CGOS API

congatec operating system (CGOS) API software developer's guide

Revision 1.4

Revision History

Revision	Date (yyyy-mm-dd)	Author	Changes
1.0	2005.08.30	SML	Initial release.
1.1	2006.03.07	SML	Added section 4.8, 5.1.4, 5.10, 5.11, 5.12 Supplemented section 1 and 2.2. Replaced parameter dwType through dwUnit.
1.2	2006.10.13	SML	Added sections 5.2.7, 5.2.8, 5.5.9, 5.5.10 and 5.5.11 Supplemented section 1 and 2.2. Added API version to each CGOS function call.
1.3	2008.02.12	SML	Added sections 1.1, 2.4, 2.5, 2.6, 2.7, 3.2, 3.3, 4.9, 5.6.9, 5.6.10, 5.6.11, 5.7. Supplemented section 1, 2.1, 2.2, 2.3, 3.1, 4.5, 4.5.1, 4.8.2, 4.8.6, 5.7
1.4	2017.04.03	GWE	Updated template and operating system support. General cleanup.

Preface

This document provides information about using the CGOS API and its functions.

Disclaimer

The information contained within this document, including but not limited to any product specification, is subject to change without notice.

congatec AG provides no warranty with regard to this document or any other information contained herein and hereby expressly disclaims any implied warranties of merchantability or fitness for any particular purpose with regard to any of the foregoing. congatec AG assumes no liability for any damages incurred directly or indirectly from any technical or typographical errors or omissions contained herein or for discrepancies between the product and this document. In no event shall congatec AG be liable for any incidental, consequential, special, or exemplary damages, whether based on tort, contract or otherwise, arising out of or in connection with this document or any other information contained herein or the use thereof.

Intended Audience

This document is intended for technically qualified personnel. It is not intended for general audiences.

Electrostatic Sensitive Device

All congatec AG products are electrostatic sensitive devices and are packaged accordingly. Do not open or handle a congatec AG product except at an electrostatic-free workstation. Additionally, do not ship or store congatec AG products near strong electrostatic, electromagnetic, magnetic, or radioactive fields unless the device is contained within its original manufacturer's packaging. Be aware that failure to comply with these guidelines will void the congatec AG Limited Warranty.

Technical Support

congatec AG technicians and engineers are committed to providing the best possible technical support for our customers so that our products can be easily used and implemented. We request that you first visit our website at www.congatec.com for the latest documentation, utilities and drivers, which have been made available to assist you. If you still require assistance after visiting our website then contact our technical support department by email at support@congatec.com

Symbols

The following are symbols used in this document.

Note

Notes call attention to important information that should be observed.

Caution

Cautions warn the user about how to prevent damage to hardware or loss of data.

Warning

Warnings indicate that personal injury can occur if the information is not observed.

Copyright Notice

Copyright © 2005, congatec AG. All rights reserved. All text, pictures and graphics are protected by copyrights. No copying is permitted without written permission from congatec AG.

congatec AG has made every attempt to ensure that the information in this document is accurate yet the information contained within is supplied "as-is".

Trademarks

Product names, logos, brands, and other trademarks featured or referred to within this user's guide, or the congatec website, are the property of their respective trademark holders. These trademark holders are not affiliated with congatec AG, our products, or our website.

Terminology

Term	Description
GB	Gigabyte
GHZ	Gigahertz
KB	Kilobyte
MB	Megabyte
Mbit	Megabit
kHz	Kilohertz
MHz	Megahertz
N.C.	Not connected
N.A.	Not available
T.B.D.	To be determined

Contents

1 Introduction

Certain hardware features found on congatec AG computer on modules (COMs) and single board computers (SBCs) are only accessible through the use of a specialized API developed by congatec AG called CGOS API (congatec operating system application programming interface). The CGOS API provides access to these features in a hardware independent manner when using common 32-bit or 64-bit operating systems. The interface works under any version of Windows and Linux and may as well be supported on other operating systems.

By the time this document was written, CGOS API support was provided for the following operating systems:

- Microsoft® Windows® 10 (32/64-bit)
- Microsoft® Windows® 10 IoT Enterprise (32/64-bit)
- Microsoft® Windows® 10 IoT Core (32/64-bit)
- Microsoft® Windows® 8 (32/64-bit)
- Microsoft® Windows® Embedded 8 Standard (32/64-bit)
- Microsoft® Windows® 7 (32/-64bit)
- Microsoft® Windows® Embedded Standard 7 (32/64bit)
- Microsoft® Windows® CE 6.0
- Microsoft® Windows® Embedded Compact 7
- Microsoft® Windows® Embedded Compact 2013
- Linux (Yocto 2.x and all major Linux distributions based on kernel versions 3.x and 4.x)
- UEFI Shell

More operating systems might be added over time whereas others might be removed from the list. In doubt please contact congatec support to obtain the latest support list.

Note

This document details the CGOS API revision 1.03. All CGOS functionality is described within this document. The availability of the functions is also dependent on the features of the BIOS found on the congatec CPU board.

1.1 Architectural overview

Each congatec COM or SBC is equipped with a rich set of additional features and functionality, which are commonly used and are a “must-have” within the industrial market. These features comprise watchdog, running time meter, boot counter, I2C bus, storage areas and more.

The biggest challenge was to design a software interface that provides access to the onboard features and yet is independent from the underlying hardware while being generic and easy to handle via all of the mainstream operating systems. The customer benefits from a generic and hardware independent interface because it can easily be included in applications to gain access to the onboard functionality without any deep knowledge of the hardware details. Furthermore, from the software prospect, moving to a different CPU board (with CGEB extension) also becomes very easy and fast because the application software doesn't need to be modified at all. Finally, having a generic interface over a broad range of operating systems enables customers to create portable code.

Figure 1.

CGOS API, driver initialization

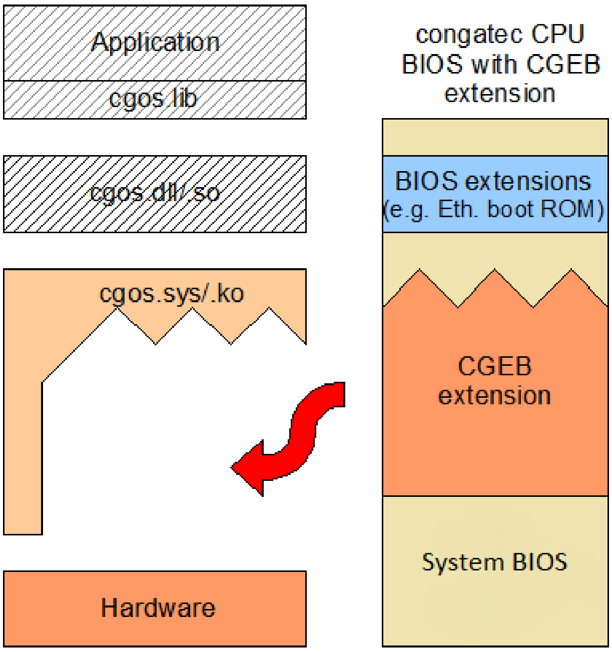
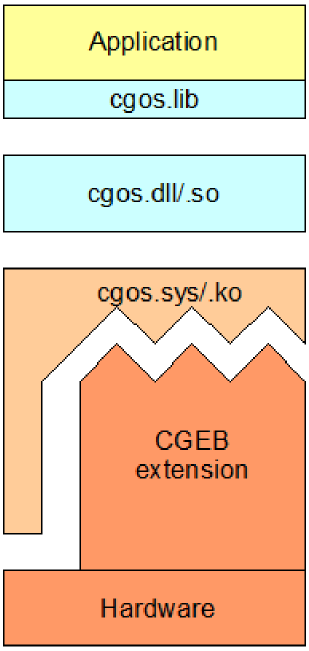


Figure 2.

CGOS API, driver up & running



The above pictures show the principle implementation of the CGOS/CGEB interface. The CGEB (congatec embedded BIOS) code is located in the board’s system BIOS. It is 32bit/64-bit native x86 object code and executable in any kind of 32bit/64-bit protected mode environment. During the driver initialization, the CGEB extension will be copied to the driver's context and becomes part of the driver. This mechanism provides independence from the hardware because all the low level hardware dependencies are already resolved by the CGEB extension code.

2 Installing the CGOS API

Please refer to the installation instructions contained in the different CGOS operating system packages for a detailed and up to date description of how to set up the CGOS support for the respective operating system.

Consider that in general it is necessary to have “Administrative Rights” in order to install the drivers included in the packages.

congatec BSPs for ‘embedded’ Windows (e.g. Windows 10 IoT Enterprise/Core, Windows Embedded Compact, …) and Linux versions (e.g. Yocto) already include all files required for CGOS API support.

In the rare case that an update of the CGOS files for these operating systems becomes necessary, please refer to the instructions in the corresponding standalone CGOS packages available on the congatec website or contact congatec support for CGOS update directions.

3 Additional Programs

3.1 CGOSDUMP

The CGOSDUMP tool prints out a lot of information about the congatec CPU board and the CGOS interface itself, such as the BIOS version, serial number of the board, the CGOS driver and library version, the running time meter, available I2C buses and storage areas plus more.

CGOSDUMP is a sample program only and was not designed to serve any applicable purpose. The source code has been provided for a better understanding of how this sample program works.

Note

CGOSDUMP is a sample program that has been created strictly for the use of software developers and should never be distributed to end users in its current form.

3.2 CGOSMON

The CGOSMON tool provides information about the different voltage and temperature sensors on the congatec CPU board.

Similar to CGOSDUMP, CGOSMON is a sample program and was not designed to serve any applicable purpose. In particular, the program might not list all sensors supported by a specific board or might find new sensors on a board defined and added after creation of the program. Thus CGOSMON might not be able to assign a meaningful, human readable sensor description and name to this sensor. The source code for CGOSMON has been provided for a better understanding of how this sample program works.

Note

CGOSMON is a sample program that has been created strictly for the use of software developers and should never be distributed to end users in its current form.

4 Programming

All the API functions are exported by the CGOS.DLL (Windows) / libcgos.so (Linux) dynamic link library.

There is one CGOS package for all Windows 7, 8 or 10 based 32/64-bit Windows versions. The installer in this package automatically selects the right driver and DLL version. For Windows CE 6.0 and Windows Embedded Compact 7 and 2013 there is a separate package including the corresponding CGOS drivers and DLLs for these operating systems.

