CgosPy Manual

Revision History:

Revision	Date (dd.mm.yy)	Author	Changes
1.0	09.09.16	LKA	Initial release
1.2	05.11.19	LKA	Updated for Python 3 support Corrections for wdogSetConfigStruct

Notes:

- Please read the README.txt file in the package root folder for installation instructions.
- Examples for using the Python shell are highlighted with grey background color.
- For general usage of the CGOS API please refer to the congatec operating system (CGOS) API software developers guide (CGOSAPImXX.pdf)

What is CgosPy?

CgosPy is a Python package containing the cgos module which provides ctypes wrapper functions for accessing the CGOS API.

cgos.py convenience Functions for opening and closing the CGOS interface:

openif()

Initializes, installs and opens the CGOS interface.

Arguments:

none

Returns:

True - success

False - error

```
>>> import cgos as cg
>>> cg.openif()
True
```

closeif()

Closes the CGOS interface.

Arguments:

none

Returns:

True - success

False - error

```
>>> import cgos as cg
>>> cg.closeif()
True
```

Function Group CgosLib

libGetVersion()

Returns the version of the CGOS API library. This 32 bit version number contains the 16 bit version number in the high word and a build or subversion number in the low word.

Arguments:

none

Returns:

Version number of the CGOS API library

```
>>> import cgos as cg

>>> cg.libGetVersion()

16973845L
```

libInitialize()

Initializes the CGOS API library.

Arguments:

none

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.libInitialize()

True
```

libUninitialize()

De-initializes the CGOS API library and removes it from memory.

Arguments:

none

Returns:

True - success

False - error

```
>>> import cgos as cg
>>> cg.libUninitialize()
False
>>> cg.openif()
True
>>> cg.libUninitialize()
True
>>> cg.closeif()
True
```

libIsAvailable()

Checks if the CGOS API library has already been initialized by a prior call to function libInitialize().

Arguments:

none

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.libIsAvailable()

False

>>> cg.openif()

True

>>> cg.libIsAvailable()

True

>>> cg.closeif()

True
```

libInstall(install)

This function can be used to install the low level CGOS driver if a prior call of libInitialize() failed.

Arguments:

1 - installs the low level CGOS driver

0 - removes the low level CGOS driver

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.libInstall(1)

True
```

libGetDrvVersion()

Returns the version of the low level CGOS driver.

Arguments:

none

Returns:

version of the low level CGOS driver

```
>>> import cgos as cg

>>> print (cg.libGetDrvVersion())

0
```

libGetLastError()

Returns the last known error code of the low level CGOS driver. Notice that this function really delivers the code of the last known CGOS driver error and not the result of the last CGOS API function call. A succeeding CGOS API call doesn't affect the return value of this function.

The following error codes are currently defined:

description	error code
generic error	-1 (0xFFFF FFFF)
invalid parameter	-2 (0xFFFF FFFE)
function not found	-3 (0xFFFF FFFD)
read error	-4 (0xFFFF FFFC)
write error	-5 (0xFFFF FFFB)
timeout	-6 (0xFFFF FFFA)

Arguments:

none

Returns:

error code

```
>>> import cgos as cg

>>> print (cg.libGetLastError())

0
```

Function Group CgosBoard

boardCount(dwClass,dwFlags)

Returns the number of installed CGOS compliant boards with the specified board class dwClass. In case of dwClass is 0, the total number of boards in the system will be returned.

Arguments:

dwClass - board class

dwFlags - flags

Returns:

number of boards

```
>>> import cgos as cg

>>> cg.openif()

True

>>> CGOS_BOARD_CLASS_CPU = 0x00000001

>>> CGOS_BOARD_CLASS_VGA = 0x00000002

>>> CGOS_BOARD_CLASS_IO = 0x0000004

>>> CGOS_BOARD_OPEN_FLAGS_DEFAULT = 0

>>> CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY = 1

>>> # Get the total number of available boards

... cg.boardCount(0,0)

1L

>>> # Get the number of primary CPU boards

... cg.boardCount(CGOS_BOARD_CLASS_CPU, CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY)

