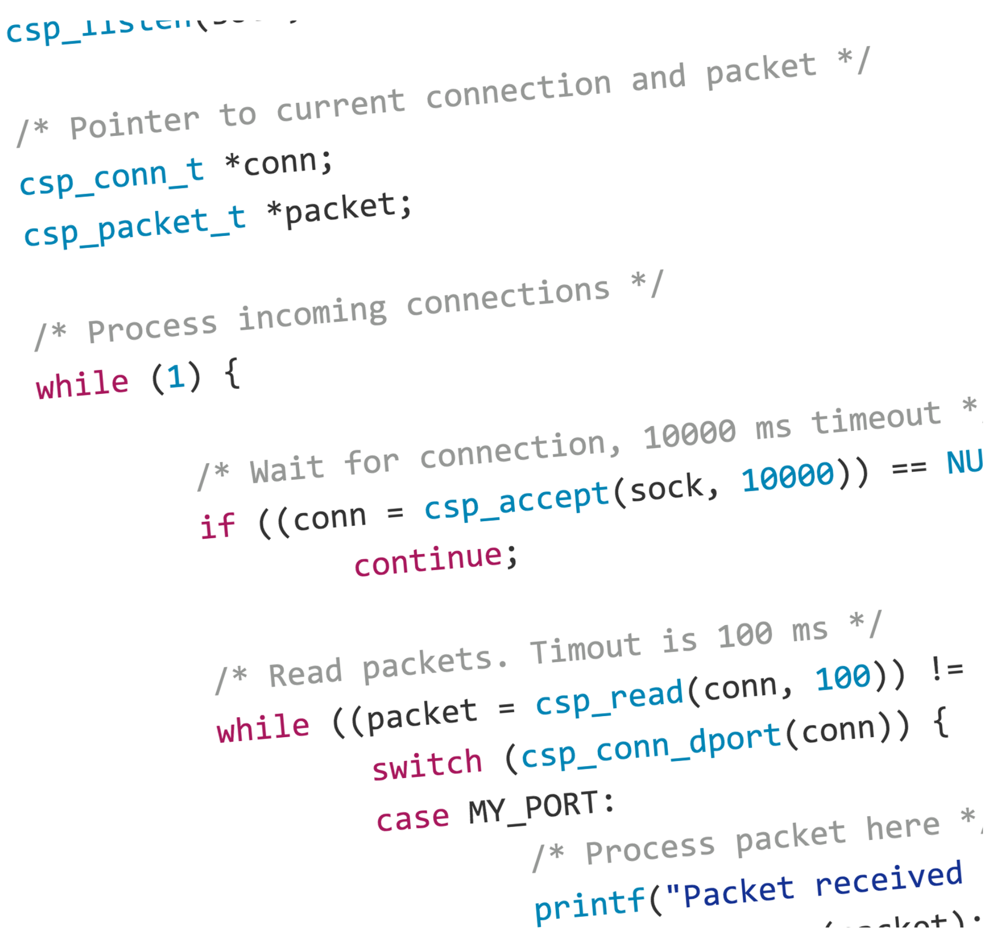


NanoSoft

Product Interface Application

Manual

Software Development Kit

Release 2.1.1

GomSpace A/S | Langagervej 6 | DK-9220 Aalborg East | Denmark
gomspace.com | sales@gomspace.com | T: +45 71 741 741

Table of Contents

1. Product Interface Application

1.1 Introduction

The Product Interface Application includes interfaces for most standard GomSpace products, e.g. NanoCom AX100, NanoCam C1U.

It comes with full source code for all clients and a number of support libraries.

It is shipped as a single archive, which includes all the source code, build scripts, documentation (this document), etc. The archive is named *gs-sw-nanosoft-product-interface-application-<version>.tar.gz*, hereinafter refered to as *<archive.tar.gz>*.

Building Product Interface Application produces a single executable file named *csp-client*.

It can also be used as an example of how to develop GomSpace compatible applications for Linux.

Included product interfaces can be found in Section 2.

SPI and I^{2}C communication is supported using the *Aardvark I2C/SPI Host Adapter*, see Section 3.1.

1.1.1 Getting started

Prerequisites

The software has been built and verified on Ubuntu 16.04.5 LTS (http://www.ubuntu.com), with following tools:

- Python 2.7.12

Install Ubuntu prerequisites:

```
$ sudo apt install build-essential libsocketcan-dev libzmq3-dev
```

Unpacking Product Interface Application

Use following steps to extract all files from the archive into a working directory on your Linux system.

1. Create a workspace folder, e.g. ~/workspace. All files will be extracted to a sub-folder within the workspace folder:

   ```
   $ mkdir ~/workspace
   ```

2. Copy the archive to the workspace folder:

   ```
   $ cp <archive.tar.gz> ~/workspace
   ```

3. Change directory to the workspace folder:

   ```
   $ cd ~/workspace
   ```

4. Unpack the archive:

   ```
   $ tar xfzv <archive.tar.gz>
   ```

You now have the source code unpacked in: *~/workspace/gs-sw-nanosoft-product-interface-application-<version>*.

Building Product Interface Application

Use the following steps to build the Product Interface Application.

Building the first time:

1. Change directory to the Product Interface Application folder:

```
$ cd ~/workspace/gs-sw-nanosoft-product-interface-application-<version>
```

2. Bootstrap buildtools:

```
$ ./tools/buildtools/gsbuildtools_bootstrap.py
```

3. Build using Waf:

```
$ ./waf distclean configure build
```

Subsequent building:

1. Change directory to the Product Interface Application folder:

```
$ cd ~/workspace/gs-sw-nanosoft-product-interface-application-<version>
```

2. Build using Waf:

```
$ ./waf
```

More details about using BuildTools can be found in Section 4.1.

Running Product Interface Application

The Product Interface Application supports a number of command line arguments:

```
$ ./build/csp-client -h
Usage: csp-client [OPTION...]
Product Interface Application

 Aardvard I2C/SPI dongle
      --aardvark-devices     Show all devices
  -I, --aardvark-i2c[=DEVICE]   Add I2C interface
                             DEVICE=0,speed=100000,address=1,device=255
  -S, --aardvark-spi[=DEVICE]   Add SPI interface
                             DEVICE=0,speed=400000,slave=255

 CSP
  -a, --csp-address=ADDR     Set address, default: 8
  -c, --csp-can[=DEVICE]     Add CAN interface
                             DEVICE=can0
  -i, --csp-i2c[=DEVICE]     Add I2C interface
                             DEVICE=0,speed=100000,address=1,device=255
  -k, --csp-kiss[=DEVICE]    Add KISS over UART interface
                             DEVICE=/dev/ttyUSB0,if=KISS,speed=500000
  -R, --csp-rtable=RTABLE    Set routing table
                             RTABLE=<address>/<mask> <interface> [mac]
                             Example: "0/0 ZMQHUB 24, 24/2 ZMQHUB"
  -z, --csp-zmq[=SERVER]     Add ZMQ interface
                             SERVER=localhost
  Examples:
  CAN: configure address 10 and CAN interface can0:
    $ <application> -a10 -ccan0
  KISS: configure address 10 and uart on /dev/ttyUSB0 at baudrate 500000:
```

```
    $ <application> -a10 -k/dev/ttyUSB0,speed=500000
  I2C: configure address 10 and I2C Aardvark dongle with id 2238384015, speed
  400K:
    $ <application> -a10 -i2238384015,speed=400000
  ZMQ: configure address 10 and ZMQ proxy on localhost:
    $ <application> -a10 -zlocalhost

  -?, --help                 Give this help list
  -h, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

2. Product Interfaces

2.1 ADCS (libadcs_client)

2.1.1 Introduction

This is the client interface for ADCS.

The client can be included in the A3200 or Linux application.

Please refer to manual for ADCS for further details.

2.1.2 Commands

Commands (GOSH) are grouped under *adcs*.

2.1.3 Parameters

Please refer to the ADCS manual, which describes the parameters present on the ADCS.

2.2 NanoCam C1U (nanocam2_client)

2.2.1 Introduction

This is the client interface for the NanoCam C1U.

The client can be included in the A3200 or Linux application.

Please see the NanoCam C1U manual for further details.

2.2.2 Client API

The client API consists of a set of wrapper functions that simplify the CSP interface to the NanoCam C1U. These functions are implemented in the `nanocam.c` file and can be integrated in custom code by including the `nanocam.h` header file. The `cmd_nanocam.c` implements the GOSH commands for the NanoCam and can be used as an additional reference for the use of the client API.

All the client functions specify a `timeout` argument that is used to specify the maximum number of milliseconds to wait for a reply. The client interface automatically performs endian conversion to network byte order on all arguments.

The functions return 0 on success and a non-zero error code on error. The error codes are listed below:

```
#define NANOCAM_ERROR_NONE          0 /** No error */
#define NANOCAM_ERROR_NOMEM         1 /** Not enough space */
#define NANOCAM_ERROR_INVAL         2 /** Invalid value */
#define NANOCAM_ERROR_IOERR         3 /** I/O error */
#define NANOCAM_ERROR_NOENT         4 /** No such file or directory */
```

Similar to the GOSH interface, the client API operates on a single C1U at a time. The CSP address of this C1U is set using the `nanocam_set_node` function. By default, the commands operate on CSP node `NANOCAM_DEFAULT_ADDRESS` which is currently set to 6. If the camera address has not been changed, it is not necessary to call `nanocam_set_node`.

void **nanocam_set_node**(uint8_t *node*)
: This function sets the CSP address of the NanoCam C1U. All other API functions use this CSP address.

Image Capture

Capture of images is provided by the `nanocam_snap` function.

int **nanocam_snap**(*cam_snap_t *snap*, *cam_snap_reply_t *reply*, unsigned int *timeout*)
: This function is used to capture an image. The capture parameters should be set in the `snap` structure argument prior to calling this function. The reply from the camera is returned in the `reply` struct.

---

cam_snap_t
: The `cam_snap_t` struct is used to specify the arguments for the snap command. Each field of the structure is documented below.

uint32_t **cam_snap_t.flags**
: This argument supplies optional flag bits that modifies the behavior of the snap request.

```
#define NANOCAM_SNAP_FLAG_AUTO_GAIN   (1 <<  0) /** Automatically adjust gain */
#define NANOCAM_SNAP_FLAG_STORE_RAM   (1 <<  8) /** Store snapped image to RAM */
#define NANOCAM_SNAP_FLAG_STORE_FLASH (1 <<  9) /** Store snapped image to flash */
#define NANOCAM_SNAP_FLAG_STORE_THUMB (1 << 10) /** Store thumbnail to flash */
#define NANOCAM_SNAP_FLAG_STORE_TAGS  (1 << 11) /** Store image tag file */
#define NANOCAM_SNAP_FLAG_NOHIST      (1 << 16) /** Do not calculate histogram */
#define NANOCAM_SNAP_FLAG_NOEXPOSURE  (1 << 17) /** Do not adjust exposure */
```

uint8_t **cam_snap_t.count**
: Number of images to capture. If set to zero a single image capture will be performed. When capturing multiple images, the `nanocam_snap` function will only return the `cam_snap_reply_t` for the first image.

uint8_t **cam_snap_t.format**
: Output format to use when `NANOCAM_SNAP_FLAG_STORE_FLASH` or `NANOCAM_SNAP_FLAG_STORE_RAM` is enabled in the flags field. Valid output formats are:

```
#define NANOCAM_STORE_RAW          0 /* Store RAW sensor output */
#define NANOCAM_STORE_BMP          1 /* Store bitmap output */
#define NANOCAM_STORE_JPG          2 /* Store JPEG compressed output */
#define NANOCAM_STORE_DNG          3 /* Store DNG output (Raw, digital negative) */
```

uint16_t **cam_snap_t.delay**
: Optional delay between captures in milliseconds. Only applicable when count > 1.

uint16_t **cam_snap_t.width**
: Image width in pixels. Set to 0 to use default (maximum = 2048) size.

uint16_t **cam_snap_t.height**
: Image height in pixels. Set to 0 to use default (maximum = 1536) size.

uint16_t **cam_snap_t.top**
: Image crop rectangle top coordinate. Must be set to 0.

uint16_t **cam_snap_t.left**
: Image crop rectangle left coordinate. Must be set to 0.

---

cam_snap_reply_t
This struct contains the reply of a image capture. The reply contains arrays with information of average brightness and distribution. A couple of defines are used for the length of these arrays:

```
#define NANOCAM_SNAP_COLORS          4
#define NANOCAM_SNAP_HIST_BINS       16
```

uint8_t **cam_snap_reply_t.result**
Result of the capture. One of the error codes listed in the introduction.

uint8_t **cam_snap_reply_t.seq**
Zero-index sequence number when capturing multiple images, i.e. when `count` > 1 in the `cam_snap_t` argument.

uint8_t[NANOCAM_SNAP_COLORS] **cam_snap_reply_t.light_avg**
Array of `NANOCAM_SNAP_COLORS` elements corresponding to the average brightness of all pixels plus the red, green and blue channel pixels. The numbers are scaled from 0-255, so e.g. 128 corresponds to an average brightness of 50%.

uint8_t[NANOCAM_SNAP_COLORS] **cam_snap_reply_t.light_peak**
Array of `NANOCAM_SNAP_COLORS` elements corresponding to the estimated peak brightness of all pixels plus the red, green and blue channel pixels.

uint8_t[NANOCAM_SNAP_COLORS] **cam_snap_reply_t.light_min**
Array of `NANOCAM_SNAP_COLORS` elements corresponding to the minimum brightness of all pixels plus the red, green and blue channel pixels.

uint8_t[NANOCAM_SNAP_COLORS] **cam_snap_reply_t.light_max**
Array of `NANOCAM_SNAP_COLORS` elements corresponding to the maximum brightness of all pixels plus the red, green and blue channel pixels.

uint8_t[NANOCAM_SNAP_COLORS][NANOCAM_SNAP_HIST_BINS] **cam_snap_reply_t.hist**
Array of `NANOCAM_SNAP_COLORS` elements each consisting on an array of `NANOCAM_SNAP_HIST_BINS` bins. Each bin contains a number from 0 to 255 matching the distribution of brightness, with the sum of all bins being 255. Thus, if a bin is 128, 50% of all pixels falls within the brightness range covered by that particular bin.

Image Storage

Storage of images captured to the snap buffer is provided by the `nanocam_store` functions. Images are stored in the `/mnt/data/images` directory on the camera file system.

int **nanocam_store**(*cam_store_t *store*, *cam_store_reply_t *reply*, unsigned int *timeout*)
This function is used to store a captured image from the snap buffer to persistant storage and/or RAM.

---

cam_store_t
This struct is used to supply store arguments to the `nanocam_store` function.

uint8_t **cam_store_t.format**
Output format of the stored image. See the argument list to `cam_snap_t.format` for a list of valid options.

uint8_t **cam_store_t.scale**
This argument is currently unused and should be set to 0.

uint32_t **cam_store_t.flags**
This argument supplies optional flag bits that modifies the behavior of the store request. If the `NANOCAM_STORE_FLAG_FREEBUF` flag is cleared, a copy of the stored image will be kept in the RAM list.

