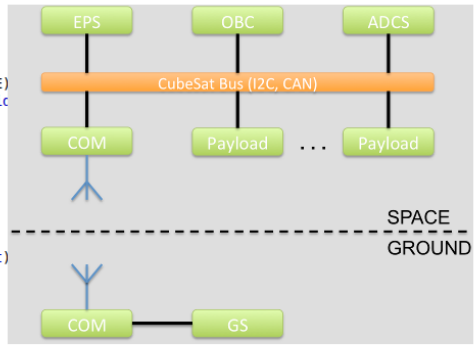
```c
int csp_send(csp_conn_t * conn, csp_packet_t * packet, uint32_t timeout) {

    int ret;

    if ((conn == NULL) || (packet == NULL) || (conn->state != CONN_OPEN)) {
        csp_log_error("Invalid call to csp_send\r\n");
        return 0;
    }

#ifdef CSP_USE_RDP
    if (conn->idout.flags & CSP_FRDP) {
        if (csp_rdp_send(conn, packet, timeout) != CSP_ERR_NONE) {
            csp_iface_t * ifout = csp_rtable_find_iface(conn->ic
            if (ifout != NULL)
                ifout->tx_error++;
            csp_log_warn("RDP send failed\r\n!");
            return 0;
        }
    }
#endif

    csp_iface_t * ifout = csp_rtable_find_iface(conn->idout.dst)
    ret = csp_send_direct(conn->idout, packet, ifout, timeout);

    return (ret == CSP_ERR_NONE) ? 1 : 0;

}

int csp_send_prio(uint8_t prio, csp_conn_t * conn, csp_packet_t * packet, uint32_t timeout) {
    conn->idout.pri = prio;
    return csp_send(conn, packet, timeout);
}

int csp_transaction_persistent(csp_conn_t * conn, uint32_t timeout, void * outbuf, int outlen, void * inbuf, int inlen) {

    int size = (inlen > outlen) ? inlen : outlen;
    csp_packet_t * packet = csp_buffer_get(size);
    if (packet == NULL)
        return 0;

    /* Copy the request */
    if (outlen > 0 && outbuf != NULL)
```



NanoMind

A3200 Command and Management SDK

Manual

Software Development Kit

Release 2.12.0

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

Product name: NanoMind A3200 Command and Management SDK

Reference: 1006899

Revision: 2.12.0

Date: 12 June 2020

Confidentiality Notice

Table of Contents

1. A3200 SDK

1.1 Introduction

The A3200 Command & Management SDK (A3200 SDK) software package provides a set of advanced and feature rich modules that enable fast and efficient development of a rich featured OBC applications for NanoMind A3200 Board. The included modules provides among other the following features:

- Board Support Package, including FreeRTOS and toolchain for A3200 development.
- CSP Communication, including drivers for I2C, CAN and USART (KISS)
- Parameter System for Configuration and Status
- Console interface for test and control
- GomSpace Sensor Bus (GSSB) interface for controlling GSSB devices
- FTP Service for upload and download of data.
- File system for Flash Storage

1.1.1 Getting started

Prerequisites

In order to develop software for the A3200 NanoMind, a toolchain (compiler and tools) must be installed on your Linux PC. Please see the chapter *A3200 Toolchain* in the manual *NanoMind A3200 Board Support Package* (*[1]*) for details on installation.

Unpacking A3200 SDK

The A3200 SDK is shipped as a single archive, which includes all the source code, build scripts, documentation (this document), etc.

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

1. Create a workspace folder for the source code, e.g. ~/workspace:

```
$ mkdir ~/workspace
```

2. Copy the archive to the workspace folder:

```
$ cp gs-sw-nanomind-a3200-command-and-management-sdk-<version>.tar.gz ~/workspace
```

3. Change directory to the workspace folder:

```
$ cd ~/workspace
```

4. Unpack the archive:

```
$ tar xvf gs-sw-nanomind-a3200-command-and-management-sdk-<version>.tar.gz
```

You now have the source code unpacked in: *~/workspace/gs-sw-nanomind-a3200-command-and-management-sdk-<version>*.

Building A3200 SDK

Use the following steps to build the A3200 SDK.

Building the first time:

1. Change directory to the A3200 SDK folder:

```
$ cd ~/workspace/gs-sw-nanomind-a3200-command-and-management-sdk-<version>
```

2. Bootstrap buildtools:

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

3. Build using Waf:

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

Subsequent building:

1. Change directory to the A3200 SDK folder:

```
$ cd ~/workspace/gs-sw-nanomind-a3200-command-and-management-sdk-<version>
```

2. Build using Waf:

```
$ ./waf
```

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

Programming NanoMind A3200

See the chapter *A3200 Toolchain* in the manual *NanoMind A3200 Board Support Package* (*[1]*) for connecting the programmer.

Programming the first time:

1. Change directory to the A3200 SDK folder:

```
$ cd ~/workspace/gs-sw-nanomind-a3200-command-and-management-sdk-<version>
```

2. Program A3200 AVR32 FLASH:

```
$ ./waf program
```

This will write the A3200 image to the AVR32 internal program FLASH and then reset the A3200 to start the new image.

1.2 Software

This chapter describes the source code provided in the A3200 SDK application.

The main part of the software handles configuration (bootstrapping) of the system, e.g., setup Parameters stores, starting tasks, registering commands, which are provided by a number libraries (located in the ./lib folder). For simplicity, the majority of the libraries compiles all features/code and the linker will throw away all un-referenced/un-used code (controlled through compile and linker options).

The A3200 SDK comes with a number source files in the *./src* folder, which compiles into a working OBC with a file system and a number of basic services - ready for adding mission specific software.

- *main.c*: basic initialisation and setup of Parameter stores (none-volatile storage). Mission specific VMEM mappings and Parameter stores should be added here.

- *mission_init.c* mission initialisation, starts services and other tasks. Mission specific code hooks should be added here.
- *scheduler.c* generic function scheduler.
- *param*: Parameter store and table definitions.
  - *a3200_sdk_store.c* layout of the A3200 SDK Parameter store, e.g. where the *scheduler* table is stored.
  - *adsb_hk.c* telemetry table, see *Parameter Table 'adsb_hk'* details.
  - *bpx_hk.c* telemetry table, see *Parameter Table 'bpx_hk'* details.
  - *eps_hk.c* telemetry table, see *Parameter Table 'eps_hk'* details.
- *hk*: Housekeeping - collecting telemetry data from nodes, not supporting *rparam* protocol.
  - *adsb_hk.c*
  - *bpx_hk.c*
  - *eps_hk.c*

1.2.1 main.c

This file contains the `main()` function, where the NanoMind A3200 starts its execution after power-on or reset:

```
int main()
{
    // Initialize: clock, watchdog, external memory, malloc
    gs_a3200_init();

    // Early log initialize and register of the console log appender (sends log to stdout)
    gs_log_init(true);

    // Spawn a task to complete the initialization and launch application/mission tasks␣
↪through hooks.
    gs_a3200_run(&hooks);

    // Will never get here, as gs_a3200_run() never returns
    gs_sys_reset(GS_SYS_BOOT_UNKNOWN);
}
```

The function `gs_a3200_init()` makes the initial hardware initialisation, e.g. memory, watchdog. Once `gs_a3200_init()` completes, it is safe to allocate memory (e.g. `malloc()`). At this point, `printf()` will not work, as the UART hasn't been configured yet.

The function `gs_a3200_run(&hooks)` performs the rest of the system initialisation, and invokes the configured hooks. All hooks are called from a FreeRTOS task, so it is safe to create new tasks and in general use FreeRTOS.

The function `gs_a3200_run(&hooks)` supports a number of hooks, which is used to customize the system:

```
static const gs_a3200_hooks_t hooks = {
    // hook called when liba3200 want to configure VMEM and the parameter stores (where␣
↪to load parameter tables from)
    .init_vmem_param = param_init_stores,
    // hook called when the basic boot process is completed, just before a task is␣
↪spawned to mount the file system
    .init_complete = mission_init,
    // hook called when the file system is mounted and the boot process is completed.
    .fs_mounted = mission_init_fs_mounted,
};
```

If more fine grained control of the boot sequence is required, please see *Add additional hooks* for details on changing the default boot behaviour.

