARD_INFO *CardInfo, unsigned long Addr, unsigned short
Data)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_WRITE_REG_CMD Cmd;
    ZESTETM1_WRITE_REG_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    // Write register to the device
    //FIXME: Do we want to be able to copy this value to flash?
    Cmd.Command = ZESTETM1_COMMAND_WRITE_REG;
    Cmd.Addr = (uint8_t)Addr;
    Cmd.Data = ((Data»8)&0xff) | (Data&0xff);
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}

ZESTETM1_STATUS ZestETM1ReadRegister(ZESTETM1_CARD_INFO *CardInfo, unsigned long Addr, unsigned short
*Data)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_READ_REG_CMD Cmd;
    ZESTETM1_READ_REG_RESPONSE Response;

    if (CardInfo==NULL || Data==NULL)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    // Read register from the device
    Cmd.Command = ZESTETM1_COMMAND_READ_REG;
    Cmd.Addr = (uint8_t)Addr;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_INTERNAL_ERROR);
    }
    *Data = ((Response.Value»8)&0xff) | ((Response.Value&0xff)«8);

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}

/************************
* Set mailbox interrupt *
************************/
ZESTETM1_STATUS ZestETM1SetInterrupt(ZESTETM1_CARD_INFO *CardInfo)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_MAILBOX_INT_CMD Cmd;
    ZESTETM1_MAILBOX_INT_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_NULL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    // Send command to set interrupt
    Cmd.Command = ZESTETM1_COMMAND_MAILBOX_INT;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.23 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/Error.c File Reference

```
#include <stdint.h>
#include "ZestETM1.h"
#include "Private.h"
```

Functions

- ZESTETM1_STATUS ZestETM1RegisterErrorHandler (ZESTETM1_ERROR_FUNC Function)
- ZESTETM1_STATUS ZestETM1GetErrorMessage (ZESTETM1_STATUS Status, char **Buffer)

Variables

- char * ZestETM1_ErrorStrings [ ]
- ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler

7.23.1 Function Documentation

7.23.1.1 ZestETM1GetErrorMessage()

```
ZESTETM1_STATUS ZestETM1GetErrorMessage (
              ZESTETM1_STATUS Status,
              char ** Buffer )
```

Definition at line 84 of file Error.c.

```
00086 {
00087     if (Status>ZESTETM1_MAX_ERROR ||
00088         (Status<ZESTETM1_ERROR_BASE && Status>=ZESTETM1_MAX_WARNING) ||
00089         (Status<ZESTETM1_WARNING_BASE && Status>=ZESTETM1_MAX_INFO))
00090     {
00091         return ZESTETM1_ILLEGAL_STATUS_CODE;
00092     }
00093 
00094     *Buffer = ZESTETM1_ERROR_STRING(Status);
00095     return ZESTETM1_SUCCESS;
00096 }
```

7.23.1.2 ZestETM1RegisterErrorHandler()

```
ZESTETM1_STATUS ZestETM1RegisterErrorHandler (
              ZESTETM1_ERROR_FUNC Function )
```

Definition at line 74 of file Error.c.

```
00075 {
00076     ZestETM1_ErrorHandler = Function;
00077     return ZESTETM1_SUCCESS;
00078 }
```

7.23.2 Variable Documentation

7.23.2.1 ZestETM1_ErrorHandler

```
ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler
```

Definition at line 67 of file Error.c.

7.23.2.2 ZestETM1_ErrorStrings

```
char* ZestETM1_ErrorStrings[]
```

Initial value:

```
=
{
    "Success (no error)",
    "Error communicating with socket",
    "An unspecified internal error occurred",
    "Status code is out of range",
    "NULL was used illegally as one of the parameter values",
    "Not enough memory to complete the requested operation",
    "The requested connection type is invalid",
    "The requested connection is invalid",
    "The connection was closed unexpectedly",
    "Operation timed out",
    "One of the parameters has an illegal value",

}
```

Definition at line 52 of file Error.c.

```
{
    "Success (no error)",
    "Error communicating with socket",
    "An unspecified internal error occurred",
    "Status code is out of range",
    "NULL was used illegally as one of the parameter values",
    "Not enough memory to complete the requested operation",
    "The requested connection type is invalid",
    "The requested connection is invalid",
    "The connection was closed unexpectedly",
    "Operation timed out",
    "One of the parameters has an illegal value",

};
```

7.24 Error.c

Go to the documentation of this file.

```
// File:      Error.c
//
// Purpose:
//    ZestETM1 Host Library
//    Error functions
//
// Version: 1.00
// Date: 11/10/12

// Copyright (C) 2012 Orange Tree Technologies Ltd. All rights reserved.
// Orange Tree Technologies grants the purchaser of a ZestETM1 the right to use and
// modify this source code in any form in designs that target the ZestETM1.
// Orange Tree Technologies prohibits the use of this source code or any modification of
// it in any form in designs that target any other hardware unless the purchaser of the
// ZestETM1 has purchased the appropriate licence from Orange Tree Technologies.
// Contact Orange Tree Technologies if you want to purchase such a licence.
```

```

//**************************************************************************************
//**
//**  Disclaimer: LIMITED WARRANTY AND DISCLAIMER. These designs are
//**              provided to you "as is". Orange Tree Technologies and its licensors
//**              make and you receive no warranties or conditions, express, implied,
//**              statutory or otherwise, and Orange Tree Technologies specifically
//**              disclaims any implied warranties of merchantability, non-infringement,
//**              or fitness for a particular purpose. Orange Tree Technologies does not
//**              warrant that the functions contained in these designs will meet your
//**              requirements, or that the operation of these designs will be
//**              uninterrupted or error free, or that defects in the Designs will be
//**              corrected. Furthermore, Orange Tree Technologies does not warrant or
//**              make any representations regarding use or the results of the use of the
//**              designs in terms of correctness, accuracy, reliability, or otherwise.
//**
//**              LIMITATION OF LIABILITY. In no event will Orange Tree Technologies
//**              or its licensors be liable for any loss of data, lost profits, cost or
//**              procurement of substitute goods or services, or for any special,
//**              incidental, consequential, or indirect damages arising from the use or
//**              operation of the designs or accompanying documentation, however caused
//**              and on any theory of liability. This limitation will apply even if
//**              Orange Tree Technologies has been advised of the possibility of such
//**              damage. This limitation shall apply notwithstanding the failure of the
//**              essential purpose of any limited remedies herein.
//**
//**************************************************************************************

#include <stdint.h>
#include "ZestETM1.h"
#include "Private.h"

/******************************************************************************
* Globals                                                                     *
******************************************************************************/
char *ZestETM1_ErrorStrings[] =
{
    "Success (no error)",
    "Error communicating with socket",
    "An unspecified internal error occurred",
    "Status code is out of range",
    "NULL was used illegally as one of the parameter values",
    "Not enough memory to complete the requested operation",
    "The requested connection type is invalid",
    "The requested connection is invalid",
    "The connection was closed unexpectedly",
    "Operation timed out",
    "One of the parameters has an illegal value",

};
ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler;


/******************************************************************************
* Register a user error handling function to be called                        *
* Set to NULL to disable error callbacks                                      *
******************************************************************************/
ZESTETM1_STATUS ZestETM1RegisterErrorHandler(ZESTETM1_ERROR_FUNC Function)
{
    ZestETM1_ErrorHandler = Function;
    return ZESTETM1_SUCCESS;
}


/******************************************************************************
* Get a human-readable error string for a status code                         *
******************************************************************************/
ZESTETM1_STATUS ZestETM1GetErrorMessage(ZESTETM1_STATUS Status,
                                      char **Buffer)
{
    if (Status>ZESTETM1_MAX_ERROR ||
        (Status<ZESTETM1_ERROR_BASE && Status>=ZESTETM1_MAX_WARNING) ||
        (Status<ZESTETM1_WARNING_BASE && Status>=ZESTETM1_MAX_INFO))
    {
        return ZESTETM1_ILLEGAL_STATUS_CODE;
    }

    *Buffer = ZESTETM1_ERROR_STRING(Status);
    return ZESTETM1_SUCCESS;
}

```

7.25 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/Main.c File Reference

```
#include <stdint.h>
#include <sys/socket.h>
#include "ZestETM1.h"
#include "Private.h"
```

Functions

- ZESTETM1_STATUS ZestETM1Init (void)
- ZESTETM1_STATUS ZestETM1Close (void)

7.25.1 Function Documentation

7.25.1.1 ZestETM1Close()

```
ZESTETM1_STATUS ZestETM1Close (
            void  )
```

Definition at line 92 of file Main.c.

```
{
#if defined(MSVC) || defined(WINGCC)
    WSACleanup();
#endif

    return ZESTETM1_SUCCESS;
}
```

7.25.1.2 ZestETM1Init()

```
ZESTETM1_STATUS ZestETM1Init (
            void  )
```

Definition at line 58 of file Main.c.

```
{
#if defined(MSVC) || defined(WINGCC)
    WORD VersionRequested;
    WSADATA WSAData;
    int Error;

    VersionRequested = MAKEWORD(2, 2);
    Error = WSAStartup(VersionRequested, &WSAData);
    if (Error!=0)
    {
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }

    // Confirm that the WinSock DLL supports 2.2.
    // Note that if the DLL supports versions greater
    // than 2.2 in addition to 2.2, it will still return
    // 2.2 in Version since that is the version we
    // requested.
    if (LOBYTE(WSAData.wVersion)!=2 ||
        HIBYTE(WSAData.wVersion)!=2)
    {
        WSACleanup( );
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }
#endif

    return ZESTETM1_SUCCESS;
}
```

7.26 Main.c

Go to the documentation of this file.

```
// File:      Main.c
//
// Purpose:
//    ZestETM1 Host Library
//    Main functions
//
// Version: 1.00
// Date: 11/10/12

// Copyright (C) 2012 Orange Tree Technologies Ltd. All rights reserved.
// Orange Tree Technologies grants the purchaser of a ZestETM1 the right to use and
// modify this source code in any form in designs that target the ZestETM1.
// Orange Tree Technologies prohibits the use of this source code or any modification of
// it in any form in designs that target any other hardware unless the purchaser of the
// ZestETM1 has purchased the appropriate licence from Orange Tree Technologies.
// Contact Orange Tree Technologies if you want to purchase such a licence.

//*****************************************************************************************
//**
//**  Disclaimer: LIMITED WARRANTY AND DISCLAIMER. These designs are
//**              provided to you "as is". Orange Tree Technologies and its licensors
//**              make and you receive no warranties or conditions, express, implied,
//**              statutory or otherwise, and Orange Tree Technologies specifically
//**              disclaims any implied warranties of merchantability, non-infringement,
//**              or fitness for a particular purpose. Orange Tree Technologies does not
//**              warrant that the functions contained in these designs will meet your
//**              requirements, or that the operation of these designs will be
//**              uninterrupted or error free, or that defects in the Designs will be
//**              corrected. Furthermore, Orange Tree Technologies does not warrant or
//**              make any representations regarding use or the results of the use of the
//**              designs in terms of correctness, accuracy, reliability, or otherwise.
//**
//**              LIMITATION OF LIABILITY. In no event will Orange Tree Technologies
//**              or its licensors be liable for any loss of data, lost profits, cost or
//**              procurement of substitute goods or services, or for any special,
//**              incidental, consequential, or indirect damages arising from the use or
//**              operation of the designs or accompanying documentation, however caused
//**              and on any theory of liability. This limitation will apply even if
//**              Orange Tree Technologies has been advised of the possibility of such
//**              damage. This limitation shall apply notwithstanding the failure of the
//**              essential purpose of any limited remedies herein.
//**
//*****************************************************************************************

#include <stdint.h>
#if defined(MSVC) || defined(WINGCC)
#include <winsock2.h>
#else
#include <sys/socket.h>
#endif
#include "ZestETM1.h"
#include "Private.h"

/**************************************************
* Main initialisation function.                  *
* Must be called before other ZestETM1 functions. *
**************************************************/
ZESTETM1_STATUS ZestETM1Init(void)
```

```
{
#if defined(MSVC) || defined(WINGCC)
    WORD VersionRequested;
    WSADATA WSAData;
    int Error;

    VersionRequested = MAKEWORD(2, 2);
    Error = WSAStartup(VersionRequested, &WSAData);
    if (Error!=0)
    {
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }

    // Confirm that the WinSock DLL supports 2.2.
    // Note that if the DLL supports versions greater
    // than 2.2 in addition to 2.2, it will still return
    // 2.2 in Version since that is the version we
    // requested.
    if (LOBYTE(WSAData.wVersion)!=2 ||
        HIBYTE(WSAData.wVersion)!=2)
    {
        WSACleanup( );
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }
#endif

    return ZESTETM1_SUCCESS;
}

/**************************************************
* Main clean up function.                         *
* Must be called after other ZestETM1 functions.  *
**************************************************/
ZESTETM1_STATUS ZestETM1Close(void)
{
#if defined(MSVC) || defined(WINGCC)
    WSACleanup();
#endif

    return ZESTETM1_SUCCESS;
}
```

7.27 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/↩ Private.h File Reference

```
#include <stdint.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
```

Classes

- struct ZESTETM1_CONNECTION_STRUCT

Macros

- #define SD_BOTH SHUT_RDWR
- #define closesocket close
- #define ZESTETM1_ERROR(f, x)

- #define ZESTETM1_ERROR_GENERAL(f, x)
- #define ZESTETM1_ERROR_CONN(f, x)
- #define ZESTETM1_ERROR_STRING(x)
- #define ZESTETM1_CONNECTION_HANDLE_MAGIC 0xdeadbed1
- #define ZESTETM1_DEFAULT_TIMEOUT 10000
- #define ZESTETM1_RATE_40MHz (0<<4)
- #define ZESTETM1_RATE_20MHz (1<<4)
- #define ZESTETM1_RATE_10MHz (2<<4)
- #define ZESTETM1_USER_DEVICE_ID (1)
- #define ZESTETM1_REVERSE(x) ((((x)&0xff)<<24) | (((x)&0xff00)<<8) | (((x)&0xff0000)>>8) | (((x)&0xff000000)>>24))

Typedefs

- typedef int SOCKET

Functions

- ZESTETM1_STATUS ZestETM1_OpenConnection (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION_TYPE Type, uint16_t Port, uint16_t LocalPort, ZESTETM1_CONNECTION ∗Connection)
- ZESTETM1_STATUS ZestETM1_CloseConnection (ZESTETM1_CONNECTION Connection)
- ZESTETM1_STATUS ZestETM1_SendCommand (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION Connection, void ∗WriteBuffer, uint32_t WriteLen, void ∗ReadBuffer, uint32_t ReadLen, int WaitForAck)
- ZESTETM1_STATUS ZestETM1_SPIReadWrite (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION Connection, int Device, int WordLen, uint32_t ∗WriteData, uint32_t ∗ReadData, uint32_t Length, int ReleaseCS, int WaitForAck)
- ZESTETM1_STATUS ZestETM1_WriteFlash (ZESTETM1_CARD_INFO ∗CardInfo, uint32_t Address, void ∗Buffer, uint32_t Length)
- ZESTETM1_STATUS ZestETM1_EraseFlashSector (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION Connection, uint32_t Address)

Variables

- ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler
- char ∗ ZestETM1_ErrorStrings [ ]

7.27.1 Class Documentation

7.27.1.1 struct ZESTETM1_CONNECTION_STRUCT

Definition at line 93 of file Private.h.

Class Members

ZESTETM1_CARD_INFO ∗	CardInfo	
uint16_t	LocalPort	
uint32_t	Magic	
uint16_t	Port	
SOCKET	Socket	
struct sockaddr_in	Target	
ZESTETM1_CONNECTION_TYPE	Type	

7.27.2 Macro Definition Documentation

7.27.2.1 closesocket

```
#define closesocket close
```

Definition at line 58 of file Private.h.

7.27.2.2 SD_BOTH

```
#define SD_BOTH SHUT_RDWR
```

Definition at line 57 of file Private.h.

7.27.2.3 ZESTETM1_CONNECTION_HANDLE_MAGIC

```
#define ZESTETM1_CONNECTION_HANDLE_MAGIC 0xdeadbed1
```

Definition at line 92 of file Private.h.

7.27.2.4 ZESTETM1_DEFAULT_TIMEOUT

```
#define ZESTETM1_DEFAULT_TIMEOUT 10000
```

Definition at line 107 of file Private.h.

7.27.2.5 ZESTETM1_ERROR

```
#define ZESTETM1_ERROR(
                f,
                x )
```

Value:

```
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
            ZestETM1_ErrorHandler(f, CardInfo, x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
```

Definition at line 65 of file Private.h.

```
00065 #define ZESTETM1_ERROR(f, x) \
00066     { \
00067         if (ZestETM1_ErrorHandler!=NULL) \
00068             ZestETM1_ErrorHandler(f, CardInfo, x, ZESTETM1_ERROR_STRING(x)); \
00069         return (x); \
00070     }
```

7.27.2.6 ZESTETM1_ERROR_CONN

```
#define ZESTETM1_ERROR_CONN(
                f,
                x )
```

Value:

```
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
        ZestETM1_ErrorHandler(f, (Conn!=NULL ? Conn->CardInfo : NULL), x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
```

Definition at line 77 of file Private.h.

```
00077 #define ZESTETM1_ERROR_CONN(f, x) \
00078     { \
00079         if (ZestETM1_ErrorHandler!=NULL) \
00080         ZestETM1_ErrorHandler(f, (Conn!=NULL ? Conn->CardInfo : NULL), x, ZESTETM1_ERROR_STRING(x)); \
00081         return (x); \
00082     }
```

7.27.2.7 ZESTETM1_ERROR_GENERAL

```
#define ZESTETM1_ERROR_GENERAL(
                f,
                x )
```

Value:

```
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
            ZestETM1_ErrorHandler(f, NULL, x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
```

Definition at line 71 of file Private.h.

```
00071 #define ZESTETM1_ERROR_GENERAL(f, x) \
00072     { \
00073         if (ZestETM1_ErrorHandler!=NULL) \
00074             ZestETM1_ErrorHandler(f, NULL, x, ZESTETM1_ERROR_STRING(x)); \
00075         return (x); \
00076     }
```

7.27.2.8 ZESTETM1_ERROR_STRING

```
#define ZESTETM1_ERROR_STRING(
                x )
```

Value:

```
    ZestETM1_ErrorStrings[(x)>=ZESTETM1_ERROR_BASE ? \
                            (x)-ZESTETM1_ERROR_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE)+(ZESTETM1_MAX_WARNING-ZESTETM1_WARNING_
      : \
                        ((x)>=ZESTETM1_WARNING_BASE ?
      (x)-ZESTETM1_WARNING_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE) : (x)-ZESTETM1_INFO_BASE)]
```

Definition at line 83 of file Private.h.

```
00083 #define ZESTETM1_ERROR_STRING(x) \
00084     ZestETM1_ErrorStrings[(x)>=ZESTETM1_ERROR_BASE ? \
00085
      (x)-ZESTETM1_ERROR_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE)+(ZESTETM1_MAX_WARNING-ZESTETM1_WARNING_BASE)
      : \
00086                             ((x)>=ZESTETM1_WARNING_BASE ?
      (x)-ZESTETM1_WARNING_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE) : (x)-ZESTETM1_INFO_BASE)]
```

7.27.2.9 ZESTETM1_RATE_10MHz

```
#define ZESTETM1_RATE_10MHz (2<<4)
```

Definition at line 112 of file Private.h.

7.27.2.10 ZESTETM1_RATE_20MHz

```
#define ZESTETM1_RATE_20MHz (1<<4)
```

Definition at line 111 of file Private.h.

7.27.2.11 ZESTETM1_RATE_40MHz

```
#define ZESTETM1_RATE_40MHz (0<<4)
```

Definition at line 110 of file Private.h.

7.27.2.12 ZESTETM1_REVERSE

```
#define ZESTETM1_REVERSE(
              x ) ((((x)&0xff)<<24) | (((x)&0xff00)<<8) | (((x)&0xff0000)>>8) | (((x)&0xff000000)>>24))
```

Definition at line 116 of file Private.h.

7.27.2.13 ZESTETM1_USER_DEVICE_ID

```
#define ZESTETM1_USER_DEVICE_ID (1)
```

Definition at line 113 of file Private.h.

7.27.3 Typedef Documentation

7.27.3.1 SOCKET

```
typedef int SOCKET
```

Definition at line 56 of file Private.h.

7.27.4 Function Documentation

7.27.4.1 ZestETM1_CloseConnection()

```
ZESTETM1_STATUS ZestETM1_CloseConnection (
              ZESTETM1_CONNECTION Connection )
```

Definition at line 225 of file Data.c.

```
00226 {
00227     ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT *)Connection;
00228 
00229     if (Conn==NULL)
00230     {
00231         return ZESTETM1_NULL_PARAMETER;
00232     }
00233     if (Conn->Magic!=ZESTETM1_CONNECTION_HANDLE_MAGIC)
00234     {
00235         return ZESTETM1_ILLEGAL_CONNECTION;
00236     }
00237 
00238     // Cleanup
00239     closesocket(Conn->Socket);
00240     Conn->Magic = 0;
00241     free(Conn);
00242 
00243     return ZESTETM1_SUCCESS;
00244 }
```

7.27.4.2 ZestETM1_EraseFlashSector()

```
ZESTETM1_STATUS ZestETM1_EraseFlashSector (
              ZESTETM1_CARD_INFO * CardInfo,
              ZESTETM1_CONNECTION Connection,
              uint32_t Address )
```

7.27.4.3 ZestETM1_OpenConnection()

```
ZESTETM1_STATUS ZestETM1_OpenConnection (
              ZESTETM1_CARD_INFO * CardInfo,
              ZESTETM1_CONNECTION_TYPE Type,
              uint16_t Port,
              uint16_t LocalPort,
              ZESTETM1_CONNECTION * Connection )
```

Definition at line 101 of file Data.c.

```
00106 {
00107     ZESTETM1_CONNECTION_STRUCT *NewStruct;
00108     SOCKET Socket = -1;
00109     char AddrBuffer[32];
00110     char PortBuffer[32];
00111 
00112     if (Connection==NULL || CardInfo==NULL)
00113     {
00114         return ZESTETM1_NULL_PARAMETER;
00115     }
00116 
00117     // Allocate data structure
00118     NewStruct = malloc(sizeof(ZESTETM1_CONNECTION_STRUCT));
00119     if (NewStruct==NULL)
00120     {
00121         return ZESTETM1_OUT_OF_MEMORY;
00122     }
```

```
    // Build target addresses
    sprintf(AddrBuffer, "%d.%d.%d.%d", CardInfo->IPAddr[0], CardInfo->IPAddr[1],
                    CardInfo->IPAddr[2], CardInfo->IPAddr[3]);
    sprintf(PortBuffer, "%d", Port);

    if (Type==ZESTETM1_TYPE_UDP)
    {
        // Open UDP connection
        struct sockaddr_in SourceIP;
        int SourceLen = (int)sizeof(struct sockaddr_in);
        Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (Socket<0)
            return ZESTETM1_SOCKET_ERROR;
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = htons(LocalPort);
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);

        NewStruct->Target.sin_family = AF_INET;
        NewStruct->Target.sin_addr.s_addr = inet_addr(AddrBuffer);
        NewStruct->Target.sin_port = htons(atoi(PortBuffer));
    }
    else if (Type==ZESTETM1_TYPE_TCP)
    {
        // Open TCP connection
        struct addrinfo *AddrResult = NULL,
                        *Ptr = NULL,
                        Hints;
        int Result;
        struct sockaddr_in SourceIP;
        int SourceLen = (int)sizeof(struct sockaddr_in);

        memset(&Hints, 0, sizeof(Hints));
        Hints.ai_family = AF_UNSPEC;
        Hints.ai_socktype = SOCK_STREAM;
        Hints.ai_protocol = IPPROTO_TCP;

        // Resolve the server address and port
        Result = getaddrinfo(AddrBuffer, PortBuffer, &Hints, &AddrResult);
        if (Result!=0)
        {
            free(NewStruct);
            return ZESTETM1_SOCKET_ERROR;
        }

        // Attempt to connect to an address until one succeeds
        for (Ptr=AddrResult; Ptr!=NULL; Ptr=Ptr->ai_next)
        {
            // Create a SOCKET for connecting to server
            Socket = socket(Ptr->ai_family, Ptr->ai_socktype,
                            Ptr->ai_protocol);
            if (Socket<0)
            {
                freeaddrinfo(AddrResult);
                free(NewStruct);
                return ZESTETM1_SOCKET_ERROR;
            }

            // Connect to ZestETM1
            Result = connect(Socket, Ptr->ai_addr, (int)Ptr->ai_addrlen);
            if (Result<0)
            {
                closesocket(Socket);
                Socket = -1;
                continue;
            }
            break;
        }

        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        bind(Socket, (const struct sockaddr *)&SourceIP, SourceLen);
        freeaddrinfo(AddrResult);
    }
    else
    {
        free(NewStruct);
        return ZESTETM1_INVALID_CONNECTION_TYPE;
    }

    if (Socket==-1)
    {
        free(NewStruct);
        return ZESTETM1_SOCKET_ERROR;
    }

    NewStruct->Magic = ZESTETM1_CONNECTION_HANDLE_MAGIC;
    NewStruct->Type = Type;
    NewStruct->Port = Port;
    NewStruct->LocalPort = LocalPort;
    NewStruct->Socket = Socket;
    NewStruct->CardInfo = CardInfo;
    *Connection = NewStruct;

    return ZESTETM1_SUCCESS;
}
```

7.27.4.4 ZestETM1_SendCommand()

```
ZESTETM1_STATUS ZestETM1_SendCommand (
        ZESTETM1_CARD_INFO * CardInfo,
        ZESTETM1_CONNECTION Connection,
        void * WriteBuffer,
        uint32_t WriteLen,
        void * ReadBuffer,
        uint32_t ReadLen,
        int WaitForAck )
```

Definition at line 453 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    unsigned long Written;
    unsigned long Received;

    // Send/receive data
    Result = ZestETM1_WriteData(Connection, WriteBuffer, WriteLen, &Written,
                               CardInfo->Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        return Result;
    }
    if (Written!=WriteLen)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }
    *((uint8_t *)ReadBuffer) = 0;
    if (WaitForAck==1)
    {
        Result = ZestETM1_ReadData(Connection, ReadBuffer, ReadLen, &Received,
                                  CardInfo->Timeout);
        if (Result!=ZESTETM1_SUCCESS)
        {
            return Result;
        }
        if (Received!=ReadLen)
        {
            return ZESTETM1_INTERNAL_ERROR;
        }
    }

    return ZESTETM1_SUCCESS;
}
```

7.27.4.5 ZestETM1_SPIReadWrite()

```
ZESTETM1_STATUS ZestETM1_SPIReadWrite (
        ZESTETM1_CARD_INFO * CardInfo,
        ZESTETM1_CONNECTION Connection,
        int Device,
        int WordLen,
        uint32_t * WriteData,
        uint32_t * ReadData,
        uint32_t Length,
        int ReleaseCS,
        int WaitForAck )
```

Definition at line 495 of file Data.c.