1L
```

```
>>> # Get the number of VGA boards
... cg.boardCount(CGOS_BOARD_CLASS_VGA, CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY)
0L
>>> # Get the number of boards with CPU functionality
... cg.boardCount(CGOS_BOARD_CLASS_CPU, CGOS_BOARD_OPEN_FLAGS_DEFAULT)
1L
>>> # Get the number of boards with VGA functionality
... cg.boardCount(CGOS_BOARD_CLASS_VGA, CGOS_BOARD_OPEN_FLAGS_DEFAULT)
1L
>>> cg.closeif()
True
```

boardOpen(dwClass, dwNum, dwFlags)

Opens a board.

Arguments:

dwClass - board class

dwNum - board number

dwFlags - board flags

Returns:

True - success

False - error

```
>>> cg.boardOpen(0,0,0)
True
```

boardOpenByName(pszName)

Opens a board by its name.

Arguments:

pszName - string containing the board name

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> boardname = cg.boardGetName()

>>> print (boardname)

QA31

>>> cg.boardOpenByName(boardname)

True

>>> cg.closeif()

True
```

boardClose()

Closes a board which was previously opened by either boardOpen() or boardOpenByName().

Arguments:

none

Returns:

True - success

False - error

boardGetName()

Determines the name of the board.

Arguments:

none

Returns:

- string containing the board name

- False - error

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.boardGetName())
QA31
>>> cg.closeif()
True
```

boardGetInfo()

Gets the board information of a CGOS API compliant board.

Arguments:

none

Returns:

Dictionary containing the CGOSBOARDINFO structure

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> result = cg.boardGetInfo()

>>> print (result)

{'dwClasses': 3L, 'wBiosInterfaceRevision': 256, 'szBoardSub': 'QA30',
'szEAN': '04250186185831', 'stManufacturingDate': {'wYear': 2013, 'wMilliseconds':
0, 'wHour': 0, 'wMonth': 9, 'wMinute': 0, 'wSecond': 0, 'wDay': 17, 'wDayOfWeek':
0}, 'dwSize': 184L, 'szManufacturer': 'OEM ', 'wProductRevision': 16690,
'stLastRepairDate': {'wYear': 0, 'wMilliseconds': 0, 'wHour': 0, 'wMonth': 0,
'wMinute': 0, 'wSecond': 0, 'wDay': 0, 'wDayOfWeek': 0}, 'dwFlags': 0L,
'dwManufacturer': 13L, 'wSystemBiosRevision': 278, 'szSerialNumber':
'000001390422', 'szPartNumber': '015104', 'dwRepairCounter': 0L, 'szReserved': '',
'szBoard': 'QA31', 'dwPrimaryClasses': 1L, 'wBiosInterfaceBuildRevision': 1}

>>> print (result['szSerialNumber'])

000001390422

>>> print (result['stManufacturingDate']['wYear'])

2013

>>> cg.closeif()

True
```

boardGetBootCounter()

Gets the current value of the boot counter.

Arguments:

none

Returns:

boot counter value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.boardGetBootCounter())

88962

>>> cg.closeif()

True
```

boardGetRunningTimeMeter()

Gets the current running time of the board measured in hours.

Arguments:

none

Returns:

running time meter value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.boardGetRunningTimeMeter())

861

>>> cg.closeif()

True
```

Function Group CgosVga

vgaCount()

Gets the number of installed VGA boards in the system.

Arguments:

none

Returns:

number of installed VGA boards

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.vgaCount())

4

>>> cg.closeif()

True
```

vgaGetBacklight(dwUnit)

Gets the backlight brightness value. The range of the value is between 0 and CGOS_VGA_BACKLIGHT_MAX (100), respectively 0 and 100%.

Arguments:

none

Returns:

backlight value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.vgaGetBacklight(0))

75

>>> cg.closeif()

True
```

vgaSetBacklight(dwUnit, dwSetting)

Sets the backlight brigthness value. This value must be between 0 and CGOS_VGA_BACKLIGHT_MAX (100), respectively 0 and 100%.

Arguments:

dwUnit - unit selector

dwSetting - backlight value (between 0 and 100)

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.vgaGetBacklight(0))

75

>>> cg.vgaSetBacklight(0,100)

True

>>> cg.vgaGetBacklight(0)

100L
```

```
>>> cg.vgaSetBacklight(0,101)

False

>>> cg.closeif()

True
```

vgaGetBacklightEnable (dwUnit)

Returns the state of the LCD's backlight.

Arguments:

dwUnit - unit selector

Returns:

backlight value; 1 = backlight is on; 0 = backlight is off.

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.vgaGetBacklightEnable(0))

1

>>> cg.closeif()

True
```

vgaSetBacklightEnable (dwUnit, dwSetting)

Turns the backlight on or off.