1.2.2 mission_init.c

This file contains a number of function hooks, called from either the boot process or the *scheduler.c*

The `mission_init()` is called during the boot process, at a point where everything is initialised - except the file system. All tasks/services that can function without a file system, should be placed here.

```
void mission_init(void)
{
    log_notice("Welcome to nanomind, model: %s, revision: %s", csp_get_model(), csp_get_
↪revision());

    // Launch server task for handling CSP service requests
    mission_server_init();

    // Start time synchronization service
    gs_a3200_timesync();

    // Register command "only" clients
    gs_adcs_register_commands();
    extern void cmd_ax100_setup(void);
    cmd_ax100_setup();
    extern void cmd_p60pdu_setup(void);
    cmd_p60pdu_setup();
    extern void cmd_p60dock_setup(void);
    cmd_p60dock_setup();
    extern void cmd_p60pdu_setup(void);
    cmd_p60pdu_setup();
    extern void cmd_p60acu_setup(void);
    cmd_p60acu_setup();
    extern void cmd_eps_setup(void);
    cmd_eps_setup();
    extern void cmd_bpx_setup(void);
    cmd_bpx_setup();
    extern void gatoss_cmd_setup(void);
    gatoss_cmd_setup();
    extern void nanocam_register_commands(void);
    nanocam_register_commands();
    extern void soft_register_commands(void);
    soft_register_commands();
    extern void cmd_power_if_setup(void);
    cmd_power_if_setup();
    gs_rgssb_register_commands();
    gs_gssb_register_commands();

    // Start scheduler
    scheduler_init(10);
}
```

The `mission_init_fs_mounted()` is called after the file system is mounted, and it the last callback from the boot process. The function will be called even if the initialisation of the file system fails (times out).

```
void mission_init_fs_mounted(void)
{
    // Initialize Flight Planner (requires missionlib)
    mission_fp_init();

    // Initialize House Keeping (requires missionlib)
```

(continues on next page)

(continued from previous page)

```
    mission_hk_init();

    // Initialize gscript/rgosh services
    mission_rgosh_init();
}
```

The `mission_schedule()` is called by the generic scheduler task, whenever there is a change to the scheduler table, see *Parameter Table 'scheduler'*. The function must ONLY be called by the scheduler task itself, as the scheduling isn't thread-safe.

```
void mission_schedule(void)
{
    extern void adsb_hk_schedule(void);
    adsb_hk_schedule();
    extern void bpx_hk_schedule(void);
    bpx_hk_schedule();
    extern void eps_hk_schedule(void);
    eps_hk_schedule();
}
```

1.2.3 scheduler.c

Generic scheduler for scheduling none time critical jobs, such as collecting housekeeping data from nodes not supporting the *rparam* protocol.

The scheduler calls schduled functions at specific intervals. Because the scheduler runs in a single task/thread, scheduling will be impacted by how long the jobs actually takes, e.g. longer time due to errors. The scheduler runs at low priority, to limit impact on more time critical activities.

The scheduler is configured through *Parameter Table 'scheduler'*. This table can be extended with more parameters for other jobs, or the configuration can be placed in different tables. If configuration is placed in different tables, the function `scheduler_reload()` can be called to trigger the scheduler to reload.

The scheduler calls `mission_schedule()` to (re)load jobs. Jobs can be enabled/disabled/modified through `scheduler_modify()`.

Note: `scheduler_modify()` must only be called from the scheduler task. Place registration functions in `mission_schedule()`.

Housekeeping

The following nodes do not support the 'rparam' protocol, and the *Housekeeping (libhk)* system can therefore not fetch data directly from these nodes. To address this their respective telemetry data is collected by the OBC in scheduled functions updating local tables from which the *Housekeeping (libhk)* can collect data.

- *NanoPower P31u (nanopower_client)*, data collected by *./src/hk/eps_hk.c* and placed in *Parameter Table 'eps_hk'*.
- *NanoPower BPX (nanopower-bpx_client)*, data collected by *./src/hk/bpx_hk.c* and placed in *Parameter Table 'bpx_hk'*.
- *NanoCom ADS-B (gatoss-uc_client)*, data collected by *./src/hk/adsb_hk.c* and placed in *Parameter Table 'adsb_hk'*.

1.2.4 Add additional hooks

For a more fine grained control of the boot sequence, additional hooks can easily be added.

The function `gs_a3200_sdk_init_task()`, in *src/sdk/sdk_init.c*, runs the main boot sequence and invokes the hooks by calling:

```
GS_A3200_CALL_HOOK(init_complete);
```

To add another hook to the boot sequence, add the function callback to *gs_a3200_hooks_t* and call it at the desired boot step - most likely in *src/sdk/sdk_init.c*.

1.3 Known Limitations

1.3.1 AVR32 toolchain

The toolchain has been upgraded to a GomSpace patched version (3.4.2_gs1), which solves the following issues when running in a multi-threaded environment;

- Enabling FreeRETOS configUSE_NEWLIB_REENTRANT causes a crash when a task/thread terminates. This is due to a bug in newlib 1.16 (part of the toolchain). configUSE_NEWLIB_REENTRANT allows formatting of float/double from multiple threads, without crashing.
- Missing file locking, when accessing files from mulitple threads. The patched toolchain now supports retargetable locks in newlib, and the locks has been implemented in libembed.

The new toolchain 3.4.2_gs1 comes with *NanoMind A3200 BSP* (a3200-bsp) version 2.5.1. The toolchain source files are available on https://github.com/GomSpace/avr32-toolchain.

1.3.2 FreeRTOS

FreeRTOS has been upgraded to version 10.2, to solve some stability issues with 8.0. The FreeRTOS version can be changed through the wscript:

```
# Options for libasf
ctx.options.freertos_version = 10
```

FreeRTOS runtime stats

Runtime stats (*Time %* from the `ps` command) is disabled (in liba3200/conf/FreeRTOSConfig.h) for the time being.

1.4 Parameter Tables

1.4.1 Scheduler

This table contains configuration of the scheduler: *scheduler.c*.

Table 1.1: Parameter Table 'scheduler'

Name	Addr.	Type	
enabled	0x00	bool	Enable/disable scheduling Default: `False`
delayed_start	0x02	uint16	Delay before scheduling is started after boot/re-load Default: `30` Unit: `second`
bpx_col_intv	0x04	uint16	BPX HK collection interval. 0 disables collection Default: `0` Unit: `second`

Continued on next page

Table 1.1 – continued from previous page

Name	Addr.	Type	
bpx_csp_addr	0x06	uint8	BPX CSP address Default: 7
eps_col_intv	0x08	uint16	EPS HK collection interval. 0 disables collection Default: 0 Unit: second
eps_csp_addr	0x0a	uint8	EPS CSP address Default: 2
adsb_col_intv	0x0c	uint16	ADSB collection interval. 0 disables collection Default: 0 Unit: second
adsb_csp_addr	0x0e	uint8	ADSB CSP address Default: 4

1.4.2 NanoPower BPX telemetry

Table 1.2: Parameter Table 'bpx_hk'

Name	Addr.	Type	
c_charge	0x00	uint16	Charging current Invalid reading: 65535 Unit: milliampere
c_dcharge	0x02	uint16	Discharging current Invalid reading: 65535 Unit: milliampere
c_heater	0x04	uint16	Heater current Invalid reading: 65535 Unit: milliampere
vbatt	0x06	uint16	Voltage of battery Invalid reading: 0 Unit: millivolt
bat_temp1	0x08	int16	Battery temperature 1 Invalid reading: -32768 Unit: degC
bat_temp2	0x0a	int16	Battery temperature 2 Invalid reading: -32768 Unit: degC
bat_temp3	0x0c	int16	Battery temperature 3 Invalid reading: -32768 Unit: degC
bat_temp4	0x0e	int16	Battery temperature 4 Invalid reading: -32768 Unit: degC
bootcount	0x10	uint32	Number of BPX reboots Invalid reading: 4294967295
bootcause	0x14	uint8	Cause of last BPX reset Invalid reading: 255

1.4.3 NanoPower P31u telemetry

Table 1.3: Parameter Table 'eps_hk'

Name	Addr.	Type	
vboost	0x00	uint16[3]	Voltage of boost converters [mV] [PV1, PV2, PV3] Invalid reading: 0 Unit: millivolt

Continued on next page

Table 1.3 – continued from previous page

Name	Addr.	Type	
vbatt	0x06	uint16	Voltage of battery Invalid reading: `0` Unit: `millivolt`
curout	0x08	uint16[6]	Current out Invalid reading: `65535` Unit: `milliampere`
curin	0x14	uint16[3]	Current in Invalid reading: `65535` Unit: `milliampere`
cursun	0x1a	uint16	Current from boost converters Invalid reading: `65535` Unit: `milliampere`
cursys	0x1c	uint16	Current out of battery Invalid reading: `65535` Unit: `milliampere`
temp	0x1e	int16[6]	Temperature sensors [0 = TEMP1, TEMP2, TEMP3, TEMP4, BATT0, BATT1] Invalid reading: `-32768` Unit: `decidegC`
out_val	0x2a	uint8[8]	Status of outputs Invalid reading: `255`
battmode	0x32	uint8	Mode for battery [0 = normal, 1 = undervoltage, 2 = overvoltage] Invalid reading: `255`
pptmode	0x33	uint8	Mode for PPT [0 = HW, 1 = MPPT, 2 = FIXED] Invalid reading: `255`
wdtI2cS	0x34	uint32	Time left on I2C wdt Invalid reading: `4294967295` Unit: `second`
wdtGndS	0x38	uint32	Time left on Gnd wdt Invalid reading: `4294967295` Unit: `second`
bootcount	0x3c	uint32	Number of EPS reboots Invalid reading: `4294967295`
cntWdtI2c	0x40	uint32	Number of WDT I2C reboots Invalid reading: `4294967295`
cntWdtGnd	0x44	uint32	Number of WDT GND reboots Invalid reading: `4294967295`
cntWdtCsp	0x48	uint32[2]	Number of WDT CSP reboots Invalid reading: `4294967295`
wdtCspC	0x50	uint32[2]	Pings left on CSP wdt Invalid reading: `4294967295`
latchups	0x58	uint16[6]	Number of latch-ups Invalid reading: `65535`
bootcause	0x64	uint8	Cause of last EPS reset Invalid reading: `255`