```
{
    uint8_t Buffer[65536];
    uint32_t *BufPtr;
    uint32_t i;
    ZESTETM1_STATUS Result;

    // Build command
    Buffer[0] = ZESTETM1_COMMAND_SPI;// Command byte
    Buffer[1] = Device;             // SPI device
    Buffer[2] = WordLen;            // Word length
    Buffer[3] = ReleaseCS;          // Release CS on completion
    BufPtr = (uint32_t *)(Buffer+4);

    BufPtr[0] = WriteData==NULL ? 0 : ZESTETM1_REVERSE(Length);    // Num words
    BufPtr[1] = ReadData==NULL ? 0 : ZESTETM1_REVERSE(Length);
    if (WriteData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            BufPtr[2+i] = ZESTETM1_REVERSE(WriteData[i]);
        }
    }
    else
        memset(BufPtr+2, 0, Length*4);

    // Send command
    Result = ZestETM1_SendCommand(CardInfo, Connection, Buffer,
                                  WriteData==NULL ? 12 : 12+Length*4,
                                  Buffer, ReadData==NULL ? 4 : 4+Length*4,
                                  WaitForAck);
    if (WaitForAck==0)
        return Result;

    // Extract results
    if (Buffer[0]!=ZESTETM1_COMMAND_SPI || Buffer[1]!=0)
    {
        return ZESTETM1_INTERNAL_ERROR;
    }

    BufPtr = (uint32_t *)(Buffer+4);
    if (ReadData!=NULL)
    {
        for (i=0; i<Length; i++)
        {
            ReadData[i] = ZESTETM1_REVERSE(BufPtr[i]);
        }
    }

    return ZESTETM1_SUCCESS;
}
```

7.27.4.6 ZestETM1_WriteFlash()

```
ZESTETM1_STATUS ZestETM1_WriteFlash (
            ZESTETM1_CARD_INFO * CardInfo,
            uint32_t Address,
            void * Buffer,
            uint32_t Length )
```

7.27.5 Variable Documentation

7.27.5.1 ZestETM1_ErrorHandler

```
ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler  [extern]
```

Definition at line 67 of file Error.c.

7.27.5.2 ZestETM1_ErrorStrings

```
char* ZestETM1_ErrorStrings[]  [extern]
```

Definition at line 52 of file Error.c.

```
{
    "Success (no error)",
    "Error communicating with socket",
    "An unspecified internal error occurred",
    "Status code is out of range",
    "NULL was used illegally as one of the parameter values",
    "Not enough memory to complete the requested operation",
    "The requested connection type is invalid",
    "The requested connection is invalid",
    "The connection was closed unexpectedly",
    "Operation timed out",
    "One of the parameters has an illegal value",

};
```

7.28 Private.h

Go to the documentation of this file.

```
// File:      Private.h
//
// Purpose:
//    ZestETM1 Host Library
//    Internal header file
//
// Version: 1.00
// Date: 11/10/12

// Copyright (C) 2012 Orange Tree Technologies Ltd. All rights reserved.
// Orange Tree Technologies grants the purchaser of a ZestETM1 the right to use and
// modify this source code in any form in designs that target the ZestETM1.
// Orange Tree Technologies prohibits the use of this source code or any modification of
// it in any form in designs that target any other hardware unless the purchaser of the
// ZestETM1 has purchased the appropriate licence from Orange Tree Technologies.
// Contact Orange Tree Technologies if you want to purchase such a licence.

//*****************************************************************************************
//**
//**  Disclaimer: LIMITED WARRANTY AND DISCLAIMER. These designs are
//**              provided to you "as is". Orange Tree Technologies and its licensors
//**              make and you receive no warranties or conditions, express, implied,
//**              statutory or otherwise, and Orange Tree Technologies specifically
//**              disclaims any implied warranties of merchantability, non-infringement,
//**              or fitness for a particular purpose. Orange Tree Technologies does not
//**              warrant that the functions contained in these designs will meet your
//**              requirements, or that the operation of these designs will be
//**              uninterrupted or error free, or that defects in the Designs will be
//**              corrected. Furthermore, Orange Tree Technologies does not warrant or
//**              make any representations regarding use or the results of the use of the
//**              designs in terms of correctness, accuracy, reliability, or otherwise.
//**
//**              LIMITATION OF LIABILITY. In no event will Orange Tree Technologies
//**              or its licensors be liable for any loss of data, lost profits, cost or
```

```
//**              procurement of substitute goods or services, or for any special,
//**              incidental, consequential, or indirect damages arising from the use or
//**              operation of the designs or accompanying documentation, however caused
//**              and on any theory of liability. This limitation will apply even if
//**              Orange Tree Technologies has been advised of the possibility of such
//**              damage. This limitation shall apply notwithstanding the failure of the
//**              essential purpose of any limited remedies herein.
//**
//****************************************************************************************

#include <stdint.h>

#if defined(MSVC) || defined(WINGCC)
#include "Winsock2.h"
#include "Ws2tcpip.h"
#else
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
typedef int SOCKET;
#define SD_BOTH SHUT_RDWR
#define closesocket close
#endif

/**************
* Error macro *
**************/
extern ZESTETM1_ERROR_FUNC ZestETM1_ErrorHandler;
#define ZESTETM1_ERROR(f, x) \
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
            ZestETM1_ErrorHandler(f, CardInfo, x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
#define ZESTETM1_ERROR_GENERAL(f, x) \
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
            ZestETM1_ErrorHandler(f, NULL, x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
#define ZESTETM1_ERROR_CONN(f, x) \
    { \
        if (ZestETM1_ErrorHandler!=NULL) \
        ZestETM1_ErrorHandler(f, (Conn!=NULL ? Conn->CardInfo : NULL), x, ZESTETM1_ERROR_STRING(x)); \
        return (x); \
    }
#define ZESTETM1_ERROR_STRING(x) \
    ZestETM1_ErrorStrings[(x)>=ZESTETM1_ERROR_BASE ? \
(x)-ZESTETM1_ERROR_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE)+(ZESTETM1_MAX_WARNING-ZESTETM1_WARNING_BASE) : \
                        ((x)>=ZESTETM1_WARNING_BASE ? (x)-ZESTETM1_WARNING_BASE+(ZESTETM1_MAX_INFO-ZESTETM1_INFO_BASE) : (x)-ZESTETM1_INFO_BASE)]
extern char *ZestETM1_ErrorStrings[];

/*******************************************
* Network connection descriptor structure *
*******************************************/
#define ZESTETM1_CONNECTION_HANDLE_MAGIC 0xdeadbed1
typedef struct
{
    uint32_t Magic;
    ZESTETM1_CARD_INFO *CardInfo;
    ZESTETM1_CONNECTION_TYPE Type;
    struct sockaddr_in Target;
    uint16_t Port;
    uint16_t LocalPort;
    SOCKET Socket;
} ZESTETM1_CONNECTION_STRUCT;

/************
* Constants *
************/
#define ZESTETM1_DEFAULT_TIMEOUT 10000

// SPI Device ID and clock
#define ZESTETM1_RATE_40MHz (0«4)
#define ZESTETM1_RATE_20MHz (1«4)
#define ZESTETM1_RATE_10MHz (2«4)
```

```
#define ZESTETM1_USER_DEVICE_ID      (1)

// Reverse bytes in 32 bit word
#define ZESTETM1_REVERSE(x) ((((x)&0xff)«24) | (((x)&0xff00)«8) | (((x)&0xff0000)»8) |
    (((x)&0xff000000)»24))

/******************
* Local functions *
******************/
ZESTETM1_STATUS ZestETM1_OpenConnection(ZESTETM1_CARD_INFO *CardInfo,
                                        ZESTETM1_CONNECTION_TYPE Type,
                                        uint16_t Port,
                                        uint16_t LocalPort,
                                        ZESTETM1_CONNECTION *Connection);
ZESTETM1_STATUS ZestETM1_CloseConnection(ZESTETM1_CONNECTION Connection);
ZESTETM1_STATUS ZestETM1_SendCommand(ZESTETM1_CARD_INFO *CardInfo,
                                     ZESTETM1_CONNECTION Connection,
                                     void *WriteBuffer, uint32_t WriteLen,
                                     void *ReadBuffer, uint32_t ReadLen,
                                     int WaitForAck);
ZESTETM1_STATUS ZestETM1_SPIReadWrite(ZESTETM1_CARD_INFO *CardInfo,
                                      ZESTETM1_CONNECTION Connection,
                                      int Device,
                                      int WordLen, uint32_t *WriteData,
                                      uint32_t *ReadData, uint32_t Length,
                                      int ReleaseCS, int WaitForAck);
ZESTETM1_STATUS ZestETM1_WriteFlash(ZESTETM1_CARD_INFO *CardInfo,
                                    uint32_t Address,
                                    void *Buffer,
                                    uint32_t Length);
ZESTETM1_STATUS ZestETM1_EraseFlashSector(ZESTETM1_CARD_INFO *CardInfo,
                                          ZESTETM1_CONNECTION Connection,
                                          uint32_t Address);


```

7.29 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/↩ UPnP.c File Reference

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include "ZestETM1.h"
#include "Private.h"
#include <sys/types.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
```

Classes

- struct ZESTETM1_GET_SETTINGS_CMD
- struct ZESTETM1_GET_SETTINGS_RESPONSE

Macros

- #define _CRT_SECURE_NO_WARNINGS
- #define _strnicmp strncasecmp
- #define ZESTETM1_UPNP_ADDR "239.255.255.250"
- #define ZESTETM1_UPNP_PORT 1900
- #define ZESTETM1_COMMAND_GET_SETTINGS 0xf0

Functions

- static int ZestETM1_HTTPGet (char ∗IPAddr, char ∗Port, char ∗FileName, void ∗Buffer, int BufferLength, int Wait)
- static ZESTETM1_STATUS ZestETM1_ReadSettings (ZESTETM1_CARD_INFO ∗CardInfo)
- static void ZestETM1_GetCardInfo (char ∗Location, uint32_t ∗NumCards, ZESTETM1_CARD_INFO ∗∗CardInfo, int Wait)
- static int ZestETM1_JoinGroup (SOCKET sd, uint32_t grpaddr, uint32_t iaddr)
- static int ZestETM1_LeaveGroup (SOCKET sd, uint32_t grpaddr, uint32_t iaddr)
- static ZESTETM1_STATUS ZestETM1_GetAllAdapters (uint32_t ∗NumAdapters, struct sockaddr_in ∗∗Adapters)
- ZESTETM1_STATUS ZestETM1CountCards (unsigned long ∗NumCards, ZESTETM1_CARD_INFO ∗∗CardInfo, unsigned long Wait)
- ZESTETM1_STATUS ZestETM1FreeCards (ZESTETM1_CARD_INFO ∗CardInfo)
- ZESTETM1_STATUS ZestETM1GetCardInfo (ZESTETM1_CARD_INFO ∗CardInfo)

Variables

- static char ∗ ZestETM1_SearchReq

7.29.1 Class Documentation

7.29.1.1 struct ZESTETM1_GET_SETTINGS_CMD

Definition at line 82 of file UPnP.c.

Class Members

uint8_t	Command	
uint8_t	Dummy[3]	

7.29.1.2 struct ZESTETM1_GET_SETTINGS_RESPONSE

Definition at line 87 of file UPnP.c.

Class Members

uint8_t	Command	
uint16_t	ControlPort	
uint8_t	Dummy1[2]	
uint8_t	Dummy2[2]	
uint32_t	Gateway	
uint16_t	HardwareVersion	
uint16_t	HTTPPort	
uint32_t	IPAddr	
uint8_t	MACAddr[6]	
uint32_t	SerialNumber	

Class Members

uint16_t	SoftwareVersion	
uint8_t	Status	
uint32_t	SubNet	

7.29.2 Macro Definition Documentation

7.29.2.1 _CRT_SECURE_NO_WARNINGS

```
#define _CRT_SECURE_NO_WARNINGS
```

Definition at line 45 of file UPnP.c.

7.29.2.2 _strnicmp

```
#define _strnicmp strncasecmp
```

Definition at line 61 of file UPnP.c.

7.29.2.3 ZESTETM1_COMMAND_GET_SETTINGS

```
#define ZESTETM1_COMMAND_GET_SETTINGS 0xf0
```

Definition at line 103 of file UPnP.c.

7.29.2.4 ZESTETM1_UPNP_ADDR

```
#define ZESTETM1_UPNP_ADDR "239.255.255.250"
```

Definition at line 78 of file UPnP.c.

7.29.2.5 ZESTETM1_UPNP_PORT

```
#define ZESTETM1_UPNP_PORT 1900
```

Definition at line 79 of file UPnP.c.

7.29.3 Function Documentation

7.29.3.1 ZestETM1_GetAllAdapters()

```
static ZESTETM1_STATUS ZestETM1_GetAllAdapters (
            uint32_t * NumAdapters,
            struct sockaddr_in ** Adapters )  [static]
```

Definition at line 431 of file UPnP.c.

```
{
#if defined(MSVC) || defined(WINGCC)
    SOCKET Socket;
    SOCKET_ADDRESS_LIST *AddressListPtr;
    DWORD BytesRequired;
    int i;
    int Count = 0;

    *NumAdapters = 0;
    *Adapters = NULL;

    Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (Socket<0)
        return ZESTETM1_INTERNAL_ERROR;

    WSAIoctl(Socket, SIO_ADDRESS_LIST_QUERY, NULL, 0,
                NULL, 0, (LPDWORD)&BytesRequired, NULL, NULL);
    AddressListPtr = (SOCKET_ADDRESS_LIST *)malloc(BytesRequired);
    if (AddressListPtr==NULL)
    {
        closesocket(Socket);
        return ZESTETM1_OUT_OF_MEMORY;
    }
    if (WSAIoctl(Socket, SIO_ADDRESS_LIST_QUERY, NULL, 0,
                    AddressListPtr, BytesRequired, &BytesRequired, NULL, NULL)==SOCKET_ERROR)
    {
        free(AddressListPtr);
        closesocket(Socket);
        return ZESTETM1_INTERNAL_ERROR;
    }

    for (i=0; i<AddressListPtr->iAddressCount; i++)
    {
        if (AddressListPtr->Address[i].iSockaddrLength==sizeof(struct sockaddr_in))
        {
            Count++;
            (*Adapters) = (struct sockaddr_in *)realloc(*Adapters, Count*sizeof(struct sockaddr_in));
            if ((*Adapters)==NULL)
            {
                free(AddressListPtr);
                closesocket(Socket);
                return ZESTETM1_OUT_OF_MEMORY;
            }
            memcpy(&(*Adapters)[Count-1], AddressListPtr->Address[i].lpSockaddr, sizeof(struct sockaddr_in));
        }
    }
    *NumAdapters = Count;
    closesocket(Socket);
    free(AddressListPtr);

    return ZESTETM1_SUCCESS;
#else
    struct ifaddrs *Interfaces;
    struct ifaddrs *Ptr;
    uint32_t Count = 0;

    if (getifaddrs(&Interfaces)!=0)
        return ZESTETM1_INTERNAL_ERROR;

    *NumAdapters = 0;
    *Adapters = NULL;

    Ptr = Interfaces;
    while (Ptr!=NULL)
    {
        if (Ptr->ifa_addr!=NULL)
        {
            Count++;
            (*Adapters) = (struct sockaddr_in *)realloc(*Adapters, Count*sizeof(struct sockaddr_in));
            if ((*Adapters)==NULL)
            {
                freeifaddrs(Interfaces);
                return ZESTETM1_OUT_OF_MEMORY;
            }
            memcpy(&(*Adapters)[Count-1], Ptr->ifa_addr, sizeof(struct sockaddr_in));
        }
        Ptr = Ptr->ifa_next;
    }
    *NumAdapters = Count;
    freeifaddrs(Interfaces);

    return ZESTETM1_SUCCESS;
#endif
}
```

7.29.3.2 ZestETM1_GetCardInfo()

```
static void ZestETM1_GetCardInfo (
            char * Location,
            uint32_t * NumCards,
            ZESTETM1_CARD_INFO ** CardInfo,
            int Wait )  [static]
```

Definition at line 279 of file UPnP.c.

```
{
    static char *Prefix = "http://";
    char Buffer[65536];
    char *IPAddr = NULL;
    char *FileName = NULL;
    char *Port = "80";
    int i;

    // Extract address and port
    for (i=0; Location[i]!=0 && Location[i]!='\r' && Location[i]!='\n'; i++)
    {
        if (IPAddr==NULL && Prefix[i]==0) IPAddr = Location+i;
        if (IPAddr==NULL && Location[i]!=Prefix[i])
            return;
        if (IPAddr!=NULL)
        {
            if (Location[i]=='/' || Location[i]==0 ||
                Location[i]=='\r' || Location[i]=='\n')
            {
                Location[i]=0;
                i++;
                break;
            }
            if (Location[i]==':')
            {
                int j=i;
                while (Location[j]!=0 && Location[j]!='\r' &&
                       Location[j]!='\n' && Location[j]!='/') j++;
                Location[i]=0;
                Location[j]=0;
                Port = Location+i+1;
                i = j+1;
                break;
            }
        }
    }
    if (IPAddr==NULL || Location[i]==0 || Location[i]=='\r' || Location[i]=='\n')
        return;

    // Extract XML filename
    FileName = Location+i;
    while (Location[i]!=0 && Location[i]!='\r' && Location[i]!='\n') i++;
    Location[i] = 0;

    // Get XML file
    memset(Buffer, 0, sizeof(Buffer));
    if (ZestETM1_HTTPGet(IPAddr, Port, FileName,
                         Buffer, sizeof(Buffer), Wait)>0)
    {
        uint8_t IPAddr[4];
        uint32_t i;
        uint16_t Port;

        // Parse XML for service description URL
        char *ControlURL = strstr(Buffer, "<controlURL>");
        if (ControlURL==NULL)
            return;

        ControlURL += 12;
        for (i=0; i<4; i++)
        {
            char *EndPtr;
            IPAddr[i] = (uint8_t)strtoul(ControlURL, &EndPtr, 10);
            if ((i!=3 && *EndPtr!='.') || (i==3 && *EndPtr!=':'))
                break;
            ControlURL = EndPtr+1;
        }
        if (i!=4)
            return;

        Port = atoi(ControlURL);

        // Make sure only unique devices are added to the list!
        if (*CardInfo!=NULL)
        {
            for (i=0; i<*NumCards; i++)
            {
                if ((*CardInfo)[i].ControlPort==Port &&
                    memcmp((*CardInfo)[i].IPAddr, IPAddr, sizeof(IPAddr))==0)
                {
                    break;
                }
            }
        }
        if (*CardInfo==NULL || i==*NumCards)
        {
            uint32_t Index = (*NumCards);
            ZESTETM1_CARD_INFO *NewBuffer;

            // Allocate space for new card info structure
            if ((*CardInfo)==NULL)
                NewBuffer = malloc(sizeof(ZESTETM1_CARD_INFO)*(Index+1));
            else
                NewBuffer = realloc(*CardInfo, sizeof(ZESTETM1_CARD_INFO)*(Index+1));

            // Get new card settings
            NewBuffer[Index].ControlPort = Port;
            memcpy(NewBuffer[Index].IPAddr, IPAddr, sizeof(IPAddr));
            NewBuffer[Index].Timeout = Wait;
            if (ZestETM1_ReadSettings(&(NewBuffer[Index]))!=ZESTETM1_SUCCESS)
            {
                if (*CardInfo==NULL)
                    free(NewBuffer);
                else
                {
                    (*CardInfo) = NewBuffer;
                    memset(NewBuffer[Index].IPAddr, 0, sizeof(NewBuffer[Index].IPAddr));
                    NewBuffer[Index].ControlPort = 0;
                }
            }
            else
            {
                NewBuffer[Index].Timeout = ZESTETM1_DEFAULT_TIMEOUT;
                (*CardInfo) = NewBuffer;
                (*NumCards)++;
            }
        }
    }
}
```

7.29.3.3 ZestETM1_HTTPGet()

```
static int ZestETM1_HTTPGet (
            char * IPAddr,
            char * Port,
            char * FileName,
            void * Buffer,
            int BufferLength,
            int Wait )  [static]
```

Definition at line 108 of file UPnP.c.

```
{
    struct addrinfo *Addr = NULL,
                    *Ptr = NULL,
                    Hints;
    int Result;
    SOCKET Socket;
    int Offset = 0;
    char Req[1024];
    char *HdrEnd;
    fd_set ReadFDS;
    struct timeval Timeout;

    Timeout.tv_sec = Wait/1000;
    Timeout.tv_usec = (Wait%1000)*1000;

    // Attempt to connect to the address
    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family = AF_UNSPEC;
    Hints.ai_socktype = SOCK_STREAM;
    Hints.ai_protocol = IPPROTO_TCP;

    // Resolve the server address and port
    Result = getaddrinfo(IPAddr, Port, &Hints, &Addr);
    if (Result!=0)
        return -1;

    for (Ptr=Addr; Ptr!=NULL; Ptr=Ptr->ai_next)
    {
        // Create a SOCKET for connecting to server
        Socket = socket(Ptr->ai_family, Ptr->ai_socktype,
                        Ptr->ai_protocol);
        if (Socket<0)
        {
            freeaddrinfo(Addr);
            return -1;
        }

        // Connect to ZestETM1
        Result = connect(Socket, Ptr->ai_addr, (int)Ptr->ai_addrlen);
        if (Result<0)
        {
            closesocket(Socket);
            Socket = -1;
            continue;
        }
        break;
    }
    freeaddrinfo(Addr);

    // Send GET request
    sprintf(Req, "GET /%s HTTP/1.1\r\nHOST: %s:%s\r\nContent-length: 0\r\n\r\n", FileName, IPAddr, Port);
    Result = send(Socket, Req, (int)strlen(Req), 0);
    if (Result!=strlen(Req))
    {
        closesocket(Socket);
        return -1;
    }

    // Get response
    ((char*)Buffer)[0] = 0;
    do
    {
        FD_ZERO(&ReadFDS);
        FD_SET(Socket, &ReadFDS);
00175         Result = select((int)Socket+1, &ReadFDS, NULL, NULL, &Timeout);
00176         if (Result<0)
00177         {
00178             closesocket(Socket);
00179             return -1;
00180         }
00181         if (!FD_ISSET(Socket, &ReadFDS)) break;
00182         Result = recv(Socket, (char *)Buffer+Offset, BufferLength-Offset, 0);
00183         if (Result<0)
00184         {
00185             closesocket(Socket);
00186             return -1;
00187         }
00188         Offset+=Result;
00189         if (Offset==BufferLength)
00190         {
00191             break;
00192         }
00193     } while (Result!=0);
00194 
00195     // Check status response
00196     if (_strnicmp("HTTP/1.1 200 OK", Buffer, 15)!=0)
00197     {
00198         closesocket(Socket);
00199         return -1;
00200     }
00201 
00202     // Remove HTTP header
00203     HdrEnd = strstr(Buffer, "\r\n\r\n");
00204     if (HdrEnd==NULL)
00205     {
00206         Offset = 0;
00207     }
00208     else
00209     {
00210         Offset -= (int)(HdrEnd+4-(char*)Buffer);
00211         memcpy(Buffer, HdrEnd+4, Offset);
00212     }
00213 
00214     closesocket(Socket);
00215     return Offset;
00216 }
```

7.29.3.4 ZestETM1_JoinGroup()

```
static int ZestETM1_JoinGroup (
              SOCKET sd,
              uint32_t grpaddr,
              uint32_t iaddr )  [static]
```

Definition at line 407 of file UPnP.c.

```
00409 {
00410     struct ip_mreq imr;
00411 
00412     imr.imr_multiaddr.s_addr  = grpaddr;
00413     imr.imr_interface.s_addr  = iaddr;
00414     return setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
00415                       (const char *)&imr, sizeof(imr));
00416 }
```

7.29.3.5 ZestETM1_LeaveGroup()

```
static int ZestETM1_LeaveGroup (
              SOCKET sd,
              uint32_t grpaddr,
              uint32_t iaddr )  [static]
```

Definition at line 417 of file UPnP.c.

```
{
    struct ip_mreq imr;

    imr.imr_multiaddr.s_addr  = grpaddr;
    imr.imr_interface.s_addr  = iaddr;
    return setsockopt(sd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                      (const char *)&imr, sizeof(imr));
}
```

7.29.3.6 ZestETM1_ReadSettings()

```
static ZESTETM1_STATUS ZestETM1_ReadSettings (
            ZESTETM1_CARD_INFO * CardInfo )  [static]
```

Definition at line 221 of file UPnP.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_GET_SETTINGS_CMD Cmd;
    ZESTETM1_GET_SETTINGS_RESPONSE Response;

    if (CardInfo==NULL)
    {
        return ZESTETM1_NULL_PARAMETER;
    }

    // Open control connection
    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP,
                                     CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        return Result;
    }

    // Get the settings from the device
    Cmd.Command = ZESTETM1_COMMAND_GET_SETTINGS;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        return Result;
    }

    // Result values (including endian conversion)
    CardInfo->FirmwareVersion = ((Response.SoftwareVersion»8)&0xff)|((Response.SoftwareVersion&0xff)«8);
    CardInfo->HardwareVersion = ((Response.HardwareVersion»8)&0xff)|((Response.HardwareVersion&0xff)«8);
    CardInfo->Gateway[0] = (uint8_t)((Response.Gateway»0)&0xff);
    CardInfo->Gateway[1] = (uint8_t)((Response.Gateway»8)&0xff);
    CardInfo->Gateway[2] = (uint8_t)((Response.Gateway»16)&0xff);
    CardInfo->Gateway[3] = (uint8_t)((Response.Gateway»24)&0xff);
    CardInfo->SubNet[0] = (uint8_t)((Response.SubNet»0)&0xff);
    CardInfo->SubNet[1] = (uint8_t)((Response.SubNet»8)&0xff);
    CardInfo->SubNet[2] = (uint8_t)((Response.SubNet»16)&0xff);
    CardInfo->SubNet[3] = (uint8_t)((Response.SubNet»24)&0xff);
    memcpy(CardInfo->MACAddr, Response.MACAddr, 6);
    CardInfo->ControlPort = ((Response.ControlPort»8)&0xff)|((Response.ControlPort&0xff)«8);
    CardInfo->HTTPPort = ((Response.HTTPPort»8)&0xff)|((Response.HTTPPort&0xff)«8);
    CardInfo->SerialNumber = ZESTETM1_REVERSE(Response.SerialNumber);

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
        return Result;

    return ZESTETM1_SUCCESS;
}
```

7.29.3.7 ZestETM1CountCards()

```
ZESTETM1_STATUS ZestETM1CountCards (
            unsigned long * NumCards,
            ZESTETM1_CARD_INFO ** CardInfo,
            unsigned long Wait )
```

Definition at line 521 of file UPnP.c.

