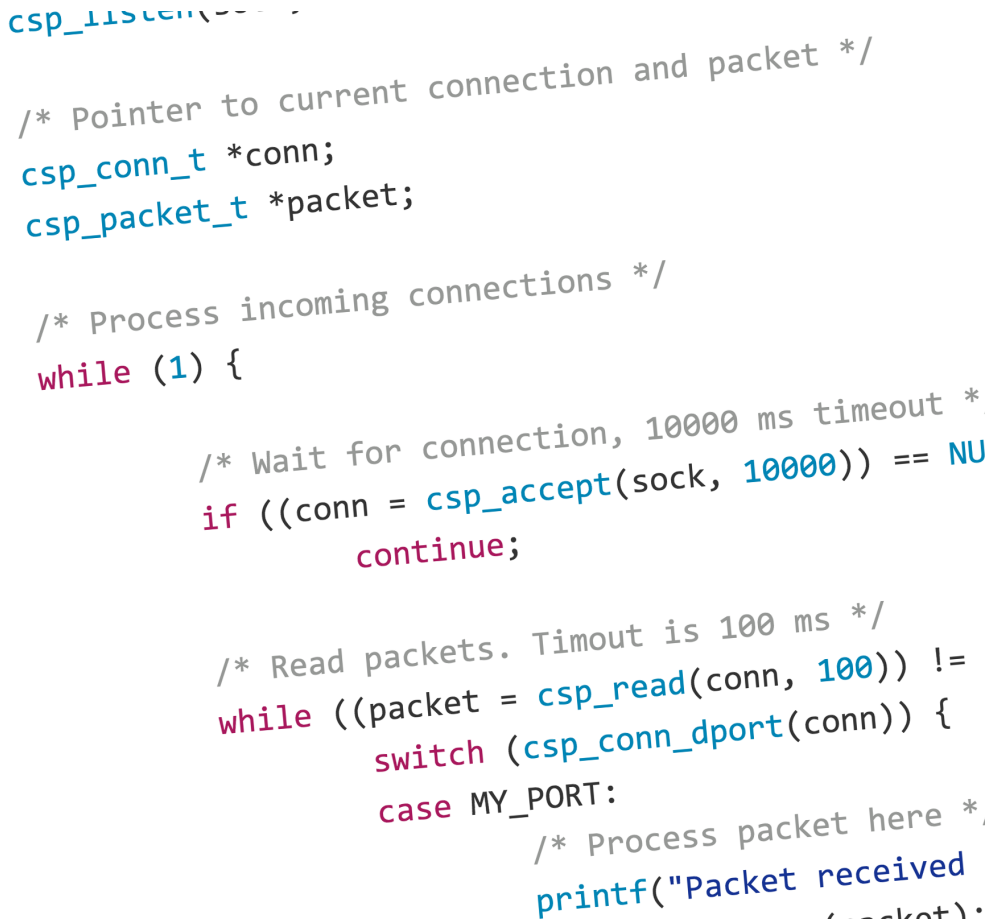


NanoSoft

Product Interface Application

Manual

Software Development Kit

Release 2.7.1

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

Product name: NanoSoft Product Interface Application

Reference: 1018560

Revision: 2.7.1

Date: 27 August 2020

Confidentiality Notice

This document is submitted for a specific purpose as agreed in writing and contains information, which is confidential and proprietary. The recipient agrees by accepting this document, that this material will not be used, transferred, reproduced, modified, copied or disclosed in whole or in part, in any manner or to any third party, except own staff to meet the purpose for which it was submitted without prior written consent.

GomSpace © 2020

Table of Contents

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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

1.2 Getting started

1.2.1 Prerequisites

The software has been built and verified on Ubuntu 18.04.4 LTS (http://www.ubuntu.com), with following tools:

- Python 2.7.17

Install Ubuntu prerequisites:

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

1.2.2 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 xvf <archive.tar.gz>
   ```

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

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

1.2.3 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.

1.2.4 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:
```

(continues on next page)

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
    $ <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.
```

1.3 CAN

GomSpace use a CAN USB dongle (https://www.peak-system.com/PCAN-USB.199.0.html?&L=1), for CAN communication with products.

Once the CAN dongle is connected to the PC, it should be visible in the USB device list:

```
$ lsusb
Bus 001 Device 108: ID 0c72:000c PEAK System PCAN-USB
```

To configure the device (name: can0, speed: 1Mbps and error handling: restart), use following commands:

```
$ sudo modprobe peak_usb
$ sudo ip link set dev can0 down
$ sudo ip link set can0 type can restart-ms 1000
$ sudo ip link set dev can0 up type can bitrate 1000000
```

The device can be used by the Product Interface Application, using option *-c*.

1.3.1 can-utils

can-utils is Linux package, containing a number of tools for CAN debugging:

- candump: dump all traffic on a CAN device
- cansend: send CAN message, consisting of a message id and up to 8 bytes payload

can-utils can be installed using standard commands:

```
$ sudo apt install can-utils
```

1.4 I2C/SPI

GomSpace uses a I2C/SPI USB dongle for I2C/SPI communication with products. See *Aardvark (libaardvark)* for details.

The device can be used by the Product Interface Application, using option *-i*, *-I* or *-S*.

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

1.5 Python bindings

Some of the clients include Python bindings for ground operations using Python. The Python bindings are experimental and in active development, the documentation and support is limited.

The bindings utilise the standard Python C-API and are special .so-files that call the C-code in the corresponding library .so-file. (e.g., libgsutil_py3.so and libgsutil.so) To compile the Python bindings, call Waf adding an extra option (or remove the corresponding line from the wscript):

```
$ ./waf distclean configure --build-disable="" build
```

To use the Python bindings paths to the .so-files must be in the LD_LIBRARY_PATH environment variable and Python module paths added to the env PYTHON_PATH environment variable. The library methods are used in Python files by e.g.,:

```
import libgsftp_client_py3 as gs_ftp
```

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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`.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

```
/* NANOCAM_PORT_SNAP */
#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:

```
/* NANOCAM_PORT_STORE */
#define NANOCAM_STORE_RAW       0 /* Store RAW sensor output (1 pixel into 2 bytes) */
#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) */
#define NANOCAM_STORE_RAW10     4 /* Store packed RAW output (4 pixels into 5 bytes) */
#define NANOCAM_STORE_UNKNOWN   UINT8_MAX
```

RAW pictures from the camera are 10 bit Bayer pattern (GRGB) stored in 16 bits. The 10 data bits are stored in bit [13-4] (mask 0x3FF0):

- [B3:B0] is expected to be low
- [B13:B4] is the image data
- [B15:B14] is expected to be high.

```
|--------------
| G R G R G R
| B G B G B G
|
```

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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*.

2.6.3 Epsslave

Epsslave is a client which can be used when the P31u is in "I^2C Slave Mode (Legacy Interface)".

It can be used with the nanosoft-product-interface-application (csp-client) and an Aardvark dongle (https://www.totalphase.com/products/aardvark-i2cspi/). The Aardvark dongle allow transfer of serial messages (I2C and SPI) to an embdded device via USB and is supported by the csp-client. More info can be found in the nanosoft-product-interface-application manual in the the Aardvark library section.

To enable epsslave in csp-client the following steps are needed:

- Set P31u in I^2C slave mode:

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

```
eps # board i2cslave 1
eps # reset
```

- Enable epsslave in csp-client:

```
src/main.c:
extern void cmd_epsslave_setup(void);
cmd_epsslave_setup();
```

- Compile
- Connect Aardvark dongle
- Start csp-client and use commands grouped under *epsslave*:

```
./build/csp-client -I
csp-client # epsslave <TAB>
```

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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*.

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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 (*libgscsp*) is a GomSpace extension to the open source *CubeSat Space Protocol* library.

The GomSpace CSP library provides:

- convience wrapping of CSP functionality, primarily initialization.
- definition of standard CSP ports (used by other GomSpace products).
- connecting low-level drivers (e.g. CAN, I2C from Embed library) with CSP interfaces
- generic CSP service dispatcher, forwards incoming connections to service handlers.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

The *libgscsp* contains a GomSpace branch (https://github.com/GomSpace/libcsp) of the open source libcsp (https://github.com/libcsp/libcsp), located in the subfolder *lib/libcsp*. The two *libcsp* branches are kept as identical as possible, as features specific to GomSpace are placed in *libgscsp*.

The section *CubeSat Space Protocol* contains the documentation from the public version of *libcsp*, and therefore doesn't contain any specifics about GomSpace systems.

3.2.2 Commands

All commands provided by the GomSpace CSP library are root commands, and therefore 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 connection oriented socket API and message/connection pools.