1.4.4 NanoCom ADS-B telemetry

Table 1.4: Parameter Table 'adsb_hk'

Name	Addr.	Type	
cur_1v2	0x00	uint16	Current consumption, 1v2. Invalid reading: `65535` Unit: `milliampere`
cur_2v5	0x02	uint16	Current consumption, 2v5. Invalid reading: `65535` Unit: `milliampere`

Continued on next page

Table 1.4 – continued from previous page

Name	Addr.	Type	
cur_3v3_fpga	0x04	uint16	Current consumption, 3v3 FPGA. Invalid reading: `65535` Unit: `milliampere`
cur_3v3_adc	0x06	uint16	Current consumption, 3v3 ADC. Invalid reading: `65535` Unit: `milliampere`
cur_5v0_board	0x08	uint16	Current consumption, 5v0 Board. Invalid reading: `65535` Unit: `milliampere`
cur_3v3_board	0x0a	uint16	Current consumption, 3v3 Board. Invalid reading: `65535` Unit: `milliampere`
cur_3v3_sd	0x0c	uint16	Current consumption, 3v3 SD. Invalid reading: `65535` Unit: `milliampere`
avg_fps_10sec	0x0e	uint16	Average frames/sec over the last 10 seconds. Invalid reading: `65535`
avg_fps_1min	0x10	uint16	Average frames/sec over the last 1 minute. Invalid reading: `65535`
avg_fps_5min	0x12	uint16	Average frames/sec over the last 5 minutes. Invalid reading: `65535`
plane_count	0x14	uint32	Number of unique airplanes with at least one position decoded since boot. Invalid reading: `4294967295`
frame_count	0x18	uint32	Number of frames received since boot. Invalid reading: `4294967295`
crc_corrected	0x1c	uint32	Number of CRC errors corrected since boot. Invalid reading: `4294967295`
last_icao24	0x20	uint32	ICAO24 address of the latest airplane received. Invalid reading: `4294967295`
last_lat	0x24	float	Latitude of latest airplane received. Invalid reading: `91.0`
last_lon	0x28	float	Longitude of latest airplane received. Invalid reading: `181.0`
last_alt	0x2c	uint32	Altitude of latest airplane received. Invalid reading: `4294967295`
last_ts	0x30	uint32	Timestamp of latest airplane received. Invalid reading: `4294967295`
bootcount	0x34	uint32	Board boot count. Invalid reading: `4294967295`
bootcause	0x38	uint16	Board boot cause. Invalid reading: `65535`
current_time	0x3c	uint32	Current board time in seconds since UNIX epoch. Invalid reading: `4294967295`
tot_frames	0x40	uint32	Total number of frames received. Invalid reading: `4294967295`
tot_planes	0x44	uint32	Total number of unique airplanes with at least one position decoded. Invalid reading: `4294967295`
tot_crc_cor	0x48	uint32	Total number of CRC errors corrected. Invalid reading: `4294967295`
fpga_crc_cnt	0x4c	uint32	Total number of CRC errors in FPGA configuration. Invalid reading: `4294967295`
flash_lw	0x50	uint32	Invalid reading: `4294967295`

2. Product Interfaces

2.1 ADCS (libadcs_client)

2.1.1 Introduction

This is the client interface for ADCS.

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

Please refer to manual for ADCS for further details.

2.1.2 Commands

Commands (GOSH) are grouped under *adcs*.

2.1.3 Parameters

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

2.2 NanoCam C1U (nanocam2_client)

2.2.1 Introduction

This is the client interface for the NanoCam C1U.

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

Please see the NanoCam C1U manual for further details.

2.2.2 Client API

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

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

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

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

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

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

Image Capture

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

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

---

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

Image Storage

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

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

---

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

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

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

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

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

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

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

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

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

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

Modifying Sensor Registers

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

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

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

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

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

In-memory Images

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

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

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

Focus Assist Routine

int **nanocam_focus**(uint8_t *algorithm*, uint32_t **af*, unsigned int *timeout*)
This function runs a single iteration of the focus assist algorithm. The `algorithm` argument is used to select between different algorithms. Currently, `NANOCAM_AF_JPEG` is the only supported option.

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

Data Partition Recovery

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

2.2.3 Commands

Commands (GOSH) are grouped under *cam*.

2.2.4 Parameters

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

2.3 NanoCom ADS-B (gatoss-uc_client)

2.3.1 Introduction

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

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

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

2.3.2 Commands

Commands (GOSH) are grouped under *gatoss*.

2.3.3 Parameters

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

2.4 NanoCom AX100 (nanocom-ax_client)

2.4.1 Introduction

This is the client interface for the NanoCom AX100.

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

Please refer to manual for NanoCom AX100 for further details.

2.4.2 Commands

Commands (GOSH) are grouped under *ax100*.

2.5 NanoPower BPX (nanopower-bpx_client)

2.5.1 Introduction

This is the client interface for the NanoPower BPX.

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

Please see the NanoPower BPX manual for further details.

2.5.2 Commands

Commands (GOSH) are grouped under *bpx*.

2.6 NanoPower P31u (nanopower_client)

2.6.1 Introduction

This is the client interface for the NanoPower P31u.

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

Please see the NanoPower P31u manual for further details.

2.6.2 Commands

Commands (GOSH) are grouped under *eps*.

2.6.3 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:

```
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.

2.9 NanoPower P60 PDU (p60-pdu_client)

2.9.1 Introduction

This is the client interface for the NanoPower P60 PDU.

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

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

2.9.2 Commands

Commands (GOSH) are grouped under *p60pdu*.

2.9.3 Parameters

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

2.10 NanoPower P60 Library (libp60_client)

2.10.1 Introduction

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

2.10.2 Commands

Commands (GOSH) are grouped under *power*.

3. Libraries

3.1 A3200 (liba3200)

3.1.1 Introduction

The A3200 Library provides drivers and initialization code for the AVR32 processor, along with the peripheral devices and sensors on the A3200 board.

The A3200 on-board computer (OBC) is designed as an efficient system for space applications with limited resources, such as e.g. for CubeSat or nano-satellite missions.

The A3200 board is a powerfull and flexible CPU board that can be used to host several types of nanosat/cubesat satellite applications, e.g. Command and Data Handling (CDH)

A3200 board

The A3200 board is equipped with a set of sensors to enable important tasks in space applications.

The A3200 board can implement many of these tasks internally on the A3200 board, but the A3200 board can also interface to other systems/subsystems internally in the satellite as well as communicating with a Ground Station via a radio. The A3200 main interface to other subsystems are CAN and I^2C.

For storage the board carries a 128 MB NOR serial flash.

The Real Time Clock (RTC) chip on the board also functions as a processor companion with 32 kB of FRAM, providing non-volatile storage.

Beside the I^2C controller for the main bus, the board also has an extra I^2C controller that can be used to interface to external I^2C components, e.g. GomSpace Sensor Bus (GSSB).

For interfacing with SPI devices, the board has one external connection with three chip selects and it also has 8 inputs to an ADC. If needed the ADC inputs can also be used as GPIO.

Highlighted Features

- High-performance AVR32 MCU with advanced power saving features
  - 512 KB build-in flash
  - IEEE 754 FPU
  - Wide range for clock speeds with build-in PLL
- Multiple CSP data interfaces: I^2C, CAN and UART
- 128 MB NOR flash (On two dies of 64 MB each)
- 32 kB FRAM for persistent configuration storage
- 32 MB SDRAM
- RTC clock
- On-board temperature sensors
- 8 external ADC channels, that also can be used as GPIO
- External SPI with 3 chip selects
- Attitude stabilization system
  - 3-Axis magnetoresistive sensor

  - 3-Axis gyroscope
  - 3 bidirectional PWM output swith current measurements
- I²C interface for GomSpace Sensor Bus (GSSB)

Block Diagram

The following block diagram shows the main blocks on the A3200 board:

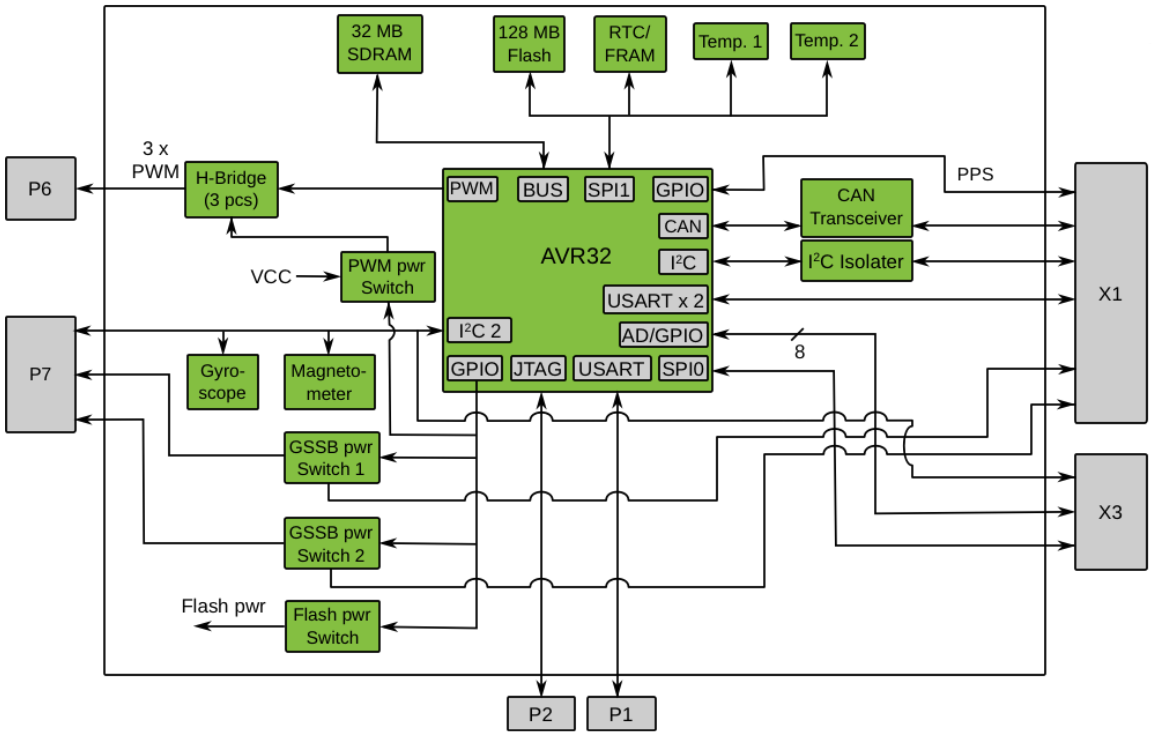


Fig. 3.1: Block diagram of the A3200 board

Please see the NanoMind A3200 Datasheet for further details on the AVR32 processor and the peripheral devices and sensors (available on the GomSpace website http://www.gomspace.com).

3.1.2 Drivers and API

The A3200 Library includes drivers and API functions for

- *SPI Bus*
- *I2C Bus*
- *CAN Bus*
- *FM33256B (FRAM / RTC)*
- *ADC Channels*
- *FL512 (NOR flash)*
- *Pulse Width Modulation (PWM)*
- *Power Switch Control (PWR)*
- *MPU3300 (Gyro)*

- *HMC5843 (Magnetometer)*

SPI Bus

The AVR32 SPI controllers are used for:

- SPI 0: wired to X3 (external use).
- SPI 1: wired to *FL512 (NOR flash)*. *FM33256B (FRAM / RTC)*, *LM71 Temperature sensor*.

The SPI interface can be initialised using:

```
// liba3200/src/init.c
void gs_a3200_init_spi0(bool decode);
void gs_a3200_init_spi1(void);
```

See *ASF (libasf)* for further details on the SPI bus.

Related APIs: *libembed/include/gs/embed/asf/drivers/spi/master.h*

I²C Bus

The I²C supports bidirectional data transfer between master and slaves, and can be configured as multi-master bus - handling arbitration between simultaneously transmitting masters without corruption of serial data on the bus.

The I²C controllers are used for:

- twi0 / I²C 0: wired to X1, through an isolator (external).
- twi2 / I²C 2: wired to P7, *HMC5843 (Magnetometer)*, *MPU3300 (Gyro)*.

The I²C interface can be initialised using:

```
// liba3200/src/init.c
void gs_a3200_init_twi0(gs_avr32_i2c_mode_t mode, uint8_t addr, uint32_t bps);
void gs_a3200_init_twi2(void);
```

See *ASF (libasf)* for further details on the I²C bus.

Related APIs: *libembed/include/gs/embed/asf/drivers/i2c/i2c.h*, *libgscsp/include/gs/csp/drivers/i2c/i2c.h*

CAN Bus

The CAN controllers are used for:

- CAN0: wired to X1, through a CAN transceiver (external).

The CAN interface can be initialised using:

```
// liba3200/src/init.c
void gs_a3200_init_can(bool enable)
```

See *ASF (libasf)* for further details on the CAN Bus.

Related APIs: *libembed/include/gs/embed/asf/avr32/drivers/can/can.h*, *libgscsp/include/gs/csp/drivers/can/can.h*

FM33256B (FRAM / RTC)

The board includes a FM33256B processor companion from Cypress, which provides:

- 32 KB FRAM, none-volatile data storage
- Real Time Clock (RTC)
- Watchdog
- Power detection (drives reset of the AVR32)

The FM33256B is connected via SPI 1.

The FRAM interface can be initialised using:

```
// liba3200/src/init.c
gs_error_t gs_a3200_init_fram(void);
```

This will also initialise the RTC of the FRAM.

Related APIs: *libthirdparty/src/fram/fm33256b.c*, *libutil/include/gs/util/rtc.h*.

The command group `fm33256b` supports the device. The command implementation can be used as a code example: *libthirdparty/src/fram/fm33256b_cmd.c*.

ADC Channels

To sample external analog values, the board supports a number of ADC channels in one of the main connectors. The ADC pins can also be configured to be GPIO instead of ADC inputs.

The ADC interface can be initialised using:

```
// liba3200/src/adc_channels.c
void gs_a3200_adc_channels_init(void);
```

Related APIs: *liba3200/include/gs/a3200/adc_channels.h*.

The command group `adc` supports the device. The command implementation can be used as a code example: *liba3200/src/cmd/cmd_adc.c*.

FL512 (NOR flash)

The FL512 Spansion chip is a 128 MB NOR flash (on two dies of 64 MB each). The flash is connected via SPI 1.

The flash interface can be initialised using:

```
// libthirdparty/src/flash/spn_fl512s.c
gs_error_t spn_fl512s_init(const spn_fl512s_config_t * config);
```

Related APIs: *libthirdparty/include/gs/thirdparty/flash/spn_fl512s.h*.

The command group `spn` supports the device. The command implementation can be used as a code example: *libthirdparty/src/flash/cmd_spn_fl512s.c*.

File system On A3200 SDK systems, the flash is used as file system (UFFS). For further details see *Storage (libstorage)*.

Mounting is initiated by calling:

```
// liba3200/src/sdk/fs.c
gs_error_t gs_a3200_fs_mount(gs_a3200_mount_fs_completed_t cb)
```

Because mounting the file system is a relative slow process, the work is done in a separate task with low priority.

Pulse Width Modulation (PWM)

The A3200 has 3 bidirectional outputs from 3 H-bridge drivers, designed to be controlled by a PWM output from the AVR32. The main purpose of these bidirectional outputs is to be used for external magnetorquers to implement attitude control. It is possible to switch the power to the PWM driver and they also support current measurements.

The PWM interface can be initialised using:

```
// liba3200/src/pwm.c
void gs_a3200_pwm_init(void);
```

Related APIs: *liba3200/include/gs/a3200/pwm.h*.

The command group `pwm` supports the device. The command implementation can be used as a code example: *liba3200/src/cmd/cmd_pwm.c*.

Power Switch Control (PWR)

The PWR turns power on/off to the on-board devices. Power switches are **off** after initilisation.

The PWR interface can be initialised using:

```
// liba3200/src/pwr_switch.c
void gs_a3200_pwr_switch_init(void)
```

Related APIs: *liba3200/include/gs/a3200/pwr_switch.h*

LM71 Temperature sensor

The 2 LM71 temperature sensors can be used to measure the temperature on the A3200 board. The sensors are connected via SPI 1.

The LM71 interface can be initialised using:

```
// liba3200/src/lm71.c
void gs_a3200_lm71_init(void);
```

Related APIs: *libthirdparty/include/gs/thirdparty/lm71/lm71.h*

The command group `lm71` supports the device. The command implementation can be used as a code example: *liba3200/src/cmd/cmd_lm71.c*.

MPU3300 (Gyro)

The A3200 board includes a 3-Axis gyroscope, the MPU3300 from InvenSense. The MPU3300 is connected via I^2C 2.

The MPU3300 interface can be initialised using:

```
// liba3200/src/mpu3300.c
gs_error_t gs_mpu3300_init(gs_mpu3300_bandwidth_t bandwidth, gs_mpu3300_full_scale_
↪reading_t full_scale)
```

Related APIs: *liba3200/include/gs/a3200/mpu3300.h*.

The command group `gyro` supports the device. The command implementation can be used as a code example: *liba3200/src/cmd/cmd_mpu3300.c*.

HMC5843 (Magnetometer)

The A3200 board includes a 3-Axis magnetometer to sense the Earth's magnetic field, the HMC5843 from Honeywell. The HMC5843 is connected via I^2C 2.

The HMC5843 interface can be initialised using:

```
// liba3200/src/hmc5843.c
gs_error_t gs_hmc5843_init(void)
```

The command group `hmc5853` supports the device. The command implementation can be used as a code example: *liba3200/src/cmd/cmd_hmc5843.c*.

3.1.3 Software Update in Orbit

It is possible to upload new software to the NanoMind A3200 in orbit. The new image(s) will be loaded into RAM and executed from RAM. Is is possible to have two RAM software images, where the second can be used as fallback in case the first fails. The original internal flash will also not be changed, so you always have this as another fallback option.

The procedure is:

1. Compile the new software for a RAM image. You need to use the `waf configure --ram` command to specify that you want to build the RAM image. When the RAM image has been built, you have a nanomind_ram.bin in the build folder.
2. Upload the new RAM image nanomind_ram.bin to the NanoMind flash file system using *FTP (libftp)*, upload to e.g. `/flash/new_image.bin`.
3. When the new image is uploaded and stored on the flash file system, you reconfigure the NanoMind to boot from the new image and how many times you will allow the NanoMind to boot from the new RAM image (this is a security mechanism to handle if a new RAM image fails to boot reliably). To do this, set the parameters `swload_img1` and `swload_cnt1`, eg `param set swload_img1 "/flash/new_image.bin"` and `param set swload_cnt1 10` to allow the new image to boot 10 times before reverting to the secondary image.
4. If the image is considdered a `golden image` then it can be set as a fallback RAM image. This is done using the parameters `swload_img2` and `swload_cnt2`, e.g. `param set swload_img2 "/flash/golden_image.bin"` and `param set swload_cnt2 10`.
5. Reboot NanoMind. When the NanoMind reboots from flash, it will detect that it needs to load the new RAM image (`/flash/new_image.bin`) to RAM and continue booting the new image. It will decrease the `swload_cnt1` parameter by 1 everytime it attempts to boot the RAM image.
6. In case the `swload_img1` fails to load or `swload_cnt1` has reached zero, then the `swload_img2` will be booted (if `swload_cnt2` did not reach zero).
7. If both image1 and image2 load count has reached zero or fails to load, the flash image is kept running as fallback.

The parameters `swload_img1 & 2` and `swload_cnt1 & 2` are defined in the configuration table *Parameter Table 'configuration'*.

The `swload_cnt1 & 2` maximum value is 100. It is defined by `MAX_SWLOAD_COUNT` in *src/sdk/swload.c* and can be changed if higher or lower value is required.

The boot from RAM image procedure is handled by the `gs_a3200_sdk_swload_task()` task. This task is started from the `gs_a3200_sdk_init_task()` (in *src/sdk/sdk_init.c*), once the file system is mounted successfully.

3.1.4 Parameters

The A3200 Library includes three parameter tables specific to NanoMind A3200 board: - board (*Parameter Table 'board'*) - configuration (*Parameter Table 'configuration'*) - telemetry (*Parameter Table 'telemetry'*)

For more information on table stores and general information on how to use the Parameter System, please refer to *Parameter System (libparam)*.

Board Table

The Board Table contains various module configuration that is only supposed to be altered in-orbit in case of emergency or failure. The table has both a Persistent and a Protected store, both located in the onboard FRAM.

On boot the table is loaded from the Persistent Store, if the copy in the Persistent Store is corrupted the table is loaded from the Protected Store.

Table 3.1: Parameter Table 'board'

Name	Addr.	Type	
uid	0x00	string	Unit IDentification If UID consists of printable characters, it will be used as *hostname* for GOSH and *cmp ident*
type	0x10	uint8	Type Optional/future, not used by standard SDK Default: `0`
rev	0x11	uint8	Revision Optional/future, not used by standard SDK Default: `0`
dock_type	0x12	string	Dock type Type of DOCK used for the A3200
csp_addr	0x22	uint8	CSP address CSP address of this node Default: `1` Reboot Required: `Yes` Valid Range: `1 <= csp_addr <= 31`
csp_rtable	0x23	string	CSP routing table Format: <address>/<mask> <interface> [MAC] (CSV list). Example: `2/5 I2C,16/1 CAN 5,0/0 CAN` Default: " " Reboot Required: `Yes`
gosh_en	0x83	bool	GOSH enabled Enable GOSH console (prompt) on the UART specified by `gosh_uart` Default: `True` Reboot Required: `Yes`
gosh_uart	0x84	uint8	GOSH UART UART used for the GOSH console, if enabled by `gosh_en` Default: `2` Reboot Required: `Yes` Valid Values: **2:** UART2 in connector P1 **4:** UART4 in connector X1
can_en	0x85	bool	CAN enabled Enable CSP via the CAN interface in connector X1 Default: `True` Reboot Required: `Yes`

Continued on next page

Table 3.1 – continued from previous page

Name	Addr.	Type	
can_brate	0x88	uint32	CAN bitrate Bitrate for CAN interface Default: `1000000` Unit: `bit/second` Reboot Required: `Yes` Valid Values: **1000000:** 1M **500000:** 500K **250000:** 250K
i2c_en	0x8c	bool	I2C enabled Enable CSP via the I2C interface in connector X1 Default: `True` Reboot Required: `Yes`
i2c_brate	0x90	uint32	I2C bitrate Bitrate for I2C interface Default: `400000` Unit: `bit/second` Reboot Required: `Yes`
kiss_en	0x94	bool	KISS enabled Enable CSP via the KISS (serial) interface set by `kiss_uart` Default: `False` Reboot Required: `Yes`
kiss_brate	0x98	uint32	KISS baudrate Baudrate for the KISS interface Default: `500000` Unit: `baud` Reboot Required: `Yes`
kiss_uart	0x9c	uint8	KISS UART UART used for the KISS interface Default: `4` Reboot Required: `Yes` Valid Values: **2:** UART2 in connector P1 **4:** UART4 in connector X1

Configuration Table

The Configuration table contains various module configuration that might change during the mission. The table has both a Persistent and a Protected store, both located in the onboard FRAM.

On boot the table is loaded from the Persistent Store, if the copy in the Persistent Store is corrupted the table is loaded from the Protected Store.

Table 3.2: Parameter Table ‘configuration’

Name	Addr.	Type	
gssb1_pwr_en	0x00	bool	GSSB1 power Enable power on pin `GSSB_VCC` in connector P7 Default: `False`
gssb2_pwr_en	0x01	bool	GSSB2 power Enable power on pin `GSSB2_VCC` in connector P7 Default: `False`
flash_pwr_en	0x02	bool	Flash power Enable power to onboard Flash Default: `False`

Continued on next page

Table 3.2 – continued from previous page

Name	Addr.	Type	
pwm_pwr_en	0x03	bool	PWM power Enable power to PWM (Pulse Width Modulation) Default: `False`
swload_img1	0x04	string	Primary program image Filename for primary program image to load and boot from memory. Only tried if `swload_cnt1` is greater than 0 (zero) Reboot Required: `Yes` Auto Persist: `Yes`
swload_cnt1	0x24	uint16	Primary program image boot count Number of times to try and boot primary program image, configured in `swload_img1` Reboot Required: `Yes` Auto Persist: `Yes` Valid Range: `0 <= swload_cnt1 <= 100`
swload_img2	0x26	string	Secondary program image Filename for secondary program image to load and boot from memory. Only tried if `swload_cnt2` is greater than 0 (zero) Reboot Required: `Yes` Auto Persist: `Yes`
swload_cnt2	0x46	uint16	Secondary program image boot count Number of times to try and boot secondary program image, configured in `swload_img2` Reboot Required: `Yes` Auto Persist: `Yes` Valid Range: `0 <= swload_cnt2 <= 100`
fs_default	0x48	uint8	Default partition Flash partition to use as default, mounted as `/flash`. The specified partition must be enabled by parameters: `fs_en_p0`, `fs_en_p1` Default: `0` Reboot Required: `Yes` Valid Values: **0:** Partition 0 **1:** Partition 1
fs_en_p0	0x49	bool	Enable /flash0 Enable mounting of the /flash0 partition Default: `True` Reboot Required: `Yes`
fs_en_p1	0x4a	bool	Enable /flash1 Enable mounting of the /flash1 partition Default: `True` Reboot Required: `Yes`
log_fram_en	0x4b	bool	Log to FRAM If enabled and using default settings, logs of type: `error`, `warning` and `notice` will be logged to the FRAM. Default: `True` Reboot Required: `Yes`
log_mask	0x4c	string	Log mask/level settings Settings applied to log groups and appenders during boot, overwriting the default levels set in the code. Format: <group|appender>=<level> (CSV list). Wildcard * is supported as the last character in the name. Level: 0 (off), e (error), w (warning), n (notice), d (debug), t (trace) Example: `hk_*=0,csp=t` Default: “ “
tsync_node	0x8c	uint8	TSYNC CSP Addr CSP address of the node to synchronize time against Valid Range: `1 <= tsync_node <= 31`

Continued on next page

Table 3.2 – continued from previous page

Name	Addr.	Type	
tsync_intv	0x8e	uint16	TSYNC Interval Interval for time synchronization, 0 disables time synchronization Default: `60` Unit: `second`
spi0_cs_mode	0x90	uint8	SPI0 CS Mode Defines how the chip select pins works on the `SPI0` device Default: `0` Reboot Required: `Yes` Valid Values: **0:** Normal (0-3) **1:** Decode (0-15)

Telemetry Table

The Telemetry table contains diagnostics, e.g. temperatures, status counters.

Table 3.3: Parameter Table 'telemetry'

Name	Addr.	Type	
fs_mounted	0x00	bool	Filesystem mounted Is flash filesystem mounted?
ram_image	0x01	bool	Is RAM image Is the program running from a RAM image?
temp_mcu	0x02	int16	MCU Temperature Temperature sensor on A3200 board, near the MCU. Unit: `decidegC`
temp_ram	0x04	int16	RAM Temperature Temperature sensor on A3200 board, near the external RAM Unit: `decidegC`
i_GSSB1	0x06	uint16	GSSB1 current Current consumption on GSSB1 Unit: `milliampere`
i_GSSB2	0x08	uint16	GSSB2 current Current consumption on GSSB2 Unit: `milliampere`
i_Flash	0x0a	uint16	Flash current Current consumption by Flash device Unit: `milliampere`
i_PWM	0x0c	uint16	PWM current Current consumption by PWM Unit: `milliampere`
resetcause	0x10	uint32	Reset cause If `bootcause` is 4 (software reset), this parameter may hold a more detailed cause. Valid Values: **0:** Unknown **3:** Stack overflow. A task exceeded its stack space **4:** Software exception, e.g. access unaligned memory address, performed illegal operation **5:** Reset initiated by GOSH command **6:** Reset initiated by a request (message) on the CSP interface **7:** Reset due to filesystem been rebuild/created. **8:** Reset due to insufficient memory

Continued on next page

Table 3.3 – continued from previous page

Name	Addr.	Type	
bootcause	0x14	uint32	Bootcause The cause of the last boot Valid Values: **0:** Unknown **1:** Brown out detection **2:** Power on **3:** Watchdog **4:** Software triggered boot, see `resetcause` for detailed reason **5:** Reset by the external reset pin **6:** Woken from sleep/standby mode **7:** Reset due to CPU error, eg illegal addressing **8:** Reset by JTAG (programmer)
bootcount	0x18	uint16	Bootcount The number of times the board has booted Auto Persist: `Yes`
clock	0x1c	uint32	Clock time Current time in seconds since Unix Epoch Unit: `second`
uptime	0x20	uint32	Uptime Time since last boot Default: `0` Unit: `second`

3.2 A3200 Dock (liba3200dock)

3.2.1 Introduction

The A3200 DOCK library provides interfaces for GomSpace A3200 DOCK's. Following dock's are supported:

- DMC3
- ADCS3
- ADCS6
- ADCS8

3.3 ASF (libasf)

3.3.1 Introduction

The ASF is basically a customised version of the Advanced Software Framework (ASF) library from Microchip.

The ASF contains the low level drivers for supporting processors like AVR32, SAMC21, etc.

GomSpace has customised the original ASF in form of configuration and minor extensions, e.g. FreeRTOS port for GomSpace used processors.

The ASF is not documented further in this manual.

Please refer to http://asf.atmel.com for detailed information about the Advanced Software Framework.

Please refer to http://www.freertos.org for detailed information about FreeRTOS.

3.4 GomSpace CSP (libgscsp)

3.4.1 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.

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.

3.4.2 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
```