Arguments:

dwUnit - unit selector

dwSetting - 1 = backlight on, 0 = backlight off

Returns:

backlight value; 1 = backlight is on; 0 = backlight is off.

False - error

vgaGetInfo(dwUnit)

Returns the contents of the CGOSVGAINFO structure for the specified unit.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the CGOSVGAINFO structure

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.vgaGetInfo(0))

{'dwNativeHeight': 0L, 'dwRequestedWidth': 0L, 'dwSize': 40L,
'dwRequestedHeight': 0L, 'dwMaxBacklight': 100L, 'dwFlags': 0L, 'dwRequestedBpp':
0L, 'dwMaxContrast': 100L, 'dwType': 131072L, 'dwNativeWidth': 0L}

>>> cg.closeif()

True
```

Function Group CgosStorageArea

storageAreaCount(dwUnit)

Gets the number of installed storage areas of the board.

Arguments:

dwUnit - unit selector

Returns:

number of storage areas

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.storageAreaCount(0))

5

>>> cg.closeif()

True
```

storageAreaType(dwUnit)

Returns the types of the storage areas of the board. This function is also used to determine the pure type of a dedicated storage area (by separating it from the unit number).

Arguments:

dwUnit - unit selector

Returns:

Storage Area Type

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.storageAreaType(0))

327680

>>> for i in range (0, cg.storageAreaCount()): print
hex(cg.storageAreaType(i))

...

0x50000L

0x10000L

0x40000L

0x20000L

0x30000L

>>> cg.closeif()

True
```

storageAreaSize(dwUnit)

Returns the size of the storage area in bytes.

Arguments:

dwUnit - unit selector

Returns:

storage area size in bytes

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.storageAreaSize(0))
```

```
64

>>> for i in range (0, cg.storageAreaCount()): print (cg.storageAreaSize(i))

...

64

32

32

65536

256

>>> cg.closeif()

True
```

storageAreaBlockSize(dwUnit)

Returns the block size of a storage area block in bytes.

Arguments:

dwUnit - unit selector

Returns:

storage area block size

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.storageAreaBlockSize(0))

64

>>> for i in range (0, cg.storageAreaCount()): print
(cg.storageAreaBlockSize(i))

...

64
```

```
32

32

65536

256

>>> cg.closeif()

True
```

storageAreaRead(dwUnit, dwOffset, dwLen)

Reads dwLen bytes from the storage area.

Arguments:

dwUnit - unit selector

dwOffset - offset

dwLen - number of bytes to be read

Returns:

List containing the values read back

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # read 10 bytes from area 1 with offset 0

... cg.storageAreaRead(1,0,10)

['0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0']

>>> cg.closeif()

True
```

storageAreaWrite(dwUnit, dwOffset, pBytes, dwLen)

Writes dwLen bytes from the buffer pBytes to the storage area .

Arguments:

dwUnit - unit selector

dwOffset - offset

pBytes - list containing the write buffer

dwLen - number of bytes to be written from the write buffer

Returns:

True - success

False - error

```
>>> import cgos as cg
>>> cg.openif()
True
>>> # read 10 bytes from area 1 with offset 0
... cg.storageAreaRead(1,0,10)
['0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0']
>>> # create a writebuffer
... wbuf = [0x12, 0x34, 0x56]
>>> # write the writebuffer into area 1 with offset 0
... cg.storageAreaWrite(1,0,wbuf,len(wbuf))
True
>>> # read 10 bytes from area 1 with offset 0
... cg.storageAreaRead(1,0,10)
['0x12', '0x34', '0x56', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0']
>>> cg.closeif()
True
```

storageAreaErase(dwUnit, dwOffset, dwLen)

Erases dwLen bytes from the storage area starting at offset dwOffset.