In the INC and LIB directories of the Windows packages you will find the CGOS API header file `cgos.h` and import libraries called `CGOS.LIB` for C/C++. The `cgos.h` header file is the same for all Windows operating system variants.

The Linux CGOS support comes as one 32bit and one 64bit open source package with build and installation guidelines suitable for any 32bit/64bit Linux version based on kernels 3.x and 4.x. Each package includes its own `Cgos.h` CGOS API header file.

In addition to these packages, congatec also offers CGOS support for UEFI drivers or UEFI Shell applications, .Net/C# packages for Windows and Python packages for Windows and Linux. Please refer to the respective files on the congatec website and the included descriptions for more information. Throughout the rest of this document only the C/C++ based Windows and Linux packages will be referred to.

Within the different operating system packages you will find the sample projects `CGOSDUMP` and `CGOSMON`, which demonstrate basic CGOS functionality. Most of the source code examples in this document are taken from `CGOSDUMP.`

4.1 Installing and Initializing the Interface

In order to use the API it is necessary to install the CGOS library and initialize the interface by using the `CgosLibInitialize` function. Additionally, it is also necessary to use the function `CgosLibUninitialize` before the application terminates. This guarantees that a proper resource cleanup has taken place before the actual termination of the application.

Code example for installing/removing the interface and library:

```
if (!CgosLibInitialize()) {
  if (!CgosLibInstall(1)) {
      //error: the driver could not be installed. Check your rights.
      exit(-1);
      }

  // the driver has been installed
  if (!CgosLibInitialize()) {
      //error: the driver still could not be opened, a reboot might be required
      exit(-1);
      }
  }

// CgosLibInitialize successful

// open board, access watchdog & VGA functions, etc.
...

// close board
...

// remove DLL
CgosLibUninitialize();
```

There are some other function calls which belong to the library management:

`CgosLibGetVersion`	determine the version of the CGOS library
`CgosLibGetDrvVersion`	determine the version of the low level CGOS driver
`CgosLibIsAvailable`	determine if the library is already installed

CgosLibGetLastError	return the last interface error
CgosLibSetLastErrorAddress	fill a variable with the last interface error

4.2 Obtaining Access to the congatec Board

Board Name

In the CGOS concept, a system consists of one or more CGOS compliant boards. A board is a physical hardware component. Each board in the system is identified by a unique board name with a maximum size of CGOS_BOARD_MAX_SIZE_ID_STRING characters.

Board Classes

The class of the board describes the functionality the board offers. Currently, there are the classes CPU, VGA, and IO. In most cases, a physical board offers more functionality than that of just one single class. For instance the conga-TC170 board offers CPU and VGA functionality. In the CGOS concept, therefore, each board has exactly one primary class and may have several secondary classes. In the case of the conga-TC170, the primary class is of type CGOS_BOARD_CLASS_CPU and the secondary class of type CGOS_BOARD_CLASS_VGA. The function CgosBoardCount might be used to determine the number of boards either for a given class or the entire system.

After successful initialization of the interface, the API functions CgosBoardOpen or CgosBoardOpenByName are used to obtain a valid board handle. The board handle is the tight relation between the CGOS driver and the application until it is closed by CgosBoardClose.

Code example for opening/closing a CGOS board:

```
// board handle
HCGOS hCgos=0;

// open the board
if (!CgosBoardOpen(0,0,0,&hCgos)) {
    //error: could not open a board
    ...
    }

// put in your code here (e.g. setup & trigger the watchdog, etc.)
...

// close
if (hCgos) CgosBoardClose(hCgos);
```

4.3 Generic Board Functions

Numerous CgosBoard* functions are designed to allow you to retrieve general board class independent information about the board.

CgosBoardGetName determines the board name for a given handle

The CgosBoardGetInfo function call is used to get the information about the current configuration and state of the board. It takes a pointer to an instance of structure CGOSBOARDINFO, which is defined as follows:

CGOSBOARDINFO

unsigned long dwSize

size of the structure itself, must be initialized with sizeof(CGOSBOARDINFO)

unsigned long dwFlags

reserved. Always set to 0.

char szReserved[CGOS_BOARD_MAX_SIZE_ID_STRING]

reserved. Always set to 0.

char szBoard[CGOS_BOARD_MAX_SIZE_ID_STRING]

the name of the board, extracted from the BIOS id

char szBoardSub[CGOS_BOARD_MAX_SIZE_ID_STRING] the sub name of the board, extracted from the manufacturing data

`char szManufacturer[CGOS_BOARD_MAX_SIZE_ID_STRING]`

the name of the board manufacturer, usually congatec

`CGOSTIME stManufacturingDate`

the date of manufacturing

`CGOSTIME stLastRepairDate`

the date of last repair

`char szSerialNumber[CGOS_BOARD_MAX_SIZE_SERIAL_STRING]`

the serial number of the board, e.g. `000000050000`

`unsigned short wProductRevision`

the product revision in ASCII notation, major revision in high-byte,

minor revision in low-byte, e.g. `0x4130` for revision A.0

`unsigned short wSystemBiosRevision`

the revision of the system BIOS, major revision in high-byte,

minor revision in low-byte, e.g. `0x0110` for revision 110

`unsigned short wBiosInterfaceRevision`

the revision of CGOS API BIOS interface, major revision in high-byte,

minor revision in low-byte, e.g. `0x0100` for revision 100

`unsigned short wBiosInterfaceBuildRevision`

the build counter of CGOS API BIOS interface, e.g. `0x001` for build 001

`unsigned long dwClasses`

this entry represents an or-ed value of all the supported board classes

see also section 4.2 subsection "Board classes" for more information

about board classes

`unsigned long dwPrimaryClass`

this entry represents the primary board class, e.g. `CGOS_BOARD_CLASS_CPU`

`unsigned long dwRepairCounter`

the repair counter

`char szPartNumber[CGOS_BOARD_MAX_SIZE_PART_STRING]`

the part number, e.g. `45287` in the case of conga-X852

`char szEAN[CGOS_BOARD_MAX_SIZE_EAN_STRING]`

the EAN code of the board

`unsigned long dwManufacturer`

the sub manufacturer of the board

`CgosBoardGetBootCounter`	delivers the boot counter value
`CgosBoardGetRunningTimeMeter`	delivers the running time of the board measured in hours

4.4 VGA Functions

Boards that belong to the VGA class utilize CgosVga* functions, which are mostly used to control LCD backlight, brightness, and contrast.

4.4.1 VGA Board Types

Following VGA board types are defined depending on the functionality:

`CGOS_VGA_TYPE_UNKNOWN`	specifies an unknown type
`CGOS_VGA_TYPE_CRT`	the board supports CRT
`CGOS_VGA_TYPE_LCD`	the board supports a local flat panel
`CGOS_VGA_TYPE_LCD_DVO`	digital display device is supported (HDMI, DVI, DP,…)
`CGOS_VGA_TYPE_LCD_LVDS`	the board supports an LVDS local flat panel
`CGOS_VGA_TYPE_TV`	the board offers TV out

4.4.2 Information Structure

The `CgosVgaGetInfo` function call is used to get the information about the current configuration and state of the VGA board. It takes a pointer to an instance of structure `CGOSVGAINFO`, which is defined as follows:

CGOSVGAINFO

`unsigned long dwSize`

size of the structure itself, must be initialized with `sizeof(CGOSVGAINFO)`

`unsigned long dwType`

see section 4.4.1 VGA Board Types

`unsigned long dwFlags`

reserved. Always set to 0.

`unsigned long dwNativeWidth`

the physical display width as it is reported from the BIOS (or 0 if unknown)

`unsigned long dwNativeHeight`

the physical display height as it is reported from the BIOS (or 0 if unknown)

`unsigned long dwRequestedWidth`

the requested display width, currently not supported

`unsigned long dwRequestedHeight`

the requested display height, currently not supported

`unsigned long dwRequestedBpp`

the requested display resolution, currently not supported

`unsigned long dwMaxBacklight`

the maximum value of the backlight setting

`unsigned long dwMaxContrast`

the maximum value of the contrast setting

`CgosVgaCount`	determines the number of VGA boards in the system
`CgosVgaGetContrast`	determines the contrast value
`CgosVgaSetContrast`	sets the contrast to the specified value
`CgosVgaGetContrastEnable`	determines the state of the contrast enable signal
`CgosVgaSetContrastEnable`	sets the state of the contrast enable signal

`CgosVgaGetBacklight`	determines the backlight value

`CgosVgaSetBacklight`	sets the backlight to the specified value
`CgosVgaGetBacklightEnable`	determines the state of the backlight enable signal
`CgosVgaSetBacklightEnable`	sets the state of the backlight enable signal

The CGOS contrast functions mainly exist for backwards compatibility. No current congatec board supports these functions.

The physical backlight controls handled by the CGOS backlight functions vary depending on the hardware implementation and software (BIOS setup) configuration of the board that is used.

`CgosVga*` functions for backlight and contrast use percentage values from 0 to 100 to indicate and control brightness and contrast.

4.5 I2C Bus Functions

congatec boards implement one or more CGOS controlled I2C or SM buses. Since the hardware implementation might change, the `CgosI2C*` functions provide an abstracted software layer to access the connected devices. This makes software handling for the customer easier because the application software can be developed independently from the CPU board and even when upgrading the board the application software shouldn't be affected.

Keep in mind that all these functions are mainly intended for controlling external I2C or SM bus devices. They shouldn't be used to access any congatec onboard devices because the addresses of these devices might differ from board to board or change in future.

Some `CgosI2C*` functions expect a `bAddr` which is the 8-bit I2C address byte as it appears on the bus. The upper 7 bits contain the real address and bit 0 is used to indicate a read/write. It should be 0 on all functions except `CgosI2CRead`. Whenever possible the byte is passed to the bus as this allows you to access some devices that are not truly I2C spec. compliant.

The `CgosI2C* Register` functions contain a `wReg` parameter that is usually an 8-bit index within the device. The remaining bits are stored into the address to allow you to easily access EEPROMs.

The functions for accessing the I2C and SM buses are `CgosI2CRead`, `CgosI2CWrite`, `CgosI2CReadRegister`, `CgosI2CWriteRegister` and `CgosI2CWriteReadCombined`.

While `CgosI2CRead` only reads one byte directly from the specified address, `CgosI2CReadRegister` addresses a specific register in the device which is followed by a subsequent read of the registers content. The same applies to `CgosI2CWrite` and `CgosI2CWriteRegister` for write accesses.