3.4.3 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.

- 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.

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

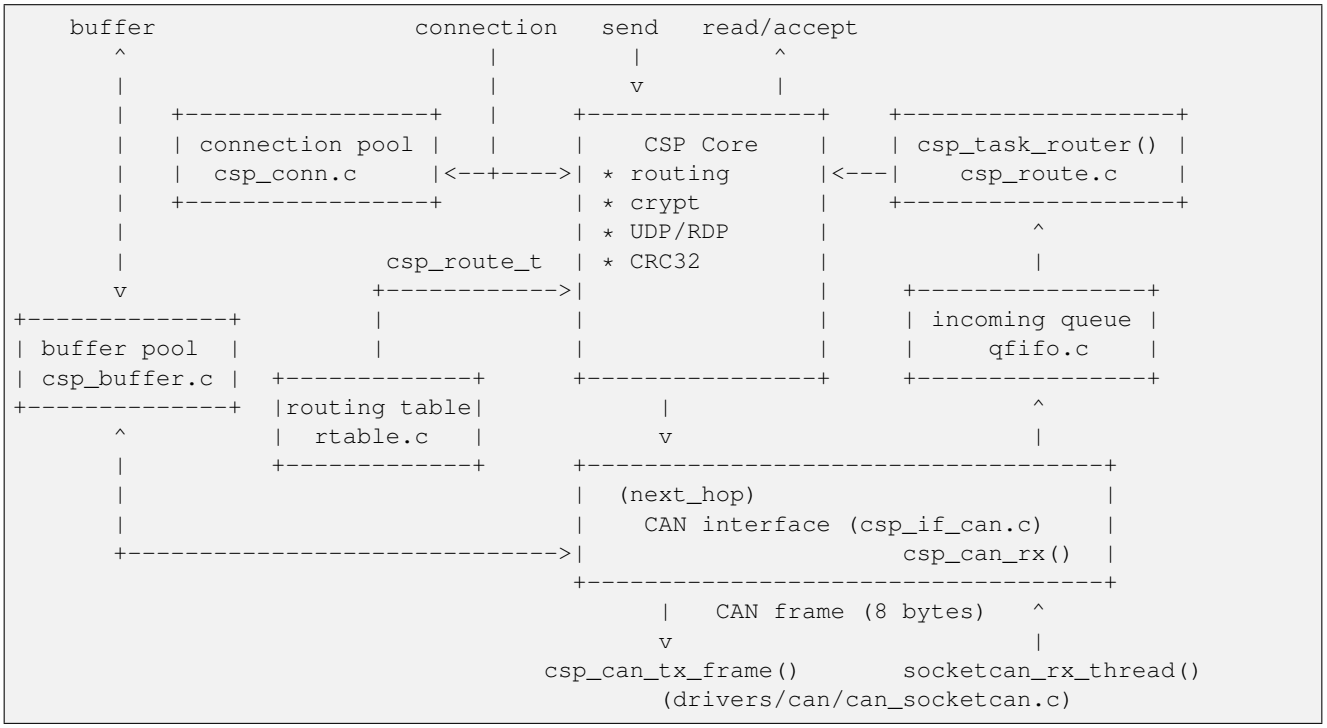
Structure

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).



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. */
```

(continues on next page)

(continued from previous page)

```
        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];
        };
} 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.

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:

```
/**
   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.

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.

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.

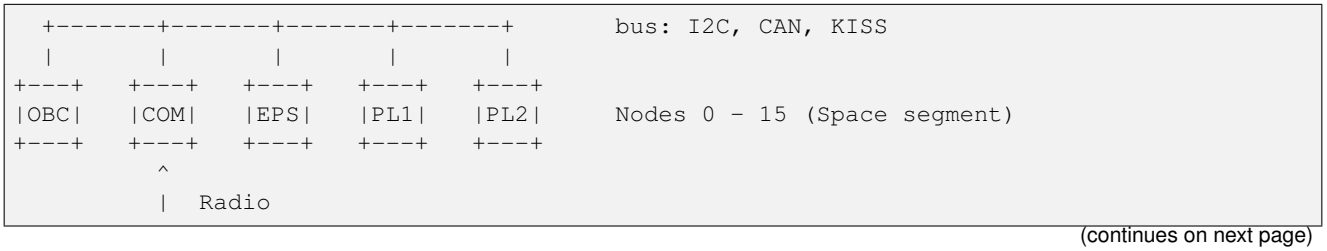
RDP CSP provides a transport layer extension called RDP (reliable datagram protocol) which is an implementation of RFC908 and RFC1151. RDP provides a few additional features:

- 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.



(continues on next page)

(continued from previous page)

```
          v
        +---+           +----+
        |TNC|           | PC |             Nodes 16 - 31 (Ground segment)
        +---+           +----+
          |                 |
          +-----------------+               bus: KISS, ZMQ
```

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 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:

  ```
  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)
```

3.5 Embed (libembed)

3.5.1 Introduction

The Embed library contains platform specific implementations of cross-platform API's (defined in the Utility library) and API's used across multiple embedded platforms.

The table below lists the supported drivers for each platform:

Driver	AVR32	AVR8	SAMC	STM32
CAN	x		x	x
I2C	x	x	x	x
SPI	x		x	x
GPIO	x		x	x
UART	x	x	x	x
FLASH	x		x	x
ADC			x	x
DAC			x	

3.5.2 Commands

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