Arguments:

dwUnit - unit selector

dwOffset - offset

dwLen - number of bytes to be erased

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # read 10 bytes from area 1 with offset 0

... cg.storageAreaRead(1,0,10)

['0x12', '0x34', '0x56', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0']

>>> # erase 10 bytes from unit 1 with offset 0

... cg.storageAreaErase(1,0,10)

True

>>> # read 10 bytes from area 1 with offset 0

... cg.storageAreaRead(1,0,10)

['0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff',
'0xff']

>>> cg.closeif()

True
```

storageAreaEraseStatus(dwUnit, dwOffset, dwLen)

Returns the status of the current area erase progress.

Arguments:

dwUnit - unit selector

dwOffset - offset

dwLen - number of bytes to be read

Returns:

lpStatus:

0 - Erasing the specified area finished successfully

1 - Erasing in progress

2 - Erase error

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # print the erase status from area 1 at offset 0, length 10 bytes

... print (cg.storageAreaEraseStatus(1,0,10))

0

>>> cg.closeif()

True
```

storageAreaLock(dwUnit, pBytes)

This function is used to write protect a storage area. Write access to a locked storage area is rejected as long as the area is unlocked with the storageAreaUnlock function call. Read access to a locked storage area isn't affected by this mechanism and therefore still permitted at any time. This kind of implementation allows you to set up features such as protected custom serial numbers or the selective enabling of software features. This function fails if the selected area is already locked. The current release of the software only supports the locking of storage areas of type CGOS_STORAGE_AREA_EEPROM. The protection mechanism for this type expects a secret string with up to 6 characters.

Arguments:

dwUnit - unit selector

pBytes - secret string

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> for i in range (0, cg.storageAreacount()): print
(hex(cg.storageAreaType(i)))

...

0x50000L

0x10000L

0x40000L

0x20000L

0x30000L

>>> # area 1 has type 0x10000 (EEPROM), thus it can be locked:

... cg.storageAreaLock(1,"congatec")

True
```

```
>>> # if we lock it again, we get a 'False' because it is already locked:

... cg.storageAreaLock(1,"congatec")

False

>>> # if we lock an area which is not of type EEPROM we also get a 'False'

... cg.storageAreaLock(0,"congatec")

False

>>> cg.closeif()

True
```

storageAreaUnlock(dwUnit, pBytes)

This function is used to unlock a write protected storage area that was previously locked using storageAreaLock(). To unlock an area the secret string must be exactly the same as the string that was used to lock the area. If the attempt to unlock an area fails, any further try to unlock the area requires a preceding power off/on cycle of the system. See CGOS API manual section 5.5.9 CgosStorageAreaLock for additional details. This function fails if the selected area is already unlocked.

Arguments:

dwUnit - unit selector

pBytes - secret string

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> for i in range (0, cg.StorageAreaCount()): print
(hex(cg.storageAreaType(i)))

...
```

```
      0x50000L

      0x10000L

      0x40000L

      0x20000L

      0x30000L

      >>> # area 1 has type 0x10000 (EEPROM), thus it can be locked:

      ... cg.storageAreaLock(1,"congatec")

      True

      >>> # area 1 has been locked. Now it can be unlocked:

      ... cg.storageAreaUnlock(1,"congatec")

      True

      >>> # if we unlock again, we get a 'False' because the area is already
unlocked:

      ... cg.storageAreaUnlock(1,"congatec")

      False

      >>> cg.closeif()

      True
```

storageAreaIsLocked(dwUnit)

This function is used to determine the locking state of a storage area. It returns 'True' if the selected area is locked. It returns 'False' if the area isn't locked or if the functionality isn't implemented. See CGOS API manual section 5.5.9 CgosStorageAreaLock for additional details.

Arguments:

dwUnit - unit selector

Returns:

True - area is locked

False - area is not locked

```
>>> import cgos as cg

>>> cg.openif()

True

>>> for i in range (0, cg.storageAreaCount()): print
(hex(cg.storageAreaType(i)))

...

0x50000L

0x10000L

0x40000L

0x20000L

0x30000L

>>> # area 1 has type 0x10000 (EEPROM) -> it can be locked:

... cg.storageAreaLock(1,"congatec")

True

>>> # area 1 has been locked. Check if it is really locked

... cg.storageAreaIsLocked(1)

True

>>> # unlock area 1

... cg.storageAreaUnlock(1,"congatec")

True

>>> # check if area 1 is really unlocked

... cg.storageAreaIsLocked(1)

False

>>> cg.closeif()

True
```

Function Group CgosI2C

i2cCount()

Gets the number of installed I²C buses in the system.