```
#define NANOCAM_STORE_FLAG_FREEBUF      (1 << 0) /* Free buffer after store */
#define NANOCAM_STORE_FLAG_THUMB        (1 << 1) /* Create thumbnail */
#define NANOCAM_STORE_FLAG_TAG          (1 << 2) /* Create tag file */
```

char[40] **cam_store_t.filename**
Filename of the stored image. The file type is not required to match the file format, but it is recommend to e.g. store JPEG images with a *.jpg* ending. Setting this field to an empty string, i.e. set filename[0] to `\0`, will only store the image in the RAM list.

---

cam_store_reply_t
This struct contains the reply of a image store command.

uint8_t **cam_store_reply_t.result**
Result of the store command. One of the error codes listed in the introduction.

uint32_t **cam_store_reply_t.image_ptr**
Address of the RAM copy of the stored image.

uint32_t **cam_store_reply_t.image_size**
Size in bytes of the RAM copy of the stored image.

Modifying Sensor Registers

The image sensor registers can be adjusted using the `nanocam_reg_read` and `nanocam_reg_write` functions. Any modifications of the registers are volatile, and may be overridden by the auto-gain and exposure setting algorithms.

Note: For normal operation, it is not necessary to adjust sensor registers directly. Instead the image configuration parameters from the image table should be used.

Please refer to the *Aptina MT9T031* datasheet for a description of individual sensor registers.

int **nanocam_reg_read** (uint8_t *reg*, uint16_t **value*, unsigned int *timeout*)
This function reads a sensor register and returns the current value. The `reg` argument contains the address of the register to read and the current value is returned in the `value` pointer.

int **nanocam_reg_write** (uint8_t *reg*, uint16_t *value*, unsigned int *timeout*)
Use this function to update the value of a register. The `reg` contains the register address and `value` contains the new value to write to the register.

In-memory Images

int **nanocam_img_list** (*nanocam_img_list_cb cb*, unsigned int *timeout*)
Call this function to list all images in the RAM list. The `nanocam_img_list_cb` callback will be called once for each image element in the list.

typedef void **(*nanocam_img_list_cb)** (int *seq*, cam_list_element_t **elm*)
Implement an image listing callback function matching this prototype, and supply it to the `nanocam_img_list` list function. If no images are available in memory, the callback is called with `elm` set to NULL.

int **nanocam_img_list_flush** (unsigned int *timeout*)
This function flushes all images stored in the RAM image list. Note that the current image in the snap buffer can not be flushed, so a single image will always be returned by `nanocam_img_list`.

Focus Assist Routine

int **nanocam_focus** (uint8_t *algorithm*, uint32_t **af*, unsigned int *timeout*)

This function runs a single iteration of the focus assist algorithm. The `algorithm` argument is used to select between different algorithms. Currently, `NANOCAM_AF_JPEG` is the only supported option.

The focus assist algorithm captures an image, JPEG compresses the center of the image and returns the size of the compressed data in the `af` pointer. The premise is that a more focused image will be more difficult to compress, giving a larger size of the compressed data. Continuously running this algorithm can thus be used to adjust the focus until a maximum size is found.

Data Partition Recovery

int **nanocam_recoverfs** (unsigned int *timeout*)

This function can be used to recreate the data file system. Note that this erases ALL images stored on the camera. If you just want to delete all captured images, using the FTP `rm` command is much faster and a safer option than rebuilding the entire file system.

2.2.3 Commands

Commands (GOSH) are grouped under *cam*.

2.2.4 Parameters

Please refer to the NanoCam C1U manual, which describes the parameters present on the NanoCam C1U.

2.3 NanoCom ADS-B (gatoss-uc_client)

2.3.1 Introduction

This is the client interface for the NanoCom ADS-B.

The client can be included in the A3200 or Linux application.

Please see the NanoCom ADS-B manual for further details.

2.3.2 Commands

Commands (GOSH) are grouped under *gatoss*.

2.3.3 Parameters

Please refer to the NanoCom ADS-B manual, which describes the parameters present on the NanoCom ADS-B.

2.4 NanoCom AX100 (nanocom-ax_client)

2.4.1 Introduction

This is the client interface for the NanoCom AX100.

The client can be included in the A3200 or Linux application.

Please refer to manual for NanoCom AX100 for further details.

2.4.2 Commands

Commands (GOSH) are grouped under *ax100*.

2.5 NanoPower BPX (nanopower-bpx_client)

2.5.1 Introduction

This is the client interface for the NanoPower BPX.

The client can be included in the A3200 or Linux application.

Please see the NanoPower BPX manual for further details.

2.5.2 Commands

Commands (GOSH) are grouped under *bpx*.

2.6 NanoPower P31u (nanopower_client)

2.6.1 Introduction

This is the client interface for the NanoPower P31u.

The client can be included in the A3200 or Linux application.

Please see the NanoPower P31u manual for further details.

2.6.2 Commands

Commands (GOSH) are grouped under *eps*.

2.7 NanoPower P60 ACU (p60-acu_client)

2.7.1 Introduction

This is the client interface for the NanoPower P60 ACU 200 and NanoPower P60 ACU 210 products.

The client can be included in the A3200 or Linux application.

Please see the NanoPower P60 ACU 200 manual for further details.

2.7.2 Commands

Commands (GOSH) are grouped under *p60acu*.

2.7.3 Parameters

Please refer to the NanoPower P60 ACU manual, which describes the parameters present on the NanoPower P60 ACU.

2.8 NanoPower P60 Dock (p60-dock_client)

2.8.1 Introduction

This is the client interface for the NanoPower P60 Dock.

The client can be included in the A3200 or Linux application.

Please refer to manual for NanoPower P60 Dock for further details.

2.8.2 Commands

Commands (GOSH) are grouped under *p60dock*.

2.8.3 Parameters

Please refer to the NanoPower P60 Dock manual, which describes the parameters present on the NanoPower P60 Dock.

2.9 NanoPower P60 PDU (p60-pdu_client)

2.9.1 Introduction

This is the client interface for the NanoPower P60 PDU.

The client can be included in the A3200 or Linux application.

Please refer to manual for NanoPower P60 PDU for further details.

2.9.2 Commands

Commands (GOSH) are grouped under *p60pdu*.

2.9.3 Parameters

Please refer to the NanoPower P60 PDU manual, which describes the parameters present on the NanoPower P60 PDU.

2.10 NanoPower P60 Library (libp60_client)

2.10.1 Introduction

NanoPower P60 Library is a support library for NanoPower P60 products: NanoPower P60 Dock, NanoPower P60 PDU, NanoPower P60 ACU 200 and NanoPower P60 ACU 210.

2.10.2 Commands

Commands (GOSH) are grouped under *power*.

3. Libraries

3.1 Aardvark (libaardvark)

3.1.1 Introduction

The Aardvark library provides GomSpace compliant drivers for the Aardvark I2C/SPI Host Adapter (https://www.totalphase.com/products/aardvark-i2cspi/).

The drivers implements the cross-platform I2C and SPI APIs, defined in the Utility library. This enables the Aardvark I2C/SPI Host Adapter to be used for interfacing to GomSpace products running I2C or SPI.

The Aardvark I2C/SPI Host Adapter can be purchased online at https://www.totalphase.com.

Product Interface Application

The Aardvark I2C/SPI Host Adapter can be used with Product Interface Application, for interfacing with GomSpace products. The Product Interface Application (*csp-client*) comes with the required driver support. When the adapter is connected and configured correctly, it will show up in the list of adapters (devices):

```
$ ./build/csp-client --aardvark-devices
2238709334
```

The adapter can now be used to interface with a GomSpace unit, in this example using the standard GomSpace Parameter Protocol over I^2C:

```
$ ./build/csp-client -I
1546944025.576460 W csp: gs_csp_rtable_load: no interfaces configured
csp-client # pp i2c_init 0 9
csp-client # pp get_uint32 4 0
value(s): 1262
```

If no adapters are detected or communication fails, please check the *Aarvark* manual, especially the part about *udev* rules.

Known limitations

Please note, that when using the Aardvark I2C/SPI Host Adapter for CSP over I^2C, one thread will be running all the time. This is due to the Aardvark I2C/SPI Host Adapter driver implementation.

3.2 GomSpace CSP (libgscsp)

3.2.1 Introduction

The GomSpace CSP library is a GomSpace extension to the open source *CubeSat Space Protocol*.

The GomSpace CSP library provides:

- convience wrapping of CSP functionality, e.g. initialization.
- definition of standard CSP ports.

- connecting low-level drivers (e.g. CAN, I2C from Embed library) with CSP interaces
- generic CSP service dispatcher, forwards an incoming connection to a service handler.

3.2.2 Commands

All commands provided by the GomSpace CSP library are root commands, and therefor not grouped under a single command:

```
nanomind #
  ping                    csp: Ping
  rps                     csp: Remote ps
  memfree                 csp: Memory free
  buffree                 csp: Buffer free
  reboot                  csp: Reboot
  shutdown                csp: Shutdown
  uptime                  csp: Uptime
  cmp                     csp: Management
  route                   csp: Show routing table
  ifc                     csp: Show interfaces
  conn                    csp: Show connection table
  rdpopt                  csp: Set RDP options
```

Most of the commands works both on the local node and a remote node. An example is the *cmp ident* command:

```
nanomind # cmp ident
Hostname: nanomind
Model:    A3200-SDK-Linux
Revision: 2.4.1-13-ge3f9a8f+
Date:     Dec 18 2018
Time:     16:11:52
nanomind # cmp ident 8
Hostname: csp-client
Model:    CSP-client
Revision: 2.0.2-6-g3c3768b+
Date:     Dec 18 2018
Time:     16:17:58
```

3.2.3 CubeSat Space Protocol

The Cubesat Space Protocol

Cubesat Space Protocol (CSP) is a small protocol stack written in C. CSP is designed to ease communication between distributed embedded systems in smaller networks, such as Cubesats. The design follows the TCP/IP model and includes a transport protocol, a routing protocol and several MAC-layer interfaces. The core of libcsp includes a router, a socket buffer pool and a connection oriented socket API.

The protocol is based on a 32-bit header containing both transport and network-layer information. Its implementation is designed for, but not limited to, embedded systems such as the 8-bit AVR microprocessor and the 32-bit ARM and AVR from Atmel. The implementation is written in GNU C and is currently ported to run on FreeRTOS or POSIX operating systems such as Linux.

The idea is to give sub-system developers of cubesats the same features of a TCP/IP stack, but without adding the huge overhead of the IP header. The small footprint and simple implementation allows a small 8-bit system with less than 4 kB of RAM to be fully connected on the network. This allows all subsystems to provide their services on the same network level, without any master node required. Using a service oriented architecture has several advantages compared to the traditional mater/slave topology used on many cubesats.

- Standardised network protocol: All subsystems can communicate with eachother

- Service loose coupling: Services maintain a relationship that minimizes dependencies between subsystems
- Service abstraction: Beyond descriptions in the service contract, services hide logic from the outside world
- Service reusability: Logic is divided into services with the intention of promoting reuse.
- Service autonomy: Services have control over the logic they encapsulate.
- Service Redundancy: Easily add redundant services to the bus
- Reduces single point of failure: The complexity is moved from a single master node to several well defines services on the network

The implementation of LibCSP is written with simplicity in mind, but it's compile time configuration allows it to have some rather advanced features as well:

Features

- Thread safe Socket API
- Router task with Quality of Services
- Connection-oriented operation (RFC 908 and 1151).
- Connection-less operation (similar to UDP)
- ICMP-like requests such as ping and buffer status.
- Loopback interface
- Very Small Footprint 48 kB code and less that 1kB ram required on ARM
- Zero-copy buffer and queue system
- Modular network interface system
- Modular OS interface, ported to FreeRTOS, windows (cygwin) and Linux
- Broadcast traffic
- Promiscuous mode
- Encrypted packets with XTEA in CTR mode
- Truncated HMAC-SHA1 Authentication (RFC 2104)

LGPL Software license The source code is available under an LGPL 2.1 license. See COPYING for the license text.

History

The idea was developed by a group of students from Aalborg University in 2008. In 2009 the main developer started working for GomSpace, and CSP became integrated into the GomSpace products. The protocol is based on a 32-bit header containing both transport, network and MAC-layer information. It's implementation is designed for, but not limited to, embedded systems such as the 8-bit AVR microprocessor and the 32-bit ARM and AVR from Atmel. The implementation is written in C and is currently ported to run on FreeRTOS and POSIX and pthreads based operating systems like Linux and BSD. The three letter acronym CSP was originally an abbreviation for CAN Space Protocol because the first MAC-layer driver was written for CAN-bus. Now the physical layer has extended to include spacelink, I2C and RS232, the name was therefore extended to the more general CubeSat Space Protocol without changing the abbreviation.

Satellites using CSP This is the known list of satellites or organisations that uses CSP.

- GomSpace GATOSS GOMX-1
- AAUSAT-3
- EgyCubeSat
- EuroLuna
- NUTS
- Hawaiian Space Flight Laboratory
- GomSpace GOMX-3

Structure

The Cubesat Space Protocol library is structured as shown in the following table:

Folder	Description
libcsp/include/csp	Main include files
libcsp/include/csp/arch	Architecture include files
libcsp/include/csp/interfaces	Interface include files
libcsp/include/csp/drivers	Drivers include files
libcsp/src	Main modules for CSP: io, router, connections, services
libcsp/src/interfaces	Interface modules for CAN, I2C, KISS, LOOP and ZMQHUB
libcsp/src/drivers/can	Driver for CAN
libcsp/src/drivers/usart	Driver for USART
libcsp/src/arch/freertos	FreeRTOS architecture module
libcsp/src/arch/macosx	Mac OS X architecture module
libcsp/src/arch/posix	Posix architecture module
libcsp/src/arch/windows	Windows architecture module
libcsp/src/rtable	Routing table module
libcsp/transport	Transport module, UDP and RDP
libcsp/crypto	Crypto module
libcsp/utils	Utilities
libcsp/bindings/python	Python wrapper for libcsp
libcsp/examples	CSP examples (source code)
libasf/doc	The doc folder contains the source code for this documentation

CSP Interfaces

This is an example of how to implement a new layer-2 interface in CSP. The example is going to show how to create a *csp_if_fifo*, using a set of [named pipes](http://en.wikipedia.org/wiki/Named_pipe). The complete interface example code can be found in *examples/fifo.c*. For an example of a fragmenting interface, see the CAN interface in *src/interfaces/csp_if_can.c*.

CSP interfaces are declared in a *csp_iface_t* structure, which sets the interface nexthop function and name. A maximum transmission unit can also be set, which forces CSP to drop outgoing packets above a certain size. The fifo interface is defined as:

```
#include <csp/csp.h>
#include <csp/csp_interface.h>

csp_iface_t csp_if_fifo = {
    .name = "fifo",
    .nexthop = csp_fifo_tx,
    .mtu = BUF_SIZE,
};
```

Outgoing traffic The nexthop function takes a pointer to a CSP packet and a timeout as parameters. All outgoing packets that are routed to the interface are passed to this function:

```
int csp_fifo_tx(csp_packet_t *packet, uint32_t timeout) {
    write(tx_channel, &packet->length, packet->length + sizeof(uint32_t) + sizeof(uint16_
↪t));
    csp_buffer_free(packet);
    return 1;
}
```

In the fifo interface, we simply transmit the header, length field and data using a write to the fifo. CSP does not dictate the wire format, so other interfaces may decide to e.g. ignore the length field if the physical layer provides start/stop flags.

Important notice: If the transmission succeeds, the interface must free the packet and return 1. If transmission fails, the nexthop function should return 0 and not free the packet, to allow retransmissions by the caller.

Incoming traffic The interface also needs to receive incoming packets and pass it to the CSP protocol stack. In the fifo interface, this is handled by a thread that blocks on the incoming fifo and waits for packets:

```
void * fifo_rx(void * parameters) {
    csp_packet_t *buf = csp_buffer_get(BUF_SIZE);
    /* Wait for packet on fifo */
    while (read(rx_channel, &buf->length, BUF_SIZE) > 0) {
        csp_qfifo_write(buf, &csp_if_fifo, NULL);
        buf = csp_buffer_get(BUF_SIZE);
    }
}
```

A new CSP buffer is preallocated with csp_buffer_get(). When data is received, the packet is passed to CSP using *csp_qfifo_write()* and a new buffer is allocated for the next packet. In addition to the received packet, *csp_qfifo_write()* takes two additional arguments:

```
void csp_qfifo_write(csp_packet_t *packet, csp_iface_t *interface, CSP_BASE_TYPE␣
↪*pxTaskWoken);
```

The calling interface must be passed in *interface* to avoid routing loops. Furthermore, *pxTaskWoken* must be set to a non-NULL value if the packet is received in an interrupt service routine. If the packet is received in task context, NULL must be passed. 'pxTaskWoken' only applies to FreeRTOS systems, and POSIX system should always set the value to NULL.

csp_qfifo_write will either accept the packet or free the packet buffer, so the interface must never free the packet after passing it to CSP.

Initialization In order to initialize the interface, and make it available to the router, use the following function found in *csp/csp_interface.h*:

```
csp_route_add_if(&csp_if_fifo);
```

This actually happens automatically if you try to call *csp_route_add()* with an interface that is unknown to the router. This may however be removed in the future, in order to ensure that all interfaces are initialised before configuring the routing table. The reason is, that some products released in the future may ship with an empty routing table, which is then configured by a routing protocol rather than a static configuration.

In order to setup a manual static route, use the following example where the default route is set to the fifo interface:

```
csp_route_set(CSP_DEFAULT_ROUTE, &csp_if_fifo, CSP_NODE_MAC);
```

All outgoing traffic except loopback, is now passed to the fifo interface's nexthop function.

Building the example The fifo examples can be compiled with:

```
% gcc csp_if_fifo.c -o csp_if_fifo -I<CSP PATH>/include -L<CSP PATH>/build -lcsp -
↪lpthread -lrt
```

The two named pipes are created with:

```
% mkfifo server_to_client client_to_server
```

How CSP uses memory

CSP has been written for small microprocessor systems. The way memory is handled is therefore a tradeoff between the amount used and the code efficiency. This section tries to give some answers to what the memory is used for and how it it used. The primary memory blocks in use by CSP is:

- Routing table
- Ports table
- Connection table
- Buffer pool
- Interface list

Tables The reason for using tables for the routes, ports and connections is speed. When a new packet arrives the core of CSP needs to do a quick lookup in the connection so see if it can find an existing connection to which the packet matches. If this is not found, it will take a lookup in the ports table to see if there are any applications listening on the incoming port number. Another argument of using tables are pre-allocation. The linker will reserve an area of the memory for which the routes and connections can be stored. This avoid an expensive *malloc()* call during initialization of CSP, and practically costs zero CPU instructions. The downside of using tables are the wasted memory used by unallocated ports and connections. For the routing table the argumentation is the same, pre-allocation is better than calling *malloc()*.

Buffer Pool The buffer handling system can be compiled for either static allocation or a one-time dynamic allocation of the main memory block. After this, the buffer system is entirely self-contained. All allocated elements are of the same size, so the buffer size must be chosen to be able to handle the maximum possible packet length. The buffer pool uses a queue to store pointers to free buffer elements. First of all, this gives a very quick method to get the next free element since the dequeue is an O(1) operation. Furthermore, since the queue is a protected operating system primitive, it can be accessed from both task-context and interrupt-context. The *csp_buffer_get* version is for task-context and *csp_buffer_get_isr* is for interrupt-context. Using fixed size buffer elements that are preallocated is again a question of speed and safety.

A basic concept of the buffer system is called Zero-Copy. This means that from userspace to the kernel-driver, the buffer is never copied from one buffer to another. This is a big deal for a small microprocessor, where a call to *memcpy()* can be very expensive. In practice when data is inserted into a packet, it is shifted a certain number of bytes in order to allow for a packet header to be prepended at the lower layers. This also means that there is a strict contract between the layers, which data can be modified and where. The buffer object is normally casted to a *csp_packet_t*, but when its given to an interface on the MAC layer it's casted to a *csp_i2c_frame_t* for example.

Interface list The interface list is a simple single-ended linked list of references to the interface specification structures. These structures are static const and allocated by the linker. The pointer to this data is inserted into the list one time during setup of the interface. Each entry in the routing table has a direct pointer to the interface element, thereby avoiding list lookup, but the list is needed in order for the dynamic route configuration to know which interfaces are available.

The Protocol Stack

The CSP protocol stack includes functionality on all layers of the TCP/IP model:

Layer 1: Drivers Lib CSP is not designed for any specific processor or hardware peripheral, but yet these drivers are required in order to work. The intention of LibCSP is not to provide CAN, I2C or UART drivers for all platforms, however some drivers has been included for some platforms. If you do not find your platform supported, it is quite simple to add a driver that conforms to the CSP interfaces. For example the I2C driver just requires three functions: *init*, *send* and *recv*. For good stability and performance interrupt driven drivers are preferred in favor of polled drivers. Where applicable also DMA usage is recommended.

Layer 2: MAC interfaces CSP has interfaces for I2C, CAN, RS232 (KISS) and Loopback. The layer 2 protocol software defines a frame-format that is suitable for the media. CSP can be easily extended with implementations for even more links. For example a radio-link and IP-networks. The file *csp_interface.h* declares the rx and tx functions needed in order to define a network interface in CSP. During initialisation of CSP each interface will be inserted into a linked list of interfaces that is available to the router. In cases where link-layer addresses are required, such as I2C, the routing table supports specifying next-hop link-layer address directly. This avoids the need to implement an address resolution protocol to translate CSP addresses to I2C addresses.

Layer 3: Network Router The router core is the backbone of the CSP implementation. The router works by looking at a 32-bit CSP header which contains the delivery and source address together with port numbers for the connection. Each router supports both local delivery and forwarding of frames to another destination. Frames will never exit the router on the same interface that they arrives at, this concept is called split horizon, and helps prevent routing loops.

The main purpose of the router is to accept incoming packets and deliver them to the right message queue. Therefore, in order to listen on a port-number on the network, a task must create a socket and call the accept() call. This will make the task block and wait for incoming traffic, just like a web-server or similar. When an incoming connection is opened, the task is woken. Depending on the task-priority, the task can even preempt another task and start execution immediately.

There is no routing protocol for automatic route discovery, all routing tables are pre-programmed into the subsystems. The table itself contains a separate route to each of the possible 32 nodes in the network and the additional default route. This means that the overall topology must be decided before putting sub-systems together, as explained in the *topology.md* file. However CSP has an extension on port zero CMP (CSP management protocol), which allows for over-the-network routing table configuration. This has the advantage that default routes could be changed if for example the primary radio fails, and the secondary should be used instead.

Layer 4: Transport Layer LibCSP implements two different Transport Layer protocols, they are called UDP (unreliable datagram protocol) and RDP (reliable datagram protocol). The name UDP has not been chosen to be an exact replica of the UDP (user datagram protocol) known from the TCP/IP model, but they have certain similarities.

The most important thing to notice is that CSP is entirely a datagram service. There is no stream based service like TCP. A datagram is defined a block of data with a specified size and structure. This block enters the transport layer as a single datagram and exits the transport layer in the other end as a single datagram. CSP preserves this structure all the way to the physical layer for I2C, KISS and Loopback interfaces are used. The CAN-bus interface has to fragment the datagram into CAN-frames of 8 bytes, however only a fully completed datagram will arrive at the receiver.

UDP UDP uses a simple transmission model without implicit hand-shaking dialogues for guaranteeing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

UDP is very practical to implement request/reply based communication where a single packet forms the request and a single packet forms the reply. In this case a typical request and wait protocol is used between the client and server, which will simply return an error if a reply is not received within a specified time limit. An error would normally lead to a retransmission of the request from the user or operator which sent the request.

While UDP is very simple, it also has some limitations. Normally a human in the loop is a good thing when operating the satellite over UDP. But when it comes to larger file transfers, the human becomes the bottleneck. When a high-speed file transfer is initiated data acknowledgment should be done automatically in order to speed up the transfer. This is where the RDP protocol can help.

RDP CSP provides a transport layer extension called RDP (reliable datagram protocol) which is an implementation of RFC908 and RFC1151. RDP provides a few additional features:

- Three-way handshake
- Flow Control
- Data-buffering
- Packet re-ordering
- Retransmission
- Windowing
- Extended Acknowledgment

For more information on this, please refer to RFC908.

Network Topology

CSP uses a network oriented terminology similar to what is known from the Internet and the TCP/IP model. A CSP network can be configured for several different topologies. The most common topology is to create two segments, one for the Satellite and one for the Ground-Station.

```
              I2C BUS
  _______________________________
 /       |       |       |       \
+---+   +---+   +---+   +---+   +---+
|OBC|   |COM|   |EPS|   |PL1|   |PL2|       Nodes 0 - 7 (Space segment)
+---+   +---+   +---+   +---+   +---+
          ^
          |  Radio
          v
        +---+           +----+
        |TNC|  -------  | PC |              Nodes 8 - 15 (Ground segment)
        +---+    USB    +----+

        Node 9          Node 10
```

The address range, from 0 to 15, has been segmented into two equal size segments. This allows for easy routing in the network. All addresses starting with binary 1 is on the ground-segment, and all addresses starting with 0 is on the space segment. From CSP v1.0 the address space has been increased to 32 addresses, 0 to 31. But for legacy purposes, the old 0 to 15 is still used in most products.

The network is configured using static routes initialised at boot-up of each sub-system. This means that the basic routing table must be assigned compile-time of each subsystem. However each node supports assigning an individual route to every single node in the network and can be changed run-time. This means that the network topology can be easily reconfigured after startup.

Maximum Transfer Unit

There are two things limiting the MTU of CSP.

1. The pre-allocated buffer pool's allocation size
2. The link layer protocol.

So let's assume that you have made a protocol called KISS with a MTU of 256. The 256 is the total amount of data that you can put into the CSP-packet. However, you need to take the overhead of the link layer into account. Typically this could consist of a length field and/or a start/stop flag. So the actual frame size on the link layer would for example be 256 bytes of data + 2 bytes sync flag + 2 bytes length field.

This requires a buffer allocation of at lest 256 + 2 + 2. However, the CSP packet itself has some reserved bytes in the beginning of the packet (which you can see in csp.h) - so the recommended buffer allocation size is MAX MTU + 16 bytes. In this case the max MTU would be 256.

If you try to pass data which is longer than the MTU, the chance is that you will also make a buffer overflow in the CSP buffer pool. However, lets assume that you have two interfaces one with an MTU of 200 bytes and another with an MTU of 100 bytes. In this case you might successfully transfer 150 bytes over the first interface, but the packet will be rejected once it comes to the second interface.

If you want to increase your MTU of a specific link layer, it is up to the link layer protocol to implement its own fragmentation protocol. A good example is CAN-bus which only allows a frame size of 8 bytes. libcsp have a small protocol for this called the "CAN fragmentation protocol" or CFP for short. This allows data of much larger size to be transferred over the CAN bus.

Okay, but what if you want to transfer 1000 bytes, and the network maximum MTU is 256? Well, since CSP does not include streaming sockets, only packet's. Somebody will have to split that data up into chunks. It might be that you application have special knowledge about the datatype you are transmitting, and that it makes sense to split the 1000 byte content into 10 chunks of 100 byte status messages. This, application layer delimitation might be good if you have a situation with packet loss, because your receiver could still make good usage of the partially delivered chunks.

But, what if you just want 1000 bytes transmitted, and you don't care about the fragmentation unit, and also don't want the hassle of writing the fragmentation code yourself? - In this case, libcsp now features a new (still experimental) feature called SFP (small fragmentation protocol) designed to work on the application layer. For this purpose you will not use csp_send and csp_recv, but csp_sfp_send and csp_sfp_recv. This will split your data into chunks of a certain size, enumerate them and transfer over a given connection. If a chunk is missing the SFP client will abort the reception, because SFP does not provide retransmission. If you wish to also have retransmission and orderly delivery you will have to open an RDP connection and send your SFP message to that connection.

Client and server example

The following examples show the initialization of the protocol stack and examples of client/server code.

Initialization Sequence This code initializes the CSP buffer system, device drivers and router core. The example uses the CAN interface function csp_can_tx but the initialization is similar for other interfaces. The loopback interface does not require any explicit initialization.

```
#include <csp/csp.h>
#include <csp/interfaces/csp_if_can.h>

```

```
/* CAN configuration struct for SocketCAN interface "can0" */
struct csp_can_config can_conf = {.ifc = "can0"};

/* Init buffer system with 10 packets of maximum 320 bytes each */
csp_buffer_init(10, 320);

/* Init CSP with address 1 */
csp_init(1);

/* Init the CAN interface with hardware filtering */
csp_can_init(CSP_CAN_MASKED, &can_conf)

/* Setup default route to CAN interface */
csp_route_set(CSP_DEFAULT_ROUTE, &csp_can_tx, CSP_HOST_MAC);

/* Start router task with 500 word stack, OS task priority 1 */
csp_route_start_task(500, 1);
```

Server This example shows how to create a server task that listens for incoming connections. CSP should be initialized before starting this task. Note the use of *csp_service_handler()* as the default branch in the port switch case. The service handler will automatically reply to ICMP-like requests, such as pings and buffer status requests.

```
void csp_task(void *parameters) {
    /* Create socket without any socket options */
    csp_socket_t *sock = csp_socket(CSP_SO_NONE);

    /* Bind all ports to socket */
    csp_bind(sock, CSP_ANY);

    /* Create 10 connections backlog queue */
    csp_listen(sock, 10);

    /* Pointer to current connection and packet */
    csp_conn_t *conn;
    csp_packet_t *packet;

    /* Process incoming connections */
    while (1) {
        /* Wait for connection, 10000 ms timeout */
        if ((conn = csp_accept(sock, 10000)) == NULL)
            continue;

        /* Read packets. Timout is 1000 ms */
        while ((packet = csp_read(conn, 1000)) != NULL) {
            switch (csp_conn_dport(conn)) {
                case MY_PORT:
                    /* Process packet here */
                default:
                    /* Let the service handler reply pings, buffer use, etc. */
                    csp_service_handler(conn, packet);
                    break;
            }
        }

        /* Close current connection, and handle next */
        csp_close(conn);
    }
}
```

Client This example shows how to allocate a packet buffer, connect to another host and send the packet. CSP should be initialized before calling this function. RDP, XTEA, HMAC and CRC checksums can be enabled per connection, by setting the connection option to a bitwise OR of any combination of *CSP_O_RDP*, *CSP_O_XTEA*, *CSP_O_HMAC* and *CSP_O_CRC*.