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, Linux (POSIX), MacOS and Windows. The primiary platforms being used are FreeRTOS and 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 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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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 defined 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 in regards to code and memory required
- Zero-copy buffer and queue system
- Modular network interface system
- OS abstraction, currently ported to: FreeRTOS, Linux (POSIX), MacOS and Windows
- 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 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 SpaceWire, I2C and RS232, the name was therefore extended to the more general CubeSat Space Protocol without changing the abbreviation.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Satellites using CSP Here is a list of some of the known satellites or organisations, that uses CSP:

- GomSpace GATOSS GOMX-1
- AAUSAT-3
- EgyCubeSat
- EuroLuna
- NUTS
- Hawaiian Space Flight Laboratory
- GomSpace GOMX-3, GOMX-4 A & B

See list of libcsp fork's here: https://github.com/libcsp/libcsp/network/members

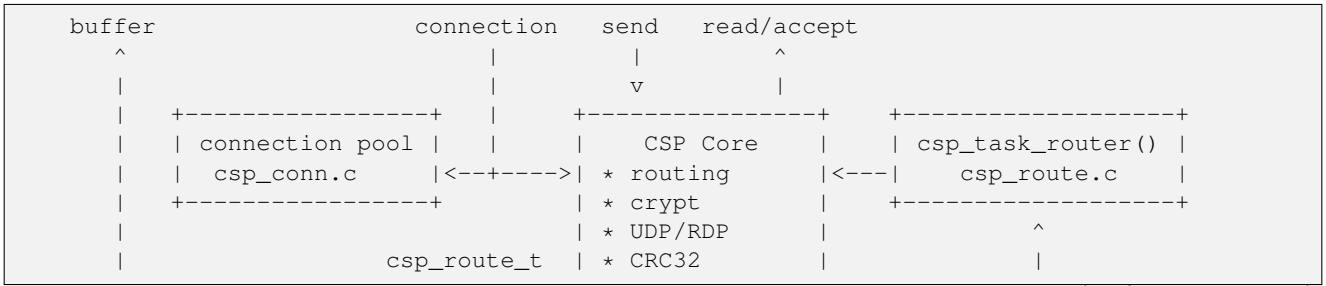
Structure

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

Folder	Description
libcsp/include/csp	Public header files
libcsp/include/csp/arch	Architecture (platform)
libcsp/include/csp/interfaces	Interfaces
libcsp/include/csp/drivers	Drivers
libcsp/src	Source modules and internal header files
libcsp/src/arch	Architecture (platform) specific code
libcsp/src/arch/freertos	FreeRTOS
libcsp/src/arch/macosx	Mac OS X
libcsp/src/arch/posix	Posix (Linux)
libcsp/src/arch/windows	Windows
libcsp/src/bindings/python	Python3 wrapper for libcsp
libcsp/src/crypto	HMAC, SHA and XTEA.
libcsp/src/drivers	Drivers, mostly platform specific (Linux)
libcsp/src/drivers/can	CAN
libcsp/src/drivers/usart	USART
libcsp/src/interfaces	Interfaces, CAN, I2C, KISS, LOOPBACK and ZMQHUB
libcsp/src/rtable	Routing tables
libcsp/src/transport	Transport layer: UDP, RDP
libcsp/utils	Utilities, Python scripts for decoding CSP headers.
libcsp/examples	CSP examples, C/Python, zmqproxy
libcsp/doc	RST based documention (this documentation)

The basics of CSP

The following diagram shows a conceptual overview of the different blocks in CSP. The shown inferface is CAN (src/interfaces/csp_if_can.c, driver: src/drivers/can/can_socketcan.c).



(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
       v                  +------------>|                 |     +----------------+
+--------------+          |             |                 |     | incoming queue |
| buffer pool  |          |             |                 |     |     qfifo.c    |
| csp_buffer.c |  +-------------+      +-----------------+     +----------------+
+--------------+  |routing table|            |                         ^
       ^          |  rtable.c   |            v                         |
       |          +-------------+      +-----------------------------------+
       |                               |  (next_hop)                       |
       |                               |    CAN interface (csp_if_can.c)   |
       +------------------------------>|                      csp_can_rx()  |
                                       +-----------------------------------+
                                             |   CAN frame (8 bytes)   ^
                                             v                         |
                                    csp_can_tx_frame()       socketcan_rx_thread()
                                             (drivers/can/can_socketcan.c)
```

Buffer All buffers are allocated once during initialization of CSP, 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.

Definition of a buffer element *csp_packet_t*:

```
/**
   CSP Packet.

   This structure is constructed to fit with all interface and protocols to prevent the
   need to copy data (zero copy).

   @note In most cases a CSP packet cannot be reused in case of send failure, because the
   lower layers may add additional data causing increased length (e.g. CRC32), convert
   the CSP id to different endian (e.g. I2C), etc.
*/
typedef struct {
        /**
           Padding. These bytes are intended for use by protocols, which want to prepend
           data before sending it, without having to copy/reorganize the entire message.
        */
        uint8_t padding[CSP_PADDING_BYTES];
        /** Data length. Must be just before CSP ID.*/
        uint16_t length;
        /** CSP id. Must be just before data, as it allows the interface to id and data
            in a single operation. */
        csp_id_t id;
        /**
           Data part of packet.
           When using the csp_buffer API, the size of the data part is set by
           csp_buffer_init(), and can later be accessed by csp_buffer_data_size()
        */
        union {
                /** Access data as uint8_t. */
                uint8_t data[0];
                /** Access data as uint16_t */
                uint16_t data16[0];
                /** Access data as uint32_t */
                uint32_t data32[0];
```

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
        };
} csp_packet_t;
```

A basic concept in 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. This is achieved by a number of *padding* bytes in the buffer, allowing for a header to be prepended at the lower layers without copying the actual payload. This also means that there is a strict contract between the layers, which data can be modified and where.

The padding bytes are used by the I2C interface, where the *csp_packet_t* will be casted to a *csp_i2c_frame_t*, when the interface calls the driver Tx function *csp_i2c_driver_tx_t*:

```
/**
   I2C frame.
   This struct fits on top of a #csp_packet_t, removing the need for copying data.
*/
typedef struct i2c_frame_s {
    //! Not used  (-> csp_packet_t.padding)
    uint8_t padding[3];
    //! Cleared before Tx  (-> csp_packet_t.padding)
    uint8_t retries;
    //! Not used  (-> csp_packet_t.padding)
    uint32_t reserved;
    //! Destination address  (-> csp_packet_t.padding)
    uint8_t dest;
    //! Cleared before Tx  (-> csp_packet_t.padding)
    uint8_t len_rx;
    //! Length of \a data part  (-> csp_packet_t.length)
    uint16_t len;
    //! CSP id + data  (-> csp_packet_t.id)
    uint8_t data[0];
} csp_i2c_frame_t;
```

Connection CSP supports both connection-less and connection-oriented connections. See more about protocols in *Layer 4: Transport Layer*.

During initialization libcsp allocates the configured number of connections. The required number of connections depends on the application. Here is a list functions, that will allocate a connection from the connection pool:

- client connection, call to *csp_connect()*
- server socket for listening *csp_socket()*
- server accepting an incmoing connection *csp_accept()*

An applications receive queue is located on the connection and is also allocated once during initialization. The length of the queue is the same for all queues, and specified in the configuration.

Send The data flow from the application to the driver, can basically be broken down into following steps:

1. if using connection-oriented communication, establish a connection> *csp_connect()*, *csp_accept()*
2. get packet from the buffer pool: *csp_buffer_get()*
3. add payload data to the packet
4. send packet, e.g. *csp_send()*, *csp_sendto()*
5. CSP looks up the destination route, using the routing table, and calls *nexthop()* on the resolved interface.
6. The interface (in this case the CAN interface), splits the packet into a number of CAN frames (8 bytes) and forwards them to the driver.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Receive The data flow from the driver to the application, can basically be broken down into following steps:

1. the driver layer forwards the raw data frames to the interface, in this case CAN frames
2. the interface will aquire a free buffer (e.g. *csp_buffer_get_isr()*) for assembling the CAN frames into a complete packet
3. once the interface has successfully assembled a packet, the packet is queued for routing - primarily to decouple the interface, e.g. if the interfacec/drivers uses interrupt (ISR).
4. the router picks up the packet from the incoming queue and routes it on - this can either to a local destination, or another interface.
5. the application waits for new packets at its Rx queue, by calling *csp_read()* or *csp_accept* in case it is a server socket.
6. the application can now process the packet, and either send it using e.g. *csp_send()*, or free the packet using *csp_buffer_free()*.

Routing table When a packet is routed, the destination address is looked up in the routing table, which results in a *csp_route_t* record. The record contains the inteface (*csp_iface_t*) the packet is to be send on, and an optional *via* address. The *via* address is used, when the sender cannot direcly reach the receiver on one of its connected networks, e.g. sending a packet from the satellite to the ground - the radio will be the *via* address.

CSP comes with 2 routing table implementations (selected at compile time).

- static: supports a one-to-one mapping, meaning routes must be configured per destination address or a single *default* address. The *default* address is used, in case there are no routes set for the specific destination address. The *static* routing table has the fastest lookup, but requires more setup.
- cidr (Classless Inter-Domain Routing): supports a one-to-many mapping, meaning routes can be configued for a range of destianation addresses. The *cidr* is a bit slower for lookup, but simple to setup.

Routes can be configured using text strings in the format:

<address>[/mask] <interface name> [via]

- address: is the destination address, the routing table will match it against the CSP header destination.
- mask (optional): determines how many MSB bits of address are to be matched. mask = 1 will only match the MSB bit, mask = 2 will match 2 MSB bits. Mask values different from 0 and 5, is only supported by the cidr rtable.
- interface name: name of the interface to route the packet on
- via (optional) address: if different from 255, route the packet to the *via* address, instead of the address in the CSP header.

Here are some examples:

- “10 I2C” route destination address 10 to the “I2C” interface and send it to address 10 (no *via*).
- “10 I2C 30” route destination address 10 to the “I2C” interface and send it to address 30 (*via*). The original destination address 10 is not changed in the CSP header of the packet.
- “16/1 CAN 4” (CIDR only) route all destinations addresses 16-31 to address 4 on the CAN interface.
- “0/0 CAN” default route, if no other matching route is found, route packet onto the CAN interface.