The I^{2}C bus specification defines two operating modes; the standard mode with a maximum clock frequency of 100 kHz and the fast mode with clock frequencies up to 400 kHz. congatec CPU boards are able to handle both modes. However, the higher frequencies also may require a more sophisticated hardware design (e.g. an active termination of the bus on the baseboard). The initial bus frequency is set to 100 kHz by default. With revision 1.3 of the CGOS API, three new functions have been introduced to control the clock frequency of the I2C bus:

`CgosI2CGetMaxFrequency`	returns the maximum speed of the bus
`CgosI2CGetFrequency`	returns the current speed of the bus
`CgosI2CSetFrequency`	is used to set the speed of the I2C bus

4.5.1 I2C bus types

The I2C buses are distinguished by their type:

CGOS_I2C_TYPE_PRIMARY	the primary I2C bus
CGOS_I2C_TYPE_SMB	the system management bus
CGOS_I2C_TYPE_DDC	the I2C bus of the DDC interface
CGOS_I2C_TYPE_UNKNOWN	this definition might be used in special cases

During any `CgosI2C*` function call, the pure type is located in the high word and the enumerated unit number within that pure type (if more units of the same type exist) is located in the low word of parameter `dwUnit`.

Note

During an I^{2}C bus enumeration, you may notice some I^{2}C bus types that are neither documented herein nor in the CGOS header file, e.g. 0x00040000, 0x40040000, etc. These bus types are for congatec internal use only and are not meant for customer use.

Code example for accessing the I^{2}C bus:

```
unsigned long cnt;
unsigned long dwUnit;
unsigned long dwType;
unsigned char bEEPAddr = 0xA0;
unsigned char bData;
unsigned short wReg = 0x0;

cnt = CgosI2CCount(hCgos); /* determines the amount of available I2C interfaces */

/* navigate to the correct I2C bus ... */
for (dwUnit=0; dwUnit<cnt; dwUnit++) {

    dwType = CgosI2CType(hCgos, dwUnit);
    if(dwType == CGOS_I2C_TYPE_PRIMARY)
        {
         /* read one byte from the primary I2C bus (I2C address 0xA0, register 0) */
         if(CgosI2CReadRegister(hCgos, dwUnit, (unsigned char) (bEEPAddr | 0x01),
         wReg, &bdata))
                        {
                        /* 1 byte successfully read */
                        ...
                        return;
                }
     }
}
```

4.6 Storage Area Functions

Each board is usually equipped with a number of different storage areas. They may be located in Flash, EEPROM, CMOS RAM, etc. A storage area is defined as a portion of physical memory that can provide constant storage for the user's application. Every `CgosStorageArea*` function call takes a type or a unit number as a second parameter, which identifies the affected area (see also section 5.1.4 Unit numbers)

4.6.1 Storage Area Functions

The storage areas are distinguished depending on their location in memory:

CGOS_STORAGE_AREA_EEPROM	provides access to the user EEPROM
CGOS_STORAGE_AREA_FLASH	provides access to the flash
CGOS_STORAGE_AREA_CMOS	provides access to the CMOS
CGOS_STORAGE_AREA_RAM	provides access to the user RAM

CGOS_STORAGE_AREA_UNKNOWN	this type is used to determine all installed areas (not just a certain type) during a CgosStorageAreaCount call

During any CgosStorageArea* function call, the pure type is located in the high word and the enumerated unit number within that pure type (if more units of the same type exist) is located in the low word of parameter dwUnit.

For example, to select the 2nd flash area of the board, dwUnit would be:

dwUnit = CGOS_STORAGE_AREA_FLASH | 0x01

Code example for accessing the storage areas:

```
unsigned long cnt;
unsigned long i;
unsigned long dwBlockSize;
unsigned long dwSize;
unsigned long dwUnit;

/* get information of the CGOS storage areas */
cnt=CgosStorageAreaCount(hCgos,0); /* determines the amount of available sorage areas */

/* for all storage areas ... */
for (i=0; i<cnt; i++) {

    dwUnit = CgosStorageAreaType(hCgos,i),   /* determines the storage area number */
    dwBlockSize = CgosStorageAreaBlockSize(hCgos,dwUnit), /* determines the block size
*/
    dwSize = CgosStorageAreaSize(hCgos,dwUnit)      /* determines the size of the area
*/

    /* print out storage areas values here */
    ...
}

/* read some (10) user bytes from eeprom to buffer */
unsigned long len = 10;
char buf[10];

if (CgosStorageAreaRead(hCgos, CGOS_STORAGE_AREA_EEPROM, 0, buf, len))
{
    /* 10 User-Bytes successfully read */
    ...
}
```

Observe that the input dwUnit variable for CgosStorageAreaType can be either an index (as shown in the example above) or a particular storage area type as described in section 5.1.4 Unit numbers

4.7 Watchdog

All congatec boards are equipped with a Watchdog component, which provides the opportunity to force the system into a defined state when the running application or the boot process has stopped or crashed.

Note

Refer to the application note AN3_Watchdog.pdf "congatec Watchdog features and implementation" to become more familiar with the basic Watchdog features, its implementations and the differences between the operation modes on different congatec products.

The congatec CGOS API provides the following functions, which are used to control the behavior or to get information about the state of the Watchdog:

CgosWDogCount

CgosWDogIsAvailable

CgosWDogTrigger

CgosWDogGetConfigStruct

CgosWDogSetConfigStruct

CgosWDogSetConfig

CgosWDogDisable

CgosWDogGetInfo

4.7.1 Mode

The mode defines the major behavior of the watchdog:

CGOS_WDOG_MODE_REBOOT_PC	the watchdog just restarts the board
CGOS_WDOG_MODE_STAGED	the watchdog operates in staged mode (preferred)

4.7.2 Operation Modes

In staged mode, the Watchdog might offer one or more operation modes:

CGOS_WDOG_OPMODE_DISABLED
CGOS_WDOG_OPMODE_ONETIME_TRIG
CGOS_WDOG_OPMODE_SINGLE_EVENT
CGOS_WDOG_OPMODE_EVENT_REPEAT

The supported modes can be determined through the CGOS API function call `CgosWDogGetInfo`. The returned value `CGOSWDINFO:dwOpModes` represents a bit mask of all supported modes. To check if the “repeated event mode” is supported by the board controller watchdog, the following example can be used:

```
CGOSWDINFO     dwi;
if (CgosWDogGetInfo(hCgos, CGOS_WDOG_TYPE_BC, &dwi))
{
        if (dwi.dwOpModes & (1<<CGOS_WDOG_OPMODE_EVENT_REPEAT))
                {
                        /* watchdog supports repeated event mode */
```

4.7.3 Events

An event is executed by the onboard hardware when the Watchdog timeout occurs. Following events are defined:

CGOS_WDOG_EVENT_INT

defines a NMI or IRQ event

Depending on the hardware implementation, this event causes an NMI (non maskable interrupt) or an IRQ (normal hardware interrupt). It's up to the user to install an appropriate IRQ handler which is able to handle this type of event.

CGOS_WDOG_EVENT_SCI

defines a SMI or a SCI event

Depending on the hardware implementation, this event causes an SMI (system management interrupt) or an SCI (ACPI interrupt). It's up to the user to install an appropriate software handler which is able to handle this type of event.

CGOS_WDOG_EVENT_RST

defines a system reset event

This event causes a system reset. Depending on the hardware implementation, this reset will be applied to the complete system or only to parts of the system.

CGOS_WDOG_EVENT_BTN

defines a power button event

This event activates the power button signal. It can be used to switch off and even to switch on the board again in the case of a multistage Watchdog implementation.

4.7.4 Stages

Depending on the implementation the Watchdog might offer multiple stages for executing events. Each stage has its own timeout value and event definition. If a stage times out, the configured event for this stage will be executed and the next stage will be entered. This offers the ability to implement a more refined error handling.

It is possible to define IRQ as first stage event and power button as second stage event: If the timeout for the first stage occurs, an IRQ is generated and stage 2 becomes active. At the same time the appropriate IRQ handler will be activated and might solve the problem (e.g. by restarting a crashed application and triggering the Watchdog). If the triggering of the Watchdog doesn't occur and as well the second stage times out then the system will be shut down.

4.7.5 Watchdog Types

Following watchdog types are currently defined:

`CGOS_WDOG_TYPE_UNKNOWN`	used when the type is not known
`CGOS_WDOG_TYPE_BC`	the watchdog is implemented via the congatec onboard controller
`CGOS_WDOG_TYPE_CHIPSET`	the watchdog functionality is available just through the board's chipset

4.7.6 Information Structure

The `CgosWDogGetInfo` function call is used to get information about the current configuration and state of the Watchdog. It takes a pointer to an instance of structure `CGOSWDINFO`, which is defined as follows:

CGOSWDINFO

`unsigned long dwSize`

size of the structure itself, must be initialized with `sizeof(CGOSWDINFO)`

`unsigned long dwFlags`

reserved. Always set to 0.

`unsigned long dwMinTimeout`

this value depends on the hardware implementation of the Watchdog and specifies the minimum value for the Watchdog trigger timeout.

`unsigned long dwMaxTimeout`

this value depends on the hardware implementation of the Watchdog and specifies the maximum value for the Watchdog trigger timeout.

`unsigned long dwMinDelay`

this value depends on the hardware implementation of the Watchdog and specifies the minimum value for the Watchdog enable delay.

`unsigned long dwMaxDelay`

this value depends on the hardware implementation of the Watchdog and specifies the maximum value for the Watchdog enable delay.

`unsigned long dwOpModes`

the mask of the supported operation modes, see section 4.7.2 Operation Modes

`unsigned long dwMaxStageCount`

the amount of supported Watchdog stages, see section 4.7.4 Stages

`unsigned long dwEvents`

the mask of the supported Watchdog events, see section 4.7.3 Events

`unsigned long dwType`

see section 4.7.5 Watchdog Types

4.7.7 Configuration

The `CgosWDogSetConfigStruct` and `CgosWDogGetConfigStruct` function calls are used to set and to determine the Watchdog configuration. Both of them take a pointer to an instance of structure `CGOSWDCONFIG` which is defined as follows:

CGOSWDCONFIG

`unsigned long dwSize`

size of the structure itself, must be initialized with sizeof(CGOSWDCONFIG)

`unsigned long dwTimeout`

it specifies the value for the Watchdog timeout. It must be in the range `CGOSWDINFO:dwMinTimeout` and `CGOSWDINFO:dwMaxTimeout`. In case of multiple stages, this value is not used because the configuration occurs through the appropriate stage structure.