```
{
    SOCKET Socket;
    struct sockaddr_in DestIP;
    struct sockaddr_in SourceIP;
    int SourceIPLength;
    int Flag = 1;
    int Result;
    char Req[1024];
    char Response[1024];
    ZESTETM1_CARD_INFO *Cards = NULL;
    uint32_t CardCount = 0;
    int i;
    struct timeval Timeout;
    fd_set ReadFDS;
    uint32_t Interface;
    uint32_t NumAdapters;
    struct sockaddr_in *Adapters;
    ZESTETM1_STATUS Status;

    *NumCards = 0;
    *CardInfo = NULL;

    // Get a list of all adapters
    Status = ZestETM1_GetAllAdapters(&NumAdapters, &Adapters);
    if (Status!=ZESTETM1_SUCCESS)
        ZESTETM1_ERROR_GENERAL("ZestETM1CountCards", Status);
    if (NumAdapters==0)
    {
        *NumCards = 0;
        return ZESTETM1_SUCCESS;
    }

    // Send queries on all interfaces
    for (Interface=0; Interface<NumAdapters; Interface++)
    {
        if (Adapters[Interface].sin_family!=AF_INET)
            continue;

        // Open socket for search requests
        Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (Socket<0)
        {
            continue;
        }

        // Set reuse port to on to allow multiple binds per host
        if (setsockopt(Socket, SOL_SOCKET, SO_REUSEADDR, (char *)&Flag,
                       sizeof(Flag))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Bind to port for receiving responses
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        Result = bind(Socket, (const struct sockaddr *)(&Adapters[Interface]), sizeof(struct
sockaddr_in));
        if (Result<0)
        {
            closesocket(Socket);
            continue;
        }

        // Join multicast group
        if (ZestETM1_JoinGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                               htonl(INADDR_ANY))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Send M-SEARCH request
        // Send more than once as UDP is unreliable
        DestIP.sin_family = AF_INET;
        DestIP.sin_addr.s_addr = inet_addr(ZESTETM1_UPNP_ADDR);
        DestIP.sin_port = htons(1900);
        sprintf(Req, ZestETM1_SearchReq, (Wait+999)/1000);
        for (i=0; i<3; i++)
        {
            Result = sendto(Socket, ZestETM1_SearchReq,
                            (int)strlen(ZestETM1_SearchReq),
                            0, (struct sockaddr *)&DestIP, sizeof(DestIP));
            if (Result!=strlen(ZestETM1_SearchReq))
            {
                ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                                   htonl(INADDR_ANY));
                closesocket(Socket);
                continue;
            }
        }

        // Read responses
        Timeout.tv_sec = Wait/1000;
        Timeout.tv_usec = (Wait%1000)*1000;
        do
        {
            FD_ZERO(&ReadFDS);
            FD_SET(Socket, &ReadFDS);
            Result = select((int)Socket+1, &ReadFDS, NULL, NULL, &Timeout);
            if (Result<0)
            {
                break;
            }
            if (!FD_ISSET(Socket, &ReadFDS)) break;

            SourceIPLength = sizeof(SourceIP);
            Result = recvfrom(Socket, Response, sizeof(Response),
                              0, (struct sockaddr *)&SourceIP, &SourceIPLength);
            if (Result<0)
            {
                // Error!
                break;
            }
            else if (Result==0)
            {
                // Clean shut down
                break;
            }
            else
            {
                // Parse results
                if (_strnicmp("NOTIFY", Response, 6)==0 ||
                    _strnicmp("HTTP/1.1 200 OK", Response, 15)==0)
                {
                    // Check its a GigExpedite and request XML description
                    char *Server = strstr(Response, "GigExpedite2");
                    char *Location = strstr(Response, "LOCATION");
                    if (Server!=NULL && Location!=NULL)
                    {
                        char *c;
                        for (c=Location+10; *c!=0 && *c!='\n' && *c!='\r'; c++);
                        *c = 0;
                        ZestETM1_GetCardInfo(Location+10, &CardCount, &Cards, Wait);
                    }
                }
            }
        } while(1);

        // Leave multicast group
        ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                           htonl(INADDR_ANY));

        // Close socket
        closesocket(Socket);
    }

    *NumCards = CardCount;
    *CardInfo = Cards;
    free(Adapters);

    return ZESTETM1_SUCCESS;
}
```

7.29.3.8 ZestETM1FreeCards()

```
ZESTETM1_STATUS ZestETM1FreeCards (
                ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 680 of file UPnP.c.

```
{
    if (CardInfo!=NULL)
        free(CardInfo);

    return ZESTETM1_SUCCESS;
}
```

7.29.3.9 ZestETM1GetCardInfo()

```
ZESTETM1_STATUS ZestETM1GetCardInfo (
                ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 691 of file UPnP.c.

```
{
    ZESTETM1_STATUS Status;

    Status = ZestETM1_ReadSettings(CardInfo);
    if (Status!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1GetCardInfo", Status);
    }

    return ZESTETM1_SUCCESS;
}
```

7.29.4 Variable Documentation

7.29.4.1 ZestETM1_SearchReq

```
char* ZestETM1_SearchReq  [static]
```

Initial value:

```
=
"M-SEARCH * HTTP/1.1\r\n"
"ST: upnp:rootdevice\r\n"
"MX: %d\r\n"
"MAN: \"ssdp:discover\"\r\n"
"HOST: 239.255.255.250:1900\r\n"
```

Definition at line 69 of file UPnP.c.

7.30 UPnP.c

Go to the documentation of this file.

```
// File:      UPnP.c
//
// Purpose:
//    ZestETM1 Host Library
//    UPnP board discovery functions
//
// Version: 1.00
// Date: 11/10/12

// Copyright (C) 2012 Orange Tree Technologies Ltd. All rights reserved.
// Orange Tree Technologies grants the purchaser of a ZestETM1 the right to use and
// modify this source code in any form in designs that target the ZestETM1.
// Orange Tree Technologies prohibits the use of this source code or any modification of
// it in any form in designs that target any other hardware unless the purchaser of the
// ZestETM1 has purchased the appropriate licence from Orange Tree Technologies.
// Contact Orange Tree Technologies if you want to purchase such a licence.

//*****************************************************************************************
//**
//**  Disclaimer: LIMITED WARRANTY AND DISCLAIMER. These designs are
//**              provided to you "as is". Orange Tree Technologies and its licensors
//**              make and you receive no warranties or conditions, express, implied,
//**              statutory or otherwise, and Orange Tree Technologies specifically
//**              disclaims any implied warranties of merchantability, non-infringement,
//**              or fitness for a particular purpose. Orange Tree Technologies does not
//**              warrant that the functions contained in these designs will meet your
//**              requirements, or that the operation of these designs will be
//**              uninterrupted or error free, or that defects in the Designs will be
//**              corrected. Furthermore, Orange Tree Technologies does not warrant or
//**              make any representations regarding use or the results of the use of the
//**              designs in terms of correctness, accuracy, reliability, or otherwise.
//**
//**              LIMITATION OF LIABILITY. In no event will Orange Tree Technologies
//**              or its licensors be liable for any loss of data, lost profits, cost or
//**              procurement of substitute goods or services, or for any special,
//**              incidental, consequential, or indirect damages arising from the use or
//**              operation of the designs or accompanying documentation, however caused
//**              and on any theory of liability. This limitation will apply even if
//**              Orange Tree Technologies has been advised of the possibility of such
//**              damage. This limitation shall apply notwithstanding the failure of the
//**              essential purpose of any limited remedies herein.
//**
//*****************************************************************************************

#define _CRT_SECURE_NO_WARNINGS
#ifdef WINGCC
#define __USE_W32_SOCKETS
#endif

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include "ZestETM1.h"
#include "Private.h"

#ifdef WINGCC
#define _strnicmp strncasecmp
#endif
#if !defined(MSVC) && !defined(WINGCC)
#define _strnicmp strncasecmp
#include <sys/types.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <ifaddrs.h>
#endif

// UPnP search string
static char *ZestETM1_SearchReq =
"M-SEARCH * HTTP/1.1\r\n"
"ST: upnp:rootdevice\r\n"
"MX: %d\r\n"
"MAN: \"ssdp:discover\"\r\n"
"HOST: 239.255.255.250:1900\r\n"
;

```

```
// UPnP broadcast address and port
#define ZESTETM1_UPNP_ADDR "239.255.255.250"
#define ZESTETM1_UPNP_PORT 1900

// Get settings command structure
typedef struct
{
    uint8_t Command;
    uint8_t Dummy[3];
} ZESTETM1_GET_SETTINGS_CMD;
typedef struct
{
    uint8_t Command;
    uint8_t Status;
    uint8_t Dummy1[2];
    uint16_t SoftwareVersion;
    uint16_t HardwareVersion;
    uint32_t SerialNumber;
    uint32_t IPAddr;
    uint32_t Gateway;
    uint32_t SubNet;
    uint16_t HTTPPort;
    uint16_t ControlPort;
    uint8_t MACAddr[6];
    uint8_t Dummy2[2];
} ZESTETM1_GET_SETTINGS_RESPONSE;
#define ZESTETM1_COMMAND_GET_SETTINGS 0xf0

/*************************
* Issue HTTP GET command *
*************************/
static int ZestETM1_HTTPGet(char *IPAddr, char *Port,
                            char *FileName, void *Buffer,
                            int BufferLength, int Wait)
{
    struct addrinfo *Addr = NULL,
                    *Ptr = NULL,
                    Hints;
    int Result;
    SOCKET Socket;
    int Offset = 0;
    char Req[1024];
    char *HdrEnd;
    fd_set ReadFDS;
    struct timeval Timeout;

    Timeout.tv_sec = Wait/1000;
    Timeout.tv_usec = (Wait%1000)*1000;

    // Attempt to connect to the address
    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family = AF_UNSPEC;
    Hints.ai_socktype = SOCK_STREAM;
    Hints.ai_protocol = IPPROTO_TCP;

    // Resolve the server address and port
    Result = getaddrinfo(IPAddr, Port, &Hints, &Addr);
    if (Result!=0)
        return -1;

    for (Ptr=Addr; Ptr!=NULL; Ptr=Ptr->ai_next)
    {
        // Create a SOCKET for connecting to server
        Socket = socket(Ptr->ai_family, Ptr->ai_socktype,
                        Ptr->ai_protocol);
        if (Socket<0)
        {
            freeaddrinfo(Addr);
            return -1;
        }

        // Connect to ZestETM1
        Result = connect(Socket, Ptr->ai_addr, (int)Ptr->ai_addrlen);
        if (Result<0)
        {
            closesocket(Socket);
            Socket = -1;
            continue;
        }
        break;
    }
    freeaddrinfo(Addr);

    // Send GET request
    sprintf(Req, "GET /%s HTTP/1.1\r\nHOST: %s:%s\r\nContent-length: 0\r\n\r\n", FileName, IPAddr, Port);
    Result = send(Socket, Req, (int)strlen(Req), 0);
    if (Result!=strlen(Req))
    {
        closesocket(Socket);
        return -1;
    }

    // Get response
    ((char*)Buffer)[0] = 0;
    do
    {
        FD_ZERO(&ReadFDS);
        FD_SET(Socket, &ReadFDS);
        Result = select((int)Socket+1, &ReadFDS, NULL, NULL, &Timeout);
        if (Result<0)
        {
            closesocket(Socket);
            return -1;
        }
        if (!FD_ISSET(Socket, &ReadFDS)) break;
        Result = recv(Socket, (char *)Buffer+Offset, BufferLength-Offset, 0);
        if (Result<0)
        {
            closesocket(Socket);
            return -1;
        }
        Offset+=Result;
        if (Offset==BufferLength)
        {
            break;
        }
    } while (Result!=0);

    // Check status response
    if (_strnicmp("HTTP/1.1 200 OK", Buffer, 15)!=0)
    {
        closesocket(Socket);
        return -1;
    }

    // Remove HTTP header
    HdrEnd = strstr(Buffer, "\r\n\r\n");
    if (HdrEnd==NULL)
    {
        Offset = 0;
    }
    else
    {
        Offset -= (int)(HdrEnd+4-(char*)Buffer);
        memcpy(Buffer, HdrEnd+4, Offset);
    }

    closesocket(Socket);
    return Offset;
}

/******************************
* Read settings from ETM1 flash *
******************************/
static ZESTETM1_STATUS ZestETM1_ReadSettings(ZESTETM1_CARD_INFO *CardInfo)
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_GET_SETTINGS_CMD Cmd;
    ZESTETM1_GET_SETTINGS_RESPONSE Response;

    if (CardInfo==NULL)
    {
        return ZESTETM1_NULL_PARAMETER;
    }

    // Open control connection
    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP,
                                     CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        return Result;
    }

    // Get the settings from the device
    Cmd.Command = ZESTETM1_COMMAND_GET_SETTINGS;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        return Result;
    }

    // Result values (including endian conversion)
    CardInfo->FirmwareVersion = ((Response.SoftwareVersion»8)&0xff)|((Response.SoftwareVersion&0xff)«8);
    CardInfo->HardwareVersion = ((Response.HardwareVersion»8)&0xff)|((Response.HardwareVersion&0xff)«8);
    CardInfo->Gateway[0] = (uint8_t)((Response.Gateway»0)&0xff);
    CardInfo->Gateway[1] = (uint8_t)((Response.Gateway»8)&0xff);
    CardInfo->Gateway[2] = (uint8_t)((Response.Gateway»16)&0xff);
    CardInfo->Gateway[3] = (uint8_t)((Response.Gateway»24)&0xff);
    CardInfo->SubNet[0] = (uint8_t)((Response.SubNet»0)&0xff);
    CardInfo->SubNet[1] = (uint8_t)((Response.SubNet»8)&0xff);
    CardInfo->SubNet[2] = (uint8_t)((Response.SubNet»16)&0xff);
    CardInfo->SubNet[3] = (uint8_t)((Response.SubNet»24)&0xff);
    memcpy(CardInfo->MACAddr, Response.MACAddr, 6);
    CardInfo->ControlPort = ((Response.ControlPort»8)&0xff)|((Response.ControlPort&0xff)«8);
    CardInfo->HTTPPort = ((Response.HTTPPort»8)&0xff)|((Response.HTTPPort&0xff)«8);
    CardInfo->SerialNumber = ZESTETM1_REVERSE(Response.SerialNumber);

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
        return Result;

    return ZESTETM1_SUCCESS;
}


/*********************************
* Read information about a card *
*********************************/
static void ZestETM1_GetCardInfo(char *Location,
                                 uint32_t *NumCards,
                                 ZESTETM1_CARD_INFO **CardInfo,
                                 int Wait)
{
    static char *Prefix = "http://";
    char Buffer[65536];
    char *IPAddr = NULL;
    char *FileName = NULL;
    char *Port = "80";
    int i;

    // Extract address and port
    for (i=0; Location[i]!=0 && Location[i]!='\r' && Location[i]!='\n'; i++)
    {
        if (IPAddr==NULL && Prefix[i]==0) IPAddr = Location+i;
        if (IPAddr==NULL && Location[i]!=Prefix[i])
            return;
        if (IPAddr!=NULL)
        {
            if (Location[i]=='/' || Location[i]==0 ||
                Location[i]=='\r' || Location[i]=='\n')
            {
                Location[i]=0;
                i++;
                break;
            }
            if (Location[i]==':')
            {
                int j=i;
                while (Location[j]!=0 && Location[j]!='\r' &&
                       Location[j]!='\n' && Location[j]!='/') j++;
                Location[i]=0;
                Location[j]=0;
                Port = Location+i+1;
                i = j+1;
                break;
            }
        }
    }
    if (IPAddr==NULL || Location[i]==0 || Location[i]=='\r' || Location[i]=='\n')
```

```
        return;

    // Extract XML filename
    FileName = Location+i;
    while (Location[i]!=0 && Location[i]!='\r' && Location[i]!='\n') i++;
    Location[i] = 0;

    // Get XML file
    memset(Buffer, 0, sizeof(Buffer));
    if (ZestETM1_HTTPGet(IPAddr, Port, FileName,
                         Buffer, sizeof(Buffer), Wait)>0)
    {
        uint8_t IPAddr[4];
        uint32_t i;
        uint16_t Port;

        // Parse XML for service description URL
        char *ControlURL = strstr(Buffer, "<controlURL>");
        if (ControlURL==NULL)
            return;

        ControlURL += 12;
        for (i=0; i<4; i++)
        {
            char *EndPtr;
            IPAddr[i] = (uint8_t)strtoul(ControlURL, &EndPtr, 10);
            if ((i!=3 && *EndPtr!='.') || (i==3 && *EndPtr!=':'))
                break;
            ControlURL = EndPtr+1;
        }
        if (i!=4)
            return;

        Port = atoi(ControlURL);

        // Make sure only unique devices are added to the list!
        if (*CardInfo!=NULL)
        {
            for (i=0; i<*NumCards; i++)
            {
                if ((*CardInfo)[i].ControlPort==Port &&
                    memcmp((*CardInfo)[i].IPAddr, IPAddr, sizeof(IPAddr))==0)
                {
                    break;
                }
            }
        }
        if (*CardInfo==NULL || i==*NumCards)
        {
            uint32_t Index = (*NumCards);
            ZESTETM1_CARD_INFO *NewBuffer;

            // Allocate space for new card info structure
            if ((*CardInfo)==NULL)
                NewBuffer = malloc(sizeof(ZESTETM1_CARD_INFO)*(Index+1));
            else
                NewBuffer = realloc(*CardInfo, sizeof(ZESTETM1_CARD_INFO)*(Index+1));

            // Get new card settings
            NewBuffer[Index].ControlPort = Port;
            memcpy(NewBuffer[Index].IPAddr, IPAddr, sizeof(IPAddr));
            NewBuffer[Index].Timeout = Wait;
            if (ZestETM1_ReadSettings(&(NewBuffer[Index]))!=ZESTETM1_SUCCESS)
            {
                if (*CardInfo==NULL)
                    free(NewBuffer);
                else
                {
                    (*CardInfo) = NewBuffer;
                    memset(NewBuffer[Index].IPAddr, 0, sizeof(NewBuffer[Index].IPAddr));
                    NewBuffer[Index].ControlPort = 0;
                }
            }
            else
            {
                NewBuffer[Index].Timeout = ZESTETM1_DEFAULT_TIMEOUT;
                (*CardInfo) = NewBuffer;
                (*NumCards)++;
            }
        }
    }
```

```
}


/*****************************************************
* Multicasting functions to join and leave a group *
*****************************************************/
static int ZestETM1_JoinGroup(SOCKET sd, uint32_t grpaddr,
                             uint32_t iaddr)
{
    struct ip_mreq imr;

    imr.imr_multiaddr.s_addr  = grpaddr;
    imr.imr_interface.s_addr  = iaddr;
    return setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                      (const char *)&imr, sizeof(imr));
}
static int ZestETM1_LeaveGroup(SOCKET sd, uint32_t grpaddr,
                              uint32_t iaddr)
{
    struct ip_mreq imr;

    imr.imr_multiaddr.s_addr  = grpaddr;
    imr.imr_interface.s_addr  = iaddr;
    return setsockopt(sd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                      (const char *)&imr, sizeof(imr));
}

/*******************************************
* Get an array with all local IP addresses *
*******************************************/
static ZESTETM1_STATUS ZestETM1_GetAllAdapters(uint32_t *NumAdapters, struct sockaddr_in **Adapters)
{
#if defined(MSVC) || defined(WINGCC)
    SOCKET Socket;
    SOCKET_ADDRESS_LIST *AddressListPtr;
    DWORD BytesRequired;
    int i;
    int Count = 0;

    *NumAdapters = 0;
    *Adapters = NULL;

    Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (Socket<0)
        return ZESTETM1_INTERNAL_ERROR;

    WSAIoctl(Socket, SIO_ADDRESS_LIST_QUERY, NULL, 0,
             NULL, 0, (LPDWORD)&BytesRequired, NULL, NULL);
    AddressListPtr = (SOCKET_ADDRESS_LIST *)malloc(BytesRequired);
    if (AddressListPtr==NULL)
    {
        closesocket(Socket);
        return ZESTETM1_OUT_OF_MEMORY;
    }
    if (WSAIoctl(Socket, SIO_ADDRESS_LIST_QUERY, NULL, 0,
                 AddressListPtr, BytesRequired, &BytesRequired, NULL, NULL)==SOCKET_ERROR)
    {
        free(AddressListPtr);
        closesocket(Socket);
        return ZESTETM1_INTERNAL_ERROR;
    }

    for (i=0; i<AddressListPtr->iAddressCount; i++)
    {
        if (AddressListPtr->Address[i].iSockaddrLength==sizeof(struct sockaddr_in))
        {
            Count++;
            (*Adapters) = (struct sockaddr_in *)realloc(*Adapters, Count*sizeof(struct sockaddr_in));
            if ((*Adapters)==NULL)
            {
                free(AddressListPtr);
                closesocket(Socket);
                return ZESTETM1_OUT_OF_MEMORY;
            }
            memcpy(&(*Adapters)[Count-1], AddressListPtr->Address[i].lpSockaddr, sizeof(struct
sockaddr_in));
        }
    }
    *NumAdapters = Count;
    closesocket(Socket);
    free(AddressListPtr);

    return ZESTETM1_SUCCESS;
#else
    struct ifaddrs *Interfaces;
    struct ifaddrs *Ptr;
    uint32_t Count = 0;

    if (getifaddrs(&Interfaces)!=0)
        return ZESTETM1_INTERNAL_ERROR;

    *NumAdapters = 0;
    *Adapters = NULL;

    Ptr = Interfaces;
    while (Ptr!=NULL)
    {
        if (Ptr->ifa_addr!=NULL)
        {
            Count++;
            (*Adapters) = (struct sockaddr_in *)realloc(*Adapters, Count*sizeof(struct sockaddr_in));
            if ((*Adapters)==NULL)
            {
                freeifaddrs(Interfaces);
                return ZESTETM1_OUT_OF_MEMORY;
            }
            memcpy(&(*Adapters)[Count-1], Ptr->ifa_addr, sizeof(struct sockaddr_in));
        }
        Ptr = Ptr->ifa_next;
    }
    *NumAdapters = Count;
    freeifaddrs(Interfaces);

    return ZESTETM1_SUCCESS;
#endif
}

/******************************************************************************
* Scan networks for ZestETM1 cards and return the number of attached devices   *
* and details about each one                                                  *
******************************************************************************/
ZESTETM1_STATUS ZestETM1CountCards(unsigned long *NumCards,
                                   ZESTETM1_CARD_INFO **CardInfo, unsigned long Wait)
{
    SOCKET Socket;
    struct sockaddr_in DestIP;
    struct sockaddr_in SourceIP;
    int SourceIPLength;
    int Flag = 1;
    int Result;
    char Req[1024];
    char Response[1024];
    ZESTETM1_CARD_INFO *Cards = NULL;
    uint32_t CardCount = 0;
    int i;
    struct timeval Timeout;
    fd_set ReadFDS;
    uint32_t Interface;
    uint32_t NumAdapters;
    struct sockaddr_in *Adapters;
    ZESTETM1_STATUS Status;

    *NumCards = 0;
    *CardInfo = NULL;

    // Get a list of all adapters
    Status = ZestETM1_GetAllAdapters(&NumAdapters, &Adapters);
    if (Status!=ZESTETM1_SUCCESS)
        ZESTETM1_ERROR_GENERAL("ZestETM1CountCards", Status);
    if (NumAdapters==0)
    {
        *NumCards = 0;
        return ZESTETM1_SUCCESS;
    }

    // Send queries on all interfaces
    for (Interface=0; Interface<NumAdapters; Interface++)
    {
        if (Adapters[Interface].sin_family!=AF_INET)
            continue;

        // Open socket for search requests
        Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (Socket<0)
        {
            continue;
        }

        // Set reuse port to on to allow multiple binds per host
        if (setsockopt(Socket, SOL_SOCKET, SO_REUSEADDR, (char *)&Flag,
                       sizeof(Flag))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Bind to port for receiving responses
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        Result = bind(Socket, (const struct sockaddr *)(&Adapters[Interface]), sizeof(struct
sockaddr_in));
        if (Result<0)
        {
            closesocket(Socket);
            continue;
        }

        // Join multicast group
        if (ZestETM1_JoinGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                               htonl(INADDR_ANY))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Send M-SEARCH request
        // Send more than once as UDP is unreliable
        DestIP.sin_family = AF_INET;
        DestIP.sin_addr.s_addr = inet_addr(ZESTETM1_UPNP_ADDR);
        DestIP.sin_port = htons(1900);
        sprintf(Req, ZestETM1_SearchReq, (Wait+999)/1000);
        for (i=0; i<3; i++)
        {
            Result = sendto(Socket, ZestETM1_SearchReq,
                            (int)strlen(ZestETM1_SearchReq),
                            0, (struct sockaddr *)&DestIP, sizeof(DestIP));
            if (Result!=strlen(ZestETM1_SearchReq))
            {
                ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                                    htonl(INADDR_ANY));
                closesocket(Socket);
                continue;
            }
        }

        // Read responses
        Timeout.tv_sec = Wait/1000;
        Timeout.tv_usec = (Wait%1000)*1000;
        do
        {
            FD_ZERO(&ReadFDS);
            FD_SET(Socket, &ReadFDS);
            Result = select((int)Socket+1, &ReadFDS, NULL, NULL, &Timeout);
            if (Result<0)
            {
                break;
            }
            if (!FD_ISSET(Socket, &ReadFDS)) break;

            SourceIPLength = sizeof(SourceIP);
            Result = recvfrom(Socket, Response, sizeof(Response),
                              0, (struct sockaddr *)&SourceIP, &SourceIPLength);
            if (Result<0)
            {
                // Error!
                break;
            }
            else if (Result==0)
            {
                // Clean shut down
                break;
            }
            else
            {
                // Parse results
                if (_strnicmp("NOTIFY", Response, 6)==0 ||
                    _strnicmp("HTTP/1.1 200 OK", Response, 15)==0)
                {
                    // Check its a GigExpedite and request XML description
                    char *Server = strstr(Response, "GigExpedite2");
                    char *Location = strstr(Response, "LOCATION");
                    if (Server!=NULL && Location!=NULL)
                    {
                        char *c;
                        for (c=Location+10; *c!=0 && *c!='\n' && *c!='\r'; c++);
                        *c = 0;
                        ZestETM1_GetCardInfo(Location+10, &CardCount, &Cards, Wait);
                    }
                }
            }
        } while(1);

        // Leave multicast group
        ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                            htonl(INADDR_ANY));

        // Close socket
        closesocket(Socket);
    }

    *NumCards = CardCount;
    *CardInfo = Cards;
    free(Adapters);

    return ZESTETM1_SUCCESS;
}

/******************************************************
* Free data structures returned by ZestETM1CountCards *
******************************************************/
ZESTETM1_STATUS ZestETM1FreeCards(ZESTETM1_CARD_INFO *CardInfo)
{
    if (CardInfo!=NULL)
        free(CardInfo);

    return ZESTETM1_SUCCESS;
}

/**********************************
* Fill in card information fields *
**********************************/
ZESTETM1_STATUS ZestETM1GetCardInfo(ZESTETM1_CARD_INFO *CardInfo)
{
    ZESTETM1_STATUS Status;