Interface The interface typically implements *Layer 2: MAC interfaces*, and uses drivers from *Layer 1: Drivers* to send/receive data. The interface is a generic struct, with no knowledge of any specific interface , protocol or driver:

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

```
/**
   CSP interface.
*/
struct csp_iface_s {
    const char *name;           //!< Name, max compare length is #CSP_IFLIST_NAME_MAX
    void * interface_data;      //!< Interface data, only known/used by the interface␣
↪layer, e.g. state information.
    void * driver_data;         //!< Driver data, only known/used by the driver layer, e.g.
↪ device/channel references.
    nexthop_t nexthop;          //!< Next hop (Tx) function
    uint16_t mtu;               //!< Maximum Transmission Unit of interface
    uint8_t split_horizon_off; //!< Disable the route-loop prevention
    uint32_t tx;                //!< Successfully transmitted packets
    uint32_t rx;                //!< Successfully received packets
    uint32_t tx_error;          //!< Transmit errors (packets)
    uint32_t rx_error;          //!< Receive errors, e.g. too large message
    uint32_t drop;              //!< Dropped packets
    uint32_t autherr;           //!< Authentication errors (packets)
    uint32_t frame;             //!< Frame format errors (packets)
    uint32_t txbytes;           //!< Transmitted bytes
    uint32_t rxbytes;           //!< Received bytes
    uint32_t irq;               //!< Interrupts
    struct csp_iface_s *next;   //!< Internal, interfaces are stored in a linked list
};
```

If an interface implementation needs to store data, e.g. state information (KISS), it can use the pointer *interface_data* to reference any data structure needed. The driver implementation can use the pointer *driver_data* for storing data, e.g. device number.

See function *csp_can_socketcan_open_and_add_interface()* in *src/drivers/can/can_socketcan.c* for an example of how to implement a CAN driver and hooking it into CSP, using the CSP standard CAN interface.

Send When CSP needs to send a packet, it calls *nexthop* on the interface returned by route lookup. If the interface succeeds in sending the packet, it must free the packet. In case of failure, the packet must not be freed by the interface. The original idea was, that the packet could be retried later on, without having to re-create the packet again. However, the current implementation does not yet fully support this as some interfaces modifies header (endian conversion) or data (adding CRC32).

Receive When receiving data, the driver calls into the interface with the received data, e.g. *csp_can_rx()*. The interface will convert/copy the data into a packet (e.g. by assembling all CAN frames). Once a complete packet is received, the packet is queued for later CSP processing, by calling *csp_qfifo_write()*.

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.

The current libcsp implementation primarily uses dynamic memory allocation during initialization, where all structures are allocated: port tables, connection pools, buffer pools, message queues, semaphores, tasks, etc.

Once the initiallization is complete, there are only a few functions that uses dynamic allocation, such as:

- csp_sfp_recv() - sending larger memory chuncks than can fit into a single CSP message.
- csp_rtable (cidr only) - adding new elements may allocate memory.

This means that there are no *alloc/free* after initialization, possibly causing fragmented memory which especially can be a problem on small systems with limited memory. It also allows for a very simple memory allocator (implementation of *csp_malloc()*), as *free* can be avoided.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Future versions of libcsp may provide a *pure* static memory layout, since newer FreeRTOS versions allows for specifying memory for queues, semaphores, tasks, etc.

The Protocol Stack

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

Layer 1: Drivers 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 only been included for some specific platforms. If you do not find your driver supported, it is quite simple to add a driver that conforms to the CSP interface. 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 a *via* 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 destination and source address together with port numbers for the connection. The router supports both local destination and forwarding to an external destination. Messages 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 sub-systems. 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 *Network Topology* section. 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 a defined 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. 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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

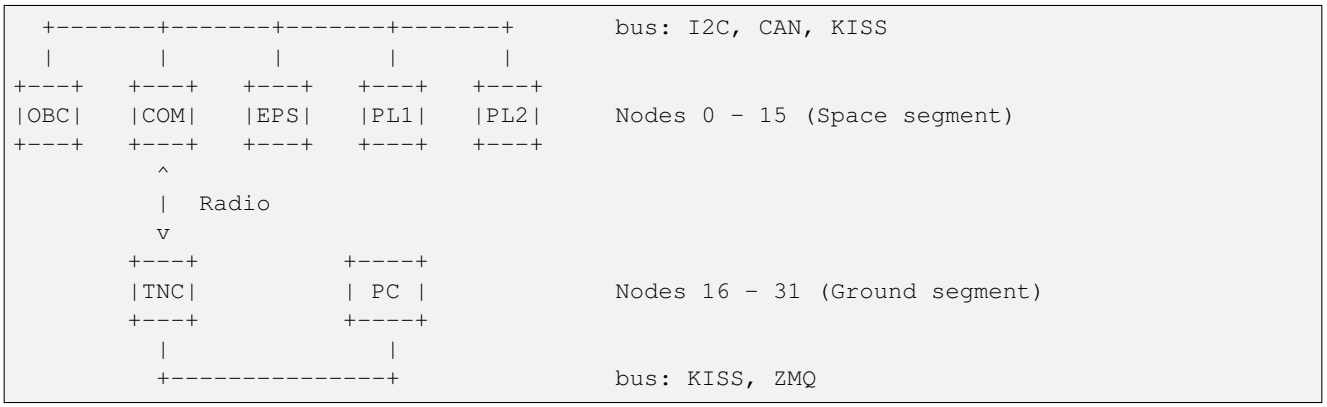
RDP CSP provides a transport layer extension called RDP (reliable datagram protocol) which is an implementation of RFC908 and RFC1151. RDP provides a few additional features:

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

For more information on this, please refer to RFC908 and RFC1151.

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.



The address range, from 0 to 31, 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.

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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 your 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 provides 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 example in *example/csp_server_client.c* provides a simple server/client setup, where the client sends a request to the server and receives a reply. The code can be compiled to an executable using *./examples/buildall.py*.

The example supports the drivers and interfaces in CSP:

- ZMQHUB: -z <host name|ip>

  Requires no extra hardware, as it uses standard network. The zmqproxy will need to be started.

- CAN: -c <can device>

  Requires a physical CAN interface. There are several CAN dongles on the market, for example https://www.peak-system.com/PCAN-USB.199.0.html.

  To achieve best performance and stabilty, following options can be set on the CAN device:

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

```
linux: sudo ip link set dev can0 down
linux: sudo ip link set dev can0 up type can bitrate 1000000 restart-ms 100
linux: sudo ip link set dev can0 txqueuelen 100
```

- KISS: -k <serial device>

  Requires a serial interface, e.g. USB dongle.

Running the example If the example is started without any interfaces, it will use the loopback interface for communication between client and server:

```
ubuntu-18:~/libcsp$ ./build/csp_server_client
1586816581.410181 Initialising CSP
Connection table
[00 0x55a00f7adee0] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[01 0x55a00f7adf68] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[02 0x55a00f7adff0] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[03 0x55a00f7ae078] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[04 0x55a00f7ae100] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[05 0x55a00f7ae188] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[06 0x55a00f7ae210] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[07 0x55a00f7ae298] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[08 0x55a00f7ae320] S:0, 0 -> 0, 0 -> 0, sock: (nil)
[09 0x55a00f7ae3a8] S:0, 0 -> 0, 0 -> 0, sock: (nil)
Interfaces
LOOP       tx: 00000 rx: 00000 txe: 00000 rxe: 00000
           drop: 00000 autherr: 00000 frame: 00000
           txb: 0 (0.0B) rxb: 0 (0.0B) MTU: 0

Route table
1/5 LOOP
0/0 LOOP
1586816581.410405 Server task started
1586816581.410453 Binding socket 0x55a00f7adf68 to port 25
1586816581.410543 Client task started
1586816582.410983 SERVICE: Ping received
1586816582.411135 Ping address: 1, result 0 [mS]
1586816582.411174 reboot system request sent to address: 1
1586816582.461341 csp_sys_reboot not supported - no user function set
1586816582.512532 Packet received on MY_SERVER_PORT: Hello World (1)
```

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>
```

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
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);
} //
```

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 FP.
fp load	Load FP from file - Will Flush current FP. Arguments: <path>: Local FP file to load. No more than 49 characters long. [fast_load_with_unique_keys]: When set ALL keys in plan MUST be unique (advanced mode)
fp store	Store current FP to file. Arguments: <path>: Local file to save current FP to. No more than 49 characters long.
fp create	Create new timer. Arguments: <timer>: Name of the timer. No more than 29 characters long. <[+]sec>: Absolute or relative time to execute the command. Prefix with '+' for relative time. <command>: Command to execute. No more than 59 characters long. [repeat]: Number of times to repeat the command execution - only valid for relative timers. [state]: Initial state of the timer, either 'active' or 'dormant'.
fp delete	Delete timer. Arguments: <timer>: Timer name.
fp active	Set timer active. Arguments: <timer>: Timer name.

Continued on next page

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Table 3.1 – continued from previous page

Command	
fp allactive	Set all timers active.
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 command. 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
```

Any line starting with a # will be treated as a comment.

The table *Flight plan file* explains each of the entries.

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Table 3.2: Flight plan file

Entry	Explanation
name	Unique id of the timer. ASCII string of a maximum of 29 characters.
command	The command to execute. ASCII string of a maximum of 119 characters.
state	Specifies whether the timer is active (0) or dormant (1).
basis	Specifies whether the timer is an absolute (0) or relative (1) timer.
last_sec	The last execution timestamp seconds part. Ignored for absolute timers.
last_nsec	The last execution timestamp nanoseconds part. Ignored for absolute timers.
when_sec	Execution timestamp seconds part.
when_nsec	Execution timestamp nanoseconds part.
repeat	Number of repeats. Only for relative timers.

If a line in a flight plan file has an error, that specific timer will be discarded, the rest will be loaded.

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
```