```
int send_packet(void) {

    /* Get packet buffer for data */
    csp_packet_t *packet = csp_buffer_get(data_size);
    if (packet == NULL) {
        /* Could not get buffer element */
        printf("Failed to get buffer element\\n");
        return -1;
    }

    /* Connect to host HOST, port PORT with regular UDP-like protocol and 1000 ms timeout
↪*/
    csp_conn_t *conn = csp_connect(CSP_PRIO_NORM, HOST, PORT, 1000, CSP_O_NONE);
    if (conn == NULL) {
        /* Connect failed */
        printf("Connection failed\\n");
        /* Remember to free packet buffer */
        csp_buffer_free(packet);
        return -1;
    }

    /* Copy message to packet */
    char *msg = "HELLO";
    strcpy(packet->data, msg);

    /* Set packet length */
    packet->length = strlen(msg);

    /* Send packet */
    if (!csp_send(conn, packet, 1000)) {
        /* Send failed */
        printf("Send failed\\n");
        csp_buffer_free(packet);
    }

    /* Close connection */
    csp_close(conn);

    return 0
}
```

3.3 Flight Planner (libfp_client)

3.3.1 Introduction

The Flight Planner Client Library contains the *client* interface for the *server* components in the Flight Planner Library.

3.3.2 API

client

```
// Flight Planner Client example

#include <gs/fp/fp_client.h>

gs_error_t creating_an_fp_entry()
{
    const char* name = "ex1";
    const char* command = "cp /flash/file /flash/anotherfile";
    uint16_t repeat = 1;
    fp_timer_state_t s = FP_TIME_DORMANT;
    fp_timer_basis_t b = FP_TIME_RELATIVE;
    gs_timestamp_t t;
    t.tv_sec = 10;
    t.tv_nsec = 0;

    /* creates a dormant fp entry that will execute 10 seconds
     * after being activated */
    if (fp_client_timer_create(name, command, b, s, &t, repeat) < 0) {
        return GS_ERROR_UNKNOWN;
    }

    /* ... */

    /* activates the dormant fp entry */
    return fp_client_timer_set_active(name) < 0 ? GS_ERROR_UNKNOWN : GS_OK;
} //
```

3.3.3 Commands

fp

A number of Commands are available for creating and editing Flight Planner entries, all placed in the *fp* command group.

Table 3.1: Command Table ‘fp’

Command	
fp server	Setup CSP address of FP server. Arguments: `<server>`: CSP address.
fp flush	Flush current flight plan.
fp load	Load flight plan from file. Arguments: `<path>`: Local file to load flight plan from.
fp store	Store current flight plan to file. Arguments: `<path>`: Local file to save current flight plan to.
fp create	Create new timer. Arguments: `<timer>`: Name of the timer. `<[+]sec>`: Absolute or relative time to execute the command. Prefix with ‘+’ for relative time. `<command>`: Command to execute. `[repeat]`: Number of times to repeat the command execution - only valid for relative timers. `[state]`: Initial state of the timer, either ‘active’ or ‘dormant’.

Continued on next page

Table 3.1 – continued from previous page

Command	
fp delete	Delete timer. Arguments: `<timer>`: Timer name.
fp active	Set timer active. Arguments: `<timer>`: Timer name.
fp allactive	Set all timers active. Arguments: `<timer>`: Timer name.
fp dormant	Set timer dormant. Arguments: `<timer>`: Timer name.
fp repeat	Set timer repeat. Arguments: `<timer>`: Timer name. `<count>`: Number of repeats.
fp time	Set execution time. Arguments: `<timer>`: Timer name. `<[+]sec>`: Absolute or relative time to execute the command. Prefix with '+' for relative time.
fp list	List timers.

To interface with a remote Flight Planner Server, you first need to specify its CSP address using the *fp server* The commands will then use this when connecting to the server.

In the example below, we connect to the server and create a new Flight Planner entry to capture an image. The entry is configured to execute 10 seconds after being created, as specified with the *+10* argument. To add an entry with an absolute execution time, specify the time as a UNIX timestamp. We omit the repeat and state arguments since we just want the entry to run once:

```
csp-term # fp server 1
csp-term # fp list
No timers in list
csp-term # fp create
usage: create <name> [+]<sec> <command> [repeat] [state]
csp-term # fp create snap +10 "cam snap -sa"
csp-term # fp list
Timer     Act Basis When       Repeat Remain Event
snap       Y   Rel   10         1      1      cam snap -sa
<10 seconds pass and snap command is executed>
csp-term # fp list
Timer     Act Basis When       Repeat Remain Event
snap       N   Rel   10         1      0      cam snap -sa
csp-term # fp active snap
csp-term # fp list
Timer     Act Basis When       Repeat Remain Event
snap       Y   Rel   10         1      1      cam snap -sa
<10 seconds pass and snap command is executed>
csp-term # fp delete snap
csp-term # fp list
No timers in list
```

The *fp list* command can be used to list the current entries. In the example above, the entry is marked as *active* with the remaining number of executions equal to 1 (`Act = Y, Remain = 1`). After being executed, a command can be activated again using *fp active* or deleted using *fp delete*.

Using the *fp store* and *fp load* commands, it is possible to store and load Flight Planner entries to the file system. Entries are stored as lines of ASCII text in the following comma-separated format:

```
name,command,state,basis,last_sec,last_nsec,when_sec,when_nsec,repeat
```

The *name* and *command* fields are just ASCII strings containing the name and command to execute of the entry. *state* specifies whether the entry is active (0) or dormant (1) and the *basis* field is 0 for absolute timers and 1 for relative. The *last_sec/nsec* and *when_sec/nsec* contain the last execution time (ignored for absolute timers) and the execution time of the entry. So an active *cam snap -sa* command named *snap* scheduled for execution on UNIX time 1500000000 would look like:

```
snap,cam snap -sa,0,0,0,0,1500000000,0,1
```

3.4 FTP (libftp_client)

3.4.1 Introduction

The FTP (client) library contains the *client* interface for the *server* components in the FTP library.

3.4.2 Commands

ftp

The FTP (client) library provides commands for interfacing with the *server* (from FTP library). The commands are grouped under *ftp*.

```
csp-client # ftp
client: File Transfer Protocol
  ls                    list files
  rm                    rm files
  mkfs                  make file system
  mkdir                 make directory in file system
  rmdir                 remove a directory from the file system
  mv                    move files
  cp                    copy files
  zip                   zip file
  unzip                 unzip file
  local_zip             zip local file
  local_unzip           unzip local file
  server                set server, chunk size and mode
  upload                Upload url
  download              Download url
  timeout               Set general ftp timeout
```

To interface with a remote FTP server, you first need to specify the CSP address of the remote node:

```
csp-client # ftp server 1
server 1 (port 9), chunk size 185 bytes
```

Once set, the following FTP commands will use these settings until changed. Now *upload* a file to a remote node:

```
csp-client # ftp upload nanomind_ram.bin /flash/nanomind_ram.bin
File size is 472252
Checksum: 0xbdfaa1dd
Transfer Status: 0 of 2553 (0.00%)
100.0% [########################################] (2553/2553)
CRC Remote: 0xbdfaa1dd, Local: 0xbdfaa1dd
```

and see if its really there:

```
csp-client # ftp ls /flash
461.2K nanomind_ram.bin
```

3.5 GOSH (libgosh_client)

3.5.1 Introduction

The GOSH (client) library contains the *client* interface for the *server* components in the GOSH library.

Note: The GOSH library is only included in the Mission Library version of the Command and Management SDK.

3.5.2 Commands

G-script

Commands are added to your platform by calling the C-function `gs_gscript_register_commands(void)`.

Commands are grouped under *gscript*.

```
nanomind # gscript
gosh: gosh: G-script
  run_shell            Run gscript written in shell
  run                  Run gscript from file
  stop                 Stop all gscript(s) on the node
  server               Set gscript server
  run_now              Run single <command> on <node> now
```

Remote Shell

Commands are added to your platform by calling the C-function `gs_gosh_remote_register_commands(void)`.

Commands are grouped under *shell*.

```
nanomind # shell
gosh: Remote shell
  node                 Set or display remote node
  run                  Run command or shell
```

3.6 GomSpace Sensor Bus (libgssb_client)

3.6.1 Introduction

GSSB is an abbrivation of GomSpace Sensor Bus. It a I^{2}C based bus, which connects different GomSpace sensors and GomSpace release devices. The group of sensors and releases devices are named GSSB devices. GomSpace Sensor Bus library provides both drivers for GSSB devices and a CSP service handler, which can be implemented on a CSP node, making it a proxy server for the GSSB devices connected to its I^{2}C bus, see Fig.3.1.

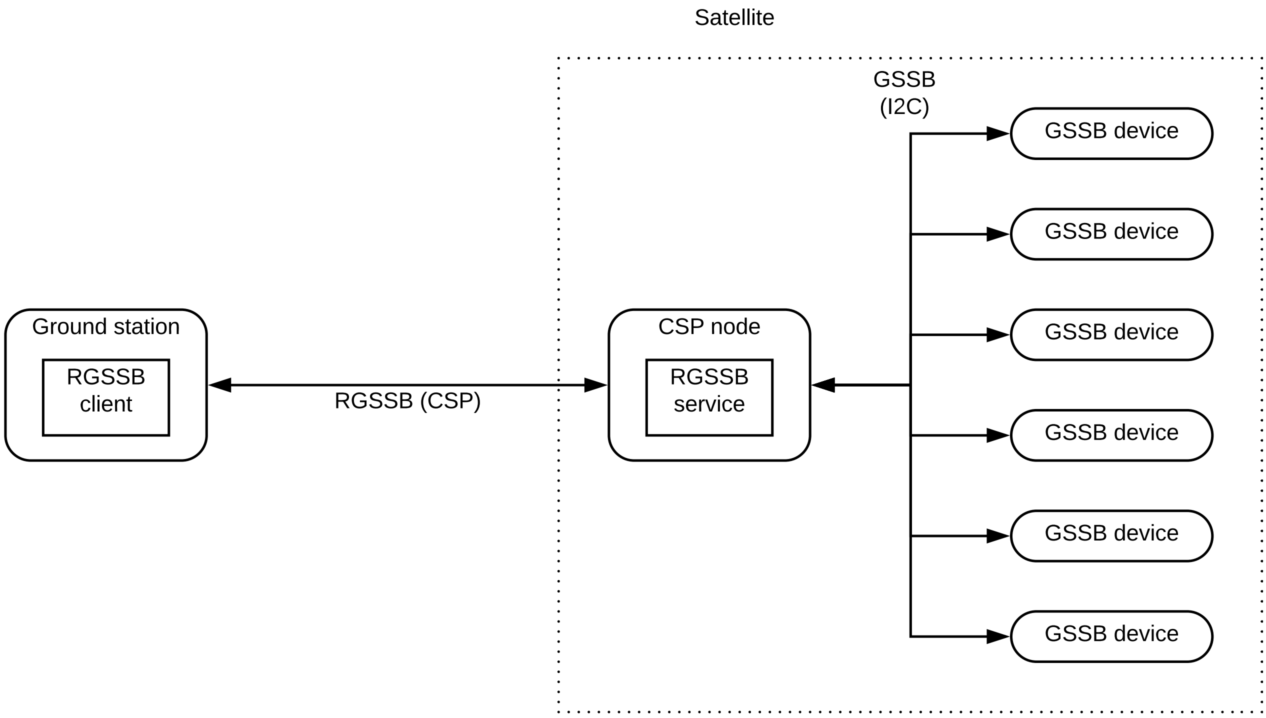


Fig. 3.1: GSSB network illustration.

Following is the list of GomSpace GSSB boards, which is a supported GSSB device:

- NanoCom ANT-6F UHF, refered to as AR6 throughout the library,
- NanoUtil AR6 and Nanoutil Interstage 3U, refered to as AR6 throughout the library,
- NanoUtil Interstage GSSB Starling, refered to as I4 throughout the library,
- NanoUtil Interstage GSSB, NanoUtil Top GSSB and NanoUtil GSSB, refered to as Interstage/Istage throughout the library,
- NanoPower GSSB MSP, refered to as MSP throughout the library,
- NanoSense Fine Sun Sensor, refered to as sunsensor throughout the library.

The library consists of generic GSSB functions, which will work on every GSSB device, such as changing address or retrieving the SW version, and more board specific functions, which are prefixed with the board type.

The library consists of to parts, libgssb and libgssb_client. The client consists of functions for both the GSSB protocol, for communication directly with GSSB devices, and functions for the RGSSB protocol, for communication with a proxy RGSSB service. The actual library libgssb holds API for the RGSSB service.

3.7 Housekeeping (libhk_client)

3.7.1 Introduction

The Housekeeping client library contains the *client* interface for the *server* components in the Housekeeping library.

3.7.2 API

client

Example of using the client API for requesting beacon packets from the Housekeeping Server.

```
// Housekeeping Client example

#include <gs/hk/client.h>

gs_error_t request_beacons()
{
    return gs_hk_request_beacons(1,      // CSP address of my housekeeping server
                                 3,      // Beacon type to request
                                 10,     // 10 second interval
                                 10,     // 10 samples
                                 GS_HK_REQUEST_BEACON_FIRST_SAMPLE_NOW,
                                 NULL); // Request packets not file
}
```

3.7.3 Commands

hk

The *hk* command group provides the following commands:

Table 3.2: Command Table 'hk'

Command	
hk server	Setup CSP address of HK server. Arguments: `<server>`: CSP address.
hk get	Request beacons from HK server. Arguments: `<type>`: Type(s) of beacon(s) to request, use comma-seperated list to request multiple types. `<interval>`: Time in seconds between beacons. `<samples>`: Number of samples/beacons to request. `[t0]`: Time (seconds since UNIX epoch) of the first/newest sample. Use 0 for 'now'. `[path]`: If set, beacons will be written to this path instead of tx'ed. `[protocol_version]`: If set, beacons will be requested using this protocol_version. Use 0 for legacy protocol. Defaults to current version
hk reload	Ask server to reload its configuration from files.

3.8 Nano Protobuf (libnanopb)

3.8.1 Introduction

The Nano Protobuf library provides the functionality to encode and decode protobuf messages in C using C-structs.

The library also contains IDL tools that can convert Protobuf files (.proto) to C source (.c/.h), Python files, and documentation in Markdown.

The Nano Protobuf library is entirely based on the open source nanopb library.

nanopb

Nanopb is a small code-size Protocol Buffers implementation in ansi C. It is especially suitable for use in microcontrollers, but fits any memory restricted system.

- Homepage: https://jpa.kapsi.fi/nanopb/
- Documentation: https://jpa.kapsi.fi/nanopb/docs/
- Source: https://github.com/nanopb/nanopb.git Tag: nanopb-0.3.9

License:

```
Copyright (c) 2011 Petteri Aimonen <jpa at nanopb.mail.kapsi.fi>

This software is provided 'as-is', without any express or
implied warranty. In no event will the authors be held liable
for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any
purpose, including commercial applications, and to alter it and
redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you
   must not claim that you wrote the original software. If you use
   this software in a product, an acknowledgment in the product
   documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and
   must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
   distribution.
```

3.9 Parameter System (libparam_client)

3.9.1 Introduction

The Parameter library (client) contains the *client* interface for the *server* components in the Parameter library.

3.9.2 Commands

rparam

The *rparam* commands can be used to modify the parameters on a remote system. The commands are grouped under *rparam* and provides following commands:

```
csp-client # rparam
Remote access to Parameter System
  init                    Set server and load table spec. from file
  download                Set server and download table spec.
  getall                  Download full table contents from server
  list                    Lists the table specification
  get                     Add a 'get' to the current query transaction
  set                     Add a 'set' to the current query transaction
  copy                    Copy table to table (version <= 3 only)
  load                    Load table
  save                    Save table
  reset                   Reset the current query
```

```
  send                    Send the current query
  wd                      Set working directory for init/download
  timeout                 Set timeout
  checksum                Set checksum
  autosend                Enable/disable autosend for set and get queries
```

The *rparam* client must first be initialized by downloading the table (specification) in question, from the remote node:

```
csp-client # rparam download 1 0
```

Now that the *rparam* client knows which parameters exists on the remote system, it is possible to request the actual table data:

```
csp-client # rparam getall
Downloading table content for table 0 from server 1
  0x0000 uid              STR ""
  0x0010 type             U8  0
  0x0011 rev              U8  0
  0x0012 dock_type        STR ""
  0x0022 csp_addr         U8  1
  0x0023 csp_rtable       STR ""
  0x0083 gosh_en          BL  true
  0x0084 gosh_uart        U8  2
  0x0085 can_en           BL  true
  0x0088 can_brate        U32 1000000
  0x008C i2c_en           BL  true
  0x0090 i2c_brate        U32 400000
  0x0094 kiss_en          BL  false
  0x0098 kiss_brate       U32 500000
  0x009C kiss_uart        U8  4
```

We can now modify one or more parameters on the remote node, using the *set* command.