Arguments:

none

Returns:

number of I²C buses

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.i2cCount())
6
>>> cg.closeif()
True
```

i2cType(dwUnit)

Gets the type of the addressed I²C bus.

Arguments:

dwUnit - unit selector

Returns:

unit type

```
>>> import cgos as cg

>>> cg.openif()

True

>>> for i in range (0, cg.i2cCount()): print (hex(cg.i2cType(i)))

...

0x10000L

0x20000L

0x30000L

0x40000L

0x20000L

0x50000L

>>> cg.closeif()

True
```

i2cIsAvailable(dwUnit)

Determines if I²C bus of type dwUnit is present.

Arguments:

dwUnit - unit selector

Returns:

True - available

False - not available

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # check if there is a i2c bus of type primary (0x10000) available

... cg.i2cIsAvailable(0x10000)
```

```
True

>>> cg.closeif()

True
```

i2cRead(dwUnit, bAddr, dwLen)

Reads dwLen subsequent bytes from the device with address bAddr at I²C bus dwUnit.

Arguments:

dwUnit - unit selector

bAddr - I²C slave address

dwLen - number of bytes to be read

Returns:

List containing the bytes read

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # read 10 bytes from bus 0, device address 0xA0

... cg.i2cRead(0,0xA0,10)

['0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff', '0xff',
'0xff']

>>> cg.closeif()

True
```

i2cWrite(dwUnit, pAddr, pBytes, dwLen)

Writes dwLen subsequent bytes from the buffer pBytes to the device with address bAddr at I²C bus dwUnit.

Arguments:

dwUnit - unit selector

pAddr - slave address

pBytes - List containing the write buffer

dwLen - Number of bytes to be written

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # create a writebuffer

... wbuf = [0x12, 0x34, 0x56]

>>> # write the whole buffer to i2c address 0xE2 attached to bus 1

... cg.i2cWrite(1, 0xE2, wbuf, len(wbuf))

True

>>> cg.closeif()

True
```

i2cReadRegister(dwUnit, bAddr, wReg)

Reads one byte from the register wReg in the device with address bAddr at I²C bus dwUnit.

Arguments:

dwUnit - unit selector

bAddr - slave address

wReg - register

Returns:

register value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Read the value of register 0x05, address 0xA0 on bus 0

... cg.i2cReadRegister(0,0xA0,0x05)

'0x0'

>>> cg.closeif()

True
```

i2cWriteRegister(dwUnit, bAddr, wReg, bData)

Writes the value of bData to the register wReg in the device with address bAddr at I²C bus dwUnit.

Arguments:

dwUnit - unit selector

bAddr - slave address

wreg - register address

bData - byte that should be written

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Read the value of register 0x05, address 0xA0 on bus 0

... cg.i2cReadRegister(0,0xA0,0x05)

'0x0'

>>> # Write 0x55 into register 0x05, address 0xA0 on bus 0

... cg.i2cWriteRegister(0,0xA0,0x05,0x55)

True

>>> # Read the value of register 0x05, address 0xA0 on bus 0

... cg.i2cReadRegister(0,0xA0,0x05)

'0x55'

>>> cg.closeif()

True
```

i2cWriteReadCombined(dwUnit, bAddr, pBytesWrite, dwLenWrite, dwLenRead)

This function combines writing to and reading from a device on the I²C bus in onestep. There will be no stop condition after writing to the device, the subsequent read cycle will be initiated with a leading start condition.
Note: in the example below, an EEPROM (address 0xA0) was used. The command i2cWriteReadCombined is not suitable for this case because the EEPROM needs about 10 milliseconds to finish a write cycle. Thus, the result is not as expected.

Arguments:

dwUnit - unit selector

bAddr - slave address

pBytesWrite - list containing the writebuffer

dwLenWrite - number of bytes to be written

dwLenRead - number of bytes to be read back

Returns:

List containing the value(s) read back

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # create a writebuffer

... wbuf = [0x01, 0x02, 0x03]

>>> # write the buffer and read 3 bytes back from bus 0, address 0xA0

... rbuf = cg.i2cWriteReadCombined(0,0xA0,wbuf,len(wbuf),3)

True

>>> print (rbuf)

['0x0L', '0x0L', '0x0L']

>>> cg.closeif()

True
```

i2cGetMaxFrequency(dwUnit)

Gets the maximum operating frequency of the I2C bus specified by unit number dwUnit in Hz.

Arguments:

dwUnit - unit selector

Returns:

frequency value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.i2cGetMaxFrequency(0))

1000000

>>> cg.closeif()

True
```

i2cGetFrequency(dwUnit)

Gets the current operating frequency of the I2C bus specified by unit number dwUnit in Hz.