`unsigned long dwDelay`

this value specifies the value for the Watchdog enable delay, see also figure 1

or figure 2 from section 4.7.10 Watchdog Timing Chart .

`unsigned long dwMode`

the current mode, see section 4.7.1 Mode

`unsigned long dwOpMode`

the mask of the supported operation modes, see section 4.7.2 Operation Modes

this value is only used in multistage mode

`unsigned long dwStageCount`

the number of available Watchdog stages, see section 4.7.4 Stages

this value is only used in multistage mode

`CGOSWDSTAGE stStages[CGOS_WDOG_EVENT_MAX_STAGES]`

this array holds the state definition of each defined stage

these values are only used in multistage mode

The function `CgosWDogSetConfig` and the configuration structure use time values with a millisecond resolution. `timeout` is the basic time during which a `CgosWDogTrigger` function must be called. `delay` adds an initial time period for the first trigger call.

In case of a multistage Watchdog implementation the array `stStages` of type `CGOSWDSTAGE` contains the stage structures which incorporate the timeout and event value for each stage. Refer also to figure 2 from section 4.7.10 Watchdog Timing Chart and the definition below:

CGOSWDSTAGE

`unsigned long dwTimeout`

it specifies the time value for the affected stage. The value must be in the range `CGOSWDINFO:dwMinTimeout` and `CGOSWDINFO:dwMaxTimeout`

`unsigned long dwEvent`

it contains the event definition for the affected stage, see section 4.7.3 Events

If the mode is set to staged, then up to three stages can be defined. The stages are run in the order they are specified after each timeout value has expired without triggering the Watchdog.

Note

The CgosWDogSetConfig *function call is provided for convenience. It offers a fast and easy way for setting up a single staged Watchdog without the necessity to handle a complex configuration structure. However, it's recommended to use* CgosWDogSetConfigStruct *to benefit from the features of a multistage Watchdog implementation.*

4.7.8 Triggering

After configuring the Watchdog by `CgosWDogSetConfigStruct` the application must continuously call `CgosWDogTrigger` that resets the Watchdog timer.

4.7.9 Disabling the Watchdog

An enabled Watchdog can be disabled by calling `CgosWDogDisable`.

4.7.10 Watchdog Timing Chart

Figure 1: single stage / single event mode

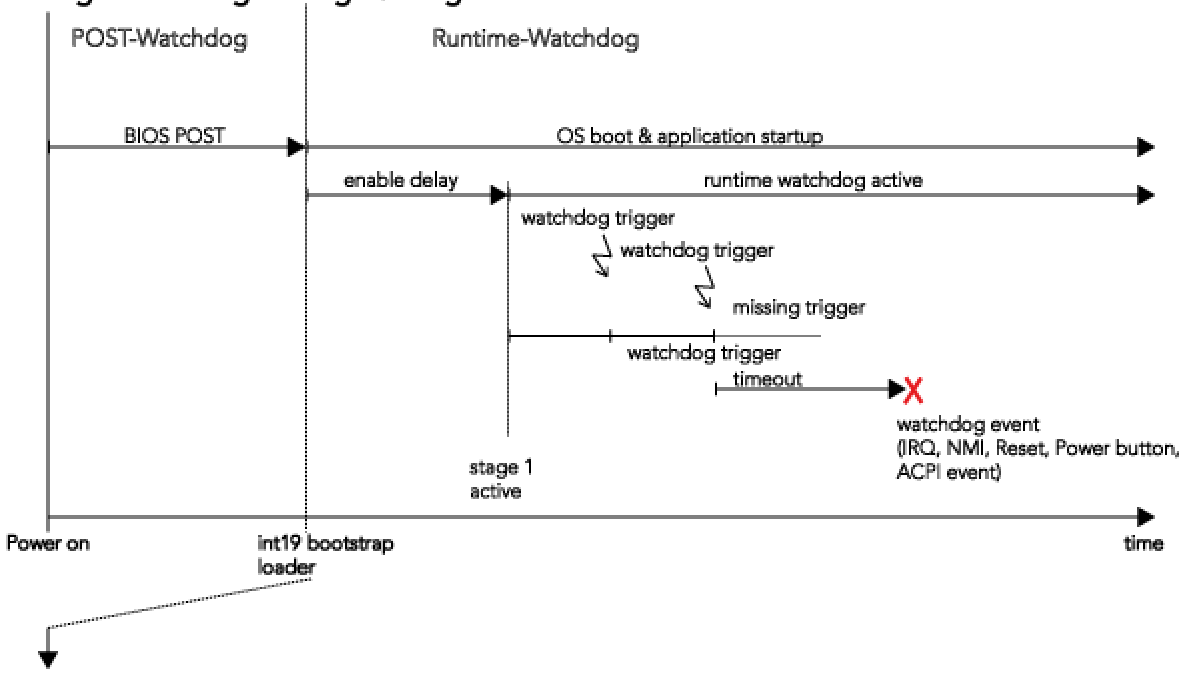
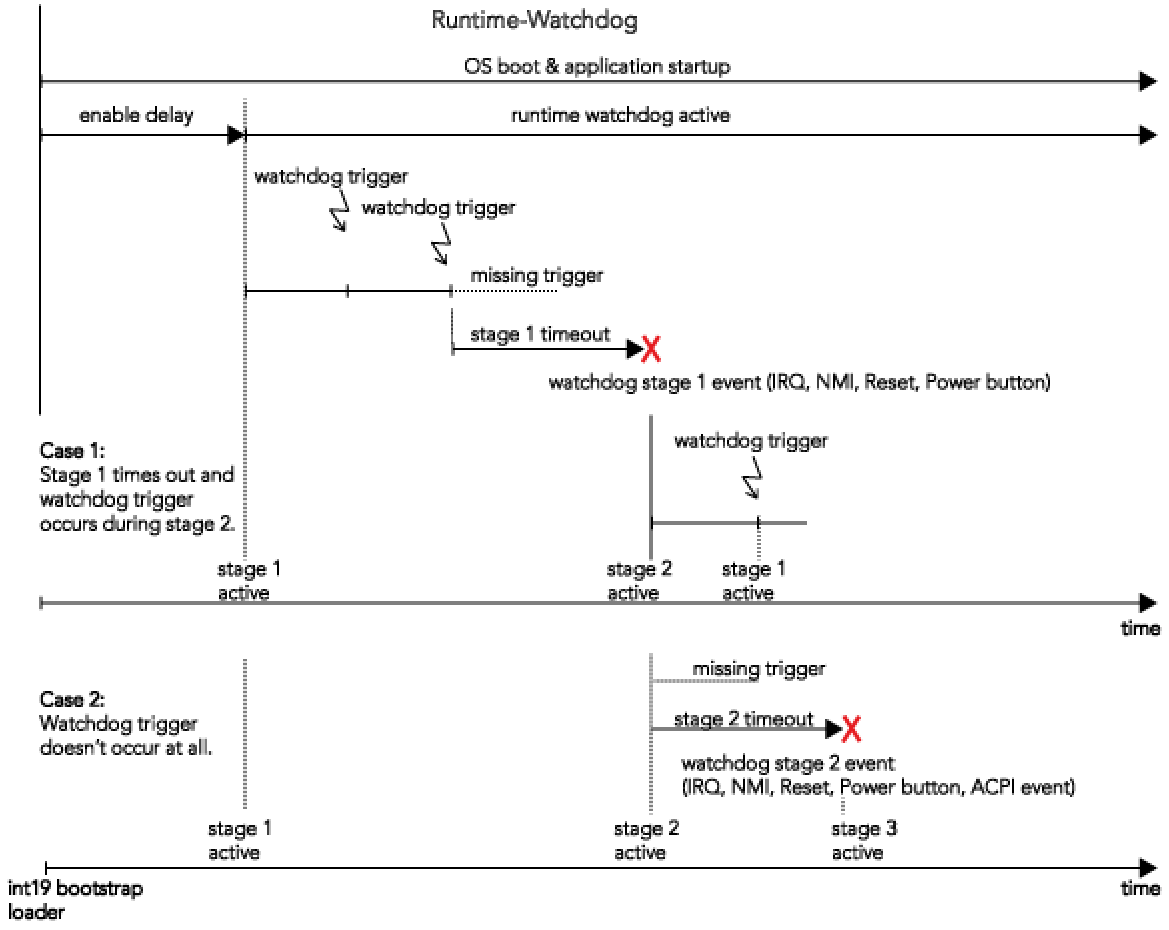


Figure 2: multi stage / single event mode



4.8 Hardware Monitoring

The CGOS interface provides access to hardware monitoring functions such as the voltage sensor, temperature sensor and fan control.

`CgosVoltageGetCount`, `CgosTemperatureGetCount` and `CgosFanGetCount` are used to determine the number of attached sensors per type.

`CgosVoltageGetInfo`, `CgosTemperatureGetInfo` and `CgosFanGetInfo` are used to determine the state and the configuration of an attached sensor.

`CgosVoltageGetCurrent`, `CgosTemperatureGetCurrent` and `CgosFanGetCurrent` are used to determine the actual measured value of an attached sensor.