```
csp-client # rparam set uid "New UID"
csp-client # rparam set type 2
csp-client # rparam send
```

autosend is off by default, causing the *set* commands to be queried in the client, until *rparam send* is performed. By enabling *autosend*, the *set* or *get* commands are sent immediately:

```
csp-client # rparam autosend 1
auto send: 1
csp-client # rparam get uid
  0x0000 uid              STR "New UID"
```

The query function is very useful when changing both the uplink and downlink baudrate of a radio. By putting multiple *set* into one request, ensures that the system will never do a partial parameter update.

3.10 Remote GOSH (librgosh_client)

3.10.1 Introduction

Every GomSpace product comes with GOSH (GOmspace SHell). GOSH provides a simple text based command interface and is usually accessed through a direct serial connection.

Remote GOSH enables a user to access the GOSH commands interface through a CSP connection instead of a direct serial connection. This enables a user to directly run commands and get returned results over the normal CSP network via CAN, SPI, UHF/VHF, etc.

By means of Remote GOSH (client) library the user is able to access all the provided GOSH commands of the product and use those in cases where dedicated client libraries are not available.

An overview of components involved in a Remote GOSH application is shown in *Using Remote GOSH for commands execution & feedback*.

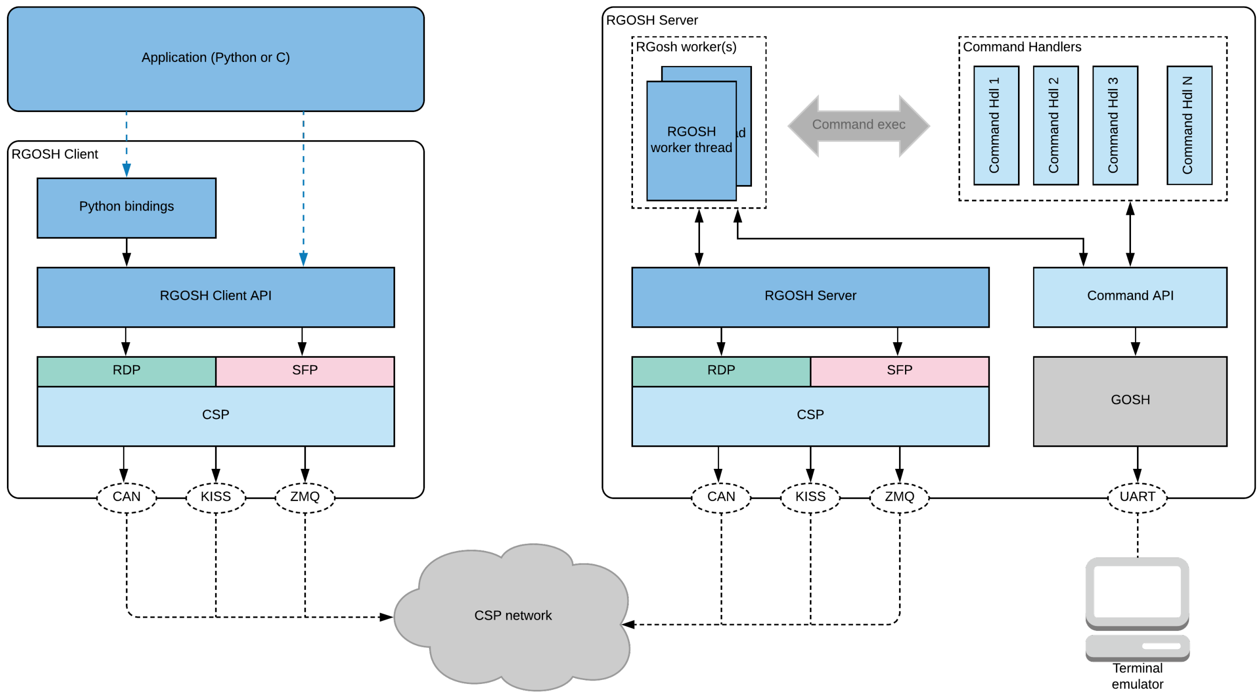


Fig. 3.2: Using Remote GOSH for commands execution & feedback

Communication between a Remote GOSH client application and a Remote GOSH server is described in the *Sequence diagram showing the use of Remote GOSH*.

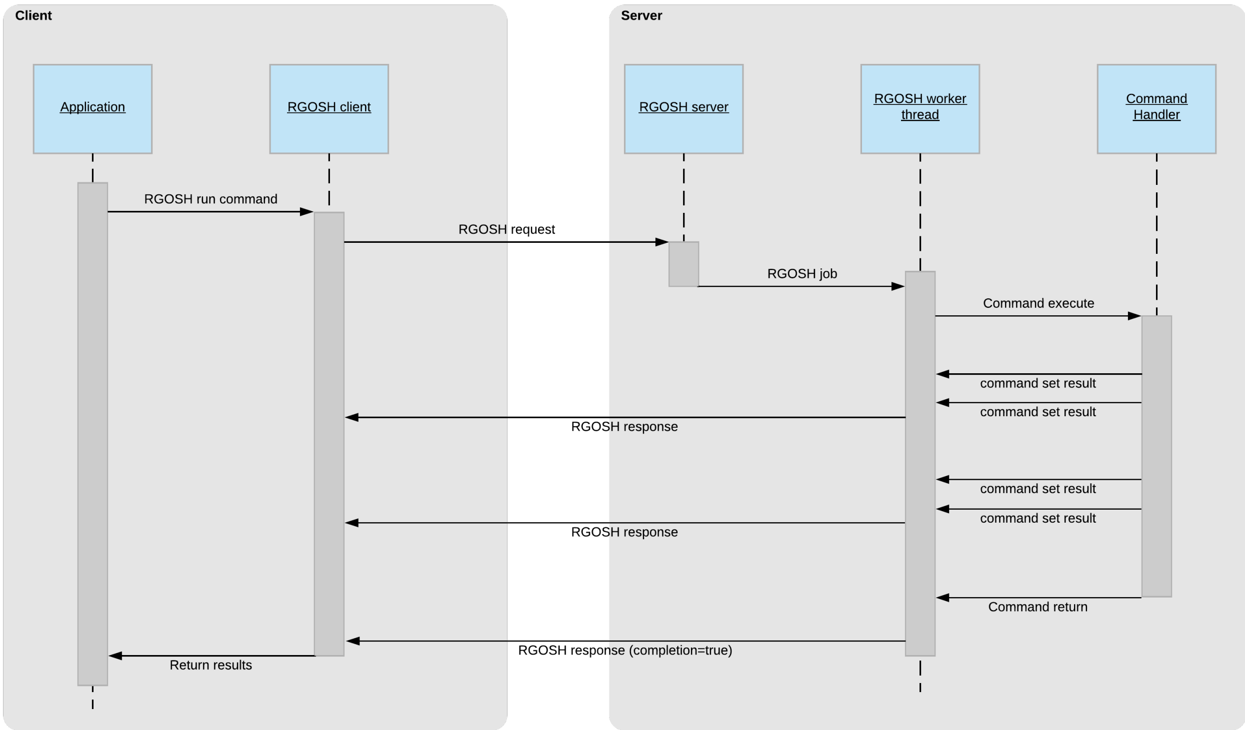


Fig. 3.3: Sequence diagram showing the use of Remote GOSH.

3.10.2 Python example

Below is an example that shows how to use the python bindings from the Remote GOSH client library to create a simple Python application.

```
#!/usr/bin/python
# Copyright (c) 2013-2018 GomSpace A/S. All rights reserved.

# bootstrap to buildtools
# ./gsbuildtools.py clone
# build with: ./waf configure build --enable-bindings --enable-crc32
#    --enable-rdp --enable-if-zmq --enable-if-can --enable-can-socketcan --enable-xtea --
↪enable-if-kiss
# from repo root with: LD_LIBRARY_PATH=build:build/lib/libutil:build/lib/libgscsp:
#                                      build/lib/libgscsp/lib/libcsp:build/lib/libnanopb
#    python tst/py_example/bindings_example.py

import os
import sys
import time
import pprint

sys.path.append("build")
sys.path.append("build/lib/libgscsp")
sys.path.append("build/lib/libgscsp/lib/libcsp")
sys.path.append("build/lib/libnanopb")
sys.path.append("python")


if sys.version_info >= (3, 0):
    # from libgsrgosh_client_py3 import *
```

```
    import libgsrgosh_client_py3 as rgosh
    import libcsp_py3 as csp
else:
    # from libgsrgosh_client_py2 import *
    import libgsrgosh_client_py2 as rgosh
    import libcsp_py2 as csp


TST_RGOSH_LOCAL_CSP_NODE = 5
TST_RGOSH_LOCAL_CSP_PORT = 12

TST_RGOSH_REMOTE_CSP_NODE = 10

if __name__ == "__main__":
    csp.buffer_init(400, 512)
    csp.init(TST_RGOSH_LOCAL_CSP_NODE)
    err = csp.zmqhub_init(TST_RGOSH_LOCAL_CSP_NODE, "localhost")
    if err != 0:
        print "Failed setting up ZMQ interface"

    err = csp.rtable_set(32, 5, "ZMQHUB", 255)
    if err != 0:
        print "Failed setting up Route table"

    err = csp.route_start_task()
    if err != 0:
        print "Failed starting router task"

    # wait for routing task to start
    time.sleep(1)

    print "Setup done - Ready to start......."

    ret = csp.ping(TST_RGOSH_REMOTE_CSP_NODE)
    if ret == -1:
        print "Failed to ping remote node!"

    print "Sending RGOSH command...."
    [err, out, res] = rgosh.run_command(TST_RGOSH_REMOTE_CSP_NODE, TST_RGOSH_LOCAL_CSP_
↪PORT, "clock")
    print "RGOSH Run Command returned: <" + str(err) + ">"
    print "RGOSH Run Command output: <" + out + ">"
    print "RGOSH Run Command results:\n" + pprint.pformat(res, 1)
```

3.10.3 “C” example

Below is an example that shows how to use the Remote GOSH client library to create a simple RGOSH client “C” application.

```
/* Copyright (c) 2013-2018 GomSpace A/S. All rights reserved. */

/**
   This example Remote GOSH client application connects to a Remote GOSH server on CSP␣
↪node addr. 10.

   The application can connect over CSP KISS, CAN or ZMQ. Run with '-h' option for more␣
↪info.
*/

#include <stdlib.h>
```

```
#include <string.h>

#include <gs/rgosh/rgosh.h>
#include <gs/util/test/cmocka.h>
#include <gs/util/log.h>
#include <gs/util/log/appender/appender.h>
#include <gs/util/time.h>
#include <gs/util/linux/command_line.h>
#include <gs/csp/linux/command_line.h>
#include <gs/csp/csp.h>
#include <gs/csp/log.h>
#include <gs/csp/port.h>
#include <gs/csp/router.h>
#include <csp/interfaces/csp_if_zmqhub.h>

#define RGOSH_CLIENT_NODE  1    /** CSP address of this node */
#define RGOSH_REMOTE_NODE  10   /** CSP address of the remote node */

static void initialize_csp()
{
    gs_csp_conf_t conf;
    gs_csp_conf_get_defaults_server(&conf);
    gs_csp_init(&conf);

    /* Load command line routing table (if set) or load default route table */
    gs_csp_rtable_load("", true, true);

    /* Start CSP router task */
    gs_csp_router_task_start(16384, GS_THREAD_PRIORITY_NORMAL);
}


static void _command_result_cb(void* ctx, const char *group, const char *key, const char␣
↪*value)
{
    printf("    Group: <%s>, Key: <%s>, Value: <%s>\n", group, key, value);
}

static void _command_stdout_cb(void* ctx, const char *out)
{
    printf("    Stdout: <%s>\n", out);
}

static gs_rgosh_run_if_t gs_rgosh_cb = {
    .stdout_cb = _command_stdout_cb,
    .result_cb = _command_result_cb,
};

int main(int argc, char *argv[])
{
    /* Initialize console logger */
    gs_log_init(true);

    const struct argp_child argp_children[] = {
        gs_help_command_line_argp,
        gs_csp_command_line_options,
        {0},
    };
    const struct argp argp = {
        .doc = "RGOSH C-example on Linux", .children = argp_children,
    };

    // Parse options/arguments - will exit on failure
```

```
    gs_argp_parse(&argp, argc, argv, 0, 0, "1.0.0");

    initialize_csp();

    gs_time_sleep_ms(1000); // Wait for the ZMQ & router threads to be running...

    gs_error_t ret;

    log_info("Running \"clock\" command on remote node");
    gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, "clock", 3000, &ret, &gs_
→rgosh_cb, NULL);

    log_info("Running \"log group list\" command on remote node");
    gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, "log group list", 3000, &
→ret, &gs_rgosh_cb, NULL);

    log_info("Running \"log appender list\" command on remote node");
    gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, "log appender list", 3000,␣
→&ret, &gs_rgosh_cb, NULL);

    log_info("Running \"log group list *\" command on remote node");
    gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, "log group list *", 3000, &
→ret, &gs_rgosh_cb, NULL);