    Status = ZestETM1_ReadSettings(CardInfo);
    if (Status!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1GetCardInfo", Status);
    }

    return ZESTETM1_SUCCESS;
}

```

7.31 C:/Users/hill35/git/camera_python/nsCamera/comms/ZestETM1/Zest↩ ETM1.h File Reference

Classes

- struct ZESTETM1_CARD_INFO

Macros

- #define ZESTETM1_VERSION_FALLBACK 0x8000
- #define ZESTETM1_INFO_BASE 0
- #define ZESTETM1_WARNING_BASE 0x4000
- #define ZESTETM1_ERROR_BASE 0x8000

Typedefs

- typedef void ∗ ZESTETM1_HANDLE
- typedef void ∗ ZESTETM1_CONNECTION
- typedef void(∗ ZESTETM1_ERROR_FUNC) (const char ∗Function, ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_STATUS Status, const char ∗Msg)

Enumerations

- enum ZESTETM1_CONNECTION_TYPE { ZESTETM1_TYPE_TCP , ZESTETM1_TYPE_UDP }
- enum ZESTETM1_SPI_RATE { ZESTETM1_SPI_RATE_35 , ZESTETM1_SPI_RATE_17_5 , ZESTETM1_SPI_RATE_8_75 }
- enum ZESTETM1_STATUS {
  ZESTETM1_SUCCESS = ZESTETM1_INFO_BASE , ZESTETM1_MAX_INFO , ZESTETM1_MAX_WARNING = ZESTETM1_WARNING_BASE , ZESTETM1_SOCKET_ERROR = ZESTETM1_ERROR_BASE , ZESTETM1_INTERNAL_ERROR , ZESTETM1_ILLEGAL_STATUS_CODE , ZESTETM1_NULL_PARAMETER , ZESTETM1_OUT_OF_MEMORY ,
  ZESTETM1_INVALID_CONNECTION_TYPE , ZESTETM1_ILLEGAL_CONNECTION , ZESTETM1_SOCKET_CLOSED , ZESTETM1_TIMEOUT ,
  ZESTETM1_ILLEGAL_PARAMETER , ZESTETM1_MAX_ERROR }

Functions

- ZESTETM1_STATUS ZestETM1Init (void)
- ZESTETM1_STATUS ZestETM1Close (void)
- ZESTETM1_STATUS ZestETM1CountCards (unsigned long ∗NumCards, ZESTETM1_CARD_INFO ∗∗CardInfo, unsigned long Wait)
- ZESTETM1_STATUS ZestETM1GetCardInfo (ZESTETM1_CARD_INFO ∗CardInfo)
- ZESTETM1_STATUS ZestETM1FreeCards (ZESTETM1_CARD_INFO ∗CardInfo)
- ZESTETM1_STATUS ZestETM1RegisterErrorHandler (ZESTETM1_ERROR_FUNC Function)
- ZESTETM1_STATUS ZestETM1GetErrorMessage (ZESTETM1_STATUS Status, char ∗∗Buffer)
- ZESTETM1_STATUS ZestETM1OpenConnection (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_CONNECTION_TYPE Type, unsigned short Port, unsigned short LocalPort, ZESTETM1_CONNECTION ∗Connection)
- ZESTETM1_STATUS ZestETM1CloseConnection (ZESTETM1_CONNECTION Connection)
- ZESTETM1_STATUS ZestETM1WriteData (ZESTETM1_CONNECTION Connection, void ∗Buffer, unsigned long Length, unsigned long ∗Written, unsigned long Timeout)
- ZESTETM1_STATUS ZestETM1ReadData (ZESTETM1_CONNECTION Connection, void ∗Buffer, unsigned long Length, unsigned long ∗Read, unsigned long Timeout)
- ZESTETM1_STATUS ZestETM1SPIReadWrite (ZESTETM1_CARD_INFO ∗CardInfo, ZESTETM1_SPI_RATE Rate, int WordLen, void ∗WriteData, void ∗ReadData, unsigned long Length, int ReleaseCS)
- ZESTETM1_STATUS ZestETM1WriteRegister (ZESTETM1_CARD_INFO ∗CardInfo, unsigned long Addr, unsigned short Data)
- ZESTETM1_STATUS ZestETM1ReadRegister (ZESTETM1_CARD_INFO ∗CardInfo, unsigned long Addr, unsigned short ∗Data)
- ZESTETM1_STATUS ZestETM1SetInterrupt (ZESTETM1_CARD_INFO ∗CardInfo)

7.31.1 Class Documentation

7.31.1.1 struct ZESTETM1_CARD_INFO

Definition at line 30 of file ZestETM1.h.

Class Members

unsigned short	ControlPort	
unsigned long	FirmwareVersion	
unsigned char	Gateway[4]	
unsigned long	HardwareVersion	
unsigned short	HTTPPort	
unsigned char	IPAddr[4]	
unsigned char	MACAddr[6]	
unsigned long	SerialNumber	
unsigned char	SubNet[4]	
unsigned long	Timeout	

7.31.2 Macro Definition Documentation

7.31.2.1 ZESTETM1_ERROR_BASE

```
#define ZESTETM1_ERROR_BASE 0x8000
```

Definition at line 77 of file ZestETM1.h.

7.31.2.2 ZESTETM1_INFO_BASE

```
#define ZESTETM1_INFO_BASE 0
```

Definition at line 75 of file ZestETM1.h.

7.31.2.3 ZESTETM1_VERSION_FALLBACK

```
#define ZESTETM1_VERSION_FALLBACK 0x8000
```

Definition at line 50 of file ZestETM1.h.

7.31.2.4 ZESTETM1_WARNING_BASE

```
#define ZESTETM1_WARNING_BASE 0x4000
```

Definition at line 76 of file ZestETM1.h.

7.31.3 Typedef Documentation

7.31.3.1 ZESTETM1_CONNECTION

```
typedef void* ZESTETM1_CONNECTION
```

Definition at line 55 of file ZestETM1.h.

7.31.3.2 ZESTETM1_ERROR_FUNC

```
typedef void(* ZESTETM1_ERROR_FUNC) (const char *Function, ZESTETM1_CARD_INFO *CardInfo, ZESTETM1_STATUS
Status, const char *Msg)
```

Definition at line 98 of file ZestETM1.h.

7.31.3.3 ZESTETM1_HANDLE

```
typedef void* ZESTETM1_HANDLE
```

Definition at line 24 of file ZestETM1.h.

7.31.4 Enumeration Type Documentation

7.31.4.1 ZESTETM1_CONNECTION_TYPE

```
enum ZESTETM1_CONNECTION_TYPE
```

Enumerator

ZESTETM1_TYPE_TCP	
ZESTETM1_TYPE_UDP	

Definition at line 56 of file ZestETM1.h.

```
00057 {
00058     ZESTETM1_TYPE_TCP,
00059     ZESTETM1_TYPE_UDP
00060 } ZESTETM1_CONNECTION_TYPE;
```

7.31.4.2 ZESTETM1_SPI_RATE

```
enum ZESTETM1_SPI_RATE
```

Enumerator

ZESTETM1_SPI_RATE_35	

Enumerator

ZESTETM1_SPI_RATE_17←_5	
ZESTETM1_SPI_RATE_8_75	

Definition at line 65 of file ZestETM1.h.

```
{
    ZESTETM1_SPI_RATE_35,
    ZESTETM1_SPI_RATE_17_5,
    ZESTETM1_SPI_RATE_8_75,
} ZESTETM1_SPI_RATE;
```

7.31.4.3 ZESTETM1_STATUS

```
enum ZESTETM1_STATUS
```

Enumerator

ZESTETM1_SUCCESS	
ZESTETM1_MAX_INFO	
ZESTETM1_MAX_WARNING	
ZESTETM1_SOCKET_ERROR	
ZESTETM1_INTERNAL_ERROR	
ZESTETM1_ILLEGAL_STATUS_CODE	
ZESTETM1_NULL_PARAMETER	
ZESTETM1_OUT_OF_MEMORY	
ZESTETM1_INVALID_CONNECTION_TYPE	
ZESTETM1_ILLEGAL_CONNECTION	
ZESTETM1_SOCKET_CLOSED	
ZESTETM1_TIMEOUT	
ZESTETM1_ILLEGAL_PARAMETER	
ZESTETM1_MAX_ERROR	

Definition at line 78 of file ZestETM1.h.

```
{
    ZESTETM1_SUCCESS = ZESTETM1_INFO_BASE,
    ZESTETM1_MAX_INFO,

    ZESTETM1_MAX_WARNING = ZESTETM1_WARNING_BASE,

    ZESTETM1_SOCKET_ERROR = ZESTETM1_ERROR_BASE,
    ZESTETM1_INTERNAL_ERROR,
    ZESTETM1_ILLEGAL_STATUS_CODE,
    ZESTETM1_NULL_PARAMETER,
    ZESTETM1_OUT_OF_MEMORY,
    ZESTETM1_INVALID_CONNECTION_TYPE,
    ZESTETM1_ILLEGAL_CONNECTION,
    ZESTETM1_SOCKET_CLOSED,
    ZESTETM1_TIMEOUT,
    ZESTETM1_ILLEGAL_PARAMETER,

    ZESTETM1_MAX_ERROR
} ZESTETM1_STATUS;
```

7.31.5 Function Documentation

7.31.5.1 ZestETM1Close()

```
ZESTETM1_STATUS ZestETM1Close (
                void )
```

Definition at line 92 of file Main.c.

```
{
#if defined(MSVC) || defined(WINGCC)
    WSACleanup();
#endif

    return ZESTETM1_SUCCESS;
}
```

7.31.5.2 ZestETM1CloseConnection()

```
ZESTETM1_STATUS ZestETM1CloseConnection (
                ZESTETM1_CONNECTION Connection )
```

Definition at line 575 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1CloseConnection", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.3 ZestETM1CountCards()

```
ZESTETM1_STATUS ZestETM1CountCards (
                unsigned long * NumCards,
                ZESTETM1_CARD_INFO ** CardInfo,
                unsigned long Wait )
```

Definition at line 521 of file UPnP.c.

```
{
    SOCKET Socket;
    struct sockaddr_in DestIP;
    struct sockaddr_in SourceIP;
    int SourceIPLength;
    int Flag = 1;
    int Result;
    char Req[1024];
    char Response[1024];
    ZESTETM1_CARD_INFO *Cards = NULL;
    uint32_t CardCount = 0;
    int i;
    struct timeval Timeout;
    fd_set ReadFDS;
    uint32_t Interface;
    uint32_t NumAdapters;
    struct sockaddr_in *Adapters;
    ZESTETM1_STATUS Status;

    *NumCards = 0;
    *CardInfo = NULL;

    // Get a list of all adapters
    Status = ZestETM1_GetAllAdapters(&NumAdapters, &Adapters);
    if (Status!=ZESTETM1_SUCCESS)
        ZESTETM1_ERROR_GENERAL("ZestETM1CountCards", Status);
    if (NumAdapters==0)
    {
        *NumCards = 0;
        return ZESTETM1_SUCCESS;
    }

    // Send queries on all interfaces
    for (Interface=0; Interface<NumAdapters; Interface++)
    {
        if (Adapters[Interface].sin_family!=AF_INET)
            continue;

        // Open socket for search requests
        Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (Socket<0)
        {
            continue;
        }

        // Set reuse port to on to allow multiple binds per host
        if (setsockopt(Socket, SOL_SOCKET, SO_REUSEADDR, (char *)&Flag,
                       sizeof(Flag))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Bind to port for receiving responses
        SourceIP.sin_family = AF_INET;
        SourceIP.sin_addr.s_addr = htonl(INADDR_ANY);
        SourceIP.sin_port = 0;
        Result = bind(Socket, (const struct sockaddr *)(&Adapters[Interface]), sizeof(struct sockaddr_in));
        if (Result<0)
        {
            closesocket(Socket);
            continue;
        }

        // Join multicast group
        if (ZestETM1_JoinGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                              htonl(INADDR_ANY))<0)
        {
            closesocket(Socket);
            continue;
        }

        // Send M-SEARCH request
        // Send more than once as UDP is unreliable
        DestIP.sin_family = AF_INET;
        DestIP.sin_addr.s_addr = inet_addr(ZESTETM1_UPNP_ADDR);
        DestIP.sin_port = htons(1900);
        sprintf(Req, ZestETM1_SearchReq, (Wait+999)/1000);
        for (i=0; i<3; i++)
        {
            Result = sendto(Socket, ZestETM1_SearchReq,
                            (int)strlen(ZestETM1_SearchReq),
                            0, (struct sockaddr *)&DestIP, sizeof(DestIP));
            if (Result!=strlen(ZestETM1_SearchReq))
            {
                ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                                   htonl(INADDR_ANY));
                closesocket(Socket);
                continue;
            }
        }

        // Read responses
        Timeout.tv_sec = Wait/1000;
        Timeout.tv_usec = (Wait%1000)*1000;
        do
        {
            FD_ZERO(&ReadFDS);
            FD_SET(Socket, &ReadFDS);
            Result = select((int)Socket+1, &ReadFDS, NULL, NULL, &Timeout);
            if (Result<0)
            {
                break;
            }
            if (!FD_ISSET(Socket, &ReadFDS)) break;

            SourceIPLength = sizeof(SourceIP);
            Result = recvfrom(Socket, Response, sizeof(Response),
                              0, (struct sockaddr *)&SourceIP, &SourceIPLength);
            if (Result<0)
            {
                // Error!
                break;
            }
            else if (Result==0)
            {
                // Clean shut down
                break;
            }
            else
            {
                // Parse results
                if (_strnicmp("NOTIFY", Response, 6)==0 ||
                    _strnicmp("HTTP/1.1 200 OK", Response, 15)==0)
                {
                    // Check its a GigExpedite and request XML description
                    char *Server = strstr(Response, "GigExpedite2");
                    char *Location = strstr(Response, "LOCATION");
                    if (Server!=NULL && Location!=NULL)
                    {
                        char *c;
                        for (c=Location+10; *c!=0 && *c!='\n' && *c!='\r'; c++);
                        *c = 0;
                        ZestETM1_GetCardInfo(Location+10, &CardCount, &Cards, Wait);
                    }
                }
            }
        } while(1);

        // Leave multicast group
        ZestETM1_LeaveGroup(Socket, inet_addr(ZESTETM1_UPNP_ADDR),
                           htonl(INADDR_ANY));

        // Close socket
        closesocket(Socket);
    }

    *NumCards = CardCount;
    *CardInfo = Cards;
    free(Adapters);

    return ZESTETM1_SUCCESS;
}
```

7.31.5.4 ZestETM1FreeCards()

```
ZESTETM1_STATUS ZestETM1FreeCards (
              ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 680 of file UPnP.c.

```
{
    if (CardInfo!=NULL)
        free(CardInfo);

    return ZESTETM1_SUCCESS;
}
```

7.31.5.5 ZestETM1GetCardInfo()

```
ZESTETM1_STATUS ZestETM1GetCardInfo (
              ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 691 of file UPnP.c.

```
{
    ZESTETM1_STATUS Status;

    Status = ZestETM1_ReadSettings(CardInfo);
    if (Status!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1GetCardInfo", Status);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.6 ZestETM1GetErrorMessage()

```
ZESTETM1_STATUS ZestETM1GetErrorMessage (
              ZESTETM1_STATUS Status,
              char ** Buffer )
```

Definition at line 84 of file Error.c.

```
{
    if (Status>ZESTETM1_MAX_ERROR ||
        (Status<ZESTETM1_ERROR_BASE && Status>=ZESTETM1_MAX_WARNING) ||
        (Status<ZESTETM1_WARNING_BASE && Status>=ZESTETM1_MAX_INFO))
    {
        return ZESTETM1_ILLEGAL_STATUS_CODE;
    }

    *Buffer = ZESTETM1_ERROR_STRING(Status);
    return ZESTETM1_SUCCESS;
}
```

7.31.5.7 ZestETM1Init()

```
ZESTETM1_STATUS ZestETM1Init (
              void  )
```

Definition at line 58 of file Main.c.

```
{
#if defined(MSVC) || defined(WINGCC)
    WORD VersionRequested;
    WSADATA WSAData;
    int Error;

    VersionRequested = MAKEWORD(2, 2);
    Error = WSAStartup(VersionRequested, &WSAData);
    if (Error!=0)
    {
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }

    // Confirm that the WinSock DLL supports 2.2.
    // Note that if the DLL supports versions greater
    // than 2.2 in addition to 2.2, it will still return
    // 2.2 in Version since that is the version we
    // requested.
    if (LOBYTE(WSAData.wVersion)!=2 ||
        HIBYTE(WSAData.wVersion)!=2)
    {
        WSACleanup( );
        ZESTETM1_ERROR_GENERAL("ZestETM1Init", ZESTETM1_SOCKET_ERROR);
    }
#endif

    return ZESTETM1_SUCCESS;
}
```

7.31.5.8 ZestETM1OpenConnection()

```
ZESTETM1_STATUS ZestETM1OpenConnection (
            ZESTETM1_CARD_INFO * CardInfo,
            ZESTETM1_CONNECTION_TYPE Type,
            unsigned short Port,
            unsigned short LocalPort,
            ZESTETM1_CONNECTION * Connection )
```

7.31.5.9 ZestETM1ReadData()

```
ZESTETM1_STATUS ZestETM1ReadData (
            ZESTETM1_CONNECTION Connection,
            void * Buffer,
            unsigned long Length,
            unsigned long * Read,
            unsigned long Timeout )
```

Definition at line 613 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_ReadData(Connection, Buffer, Length, Read, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1ReadData", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.10 ZestETM1ReadRegister()

```
ZESTETM1_STATUS ZestETM1ReadRegister (
            ZESTETM1_CARD_INFO * CardInfo,
            unsigned long Addr,
            unsigned short * Data )
```

Definition at line 728 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_READ_REG_CMD Cmd;
    ZESTETM1_READ_REG_RESPONSE Response;

    if (CardInfo==NULL || Data==NULL)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    // Read register from the device
    Cmd.Command = ZESTETM1_COMMAND_READ_REG;
    Cmd.Addr = (uint8_t)Addr;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1ReadRegister", ZESTETM1_INTERNAL_ERROR);
    }
    *Data = ((Response.Value»8)&0xff) | ((Response.Value&0xff)«8);

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1ReadRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.11 ZestETM1RegisterErrorHandler()

```
ZESTETM1_STATUS ZestETM1RegisterErrorHandler (
              ZESTETM1_ERROR_FUNC Function )
```

Definition at line 74 of file Error.c.

```
{
    ZestETM1_ErrorHandler = Function;
    return ZESTETM1_SUCCESS;
}
```

7.31.5.12 ZestETM1SetInterrupt()

```
ZESTETM1_STATUS ZestETM1SetInterrupt (
              ZESTETM1_CARD_INFO * CardInfo )
```

Definition at line 780 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_MAILBOX_INT_CMD Cmd;
    ZESTETM1_MAILBOX_INT_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_NULL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    // Send command to set interrupt
    Cmd.Command = ZESTETM1_COMMAND_MAILBOX_INT;
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1SetInterrupt", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SetInterrupt", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.13 ZestETM1SPIReadWrite()

```
ZESTETM1_STATUS ZestETM1SPIReadWrite (
              ZESTETM1_CARD_INFO * CardInfo,
              ZESTETM1_SPI_RATE Rate,
              int WordLen,
              void * WriteData,
              void * ReadData,
              unsigned long Length,
              int ReleaseCS )
```

Definition at line 635 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    unsigned long RateVal = Rate==ZESTETM1_SPI_RATE_35 ? ZESTETM1_RATE_40MHz :
                            Rate==ZESTETM1_SPI_RATE_17_5 ? ZESTETM1_RATE_20MHz : ZESTETM1_RATE_10MHz;

    if (CardInfo==NULL || (WriteData==NULL && ReadData==NULL))
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_NULL_PARAMETER);
    }
    if (WordLen<1 || WordLen>32 || Length>16384)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_SPIReadWrite(CardInfo, Connection, ZESTETM1_USER_DEVICE_ID|RateVal, WordLen,
      (uint32_t *)WriteData, (uint32_t *)ReadData, Length, ReleaseCS, 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1SPIReadWrite", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.14 ZestETM1WriteData()

```
ZESTETM1_STATUS ZestETM1WriteData (
              ZESTETM1_CONNECTION Connection,
              void * Buffer,
              unsigned long Length,
              unsigned long * Written,
              unsigned long Timeout )
```

Definition at line 592 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION_STRUCT *Conn = (ZESTETM1_CONNECTION_STRUCT*)Connection;

    Result = ZestETM1_WriteData(Connection, Buffer, Length, Written, Timeout);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR_CONN("ZestETM1WriteData", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.31.5.15 ZestETM1WriteRegister()

```
ZESTETM1_STATUS ZestETM1WriteRegister (
        ZESTETM1_CARD_INFO * CardInfo,
        unsigned long Addr,
        unsigned short Data )
```

Definition at line 678 of file Data.c.

```
{
    ZESTETM1_STATUS Result;
    ZESTETM1_CONNECTION Connection;
    ZESTETM1_WRITE_REG_CMD Cmd;
    ZESTETM1_WRITE_REG_RESPONSE Response;

    if (CardInfo==NULL)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_NULL_PARAMETER);
    }
    if (Addr>127)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_ILLEGAL_PARAMETER);
    }

    Result = ZestETM1_OpenConnection(CardInfo, ZESTETM1_TYPE_TCP, CardInfo->ControlPort, 0, &Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    // Write register to the device
    //FIXME: Do we want to be able to copy this value to flash?
    Cmd.Command = ZESTETM1_COMMAND_WRITE_REG;
    Cmd.Addr = (uint8_t)Addr;
    Cmd.Data = ((Data»8)&0xff) | (Data&0xff);
    Result = ZestETM1_SendCommand(CardInfo, Connection,
                                  &Cmd, sizeof(Cmd),
                                  &Response, sizeof(Response), 1);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }
    if (Response.Command!=Cmd.Command || Response.Status!=0)
    {
        ZestETM1_CloseConnection(Connection);
        ZESTETM1_ERROR("ZestETM1WriteRegister", ZESTETM1_INTERNAL_ERROR);
    }

    Result = ZestETM1_CloseConnection(Connection);
    if (Result!=ZESTETM1_SUCCESS)
    {
        ZESTETM1_ERROR("ZestETM1WriteRegister", Result);
    }

    return ZESTETM1_SUCCESS;
}
```

7.32 ZestETM1.h

Go to the documentation of this file.

```
/*********************************************************
*                                                        *
* (c) 2012 Orange Tree Technologies Ltd                  *
*                                                        *
* ZestETM1.h                                             *
* Version 1.0                                            *
*                                                        *
* Header file for ZestETM1 Ethernet module               *
*                                                        *
*********************************************************/

#ifndef __ZESTETM1_H__
#define __ZESTETM1_H__

#ifdef __cplusplus
extern "C"
{
#endif


/*********************************
* Handle for referencing modules *
*********************************/
typedef void *ZESTETM1_HANDLE;


/*****************************
* Card information structure *
*****************************/
typedef struct
{
    // These must be filled in before calling functions
    unsigned char IPAddr[4];
    unsigned short ControlPort;
    unsigned long Timeout;

    // These are for information purposes only
    unsigned short HTTPPort;
    unsigned char MACAddr[6];
    unsigned char SubNet[4];
    unsigned char Gateway[4];
    unsigned long SerialNumber;
    unsigned long FirmwareVersion;
    unsigned long HardwareVersion;
} ZESTETM1_CARD_INFO;

// Fallback mask
// This will be set if the GigExpedite is operating in firmware version fallback mode
// due to a failed upload of firmware
#define ZESTETM1_VERSION_FALLBACK 0x8000

/****************************
* Data transfer definitions *
****************************/
typedef void *ZESTETM1_CONNECTION;
typedef enum
{
    ZESTETM1_TYPE_TCP,
    ZESTETM1_TYPE_UDP
} ZESTETM1_CONNECTION_TYPE;

/*************************
* Master SPI clock rates *
*************************/
typedef enum
{
    ZESTETM1_SPI_RATE_35,
    ZESTETM1_SPI_RATE_17_5,
    ZESTETM1_SPI_RATE_8_75,
} ZESTETM1_SPI_RATE;

/***********************
* Function return codes *
***********************/
#define ZESTETM1_INFO_BASE 0
#define ZESTETM1_WARNING_BASE 0x4000
#define ZESTETM1_ERROR_BASE 0x8000
typedef enum
{
    ZESTETM1_SUCCESS = ZESTETM1_INFO_BASE,
    ZESTETM1_MAX_INFO,

    ZESTETM1_MAX_WARNING = ZESTETM1_WARNING_BASE,

    ZESTETM1_SOCKET_ERROR = ZESTETM1_ERROR_BASE,
    ZESTETM1_INTERNAL_ERROR,
    ZESTETM1_ILLEGAL_STATUS_CODE,
    ZESTETM1_NULL_PARAMETER,
    ZESTETM1_OUT_OF_MEMORY,
    ZESTETM1_INVALID_CONNECTION_TYPE,
    ZESTETM1_ILLEGAL_CONNECTION,
    ZESTETM1_SOCKET_CLOSED,
    ZESTETM1_TIMEOUT,
    ZESTETM1_ILLEGAL_PARAMETER,

    ZESTETM1_MAX_ERROR
} ZESTETM1_STATUS;
typedef void (*ZESTETM1_ERROR_FUNC)(const char *Function,
                                   ZESTETM1_CARD_INFO *CardInfo,
                                   ZESTETM1_STATUS Status,
                                   const char *Msg);


/**********************
* Function prototypes *
**********************/
ZESTETM1_STATUS ZestETM1Init(void);
ZESTETM1_STATUS ZestETM1Close(void);
ZESTETM1_STATUS ZestETM1CountCards(unsigned long *NumCards,
                                 ZESTETM1_CARD_INFO **CardInfo,
                                 unsigned long Wait);
ZESTETM1_STATUS ZestETM1GetCardInfo(ZESTETM1_CARD_INFO *CardInfo);
ZESTETM1_STATUS ZestETM1FreeCards(ZESTETM1_CARD_INFO *CardInfo);

ZESTETM1_STATUS ZestETM1RegisterErrorHandler(ZESTETM1_ERROR_FUNC Function);
ZESTETM1_STATUS ZestETM1GetErrorMessage(ZESTETM1_STATUS Status,
                                      char **Buffer);

ZESTETM1_STATUS ZestETM1OpenConnection(ZESTETM1_CARD_INFO *CardInfo,
                                     ZESTETM1_CONNECTION_TYPE Type,
                                     unsigned short Port,
                                     unsigned short LocalPort,
                                     ZESTETM1_CONNECTION *Connection);
ZESTETM1_STATUS ZestETM1CloseConnection(ZESTETM1_CONNECTION Connection);
ZESTETM1_STATUS ZestETM1WriteData(ZESTETM1_CONNECTION Connection,
                                void *Buffer,
                                unsigned long Length,
                                unsigned long *Written,
                                unsigned long Timeout);
ZESTETM1_STATUS ZestETM1ReadData(ZESTETM1_CONNECTION Connection,
                               void *Buffer,
                               unsigned long Length,
                               unsigned long *Read,
                               unsigned long Timeout);

ZESTETM1_STATUS ZestETM1SPIReadWrite(ZESTETM1_CARD_INFO *CardInfo, ZESTETM1_SPI_RATE Rate,
                                     int WordLen, void *WriteData,
                                     void *ReadData, unsigned long Length,
                                     int ReleaseCS);

ZESTETM1_STATUS ZestETM1WriteRegister(ZESTETM1_CARD_INFO *CardInfo, unsigned long Addr, unsigned short Data);
ZESTETM1_STATUS ZestETM1ReadRegister(ZESTETM1_CARD_INFO *CardInfo, unsigned long Addr, unsigned short *Data);
ZESTETM1_STATUS ZestETM1SetInterrupt(ZESTETM1_CARD_INFO *CardInfo);

#ifdef __cplusplus
}
#endif

#endif // __ZESTETM1_H__

```

7.33 C:/Users/hill35/git/camera_python/nsCamera/sensors/daedalus.py File Reference

Classes

- class nsCamera.sensors.daedalus.daedalus

Namespaces

- namespace nsCamera
- namespace nsCamera.sensors
- namespace nsCamera.sensors.daedalus