```
nanomind #
  reset                mcu: Reset now
  ps                   mcu: List tasks
  peek                 mcu: Read byte(s) from memory
  poke                 mcu: Write byte to memory
  free                 mcu: Show memory usage
```

These commands are primarily for debugging purposes and verifying how the system is running, e.g. resource usage.

3.6 Emulation (libemul)

3.6.1 Introduction

The Emulation library makes it possible to build and run the A3200 NanoMind application on a standard Linux, with almost full feature set.

Using the cross-platform API's defined in the Utility library, allows A3200 software to be compiled and run on Linux. This has some great advantages related to test (automatic).

Known limitations

Building the A3200 SDK on Linux is currently not supported in the officially released SDK.

3.7 Flight Planner (libfp)

3.7.1 Introduction

The Flight Planner Library implements a service to execute commands at certain timestamps, either absolute or relative to the time of creation. The flight planner service is configured via the GOSH interface, so entries can

be manipulated using commands directly on the NanoMind or from CSP-term.

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

On the NanoMind, the Flight Planner is divided into a server task and a single executor task. Because the executor is a single thread, only one flight planner command can run at a time, but multiple entries can be scheduled for future execution.

The Flight Planner Client Library implements the client side of the Flight Planner protocol.

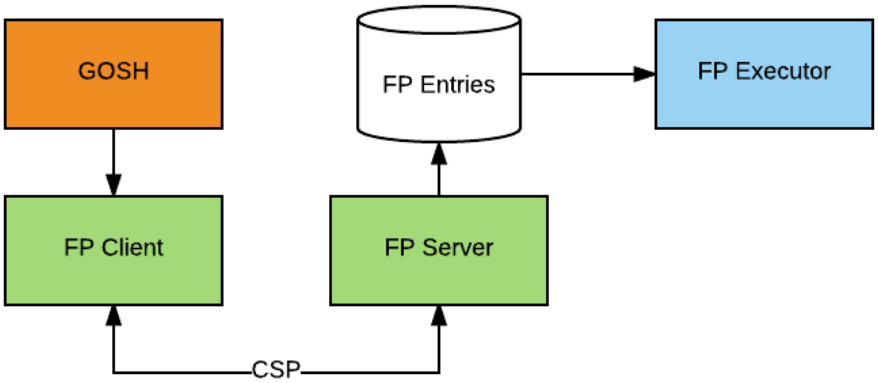


Fig. 3.2: Flight Planner components.

The Flight Planner operates on entries, each containing a single command to execute. Entries are tagged with an execution time which can be relative or absolute. Relative time tagged entries can have a number of repetitions to execute the command multiple times at regular intervals. All entries can be marked as active (default) or dormant.

The Flight Planner can hold a maximum of 5000 entries.

3.7.2 Commands

fp

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

Table 3.4: 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)

Continued on next page

Table 3.4 – continued from previous page

Command	
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.
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.

Table 3.5: 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
```

3.7.3 API

client

```
// Flight Planner Client example

#include <gs/fp/fp_client.h>

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

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

(continues on next page)

(continued from previous page)

```
    /* ... */

    /* activates the dormant fp entry */
    return fp_client_timer_set_active(name);
} //
```

3.7.4 Limitations

- The Flight Planner can hold a maximum of 5000 entries.

3.8 FTP (libftp)

3.8.1 Introduction

The FTP library provides a generic File Transfer Protocol (FTP) running over CSP. FTP allows remote transfers and manipulation of files to and from a remote node, e.g. NanoMind. The file transfer functionality is built on top of CSP and RDP as a transport layer protocol. This ensures flow control and correct ordering of messages. FTP uses a simple chunk based protocol with a bitmap file to mark which chunks has been transferred.

The protocol is for example used for uploading new software images to the NanoMind.

The protocol implementation consists of a *server* and a *client*. The *server* must run on any nodes, where you want access to the local file system. The *client* can co-exist with the *server*. The figure below shows major components and call chains.

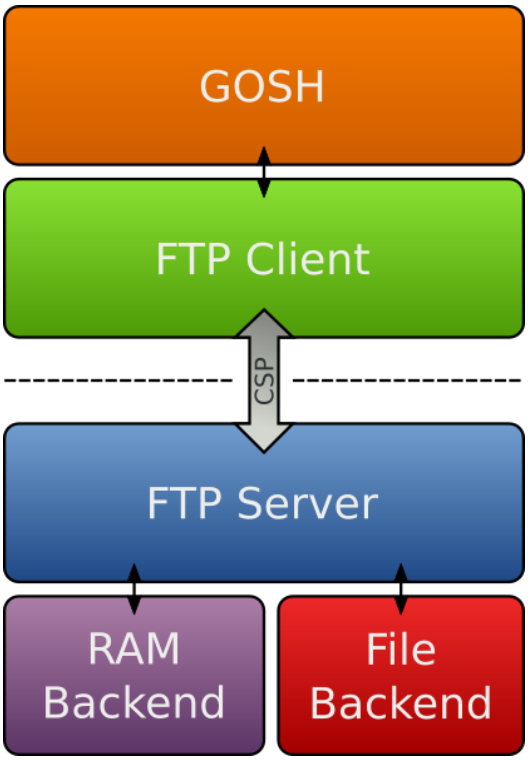


Fig. 3.3: File Transfer Protocol components.

The *server* component supports following *backends*: - file - access to the local file system - RAM - access to memory, only supported on embedded systems e.g. NanoMind.

What *backend* to interface with, is determined by the URL given to the *client* interface. URL's can take following forms:

- file://<path_to_file>
- mem://<addr>[++size]
- <path_to_file> . . . same as writting file://<path_to_file>

The FTP (client) library provides a C API and a set of commands (see *Commands*) for interfacing with the *server*.

Limitations

The FTP library has a max file size of 2GB, but normally the underlying file system will set the limit.

3.8.2 Commands

ftp

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

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

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

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

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

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

and see if its really there:

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

3.9 GOSH (libgosh)

3.9.1 Introduction

The GOSH library provides a number of extended features to the *command* framework provided by the Utility library. Originally the *command* framework was contained in this library (hence the name *GOSH*), but the basic features were moved to the Utility library in order to remove circullar dependencies.

G-script

GomSpace script engine.

The engine can be controlled through a set of commands provided by the GOSH (client) library, see *G-script*.

Remote

Execute remote shell command over CSP.

Commands are provided by the GOSH (client) library, see *Remote Shell*.

Redirect

Redirection of stdio over TCP/IP socket.

3.9.2 Commands

G-script

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

Commands are grouped under *gscript*.

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

Remote Shell

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

Commands are grouped under *shell*.

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

3.10 GomSpace Sensor Bus (libgssb)

3.10.1 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.3.4.

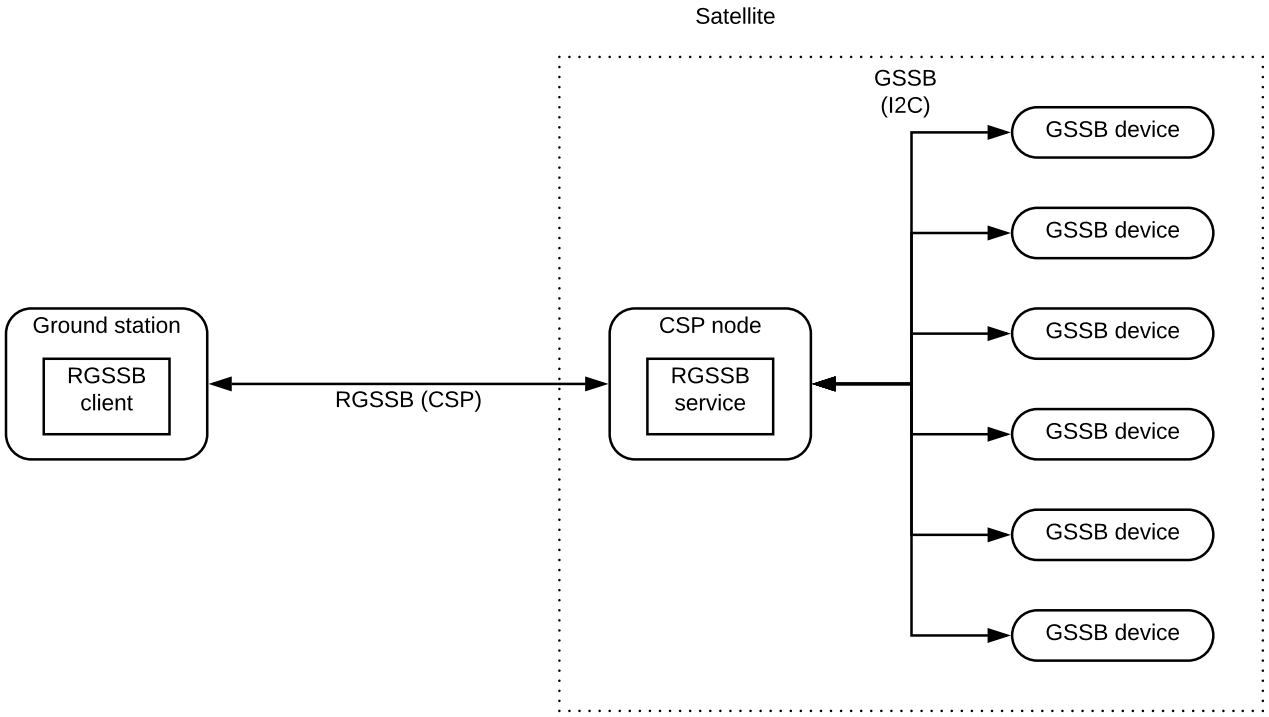


Fig. 3.4: GSSB network illustration.

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

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

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

The library consists of to parts, libgssb and libgssb_client. The client consists of functions for both the GSSB

protocol, for communication directly with GSSB devices, and functions for the RGSSB protocol, for communication with a proxy RGSSB service. The actual library libgssb holds API for the RGSSB service.

3.11 Housekeeping (libhk)

3.11.1 Introduction

The Housekeeping library provides functions and services for collecting telemetry parameters from individual nodes and for encoding and transmitting telemetry beacons to ground station.

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

3.11.2 System overview

The Housekeeping is logically separated into five components:

- **Configuration**: The Configuration module is used to tell the Housekeeping which telemetry tables should be sampled and what beacon types it should know about.
- **Collector**: The Collector is a number of tasks that sample, store and persist telemetry tables.
- **Telemetry Cache**: Collector uses the Telemetry Cache to store and persist telemetry table samples.
- **Beacon Service**: This module defines the network interfaces and implements a CSP service task to process incoming requests for telemetry beacons.
- **Beacon Parser**: This module handles the decoding and parsing of telemetry beacons.

The **Beacon Parser** component is part of the Housekeeping client library.

These main components and their relations can be illustrated in the following way:

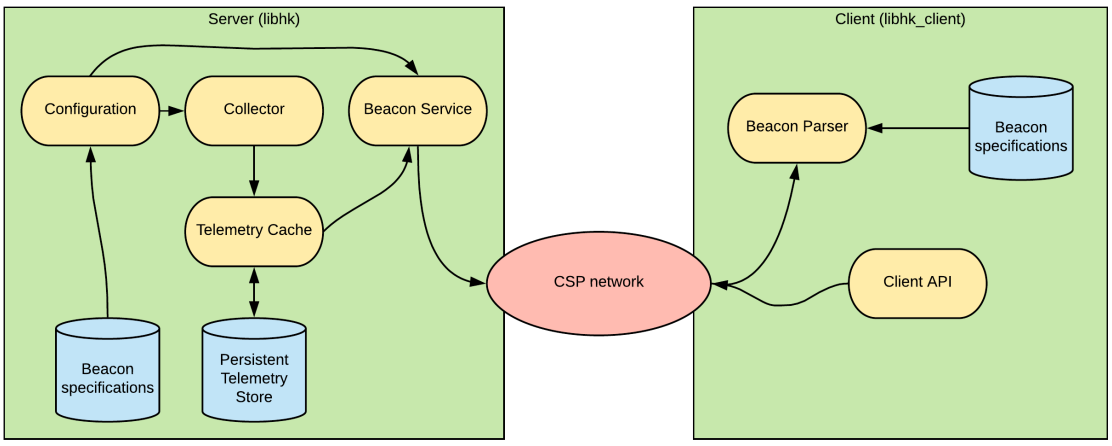


Fig. 3.5: Components in the Housekeeping library.

3.11.3 Configuration

The main interface for configuring the Housekeeping is a number of *.json* files each containing a beacon specification. Active beacon specifications are stored in *<data_path>/bcn_types* - the *data_path* parameter can be found in *Parameter Table 'hkcfg'*. Beacon specifications can also be stored in other parts of the filesystem and copied to the *bcn_types* folder when needed and activated using the *hk reload* command.

Listing 3.1: Beacon specification example 1.