4.8.1 Sensor Status Flags

The sensor status flags (`unsigned long dwFlags`), which are defined in the `CGOS*INFO` structure, represent the capabilities of the related sensor. The status flags can be determined using a `Cgos*GetInfo` function call. The following sensor status flags are defined:

`CGOS_SENSOR_ACTIVE` the sensor is active and usable

`CGOS_SENSOR_ALARM` the sensor supports alarm indication

`CGOS_SENSOR_BROKEN` there's no physical sensor attached

`CGOS_SENSOR_SHORTCIRCUIT` the sensor has a short circuit

4.8.2 Temperature Sensor Types

The following types of temperature sensors are defined and are dependent on their location within the system:

`CGOS_TEMP_CPU` the sensor that measures the CPU temperature

`CGOS_TEMP_BOX` the sensor that measures the temperature within the system chassis

`CGOS_TEMP_ENV` the sensor that measures the temperature of the system environment

`CGOS_TEMP_BOARD` the sensor that measures the board temperature

`CGOS_TEMP_BACKPLANE` the sensor that measures the temperature on the backplane

`CGOS_TEMP_CHIPSETS` the sensor that measures the temperature of the chipset

`CGOS_TEMP_VIDEO` the sensor that measures the temperature of the video chip

`CGOS_TEMP_TOPDIMM_ENV` the sensor that measures the temperature of the DRAM module on the topside of the CPU board

`CGOS_TEMP_BOTDIMM_ENV` the sensor that measures the temperature of the DRAM module on the bottom side of the CPU board

`CGOS_TEMP_OTHER` all other temperature sensors found within the system

4.8.3 Temperature Information Structure

The `CgosTemperatureGetInfo` function call is used to get information about the current configuration and state of the temperature sensor. It takes a pointer to an instance of structure `CGOSTEMPERARUREINFO`, which is defined as follows:

CGOSTEMPERATUREINFO

`unsigned long dwSize`

size of the structure itself, must be initialized with `sizeof(CGOSTEMPERATUREINFO)`

`unsigned long dwType`

see section 4.8.2.Temperature Sensor Types

`unsigned long dwFlags`

see section 4.8.1. Sensor Status Flags

`unsigned long dwAlarm`

alarm type / action (currently not used)

`unsigned long dwRes`

this value defines the granularity of the temperature sensor

`unsigned long dwMin`

this is the minimum value that can be measured by the sensor

`unsigned long dwMax`

this is the maximum value that can be measured by the sensor

`unsigned long dwAlarmHi`

high temperature alarm value (currently not used)

`unsigned long dwHystHi`

hystheresis for releasing high temperature alarm (currently not used)

`unsigned long dwAlarmLo`

low temperature alarm value (currently not used)

`unsigned long dwHystLo`

hystheresis for releasing low temperature alarm (currently not used)

All temperature values are in units of 1/1000th degree centigrade.

If any field value is -1 then the respective temperature value is not supported or known.

4.8.4 Fan Sensor Types

The following types of fan sensors are defined and are dependent on their location within the system:

`CGOS_FAN_CPU`	the sensor that represents the CPU fan
`CGOS_FAN_BOX`	the sensor that represents the fan on the chassis
`CGOS_FAN_CHIPSET`	the sensor that represents the fan on the chipset
`CGOS_FAN_VIDEO`	the sensor that represents the fan on the video chip
`CGOS_FAN_OTHER`	all other fan sensors found within the system

4.8.5 Fan Information Structure

The `CgosFanGetInfo` function call is used to get information about the current configuration and state of the fan control. It takes a pointer to an instance of structure `CGOSFANINFO`, which is defined as follows:

CGOSFANINFO

`unsigned long dwSize`

size of the structure itself, must be initialized with `sizeof(CGOSFANINFO)`

`unsigned long dwType`

see section 4.8.4.Fan Sensor Types

`unsigned long dwFlags`

see section 4.8.1. Sensor Status Flags

`unsigned long dwAlarm`

alarm type / action (currently not used)

`unsigned long dwSpeedNom`

this value defines the nominal speed of the fan.

`unsigned long dwMin`

this is the minimum speed of the fan

`unsigned long dwMax`

this is the maximum speed of the fan

`unsigned long dwAlarmHi`

high fan speed alarm value (currently not used)

`unsigned long dwHystHi`

hystheresis for releasing high fan speed alarm (currently not used)

`unsigned long dwAlarmLo`

low fan speed alarm value (currently not used)

`unsigned long dwHystLo`

hystheresis for releasing low fan speed alarm (currently not used)

`unsigned long dwOutMin`

this sets the minimum speed for the fan (currently not used)

`unsigned long dwOutMax`

this sets the maximum speed for the fan

When using CGOSFANINFO in CgosTemperatureSetLimits, the value in this field can be set from 0 to 100 to set the current fan speed from 0% to 100% of its nominal RPM value.

All fan speed values are in RPM (revolutions per minute).

If any field value is -1 then the respective speed value is not supported or known.

4.8.6 Voltage Sensor Types

The following types of voltage sensors are defined and are dependent on their location within the system:

`CGOS_VOLTAGE_BAT_CMOS`	the sensor that measures the CMOS battery
`CGOS_VOLTAGE_BAT_POWER`	the sensor that measures the battery voltage in a mobile system
`CGOS_VOLTAGE_5V_S0`	the sensor that measures the 5V standard voltage

	Supply voltage on 5V powered boards
`CGOS_VOLTAGE_5V_S5`	the sensor that measures the 5V standby voltage
`CGOS_VOLTAGE_33V_S0`	the sensor that measures the 3.3V standard voltage
`CGOS_VOLTAGE_33V_S5`	the sensor that measures the 3.3V standby voltage
`CGOS_VOLTAGE_12V_S0`	the sensor that measures the 12V standard voltage
	Supply voltage on 12V powered boards
`CGOS_VOLTAGE_VCOREA`	The sensor that measures the first core voltage (often used as CPU voltage)
`CGOS_VOLTAGE_VCOREB`	The sensor that measures the second core voltage (often used as memory and chipset voltage)
`CGOS_VOLTAGE_DC`	any sensor that measures an onboard voltage that can't be covered by the previous definitions
	Supply voltage on wide range voltage powered boards
`CGOS_VOLTAGE_DC_STANDBY`	any sensor that measures a standby voltage that can't be covered by the previous definitions
`CGOS_VOLTAGE_OTHER`	specified if none of the above can be applied

Due to missing CGOS current sensor definitions the values of the existing input current sensors are returned in CGOSVOLTAGEINFO as voltage drop of the respective input current on a 1 Ohm resistor. This allows direct interpretation of this voltage as input current value in 1/1000th amperes.

The following types of input current sensor voltages are defined :

`CGOS_VCURRENT_DC`	voltage drop created by DC (wide range) board supply voltage driven input current on 1 Ohm resistor
`CGOS_VCURRENT_5V_S0`	voltage drop created by 5V board supply voltage driven input current on 1 Ohm resistor
`CGOS_VCURRENT_12V_S0`	voltage drop created by 12V board supply voltage driven input current on 1 Ohm resistor

The `CgosVoltageGetInfo` function call is used to get information about the current configuration and state of the voltage control. It takes a pointer to an instance of structure `CGOSVOLTAGEINFO`, which is defined as follows:

CGOSVOLTAGEINFO

`unsigned long dwSize`

size of the structure itself, must be initialized with `sizeof(CGOSVOLTAGEINFO)`

`unsigned long dwType`

see section 4.8.5.Voltage Sensor Types

`unsigned long dwNom`

this value defines the nominal voltage of the sensor.

If the value is -1 then the nominal voltage is not supported or known

`unsigned long dwFlags`

see section 4.8.1. Sensor Status Flags

`unsigned long dwAlarm`

alarm type / action (currently not used)

`unsigned long dwRes`

this value defines the granularity of the voltage sensor

`unsigned long dwMin`

this is the minimum value that can be determined by the sensor

`unsigned long dwMax`

this is the maximum value that can be determined by the sensor

`unsigned long dwAlarmHi`

high voltage alarm value (currently not used)

`unsigned long dwHystHi`

hystheresis for releasing high voltage alarm (currently not used)

`unsigned long dwAlarmLo`

low voltage alarm value (currently not used)

`unsigned long dwHystLo`

hystheresis for releasing low voltage alarm (currently not used)

All of the above mentioned voltage values are in units of 1/1000th volt.

If any field value is -1 then the respective voltage value is not supported or known.

Code example to enumerate through all the voltage sensors:

```
static CGOSVOLTAGEINFO voltageInfo = {0};
unsigned long i, setting, status, monCount = 0;

voltageInfo.dwSize = sizeof (voltageInfo);
monCount = CgosVoltageCount(hCgos);
printf("\nNumber of voltage monitors: %d\n", monCount);
if(monCount != 0)
{
 for(i = 0; i < monCount; i++)
 {
     if(CgosVoltageGetInfo(hCgos, i, &voltageInfo))
     {
         printf("Voltage monitor %d information:\n", i);
         printf("Type:           %d\n", voltageInfo.dwType);
         printf("Resolution:     %d\n", voltageInfo.dwRes);
         printf("Nominal value:  %d\n", voltageInfo.dwNom);
         printf("Max. Value:     %d\n", voltageInfo.dwMax);
         printf("Min. Value:     %d\n", voltageInfo.dwMin);
     }

     if(CgosVoltageGetCurrent(hCgos, i, &setting, &status))
     {
         printf("\n");
         printf("Current setting:    %d\n", setting);
         printf("Current status:     %d\n", status);
     }
     printf("\nPress key to continue...\n");
     getch();
 }
}
```

4.9 GPIO Functions

Various industrial standards, such as COM Express™, specify pins for general purpose I/Os. The CGOS interface provides functions to control these hardware GPIO pins.

The function call `CgosIOCount` is used to determine the amount of available GPIO units. Each GPIO unit is able to handle up to 32 GPIs/GPOs/GPIOs.

Similar to each other group of functions, a call of `CgosIOIsAvailable` is used to determine the availability of the desired GPIO unit.

With the function calls `CgosIORead` and `CgosIOWrite`, it is possible to read from or write to the GPIO pins.

`CgosIOGetDirectionCaps` returns the direction capabilities of the pins handled by the selected GPIO unit. A bit set in the input pin field indicates that this bit can handle a GPI. A bit set in the output pin field indicates that this bit can handle a GPO. A bit set in input and output pin field indicates that the corresponding pin's direction can be changed, i.e. this bit handles a GPIO. A bit set only in the input pin field handles a hardwired GPI. A bit set only in the output pin field handles a hardwired GPO. Bit positions set neither in the input nor the output pin fields have no corresponding pin at all.

The function call `CgosIOGetDirection` returns the current direction of the GPIO pins. A bit set to 1 in this field indicates that the respective pin is configured as an input while 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 either 0 or 1. Therefore, it's recommended to cross check the result of `CgosIOGetDirection` with the result of `CgosIOGetDirectionCaps`.

Example:

```
 unsigned long ulCurrentPinDirection;
 unsigned long ulInputPins, ulOutputPins;
 unsigned long ulInputValue, ulOutputValue;


  if(CgosIOGetDirectionCaps(hCgos, ulUnit, &ulInputPins, &ulOutputPins))
       {
               /* if the result is: ulInputPins = 0x0000000F, ulOutputPins = 0x000000F0
*/
               /* then */
               /* pins 0 ... 3 are GPIs (general purpose inputs) */
               /* pins 4 ... 7 are GPOs (general purpose outputs) */
         if(CgosIOGetDirection(hCgos, ulUnit, &ulCurrentPinDirection))
              {
                      /* all availabe & configured input pins */
                     ulInputPins &= ulCurrentPinDirection;

                      /* all availabe & configured output pins */
                     ulOutputPins &= ~ulCurrentPinDirection;

                     /* get the value of the input pins */
                     CgosIORead(hCgos, ulUnit, &ulInputValue);

                     /* set the value of the output pins (e.g. all to 1) */
                     ulOutputValue = ulOutputPins;
                     CgosIOWrite(hCgos, ulUnit, ulOutputValue);
              }
             }
```

Furthermore, `CgosIOSetDirection` is used to change the direction of a GPIO pin. Notice that changing the pin direction configuration is not supported for the COM Express™ GPIO unit as GPI/GPO configuration is fixed by spec./design. Therefore, the respective function will fail for COM Express™ and is only added here for completeness.