7.34 daedalus.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Parameters and functions specific to the daedalus three-frame sensor


Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import logging
import numbers
from collections import OrderedDict

import numpy as np

from nsCamera.sensors.sensorBase import sensorBase
from nsCamera.utils.misc import flattenlist


class daedalus(sensorBase):
    specwarn = ""
    minframe = 0  # fixed value for sensor
    maxframe = 2  # fixed value for sensor
    maxwidth = 512  # fixed value for sensor
    maxheight = 1024  # fixed value for sensor
    bytesperpixel = 2
    fpganumID = 2  # last nybble of FPGA_NUM
    detect = "DAEDALUS_DET"
    sensfam = "Daedalus"
    loglabel = "[Daedalus] "
    ZDT = False
    HFW = False
    firstframe = 0
    lastframe = 2
    nframes = 3
    width = 512
    height = 1024
    firstrow = 0
    lastrow = 1023
    interlacing = [0, 0]
    columns = 1
    padToFull = True
    toffset = -165.76  # default temperature sensor offset
    tslope = 81.36  # default temperature sensor slope

    def __init__(self, ca):
        self.caca = ca
        super(daedalus, self).__init__(ca)

        self.sens_registers = OrderedDict(
            {
                "HST_READBACK_A_LO": "018",
                "HST_READBACK_A_HI": "019",
                "HST_READBACK_B_LO": "01A",
                "HST_READBACK_B_HI": "01B",
                "HSTALLWEN_WAIT_TIME": "03F",
                "VRESET_HIGH_VALUE": "04A",
                "FRAME_ORDER_SEL": "04B",
                "EXT_PHI_CLK_SH0_ON": "050",
                "EXT_PHI_CLK_SH0_OFF": "051",
                "EXT_PHI_CLK_SH1_ON": "052",
                "EXT_PHI_CLK_SH1_OFF": "053",
                "EXT_PHI_CLK_SH2_ON": "054",
                "HST_TRIGGER_DELAY_DATA_LO": "120",
                "HST_TRIGGER_DELAY_DATA_HI": "121",
                "HST_PHI_DELAY_DATA": "122",
                "HST_EXT_CLK_HALF_PER": "129",
                "HST_COUNT_TRIG": "130",
                "HST_DELAY_EN": "131",
                "RSL_HFW_MODE_EN": "133",
                "RSL_ZDT_MODE_B_EN": "135",
                "RSL_ZDT_MODE_A_EN": "136",
                "BGTRIMA": "137",
                "BGTRIMB": "138",
                "COLUMN_TEST_EN": "139",
                "RSL_CONFIG_DATA_B0": "140",
                "RSL_CONFIG_DATA_B1": "141",
                "RSL_CONFIG_DATA_B2": "142",
                "RSL_CONFIG_DATA_B3": "143",
                "RSL_CONFIG_DATA_B4": "144",
                "RSL_CONFIG_DATA_B5": "145",
                "RSL_CONFIG_DATA_B6": "146",
                "RSL_CONFIG_DATA_B7": "147",
                "RSL_CONFIG_DATA_B8": "148",
                "RSL_CONFIG_DATA_B9": "149",
                "RSL_CONFIG_DATA_B10": "14A",
                "RSL_CONFIG_DATA_B11": "14B",
                "RSL_CONFIG_DATA_B12": "14C",
                "RSL_CONFIG_DATA_B13": "14D",
                "RSL_CONFIG_DATA_B14": "14E",
                "RSL_CONFIG_DATA_B15": "14F",
                "RSL_CONFIG_DATA_B16": "150",
                "RSL_CONFIG_DATA_B17": "151",
                "RSL_CONFIG_DATA_B18": "152",
                "RSL_CONFIG_DATA_B19": "153",
                "RSL_CONFIG_DATA_B20": "154",
                "RSL_CONFIG_DATA_B21": "155",
                "RSL_CONFIG_DATA_B22": "156",
                "RSL_CONFIG_DATA_B23": "157",
                "RSL_CONFIG_DATA_B24": "158",
                "RSL_CONFIG_DATA_B25": "159",
                "RSL_CONFIG_DATA_B26": "15A",
                "RSL_CONFIG_DATA_B27": "15B",
                "RSL_CONFIG_DATA_B28": "15C",
                "RSL_CONFIG_DATA_B29": "15D",
                "RSL_CONFIG_DATA_B30": "15E",
                "RSL_CONFIG_DATA_B31": "15F",
                "RSL_CONFIG_DATA_A0": "160",
                "RSL_CONFIG_DATA_A1": "161",
                "RSL_CONFIG_DATA_A2": "162",
                "RSL_CONFIG_DATA_A3": "163",
                "RSL_CONFIG_DATA_A4": "164",
                "RSL_CONFIG_DATA_A5": "165",
                "RSL_CONFIG_DATA_A6": "166",
                "RSL_CONFIG_DATA_A7": "167",
                "RSL_CONFIG_DATA_A8": "168",
                "RSL_CONFIG_DATA_A9": "169",
                "RSL_CONFIG_DATA_A10": "16A",
                "RSL_CONFIG_DATA_A11": "16B",
                "RSL_CONFIG_DATA_A12": "16C",
                "RSL_CONFIG_DATA_A13": "16D",
                "RSL_CONFIG_DATA_A14": "16E",
                "RSL_CONFIG_DATA_A15": "16F",
                "RSL_CONFIG_DATA_A16": "170",
                "RSL_CONFIG_DATA_A17": "171",
                "RSL_CONFIG_DATA_A18": "172",
                "RSL_CONFIG_DATA_A19": "173",
                "RSL_CONFIG_DATA_A20": "174",
                "RSL_CONFIG_DATA_A21": "175",
                "RSL_CONFIG_DATA_A22": "176",
                "RSL_CONFIG_DATA_A23": "177",
                "RSL_CONFIG_DATA_A24": "178",
                "RSL_CONFIG_DATA_A25": "179",
                "RSL_CONFIG_DATA_A26": "17A",
                "RSL_CONFIG_DATA_A27": "17B",
                "RSL_CONFIG_DATA_A28": "17C",
                "RSL_CONFIG_DATA_A29": "17D",
                "RSL_CONFIG_DATA_A30": "17E",
                "RSL_CONFIG_DATA_A31": "17F",
            }
        )

        self.sens_subregisters = [

            ("HST_MODE", "HS_TIMING_CTL", 0, 1, True),
            ("SLOWREADOFF_0", "CTRL_REG", 4, 1, True),
            ("SLOWREADOFF_1", "CTRL_REG", 5, 1, True),
            ("MANSHUT_MODE", "CTRL_REG", 8, 1, True),
            ("INTERLACING_EN", "CTRL_REG", 9, 1, True),
            ("HFW", "RSL_HFW_MODE_EN", 0, 1, True),
            ("ZDT_A", "RSL_ZDT_MODE_A_EN", 0, 1, True),
            ("ZDT_B", "RSL_ZDT_MODE_B_EN", 0, 1, True),
            ("HST_DEL_EN", "HST_DELAY_EN", 0, 1, True),
            ("PHI_DELAY_A", "HST_PHI_DELAY_DATA", 9, 10, True),
            ("PHI_DELAY_B", "HST_PHI_DELAY_DATA", 29, 10, True),
            # Assume that daedalus is not to be used with v1 board
            ("VRESET_HIGH", "VRESET_HIGH_VALUE", 15, 16, True),

            ("STAT_SH0RISEUR", "STAT_REG", 3, 1, False),
            ("STAT_SH0FALLUR", "STAT_REG", 4, 1, False),
            ("STAT_RSLNALLWENA", "STAT_REG", 12, 1, False),
            ("STAT_RSLNALLWENB", "STAT_REG", 15, 1, False),
            # ("STAT_CONFIGHSTDONE", "STAT_REG", 16, 1, False),
        ]

    # TODO: add warning if daedalus and v1 board are together
    def sensorSpecific(self):
        """
        Returns:
            list of tuples, (Sensor-specific register, default setting)
        """
        return [
            ("FPA_FRAME_INITIAL", "00000000"),
            ("FPA_FRAME_FINAL", "00000002"),
            ("FPA_ROW_INITIAL", "00000000"),
            ("FPA_ROW_FINAL", "000003FF"),
            ("HS_TIMING_DATA_ALO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
            ("HS_TIMING_DATA_AHI", "00000000"),
            ("HS_TIMING_DATA_BLO", "00006666"),
            ("HS_TIMING_DATA_BHI", "00000000"),
            ("FRAME_ORDER_SEL", "00000000"),
            ("RSL_HFW_MODE_EN", "00000000"),
            ("RSL_ZDT_MODE_B_EN", "00000000"),
            ("RSL_ZDT_MODE_A_EN", "00000000"),
            ("RSL_CONFIG_DATA_B0", "00000000"),
            ("RSL_CONFIG_DATA_B1", "00000000"),
            ("RSL_CONFIG_DATA_B2", "00000000"),
            ("RSL_CONFIG_DATA_B3", "00000000"),
            ("RSL_CONFIG_DATA_B4", "00000000"),
            ("RSL_CONFIG_DATA_B5", "00000000"),
            ("RSL_CONFIG_DATA_B6", "00000000"),
            ("RSL_CONFIG_DATA_B7", "00000000"),
            ("RSL_CONFIG_DATA_B8", "00000000"),
            ("RSL_CONFIG_DATA_B9", "00000000"),
            ("RSL_CONFIG_DATA_B10", "00000000"),
            ("RSL_CONFIG_DATA_B11", "00000000"),
            ("RSL_CONFIG_DATA_B12", "00000000"),
            ("RSL_CONFIG_DATA_B13", "00000000"),
            ("RSL_CONFIG_DATA_B14", "00000000"),
            ("RSL_CONFIG_DATA_B15", "00000000"),
            ("RSL_CONFIG_DATA_B16", "00000000"),
            ("RSL_CONFIG_DATA_B17", "00000000"),
            ("RSL_CONFIG_DATA_B18", "00000000"),
            ("RSL_CONFIG_DATA_B19", "00000000"),
            ("RSL_CONFIG_DATA_B20", "00000000"),
            ("RSL_CONFIG_DATA_B21", "00000000"),
            ("RSL_CONFIG_DATA_B22", "00000000"),
            ("RSL_CONFIG_DATA_B23", "00000000"),
            ("RSL_CONFIG_DATA_B24", "00000000"),
            ("RSL_CONFIG_DATA_B25", "00000000"),
            ("RSL_CONFIG_DATA_B26", "00000000"),
            ("RSL_CONFIG_DATA_B27", "00000000"),
            ("RSL_CONFIG_DATA_B28", "00000000"),
            ("RSL_CONFIG_DATA_B29", "00000000"),
            ("RSL_CONFIG_DATA_B30", "00000000"),
            ("RSL_CONFIG_DATA_B31", "00000000"),
            ("RSL_CONFIG_DATA_A0", "00000000"),
            ("RSL_CONFIG_DATA_A1", "00000000"),
            ("RSL_CONFIG_DATA_A2", "00000000"),
            ("RSL_CONFIG_DATA_A3", "00000000"),
            ("RSL_CONFIG_DATA_A4", "00000000"),
            ("RSL_CONFIG_DATA_A5", "00000000"),
            ("RSL_CONFIG_DATA_A6", "00000000"),
            ("RSL_CONFIG_DATA_A7", "00000000"),
            ("RSL_CONFIG_DATA_A8", "00000000"),
            ("RSL_CONFIG_DATA_A9", "00000000"),
            ("RSL_CONFIG_DATA_A10", "00000000"),
            ("RSL_CONFIG_DATA_A11", "00000000"),
            ("RSL_CONFIG_DATA_A12", "00000000"),
            ("RSL_CONFIG_DATA_A13", "00000000"),
            ("RSL_CONFIG_DATA_A14", "00000000"),
            ("RSL_CONFIG_DATA_A15", "00000000"),
            ("RSL_CONFIG_DATA_A16", "00000000"),
            ("RSL_CONFIG_DATA_A17", "00000000"),
            ("RSL_CONFIG_DATA_A18", "00000000"),
            ("RSL_CONFIG_DATA_A19", "00000000"),
            ("RSL_CONFIG_DATA_A20", "00000000"),
            ("RSL_CONFIG_DATA_A21", "00000000"),
            ("RSL_CONFIG_DATA_A22", "00000000"),
            ("RSL_CONFIG_DATA_A23", "00000000"),
            ("RSL_CONFIG_DATA_A24", "00000000"),
            ("RSL_CONFIG_DATA_A25", "00000000"),
            ("RSL_CONFIG_DATA_A26", "00000000"),
            ("RSL_CONFIG_DATA_A27", "00000000"),
            ("RSL_CONFIG_DATA_A28", "00000000"),
            ("RSL_CONFIG_DATA_A29", "00000000"),
            ("RSL_CONFIG_DATA_A30", "00000000"),
            ("RSL_CONFIG_DATA_A31", "00000000"),
            ("HST_TRIGGER_DELAY_DATA_LO", "00000000"),
            ("HST_TRIGGER_DELAY_DATA_HI", "00000000"),
            ("HST_PHI_DELAY_DATA", "00000000"),
            ("SLOWREADOFF_0", "0"),
            ("SLOWREADOFF_1", "0"),
        ]

    def setInterlacing(self, ifactor=None, side=None):
        """
        Sets interlacing factor. NOTE: if called directly when HFW or ZDT mode is active,
        this will disengage those modes automatically. If hemispheres have different
        factors when the image is acquired, the resulting frames are separated into
        half-width images

        Args:
            ifactor: number of interlaced lines (generates ifactor + 1 images per frame)
              defaults to 0 (no interlacing)
            side: identify particular hemisphere (A or B) to control. If left blank,
              control both hemispheres

        Returns:
            integer: active interlacing factor (unchanged if error)
        """
        logging.debug(self.logdebug + "setInterlacing; ifactor = " + str(ifactor))
        if ifactor is None:
            ifactor = 0
        if (
            not isinstance(ifactor, int)
            or ifactor < 0
            or ifactor > (self.maxheight - 1)
        ):
            err = (
                self.logerr + "invalid interlacing factor submitted. "
                "Interlacing remains unchanged. "
            )
            logging.error(err)
            return self.interlacinginterlacing
        if self.HFWHFW:
            logging.warning(
                self.logwarn + "HFW mode will be disengaged because of new "
                "interlacing setting "
            )
            self.setHighFullWellsetHighFullWell(False)
        if self.ZDTZDT:
            logging.warning(
                self.logwarn + "ZDT mode will be disengaged because of new "
                "interlacing setting "
            )
            self.setZeroDeadTimesetZeroDeadTime(False)
        if ifactor == 0:
            bitscheme = self.maxheight * [0]
            # deactivating one side shouldn't turn off enable for both sides
            # TODO: is it a problem if sides are set separately, so interlacing is zero
            #     but still enabled?
            if side is None:
                self.caca.setSubregister("INTERLACING_EN", "0")
        else:
            pattern = [0] + ifactor * [1]
            reps = 1 + self.maxheight // (ifactor + 1)
            bitscheme = (reps * pattern)[0 : self.maxheight]
            self.caca.setSubregister("INTERLACING_EN", "1")
        err = ""
        for regnum in range(32):
            regbits = bitscheme[32 * regnum : 32 * (regnum + 1)]
            logging.debug(self.logdebug + "regbits = " + str(regbits))
            # generated pattern is reverse order from placement in register (element 0
            #   of the list is the LSB of the register)
            bitsrev = regbits[::-1]
            s = [str(i) for i in bitsrev]
            b = "".join(s)  # assemble as binary number for processing
            hexval = "%x" % int(b, 2)
            val = hexval.zfill(8)
            err0 = ""
            err1 = ""
            if side is None or side.lower() == "a":
                lname = "RSL_CONFIG_DATA_A" + str(regnum)
                err1, _ = self.caca.setRegister(lname, val)
                self.interlacinginterlacing[1] = ifactor
            if side is None or side.lower() == "b":
                rname = "RSL_CONFIG_DATA_B" + str(regnum)
                err0, _ = self.caca.setRegister(rname, val)
                self.interlacinginterlacing[0] = ifactor
            err = err + err0 + err1
        if err:
            logging.error(self.logerr + "interlacing may not be set correctly: " + err)
        logging.info(self.loginfo + "Interlacing set to " + str(self.interlacinginterlacing))
        if self.interlacinginterlacing[0] == self.interlacinginterlacing[1]:
            self.columnscolumns = 1
        else:
            self.columnscolumns = 2
        return self.interlacinginterlacing

    def setHighFullWell(self, flag):
        """
        Activates High Full Well mode. All frames are acquired simultaneously. Zero Dead
          Time mode and interlacing will be automatically deactivated and column number
          will be reset to 0. NOTE: after deactivating HFW, the board remains in
          uninterlaced mode (interlacing = 0)

        Args:
            flag: True to activate HFW mode, False to deactivate

        Returns:
            Error message
        """
        logging.debug(self.logdebug + "setHighFullWell; flag = " + str(flag))
        err0 = ""
        if flag:
            if self.ZDTZDT:
                logging.warning(
                    self.logwarn + "ZDT mode will be disengaged because of HFW "
                    "setting "
                )
                err0 = self.setZeroDeadTimesetZeroDeadTime(False)
            err1, _ = self.caca.setSubregister("HFW", "1")
            self.HFWHFW = False  # preclude HFW deactivation message in setInterlacing
            self.setInterlacingsetInterlacing(0)
            self.HFWHFW = True
            logging.info(self.loginfo + "High Full Well mode active")
        else:
            self.HFWHFW = False
            err1, _ = self.caca.setSubregister("HFW", "0")
            logging.info(self.loginfo + "High Full Well mode inactivate")
        err = err0 + err1
        if err:
            logging.error(self.logerr + "HFW option may not be set correctly ")
        return err

    def setZeroDeadTime(self, flag=True, side=None):
        """
        Activates Zero Dead Time mode. Even rows follow the assigned HST schedule; odd
          rows are acquired while the 'shutter' for the even rows are closed. High Full
          Well mode and interlacing will be automatically deactivated.
        *NOTE* after deactivating ZDT, the board reverts to uninterlaced mode
          (interlacing = 0)

        Args:
            flag: True to activate ZDT mode, False to deactivate
            side: identify particular hemisphere (A or B) to control. If left blank,
              control both hemispheres

        Returns:
            Error message
        """
        logging.debug(self.logdebug + "setZeroDeadTime; flag = " + str(flag))
        err0 = ""
        err1 = ""
        err2 = ""
        if flag:
            if self.HFWHFW:
                logging.warning(
                    self.logwarn + "HFW mode will be disengaged because of ZDT "
                    "setting "
                )
                err0 = self.setHighFullWellsetHighFullWell(False)
            if side is None or side.lower() == "a":
                err2, _ = self.caca.setSubregister("ZDT_A", "1")
                self.interlacinginterlacing[0] = 1
            if side is None or side.lower() == "b":
                err1, _ = self.caca.setSubregister("ZDT_B", "1")
                self.interlacinginterlacing[1] = 1
            # self.ZDT = False  # preclude ZDT deactivation message in setInterlacing
            # if self.interlacing != [0, 0]:
            #     self.setInterlacing(0)
            # TODO: need to handle flags when ZDT active for just one side
            self.ZDTZDT = True
            logging.info(
                self.loginfo + "Zero Dead Time mode active; actual interlacing = 1"
            )
        else:
            self.ZDTZDT = False
            if side is None or side.lower() == "a":
                err2, _ = self.caca.setSubregister("ZDT_A", "0")
            if side is None or side.lower() == "b":
                err1, _ = self.caca.setSubregister("ZDT_B", "0")
            self.setInterlacingsetInterlacing(0)
            logging.info(self.loginfo + "Zero Dead Time mode inactivate")
        err = err0 + err1 + err2
        if err:
            logging.error(self.logerr + "ZDT option may not be set correctly ")
        return err

    def selectOscillator(self, osc=None):
        """
        Selects oscillator to control sensor timing
        Args:
            osc: 500|100|'ring'|external', defaults to 500 MHz

        Returns:
            error message as string
        """
        logging.info(self.loginfo + "selectOscillator; osc = " + str(osc))
        if osc is None:
            osc = 500
        osc = str(osc)
        if osc[:3] == "500":
            payload = "00"
        elif osc[:3] == "100":
            payload = "01"
        elif osc.upper()[:3] == "RIN":
            payload = "10"
        elif osc.upper()[:3] in ["EXT"]:
            payload = "11"
        else:
            err = (
                self.logerr + "selectOscillator: invalid parameter supplied. "
                "Oscillator selection is unchanged."
            )
            logging.error(err)
            return err
        self.caca.setSubregister("OSC_SELECT", payload)

    def setTriggerDelay(self, delay=0):
        """
        Use trigger delay timer. Actual delay is rounded down to multiple of .15 ns, up
          to a maximum delay of 6 ns

        Args:
            delay: trigger delay in ns

        Returns:
            String of errors, if any
        """
        logging.debug(self.logdebug + "setTriggerDelay; delay = " + str(delay))
        if (
            not (isinstance(delay, int) or isinstance(delay, float))
            or delay < 0
            or delay > 6
        ):
            err = (
                self.logerr + "invalid trigger delay submitted. Delay remains "
                "unchanged. "
            )
            logging.error(err)
            return err
        delayblocks = int(delay / 0.15)
        if delayblocks < 0:
            delayblocks = 0
        if delayblocks > 40:
            delayblocks = 40
        delayseq = (40 - delayblocks) * [0] + delayblocks * [1]
        seqstr = "".join(str(x) for x in delayseq)
        seqhex = "%x" % int(seqstr, 2)
        logging.debug(self.logdebug + "seqhex = " + str(seqhex))
        highpart = seqhex[-10:-8].zfill(8)
        lowpart = seqhex[-8:].zfill(8)
        self.caca.setSubregister("HST_DEL_EN", "1")
        err0, _ = self.caca.setRegister("HST_TRIGGER_DELAY_DATA_LO", lowpart)
        err1, _ = self.caca.setRegister("HST_TRIGGER_DELAY_DATA_HI", highpart)
        err2, _ = self.caca.setSubregister("HST_MODE", "1")
        delayed = delayblocks * 0.15
        logging.info(self.loginfo + "Actual trigger delay = " + str(delayed) + " ns")
        return err0 + err1 + err2

    def setPhiDelay(self, side=None, delay=0):
        """
        Use phi delay timer. Actual delay is rounded down to multiple of .15 ns, up to a
          maximum delay of 1.5 ns
        Args:
            side: hemisphere to delay; if None, delay both hemispheres
            delay: phi delay in ns

        Returns:
            String of errors, if any
        """
        logging.debug(self.logdebug + "setPhiDelay; delay = " + str(delay))
        if (
            not (isinstance(delay, int) or isinstance(delay, float))
            or delay < 0
            or delay > 1.5
        ):
            err = (
                self.logerr + "invalid phi delay submitted. Delay remains "
                "unchanged. "
            )
            logging.error(err)
            return err
        delayblocks = int(delay / 0.15)
        if delayblocks < 0:
            delayblocks = 0
        if delayblocks > 10:
            delayblocks = 10
        delayseq = (10 - delayblocks) * [0] + delayblocks * [1]
        seqstr = "".join(str(x) for x in delayseq)
        err1 = ""
        err2 = ""
        if side is None or side.upper() == "A":
            err1, _ = self.caca.setSubregister("PHI_DELAY_A", seqstr)
        if side is None or side.upper() == "B":
            err2, _ = self.caca.setSubregister("PHI_DELAY_B", seqstr)
        delayed = delayblocks * 0.15
        logging.info(self.loginfo + "Actual phi delay = " + str(delayed) + " ns")
        return err1 + err2

    def setExtClk(self, dilation=None, frequency=None):
        """
        Override the standard board clock with the external clock.
        Args:
            dilation: ratio of base frequency (500 MHz) to desired external clock
              frequency. Default is 25. Overridden if frequency parameter is provided
            frequency: Desired frequency for phi clock.
        Returns:
            error message as string
        """
        logging.debug(
            self.logdebug
            + "setExtClk; dilation = "
            + str(dilation)
            + "; frequency = "
            + str(frequency)
        )
        if not (isinstance(frequency, int) or isinstance(frequency, float)):
            err = (
                self.logerr
                + "invalid external clock frequency submitted. Clock is not "
                + "operating"
            )
            logging.error(err)
            return err
        self.caca.selectOscillator("external")
        if not dilation:
            dilation = 25
        if not frequency:
            frequency = 5e7 / float(dilation)
        count = 2e7 / float(frequency) - 1  # base phi clock is 20 MHz?
        if count < 0:
            count = 0
            warn = (
                self.logwarn
                + "external clock frequency exceeding maximum. Frequency set to "
                + "maximum (20 MHz)"
            )
            logging.warning(warn)
        if count > 0xFFFFFFFF:
            count = 0xFFFFFFFF
        counthex = hex(int(count))[2:].zfill(8)
        self.caca.setRegister("HST_EXT_CLK_HALF_PER", counthex)

    # TODO: enable exponential form for all large number inputs (accept floats)

    def setManualShutters(self, timing=None):
        """
        Legacy alias for setManualTiming()
        """
        self.setManualTimingsetManualTiming(timing)

    def setManualTiming(self, timing=None):
        """
        Manual shutter timing, five intervals given in nanoseconds, e.g.,
          [100,50,100,50,100] for frame 0 open for 100 ns, an interframe pause of 50 ns,
           frame 1 open for 100 ns, etc. Timing is set for both hemispheres.