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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
```

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

3.6 GomSpace Sensor Bus (libgssb_client)

3.6.1 Introduction

GSSB is an abbrivation of GomSpace Sensor Bus. It a I^2C 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^2C bus, see Fig.3.1.

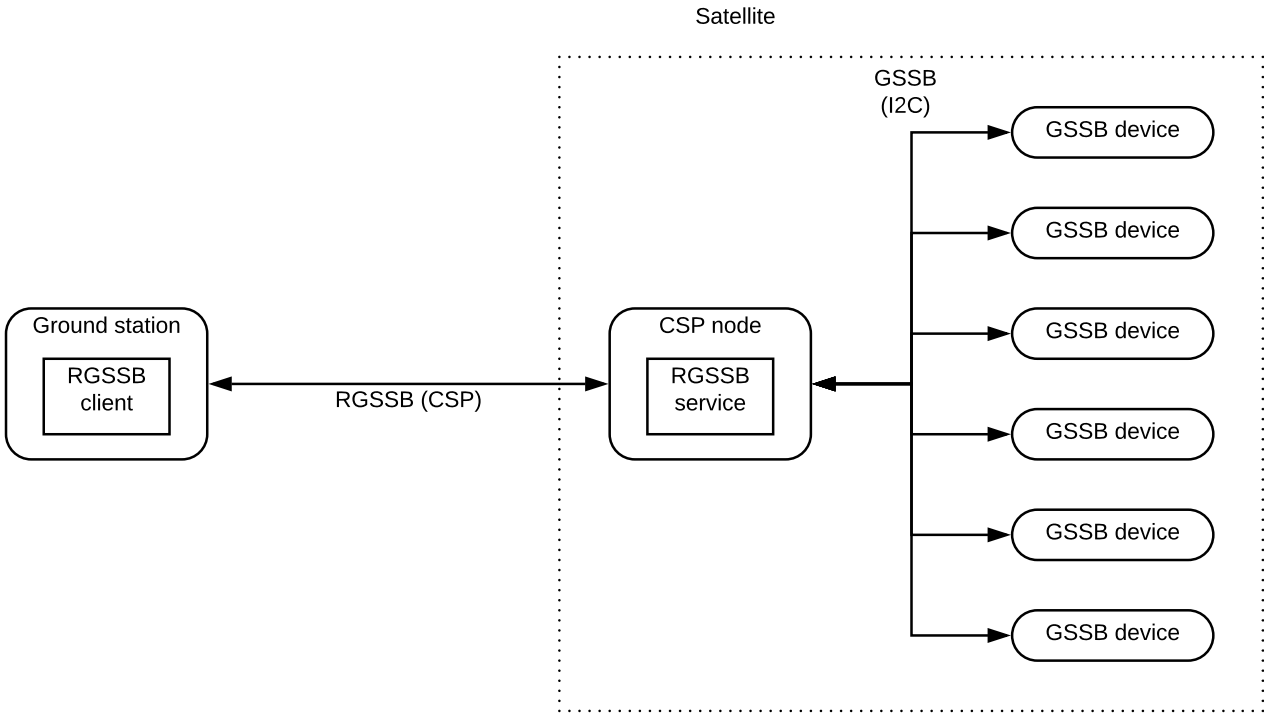


Fig. 3.1: GSSB network illustration.

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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 3.3: Command Table 'hk'

Command	
hk server	Setup CSP address of HK server. Arguments: <server>: CSP address. [burst_size]: Size of bursts when doing 'hk get'. [burst_idle]: Idle time (milliseconds) between bursts when doing 'hk get'.
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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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 micro-controllers, 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:

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

```
csp-client # rparam
Parameter System client
  init                Set server and load table specification from file
  download            Set server and download table specification from server
  getall              Download all table values from server
  list                List cached table values (use getall to download values)
  copy                Copy table on remote server (version <= 3 only)
  load                Load table on remote server
  save                Save table on remote server
  get                 query: add 'get' to current transactions
  set                 query: add 'set' to current transactions
  reset               query: reset current transactions
  send                query: send current transactions
  autosend            query: Enable/disable autosend for set & get
  wd                  Set working directory for init/download
  timeout             Set timeout
  checksum            Set checksum
```

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 "A3200"
  0x0010 type            U8  0
  0x0011 rev             U8  0
  0x0012 dock_type       STR ""
  0x0022 csp_addr        U8  1
  0x0023 csp_rtable      STR "0/0 CAN, 10/5 I2C, 11/5 KISS"
  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  true
  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 queued 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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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, I2C, 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*.

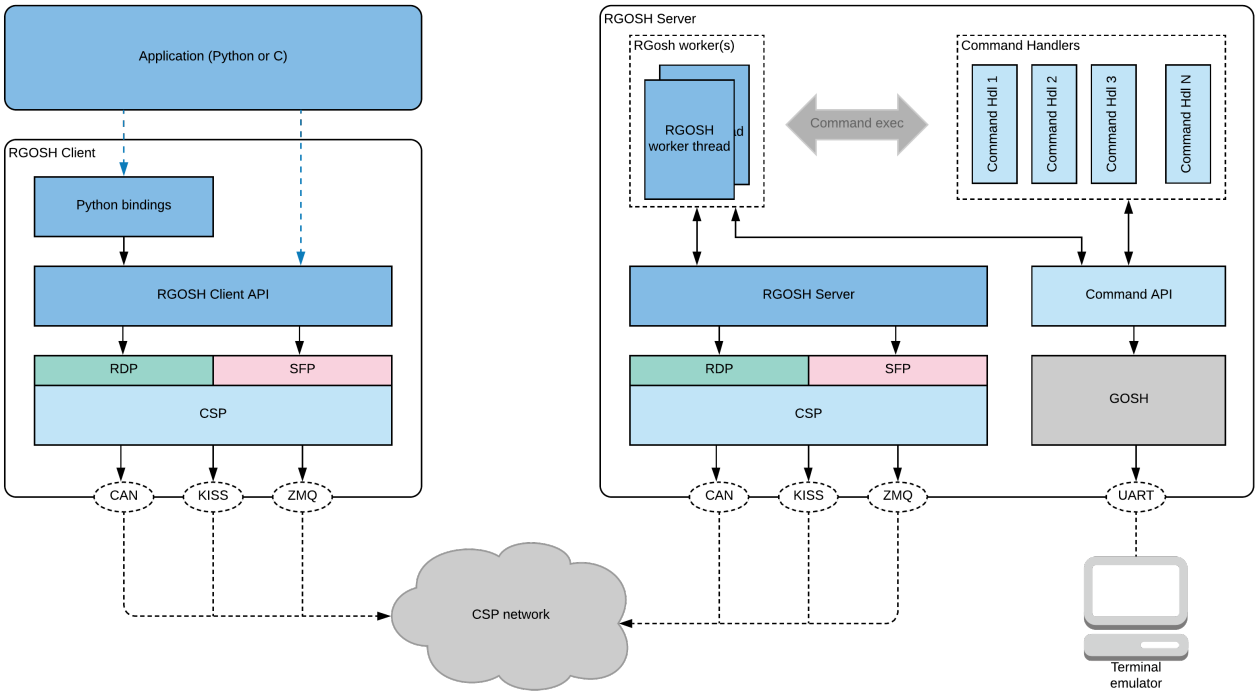


Fig. 3.2: Using Remote GOSH for commands execution & feedback

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

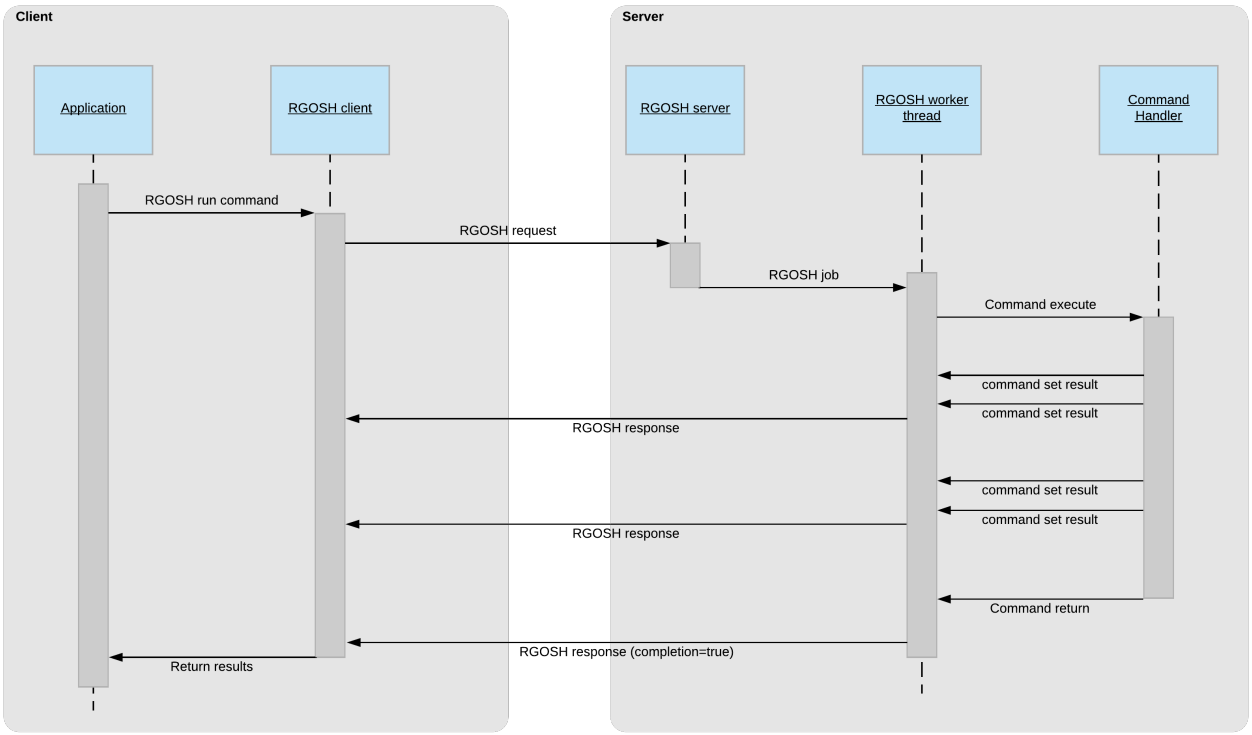


Fig. 3.3: Sequence diagram showing the use of Remote GOSH.

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 *
```

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
    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 address: 1.