5 CGOS Library API Programmer's Reference

5.1 General

The CGOS (congatec operating system) API provides access to congatec specific board information and features.

The API is compatible and identical across all congatec boards and all supported operating systems. It is divided into function groups for:

CgosLib*	Management functions for the library API itself
CgosBoard*	Board information
CgosVga*	VGA or LCD information and control
CgosStorageArea*	Storage Area (EEPROM, Flash, ...) access
CgosI2C*	I2C bus access
CgosIO*	GPIO access
CgosWDog*	Watchdog control
CgosPerformance*	Performance information and control
CgosTemperature*	Temperature information and control
CgosFan*	Fan information
CgosVoltage*	Voltage information

Note

The function group for Performance is not available in the currently released CGOS API. When calling these functions, the result will be 0 (failure).

All of them provide a `Cgos*Count()` ***function to retrieve the number of available units. All other functions within that group require a*** `dwUnit` ***parameter. In all cases this can simply be the zero based unit number.***

Some functions and structures contain version numbers. All 16 bit version numbers contain the major number in the high byte and the minor in the low byte in BCD. BIOS and board controller version numbers should simply be treated as 3 BCD digits as only that combination together with the board name yields useful information.

All 32 bit version numbers contain the 16 bit version number in the high word and a build or subversion number in the low word.

For function call details and parameters also refer to the `cgos.h` ***header file.***

5.1.1 Return Values

Unless they return a count or version number, all Cgos* functions return 1 for success and 0 for failure. Other return values are stored in pointers passed to the function.

5.1.2 Board Classes

In a system with several CGOS compliant boards, the board class is used to distinguish between the hardware types of the installed boards. Currently, board classes are defined for CPU, VGA and IO boards, respectively:

```
CGOS_BOARD_CLASS_CPU
CGOS_BOARD_CLASS_VGA
CGOS_BOARD_CLASS_IO
```

5.1.3 Information Structures

The API defines several information structures in `cgos.h` They are used to store the returned values during `Cgos*GetInfo` calls. Before using these structures, the `dwSize` entry of each info structure must be initialized with the size of the structure itself `(sizeof(CGOS*INFO)).` This provides independence between the application and the library if the structure is extended in future releases of the library.

5.1.4 Unit numbers

Almost all function calls take a unique unit number that is used to identify a dedicated unit. Usually the unit number is between 0 and the return value -1 of the related `Cgos*Count` function call. It can be taken as an index for devices of the same type. The following example shows how to determine the current value of the CPU temperature sensor:

Example 1.

```
static CGOSTEMPERATUREINFO temperatureInfo = {0};

unsigned long dwUnit, monCount = 0, dwTemp, dwState;

temperatureInfo.dwSize = sizeof (temperatureInfo);

// determine number of temperature sensors
monCount = CgosTemperatureCount(hCgos);
printf("Number of temperature monitors: %d\n", monCount);
if(monCount != 0)
{
for(dwUnit = 0; dwUnit < monCount; dwUnit++)
{
       if(CgosTemperatureGetInfo(hCgos, dwUnit, &temperatureInfo))
       {
               if (temperatureInfo.dwType == CGOS_TEMP_CPU)
               {
               // temperatureInfo now contains the info structure of the cpu sensor
               // dwUnit points to the cpu temperature sensor
               if (CgosTemperatureGetCurrent(hCgos, dwUnit, &dwTemp, &dwState)
               {
                         //  dwTemp  and  dwState  contain  the  actual  values  of  the  cpu
               sensor
               }
               }
            }
         }
}
```

A device enumeration can always be set up as shown above.

Additionally, some function calls such as all of the `CgosStorageArea*` and `CgosI2C*` function calls can take a type number as `dwUnit` parameter.

The following examples used to determine the storage area size of the user EEPROM (type CGOS_STORAGE_AREA_EEPROM) are equivalent:

Example 2.

```
   unsigned long dwUnit;
   unsigned long dwSize;
   unsigned long areaCount = CgosStorageAreaCount(hCgos,CGOS_STORAGE_AREA_UNKNOWN);

   for(dwUnit = 0; dwUnit < areaCount; dwUnit++)
   {
        if (CgosStorageAreaType(hCgos,dwUnit) == CGOS_STORAGE_AREA_EEPROM))
        {
          dwSize = CgosStorageAreaSize(hCgos,dwUnit);
        }
   }
```

Example 3.

```
   unsigned long dwSize;
   dwSize = CgosStorageAreaSize(hCgos,CGOS_STORAGE_AREA_EEPROM);
```

Note

The device enumeration as shown in Example 1 is the preferred way to obtain access to the unit information and works for all function groups. Example 3 shows a convenient way to access the unit through its type definition but keep in mind that this method is not available for all function groups.

5.2 Function Group CgosLib*

The `CgosLib*` functions are used to initialize and to remove the CGOS Library. The library provides the basic layer for the application to access all the CGOS API functions. The library must be installed before any call to CGOS API functions can be executed successfully.

5.2.1 CgosLibGetVersion

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosLibGetVersion(void)
```

Note

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.

5.2.2 CgosLibInitialize

CGOS API version

1.00.000 and later

Declaration

```
bool CgosLibInitialize(void)
```

Remark

Initializes the CGOS API library.

5.2.3 CgosLibUninitialize

CGOS API version

1.00.000 and later

Declaration

```
bool CgosLibUninitialize(void)
```

Note

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

5.2.4 CgosLibIsAvailable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosLibIsAvailable(void)
```

Note

Checks if the CGOS API library has already been initialized by a prior call to function `CgosLibInitialize`***.***

5.2.5 CgosLibInstall

CGOS API version

1.00.000 and later

Declaration

```
bool CgosLibInstall(unsigned int install)
```

Input

`install`	1 – installs the low level CGOS driver
	0 – removes the low level CGOS driver

Note

This function can be used to install the low level CGOS driver if a prior call of `CgosLibInitialize` ***failed.***

Keep in mind that you might need administrative privileges for executing this function successfully.

See also section 4.1 Installing the DLL for a more detailed description about installing the CGOS API library.

5.2.6 CgosLibGetDrvVersion

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosLibGetDrvVersion(void)
```

Note

Returns the version of the low level CGOS driver.

5.2.7 CgosLibGetLastError

CGOS API version

1.02.000 and later

Declaration

```
ulong CgosLibGetLastError(void)
```

Note

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)

5.2.8 CgosLibSetLastErrorAddress

CGOS API version

1.02.000 and later

Declaration

```
bool CgosLibSetLastErrorAddress(unsigned long *pErrNo)
```

Input

`pErrNo`	buffer where the error code will be stored

Note

With this function it's possible to specify a local memory location in the context of the application where the last error code will be stored. It provides a convenient way of implementing error handling without calling the `CgosLibGetLastError` ***function after each regular CGOS API function call.***

See section 5.2.7 CgosLibGetLastError for a detailed list of valid error codes.

5.3 Function Group CgosBoard*

The `CgosBoard*` routines are used to obtain a handle to a dedicated board and specific board information like the number of boots or the total running time.

5.3.1 CgosBoardCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosBoardCount(unsigned long dwClass,unsigned long dwFlags)
```

Input

`dwClass` the hardware class of the board, see also 4.2 subsection "Board classes"

`dwFlags` either `CGOS_BOARD_OPEN_FLAGS_DEFAULT` or `CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY`

`CGOS_BOARD_OPEN_FLAGS_DEFAULT`
counts all boards of the given hardware class

`CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY`
counts only boards which primary board class

matches the given hardware class

Note

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.***

5.3.2 CgosBoardOpen

CGOS API version
1.00.000 and later

Declaration

```
bool  CgosBoardOpen(unsigned  long  dwClass,  unsigned  long  dwNum,
unsigned long dwFlags, HCGOS *phCgos)
```

Input

`dwClass` the hardware class of the board, see also 4.2 subsection "Board classes"

`dwNum` the subsequent number of the selected board in its class, starting from 0

`dwFlags` either `CGOS_BOARD_OPEN_FLAGS_DEFAULT` or `CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY`

`CGOS_BOARD_OPEN_FLAGS_DEFAULT`
scans for all boards of the specified hardware class, regardless if it's the primary class or the secondary class

`CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY`
scans for boards which primary board class matches the specified hardware class

`phCgos` buffer where the board handle will be stored

Note

Each CGOS compliant board in the system will be addressed with its own unique board handle. This function is used to open such a board and to obtain a valid board handle. If there is more than one CGOS board in the system, each board can be individually selected by its board class `dwClass` *and a subsequent enumeration of* `dwNum`*. On success, the function returns the board handle in* `*phCgos`*.*

`CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY` *might be used for* `dwFlags` *to select a board of a dedicated board class. Together with an enumerated counter starting from 0 the board can be addressed exactly. For instance, the call to open the 2nd (cgos compliant) vga board would be:*

```
HCGOS hcgos;

CgosBoardOpen(CGOS_BOARD_CLASS_VGA,1,CGOS_BOARD_OPEN_FLAGS_PRIMARYONLY
,&hcgos);
```

5.3.3 CgosBoardOpenByName

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardOpenByName(const char *pszName, HCGOS *phCgos)
```

Input

`pszName`	the name of the board, e.g. “X855” in case of conga-X855 board
`phCGOS`	buffer where the board handle will be stored

Note

This function behaves like `CgosBoardOpen` *except that the board is specified by its name. On success, the function returns the board handle in* `*phCgos`*.*

5.3.4 CgosBoardClose

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardClose(HCGOS hCgos)
```

Input

`hCgos` the board handle

Note

Closes a board which was previously opened by either `CgosBoardOpen` *or* `CgosBoardOpenByName`*.*