        The actual timing is rounded down to the nearest multiple of 25 ns. (Each
          count = 25 ns. e.g., a request for 140 ns rounds down to a count of '5',
          which corresponds to 125 ns))
            - Minimum timing is 75 ns
            - Maximum is 25 * 2^30 ns (approximately 27 seconds)

        Args:
            timing: 5-element list in nanoseconds

        Returns:
            tuple (error string, response string from final message)
        """
        if timing is None:
            logging.info(
                self.loginfo
                + "No manual timing setting provided, defaulting to (100, 150, 100, "
                " 150, 100, 150, 100) for both hemispheres"
            )
            timing = [(100, 150, 100, 150, 100)]

        logging.info(self.loginfo + "Manual shutter sequence: " + str(timing))
        flattened = flattenlist(timing)
        if (
            len(flattened) != 5
            or not all(isinstance(x, (int, float)) for x in flattened)
            or not all(x >= 25 for x in flattened)
        ):
            err = self.logerr + "Invalid manual shutter timing list: " + str(timing)
            logging.error(err + "; timing settings unchanged")
            return err, "00000000"

        timecounts = [int(a // 25) for a in flattened]
        self.caca.sensmanual = timing
        self.caca.senstiming = {}  # clear HST settings from ca object

        control_messages = [
            ("MANSHUT_MODE", "1"),
            ("EXT_PHI_CLK_SH0_ON", "{0:#0{1}x}".format(timecounts[0], 10)[2:10]),
            ("EXT_PHI_CLK_SH0_OFF", "{0:#0{1}x}".format(timecounts[1], 10)[2:10]),
            ("EXT_PHI_CLK_SH1_ON", "{0:#0{1}x}".format(timecounts[2], 10)[2:10]),
            ("EXT_PHI_CLK_SH1_OFF", "{0:#0{1}x}".format(timecounts[3], 10)[2:10]),
            ("EXT_PHI_CLK_SH2_ON", "{0:#0{1}x}".format(timecounts[4], 10)[2:10]),
        ]
        return self.caca.submitMessages(control_messages, " setManualShutters: ")

    def getManualTiming(self):
        """
        Read off manual shutter timing settings
        Returns:
            list of manual timing intervals
        """
        timing = []
        for reg in [
            "EXT_PHI_CLK_SH0_ON",
            "EXT_PHI_CLK_SH0_OFF",
            "EXT_PHI_CLK_SH1_ON",
            "EXT_PHI_CLK_SH1_OFF",
            "EXT_PHI_CLK_SH2_ON",
        ]:
            _, reghex = self.caca.getRegister(reg)
            timing.append(25 * int(reghex, 16))
        return timing

    def getSensTemp(self, scale=None, offset=None, slope=None, dec=1):
        """
        Read temperature sensor located on the Daedalus sensor
        Args:
            scale: temperature scale to report (defaults to C, options are F and K)
            offset: offset of linear fit of sensor response (defaults to self.toffset)
            slope: slope of linear fit of sensor response (defaults to self.tslope)
            dec: round to 'dec' digits after the decimal point

        Returns:
            temperature as float on given scale, rounded to .1 degree
        """
        err, rval = self.caca.getMonV("MON_TSENSE_OUT", errflag=True)
        if err:
            logging.error(
                self.logerr + "unable to retrieve temperature information ("
                'getTemp), returning "0" '
            )
            return 0.0
        if offset is None:
            offset = self.toffset
        if slope is None:
            slope = self.tslope

        ctemp = offset + slope * rval
        if scale == "K":
            temp = round(ctemp + 273.15, dec)
        elif scale == "F":
            temp = round(1.8 * ctemp + 32, dec)
        else:
            temp = round(ctemp, dec)
        return temp

    def parseReadoff(self, frames, columns):
        """
        Parses frames from board into images
        Args:
            frames: list of data arrays (frames) returned from board
            columns: 1 (full width image) or 2 (hemispheres generate distinct images)
        Returns:
            list of data arrays (frames) reordered and deinterlaced
        """
        logging.debug(self.logdebug + "parseReadoff")
        w = self.width
        if hasattr(self, "ca"):  # TODO: this may no longer be necessary
            padIt = self.caca.padToFull
        else:
            padIt = self.padToFull
        if padIt:
            rows = self.maxheight
        else:
            rows = self.lastrow - self.firstrow + 1
        parsed = []
        for frame in frames:
            current = np.zeros((rows, w), dtype=np.uint16)
            mapped = np.zeros((rows, w), dtype=np.uint16)
            frame = frame.reshape(rows, w)

            for entry in range(int(w / 2)):
                col = 32 * (entry % 8) + entry // 8  # lookup from daedlookup.xls
                for row in range(rows):
                    current[row][col] = frame[row][2 * entry]
                    current[row][col + 256] = frame[row][2 * entry + 1]

            for row in range(rows):
                mapped[row][0:32] = current[row][320:352]
                mapped[row][32:64] = current[row][352:384]
                mapped[row][64:96] = current[row][192:224]
                mapped[row][96:128] = current[row][160:192]
                mapped[row][128:160] = current[row][256:288]
                mapped[row][160:192] = current[row][288:320]
                mapped[row][192:224] = current[row][416:448]
                mapped[row][224:256] = current[row][32:64]
                mapped[row][256:288] = current[row][128:160]
                mapped[row][288:320] = current[row][224:256]
                mapped[row][320:352] = current[row][384:416]
                mapped[row][352:384] = current[row][448:480]
                mapped[row][384:416] = current[row][480:512]
                mapped[row][416:448] = current[row][0:32]
                mapped[row][448:480] = current[row][64:96]
                mapped[row][480:512] = current[row][96:128]
            parsed.append(mapped)

        images = self.caca.partition(parsed, columns)
        flatimages = [flattenlist(x) for x in images]
        return flatimages

    def reportStatusSensor(self, statusbits, statusbits2):
        """
        Print status messages from sensor-specific bits of status register or object
          status flags

        Args:
            statusbits: result of checkStatus()
            statusbits2: result of checkStatus2()
        """
        if int(statusbits[3]):
            print(self.loginfo + "SH0_rise_B_edge detected")
        if int(statusbits[4]):
            print(self.loginfo + "SH0_fall_B_edge detected")
        if int(statusbits[12]):
            print(self.loginfo + "RSLNALLWENB detected")
        if int(statusbits[15]):
            print(self.loginfo + "RSLNALLWENA detected")
        if self.HFWHFW:
            print(self.loginfo + "High Full Well mode active")
        # TODO: handle two hemispheres for ZDT
        elif self.ZDTZDT:
            print(self.loginfo + "Zero Dead Time mode active")
        elif self.interlacinginterlacing != [0, 0]:
            print(
                "{loginfo}Interlacing active: {interlacing}".format(
                    loginfo=self.loginfo, interlacing=str(self.interlacinginterlacing)
                )
            )
        if self.caca.sensmanual == []:
            print(
                "{loginfo}High-speed timing: A:{Atiming}, B:{Btiming}".format(
                    loginfo=self.loginfo,
                    Atiming=self.getTiming(side="A", actual=True),
                    Btiming=self.getTiming(side="B", actual=True),
                )
            )
        else:
            print(
                "{loginfo}Manual timing set to {timing}".format(
                    loginfo=self.loginfo, timing=self.getManualTiminggetManualTiming()
                )
            )


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.35 C:/Users/hill35/git/camera_python/nsCamera/sensors/icarus.py File Reference

Classes

- class nsCamera.sensors.icarus.icarus

Namespaces

- namespace nsCamera
- namespace nsCamera.sensors
- namespace nsCamera.sensors.icarus

7.36 icarus.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Parameters and functions specific to the icarus two-frame sensor

***Do not use this file as a template for new code development***

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
```

```
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import logging
from collections import OrderedDict

from nsCamera.sensors.sensorBase import sensorBase


class icarus(sensorBase):
    specwarn = " and the use of the Icarus model 1 sensor"
    minframe = 1  # fixed value for sensor
    maxframe = 2  # fixed value for sensor
    # WARNING: the camera will always 'acquire' four frames, but will only generate
    #   images for the middle two; HST and manual shutters will manage all four
    #   frames
    maxwidth = 512  # fixed value for sensor
    maxheight = 1024  # fixed value for sensor
    bytesperpixel = 2
    icarustype = 1  # 2-frame version
    fpganumID = 1  # last nybble of FPGA_NUM
    detect = "ICARUS_DET"
    sensfam = "Icarus"
    loglabel = "[Icarus1] "
    firstframe = 1
    lastframe = 2
    nframes = 2
    width = 512
    height = 1024
    firstrow = 0
    lastrow = 1023
    interlacing = [0, 0]  # N/A for icarus
    columns = 1
    padToFull = True

    def __init__(self, ca):
        self.caca = ca
        super(icarus, self).__init__(ca)

        self.sens_registers = OrderedDict(
            {
                "VRESET_WAIT_TIME": "03E",
                "ICARUS_VER_SEL": "041",
                "VRESET_HIGH_VALUE": "04A",
                "MISC_SENSOR_CTL": "04C",
                "MANUAL_SHUTTERS_MODE": "050",
                "W0_INTEGRATION": "051",
                "W0_INTERFRAME": "052",
                "W1_INTEGRATION": "053",
                "W1_INTERFRAME": "054",
                "W2_INTEGRATION": "055",
                "W2_INTERFRAME": "056",
                "W3_INTEGRATION": "057",
                "W0_INTEGRATION_B": "058",
                "W0_INTERFRAME_B": "059",
                "W1_INTEGRATION_B": "05A",
                "W1_INTERFRAME_B": "05B",
                "W2_INTEGRATION_B": "05C",
                "W2_INTERFRAME_B": "05D",
                "W3_INTEGRATION_B": "05E",
                "TIME_ROW_DCD": "05F",
            }
        )

        self.sens_subregisters = [

            ("MANSHUT_MODE", "MANUAL_SHUTTERS_MODE", 0, 1, True),
            ("REVREAD", "CTRL_REG", 4, 1, True),
            ("PDBIAS_LOW", "CTRL_REG", 6, 1, True),
            ("ROWDCD_CTL", "CTRL_REG", 7, 1, True),
            ("ACCUMULATION_CTL", "MISC_SENSOR_CTL", 0, 1, True),
            ("HST_TST_ANRST_EN", "MISC_SENSOR_CTL", 1, 1, True),
            ("HST_TST_BNRST_EN", "MISC_SENSOR_CTL", 2, 1, True),
            ("HST_TST_ANRST_IN", "MISC_SENSOR_CTL", 3, 1, True),
            ("HST_TST_BNRST_IN", "MISC_SENSOR_CTL", 4, 1, True),
            ("HST_PXL_RST_EN", "MISC_SENSOR_CTL", 5, 1, True),
            ("HST_CONT_MODE", "MISC_SENSOR_CTL", 6, 1, True),
            ("COL_DCD_EN", "MISC_SENSOR_CTL", 7, 1, True),
            ("COL_READOUT_EN", "MISC_SENSOR_CTL", 8, 1, True),
            ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True),

            ("STAT_W3TOPAEDGE1", "STAT_REG", 3, 1, False),
            ("STAT_W3TOPBEDGE1", "STAT_REG", 4, 1, False),
            ("STAT_HST_ALL_W_EN_DETECTED", "STAT_REG", 12, 1, False),
            ("PDBIAS_UNREADY", "STAT_REG2", 5, 1, False),
        ]

        if self.caca.boardname == "llnl_v1":
            self.sens_subregisters.append(
                ("VRESET_HIGH", "VRESET_HIGH_VALUE", 7, 8, True)
            )
        else:
            self.sens_subregisters.extend(
                [
                    ("VRESET_HIGH", "VRESET_HIGH_VALUE", 15, 16, True),
                    ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True),
                ]
            )
            self.sens_registers.update({"DELAY_ASSERTION_ROWDCD_EN": "04F"})

    def checkSensorVoltStat(self):
        """
        Checks register tied to sensor select jumpers to confirm match with sensor
          object

        Returns:
            boolean, True if jumpers select for Icarus sensor
        """
        logging.debug(self.logdebug + "checkSensorVoltStat")
        err, status = self.caca.getSubregister("ICARUS_DET")
        if err:
            logging.error(self.logerr + "unable to confirm sensor status")
            return False
        if not int(status):
            logging.error(self.logerr + "Icarus sensor not detected")
            return False
        return True

    def sensorSpecific(self):
        """
        Returns:
            list of tuples, (Sensor-specific register, default setting)
        """
        icarussettings = [
            ("ICARUS_VER_SEL", "00000001"),
            ("FPA_FRAME_INITIAL", "00000001"),
            ("FPA_FRAME_FINAL", "00000002"),
            ("FPA_ROW_INITIAL", "00000000"),
            ("FPA_ROW_FINAL", "000003FF"),
            ("VRESET_WAIT_TIME", "000927C0"),
            ("HS_TIMING_DATA_BHI", "00000000"),
            ("HS_TIMING_DATA_BLO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
            ("HS_TIMING_DATA_AHI", "00000000"),
            ("HS_TIMING_DATA_ALO", "00006666"),
        ]
        if self.caca.boardname == "llnl_v1":
            icarussettings.append(
                ("VRESET_HIGH_VALUE", "000000D5")  # 3.3 V (FF = 3.96)
            )
        else:
            icarussettings.append(("VRESET_HIGH_VALUE", "0000FFFF"))
        return icarussettings


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.37 C:/Users/hill35/git/camera_python/nsCamera/sensors/icarus2.py File Reference

Classes

- class nsCamera.sensors.icarus2.icarus2

Namespaces

- namespace nsCamera
- namespace nsCamera.sensors
- namespace nsCamera.sensors.icarus2

7.38 icarus2.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Parameters and functions specific to the four-frame icarus2 sensor

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from collections import OrderedDict

from nsCamera.sensors.sensorBase import sensorBase


class icarus2(sensorBase):
    specwarn = ""
    minframe = 0  # fixed value for sensor
    maxframe = 3  # fixed value for sensor
    maxwidth = 512  # fixed value for sensor
    maxheight = 1024  # fixed value for sensor
    bytesperpixel = 2
    icarustype = 0  # 4-frame version
    fpganumID = 1  # last nybble of FPGA_NUM
    detect = "ICARUS_DET"
    sensfam = "Icarus"
    loglabel = "[Icarus2] "
    firstframe = 0
    lastframe = 3
    nframes = 4
    width = 512
    height = 1024
    firstrow = 0
    lastrow = 1023
    interlacing = [0, 0]  # N/A for icarus
    columns = 1
    padToFull = True

    def __init__(self, ca):
        self.caca = ca
        super(icarus2, self).__init__(ca)

        self.sens_registers = OrderedDict(
            {
                "VRESET_WAIT_TIME": "03E",
                "ICARUS_VER_SEL": "041",
                "MISC_SENSOR_CTL": "04C",
                "MANUAL_SHUTTERS_MODE": "050",
                "W0_INTEGRATION": "051",
                "W0_INTERFRAME": "052",
                "W1_INTEGRATION": "053",
                "W1_INTERFRAME": "054",
                "W2_INTEGRATION": "055",
                "W2_INTERFRAME": "056",
                "W3_INTEGRATION": "057",
                "W0_INTEGRATION_B": "058",
                "W0_INTERFRAME_B": "059",
                "W1_INTEGRATION_B": "05A",
                "W1_INTERFRAME_B": "05B",
                "W2_INTEGRATION_B": "05C",
                "W2_INTERFRAME_B": "05D",
                "W3_INTEGRATION_B": "05E",
                "TIME_ROW_DCD": "05F",
            }
        )

        self.sens_subregisters = [

            ("MANSHUT_MODE", "MANUAL_SHUTTERS_MODE", 0, 1, True),
            ("REVREAD", "CTRL_REG", 4, 1, True),
            ("PDBIAS_LOW", "CTRL_REG", 6, 1, True),
            ("ROWDCD_CTL", "CTRL_REG", 7, 1, True),
            ("ACCUMULATION_CTL", "MISC_SENSOR_CTL", 0, 1, True),
            ("HST_TST_ANRST_EN", "MISC_SENSOR_CTL", 1, 1, True),
            ("HST_TST_BNRST_EN", "MISC_SENSOR_CTL", 2, 1, True),
            ("HST_TST_ANRST_IN", "MISC_SENSOR_CTL", 3, 1, True),
            ("HST_TST_BNRST_IN", "MISC_SENSOR_CTL", 4, 1, True),
            ("HST_PXL_RST_EN", "MISC_SENSOR_CTL", 5, 1, True),
            ("HST_CONT_MODE", "MISC_SENSOR_CTL", 6, 1, True),
            ("COL_DCD_EN", "MISC_SENSOR_CTL", 7, 1, True),
            ("COL_READOUT_EN", "MISC_SENSOR_CTL", 8, 1, True),

            ("STAT_W3TOPAEDGE1", "STAT_REG", 3, 1, False),
            ("STAT_W3TOPBEDGE1", "STAT_REG", 4, 1, False),
            ("STAT_HST_ALL_W_EN_DETECTED", "STAT_REG", 12, 1, False),
            ("PDBIAS_UNREADY", "STAT_REG2", 5, 1, False),
        ]

        if self.caca.boardname == "llnl_v4":
            self.sens_subregisters.append(
                ("READOFF_DELAY_EN", "TRIGGER_CTL", 4, 1, True)
            )
            self.sens_registers.update({"DELAY_ASSERTION_ROWDCD_EN": "04F"})

    # TODO: clean up static methods
    def sensorSpecific(self):
        """
        Returns:
            list of tuples, (Sensor-specific register, default setting)
        """
        return [
            ("ICARUS_VER_SEL", "00000000"),
            ("FPA_FRAME_INITIAL", "00000000"),
            ("FPA_FRAME_FINAL", "00000003"),
            ("FPA_ROW_INITIAL", "00000000"),
            ("FPA_ROW_FINAL", "000003FF"),
            ("HS_TIMING_DATA_BHI", "00000000"),
            ("HS_TIMING_DATA_BLO", "00006666"),  # 0db6 = 2-1; 6666 = 2-2
            ("HS_TIMING_DATA_AHI", "00000000"),
            ("HS_TIMING_DATA_ALO", "00006666"),
        ]


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
```

```
"""
```

7.39 C:/Users/hill35/git/camera_python/nsCamera/sensors/sensorBase.py File Reference

Classes

- class nsCamera.sensors.sensorBase.sensorBase

Namespaces

- namespace nsCamera
- namespace nsCamera.sensors
- namespace nsCamera.sensors.sensorBase

7.40 sensorBase.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Superclass for nsCamera sensors

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""
import itertools
import logging
import numbers

from nsCamera.utils.misc import flattenlist, makeLogLabels


class sensorBase(object):
    """
    Base class for sensors. 'Virtual' methods below default to Icarus behavior.
    daedalus.py overrides these methods as necessary
    """

    def __init__(self, camassem):
        self.ca = camassem
        # skip board settings if no board object exists
        if hasattr(self.ca, "board"):
            self.init_board_specific()

        (
            self.logcrit,
            self.logerr,
            self.logwarn,
            self.loginfo,
            self.logdebug,
        ) = makeLogLabels(self.ca.logtag, self.loglabel)

        # skip assignment if no comms object exists
        if hasattr(self.ca, "comms"):
            self.ca.comms.payloadsize = (
                self.width * self.height * self.nframes * self.bytesperpixel
            )

        logging.info(self.loginfo + "Initializing sensor object")

    def init_board_specific(self):
        """Initialize aliases and subregisters specific to the current board and sensor."""

        if self.ca.sensorname == "icarus" or self.ca.sensorname == "icarus2":
            self.ca.board.subreg_aliases = self.ca.board.icarus_subreg_aliases
            self.ca.board.monitor_controls = self.ca.board.icarus_monitor_controls
        else:
            self.ca.board.subreg_aliases = self.ca.board.daedalus_subreg_aliases
            self.ca.board.monitor_controls = self.ca.board.daedalus_monitor_controls

    # TODO: Check if 'jumpers' still apply for newer boards
    def checkSensorVoltStat(self):
        """
        Checks register tied to sensor select jumpers to confirm match with sensor
        object

        Returns:
            boolean, True if jumpers select for Icarus sensor
        """
        logging.debug(self.logdebug + "checkSensorVoltStat")
        err, status = self.ca.getSubregister(self.detect)
        if err:
            logging.error(self.logerr + "Unable to confirm sensor status")
            return False
        if not int(status):
            logging.error(self.logerr + self.sensfam + " sensor not detected")
            return False
        return True

    def setInterlacing(self, ifactor):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py

        Returns:
            integer 0
        """
        if ifactor:
            logging.warning(
                self.logwarn + "Interlacing is not supported by Icarus sensors. "
            )
        return 0

    def setHighFullWell(self, flag):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if flag:
            logging.warning(
                self.logwarn + "HighFullWell mode is not supported by Icarus sensors. "
            )

    def setZeroDeadTime(self, flag):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if flag:
            logging.warning(
                self.logwarn + "ZeroDeadTime mode is not supported by Icarus sensors. "
            )

    def setTriggerDelay(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "Trigger Delay is not supported by Icarus sensors. "
            )

    def setPhiDelay(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "Phi Delay is not supported by Icarus sensors. "
            )

    def setExtClk(self, delay):
        """
        Virtual function; feature is not implemented on Icarus
        Overridden in daedalus.py
        """
        if delay:
            logging.warning(
                self.logwarn + "External Phi Clock is not supported by Icarus sensors. "
            )

    # TODO: update docs to reflect all sensors
    # TODO: remove default timing?
    # TODO: double-check o+c>20 ns in doc block
    def setTiming(self, side="AB", sequence=None, delay=0):
        """
        Sets timing registers based on 'sequence.' Requesting (0,0) timing will clear the
          timing register.
        *WARNING* if the entire sequence does not fit into the 40-bit register space,
          then the actual timings generated may differ from those requested. If the
          timing sequence fits only once into the register space (i.e., for a single
          frame, open + closed > 20 ns), then the actual timing will be (n, 40-n),
          irrespective of the setting of second parameter, e.g. (35,1) will actually
          result in (35,5) timing.
        *NOTE* Icarus sensors generally cannot use 1 ns timing, so all values for these
          devices (besides the delay) should be at least 2 ns

        Args:
            side: Hemisphere 'A', 'B', 'AB'
            sequence: two-element tuple of timing durations in ns, e.g., '(5,2)'
            delay: initial delay in ns (1 ns delay is acceptable)

        Returns:
            tuple (error string, 10-character hexadecimal representation of timing
              sequence)
        """
        logging.info(
            "{}setTiming; side={}, sequence={}, delay={}".format(
                self.loginfo, side, sequence, delay
            )
        )
        if sequence is None:
            sequence = (3, 2)
        if delay is None:
            delay = 0
        logging.info(
            self.loginfo
            + "HST side "
            + side.upper()
            + ": "
            + str(sequence)
            + "; delay = "
            + str(delay)
        )
        err = ""
        if len(sequence) != 2:
            err = (
                self.logerr
                + "Invalid sequence setting for side: "
                + side
                + "; timing settings are unchanged"
            )
            logging.error(err)
            return err, "0000000000"
        if side.upper() == "AB":
            err1, _ = self.setTiming(side="A", sequence=sequence, delay=delay)
            err2, full40hex = self.setTiming(side="B", sequence=sequence, delay=delay)
            return err1 + err2, full40hex
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            err = (
                self.logerr
                + "setTiming: Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            logging.error(err)
            return err, "0000000000"
        if (sequence[0] + sequence[1]) + delay > 40:
            err = (
                self.logerr
                + "setTiming:  Timing sequence is too long to be implemented; "
                "timing settings unchanged "
            )
            logging.error(err)
            return err, "0000000000"

        self.ca.senstiming[side.upper()] = (sequence, delay)
        self.ca.sensmanual = []  # clear manual settings from ca

        full40 = [0] * 40
        bitlist = []
        flag = 1
        sequence = sequence[:2]  # TODO: is this redundant?
        for a in sequence:
            add = [flag] * a
            bitlist += add
            if flag:
                flag = 0
            else:
                flag = 1
        # automatically truncates sequence to 39 characters
        logging.debug(self.logdebug + "bitlist = " + str(bitlist))
        if bitlist:  # skip this if timing is [0,0]
            reversedlist = bitlist[39::-1]
            trunclist = reversedlist[:]
            while trunclist[0] == 0:
                trunclist.pop(0)
            # fullrepeat counts open/closed cycles, doesn't include final frame
            fullrepeats = (40 - len(trunclist) - delay) // len(reversedlist)
            logging.debug(self.logdebug + "fullrepeats = " + str(fullrepeats))
            # Pattern from sequence repeated to fit inside 40 bits
            repeated = trunclist + reversedlist * fullrepeats
            full40[-(len(repeated) + delay + 1) : -(delay + 1)] = repeated
        else:
            logging.warning(self.logwarn + "setTiming: all-zero timing supplied")
            fullrepeats = self.nframes
        full40bin = "".join(str(x) for x in full40)
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        full40hex = "%x" % int(full40bin, 2)
        logging.debug(self.logdebug + "full40hex = " + str(full40hex))
        highpart = full40hex[-10:-8].zfill(8)
        lowpart = full40hex[-8:].zfill(8)
        err0, _ = self.ca.setRegister(lowreg, lowpart)
        err1, _ = self.ca.setRegister(highreg, highpart)
        err2, _ = self.ca.setSubregister("MANSHUT_MODE", "0")
        err3, _ = self.ca.setSubregister("HST_MODE", "1")
        err = err0 + err1 + err2 + err3
        if err:
            logging.error(
                self.logerr + "setTiming: Timing may not have been set correctly"
            )
        if fullrepeats < self.nframes - 1:
            actual = self.getTiming(side, actual=True)
            if self.fpganumID == 2:
                expected = [delay] + 2 * list(sequence) + [sequence[0]]
            else:
                expected = [delay] + 3 * list(sequence) + [sequence[0]]
            if actual != expected:
                logging.warning(
                    self.logwarn
                    + "setTiming: Due to sequence length"
                    + self.specwarn
                    + ", the actual timing "
                    "sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0])
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        elif self.ca.sensorname == "icarus":
            f0delay = sequence[0] + sequence[1]
            logging.warning(
                self.logwarn + "setTiming: Due to use of the Icarus model 1 sensor, the"
                " initial delay for side "
                + side
                + " will actually be "
                + str(delay + f0delay)
                + " nanoseconds"
            )
        return err, full40hex

    # TODO: restore after power cycle?
    # TODO: smart interpretation of Icarus1 timing?
    # TODO: error checking like in getTiming
    def setArbTiming(self, side="AB", sequence=None):
        """
        Set arbitrary high-speed timing sequence.
        Args:
            side: Hemisphere 'A', 'B', 'AB'
            sequence: list of arbitrary timing intervals, beginning with initial delay.
              The conventional timing (3,2) with delay = 0 would be represented by
              [0,3,2,3,2,3,2,3] on icarus devices, [0,3,2,3,2,3] on daedalus. If used
              for interlacing or ZDT, you should populate the entire 40-bit register,
              e.g., [0,3,2,3,2,3,2,3,2,3,2,3,2,3,2,3,2]

            *NOTE* Icarus sensors generally cannot use 1 ns timing, so should use at
              least 2 ns for frames 2 and 3 integration and interframe times (an initial
              delay of only 1 ns is acceptable)

            *NOTE* although the Icarus model 1 only images the middle two frames, timing
              entries must be provided for all four frames; to implement frame 1 open
              for X ns, shutter closed for Y ns, and frame 2 open for Z ns, use the
              sequence [0,1,1,X,Y,Z,1,1]

            *WARNING* arbitrary timings will not be restored after a board power cycle

        Returns:
            list: Actual timing results
        """
        logging.info(
            "{}setArbTiming; side={}, sequence={}".format(self.loginfo, side, sequence)
        )
        if sequence is None:
            if self.sensfam == "Daedalus":
                sequence = [0, 2, 3, 4, 5, 6]
            else:
                sequence = [0, 2, 3, 4, 5, 6, 7, 8]
        logging.info(
            self.loginfo + "HST side " + side.upper() + " (arbitrary): " + str(sequence)
        )
        if side.upper() == "AB":
            err1, _ = self.setArbTiming(side="A", sequence=sequence)
            err2, actual = self.setArbTiming(side="B", sequence=sequence)
            return err1 + err2, actual
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            err = (
                self.logerr
                + "Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            logging.error("{}setArbTiming: {}".format(self.logerr, err))
            return err, "0000000000"

        full40 = [0] * 40
        bitlist = []
        flag = 0  # similar to setTiming, but starts with delay

        for a in sequence:
            add = [flag] * a
            bitlist += add
            if flag:
                flag = 0
            else:
                flag = 1

        logging.debug(self.logdebug + "bitlist = " + str(bitlist))
        reversedlist = bitlist[39::-1]
        full40[-(len(reversedlist) + 1) : -1] = reversedlist
        full40bin = "".join(str(x) for x in full40)
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        full40hex = "%x" % int(full40bin, 2)
        logging.debug(self.logdebug + "full40hex = " + str(full40hex))
        highpart = full40hex[-10:-8].zfill(8)
        lowpart = full40hex[-8:].zfill(8)
        self.ca.setRegister(lowreg, lowpart)
        self.ca.setRegister(highreg, highpart)
        # deactivates manual shutter mode if previously engaged
        self.ca.setSubregister("MANSHUT_MODE", "0")
        self.ca.setSubregister("HST_MODE", "1")
        actual = self.getTiming(side, actual=True)
        f0delay = sequence[1] + sequence[2]

        if self.ca.sensorname == "icarus":
            if actual != sequence[:1] + sequence[3:6]:
                logging.warning(
                    self.logwarn + "Due to sequence length and use of the Icarus model "
                    "1 sensor, the actual timing sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0] + f0delay)
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
            else:
                logging.warning(
                    self.logwarn + "Due to use of the Icarus model 1 sensor, the actual"
                    " timing sequence for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0] + f0delay)
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        else:
            if actual != sequence:
                logging.warning(
                    self.logwarn + "Due to sequence length, actual timing sequence "
                    "for side "
                    + side
                    + " will be "
                    + "{"
                    + str(actual[0])
                    + "}"
                    + " "
                    + str(actual[1 : 2 * self.nframes])
                )
        return "", actual

    # TODO: figure out how to handle interlacing?
    def getTiming(self, side, actual):
        """
        actual = True: returns actual high speed intervals that will be generated by the
                    FPGA as list
                 False: Returns high speed timing settings as set by setTiming. Assumes
                    that timing was set via the setTiming method--it will not accurately
                    report arbitrary timings set by direct register sets or manual
                    shutter control

        Args:
            side: Hemisphere 'A' or 'B'
            actual: False: return HST settings
                    True: calculate and return actual HST behavior

        Returns:
            actual= True: list of shutter intervals;
```

```
                        icarus: [delay, open0, closed0, open1, closed1, open2, closed2,
                                 open3]
                        daedalus: [delay, open0, closed0, open1, closed1, open2]
                    False: tuple (hemisphere label,
                                    'open shutter' in ns,
                                    'closed shutter' in ns,
                                    initial delay in ns)

        """
        logging.info("{}getTiming".format(self.loginfo))
        if side is None:
            side = "A"
        logging.info(self.loginfo + "get timing, side " + side.upper())
        if side.upper() == "A":
            lowreg = "HS_TIMING_DATA_ALO"
            highreg = "HS_TIMING_DATA_AHI"
        elif side.upper() == "B":
            lowreg = "HS_TIMING_DATA_BLO"
            highreg = "HS_TIMING_DATA_BHI"
        else:
            logging.error(
                self.logerr
                + "Invalid sensor side: "
                + side
                + "; timing settings unchanged"
            )
            return "", 0, 0, 0
        err, lowpart = self.ca.getRegister(lowreg)
        err1, highpart = self.ca.getRegister(highreg)
        if err or err1:
            logging.error(
                self.logerr + "Unable to retrieve timing setting (getTiming), "
                "returning zeroes "
            )
            return side.upper(), 0, 0, 0
        full40hex = highpart[-2:] + lowpart.zfill(8)
        full40bin = "{0:0=40b}".format(int(full40hex, 16))
        logging.debug(self.logdebug + "full40bin = " + str(full40bin))
        if actual:
            if full40bin == "0" * 40:  # all-zero timing
                if self.fpganumID == 2:
                    times = [0] * 6
                else:
                    times = [0] * 8
            else:
                full160 = 4 * full40bin
                gblist = [[k, len(list(g))] for k, g in itertools.groupby(full160)]
                if self.fpganumID == 2:
                    times = [int(x[1]) for x in gblist[:-7:-1]]
                else:
                    times = [int(x[1]) for x in gblist[:-9:-1]]
                times[0] = times[0] - 1
            if self.ca.sensorname == "icarus":
                # get timing for frames 1 and 2, keep delay as offset
                # TODO: should this give a 'correct' offset from frame 0?
                times12 = [times[0]] + times[3:6]
                return times12
            return times
        else:
            if full40bin == "0" * 40:  # all-zero timing
                timeon, timeoff, delay = (0, 0, 0)
            else:
                gblist = [[k, len(list(g))] for k, g in itertools.groupby(full40bin)]
                delay = gblist[-1][1] - 1
                timeon = gblist[-2][1]

                if self.ca.sensorname == "icarus":
                    if len(gblist) == 2:  # 39,1 corner case
                        timeoff = 1
                    elif len(gblist) == 3:  # sequence fits only once
                        timeoff = 40 - timeon
                    else:
                        timeoff = gblist[-3][1]
                else:
                    if len(gblist) < self.nframes:  # sequence fits only once
                        timeoff = 40 - timeon
                    else:
                        # TODO: confirm '-3' works for daedalus
                        timeoff = gblist[-3][1]
            return side.upper(), timeon, timeoff, delay