Arguments:

dwUnit - unit selector

Returns:

frequency value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.i2cGetFrequency(0))

1000000

>>> cg.closeif()

True
```

i2cSetFrequency(dwUnit, pdwSetting)

Sets the current operating frequency of the I2C bus specified by unit number dwUnit in Hz. Commonly used values are 100000 and 400000.

Arguments:

dwUnit - unit selector

pdwSetting - frequency value

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.i2cGetFrequency(0))

1000000

>>> # set the frequency of bus 0 to 400 kHz

... cg.i2cSetFrequency(0,400000)

True
```

```
>>> print (cg.i2cGetFrequency(0))

400000

>>> cg.closeif()

True
```

Function Group CgosIO

ioCount()

Gets the number of installed IO units in the system. Each IO unit is able to handle up to 32 GPIs (general purpose inputs), GPOs (general purpose outputs) or GPIOs (general purpose I/Os).

Arguments:

none

Returns:

IO count

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.ioCount())

1

>>> cg.closeif()

True
```

ioIsAvailable(dwUnit)

Reads the value of the input pins of IO unit dwUnit. It's recommended to combine this value with the result of CgosIOGetDirectionCaps. See section 4.9.GPIO Functions for details

Arguments:

dwUnit - unit selector

Returns:

True - available

False - not available

```
>>> import cgos as cg

>>> cg.openif()

True

>>> cg.ioIsAvailable()

True

>>> cg.closeif()

True
```

ioRead(dwUnit)

Reads the value of the input pins of IO unit dwUnit. It's recommended to combine this value with the result of CgosIOGetDirectionCaps. See section 4.9.GPIO Functions of the CGOS API manual for details.

Arguments:

dwUnit - unit selector

Returns:

value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Read the IO Pins of Unit 0

... print cg.ioRead(0)
```

```
15

>>> print (hex(cg.ioRead(0)))

0xfL

>>> cg.closeif()

True
```

ioWrite(dwUnit, dwData)

Writes the value dwData to the output pins of IO unit dwUnit. It's recommended to combine this value with the result of CgosIOGetDirectionCaps. See section 4.9.GPIO Functions for details.

Arguments:

dwUnit - unit selector

dwData - value

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Set all output pins of unit 0 to HIGH

... cg.ioWrite(0,0xFFFFFFFF)

True

>>> # Set all output pins of unit 0 to LOW

... cg.ioWrite(0,0x00000000)

True

>>> cg.closeif()

True
```

ioGetDirectionCaps(dwUnit)

Determines the input and the output capabilities of the IO unit dwUnit. Each GPI/GPO/GPIO is represented by a bit in the variables pdwInputs and pdwOutputs. If the pin has input capabilities, the respective pin in pdwInputs is set to 1. If the pin has output capabilities, the respective pin in pdwOutputs is set to 1. If the pin has input and output capabilities, both respective bits in pdwInputs and pdwOutputs are set to 1. In this case, the data direction (if input or output) may be controlled by the CgosIOSetDirection function call. See section 4.9.GPIO Functions for details.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the IO capabilities of dwUnit

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Get the directions of unit 0

... result = cg.ioGetDirectionCaps(0)

>>> print (result)

{'pdwOutputs' : 240L, 'pdwInputs': 15L}

>>> print (hex(result['pdwInputs']))

0xfL

>>> cg.closeif()

True
```

ioGetDirection(dwUnit)

Determines the current data direction of the respective GPI/GPO/GPIO pin. A bit set to 1 in this field indicates that the respective pin is configured as an input, a bit set to 0 indicates that the respective pin is configured as an output. Notice that the binary values for pins that are not implemented are unspecified and can be 0 or 1. Therefore, it's recommended to cross check the result of CgosIOGetDirection with the result of CgosIOGetDirectionCaps.

Arguments:

dwUnit - unit selector

Returns:

direction value

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> print (cg.ioGetDirection(0))

0xffffff0fL

>>> cg.closeif()

True
```

ioSetDirection(dwUnit, dwData)

Currently not supported.

Function Group CgosWDog

wdogCount()

Returns the number of installed Watchdogs in the system.

Arguments:

none

Returns:

watchdog count

False - error

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.wdogCount())
1
>>> cg.closeif()
True
```

wdogIsAvailable(dwUnit)

Determines if the Watchdog is present.