5.3.5 CgosBoardGetName

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardGetName(HCGOS hCgos, const char *pszName, unsigned long
dwSize)
```

Input

`hCgos`	the board handle
`pszName`	buffer where the board name will be stored
`dwSize`	size of the buffer in bytes, should be at least `CGOS_BOARD_MAX_SIZE_ID_STRING`

Note

Determines the name of the board addressed by `hCgos`***.***

5.3.6 CgosBoardGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardGetInfo(HCGOS hCgos, CGOSBOARDINFO *pBoardInfo)
```

Input

`hCgos`	the board handle
`pBoardInfo`	the buffer where the board information will be stored

Note

Gets the board information of a CGOS API compliant board addressed by hCgos. See section 4.3 Generic Board Functions for a detailed description of the `CGOSBOARDINFO` ***structure.***

5.3.7 CgosBoardGetBootCounter

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardGetBootcounter(HCGOS hCgos, unsigned long *pdwCount)
```

Input

`hCgos`	the board handle
`pdwCount`	the variable where the boot counter value will be stored

Note

Gets the current value of the boot counter.

5.3.8 CgosBoardGetRunningTimeMeter

CGOS API version

1.00.000 and later

Declaration

```
bool CgosBoardGetRunningTimeMeter(HCGOS hCgos, unsigned long *pdwCount)
```

Input

`hCgos`	the board handle
`pdwCount`	the variable where the value of the running time meter will be stored

Note

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

5.4 Function Group CgosVga*

The `CgosVga*` functions are used to control all functionality, which belongs to VGA or LCD (like enabling backlight, etc.).

5.4.1 CgosVgaCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosVgaCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Gets the number of installed VGA boards in the system.

5.4.2 CgosVgaGetBacklight

CGOS API version

1.00.000 and later

Declaration

```
bool CgosVgaGetBacklight(HCGOS hCgos, unsigned long dwUnit, unsigned long *pdwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	the variable where the backlight brigthness will be stored

Note

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

5.4.3 CgosVgaSetBacklight

CGOS API version

1.00.000 and later

Declaration

```
bool CgosVgaSetBacklight(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwSetting`	the backlight value

Note

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

5.4.4 CgosVgaGetBacklightEnable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosVgaGetBacklightEnable(HCGOS hCgos, unsigned long dwUnit,
unsigned long *pdwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	the variable where the backlight enable value will be stored

Return

`*pdwSetting = 0`	backlight is off
`*pdwSetting = 1`	backlight is on

Note

Returns the state of the LCD's backlight.

5.4.5 CgosVgaSetBacklightEnable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosVgaSetBacklightEnable(HCGOS hCgos, unsigned long dwUnit,
unsigned long dwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwSetting`	the backlight enable value

Note

Turns the backlight on or off.

5.4.6 CgosVgaGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosVgaGetInfo(HCGOS  hCgos,  unsigned  long  dwUnit,  CGOSVGAINFO
*pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	the buffer where the VGA information will be stored

Note

Gets the VGA board information of a CGOS API compliant board addressed by `hCgos`***.***

See section 4.4 VGA Functions for a detailed description of the CGOSVGAINFO structure.

5.5 Function Group CgosStorageArea*

The `CgosStorageArea*` functions are used to control and access all different types of storage areas on the board. A storage area can be the complete flash ROM, a part of the flash ROM, the onboard EEPROM or the CMOS RAM. See also section 4.6.1 Storage Area Types.

Caution

Improper use of these functions may lead to permanent damage to your system thus preventing it from booting. For instance, the complete BIOS can be destroyed by accidentally writing to `CGOS_STORAGE_AREA_FLASH`***.***

5.5.1 CgosStorageAreaCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosStorageAreaCount(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	the dedicated storage area type (see section 4.6.1.Storage Area Types) or `CGOS_STORAGE_AREA_UNKNOWN` for all storage areas

Note

Gets the number of installed storage areas of the board.

5.5.2 CgosStorageAreaType

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosStorageAreaType(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Return

Returns an or-ed value depending on the installed areas:

```
      CGOS_STORAGE_AREA_EEPROM
      CGOS_STORAGE_AREA_FLASH
      CGOS_STORAGE_AREA_CMOS
      CGOS_STORAGE_AREA_RAM
```

or `CGOS_STORAGE_AREA_UNKNOWN` if the type is not known.

Note

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).

5.5.3 CgosStorageAreaSize

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosStorageAreaSize(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Note

Returns the size of the storage area in bytes.

5.5.4 CgosStorageAreaBlockSize

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosStorageAreaBlockSize(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Note

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

5.5.5 CgosStorageAreaRead

CGOS API version

1.00.000 and later

Declaration

```
bool CgosStorageAreaRead(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwOffset, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwOffset`	byte offset where the data is read from
`pBytes`	`pointer` to the destination buffer
`dwLen`	number of bytes to read

Note

Reads* **dwLen** *bytes from the storage area into buffer* `pBytes`*.

5.5.6 CgosStorageAreaWrite

CGOS API version

1.00.000 and later

Declaration

```
bool CgosStorageAreaWrite(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwOffset, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwOffset`	byte offset where the data writes to
`pBytes`	pointer to the source buffer
`dwLen`	number of bytes to write

Note

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

5.5.7 CgosStorageAreaErase

CGOS API version

1.00.000 and later

Declaration

```
bool CgosStorageAreaErase(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwOffset, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwOffset`	byte offset to the area, which will be erased
`dwLen`	number of bytes to erase

Note

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

5.5.8 CgosStorageAreaEraseStatus

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosStorageAreaEraseStatus(HCGOS  hCgos,  unsigned  long  dwUnit,
unsigned long dwOffset, unsigned long dwLen, unsigned long *lpStatus)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwOffset`	byte offset to the which will be erased
`dwLen`	number of bytes to erase
`lpStatus`	pointer to the status

Note

Returns the status of the current area erase progress in `lpStatus`***:***

0 Erasing the specified area finished successfully

1 Erasing in progress

2 Erase error

5.5.9 CgosStorageAreaLock

CGOS API version

1.02.000 and later

Declaration

```
bool CgosStorageAreaLock(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwFlags, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwFlags`	reserved for future use, set to 0
`pBytes`	pointer to the source buffer containing the secret string
`dwLen`	number of bytes to write

Note

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 `CgosStorageAreaUnlock` ***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. The length of the string must be specified in dwLen.***

5.5.10 CgosStorageAreaUnlock

CGOS API version

1.02.000 and later

Declaration

```
bool CgosStorageAreaUnlock(HCGOS hCgos, unsigned long dwUnit, unsigned
long dwFlags, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwFlags`	reserved for future use, set to 0
`pBytes`	pointer to the source buffer containing the secret string
`dwLen`	number of bytes to write

Note

This function is used to unlock a write protected storage area that was previously locked using `CgosStorageAreaLock`***. 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 section 5.5.9 CgosStorageAreaLock for additional details.***

This function fails if the selected area is already unlocked.

5.5.11 CgosStorageAreaIsLocked

CGOS API version

1.02.000 and later

Declaration

```
bool   CgosStorageAreaIsLocked(HCGOS   hCgos,   unsigned   long   dwUnit,
unsigned long dwFlags)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwFlags`	reserved for future use, set to 0

Note

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 section 5.5.9 CgosStorageAreaLock for additional details.

5.6 Function Group CgosI2C*

The `CgosI2C*` functions are used to control and access the board I2C and SM buses.

Caution

Improper use of these functions in combination with certain devices and buses could possibly lead to permanent damage to your system thus preventing it from booting. For example if the configuration data of EEPROM located on the RAM module, which is attached to SMBus, was accidentally overwritten the RAM module would become inaccessible therefore preventing the system from completing the boot process.

5.6.1 CgosI2CCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosI2CCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Gets the number of installed I2C and SM buses in the system.

5.6.2 CgosI2CType

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosI2CType(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Return

Returns one of following values:

`CGOS_I2C_TYPE_PRIMARY`	the primary I2C bus
`CGOS_I2C_TYPE_SMB`	the system management bus
`CGOS_I2C_TYPE_DDC`	the I2C bus of the DDC interface
or	
`CGOS_I2C_TYPE_UNKNOWN`	for unknown or special purposes if the type is not known.

Note

Gets the type of the addressed I2C bus.

5.6.3 CgosI2CIsAvailable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosI2CIsAvailable(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Note

Determines if I2C bus of type dwUnit is present.

5.6.4 CgosI2CRead

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosI2CRead(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned  char
bAddr, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`bAddr`	the 8-bit address of the affected device on the bus (bit 0 must be logical 1 to indicate a read operation)
`pBytes`	the pointer to the destination buffer
`dwLen`	the number of sequential bytes to read

Note

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

5.6.5 CgosI2CWrite

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosI2CWrite(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned  char
bAddr, unsigned char *pBytes, unsigned long dwLen)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`bAddr`	the 8-bit address of the affected device on the bus (bit 0 must be logical 0 to indicate a write operation)
`pBytes`	the pointer to the source buffer
`dwLen`	the number of sequential bytes to write

Note

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

5.6.6 CgosI2CReadRegister

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosI2CReadRegister(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned
char bAddr, unsigned short wReg, unsigned char *pDataByte)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`bAddr`	the 8-bit address of the affected device on the bus (bit 0 must be logical 1 to indicate a read operation)
`wReg`	the number of the register to read
`pDataByte`	the pointer to the destination buffer

Note

Reads one byte from the register `wReg` ***in the device with address*** `bAddr` ***at I2C bus*** `dwUnit` ***to buffer*** `pDataByte`.

5.6.7 CgosI2CWriteRegister

CGOS API version

1.00.000 and later

Declaration

```
bool CgosI2CWriteRegister(HCGOS hCgos, unsigned long dwUnit, unsigned
char bAddr, unsigned short wReg, unsigned char bData)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`bAddr`	the 8-bit address of the affected device on the bus (bit 0 must be logical 0 to indicate a write operation)
`wReg`	the number of the register to write to
`bData`	the byte value to write

Note

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

5.6.8 CgosI2CWriteReadCombined

CGOS API version

1.00.000 and later

Declaration

```
bool CgosI2CWriteReadCombined(HCGOS hCgos, unsigned long dwUnit,
unsigned char bAddr, unsigned char *pBytesWrite, unsigned long
dwLenWrite, unsigned char *pBytesRead, unsigned long dwLenRead)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`bAddr`	the 8-bit address of the affected device on the bus (bit 0 must be logical 0)
`pBytesWrite`	the pointer to the source buffer which contains the bytes to write
`dwLenWrite`	the amount of bytes to write
`pBytesRead`	the pointer to the destination buffer
`dwLenRead`	the amount of bytes to read

Note

This function combines writing to and reading from a device on the I2C bus in one step. There will be no stop condition after writing to the device, the subsequent read cycle will be initiated with a leading start condition.

5.6.9 CgosI2CMaxFrequency

CGOS API version

1.03.000 and later

Declaration

```
bool CgosI2CGetMaxFrequency(HCGOS hCgos, unsigned long dwUnit,
unsigned long *pdwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	the variable where the maximum frequency setting will be stored

Note

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

5.6.10 CgosI2CGetFrequency

CGOS API version

1.03.000 and later

Declaration

```
bool CgosI2CGetFrequency(HCGOS hCgos, unsigned long dwUnit, unsigned
long *pdwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	the variable where the current frequency setting will be stored

Note

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

5.6.11 CgosI2CSetFrequency

CGOS API version

1.03.000 and later

Declaration

```
bool CgosI2CSetFrequency(HCGOS hCgos, unsigned long dwUnit, unsigned
long pdwSetting)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	the frequency setting in Hz

Note

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

5.7 Function Group CgosIO*

The `CgosIO*` function group provides access to general purpose I/O pins (if there are any).