```

```
    def setManualShutters(self, timing=None):
        """
        Legacy alias for setManualTiming()
        """
        self.setManualTiming(timing)

    def setManualTiming(self, timing=None):
        """
        Manual shutter timing, seven intervals to assign to both hemispheres, e.g.,
          [(100,150,100,150,100,150,100)] for frame 0 open for 100 ns, an interframe
          pause of 50 ns,frame 1 open for 100 ns, etc.
        Provide two sets of seven intervals, e.g., [(100,150,100,150,100,150,100),
          (200,250,200,250,200,250,200)] to program the A and B hemispheres
          independently

        Overridden in daedalus.py

        The timing list is flattened before processing; the suggested tuple structure is
          just for clarity (first tuple is A, second is B) and is optional.

        The actual timing is rounded down to the nearest multiple of 25 ns. (Each
          count = 25 ns. e.g., a request for 140 ns rounds down to a count of '5',
          which corresponds to 125 ns))
            - Minimum timing is 75 ns
            - Maximum is 25 * 2^30 ns (approximately 27 seconds)

        Args:
            timing: 7- or 14-element list (substructure optional) in nanoseconds

        Returns:
            tuple (error string, response string from final message)
        """
        if timing is None:
            logging.info(
                self.loginfo
                + "No manual timing setting provided, defaulting to (100, 150, 100, "
                " 150, 100, 150, 100) for both hemispheres"
            )
            timing = [
                (100, 150, 100, 150, 100, 150, 100),
                (100, 150, 100, 150, 100, 150, 100),
            ]
        logging.info(self.loginfo + "Manual shutter sequence: " + str(timing))
        flattened = flattenlist(timing)
        if len(flattened) == 7:
            flattened = 2 * flattened
        if (
            len(flattened) != 14
            or not all(isinstance(x, numbers.Real) for x in flattened)
            or not all(x >= 75 for x in flattened)
            or not all(x <= 26843545600 for x in flattened)
        ):
            err = self.logerr + "Invalid manual shutter timing list: " + str(timing)
            logging.error(err + "; timing settings unchanged")
            return err, "00000000"

        timecounts = [int(a // 25) for a in flattened]
        self.ca.sensmanual = timing
        self.ca.senstiming = {}  # clear HST settings from ca object

        control_messages = [
            ("W0_INTEGRATION", "{0:#0{1}x}".format(timecounts[0], 10)[2:10]),
            ("W0_INTERFRAME", "{0:#0{1}x}".format(timecounts[1], 10)[2:10]),
            ("W1_INTEGRATION", "{0:#0{1}x}".format(timecounts[2], 10)[2:10]),
            ("W1_INTERFRAME", "{0:#0{1}x}".format(timecounts[3], 10)[2:10]),
            ("W2_INTEGRATION", "{0:#0{1}x}".format(timecounts[4], 10)[2:10]),
            ("W2_INTERFRAME", "{0:#0{1}x}".format(timecounts[5], 10)[2:10]),
            ("W3_INTEGRATION", "{0:#0{1}x}".format(timecounts[6], 10)[2:10]),
            ("W0_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[7], 10)[2:10]),
            ("W0_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[8], 10)[2:10]),
            ("W1_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[9], 10)[2:10]),
            ("W1_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[10], 10)[2:10]),
            ("W2_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[11], 10)[2:10]),
            ("W2_INTERFRAME_B", "{0:#0{1}x}".format(timecounts[12], 10)[2:10]),
            ("W3_INTEGRATION_B", "{0:#0{1}x}".format(timecounts[13], 10)[2:10]),
            ("HST_MODE", "0"),
            ("MANSHUT_MODE", "1"),
        ]
        return self.ca.submitMessages(control_messages, " setManualShutters: ")

    def getManualTiming(self):
        """
        Read off manual shutter timing settings
        Overridden in daedalus.py
        Returns:
            list of 2 lists of timing from A and B sides, respectively
        """
        aside = []
        bside = []
        for reg in [
            "W0_INTEGRATION",
            "W0_INTERFRAME",
            "W1_INTEGRATION",
            "W1_INTERFRAME",
            "W2_INTEGRATION",
            "W2_INTERFRAME",
            "W3_INTEGRATION",
        ]:
            _, reghex = self.ca.getRegister(reg)
            aside.append(25 * int(reghex, 16))
        for reg in [
            "W0_INTEGRATION_B",
            "W0_INTERFRAME_B",
            "W1_INTEGRATION_B",
            "W1_INTERFRAME_B",
            "W2_INTEGRATION_B",
            "W2_INTERFRAME_B",
            "W3_INTEGRATION_B",
        ]:
            _, reghex = self.ca.getRegister(reg)
            bside.append(25 * int(reghex, 16))
        return [aside, bside]

    def getSensTemp(self, scale=None, offset=None, slope=None, dec=None):
        """
        Virtual method (Temperature sensor is not present on Icarus sensors). Returns 0.
        Overridden by Daedalus method
        """
        return 0

    def selectOscillator(self, osc=None):
        """
        Selects oscillator to control sensor timing
        Overridden in daedalus.py
        Args:
            osc: 'relaxation'|'ring'|'ringnoosc'|'external', defaults to relaxation

        Returns:
            error message as string
        """
        logging.info(self.loginfo + "selectOscillator; osc = " + str(osc))
        if osc is None:
            osc = "rel"
        osc = str(osc)
        if osc.upper()[:3] == "REL":
            payload = "00"
        elif osc.upper()[:3] == "RIN":
            if "NO" in osc.upper() or "0" in osc:
                payload = "10"
            else:
                payload = "01"
        elif osc.lower()[:3] in ["ext", "phi"]:
            payload = "11"
        else:
            err = (
                self.logerr + "selectOscillator: invalid parameter supplied. "
                "Oscillator selection is unchanged."
            )
            logging.error(err)
            return err
        self.ca.setSubregister("OSC_SELECT", payload)

    def parseReadoff(self, frames, columns):
        """
        Virtual method (Order parsing is unnecessary for Icarus, continue to hemisphere
          parsing.)
        Overridden by Daedalus method
        """
        return self.ca.partition(frames, columns)

    def getSensorStatus(self):
        """
        Wrapper for reportSensorStatus so that the user doesn't have to query statusbits
        """
        sb1 = self.ca.board.checkstatus()
        sb2 = self.ca.board.checkstatus2()
        self.reportStatusSensor(sb1, sb2)

    def reportStatusSensor(self, statusbits, statusbits2):
        """
        Print status messages from sensor-specific bits of status register, default for
          Icarus family sensors
        Args:
            statusbits: result of checkStatus()
            statusbits2: result of checkStatus2()
        """
        if int(statusbits[3]):
            print(self.loginfo + "W3_Top_A_Edge1 detected")
        if int(statusbits[4]):
            print(self.loginfo + "W3_Top_B_Edge1 detected")
        if int(statusbits[12]):
            print(self.loginfo + "HST_All_W_En detected")
        if self.ca.boardname == "llnl_v4" and int(statusbits2[5]):
            print(self.loginfo + "PDBIAS Unready")


# TODO: add function to control TIME_ROW_DCD delay

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.41 C:/Users/hill35/git/camera_python/nsCamera/utils/crc16pure.py File Reference

Namespaces

- namespace nsCamera
- namespace nsCamera.utils
- namespace nsCamera.utils.crc16pure

Functions

- nsCamera.utils.crc16pure._crc16 (data, crc, table)
- nsCamera.utils.crc16pure.crc16xmodem (data, crc=0)

Variables

- list nsCamera.utils.crc16pure.CRC16_XMODEM_TABLE

7.42 crc16pure.py

Go to the documentation of this file.

```
#!/usr/bin/env python
"""Pure python library for calculating CRC16
    NOTE: modified slightly to combine Python 2 and Python 3 versions in single file
"""



import sys

# table for calculating CRC
# this particular table was generated using pycrc v0.7.6, http://www.tty1.net/pycrc/
# using the configuration:
#  *    Width         = 16
#  *    Poly          = 0x1021
#  *    XorIn         = 0x0000
#  *    ReflectIn     = False
#  *    XorOut        = 0x0000
#  *    ReflectOut    = False
#  *    Algorithm     = table-driven
# by following command:
#   python pycrc.py --model xmodem --algorithm table-driven --generate c
CRC16_XMODEM_TABLE = [
    0x0000,
    0x1021,
    0x2042,
    0x3063,
    0x4084,
    0x50A5,
    0x60C6,
    0x70E7,
    0x8108,
    0x9129,
    0xA14A,
    0xB16B,
    0xC18C,
    0xD1AD,
    0xE1CE,
    0xF1EF,
    0x1231,
    0x0210,
    0x3273,
    0x2252,
    0x52B5,
    0x4294,
    0x72F7,
    0x62D6,
    0x9339,
    0x8318,
    0xB37B,
    0xA35A,
    0xD3BD,
    0xC39C,
    0xF3FF,
    0xE3DE,
    0x2462,
    0x3443,
    0x0420,
    0x1401,
    0x64E6,
    0x74C7,
    0x44A4,
    0x5485,
    0xA56A,
    0xB54B,
    0x8528,
    0x9509,
    0xE5EE,
    0xF5CF,
    0xC5AC,
    0xD58D,
    0x3653,
    0x2672,
    0x1611,
    0x0630,
    0x76D7,
    0x66F6,
    0x5695,
    0x46B4,
    0xB75B,
    0xA77A,
    0x9719,
    0x8738,
    0xF7DF,
    0xE7FE,
    0xD79D,
    0xC7BC,
    0x48C4,
    0x58E5,
    0x6886,
    0x78A7,
    0x0840,
    0x1861,
    0x2802,
    0x3823,
    0xC9CC,
    0xD9ED,
    0xE98E,
    0xF9AF,
    0x8948,
    0x9969,
    0xA90A,
    0xB92B,
    0x5AF5,
    0x4AD4,
    0x7AB7,
    0x6A96,
    0x1A71,
    0x0A50,
    0x3A33,
    0x2A12,
    0xDBFD,
    0xCBDC,
    0xFBBF,
    0xEB9E,
    0x9B79,
    0x8B58,
    0xBB3B,
    0xAB1A,
    0x6CA6,
    0x7C87,
    0x4CE4,
    0x5CC5,
    0x2C22,
    0x3C03,
    0x0C60,
    0x1C41,
    0xEDAE,
    0xFD8F,
    0xCDEC,
    0xDDCD,
    0xAD2A,
    0xBD0B,
    0x8D68,
    0x9D49,
    0x7E97,
    0x6EB6,
    0x5ED5,
    0x4EF4,
    0x3E13,
    0x2E32,
    0x1E51,
    0x0E70,
    0xFF9F,
    0xEFBE,
    0xDFDD,
    0xCFFC,
    0xBF1B,
    0xAF3A,
    0x9F59,
    0x8F78,
    0x9188,
    0x81A9,
    0xB1CA,
    0xA1EB,
    0xD10C,
    0xC12D,
    0xF14E,
0xE16F,
0x1080,
0x00A1,
0x30C2,
0x20E3,
0x5004,
0x4025,
0x7046,
0x6067,
0x83B9,
0x9398,
0xA3FB,
0xB3DA,
0xC33D,
0xD31C,
0xE37F,
0xF35E,
0x02B1,
0x1290,
0x22F3,
0x32D2,
0x4235,
0x5214,
0x6277,
0x7256,
0xB5EA,
0xA5CB,
0x95A8,
0x8589,
0xF56E,
0xE54F,
0xD52C,
0xC50D,
0x34E2,
0x24C3,
0x14A0,
0x0481,
0x7466,
0x6447,
0x5424,
0x4405,
0xA7DB,
0xB7FA,
0x8799,
0x97B8,
0xE75F,
0xF77E,
0xC71D,
0xD73C,
0x26D3,
0x36F2,
0x0691,
0x16B0,
0x6657,
0x7676,
0x4615,
0x5634,
0xD94C,
0xC96D,
0xF90E,
0xE92F,
0x99C8,
0x89E9,
0xB98A,
0xA9AB,
0x5844,
0x4865,
0x7806,
0x6827,
0x18C0,
0x08E1,
0x3882,
0x28A3,
0xCB7D,
0xDB5C,
0xEB3F,
0xFB1E,
0x8BF9,
0x9BD8,
0xABBB,
0xBB9A,
    0x4A75,
    0x5A54,
    0x6A37,
    0x7A16,
    0x0AF1,
    0x1AD0,
    0x2AB3,
    0x3A92,
    0xFD2E,
    0xED0F,
    0xDD6C,
    0xCD4D,
    0xBDAA,
    0xAD8B,
    0x9DE8,
    0x8DC9,
    0x7C26,
    0x6C07,
    0x5C64,
    0x4C45,
    0x3CA2,
    0x2C83,
    0x1CE0,
    0x0CC1,
    0xEF1F,
    0xFF3E,
    0xCF5D,
    0xDF7C,
    0xAF9B,
    0xBFBA,
    0x8FD9,
    0x9FF8,
    0x6E17,
    0x7E36,
    0x4E55,
    0x5E74,
    0x2E93,
    0x3EB2,
    0x0ED1,
    0x1EF0,
]


def _crc16(data, crc, table):
    """Calculate CRC16 using the given table.
    `data`      - data for calculating CRC, must be a string
    `crc`       - initial value
    `table`     - table for caclulating CRC (list of 256 integers)
    Return calculated value of CRC
    """
    for byte in data:
        if sys.version_info > (3,):
            crc = ((crc « 8) & 0xFF00) ^ table[((crc » 8) & 0xFF) ^ byte]
        else:
            crc = ((crc « 8) & 0xFF00) ^ table[((crc » 8) & 0xFF) ^ ord(byte)]