Arguments:

dwUnit - unit selector

Returns:

True - available

False - not available

```
>>> import cgos as cg

>>> cg.openif()

True

>>> cg.wdogIsAvailable(0)

True

>>> cg.closeif()

True
```

wdogTrigger(dwUnit)

Triggers the Watchdog.

Arguments:

dwUnit - unit selector

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> cg.wdogTrigger(0)

True

>>> cg.closeif()

True
```

wdogGetConfigStruct(dwUnit)

Determines the configuration of the Watchdog.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the CGOSWDCONFIG struct

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> result = cg.wdogGetConfigStruct(0)

>>> print (result )

{'stStages2': {'dwTimeout': 0L, 'dwEvent': 0L}, 'dwSize': 48L, 'stStages1':
{'dwTimeout': 0L, 'dwEvent': 0L}, 'stStages0': {'dwTimeout': 0L, 'dwEvent': 0L},
'dwDelay': 0L, 'dwStageCount': 0L, 'dwMode': 128L, 'dwTimeout': 0L, 'dwOpMode':
0L}

>>> print (hex(result['dwMode']))

0x80L

>>> print (result['stStages0']['dwTimeout'])

0

>>> cg.closeif()

True
```

wdogSetConfigStruct(dwUnit, pConfig)

Sets the configuration of the Watchdog.

Arguments:

dwUnit - unit selector

pConfig - dictionary containing the values to be changed

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> CGOS_WDOG_MODE_STAGED = 0x80

>>> CGOS_WDOG_EVENT_BTN = 3

>>> CGOS_WDOG_EVENT_RST = 2

>>> # create a dictionary for the new configuration

... config = {}

>>> # enter the fields that should be set

... config['dwMode'] = CGOS_WDOG_MODE_STAGED

>>> config['dwStageCount'] = 2

>>> config['dwDelay'] = 30000

>>> config['dwOpMode'] = CGOS_WDOG_OPMODE_SINGLE_EVENT

>>> config['stStages0'] = {'dwEvent' : CGOS_WDOG_EVENT_BTN, 'dwTimeout' :
10000}

>>> config['stStages1'] = {'dwEvent' : CGOS_WDOG_EVENT_RST, 'dwTimeout' :
20000}

# set the configuration structure. The watchdog will be enabled if 'True'!

>>> cg.wdogSetConfigStruct(0, config)
```

```
True

# disable the watchdog again to prevent system shutdown

>>> cg.wdogDisable(0)

True

>>> cg.closeif()

True
```

wdogSetConfig(dwUnit, timeout, delay, mode)

Sets the configuration of the Watchdog. While wdogSetConfigStruct takes a complete structure, wdogSetConfig takes single values. Use wdogSetConfigStruct to benefit from the advantages of a staged Watchdog.

Arguments:

dwUnit - unit selector

timeout - value for timeout

delay - value for delay in ms

mode - watchdog mode

Returns:

True - success

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> CGOS_WDOG_MODE_REBOOT_PC = 0

>>> # generate a reset event after 30 secs and a timeout of 10 secs for unit
0

>>> cg.wdogSetConfig(0, 10000, 30000, CGOS_WDOG_MODE_REBOOT_PC)

True
```

```
>>> # disable the watchdog again to prevent a system shutdown
... cg.wdogDisable(0)
True
>>> cg.closeif()
True
```

wdogDisable(dwUnit)

Disables the Watchdog.

Arguments:

dwUnit - unit selector

Returns:

True - success

False - error

wdogGetInfo(dwUnit)

Gets the information structure of the Watchdog.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the CGOSWDINFO structure

False - error

```
>>> import cgos as cg
>>> cg.openif()
True
>>> wdinfo = cg.wdogGetInfo(0)
```

```
      >>> print (wdinfo)

      {'dwEvents': 15L, 'dwMinDelay': 0L, 'dwSize': 40L, 'dwMaxStageCount': 3L,
'dwMinTimeout': 0L, 'dwFlags': 0L, 'dwMaxDelay': 16777215L, 'dwMaxTimeout':
16777215L, 'dwType': 131072L, 'dwOpModes': 15L}

      >>> print (hex(wdinfo['dwType']))

      0x20000L

      >>> cg.closeif()

      True
```

Function Group CgosTemperature

temperatureCount()

Returns the number of installed temperature sensors in the system.