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

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```c
#include <stdlib.h>
#include <string.h>

#include <gs/rgosh/rgosh.h>
#include <gs/util/log.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/router.h>


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_log_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_csp_conf_t conf;
        gs_csp_conf_get_defaults_server(&conf);

        gs_csp_init(&conf);

        // Configure routes - add default routes or use command line option
        gs_csp_rtable_load(NULL, true, true);

        // Start CSP router task
        gs_csp_router_task_start(0, GS_THREAD_PRIORITY_HIGH);
    }

    // Wait for the ZMQ & router threads to be running...
```

(continues on next page)

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
    gs_time_sleep_ms(500);

    // CSP address of the remote node
    const uint8_t RGOSH_REMOTE_NODE = 1;

    gs_error_t remote_error;
    const char * command = "clock";
    log_info("Running \"%s\" command on remote node %u", command, RGOSH_REMOTE_NODE);
    gs_error_t error = gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, command,
↪ 3000, &remote_error, &gs_rgosh_cb, NULL);
    if (error || remote_error) {
        log_error("Failed to execute command: \"%s\" on node %u, error %d, remote-error %d
↪", command, RGOSH_REMOTE_NODE, error, remote_error);
    }

    command = "log group list";
    log_info("Running \"%s\" command on remote node %u", command, RGOSH_REMOTE_NODE);
    error = gs_rgosh_run_command(RGOSH_REMOTE_NODE, GS_CSP_PORT_RGOSH, command, 3000, &
↪remote_error, &gs_rgosh_cb, NULL);
    if (error || remote_error) {
        log_error("Failed to execute command: \"%s\" on node %u, error %d, remote-error %d
↪", command, RGOSH_REMOTE_NODE, error, remote_error);
    }

    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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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

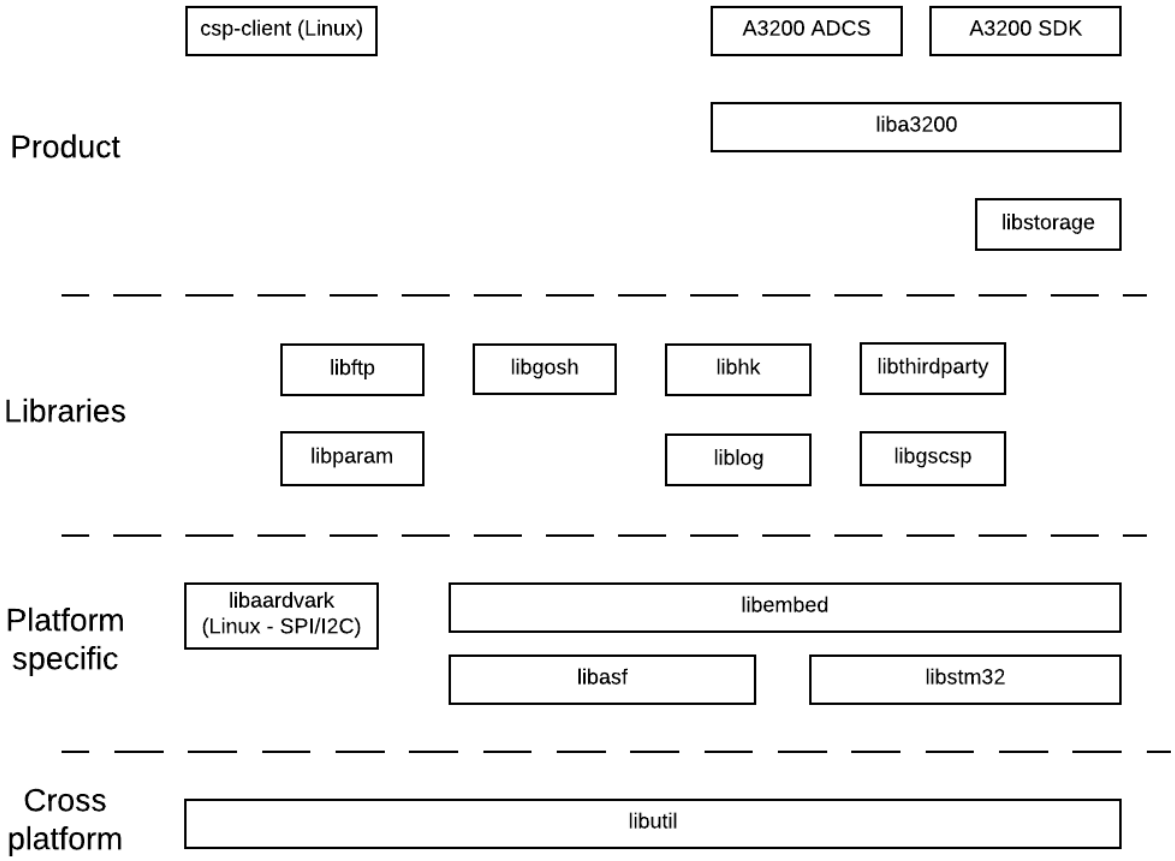
This section provides a general introduction to common concepts and terminology used in documentation and software nodules. This will make it easier to understand and navigate the software SDK's.

- *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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.



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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

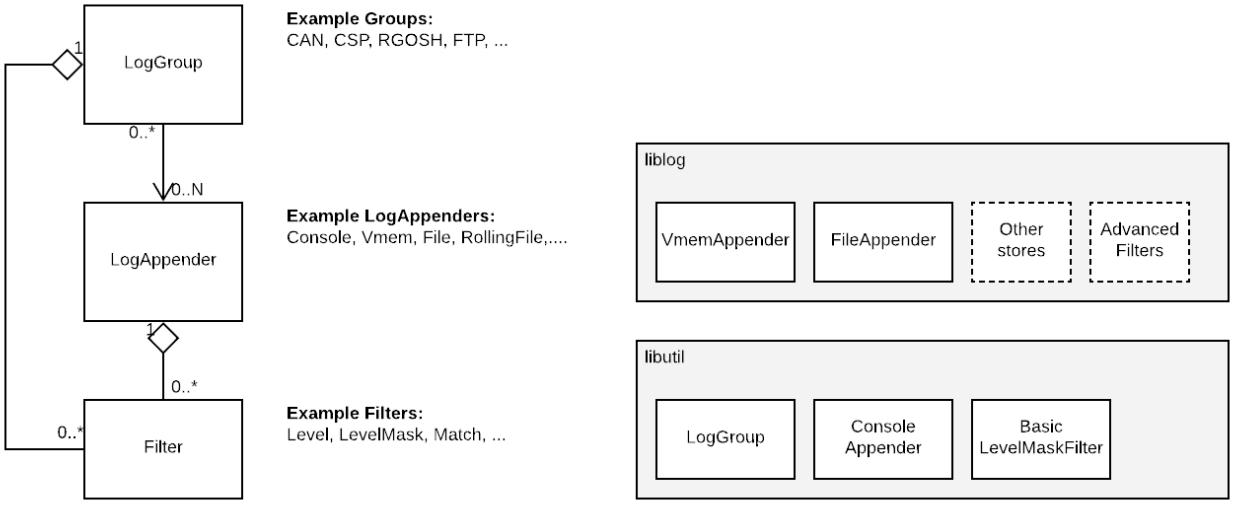
Folder	Description
include	Public header files.
include/gs/<module>	To avoid clashes with other modules and 3rd part libraries, all files are placed in scoped sub-folders.
include/gs/<module>/internal	The *internal* folder is reserved for shared files between *server* & *client*, and only present in 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).

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

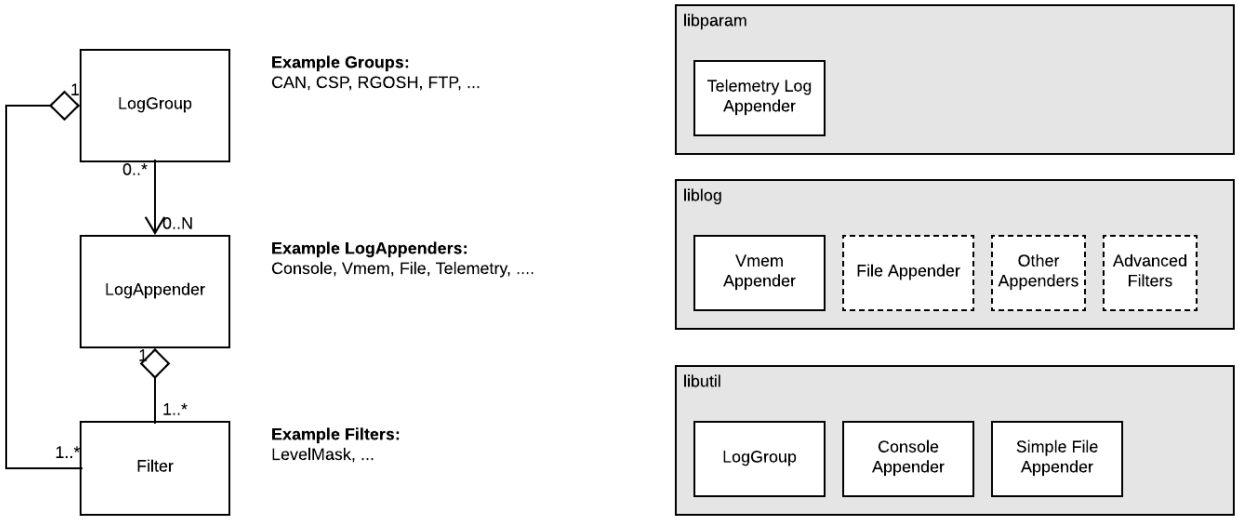
Logging hierarchy

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.



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>
#include <gs/util/vmem.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 = {
```

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
    .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);
    gs_vmem_register_commands();

    // 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.