    return crc & 0xFFFF


def crc16xmodem(data, crc=0):
    """Calculate CRC-CCITT (XModem) variant of CRC16.
    `data`      - data for calculating CRC, must be a string
    `crc`       - initial value
    Return calculated value of CRC
    """
    return _crc16(data, crc, CRC16_XMODEM_TABLE)
```

7.43 C:/Users/hill35/git/camera_python/nsCamera/utils/FlatField.py File Reference

Namespaces

- namespace nsCamera

- namespace nsCamera.utils
- namespace nsCamera.utils.FlatField

Functions

- nsCamera.utils.FlatField.getFilenames (frame="Frame 1")
- nsCamera.utils.FlatField.getROIvector (imgfilename, roi)
- nsCamera.utils.FlatField.tslopes (x, y)
- nsCamera.utils.FlatField.generateFF (FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"], roi=[0, 0, 512, 1024], directory="", ncores=-1)
- nsCamera.utils.FlatField.removeFF (filename, directory="", roi=[0, 0, 512, 1024])
- nsCamera.utils.FlatField.removeFFall (directory="", FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"], roi=[0, 0, 512, 1024])

Variables

- nsCamera.utils.FlatField.parser = argparse.ArgumentParser()
- nsCamera.utils.FlatField.action
- nsCamera.utils.FlatField.dest
- nsCamera.utils.FlatField.default
- nsCamera.utils.FlatField.help
- nsCamera.utils.FlatField.nargs
- nsCamera.utils.FlatField.args = parser.parse_args()
- list nsCamera.utils.FlatField.framelist = ["Frame_" + str(frame) for frame in args.frames]
- nsCamera.utils.FlatField.directory

7.44 FlatField.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Functions for batch flat-field image corrections

***Do not use this file as a template for new code development***

Author: Jeremy Martin Hill (jerhill@llnl.gov)
Author: Matthew Dayton (dayton5@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import os
import re

import numpy as np
from PIL import Image
from joblib import parallel, delayed
from scipy.stats import theilslopes
from skimage.external.tifffile import imread
```

```


def getFilenames(frame="Frame 1"):
    """
    get a list of tiff filenames in current working director for frame
    """
    onlyfiles = next(os.walk("./"))[2]
    return [k for k in onlyfiles if frame in k and "tif" in k]


def getROIvector(imgfilename, roi):
    """
    return a numpy row vector of version of the image
    """
    img = imread(imgfilename)
    vroi = img[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])].flattenlist()
    return vroi


def tslopes(x, y):
    """
    theilslopes implements a method for robust linear regression.
    It computes the slope as the median of all slopes between paired values.
    """
    val = theilslopes(x, y)
    return [val[0], val[1]]


def generateFF(
    FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"],
    roi=[0, 0, 512, 1024],
    directory="",
    ncores=-1,
):
    # TODO: documentation
    # use of ROI here not compatible with use of ROI in removeFF

    if directory:
        cwd = os.getcwd()
        newpath = os.path.join(cwd, directory)
        os.chdir(newpath)
    if not FRAMES:
        print("No framelist provided, defaulting to four frames")
        FRAMES = ["Frame_0", "Frame_1", "Frame_2", "Frame_3"]
    for f in FRAMES:
        files = getFilenames(frame=f)
        imgslist = [getROIvector(fn, roi) for fn in files]  # a list of flattened images
        imgsarray = np.vstack(imgslist)  # turn the list into an array
        npix = np.shape(imgsarray)[1]  # total number of pixels
        x = np.median(imgsarray, axis=1)  # median of each image used for flat fielding
        y = []
        for i in range(npix):
            # each member of y represents a pixel, as a list of magnitudes over all the
            #   images
            y.append(imgsarray[:, i])
        # get pixel gain and offset for flatfield ff using Thiel-Sen slopes
        ff = []
        ff = parallel.Parallel(n_jobs=ncores, verbose=5, pre_dispatch="2 * n_jobs")(
            delayed(tslopes)(x, pixel) for pixel in y
        )
        # x is the dependent variable; here uses median of image as characteristic of
        #   noise level
        m, c = zip(*ff)  # separate into gain and offset
        m = np.array(m)
        m[m < 0.1] = 0.1  # handle outliers
        m[m > 1000] = 1000  # handle outliers
        m = 1.0 / m
        m = m.reshape(roi[3] - roi[1], roi[2] - roi[0])  # turn into matrix
        c = np.array(c).reshape(roi[3] - roi[1], roi[2] - roi[0])  # turn into matrix

        with open("px_gain_%s.txt" % f.replace("Frame_", "f"), "w+") as file:
            np.savetxt(file, m)
        with open("px_off_%s.txt" % f.replace("Frame_", "f"), "w+") as file:
            np.savetxt(file, c)


def removeFF(filename, directory="", roi=[0, 0, 512, 1024]):
    if directory:
        cwd = os.getcwd()
        newpath = os.path.join(cwd, directory)
        os.chdir(newpath)
    framenum = re.search("Frame_(\d)", filename).group(1)
    gainFilename = "px_gain_f" + framenum + ".txt"
    gainall = np.loadtxt(gainFilename)
    gain = gainall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]
    offFilename = "px_off_f" + framenum + ".txt"
    offsetall = np.loadtxt(offFilename, dtype="uint32")
    offset = offsetall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]

    beforeImageall = imread(filename)
    beforeImage = beforeImageall[(roi[1]) : (roi[3]), (roi[0]) : (roi[2])]
    imageMed = np.median(beforeImage)

    flat = imageMed * gain + offset
    flat = flat.clip(0)
    fix = beforeImage - flat
    clipped = fix.clip(0)
    fixinit = clipped.astype("uint16")
    fiximg = Image.fromarray(fixinit)

    fixFilename = filename[:-4] + "ff" + filename[-4:]
    fiximg.save(fixFilename)

def removeFFall(
    directory="",
    FRAMES=["Frame_0", "Frame_1", "Frame_2", "Frame_3"],
    roi=[0, 0, 512, 1024],
):
    cwd = os.getcwd()
    if directory:
        newpath = os.path.join(cwd, directory)
    else:
        newpath = cwd
    os.chdir(newpath)
    files = next(os.walk("./"))[2]
    filelist = []
    for frame in FRAMES:
        filelist.extend([k for k in files if frame in k and "tif" in k])
    for fname in filelist:
        removeFF(fname, directory, roi)


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-d", action="store", dest="directory", default="", help="VRST scan directory"
    )
    parser.add_argument(
        "-f",
        nargs="+",
        action="store",
        dest="frames",
        default="",
        help="Frame numbers to process, eg. -f 2 3",
    )
    args = parser.parse_args()
    framelist = ["Frame_" + str(frame) for frame in args.frames]
    generateFF(framelist, directory=args.directory)

"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.45 C:/Users/hill35/git/camera_python/nsCamera/utils/misc.py File Reference

Classes

- class nsCamera.utils.misc.fakeCA

Namespaces

- namespace nsCamera
- namespace nsCamera.utils
- namespace nsCamera.utils.misc

Functions

- nsCamera.utils.misc.makeLogLabels (logtag, label)
- nsCamera.utils.misc.getEnter (text)
- nsCamera.utils.misc.checkCRC (rval)
- nsCamera.utils.misc.str2bytes (astring)
- nsCamera.utils.misc.bytes2str (bytesequence)
- nsCamera.utils.misc.str2nparray (valstring)
- nsCamera.utils.misc.flattenlist (x)
- nsCamera.utils.misc.generateFrames (camassem, data, columns=1)
- nsCamera.utils.misc.loadDumpedData (filename="frames.txt", path=None, filetype="txt", sensor="daedalus", firstframe=None, lastframe=None, width=None, height=None, padToFull=None, firstrow=None, lastrow=None, maxwidth=None, maxheight=None, bytesperpixel=None, interlacing=None, columns=1)
- nsCamera.utils.misc.saveTiffs (self, frames, path=None, filename="Frame", prefix=None, index=None)
- nsCamera.utils.misc.plotFrames (self, frames, index=None)
- nsCamera.utils.misc.partition (self, frames, columns)

7.46 misc.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Miscellaneous utilities, including batch processing of images acquired using the
  nsCamera. These are functions that don't require a cameraAssembler object to be
  instantiated before use.

Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

import binascii
```

```python
import collections
import logging
import os
import sys
from datetime import datetime

import numpy as np
from matplotlib import pyplot as plt
from past.builtins import raw_input
from PIL import Image

# TODO: is the self-reference necessary?
# import nsCamera.utils.misc
from nsCamera.utils import crc16pure


# TODO: check error messages
def makeLogLabels(logtag, label):
    if logtag is None:
        logtag = ""

    logcritbase = "CRITICAL{logtag}: ".format(logtag=logtag)
    logerrbase = "ERROR{logtag}: ".format(logtag=logtag)
    logwarnbase = "WARNING{logtag}: ".format(logtag=logtag)
    loginfobase = "INFO{logtag}: ".format(logtag=logtag)
    logdebugbase = "DEBUG{logtag}: ".format(logtag=logtag)

    logcrit = "{base}{label}".format(base=logcritbase, label=label)
    logerr = "{base}{label}".format(base=logerrbase, label=label)
    logwarn = "{base}{label}".format(base=logwarnbase, label=label)
    loginfo = "{base}{label}".format(base=loginfobase, label=label)
    logdebug = "{base}{label}".format(base=logdebugbase, label=label)

    return logcrit, logerr, logwarn, loginfo, logdebug


def getEnter(text):
    """
    Wait for enter key to be pressed.

    Args:
        text: message asking for keypress
    """
    python, _, _, _, _ = sys.version_info
    if python >= 3:
        input(text)
    else:
        raw_input(text)


def checkCRC(rval):
    """
    Calculate CRC for rval[:-4] and compare with expected CRC in rval[-4:]

    Args:
        rval: hexadecimal string

    Returns:
        boolean, True if CRCs match, False if they don't match or the input is invalid
    """
    if not isinstance(rval, str) or len(rval) < 5:
        logging.error("ERROR: checkCRC: Invalid input: {rval}".format(rval=rval))
        return False
    data_crc = int(rval[-4:], base=16)
    CRC_calc = crc16pure.crc16xmodem(str2bytes(rval[:-4]))
    return CRC_calc == data_crc


def str2bytes(astring):
    """
    Python-version-agnostic converter of hexadecimal strings to bytes

    Args:
        astring: hexadecimal string without '0x'

    Returns:
        byte string equivalent to input string
    """

    python, _, _, _, _ = sys.version_info
    if python >= 3:
        try:
            dbytes = binascii.a2b_hex(astring)
        except:
            logging.error(
                "ERROR: str2bytes: invalid input: '{astring}'; returning zero"
                " byte".format(astring=astring)
            )
            dbytes = b"\x00"
    else:
        try:
            dbytes = astring.decode("hex")
        except:
            logging.error(
                "ERROR: str2bytes: invalid input: '{astring}'; returning zero "
                " byte".format(astring=astring)
            )
            dbytes = b"\x00"
    return dbytes


def bytes2str(bytesequence):
    """
    Python-version-agnostic converter of bytes to hexadecimal strings

    Args:
        bytesequence: sequence of bytes as string (Py2) or bytes (Py3)

    Returns:
        hexadecimal string representation of 'bytes' without '0x'
    """
    try:
        estring = binascii.b2a_hex(bytesequence)
    except TypeError:
        logging.error(
            "ERROR: bytes2str: Invalid byte sequence: '{bytesequence}'; returning an"
            " empty string".format(bytesequence=bytesequence)
        )
        return ""
    python, _, _, _, _ = sys.version_info
    if python >= 3:
        estring = str(estring)[2:-1]
    return estring


def str2nparray(valstring):
    """
    Convert string into array of uint16s

    Args:
        valstring: string of hexadecimal characters

    Returns:
        numpy array of uint16
    """
    if not isinstance(valstring, str):
        logging.error(
            "ERROR: str2nparray: Invalid input: {valstring} is not a string. Returning"
            " an empty array".format(valstring=valstring)
        )
        return np.array([])
    stringlen = len(valstring)
    arraylen = int(stringlen / 4)
    outarray = np.empty(int(arraylen), dtype="uint16")

    for i in range(0, arraylen):
        try:
            outarray[i] = int(valstring[4 * i : 4 * i + 4], 16)
        except ValueError:
            logging.error(
                "ERROR: str2nparray: input string does not represent a hexadecimal"
                " integer. Returning an empty array"
            )
            return np.array([])
    return outarray


def flattenlist(x):
    """
    Flatten list of lists recursively into single list
    """
    python, _, _, _, _ = sys.version_info
    try:
        if python >= 3:
            if isinstance(x, collections.abc.Iterable):
                return [a for i in x for a in flattenlist(i)]
            else:
                return [x]
        else:
            if isinstance(x, collections.Iterable):
                return [a for i in x for a in flattenlist(i)]
            else:
                return [x]
    except RecursionError:
        logging.error(
            "ERROR: flattenlist: input '{x}' is pathological and cannot be flattened."
            " Attempting to return the input unchanged"
        )
        return x


class fakeCA:
    """
    Fake 'cameraAssembler' object to use as a parameter object in offline functions.
      Returned by generateFrames(), it contains the frame details required to save and
      plot images.
    """

    def __init__(
        self,
        sensorname="icarus2",
        firstframe=0,
        lastframe=3,
        width=512,
        height=1024,
        padToFull=True,
        firstrow=0,
        lastrow=1023,
        maxwidth=512,
        maxheight=1024,
        bytesperpixel=2,
        interlacing=None,
        columns=1,
        logtag=None,
    ):
        self.sensorname = sensorname
        self.boardname = None
        self.padToFull = padToFull
        if logtag is None:
            self.logtag = ""
        else:
            self.logtag = logtag

        self.logcritbase = "CRITICAL" + self.logtag + ": "
        self.logerrbase = "ERROR" + self.logtag + ": "
        self.logwarnbase = "WARNING" + self.logtag + ": "
        self.loginfobase = "INFO" + self.logtag + ": "
        self.logdebugbase = "DEBUG" + self.logtag + ": "

        self.logcrit = self.logcritbase + "[FS] "
        self.logerr = self.logerrbase + "[FS] "
        self.logwarn = self.logwarnbase + "[FS] "
        self.loginfo = self.loginfobase + "[FS] "
        self.logdebug = self.logdebugbase + "[FS] "

        if self.sensorname == "icarus":
            import nsCamera.sensors.icarus as snsr
        elif self.sensorname == "icarus2":
            import nsCamera.sensors.icarus2 as snsr
        elif self.sensorname == "daedalus":
            import nsCamera.sensors.daedalus as snsr

        self.sensor = snsr(self)

    def partition(self, parsed, columns):
        # return nsCamera.utils.misc.partition(self, parsed, columns)
        return partition(self, parsed, columns)


def generateFrames(camassem, data, columns=1):
    """
    Processes data stream from board into frames and applies sensor-specific parsing.
      Generates padded data for full-size option of setRows.
    If used for offline processing, replace the 'self' object with the parameter object
      returned by loadDumpedData().
    If the data stream is incomplete (e.g., from an interrupted download), the data is
      padded with zeros to the correct length.

    Args:
        self: during normal operation, is the CameraAssembler object. During offline
          operation, is a parameters object as returned by loadDumpedData()
        data: text stream from board or loaded file, or numpy from loaded file
        columns: 1 for single image, 2 for separate hemisphere images

    Returns: list of parsed frames
    """
    logging.debug("DEBUG" + camassem.logtag + ": generateFrames")
    if isinstance(data[0], str):
        allframes = str2nparray(data)
    else:
        allframes = data
    nframes = camassem.sensor.lastframe - camassem.sensor.firstframe + 1
    frames = [0] * nframes
    framesize = camassem.sensor.width * (
        camassem.sensor.lastrow - camassem.sensor.firstrow + 1
    )
    if hasattr(camassem, "ca"):
        padIt = camassem.ca.padToFull
    else:
        padIt = camassem.padToFull
    if padIt:
        toprows = camassem.sensor.firstrow
        botrows = (camassem.sensor.maxheight - 1) - camassem.sensor.lastrow
        padtop = np.zeros(toprows * camassem.sensor.maxwidth, dtype=int)
        padbot = np.zeros(botrows * camassem.sensor.maxwidth, dtype=int)
        for n in range(nframes):
            thisframe = np.concatenate(
                (padtop, allframes[n * framesize : ((n + 1) * framesize)], padbot)
            )
            frames[n] = thisframe
    else:
        for n in range(nframes):
            frames[n] = allframes[n * framesize : (n + 1) * framesize]
    # self.clearStatus()
    parsed = camassem.sensor.parseReadoff(frames, columns)
    return parsed


def loadDumpedData(
    filename="frames.txt",
    path=None,
    filetype="txt",
    sensor="daedalus",
    firstframe=None,
    lastframe=None,
    width=None,
    height=None,
    padToFull=None,
    firstrow=None,
    lastrow=None,
    maxwidth=None,
    maxheight=None,
    bytesperpixel=None,
    interlacing=None,
    columns=1,
):
    """_summary_

        Output can be piped directly to saveTiffs:
            pars,frames=loadDumpedData(filename="Dump.npy")
            saveTiffs(pars,frames)
    Args:
        filename (str, optional): _description_. Defaults to "frames.txt".
        path (_type_, optional): _description_. Defaults to None.
        filetype (str, optional): _description_. Defaults to "txt".
        sensor (str, optional): _description_. Defaults to "daedalus".
        firstframe (_type_, optional): _description_. Defaults to None.
        lastframe (_type_, optional): _description_. Defaults to None.
        width (_type_, optional): _description_. Defaults to None.
        height (_type_, optional): _description_. Defaults to None.
        padToFull (_type_, optional): _description_. Defaults to None.
        firstrow (_type_, optional): _description_. Defaults to None.
        lastrow (_type_, optional): _description_. Defaults to None.
        maxwidth (_type_, optional): _description_. Defaults to None.
        maxheight (_type_, optional): _description_. Defaults to None.
        interlacing (_type_, optional): _description_. Defaults to None.
        columns (int, optional): _description_. Defaults to 1.


    Returns:
        Tuple (parameter object, list of data frames)
    """
    logging.debug("DEBUG: loadDumpedData")
    if sensor.lower() == "daedalus":
        import nsCamera.sensors.daedalus as snsr
    elif sensor.lower() == "icarus":
        import nsCamera.sensors.icarus as snsr
    elif sensor.lower() == "icarus2":
        import nsCamera.sensors.icarus2 as snsr
    else:
        logging.error(
            "ERROR loadDumpedData: invalid sensor type provided, defaulting to icarus2"
        )
        import nsCamera.sensors.icarus2 as snsr

    def buildEmptyFrames():
        cols = [0] * 512
        frame = np.array([cols] * (lastrow - firstrow + 1))
        return [frame] * (lastframe - firstframe + 1)

    # get defaults from class declarations if not specified as parameter
    if firstframe is None:
        firstframe = snsr.firstframe
    if lastframe is None:
        lastframe = snsr.lastframe
    # TODO: first frame number
    if width is None:
        width = snsr.width
    if height is None:
        height = snsr.height
    if firstrow is None:
        firstrow = snsr.firstrow
    if lastrow is None:
        lastrow = snsr.lastrow
    if maxwidth is None:
        maxwidth = snsr.maxwidth
    if maxheight is None:
        maxheight = snsr.maxheight
    if bytesperpixel is None:
        bytesperpixel = snsr.bytesperpixel
    if interlacing is None:
        interlacing = snsr.interlacing

    parameters = fakeCA(
        sensor,
        firstframe,
        lastframe,
        width,
        height,
        padToFull,
        firstrow,
        lastrow,
        maxwidth,
        maxheight,
        bytesperpixel,
        interlacing,
    )

    if path is None:
        path = os.path.join(os.getcwd())
    datafile = os.path.join(path, filename)
    if filename[-3:].lower() == "txt":
        filetype = "txt"
    elif filename[-3:].lower() == "npy":
        filetype = "npy"
    # TODO: return empty frames if error

    if filetype == "npy":
        expectedlength = (lastframe - firstframe + 1) * (lastrow - firstrow + 1) * width
        try:
            f = np.load(datafile)
            padding = expectedlength - len(f)
            if padding:
                logging.warning(
                    "{logwarn}loadDumpedData: Payload is shorter than expected."
                    " Padding with '0's".format(logwarn=parameters.logwarn)
                )
                f = np.pad(f, (0, padding), "constant", constant_values=(0))
            frames = generateFrames(parameters, f, columns)
            return parameters, frames

        except OSError as err:
            logging.error(
                "{logerr}loadDumpedData: OS error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=err)
            )
            return parameters, buildEmptyFrames()
        except:
            logging.error(
                "{logerr}loadDumpedData: Unexpected error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=str(sys.exc_info()[0]))
            )
            return parameters, buildEmptyFrames()
    # if filetype is not explicitly npy, try loading as text
    else:
        # Payload size as string implied by provided parameters
        expectedlength = (
            4 * (lastframe - firstframe + 1) * (lastrow - firstrow + 1) * width
        )

        try:
            f = open(datafile, "r")
            s = f.read()

            padding = expectedlength - len(s)
            if padding:
                logging.warning(
                    "{logwarn}loadDumpedData: Payload is shorter than expected."
                    " Padding with '0's".format(logwarn=parameters.logwarn)
                )
                s = s.ljust(expectedlength, "0")

            frames = generateFrames(parameters, s)
            return parameters, frames

        except OSError as err:
            logging.error(
                "{logerr}loadDumpedData: OS error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=err)
            )
            return parameters, buildEmptyFrames()
        except ValueError:
            logging.error(
                "{logerr}loadDumpedData: Could not convert data to an integer."
                " Returning empty frames.".format(logerr=parameters.logerr)
            )
            return parameters, buildEmptyFrames()
        except:
            logging.error(
                "{logerr}loadDumpedData: Unexpected error: {err}. Returning empty"
                " frames.".format(logerr=parameters.logerr, err=str(sys.exc_info()[0]))
            )
            return parameters, buildEmptyFrames()


def saveTiffs(
    self,
    frames,
    path=None,
    filename="Frame",
    prefix=None,
    index=None,
):
    """
    Save numpy array or list of numpy arrays or single array to disk as individual
      tiffs, with frame number appended to filename. If used for standalone, use the
      parameter object returned by loadDumpedData() as the first argument

    Args:
        self: during normal operation, is cameraAssembler object. During offline
          operation, is the parameter object returned by loadDumpedData()
        frames: numpy array or list of numpy arrays
        path: save path, defaults to './output'
        filename: defaults to 'Frame' followed by frame number
        prefix: prepended to 'filename', defaults to time/date
          (e.g. '160830-124704_')
        index: number to start frame numbering

    Returns:
        Error string
    """
    # logging.info("INFO" + self.logtag + ": saveTiffs")
    logging.info("{loginfo}: saveTiffs".format(loginfo=self.loginfo))
    err = ""
    if path is None:
        path = os.path.join(os.getcwd(), "output")
    if prefix is None:
        prefix = datetime.now().strftime("%y%m%d-%H%M%S%f")[:-5] + "_"
    if not os.path.exists(path):
        os.makedirs(path)
    if index is None:
        firstnum = self.sensor.firstframe
    else:
        firstnum = index

    # if this is a text string from fast readoff, do the numpy conversion now
    if isinstance(frames[0], str):
        frames = generateFrames(frames)

    framestemp = np.copy(frames)
    if np.issubdtype(type(framestemp[0]), np.number):
        # if type(framestemp[0]) is np.uint16:
        # single frame needs to be a list containing one frame
        framestemp = [framestemp]
    for idx, frame in enumerate(framestemp):
        if idx < len(framestemp) / 2:
            interlacing = self.sensor.interlacing[0]
        else:
            interlacing = self.sensor.interlacing[1]
        try:
            if self.padToFull:
                frame = np.reshape(
                    frame, (self.sensor.maxheight // (interlacing + 1), -1)
                )
            else:
                frame = np.reshape(
                    frame,
                    (
                        (self.sensor.lastrow - self.sensor.firstrow + 1)
                        // (interlacing + 1),
                        -1,
                    ),
                )
            frameimg = Image.fromarray(frame, "I;16")
            namenum = filename + "_%d" % firstnum
            tifpath = os.path.join(path, prefix + namenum + ".tif")
            frameimg.save(tifpath)
            firstnum += 1
        except Exception:
            err = "saveTiffs: unable to save images"
            # logging.error("ERROR" + self.logtag + ": " + err)
            logging.error("{logerr}: {err}".format(logerr=self.logerr, err=err))
    return err


def plotFrames(self, frames, index=None):
    """
    Plot frame or list of frames as individual graphs.

    Args:
        self: during normal operation, is cameraAssembler object. During offline
          operation, is the parameter object returned by loadDumpedData()
        frames: numpy array or list of numpy arrays
        index: number to start frame numbering

    Returns:
        Error string
    """
    # logging.info(self.loginfo + "plotFrames: index = " + str(index))
    logging.info(
        "{loginfo}: plotFrames: index = {index}".format(
            loginfo=self.loginfo, index=index
        )
    )
    err = ""
    if index is None:
        nframe = self.sensor.firstframe
    else:
        nframe = index

    if not isinstance(frames, list):
        frames = [frames]

    # if this is a text string from fast readoff, do the numpy conversion now
    if isinstance(frames[0], str):
        frames = generateFrames(frames)

    framestemp = np.copy(frames)
    for idx, frame in enumerate(framestemp):
        if idx < len(framestemp) / 2:
            interlacing = self.sensor.interlacing[0]
        else:
            interlacing = self.sensor.interlacing[1]
        try:
            if self.padToFull:
                frame = np.reshape(
                    frame, (self.sensor.maxheight // (interlacing + 1), -1)
                )
            else:
                frame = np.reshape(
                    frame,
                    (
                        (self.sensor.lastrow - self.sensor.firstrow + 1)
                        // (interlacing + 1),
                        -1,
                    ),
                )
        except:
            err = "{logerr}plotFrames: unable to plot frame".format(logerr=self.logerr)
            logging.error(err)
            continue
        plt.imshow(frame, cmap="gray")
        name = "Frame %d" % nframe
        plt.title(name)
        plt.show()
        nframe += 1
    return err


#  TODO: separate images for hemispheres with different timing


def partition(self, frames, columns):
    """
    Extracts interlaced frames and divides images by hemispheres. If interlacing does
      not evenly divide the height, remainder lines will be dropped

    Args:
        self: during normal operation, is sensor object. During offline
          operation, is the parameter.sensor object returned by loadDumpedData()
        frames: list of full-sized frames
        columns: 1 for single image, 2 for separate hemisphere images

    Returns: list of deinterlaced frames
    """
    logging.debug(
        "{logdebug}partition: columns = {columns}, interlacing = {interlacing}".format(
            logdebug=self.logdebug, columns=columns, interlacing=self.sensor.interlacing
        )
    )

    def unshuffle(frames, ifactor):
        warntrimmed = False
        if self.padToFull:
            newheight = self.sensor.maxheight // (ifactor + 1)
            if newheight != (self.sensor.maxheight / (ifactor + 1)):
                warntrimmed = True
        else:
            newheight = self.sensor.height // (ifactor + 1)
            if newheight != (self.sensor.height / (ifactor + 1)):
                warntrimmed = True

        if warntrimmed:
            logging.warning(
                "{logwarn} partition: interlacing setting requires dropping of lines to"
                " maintain consistent frame sizes ".format(logwarn=self.logwarn)
            )
        delaced = []
        for frame in frames:
            for sub in range(ifactor + 1):
                current = np.zeros((newheight, self.sensor.width // columns), dtype=int)
                for line in range(newheight):
                    current[line] = frame[(ifactor + 1) * line + sub]
                delaced.append(current)
        nframes = self.sensor.lastframe - self.sensor.firstframe + 1
        resorted = [None] * len(delaced)
        for sub in range(ifactor + 1):
            for idx, frame in enumerate(frames):
                resorted[sub * nframes + idx] = delaced[idx * (ifactor + 1) + sub]
        return resorted

    if self.sensor.interlacing[0] != self.sensor.interlacing[1]:
        columns = 2  # true even if not explicitly requested by readoff
    if columns == 1:
        if self.sensor.interlacing == [0, 0]:  # don't do anything
            return frames
        else:
            return unshuffle(frames, self.sensor.interlacing[0])
    else:
        # reshape frame into the proper shape, then split horizontally
        if self.padToFull:
            framesab = [
                np.hsplit(frame.reshape(self.sensor.maxheight, -1), 2)
                for frame in frames
            ]
        else:
            framesab = [
                np.hsplit(
                    frame.reshape((self.sensor.lastrow - self.sensor.firstrow + 1), -1),
                    2,
                )
                for frame in frames
            ]
        framesa = [hemis[0] for hemis in framesab]
        framesb = [hemis[1] for hemis in framesab]
    if self.sensor.interlacing == [0, 0]:
        return framesa + framesb
    else:
        return unshuffle(framesa, self.sensor.interlacing[0]) + unshuffle(
            framesb, self.sensor.interlacing[1]
        )


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.47 C:/Users/hill35/git/camera_python/nsCamera/utils/Packet.py File Reference

Classes

- class nsCamera.utils.Packet.Packet

Namespaces

- namespace nsCamera
- namespace nsCamera.utils
- namespace nsCamera.utils.Packet

7.48 Packet.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Packet object for communication with boards

Author: Brad Funsten (funsten1@llnl.gov)
Author: Jeremy Hill (hill35@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""

from __future__ import absolute_import

# mport binascii
import sys

from nsCamera.utils import crc16pure
from nsCamera.utils.misc import str2bytes


class Packet:
    """
    Packet object for communication with boards. See ICD for details.

    Single Command/Response packet:
    +----------+-----------+-----------+------------+-----------+
    | 16 bits  |  4 bits   |  12 bits  |   32 bits  |  16 bits  |
    | Preamble |  Command  |  Address  |    Data    |   CRC16   |
    +----------+-----------+-----------+------------+-----------+

    Read Burst Response packet:

    +----------+-----------+---------------+---------------+
    | 16 bits  |  4 bits   |    4 bits     |    16 bits    %
    | Preamble |  Command  |  Sub-command  |  Sequence ID  %
    +----------+-----------+---------------+---------------+
                                            +-----------------+------------+-----------+
                                            %      16 bits    |  Variable  |  16 bits  |
                                            %  Payload Length |  Payload   |   CRC16   |
                                            +-----------------+------------+-----------+

    """

    def __init__(
        # NOTE: 'numerical' components are handled as hex strings
        self,
        preamble="aaaa",
        cmd="0",
        addr="",
        data="00000000",
        seqID="",
        payload_length="",
        payload="",
        crc="",
    ):
        self.PY3 = sys.version_info > (3,)
        self.preamble = preamble  # 16 bit packet preamble
        self.cmd = str(cmd)  # 4 bit command packet
        self.addr = addr.zfill(3)  # 12 bit address packet
        self.data = data.zfill(8)  # 32 bit data packet
        # 16 bit sequence ID packet (only Read Burst)
        self.seqID = seqID
        # 16 bit payload packet (only Read Burst)
        self.payload_length = payload_length
        # variable payload packet (only Read Burst) for now it's 16 bits
        self.payload = payload
        # 16 bit CRC-CCIT (XModem) packet
        self.crc = crc
        self.type = ""
        if self.crc == "":  # check if packet to be sent needs crc appended
            self.crc = self.calculateCRC()

    def pktStr(self):
        """
        Generate hexadecimal string form of packet

        Returns:
            packet as hexadecimal string without '0x'
        """
        if self.seqID != "":
            # Read burst response
            packetparts = [
                self.preamble,
                self.cmd,
                self.seqID,
                self.payload_length,
                self.payload,
                self.crc,
            ]
        else:
            # Single Command/Response response
            packetparts = [self.preamble, self.cmd, self.addr, self.data, self.crc]
        stringparts = [
            part.decode("ascii") if isinstance(part, bytes) else part
            for part in packetparts
        ]
        out = "".join(stringparts)
        return out

    def calculateCRC(self):
        """
        Calculate CRC-CCIT (XModem) (2 bytes) from 8 byte packet for send and rcv

        Returns:
            CRC as hexadecimal string without '0x'
        """
        preamble = self.preamble
        crc = self.crc
        self.crc = ""
        self.preamble = ""

        CRC_dec = crc16pure.crc16xmodem(str2bytes(self.pktStr()))
        # input = int type decimal, output = hex string with 0x at the beginning
        CRC_hex_0x = "0x%0.4X" % CRC_dec
        # make all hex letters lower case for comparison
        CRC_hex = CRC_hex_0x.lower()
        # input = hex string with 0x at the beginning, output = hex str with 0x removed
        CRC_hex = CRC_hex[2:]
        self.preamble = preamble
        self.crc = crc
        return CRC_hex

    def checkCRC(self):
        """
        Returns: boolean, True if CRC check passes
        """
        return self.calculateCRC() == self.crc

    def checkReadPacket(self, resppkt):
        """
        Confirm that Read Single occurred without error
        Args:
            resppkt: response packet

        Returns:
            tuple (error string, response packet as string)
        """
        err = ""
        if int(resppkt.cmd.upper(), 16) - int(self.cmd.upper(), 16) != 0x8:
            err = "invalid command; "
        if resppkt.addr.upper() != self.addr.upper():
            err += "invalid address; "
        if resppkt.crc.upper() != resppkt.calculateCRC().upper():
            err += "invalid CRC; "
        return err, resppkt.pktStr()

    def checkResponsePacket(self, resppkt):
        """
        Confirm that Write Single occurred without error
        Args:
            resppkt: response packet

        Returns:
            tuple (error string, response packet as string)
        """
        err = ""
        if int(resppkt.data, 16) & 1:
            err += "Checksum error; "
        if int(resppkt.data, 16) & 2:
            err += "Invalid command / command not executed; "
        err1, rval = self.checkReadPacket(resppkt)
        err += err1
        return err, rval

    def checkResponseString(self, respstr):
        """
        Checks response string for error indicators
        Args:
            respstr: packet as hexadecimal string

        Returns:
            tuple (error string, response packet string)
        """
        respstring = respstr.decode(encoding="UTF-8")
        resppkt = Packet(
            preamble=respstring[0:4],
            cmd=respstring[4],
            addr=respstring[5:8],
            data=respstring[8:16],
        )

        if resppkt.cmd == "8":
            # verify response to write command
            err, rval = self.checkResponsePacket(resppkt)
        elif resppkt.cmd == "9":
            err, rval = self.checkReadPacket(resppkt)  # verify response to read command
        else:
            err = "Packet command invalid; "
            rval = ""
        return err, rval


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

7.49 C:/Users/hill35/git/camera_python/nsCamera/utils/Subregister.py File Reference

Classes

- class nsCamera.utils.Subregister.SubRegister

Namespaces

- namespace nsCamera
- namespace nsCamera.utils
- namespace nsCamera.utils.Subregister

7.50 Subregister.py

Go to the documentation of this file.

```
# -*- coding: utf-8 -*-
"""
Subregister object represents a subset of a full register

Author: Matthew Dayton (dayton5@llnl.gov)
Author: Jeremy Martin Hill (jerhill@llnl.gov)

Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.

Version: 2.1.2 (February 2025)
"""


class SubRegister:
    """
    Represents a subset of a 32-bit register [31..0] starting at 'start_bit' consisting
      of 'width' bits. Consistent with the ICD usage, start_bit is MSB e.g., for [7..0],
      the start_bit is '7'.
    """

    def __init__(
        self,
        board,
        name,
        register,
        start_bit=31,
        width=8,
        writable=False,
        value=255,
        minV=0,
        maxV=5,
    ):
        self.name = name
        self.register = register
        self.addr = board.registers[register]
        self.start_bit = start_bit
        self.width = width
        self.value = value
        self.max_value = 2**width - 1  # used to normalize the input values to 1
        self.min = 0
        self.max = self.max_value
        self.writable = writable
        self.minV = minV
        self.maxV = maxV
        # resolution should be reset after init if actual min and max are different
        self.resolution = (1.0 * maxV - minV) / self.max_value


"""
Copyright (c) 2025, Lawrence Livermore National Security, LLC.  All rights reserved.
LLNL-CODE-838080

This work was produced at the Lawrence Livermore National Laboratory (LLNL) under
contract no. DE-AC52-07NA27344 (Contract 44) between the U.S. Department of Energy (DOE)
and Lawrence Livermore National Security, LLC (LLNS) for the operation of LLNL.
'nsCamera' is distributed under the terms of the MIT license. All new contributions must
be made under this license.
"""
```

Index