5.7.1 CgosIOCount

CGOS API version

1.02.015 and later

Declaration

```
ulong CgosIOCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

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).

5.7.2 CgosIOIsAvailable

CGOS API version

1.02.015 and later

Declaration

```
bool CgosIOIsAvailable(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Note

Determines if IO unit dwUnit is present.

5.7.3 CgosIORead

CGOS API version

1.02.015 and later

Declaration

```
bool CgosIORead(HCGOS hCgos, unsigned long dwUnit, unsigned long
*pdwData)
```

Input

`hCgos`	the board handle

`dwUnit`	see section 5.1.4 Unit numbers
`pdwData`	the pointer to the destination buffer

Note

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.***

5.7.4 CgosIOWrite

CGOS API version

1.02.015 and later

Declaration

```
bool CgosIOWrite(HCGOS hCgos, unsigned long dwUnit, unsigned long dwData)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwData`	the data to write

Note

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.***

5.7.5 CgosIOGetDirectionCaps

CGOS API version

1.02.015 and later

Declaration

```
bool CgosIOGetDirectionCaps(HCGOS hCgos, unsigned long dwUnit, unsigned long *pdwInputs, unsigned long *pdwOutputs)
```

Input

hCgos	the board handle
dwUnit	see section 5.1.4 Unit numbers
pdwInputs	the pointer to the destination buffer of the input capabilities
pdwOutputs	the pointer to the destination buffer of the output capabilities

Note

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.***

5.7.6 CgosIOGetDirection

CGOS API version

1.02.015 and later

Declaration

```
bool  CgosIOGetDirection(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned
long *pdwData)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwData`	the pointer to the destination buffer of the direction information

Note

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`***.***

5.7.7 CgosIOSetDirection

CGOS API version

1.02.015 and later

Declaration

```
bool  CgosIOSetDirection(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned
long dwData)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`dwData`	the direction information

Note

Sets the current data direction of the respective GPI/GPO/GPIO pin. A bit set to 1 in this field indicates that the related pin is configured to be an input, a bit set to 0 indicates that the related pin is configured to be an output. Notice that the binary values for pins that are not implemented are unspecified and should be written as 0.

5.8 Function Group CgosWDog*

5.8.1 CgosWDogCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosWDogCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Returns the number of installed Watchdogs in the system.

5.8.2 CgosWDogIsAvailable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosWDogIsAvailable(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit Numbers

Note

Determines if the Watchdog is present.

5.8.3 CgosWDogTrigger

CGOS API version

1.00.000 and later

Declaration

```
bool CgosWDogTrigger(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers

Note

Triggers the Watchdog, i.e. restarts the Watchdog timer.

5.8.4 CgosWDogGetConfigStruct

CGOS API version

1.00.000 and later

Declaration

```
bool   CgosWDogGetConfigStruct(HCGOS   hCgos,   unsigned   long   dwUnit,
CGOSWDCONFIG *pConfig)
```

Input

`hCgos` the board handle

`dwUnit` see section 5.1.4 Unit numbers

`pConfig` the pointer to the configuration structure

Note

Determines the configuration of the Watchdog.

5.8.5 CgosWDogSetConfigStruct

CGOS API version

1.00.000 and later

Declaration

```
bool   CgosWDogSetConfigStruct(HCGOS   hCgos,   unsigned   long   dwUnit,
CGOSWDCONFIG *pConfig)
```

Input

`hCgos` the board handle

`dwUnit` see section 5.1.4 Unit numbers

`pConfig` the pointer to the configuration structure

Note

Sets the configuration of the Watchdog.

5.8.6 CgosWDogSetConfig

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosWDogSetConfig(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned
long timeout, unsigned long delay, unsigned long mode)
```

Input

`hCgos` the board handle

`dwUnit` see section 5.1.4 Unit numbers

`timeout` the value in milliseconds before the Watchdog times out. An application which is observed by the Watchdog must call `CgosWDogTrigger` within the specified time.

`delay` the delay before the Watchdog starts working. This is required to prevent a reboot while the operating system or the application initializes.

Note

Sets the configuration of the Watchdog. While `CgosWDogSetConfigStruct` ***takes a complete structure,*** `CgosWDogSetConfig` ***takes single values. Use*** `CgosWDogSetConfigStruct` ***to benefit from the advantages of a staged Watchdog.***

5.8.7 CgosWDogDisable

CGOS API version

1.00.000 and later

Declaration

```
bool CgosWDogDisable(HCGOS hCgos, unsigned long dwUnit)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit Numbers

Note

Disables the Watchdog.

5.8.8 CgosWDogGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosWDogGetInfo(HCGOS  hCgos,  unsigned  long  dwUnit,  CGOSWDINFO
*pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the Watchdog information structure

Note

Gets the information structure of the Watchdog.

5.9 Function Group CgosPerformance*

The `CgosPerformance*` function group is not implemented in the current release of the CGOS API. Calling one of these functions returns 0.

Note

Although there are already function declarations in `cgos.h` ***for*** `CgosPerformance*` ***the development is still in progress and the function declarations for this group may change in future.***

5.10 Function Group CgosTemperature*

The `CgosTemperature*` function group is used to access and control all the temperature sensors in the system.

5.10.1 CgosTemperatureCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosTemperatureCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Returns the number of installed temperature sensors in the system.

5.10.2 CgosTemperatureGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosTemperatureGetInfo(HCGOS  hCgos,  unsigned  long  dwUnit,
CGOSTEMPERATUREINFO *pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure
	see also section 4.8.3 Temperature Information Structure

Note

Gets the information structure of the specified temperature sensor.

5.10.3 CgosTemperatureGetCurrent

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosTemperatureGetCurrent(HCGOS  hCgos,  unsigned  long  dwUnit,
unsigned long *pdwSetting, unsigned long *pdwStatus)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	pointer to the sensor's current measured value

`pdwStatus` pointer to the sensor's current status value

see also section 4.8.1.Sensor Status Flags

Note

Gets the actual value of the specified temperature sensor.

5.10.4 CgosTemperatureSetLimits

CGOS API version

1.00.000 and later

Declaration

```
bool   gosTemperatureSetLimits(HCGOS   hCgos,   unsigned   long   dwUnit,
CGOSTEMPERATUREINFO *pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure see also section 4.8.3 Temperature Information Structure

Note

Meant to set the limits for the specified temperature sensor.

5.11 Function Group CgosFan*

The `CgosFan*` function group is used to access and control all the fans sensors in the system.

5.11.1 CgosFanCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosFanCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Returns the number of installed fan sensors in the system.

5.11.2 CgosFanGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosFanGetInfo(HCGOS  hCgos,  unsigned  long  dwUnit,  CGOSFANINFO
*pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure
	see also section 4.8.5 Fan Information structure

Note

Gets the information structure of the specified fan sensor.

5.11.3 CgosFanGetCurrent

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosFanGetCurrent(HCGOS  hCgos,  unsigned  long  dwUnit,  unsigned
long *pdwSetting, unsigned long *pdwStatus)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	pointer to the sensor's current measured value
`pdwStatus`	pointer to the sensor's current status value
	see also section 4.8.1 Sensor Status Flags

Note

Gets the actual value of the specified fan sensor.

5.11.4 CgosFanSetLimits

CGOS API version

1.00.000 and later

Declaration

```
bool  CgosFanSetLimits(HCGOS  hCgos,  unsigned  long  dwUnit,  CGOSFANINFO
*pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure
	see also section 4.8.5 Fan Information structure

Note

Set the limits for the specified fan sensor. Writing a value from 0 to 100 to the field `dwOutMax` ***in*** `CGOSFANINFO` ***sets the selected fan to 0% to 100% of its nominal speed.***

5.12 Function Group CgosVoltage*

The `CgosVoltage*` function group is used to access and control all the voltage sensors in the system.

5.12.1 CgosVoltageCount

CGOS API version

1.00.000 and later

Declaration

```
ulong CgosVoltageCount(HCGOS hCgos)
```

Input

`hCgos`	the board handle

Note

Returns the number of installed voltage sensors in the system.

5.12.2 CgosVoltageGetInfo

CGOS API version

1.00.000 and later

Declaration

```
bool CgosVoltageGetInfo(HCGOS hCgos, unsigned long dwUnit, CGOSVOLTAGEINFO *pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure
	see also section 4.8.7 Voltage Information structure

Note

Gets the information structure of the specified voltage sensor.

5.12.3 CgosVoltageGetCurrent

CGOS API version

1.00.000 and later

Declaration

```
bool CgosFanGetCurrent(HCGOS hCgos, unsigned long dwUnit, unsigned long *pdwSetting, unsigned long *pdwStatus)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pdwSetting`	pointer to the sensor's current measured value

`pdwStatus`	pointer to the sensor's current status value see also section 4.8.1 Sensor Status Flags

Note

Gets the actual value of the specified voltage sensor.

5.12.4 CgosVoltageSetLimits

CGOS API version

1.00.000 and later

Declaration

```
bool   CgosVoltageSetLimits(HCGOS   hCgos,   unsigned   long   dwUnit,
CGOSVOLTAGEINFO *pInfo)
```

Input

`hCgos`	the board handle
`dwUnit`	see section 5.1.4 Unit numbers
`pInfo`	pointer to the sensor information structure see also section 4.8.7 Voltage Information structure

Note

Meant to set the limits for the specified voltage sensor.