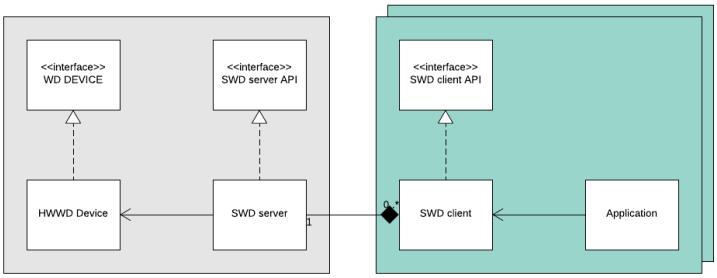


Fig. 3.4: Software Watchdog Client/Server modules

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.

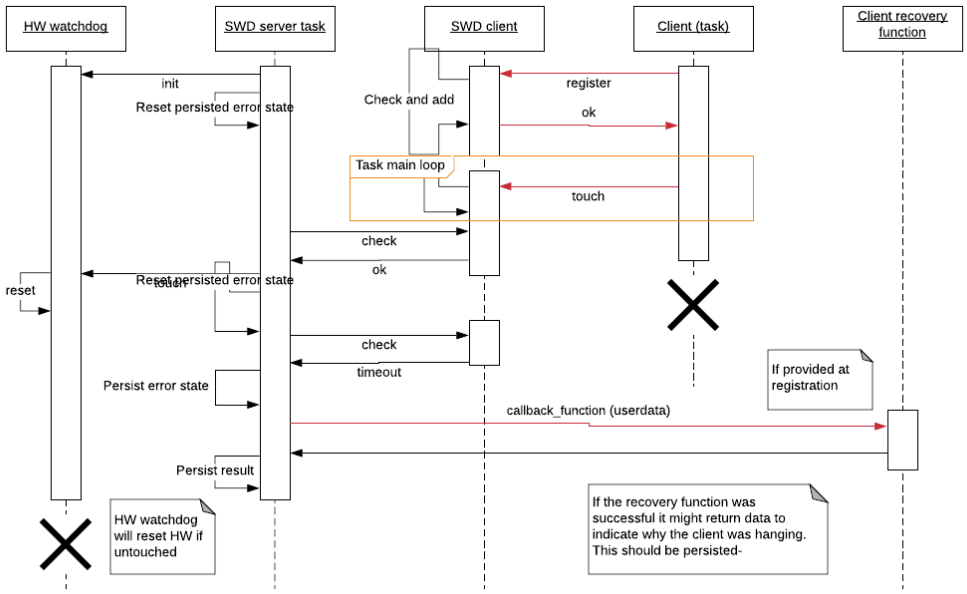


Fig. 3.5: Software Watchdog Client/Server sequence diagram

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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 3.4: 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 the first word in the list of root-commands. Commands can be divided into two groups:

- *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:

```
util.app # help
  help                Show help
  sleep               Sleep mS
  watch               Run commands at intervals (abort on key)
  watch_check         Run commands at intervals (abort on key or failure)
  clock               Get/set system clock
  exit                Exit program
  log                 Log system
  debug               Set log group mask: e|w|n|i|d|t|stand|all|off
```

vmem is an example of a command group:

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

```
util.app # vmem
Virtual memory
  read                Read from virtual memory
  write               Write to virtual memory
  lock                Lock the virtual memory
  unlock              Unlock the virtual memory
  list                Show virtual memory mappings
  info                Show virtual memory mappings + info
```

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>:

```
util.app # vmem read
Read from virtual memory
usage: read <addr> <length>
```

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 root_commands[] = {
    {
        .name="help",
        .help="Show help",
        .usage="[command]...",
        .optional_args=255,
        .handler=gs_command_cmd_help_handler,
    },
```

The next example shows how to add sub-commands to the 'vmem' command (*Command Table 'vmem'*):

First the sub-commands are defined:

```
static const gs_command_t GS_COMMAND_SUB vmem_commands[] = {
    {
        .name="read",
        .help="Read from virtual memory",
        .usage="<addr> <length>",
        .mandatory_args=2,
        .handler=gs_vmem_cmd_vmem_read_handler,
    },
    {
        .name="write",
        .help="Write to virtual memory",
        .usage="<addr> <data>",
        .mandatory_args=2,
        .handler=gs_vmem_cmd_vmem_write_handler,
    },
    {
        .name="lock",
        .help="Lock the virtual memory",
        .usage="<entry>",
        .mandatory_args=1,
        .handler=gs_vmem_cmd_vmem_lock_handler,
    },
    {
        .name="unlock",
        .help="Unlock the virtual memory",
```

(continues on next page)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

(continued from previous page)

```
        .usage="<entry>",
        .mandatory_args=1,
        .handler=gs_vmem_cmd_vmem_unlock_handler,
    },
    {
        .name="list",
        .help="Show virtual memory mappings",
        .mandatory_args=GS_COMMAND_NO_ARGS,
        .handler=gs_vmem_cmd_vmem_list_handler,
    },
    {
        .name="info",
        .help="Show virtual memory mappings + info",
        .mandatory_args=GS_COMMAND_NO_ARGS,
        .handler=gs_vmem_cmd_vmem_info_handler,
    },
};
```

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

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

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:

```
int gs_command_cmd_sleep_handler(gs_command_context_t * context)
{
    uint32_t sleep_ms;
    if (gs_string_to_uint32(context->argv[1], &sleep_ms) != GS_OK) {
        return GS_ERROR_ARG;
    }

    gs_time_sleep_ms(sleep_ms);

    return GS_OK;
}
```

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

The return code is expected to be a *gs_error_t* type, which allows the framework 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*.

3.11.6 Commands

Miscellaneous commands

A set of miscellaneous (default) commands provides basic help and simple execution of commands.

Table 3.5: Command Table 'command'

Command	
help	Show help. Arguments: `[command]`: Show command help.
sleep	Sleep mS. Arguments: `<mS>`: Number of mS to sleep.
watch	Run commands at intervals (abort on key). Arguments: `<interval mS>`: Interval between executions `<command>`: Command to execute. `[arg]`: Command argument(s)
watch_check	Run commands at intervals (abort on key/failure). Arguments: `<interval mS>`: Interval between executions `<command>`: Command to execute. `[arg]`: Command argument(s)
clock	Get/set system clock. Time is UTC. Arguments: `[<sec.nsec> | <YYYY-MM-DDTHH:MM:SSZ>]`: Time (UTC) to set.
exit	Exit program. Only on Linux systems.

log

The *log* command group provides access to the log system, e.g. change log settings.

Table 3.6: Command Table 'log'

Command	
log group list	List groups. Arguments: `[group]`: Name of group.
log group mask	Set mask: e|w|n|i|d|t|stand|all|off. Arguments: `<group[,group]>`: Name of groups. `<[+-]level[,level]>`: Level mask.
log group insert	Log message to group. Arguments: `<group>`: Name of group. `<level>`: Level to use. `<message>`: Log message, use quotes if message contains spaces.
log group color	Enable/disable color logging (stdout). Arguments: `<on>`: true for color logging, false for none color.

Continued on next page

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Table 3.6 – continued from previous page

Command	
log appender list	List appenders. Arguments: `[appender]`: Name of appender.
log appender mask	Set mask: e|w|n|i|d|t|stand|all|off. Arguments: `<appender[,appender]>`: Name of appenders. `<[+-]level[,level]>`: Level mask.
log appender hist	Show history. Arguments: `<appender>`: Name of appender. `[count]`: Max number of logs. `[pattern]`: Pattern to match.
log appender clear	Clear history. Arguments: `<appender>`: Name of appender.
debug	Set log group mask: e|w|n|i|d|t|stand|all|off. Arguments: `<group[,group]>`: Name of groups. `<[+-]level[,level]>`: Level mask.

vmem

The *vmem* command group provides access to the Virtuel Memory system. Not all prodocts uses *vmem*.

Table 3.7: Command Table 'vmem'

Command	
vmem read	Read from virtual memory. Arguments: `<addr>`: Address. `<length>`: Bytes to read.
vmem write	Write to virtual memory. Arguments: `<addr>`: Address. `<data>`: Data to write, e.g. 1c01 to write 2 bytes 0x1c 0x01.
vmem lock	Lock the virtual memory. Arguments: `<entry>`: Name vmem slot.
vmem unlock	Unlock the virtual memory. Arguments: `<entry>`: Name vmem slot.
vmem list	Show virtual memory mappings.
vmem info	Show virtual memory mappings + info.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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.