Arguments:

none

Returns:

number of temperature sensors

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.temperatureCount())
2
>>> cg.closeif()
True
```

temperatureGetInfo(dwUnit)

Gets the information structure of the specified temperature sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the CGOSTEMPERATUREINFO structure

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Get temperature info for unit 0

... result = cg.temperatureGetInfo(0)

>>> print (result)

{'dwHystLo': -1, 'dwMax': 128000, 'dwSize': 44L, 'dwHystHi': -1, 'dwAlarm':
0L, 'dwFlags': 0L, 'dwAlarmLo': -1, 'dwRes': 1000, 'dwType': 65536L, 'dwAlarmHi':
-1, 'dwMin': -127000}

>>> print (hex(result['dwType']))

0x10000L

>>> print (result['dwMax']/1000)

128

>>> cg.closeif()

True
```

temperatureGetCurrent(dwUnit)

Gets the actual value of the specified temperature sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing status and setting of the selected sensor

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> # Get current temperature of unit 1
```

```
... result = cg.temperatureGetCurrent(1)

>>> print (result)

{'pdwStatus': 1L, 'pdwSetting': 41800L}

>>> print (str(result['pdwSetting']/1000) + " °C")

41 °C

>>> cg.closeif()

True
```

Function Group CgosFan

fanCount()

Returns the number of installed fan sensors in the system.

Arguments:

none

Returns:

number of sensors

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.fanCount())
1
>>> cg.closeif()
True
```

fanGetInfo(dwUnit)

Gets the information structure of the specified fan sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing the CGOSFANINFO structure

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> result = cg.fanGetInfo(0)

>>> print (result)

{'dwHystLo': -1, 'dwMax': 65535, 'dwSize': 52L, 'dwOutMax': -1, 'dwHystHi':
-1, 'dwSpeedNom': -1, 'dwAlarm': 0L, 'dwFlags': 0L, 'dwAlarmLo': -1, 'dwType':
65536L, 'dwAlarmHi': -1, 'dwOutMin': -1, 'dwMin': 100}

>>> print (hex(result['dwType']))

0x10000L

>>> cg.closeif()

True
```

fanGetCurrent(dwUnit)

Gets the actual value of the specified fan sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing status and setting of the selected unit

False - error

```
>>> import cgos as cg

>>> cg.openif()

True

>>> result = cg.fanGetInfo(0)

>>> print (cg.fanGetCurrent(0))

{'pdwStatus': 1L, 'pdwSetting': 0L}

>>> cg.closeif()

True
```

Function Group CgosVoltage

voltageCount()

Returns the number of installed voltage sensors in the system.

Arguments:

none

Returns:

number of installed voltage sensors

```
>>> import cgos as cg
>>> cg.openif()
True
>>> print (cg.voltageCount())
3
>>> cg.closeif()
True
```

voltageGetInfo(dwUnit)

Gets the information structure of the specified voltage sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing CGOSVOLTAGEINFO structure

False - error

```
>>> import cgos as cg

>>> cg.openif()

>>> result = cg.voltageGetInfo(0)

>>> print (result)

{'dwHystLo': -1, 'dwMax': 6640, 'dwSize': 48L, 'dwHystHi': -1, 'dwAlarm':
0L, 'dwFlags': 0L, 'dwAlarmLo': -1, 'dwRes': 26, 'dwType': 524288L, 'dwAlarmHi': -
1, 'dwMin': 0, 'dwNom': 5000}

>>> print (hex(result['dwType']))

0x80000L

>>> cg.closeif()

True
```

voltageGetCurrent(dwUnit)

Gets the actual value of the specified voltage sensor.

Arguments:

dwUnit - unit selector

Returns:

Dictionary containing status and setting of the selected unit

False - error

```
>>> import cgos as cg

>>> cg.openif()

>>> result = cg.voltageGetCurrent(1)

>>> print (result)

{'pdwStatus': 1L, 'pdwSetting': 5090L}

>>> for i in range (0, cg.voltageCount()): print (cg.voltageGetCurrent(i)
['pdwSetting'])

...
```

```
4997

5090

566

>>> cg.closeif()

True
```