    return 0;
}
```

3.11 Utility (libutil)

3.11.1 Introduction

The Utility library provides cross-platform API's for common functionality, for use in both embedded systems and standard PC's running Linux.

For some API's, the Utility library doesn't contain the actual implementation, but only the prototypes. The implementation for specific platforms can typically be found in the Embed library.

Features

- Time
  - Time get and set
  - Delay and sleep
  - Time conversions
  - Timestamp formatting and parsing
  - Real Time Clock (RTC) interface
- Threading
  - Queue
  - Mutex
  - Thread
  - Semaphore
  - Software watchdog

- String
  - Number to string
  - Other string parsing and creation
- Checksum/Hash
  - Fletcher16
  - CRC8
  - CRC32
- Command (GOSH)
  - Console
- Logging
  - Groups
  - Appenders
- Drivers
  - SPI
  - I2C
  - CAN
  - GPIO
- Zip file creation and extraction

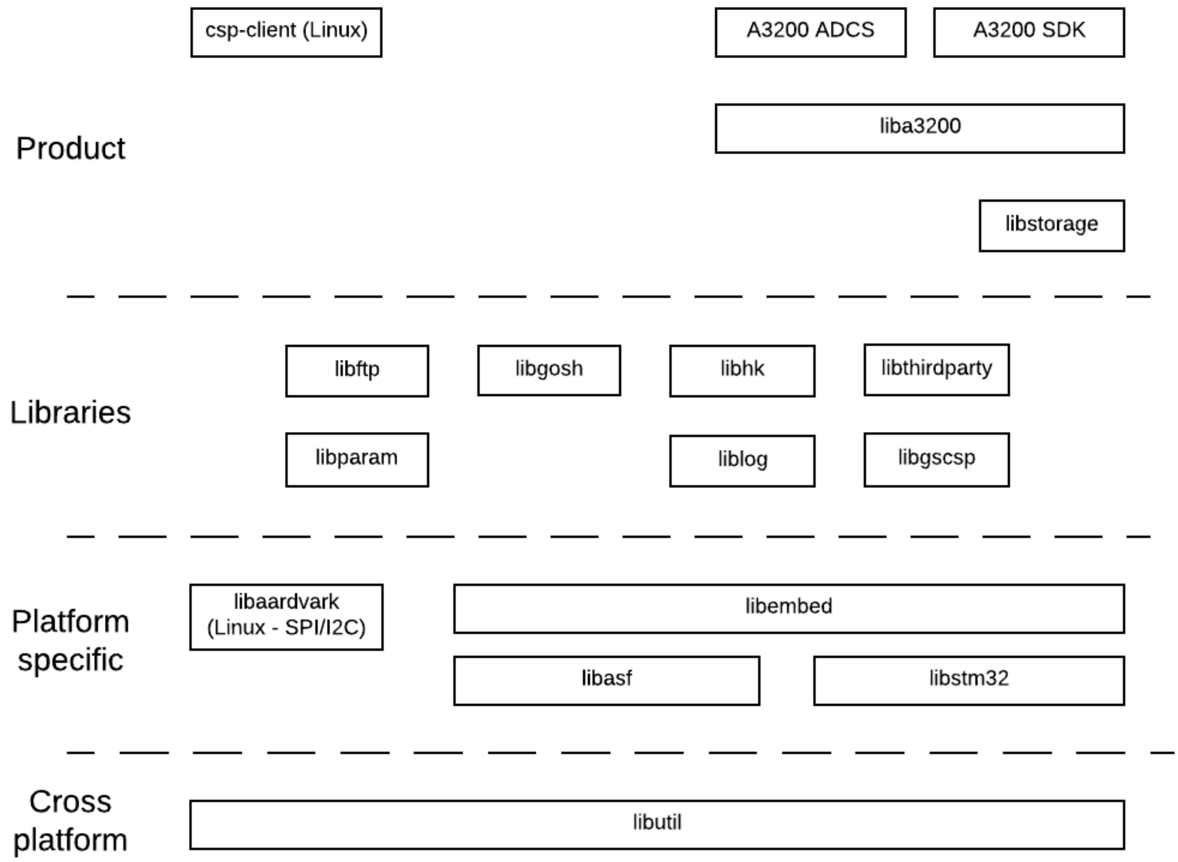
3.11.2 Software concepts and terminology

This section provides a general introduction to common concepts and terminology used in documentation and software nodules. This will make it easier to understand and navigate the software SDK's.

- *Software Stack*
- *Client / Server*
- *Directory Structure*

Software Stack

The Software Stack consists of several modules divided into four layers. The four layers and a selection of the modules are shown on the image below and described in the following sections.



Cross Platform Layer The Cross Platform Layer contains primarily the Utility library, which is used by nearly all products and libraries. The Utility library provides cross-platform API's for common functionality, for use in both embedded systems and standard PC's running Linux. For some API's, the Utility library doesn't contain the actual implementation, but only the prototypes. The implementation for specific platforms can be found on higher layers, e.g. Embed library, Aardvark library.

The primary purpose of using cross-platform code, is to re-use/run the same code on multiple platforms - and especially support (unit) testing of embedded code, on a more standard PC/Linux platform.

Platform Specific Layer The Platform Specific Layer defines APIs that are not cross platform. All APIs that can be defined across multiple embedded platforms are normally defined in the Embed library, which also holds the implementation of some of the API's defined in the Utility library.

The vendor specific libraries ASF and STM32 are almost *as is*, with minimum GomSpace modifications/fixes. All GomSpace modifactions are mainly done in the Embed library. Both libraries comes with a FreeRTOS version.

The Platform Specific Layer also contain specific Linux implementations, e.g. Aardvark library which provices driver support for SPI and I^2C - providing the same API, used on an embedded platform. This allow tests of drivers for components with a SPI or I^2C interface on Linux.

Libraries Layer The Libraries Layer contains services and high-level components. These include Parameter System, Housekeeping, CSP, etc.

Product Layer The top layer is the Product Layer. The modules in the Product Layer define and implement functionality that is targeted at a specific product. This could be an application running on Linux (e.g. csp-client) or the software running on an A3200 platform (e.g. A3200 ADCS / SDK).

Client / Server

Some components (e.g. *libftp*) are split into 2 modules: a *server* (or *host*) and a *client* module. The *server* module provides the backend functionality, e.g. FTP server. The *client* module typically provides an API and/or a set of commands for interfacing with the *server*.

The name of the *client* module is normamly the *server* name, suffixed with *_client*.

Directory Structure

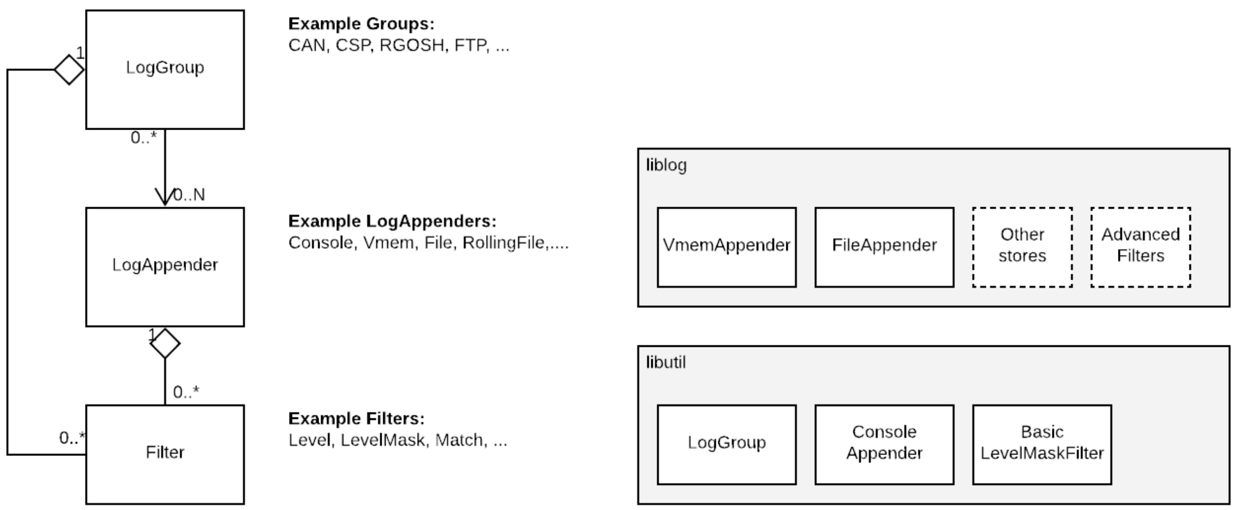
Modules are structured in the same way, which makes it easy to find the relevant files.

Folder	Description
include	Public header files.
include/gs/<module>	To avoid clashes with other modules and 3rd part libraries, all files are placed in scoped sub-folders.
include/gs/<module>/internal	The *internal* folder is reserved for shared files between *server* & *client*, and only present in the *client*. These files should never be included.
conf	Configuration, e.g. header files, linker scripts.
src	Source code, the actual implementation. These files are considered private for the module and typically organized to reflect the internal sub-modules. NOTE this folder is only present, if the module is shipped with source code.
doc	Documentation, PDF and/or HTML. This folder is normally only present in SDK's.
lib	If the module is a *server* / *client* module, this folder will contain the *client*. The structure of the *client* follows the same standard structure.
obj	Pre-compiled object files. NOTE this folder is only present, if the module is shipped as *binary* (no source code).

3.11.3 Logging framework

The Gomspace logging framework provides the ability to log to different destinations and in different domains/groups. The logging framework design is modeled after the *log4xx* design (for reference see: log4j 2.x architecture), with a number of simplifications in order to address small embedded system. For example is the logger hierarchy concept has been simplified, and formatting can only be controlled on *appender* level.

A high-level design of the logging framework is shown below.



The LogGroup provides the logging functionalities for the individual log domains/groups in the product - e.g CSP, Command, FTP, ... Each product can implement any number of logging groups. Each logging group support one filter type at any given time. Currently the only filter supported is a level mask filter.

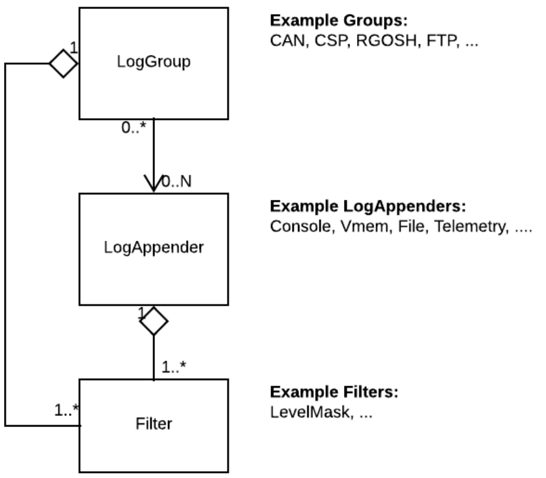
Each LogGroup can reference any number of LogAppenders (stores). The LogAppenders is responsible for writing the actual log statement to the "store". The LogAppenders can also choose to filter the log statement depending on the associated filter. Currently the only supported filter of the LogAppender is a level mask filter.

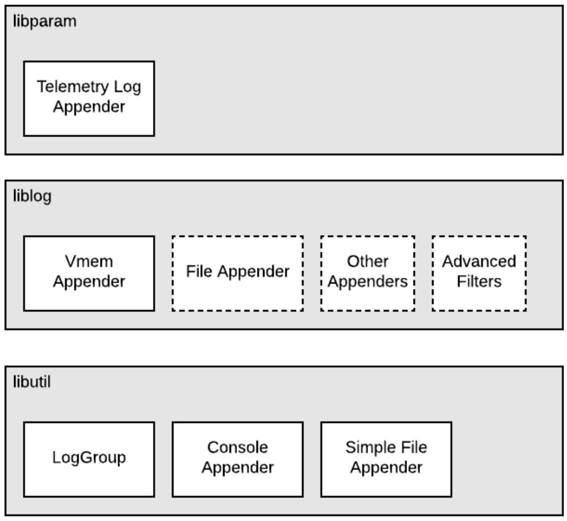
Logging hierarchy

There are use-cases in the logging system where a hierarchy of LogAppenders are beneficial. For instance all logging groups could log to the same overall "log telemetry" appender. The log telemetry appender will gather statistics of for instance error and warnings for each group. This will enable you to gather information on the current (health) status of the system. It is important that all groups logs to this telemetry LogAppender, otherwise it will not provide a true system status. A Logger Hierarchy will enable you to have LogAppenders that are default added to all groups whereas other LogAppenders will only be applied to certain log groups.

An example of the LogGroup hierarchy is shown below. Each of the log groups is connected to a "Root" logger. This concept is identical to the *log4xx* concept. There is however only support for one level in the hierarchy of the GomSpace logging system. The Root logger group is as such similar to any other log group in the system, where a number of appenders can be attached. The root logger group will always be present in the system.

The appenders attached to the root logger will always be called, so this way we can ensure that certain appenders are always called for all log groups. For each of the log groups a number of specific appenders can be added. These appenders referenced by the different log groups will be called - in addition to the appenders at the root.





The above figure shows three categories of LogAppenders:

- Obligatory LogAppenders (attached to root logger)
  - Console logger
  - Telemetry bit mask (error/warning bitmask)
- Default LogAppenders (in this example attached to a number of groups, normally it will be attached to the root logger)
  - FRAM Appender
- Customer Appenders (attached to one or more groups)
  - Audit log for commands
  - File store dedicated to param
  - Other..

Logging example application

Below is a simple application that shows an example of how to configure the logging framework for logging all to console, and executed commands to a file.

```
/* Copyright (c) 2013-2018 GomSpace A/S. All rights reserved. */

#include <gs/util/gosh/command.h>
#include <gs/util/gosh/console.h>
#include <gs/util/log/log.h>
#include <gs/util/log/appender/simple_file.h>
#include <gs/util/time.h>
#include <gs/util/thread.h>

static const gs_log_appender_simple_file_config_t simple_file_conf = {
    .filename = "logfile.txt",
    .truncate = true,
    .use_local_time = false,
};

static gs_log_appender_t simple_file_appender = {
    .name = "logfile",
```

```
    .drv   = &gs_log_appender_simple_file_driver,
    .drv_config = &simple_file_conf,
    .drv_data = NULL,
    .mask = LOG_ERROR_MASK | LOG_WARNING_MASK | LOG_INFO_MASK,
};

int main(void)
{
    /* Initialize log system and enable log to console */
    gs_log_init(true);

    printf("Welcome to the Util-app test application!\r\n");

    // Initialize command framework
    gs_command_init(0);

    // Register logger for logning all executed commands to the group 'command'
    gs_command_register_logger(gs_command_logger_default, NULL);
    gs_log_group_set_level_mask("command", LOG_ERROR_MASK | LOG_WARNING_MASK | LOG_INFO_
↪MASK);

    // Add simple file log-appender on the 'command' gruop */
    gs_log_appender_add(&simple_file_appender, 1);
    gs_log_group_register_appender("command", simple_file_appender.name);

    // Start console
    gs_console_start("util.app", 0);

    // Block forever
    gs_thread_block() ;

    return 0;
}
```

3.11.4 Software Watchdog

The purpose of the software watchdog is to act as a layer between the actual hardware watchdog and the different clients, where a client can be either a thread, task, API, communcation channel - basically anything that needs supervision.

The software watchdog is divided in a software watchdog client API and a server. Multiple client instances can exist at any time, whereas only one server is available at any given time.

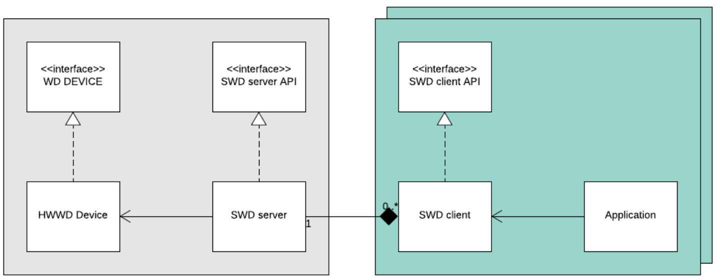


Fig. 3.4: Software Watchdog Client/Server modules

The software watchdog support dynamic registry of client instances. When a client is registered/de-registered it is allocated/released from a pre-allocated pool, to avoid run-time dynamic memory allocation. The size of this pool is determined during initialization of the watchdog API.

The SWD client interface supports the following features:

- Touch: Touch the client instance, and prevent the voerall watchdog from timing out and resetting the system.
- Timeout: The maximum time between a client touches the watchdog.
- Dynamic register/de-register of clients.
- Callback (optional): called when the client times out. Can be used for gathering data related to the missing *touch*.

The SWD server interface supports the following features:

- Passive: no active threads, check() must be called manually to verify clients and prevent system from resetting.
- Active: A local thread performes check() at intervals.
- SWD server owns the HWD: Only the SWD server should service the HWWD. This means that all services requiring watchdog functionality should utilize the SWD client API.

An example sequence diagram is shown below of how the framework is implemented and intended to be used.

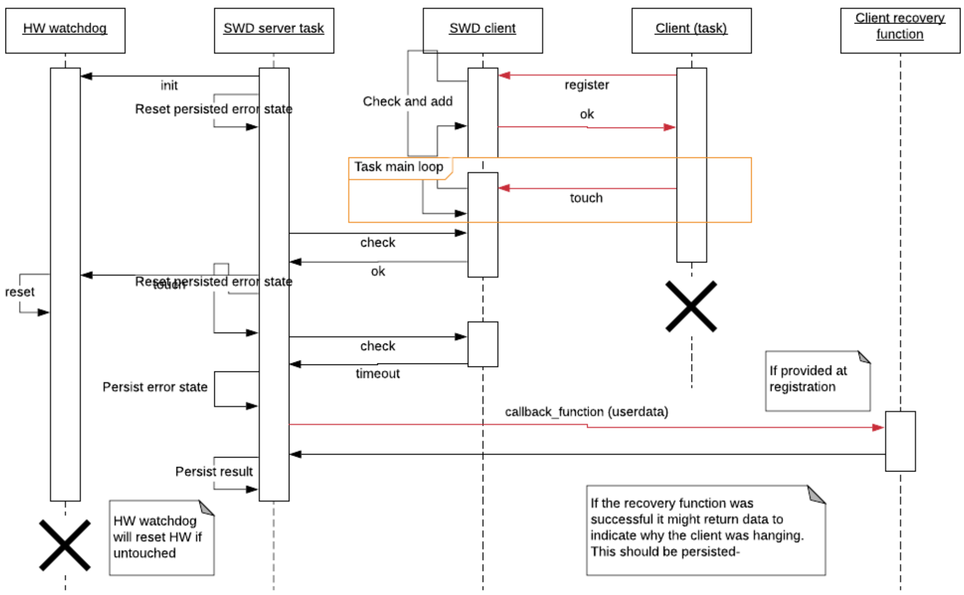


Fig. 3.5: Software Watchdog Client/Server sequence diagram

3.11.5 Command (GOSH)

The *command* interface provides a simple text based interface, mapping textual commands (i.e. command and arguments) to a handler (C function).

In daily terms, the command interace is often refered to as GOSH (GOmspace SHell). GOSH is a combination of the *Console* and the command internface. But commands can also be executed from other sub-systems such as g-script, Flight Planner, Health System.

The command/GOSH interface is normally available on all GomSpace products, and primarily used for configuration and debugging.

Console

The console interface reads from an input stream (e.g. stdin), and forwards the input the command interface for execution. In this case, output from the command execution will be written to an output stream (e.g. stdout). stdin/stdou is typically an UART port on the product.

Any user who have tried a text-interface before, should feel right at home.

The console is supported on Linux and FreeRTOS. The console assumes a standard VT102 terminal emulator, but provides a few fixes for the quirks in 'minicom' (application). The console prompt is controlled by the application. Here is an example prompt:

Thie console behaves like a normal unix shell, where entire commands (and arguments) are written as strings and executed when <enter> is pushed.

The console uses a traditional keyboard shortcut layout for navigating history, line editing and includes tab completion. The complete list of command shortcuts are:

Table 3.3: console shortcut keys

Key	Description
<ctrl><a>	go to beginning of line
<ctrl><b> or <left>	go back a char
<ctrl><d>	delete char to the right of cursor
<ctrl><e>	go to end of line
<ctrl><f> or <right>	go forward a char
<ctrl><h> or <backspace>	backspace
<ctrl><k>	kill rest of line line
<ctrl><l>	clear terminal
<ctrl><n> or <up>	next in history
<ctrl><p> or <dn>	prev in history
<ctrl><t>	transpose chars
<ctrl><u>	kill line from beginning
<enter>	execute command
<tab>	try and complete command

Command Parser

The command parser splits a text string into a command and arguments, using space as delimter. It searches for tl

- *root commands* are at the top of the hierachy and can be seen by typing help, or pressing <tab> on an empty prompt.
- *sub commands* are located below *root commands* or other *sub commands*.

root commands are usually more a grouping, than an actual command. The framework comes with a set of 'built in commands:

vmem is an example of a command group:

Commands that does not have sub-command(s), has an associated *Command handler*. A handler is a function that will be called with the specified arguments, when the command is executed. It also has a help and usage text. The help text is shown in the help output as above, and the usage text is shown when completing the command using <tab>:

Defining commands

In order to define a new *root command*, define a command structure in one of your c-files like this:

```
static const gs_command_t GS_COMMAND_ROOT cmd_default[] = {
    {
        .name = "help",
        .help = "Show help",
        .usage = "[command[ subcommand[ arg ...]]]",
        .handler = cmd_help,
        .optional_args = 100,
    },{
```

The next example shows how to add sub-commands to the 'vmem' command:

First the sub-commands are defined:

```
static const gs_command_t GS_COMMAND_SUB cmd_vmem_sub[] = {
    {
        .name = "read",
        .help = "Read from virtual memory",
        .usage = "<addr> <length>",
        .handler = cmd_vmem_read,
        .mandatory_args = 2,
    },{
        .name = "write",
        .help = "Write to virtual memory",
        .usage = "<addr> <data>",
        .handler = cmd_vmem_write,
        .mandatory_args = 2,
    },{
        .name = "lock",
        .help = "Lock the virtual memory",
        .usage = "<entry>",
        .handler = cmd_vmem_lock,
        .mandatory_args = 1,
    },{
        .name = "unlock",
        .help = "Unlock the virtual memory",
        .usage = "<entry>",
        .handler = cmd_vmem_unlock,
        .mandatory_args = 1,
    },{
        .name = "list",
        .help = "Show virtual memory mappings",
        .handler = cmd_vmem_list,
        .mandatory_args = GS_COMMAND_NO_ARGS,
    }
};
```

Secondly the sub-commands are placed under the parent command 'vmem':

```
static const gs_command_t GS_COMMAND_ROOT cmd_vmem[] = {
    {
        .name = "vmem",
        .help = "Virtual memory",
        .chain = GS_COMMAND_INIT_CHAIN(cmd_vmem_sub),
    },
```

Remember to include *<gs/util/gosh/command.h>* to get the gs_command_t declaration.

Linker optimization

Instead of building the list of commands run-time using a linked list or similar data structure, the linker can group all root-commands in the same section. This is done by using a special GCC-attribute to pack all command struct's into the same memory area. In other words, the entire command list is always initialised as default and therefore requires zero time to initialize during startup. For this to work, all root-commands must be tagged with 'GS_COMMAND_ROOT', as shown in *Defining commands*.

In order to prevent the linker from removing the command object (optimization), it is necessary to register the command by calling *GS_COMMAND_REGISTER(...)*. Calling register on a already regsitered command is simply ignored.

Command handler

The handler function is a simple function, taking a single argument containing the command's *arguments* in a standard argc/argv notation:

```
static int cmd_sleep(gs_command_context_t * context)
{
    uint32_t sleep_ms;
    gs_error_t error = gs_string_to_uint32(context->argv[1], &sleep_ms);
    if (error) {
        return error;
    }

    gs_time_sleep_ms(sleep_ms);

    return GS_OK;
}
```

The return code is expected to be a *gs_error_t* type, which allows the caller to use 'gs_error_string()' to provide more verbose feedback to the user. But in practice, returning 0 means *success* and any other value means *failure*.

4. Tools

4.1 BuildTools

4.1.1 Introduction

BuildTools offers a range of various tools and scripts to ease the building of projects and libraries from GomSpace.

Software projects are build using Waf (https://waf.io/), which is a Python based build system. GomSpace has added some extensions in form of scripts and helper functions. These extensions can clearly be identified by the `gs` prefix.

4.1.2 Setting up and building a project

To setup a project to use BuildTools the following command must be run in the root directory of the project.

```
user@host$ ./tools/buildtools/gsbuildtools_bootstrap.py
```

This will create symlinks for the Waf binary throughout the project. If new modules are added after the project has been bootstrapped, this command must be run again.

Once the project has been bootstrapped, the project can be build by invoking the Waf program.

```
user@host$ ./waf distclean configure build
```

The *distclean* option will delete the *build* folder - removing all build artifacts.

The *configure* option instructs Waf to run all the *configure* methods in all wscript's (recursively). The *configure* step is typically used for configurering the build, checking if the necessary tools/compilers are available, etc.

The *build* option performs the actual build. Waf hashes all source files, so it only builds what has changed since last build. In order to build changes, simply run the Waf program (wihtout any options):

```
user@host$ ./waf
```

For further details on Waf, please see the officiel documentation on https://waf.io/. *The Waf Book* (https://waf.io/book/) is a good starting point for first time users.

4.1.3 Using BuildTools in scripts

BuildTools can also be used in a script to run a series of Waf commands. This can be achieved by creating a Python script in the root folder of the project.

```python
#!/usr/bin/env python
import gsbuildtools

from gs.buildtools import util

options1 = ['--my-opt1']
options2 = options1 + ['--my-opt2']
options3 = options2 + ['--my-opt3']