```
#!/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)
```

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

5. Appendix

5.1 Software Changelogs

5.1.1 Product Interface Application

2.7.1 (2020-08-27)

- Improvement: Updated nanocam2_client library.

2.7.0 (2020-05-15)

- Improvement: Updated libraries, libgscsp 2.x -> 3.x.
- Improvement: Documentation includes details on building experimental Python bindings

2.6.1 (2019-10-17)

- Improvement: Updated libraries.

2.5.1 (2019-10-04)

- Improvement: Updated nanocam2_client library.

2.4.1 (2019-07-03)

- Improvement: Updated libraries.

2.3.2 (2019-06-12)

- Improvement: Updated libraries.

2.3.1 (2019-05-20)

- Improvement: Updated libraries.

2.2.1 (2019-04-10)

- Improvement: Updated libraries.

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

6.0.2 (2020-03-26)

- Improvement: Updated dependencies
- Bug: fixed array size in adcs_telem3_152.json for UKF from 15 to 14.

6.0.1 (2019-12-12)

- Improvement: Reintroduced stylecheck

6.0.0 (2019-11-01)

- Breaking: Added new parameters for Startracker implementation in the UKF

5.1.0 (2019-03-28)

- Improvement: Added Variable Frequency (VARF) as a parameter (en_gps_varf)
- Improvement: Split pwm_deadtime into mtq_decay_t and mag_sample_t. This allows for tuning based on the placement and number of magnetometers.

5.0.3 (2019-03-01)

- Bug: Torquerduty was not reported in body frame

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)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

5.0.5 (2020-08-26)

- Improvement: added RAW description in API section of documentation

5.0.4 (2020-02-21)

- Improvement: updated documentation for gain-global usage

5.0.3 (2020-02-19)

- Bug: cam snap missing width and height in help text

5.0.2 (2019-10-28)

- Bug: cam peek, use smallest value of CSP_CMP_PEEK_MAX_LEN and GS_CSP_DEFAULT_MAX_PAYLOAD_SIZE

5.0.1 (2019-10-25)

- Improvement: added retry count in cam peek command
- Improvement: updated build scripts and dependencies
- Improvement: remove wscript option, not needed anymore since commands must be registered.
- Improvement: added command register function to public header file.
- Improvement: replaced sscanf / command_args(ctx), with gs_string_to_xxx()
- Improvement: use mandatory_args and/or optional_args, instead of checking for minimum args in functions
- Improvement: use new GS_CSP_DEFAULT_MAX_PAYLOAD_SIZE (libgscsp), instead of CSP_CMP_PEEK_MAX_LEN
- Bug: cam peek now only enabled for linux as it does not work for freertos
- Improvement: register write function, now reads back the value and print it
- Improvement: libgscsp updated to 2.7.2

5.0.0 (2019-09-19)

- Breaking: switched to libparam 4.7 (new param layout)
- Breaking: renamed kiss-interface to kiss-device in table 0

4.16.6 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

4.16.5 (2019-05-20)

- Improvement: Updated command help text.

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)

3.3.7 (2019-07-03)

- Feature: Added gatoss_hk_p() for getting housekeeping data from a specific node.
- Improvement: Changed command/gosh definitions to const.

3.3.6 (2019-05-20)

- Improvement: Fixed ‘const’ warnings (if used with libutil 3.9.1)
- Improvement: Updated command help text.

3.3.5 (2019-02-27)

- Improvement: Updated dependencies

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)

3.12.1 (2020-03-19)

- Improvement: Updated buildtools.

3.12.0 (2019-09-20)

- Improvement: Add dummy payload to cmd_ax100_gndwdt_clear
- Feature: Add kiss_baud parameter to table 0
- Bug: Drop bcn_interval and bcn_holdoff parameters

3.11.1 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

3.11.0 (2019-03-05)

- Bug: Improved HMAC authentication. Legacy mode available through configuration.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

3.9.12 (2018-11-22)

- Feature: Moved client into own repository.

NanoPower BPX (nanopower-bpx_client)

3.0.7 (2020-03-20)

- Improvement: Modernised used commands structs.
- Improvement: Updated dependencies.

3.0.6 (2019-07-03)

- Feature: Added bpx_hk_get_p() for getting housekeeping data from a specific node.
- Improvement: Changed command/gosh definitions to const.

3.0.5 (2019-05-20)

- Improvement: Updated command help text.

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)

2.22.9 (2020-03-19)

- Improvement: Updated epsslave documentation
- Improvement: Removed unused options in wscript

2.22.8 (2019-10-18)

- Improvement: Updated epsslave

2.22.7 (2019-07-03)

- Feature: Added eps_hk_get_p() for getting housekeeping data from a specific node.
- Improvement: Changed command/gosh definitions to const.

2.22.6 (2019-05-20)

- Improvement: Updated command help text.

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”

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

1.2.18 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

1.2.17 (2019-05-20)

- Improvement: Updated command help text.

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)

2.2.9 (2020-03-20)

- Improvement: Split table definitions for cal and conf into separate files to lower memory footprint

2.2.8 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

2.2.7 (2019-05-20)

- Improvement: Updated command help text.

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)

1.2.21 (2020-03-20)

- Improvement: Split table definitions for cal and conf into separate files to lower memory footprint

1.2.20 (2019-07-03)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- Improvement: Changed command/gosh definitions to const.

1.2.19 (2019-05-20)

- Improvement: Updated command help text.

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)

2.0.2 (2019-12-02)

- Bug: power status command with more than two parameters would behave as power off (P60-100)
- Feature: Added python bindings

2.0.1 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

2.0.0 (2019-05-20)

- Breaking: Power interface commands prefixed with `p60_`
- Bug: rgssb port changed from 15 to 16

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)

1.3.1 (2020-03-18)

- Improvement: Increased MTU size on I2C to 4096 bytes.
- Breaking: Only build on 64 bit linux platform - there is no 32 bit version.

1.2.1 (2019-04-09)

- Improvement: Updated dependencies.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

Gomspace CSP extension (libgscsp)

3.0.1 (2020-05-15)

- Breaking: Updated to libcsp v1.6 - see libcsp CHANGELOG for details. APIs are primarily broken in regards to csp_init() and interfaces impl.

2.7.4 (2020-03-18)

- Improvement: Reduced GS_CSP_DEFAULT_MAX_PAYLOAD_SIZE to make room for the RDP header.

2.7.3 (2019-11-13)

- Improvement: Updated libcsp: Always install all header files.

2.7.2 (2019-10-16)

- Feature: Added default value for max CSP packet payload, GS_CSP_DEFAULT_MAX_PAYLOAD_SIZE.
- Feature: Added 'count' option on ping command, pings destination 'count' times and shows average ping time.

2.7.1 (2019-07-03)

- Feature: Added support for ping without CRC32.
- Improvement: Changed formatting of 'uptime' command.
- Improvement: Changed CAN logging to match CAN logging from libembed.
- Improvement: Updated libcsp: CAN driver: get time correctly depending on context - ISR or task.

2.6.2 (2019-05-16)

- Improvement: Include CAN support in .deb package.

2.6.1 (2019-04-08)

- Improvement: Support longer device names on command line.

2.5.1 (2019-03-10)

- Bug: Made CAN driver use the chosen CAN device in both initialization and TX and RX.
- Feature: Added support for multiple csp_zmq interfaces in process.
- Bug: libcsp, fixed bug in linux usart driver, didn't detect read error as expected.
- Breaking: libcsp, removed static "csp_if_zmqhub" interface.
- Improvement: added error handling in usart driver (linux), fail if device cannot be opened.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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 (GomSpace)

- Use GomSpace log system instead of libcsp.
- ZMQHUB, transfer 'via' information between zmqproxies.

libcsp 1.6, 16-04-2020

- Updated documentation.
- Updated Python bindings, removed Python2 support.
- Removed timeout for send (internally only) and interface tx functions - the timeout was only used on I2C interface.
- Updated example code to a single csp_server_client.c implementation.
- Renamed (scoped) clock_set_time()/clock_get_time() to csp_clock_set_time()/csp_clock_get_time()
- Changed csp_sys_reboot()/csp_sys_shutdown() to use callbacks, default POSIX impl. in csp/arch/posix/csp_system.h.
- Added support for timestamps in logs by setting CSP_DEBUG_TIMESTAMP (uses csp_clock_get_time()).
- Renamed mac to via (structs, functions, examples and documentation)
- RDP: Ensure connection is kept in CLOSE_WAIT for period of time. In some cases, the connection would switch to CLOSED immediately.
- RDP: Fixed connection leak, if a RST segment was received on a closed connecton.
- RDP: Ensure connection is closed from both userspace and protocol, before closing completely (preventing undetermined behaviour).
- RDP: Added support for fast close of a connection (skipping the CLOSE_WAIT period), but only if both ends agree on close.
- RDP: Fixed issue "Possible bug in RDP TX timeout" (#109), see issue on github for further details.
- Added Travis-CI support on Github, build: Linux, Mac and Windows.
- Changed #ifdef/ifndef to single #if in order to support forced disabling and alignment with log macro's.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- Updated thread API, documentation, aligned implementation
- **Refactored all CSP interfaces.**
  - Accept csp_route_t, instead of csp_iface_t.
  - **No static members -> multiple interfaces of all types.**
    - Added csp_iface_t.interface_data for interface data.
    - Added csp_iface_t.driver_data for driver data (unknown to interface level).
  - Check for buffer overrun
  - Set MTU if not already set, and it make sense on the respective interface.
  - Driver Tx function is now a callback, must be set by the application (in the interface data).
- csp_packet_t (and other structs) are no longer packed (increases performance), padding is increased from 8->10 bytes for alignment.
- **Refactored rtable CIDR/static implementation:**
  - Use same format for storing table (text).
  - Added csp_route_t to hold route entry, e.g. interface and via.
  - Replaced csp_rtable_find_iface()/csp_rtable_find_mac() with csp_rtable_find_route()
- Added csp_get_memfree()/csp_get_buf_free()/csp_get_uptime(), which returns an error code.
- Improved logging, check level before doing the actual call. Added support for external log macros.
- Removed unused defines/functions: csp_conn_lock, csp_conn_unlock, CSP_PROMISC
- Replaced CSP_MAX_DELAY and CSP_INFINITY with CSP_MAX_TIMEOUT (same value).
- Added csp_calloc(), csp/arch/csp_malloc.h
- Added csp_buffer_data_size(), csp/csp_buffer.h
- Updated header documentation (doxygen).
- Restructured header includes.