util.waf_command(options1)
```

```
util.waf_command(options2)
util.waf_command(options3)
```

This script will run waf three times with three different options configured.

5. Appendix

5.1 Software Changelogs

5.1.1 Product Interface Application

2.1.1 (2019-02-27)

- Improvement: Updated libraries.

2.0.3 (2019-01-30)

- Improvement: Updated documentation and libraries.

2.0.2 (2018-11-23)

- Feature: Changed argument parser to GNU argp.
- Feature: Updated clients and libraries.

2.0.1 (2018-08-14)

- Feature: Updated libraries.
- Feature: Replaced server task with gs_csp_service_dispatcher().

1.2 (2017-08-30)

- Feature: clients: Updated nanocam2 client module

1.1 (2017-06-04)

- Bug: clients: Fix for 'ax100 hk' command in nanocom-ax client
- Bug: clients: Fix for detecting timeout on 'eps hk' command in nanopower command
- Feature: clients: Updated P60 client modules
- Feature: clients: Added libfp and libhk clients

1.0 (2016-11-17)

- Feature: Initial release with all clients as well as ftp and rparam clients

5.1.2 Product Interfaces

ADCS (libadcs_client)

5.0.2 (2019-02-21)

- Improvement: Updated documentation.

5.0.1 (2019-02-07)

- Improvement: Updated json telemetry tables to match acutal tables

5.0.0 (2019-02-01)

- Breaking: Removed STR from table sensors_common and added its own
- Breaking: Updated GPS parameters in sensors_common
- Breaking: Updated telemetry table1 layout

- Breaking: Moved css_{initmax, absmax, th} the the sensor_css table, with the names max_init, abs_max and threshold
- Breaking: Moved tleline1, tleline2 and teme2eci from gnc_ads to gnc_common
- Feature: Added sun pointing mode
- Feature: Added orbit pointing mode
- Feature: Added handle to control VARF

4.0.0 (2018-01-04)

- Improvement: Updated documentation for libadcs 4.0.0

3.3.1 (2018-12-20)

- Improvement: Updated documentation.

3.3.0 (2018-11-22)

- Breaking: Updated the reaction wheel parameter table. It now assumes all wheels are the same type.
- Breaking: Created a new table to represent the on-board (internal) sensors of the A3200 (mpu3300 and hmc5843)
- Bug: Updated the GOSH command *gps ascii <timout> <cmd>* so it can't crash the A3200 on garbage input

3.2.2 (2018-09-24)

- Improvement: Separated solar panel control out to its own repository
- Improvement: Added a *type* field to the CSS parameter table
- Bug: Fixed an error where reading the *suns_semp* parameter would return garbage data

3.2.0 (2018-09-19)

- Feature: Initial SDK version

NanoCam C1U (nanocam2_client)

4.16.4 (2019-02-21)

- Improvement: Updated documentation.

4.16.3 (2018-12-20)

- Improvement: Updated documentation.

4.16.2 (2018-11-22)

- Improvement: Updated dependencies.

NanoCom ADS-B Receiver (gatoss-uc_client)

3.3.4 (2019-02-21)

- Improvement: Updated documentation.

3.3.3 (2018-12-20)

- Improvement: Updated documentation.

3.3.2 (2018-11-22)

- Improvement: Updated dependencies

3.3.1 (2018-08-14)

- Improvement: Updated dependencies

NanoCom AX100 (nanocom-ax_client)

3.10.0 (2018-12-19)

- Feature: HMAC list was added and contains a list of HMAC keys, selectable by remote command.
- Bug: Improved HMAC authentication. Legacy mode available through configuration.

3.9.12 (2018-11-22)

- Feature: Moved client into own repository.

NanoPower BPX (nanopower-bpx_client)

3.0.4 (2019-01-10)

- Improvement: Added optional arguments to bpx conf edit [<heater mode> <heater low temp> <heater high temp>]

3.0.3 (2018-12-20)

- Improvement: Updated documentation.

3.0.2 (2018-11-22)

- Improvement: Updated dependencies.

3.0.0 (2017-08-22)

- Bug: Fix of manual heating in I2C-slave mode
- Feature: Compatible with nanopower-bpx v3.0.0

NanoPower P31u (nanopower_client)

2.22.5 (2019-02-26)

- Feature: Added –enable_nanopower_config_commands option (default True). It enables/disables config commands to limit space needed

2.22.4 (2019-01-10)

- Improvement: Added option to edit configurations without input queries e.g., “eps conf output 0 1 1 0 0”
- Improvement: Python bindings updated with config get/print methods (eps_config_get…)

2.22.3 (2018-12-20)

- Improvement: Updated documentation.

2.22.2 (2018-11-22)

- Feature: Moved client to own repository.

NanoPower P60 ACU 200 (p60-acu_client)

1.2.16 (2019-02-21)

- Bug: Moved non-deprecated header files out of deprecated folder.
- Improvement: Updated documentation.

1.2.15 (2018-12-20)

- Improvement: Updated documentation.

1.2.14 (2018-11-22)

- Feature: Moved client to own repository.

NanoPower P60 Dock (p60-dock_client)

2.2.6 (2018-02-21)

- Bug: Moved non-deprecated header files out of deprecated folder.
- Improvement: Updated documentation.

2.2.5 (2018-12-20)

- Improvement: Updated documentation.

2.2.4 (2018-11-22)

- Feature: Moved client to own repository.

NanoPower P60 PDU (p60-pdu_client)

1.2.18 (2019-02-21)

- Bug: Moved non-deprecated header files out of deprecated folder.

1.2.17 (2018-12-20)

- Improvement: Updated documentation.

1.2.16 (2018-11-22)

- Feature: Moved client to own repository.

NanoPower P60 client (libp60_client)

1.0.3 (2018-12-20)

- Improvement: Updated documentation.

1.0.2 (2018-11-22)

- Feature: Initial release

5.1.3 Libraries

Aardvark Interface (libaardvark)

1.1.4 (2019-01-30)

- Improvement: Updated documentation.

1.1.3 (2018-12-20)

- Improvement: Updated documentation.

1.1.2 (2018-11-22)

- Feature: Added support for command line options, using argp.

Gomspace CSP extension (libgscsp)

2.4.3 (2019-01-29)

- Improvement: Register all used log groups.
- Improvement: libcsp: Support changing MTU on CAN interface.
- Bug: libcsp: Fixed race condition when allocating dynamic port (client connection).

2.4.2 (2018-12-19)

- Improvement: Added defines for GS_CSP_PORT_AIS(13) and GS_CSP_PORT_ADSB(14) to prevent future clashing
- Improvement: Updated documentation.

2.4.1 (2018-11-21)

- Improvement: Added gs_csp_init(), gs_csp_rtable_load() and command line support.

2.3.2 (2018-09-19)

- Improvement: Remove a few logs from ISR context, could cause crashes.
- Improvement: RDP - limit pending messages to window size (not windows size +1).
- Bug: Fixed possible RDP csp_send() deadlock issue (LIBGSCSP-6).
- Breaking: Updated gs_service_dispatcher and gs_router APIs to improve automatic testing.
- Feature: updated libcsp to public version 1.5 (release-1.5 branch).

2.3.1 (2018-08-13)

- Feature: Added gs_csp_service_dispatcher().
- Feature: Enable CRC32 on all connections as default.

2.2.2 (2018-05-30)

- Breaking: Updated internal use of reset API from libembed, which now takes a reset cause.
- Feature: Added support for CSP logging through libutil::log.

2.1.2 (2018-04-17)

- Breaking: Changed name of dynamic library: libgsgscsp -> libgscsp

2.0.0 (2018-03-07)

Software changelog (libcsp)

libcsp 1.4, 07-05-2015

- new: General rtable interface with support for STATIC or CIDR format
- new: CIDR (classless interdomain routing) route table format with netmasks
- new: Bridge capability
- new: Added routing table (de)serialization functions for load/save
- new: Automatic packet deduplication using CRC32 (compile time option)
- new: Autogenerated python bindings using ctypesgen
- new: Task-less operation with router invocation from external scheduler function
- api: Refactor route_if_add to csp_iflist_add
- api: Refactor route_set and friends to rtable_set

- api: Refactor csp_fifo_qos to csp_qfifo
- api: Added defined to be backwards compatible with 1.x
- interfaces: Drop packets on LOOP interface not for own address (blackhole)
- interfaces: New ZMQHUB interface (using zeroMQ over TCP)
- other: Increase stack size from 250 to 1100 for csp_can_rx_task
- other: Cleanup in csp_route.c
- other: Show incoming interface name in debug message
- other: Remove newlines from debug calls
- improvement: Reduce debug hook function complexity with valist passing
- fix: csp_sleep_ms did not work

libcsp 1.3, 07-05-2015

- new: Split long process lists into multiple packets
- new: Added posix csp_clock.h
- new: cmp clock functions (requires that you provide csp_clock.h implementation)
- new: Added SFP (Small fragmentation protocol) for larger data chunks
- fix: csp_if_fifo example
- fix: NULL char at the end of rps
- doc: Updated mtu documentation
- other: Tested with FreeRTOS 8.0
- other: Added disable-stlib option to build only object files

libcsp 1.2, 25-10-2013

- Feature release
- New: CMP service for peek and poke of memory
- New: CMP interface statistics struct is now packed
- New: Faster O(1) buffer system with reference counting and automatic alignment
- New: Thread safe KISS driver with support for multiple interfaces
- New: CSP interface struct now holds an opaque pointer to driver handle
- New: removed TXBUF from KISS driver entirely to minimize stack usage, added TX lock instead
- New: Pre-calculated CRC table .romem or PROGMEM on __avr__
- New: Added buffer overflow protection to KISS interface
- New: Allow posting null pointers on conn RX queues
- New: Lower memory usage on AVR8
- New: csp_route_save and csp_route_load functions
- New: option –disable-verbose to disable filenames and linenumber on debug
- Protocol: KISS uses csp_crc32 instead of it own embedded crc32
- Improvement: Use buffer clone function to copy promisc packets
- Bugfix: Fix pointer size (32/16bit) in cmp_peek/poke

- Bugfix: Issue with double free in KISS fixed
- Bugfix: Change rdp_send timeout from packet to connection timeout to make sending task block longer
- Bugfix: Fix conn pool leak when using security check and discarding new packets
- Bugfix: Add packet too short check for CRC32
- Bugfix: Accept CRC32 responses from nodes without CRC support
- Bugfix: Ensure csp_ping works for packets > 256 bytes
- Bugfix: Cleanup printf inside ISR functions
- Bugfix: Do not add forwarded packets to promisc queue twice
- Bugfix: Fix return value bug of csp_buffer_get when out of buffers
- Bugfix: Always post null pointer with lowest priority, not highest
- Bugfix: Add check on debug level before calling do_csp_debug, fixes #35
- Other: Export csp/arch include files
- Other: Remove the use of bool type from csp_debug
- Other: Moved csp debug functions to csp_debug.h instead of csp.h
- Other: Ensure assignment of id happens using the uint32_t .ext value of the union, quenches warning

libcsp 1.1, 24-08-2012

- Feature release
- Defacto stable since Feb 2012
- New: I2C interface
- New: KISS interface
- New: USART drivers for Linux, Mac and Windows
- New: Windows/MinGW support
- New: MacOSX support
- New: Interface register function
- New: Interface search function
- New: CMP service for remote route updating
- New: CMP service for interface statistics
- Improvement: Better QoS support
- Improvement: Send RDP control messages with high priority
- Improvement: WAF distcheck now works
- Improvement: Automatic endian discovery
- Improvement: Accept packets with CRC32 checksum if compiled without CRC32 support
- Improvement: Do not wake the router task if RDP is not enabled
- Improvement: Save 102 bytes of RAM by packing route entries
- Cleanup: Simplify CAN configuration
- Cleanup: Move architecture specific code to src/arch
- Bugfix: CSP_MEMFREE gives wrong answer on freertos AVR due to truncation
- Bugfix: Fixed wrong 64-bit size_t in csp_service_handler

- Bugfix: Fixed problem in csp_if_kiss when out of buffers
- Bigfix: Handle bus-off CAN IRQ for AT90CAN128

libcsp 1.0.1, 30-10-2011

- Hotfix release
- Bugfix: missing extern in csp_if_lo.h

libcsp 1.0, 24-10-2011

- First official release
- New: CSP 32-bit header 1.0
- Features: Network Router with promiscous mode, broadcast and QoS
- Features: Connection-oriented transport protocol w. flow-control
- Features: Connection-less “UDP” like transport
- Features: Encryption, Authentication and message check
- Features: Loopback interface
- Features: Python Bindings
- Features: CAN interface w. drivers for several chips
- Features: CSP-services (ping, reboot, uptime, memfree, buffree, ident)

Flight Planner (libfp_client)

3.4.1 (2019-02-26)

- Improvement: Improved error handling.

3.3.3 (2019-01-29)

- Improvement: Updated API documentation.

3.3.2 (2018-12-19)

- Improvement: Updated documentation.
- Feature: Autogenerated command documentation.

3.3.1 (2018-11-21)

- Feature: Moved client to separate repository.

File Transfer Protocol (libftp_client)

5.0.3 (2018-12-19)

- Bug: Fixed issue with ftp_list in python bindings
- Feature: Added support for FTP mode: GATOSS - enabling FTP access to a GATOSS node.
- Improvement: Updated documentation.

5.0.2 (2018-11-29)

- Bug: Fixed warnings generated by newer GCC compiler.

5.0.1 (2018-11-21)

- Breaking: client API simplified
- Breaking: all API functions return gs_error_t
- Breaking: no longer prints in colors (better support for rgosh)
- Breaking: All API functions prefixed with gs
- Breaking: ftp_upload_file/mem, ftp_download_file/mem is deprecated and hidden in GOSH
- Breaking: ftp_backend is removed
- Feature: No more global state variables in client
- Feature: upload / download now takes URL

4.3.3 (2018-09-21)

- Bug: Download of zero sized file fails

4.3.1 (2018-08-13)

- Bug: fixed leak in "ftp list" due to missing csp_close().

4.2.1 (2018-05-30)

- Feature: Added local zip/un-zip commands.

Gomspace Shell (libgosh_client)

3.4.2 (2018-12-19)

- Improvement: Updated documentation.

3.4.1 (2018-11-21)

- Feature: Split libgosh into libgosh (server) and libgosh_client (client).

Gomspace Sensor Bus (libgssb_client)

4.2.6 (2019-02-11)

- Improvement: Added src folder to dist.

4.2.5 (2019-01-28)

- Improvement: Updated API doc.

4.2.4 (2018-12-19)

- Improvement: Moved GSSB functionality to the client (backward compatible).

4.2.3 (2018-11-21)

- Feature: Added extern cplusplus to headers (support C++ usage).

4.2.2 (2018-08-13)

- Feature: Added command register API.

4.2.1 (2018-05-30)

4.0.1 (2018-03-01)

Housekeeping System (libhk_client)

4.2.1 (2019-02-26)

- Breaking: Moved host commands to libhk.
- Improvement: Improved beacon file loader.

4.1.4 (2019-01-29)

- Improvement: Removed files not used by the client (parameter files).

4.1.3 (2018-12-20)

- Improvement: Updated documentation.
- Bug: Fixed issue with logging in python parser.

4.1.2 (2018-12-03)

- Improvement: Fixed commands documentation

4.1.1 (2018-11-22)

- Feature: Added flag to completely enable/disable the Housekeeping Server.
- Breaking: Renamed root Command to better match overall scheme (hk_client -> hk, hk -> hk_srv).
- Feature: Added Command for asking Housekeeping Server to reload its configuration.
- Feature: Adding option to request beacon from old hksrv (command, and python bindings).

4.0.1 (2018-10-17)

- Feature: The client is now in a seperate repository.

Nano Protobuf (libnanopb)

1.1.1 (2019-02-26)

- Feature: Added support for Python output, handler: nano_proto_gen_1_1.
- Feature: Added support for Markdown documentation output, handler: nano_proto_gen_1_1

1.0.1 (2018-12-19)

- Improvement: Updated documentation.

1.0.0 (2018-11-21)

- Improvement: Updated documentation and added license.

0.1.2 (2018-08-14)

- Feature: Nanopb version 0.3.9 imported.
- Feature: Nanopb plugins for proto->c conversion.

Parameter System (libparam_client)

4.5.4 (2019-02-26)

- Improvement: Adjusted table columnspace in auto-generated rsti files.

4.5.3 (2019-01-29)

- Feature: Added rparam_set_data/rparam_get_data to bindings.

4.5.2 (2018-12-19)

- Bug: python binding for param_get_double reversed value and size.

- Bug: fixed rparam save/load requests - mixing file-id and table-id.
- Bug: fixed bug in index'ing arrays using auot-generated macros, if the index is something like 'x + y'
- Feature: added new rparam load/save from/to named stores.
- Improvement: Updated documentation.

4.5.1 (2018-11-21)

- Breaking: removed 'rparam clear' functionality.
- Improvement: rparam.py json matches param specifications (mem_id->id, array_size->array-size)
- Bug: Table 0 was not always read correctly through the Python API.

4.4.1 (2018-09-20)

- Bug: prevent stripping leading spaces, when setting string parameters.
- Feature: Added gs_param_table_instance_alloc() for allocating a table instance.

4.3.1 (2018-08-13)

- Feature: command: "rparam download" table-specification, only save if current work directory is defined (gs_getcwd()).
- Breaking: rparam and serialize API. REfactored to support checksum.
- Breaking: rparam command - auto-send changed to default false.
- Breaking: moved rparam query API internally for now - only used by rparam commands.
- Feature: added support for showing float/double using scientific notation.

4.2.2 (2018-05-30)

- Breaking: Renamed Parameter IO API to Parameter Protocol (PP).
- Bug: Fixed potential crash issue if parameter name to long, missing NULL termination.

Remote GOSH (librgosh_client)

1.0.2 (2019-01-28)

- Improvement: Register all used log groups.

1.0.1 (2018-12-19)

- Improvement: Updated documentation.

1.0.0 (2018-11-21)

- Feature: Added support for RGOSH client/server communication over CSP-RDP connection.
- Breaking: RGOSH port has been changed from the GSCRIPT port to it's own dedicated RGOSH port.
- Breaking: Updated/Improved the RGOSH protobuf messages.
- Improvement: More responses supported per request - Allows extensive data to be transferred.

0.1.2 (2018-08-14)

- Feature: RGOSH client API
- Feature: RGOSH python bindings.
- Feature: Run GOSH commands on remote CSP nodes - Results/output supported.
- Feature: RGOSH communication with RGOSH server using RGOSH protobuf messaging.

Utility (libutil)

3.7.1 (2019-02-26)

- Breaking: gs_command_init(): removed automatic registration of vmem commands, use gs_vmem_register_commands() to register commands.
- Breaking: fixed bugs in command_gen.py (Command generator JSON -> c/h). The register function will always uses the JSON filename.

3.6.1 (2019-01-30)

- Improvement: Added support for stopping console thread.
- Improvement: Enabled hexdump() to dump memory from address 0.
- Breaking: Removed attribute "log_groups" from log group definition, requiring log gruops to be registered to show up in lists.
- Improvement: Updated API documentation.

3.5.2 (2018-12-19)

- Improvement: Updated documentation.

3.5.1 (2018-11-21)

- Breaking: Re-factored the log framework. Log statements are not affected. Only log configuration/setup.
- Breaking: Removed gs_time_rel_s() and gs_time_rel_s_isr().
- Breaking: Linux, gs_command_register() must always be called to register commands.
- Improvement: Added gs_console_start(), which simplifies console initialization.

3.4.1 (2018-09-20)

- Feature: Added gs_thread_create_with_stack, so that stack buffer can be manually allocated.
- Feature: Added gs_string_endswith function.
- Improvement: Added logging functionality from interrupt (ISR) context , e.g. gs_log_isr().
- Feature: Added gs_crc8 API.
- Improvement: Increased accepted number commands arguments from 25 to 30.
- Feature: Added gs_thread_join() - primarily for testing.
- Feature: Added support for logging of gosh commands. Default logger available.

3.3.2 (2018-08-13)

- Feature: Added unistd::gs_getcwd().
- Feature: GOSH/command added support for specifying mandatory and optional arguments.
- Feature: GOSH/command added support for other IO streams than stdio.
- Feature: Added simple log file store.
- Breaking: GOSH/command, changed gs_command_context_t, gs_command_t and 'complete' callback.
- Breaking: Removed driver_debug API - use Log API instead.

3.2.1 (2018-05-30)

- Breaking: Changed GPIO ISR callback to make use of context switch.
- Feature: Added 'no check' versions of GPIO get/set for better perfomance.
- Feature: Added microsecond delays since provided timestamp
- Feature: Added bytebuffer API.

3.1.2 (2018-04-17)

- Breaking: Changed I2C interface for getting buffers, gs_i2c_slave_get_rx_buf_t / gs_i2c_slave_set_get_rx_buf().
- Feature: Added gs_fletcher16() functions for handling streaming data.
- Feature: re-added deprecated color_printf and GOSH APIs in order to support “old” clients.

3.0.1 (2018-03-14)

- Breaking: Replaced LZO with miniz (Zip)
- Feature: GOSH (for local commands) moved to libutil
- Feature: Generic GPIO prototype and x86_64 implementation
- Feature: Additional string functions

3.0.0 (2017-12-04)

- Feature: Software watchdog
- Improvement: Updated to LZO 2.10
- Feature: Additional string functions
- Feature: Additional date/time functions
- Feature: Prototypes for stdio

2.1.0 (2017-07-28)

- Improvement: Functionality for 8-bit architectures

2.0.0 (2017-06-23)

- Improvement: Use gsbuildtools.
- Feature: Added cross-platform API’s for various IO functionality, e.g. SPI, I2C, CAN.

1.0 (2015-05-08)

- Feature: Allow time to be set on linux
- Feature: Support for liblog
- Feature: LZO re-added and tested on linux
- Feature: VMEM system
- Feature: Added documentation folder in sphinx format
- Feature: High resolution tick timer system

5.1.4 Tools

Buildtools (buildtools)

2.6.5 (2018-12-19)

- Bug: keep dependency towards libgscsp / libcsp when building through bitbake.
- Improvement: Updated documentation.

2.6.4 (2018-11-26)

- Bug: Fixed ‘status’, causing warning about wrong versions.
- Improvement: Added check ‘status’ for multiple unique/checkout keys: sha/min_version/branch

2.6.2 (2018-11-21)

- Feature: Allow ‘master’ branches to use ‘release-‘ branches (disabled check).

2.6.1 (2018-11-14)

- Feature: Added more specialized functions for generating SDK and Product manuals (confidentiality classification is set to empty).

2.5.1 (2018-10-10)

- Feature: Better docker building support. Force docker on/off, mount entire workspace, bind to specific version of image.

2.4.2 (2018-08-13)

- Breaking: removed option “–manifest-file” from command “gitinfo”. Generate internal and external manifests.
- Feature: updated Waf, 1.9.7 -> 2.0.9.

2.3.1 (2018-05-30)

- Breaking: changed arguments/inputs for gs_doc::doxygen, changed to lists or Waf environment.
- Feature: added gs_add_doxygen() to support building API documentation for multiple modules.

2.1.2 (2018-04-18)

- Feature: added gs_gcc.gs_recurse() and gs_gcc.gs_recurse_unit_test().
- Feature: gs_doc.add_task_doc: added support for generating API documentation, using doxygen.
- Breaking: gs_test_cmocka: removed ‘skipping build’ in case of missing use relations.

2.0.1 (2018-03-14)

1.0.0 (2017-06-23)

5.2 API Documentation

Please refer to HTML documentation