- api: Added const for read-only pointers.
- api: Changed a number of uint8_t pointers to void to avoid cast.

libcsp 1.5-internal, 21-08-2018

- Internal release.
- New python bindings with examples and python3 support.
- Introduced csp_conf_t, which replaces a number of compile options. This has broken csp_init().
- Refactored CAN interface.
- Bug fixes.

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)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

3.8.0 (2020-03-18)

- Feature: Added fast_load_with_unique_keys option to fp load (BW compatible)

3.7.1 (2019-12-20)

- Feature: Added Python bindings.
- Feature: Commands in the flightplan can now be 119 characters.
- Feature: A flightplan can now load a new flightplan.
- Improvement: Updated dependencies.

3.6.3 (2019-10-16)

- Improvement: Use CSP port definition from libgscsp.

3.6.2 (2019-05-19)

- Improvement: Fixed ‘const’ warnings (if used with libutil 3.9.1)

3.6.1 (2019-04-09)

- Improvement: Updated dependencies.

3.5.1 (2019-03-26)

- Bug: Fixed bug in listing FP entries.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- Improvement: Improved command (GOSH) argument and error handling.
- Improvement: Updated documentation.

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)

5.2.2 (2020-03-20)

- Feature: ftp transfer is now done to a temporary file, and renamed on successful transfer.
- Bug: fixed percent complete calculation (Nan error) on small file size.

5.2.1 (2019-10-16)

- Improvement: Updated API documentation and command(s) to use common term ‘checksum’.
- Improvement: Replaced local time-to-string function with gs_time_to_string() from libutil.

5.1.2 (2019-05-19)

- Improvement: Fixed ‘const’ warnings (if used with libutil 3.9.1)

5.1.1 (2019-04-09)

- Improvement: Updated dependencies.

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)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

3.5.3 (2019-12-03)

- Improvement: Updated dependencies.

3.5.2 (2019-05-16)

- Improvement: Updated dependencies.

3.5.1 (2019-04-09)

- Improvement: Added gs_gosh_remote_shell_stdio() to improve testability.

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)

4.4.3 (2020-03-20)

- Bug: Fixed issue with `gssb scanbus` command, sometimes address 0 was printed - but never scanned.

4.4.2 (2019-07-03)

- Bug: Fixed parameter parsing of `rgssb node` command.

4.4.1 (2019-05-19)

- Improvement: Added optional port parameter in rgssb node command.

4.3.1 (2019-04-10)

- Improvement: Made ant6 and ar6 autodeploy algorithms burn once even when status shows released.

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

4.2.1 (2018-05-30)

4.0.1 (2018-03-01)

Housekeeping System (libhk_client)

4.5.2 (2019-10-16)

- Feature: In parser, added support for parsing legacy (GOMX4) beacon types.
- Improvement: In parser, adding info log line for each beacon received.

4.5.1 (2019-07-01)

- Breaking: In parser, parameter callback now returns a list of value,index tuples instead of a list of values.
- Feature: In parser, support for having an array_index value of a single index instead of a range (LIBHK-117).

4.4.1 (2019-05-20)

- Improvement: In parser, beacons will now be offset if older than 10 years.
- Improvement: In parser, user can now add beacon time offsets + new offsets will be 'written' back to user.
- Improvement: In parser, satellite id is now also used when looking up beacon time offsets.
- Feature: It is now possible to tweak burst_idle_time and burst_size in commands and in the API.

4.3.1 (2019-04-10)

- Improvement: Updated dependencies.

4.2.3 (2019-04-10)

- Improvement: Added new option to examples

4.2.2 (2019-04-08)

- Bug: fixing request of file-based telemetry.
- Bug: fixing issue when using t0=0 in requests.

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)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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

Nano Protobuf (libnanopb)

1.3.1 (2020-01-22)

- Improvement: Hardcoded PB_FIELD_16BIT in public pb.h, instead of hardcoding it through Waf (solves bitbake issue).
- Improvement: Removed GomSpace introduced 'nanopb' scope of header files - auto-generated protobuf code doesn't use this scope (bitbake).

1.2.2 (2019-10-16)

- Feature: Added support for CPP output.

1.2.1 (2019-04-09)

- Improvement: Updated dependencies.

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)

4.8.1 (2020-03-19)

- Feature: Added gs_rparam_get_array() for getting an array of parameters.
- Breaking: Python/rparam, removed automatic load of tables on module import/load.
- Feature: Python/rparam, added rparam_load_tables_from_dir().

4.7.3 (2019-12-03)

- Improvement: param_gen, improved formatting of 'validation' - 'bits' and 'discrete'.
- Improvement: param_gen: added support for new line (n) in parameter description.

4.7.2 (2019-10-16)

- Improvement: Clear rparam query and cached table data on rparam init/download.

4.7.1 (2019-05-16)

- Improvement: Updated libutil dependency and fixed warning due to const improvement in libutil.

4.6.1 (2019-04-09)

- Bug: fixed issue with rparam download (failed to save table if working folder > 100 characters).
- Improvement: Updated dependencies.

4.5.4 (2019-02-26)

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- 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)

1.2.3 (2020-03-24)

- Bug: Initialized error code, that may not be written to in case of framework error.

1.2.2 (2020-03-18)

- Improvement: Updated documentation and example code.

1.2.1 (2020-01-23)

- Bug: Fixed formatting issue in 'rgosh server' command.
- Improvement: Updated dependencies.

1.1.3 (2019-10-16)

- Bug: Reduced max CSP package size from 226 to 180 bytes, so it can be send via the AX100.

1.1.2 (2019-05-19)

- Improvement: Updated dependencies.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

1.1.1 (2019-04-10)

- Improvement: Increased client timeout and improved debug logs.

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)

3.10.2 (2020-03-18)

- Improvement: Log: Do not call gs_clock_get_time() from ISR context.

3.10.1 (2020-01-23)

- Feature: User space interrupt support for linux sysfs gpio driver.
- Feature: Linux - support for native I2C and SPI driver.
- Improvement: Linux - improved gs_time_sleep_ns() by compensating for signals.

3.9.3 (2019-10-16)

- Bug: Fixed check in gs_vmem_cpy() for last entry, checked size - not size (as other VMEM functions).
- Feature: Made gs_lock API public (moved header file to public scope).
- Feature: Added vmem test interface used for production checkout.
- Improvement: log appender hist: added pattern match support.
- Bug: gs_string_match() fixed wildcard bug, preventing a valid match.
- Improvement: Disable stdio buffering from within console thread, so it works with newlib reentrant.
- Feature: Added prototype for gs_stdio_put_direct() and linux implementation.

3.9.2 (2019-07-03)

- Feature: Added gs_time_to_string() API.

3.9.1 (2019-05-16)

- Improvement: Changed strings in command context to const (prevent modification).

3.8.1 (2019-04-08)

- Feature: Added GS_PATH_MAX, GS_PLATFORM_64, gs_snprintf().
- Improvement: Changed logs from ISR to print to stderr, so these can be handled differently.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- Improvement: Aligned help text for log groups and appenders.
- Breaking: Added return value to log appender interface.
- Feature: Added log appender ‘clear’ command.
- Breaking: Extended vmem interface with a check() function. Changed “size” formatting in vmem list.
- Feature: Added “vmem info” command.

3.7.2 (2019-03-10)

- Feature: Added support for setting log levels on command line (linux only).

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.

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

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)

2.11.3 (2020-04-21)

- Improvement: Updated internal tools.
- Improvement: All document types now include the signoff box with revision details

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

2.11.2 (2020-03-18)

- Improvement: Updated build containers.

2.11.1 (2020-03-10)

- Improvement: Updated internal tools (bitnake builds).
- Improvement: Updated build containers.
- Improvement: Changed default Linux compiler and unit tests to Ubuntu 18.04, GCC 7.3.
- Breaking: Removed scripts for generating Eclipse files.

2.10.4 (2020-01-21)

- Improvement: Updated internal tools (bitbake builds).
- Improvement: Updated build containers.
- Feature: Added support for STM32L431 chip.

2.10.3 (2019-12-03)

- Improvement: Updated build containers.
- Breaking: Changed default naming of dist. tar.gz files to use GiT repo name.

2.10.2 (2019-11-13)

- Improvement: Updated internal tools.

2.10.1 (2019-10-16)

- Improvement: Updated build containers, includes new toolchain for AVR32/A3200.
- Improvement: Updated Waf from 2.0.17 to 2.0.18.

2.9.1 (2019-07-03)

- Breaking: Renamed linux build part “x86_64-gcc-7.3” to “x86_64-gcc-7”.
- Improvement: Updated build containers, changed GCC for part=”x86_64-gcc-7” from 7.3.0 to 7.4.0.
- Improvement: Updated Waf from version 2.0.9 to 2.0.17.
- Feature: Added support for changelog tag “Limitation”.

2.8.5 (2019-06-11)

- Improvement: Changed disclaimer (in document templates) to regular chapter.

2.8.4 (2019-05-24)

- Feature: Changed order of reading version information, try VERSION file before GiT.

2.8.3 (2019-05-16)

- Improvement: If no version information available, try read file VERSION.

2.8.2 (2019-04-10)

- Improvement: Internal changes

2.8.1 (2019-04-08)

- Improvement: Internal changes to documentation

2.8.0 (2019-04-03)

- Improvement: Adding support for building to Ubuntu 18.04
- Improvement: Adding support for building unit tests in 32 bit

2.7.0 (2019-02-28)

- Improvement: Add support for BibTeX in documentation

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

- Improvement: Adding support for python unit tests
- Improvement: Updated documentation to be in line with newest template

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

© 2020 GomSpace A/S.
All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.

6. Disclaimer

Information contained in this document is up-to-date and correct as at the date of issue. As GomSpace A/S cannot control or anticipate the conditions under which this information may be used, each user should review the information in specific context of the planned use. To the maximum extent permitted by law, GomSpace A/S will not be responsible for damages of any nature resulting from the use or reliance upon the information contained in this document. No express or implied warranties are given other than those implied mandatory by law.

© 2020 GomSpace A/S.

All printed copies, and all electronic copies and versions except the one accessible on the GomSpace A/S server, are considered uncontrolled copies used for reference only.