```
{
    "type": 1,
    "version": 1,
    "samplerate": "high",
    "auto_beacon_policy": "always",
    "max_samples": 900,
    "elements" : [
        {
            "node_address": 1,
            "table_id": 4,
            "params": [
                { "name": "i_PWM" },
                { "name": "i_Flash" }
            ]
        }
    ]
}
```

When loaded by the Housekeeping the beacon specification in *Beacon specification example 1.* will trigger the Collector to sample table 4 from node 1 every 10'th second (*samplerate*). The Beacon Service will know how to generate the beacon type 1 in version 1 and due to the *auto_beacon_policy* setting a beacon will be generated and sent every 10'th second. The Collector will keep a max 900 samples of table 4 from node 1, making it possible to request telemetry beacons of this type 2,5 hours into the past. See *Autobeacon policies* and *Samplerates* tables for details on the *auto_beacon_policy* and *samplerate* settings. Telemetry beacons of type 1 will contain the *i_PWM* and *i_Flash* parameters from table 4 on node 1.

To update beacon type 1 a new specification would have to be uploaded and the *hk reload* command would be used to activate the new version.

It is possible to temporarily change the *samplerate* by using the *hk beacon samplerate* command. It is also possible to temporarily change the *auto_beacon_policy* using the *hk beacon policy* command. Both settings are reverted on system reboot or configuration reload.

Listing 3.2: Beacon specification example 2.

```
{
    "type": 2,
    "version": 1,
    "samplerate": "medium",
    "auto_beacon_policy": "off",
    "max_samples": 1200,
    "elements" : [
        {
            "node_address": 1,
            "table_id": 4,
            "params": [
                { "name": "i_PWM" },
                { "name": "i_Flash" }
            ]
        },{
            "node_address": 2,
            "table_id": 4,
            "download_timeout": 500,
```

(continues on next page)

(continued from previous page)

```
            "params": [
                { "name": "temp_int", "array_index": "0-4" },
                { "name": "temp_ext" }
            ]
        }
    ]
}
```

The beacon specification in *Beacon specification example 2.* is an example where a beacon contains data from mulitple tables. The beacon is never sent automatically (*auto_beacon_policy*) and both tables are sampled every 30 seconds (*samplerate*), storing maximum 1200 samples (*max_samples*). This makes it possible to request the beacon type 10 hours into the past. Telemetry beacons of type 2 will contain the *i_PWM* and *i_Flash* parameters from table 4 on node 1 and the *temp_int* and *temp_ext* parameters from table 4 on node 2. The parameter *temp_int* is an example of how to handle arrays in telemetry parameters by requesting index 0-4 in the *temp_int* array. The *array_index* option can also be given the value of a single index instead of a range.

Important: Even though a table is required in mulitple beacons types, collection of the table is only configured once. The table will be collected with the most frequent samplerate, using the highest number for *max_samples*. This means that if *Beacon specification example 1.* and *Beacon specification example 2.* are configured on the same Housekeeping System, table 4 from node 1 would be collected using a *high* samplerate (1 every 10 seconds), storing 1200 samples. This means that you can data from table 4 from node 1 will be available 3 hours and 20 minutes into the past instead of the 10 hours achieved if only *Beacon specification example 2.* was configured.

The *download_timeout* setting can be used to tweak the time which the Collector waits for a response when collecting telemetry samples. This can be useful if a given node responds slowly for some reason. An example of use of this setting is show in *Beacon specification example 2.*. The default timeout value is 200 milliseconds.

To save a bit of bandwidth when uploading beacon specifications, it is possibly to compact specifications by removing all whitespace - *Beacon specification example 3 - compacted specification.* is an example of such.

Listing 3.3: Beacon specification example 3 - compacted specification.

```
{"type":3,"version":1,"samplerate":"high","auto_beacon_policy":"always","max_samples":900,
↪"elements":[{"node_address":4,"table_id":4,"params":[{"name":"i_PWM"},{"name":"i_Flash"}
↪,{"name":"bananas","array_index":"0-2"}]}]}
```

Table 3.6: Samplerates

Samplerate setting	Table sample interval
highest	1 / second
high	1 / 10 seconds
medium	1 / 30 seconds
low (default)	1 / 60 seconds
lowest	1 / 180 seconds

Table 3.7: Autobeacon policies

Setting	Function
always	Send no matter the state of the sampled tables.
complete	Send only if all the sampled tables are being sampled correctly.
off (default)	Only send if requested.

3.11.4 Collector & Telemetry Cache

The Collector samples telemetry tables from remote nodes and locally. It samples tables based on what tables are required for the loaded beacon specifications. Sampling is done into the Telemetry Cache and the *max_samples* parameter in the beacon specification is controlling how many samples are stored. Samplerate it controlled by the *samplerate* parameter in the beacon specification, see *Samplerates* for details.

Note that the collector samples full tables, no matter how many parameters are needed in beacons.

The Telemetry Cache can persist its cached telemetry samples to the local file system. It is possible to turn the persistence to local file system on and off using the *persist_en* parameter in the *Parameter Table 'hkcfg'*. It is also possible to configure how often the telemetry sample cache is persisted to local file system using the *persist_interval* parameter from *Parameter Table 'hkcfg'*. In case of a system reboot, the telemetry cache is re-initialised from the persisted telemetry samples.

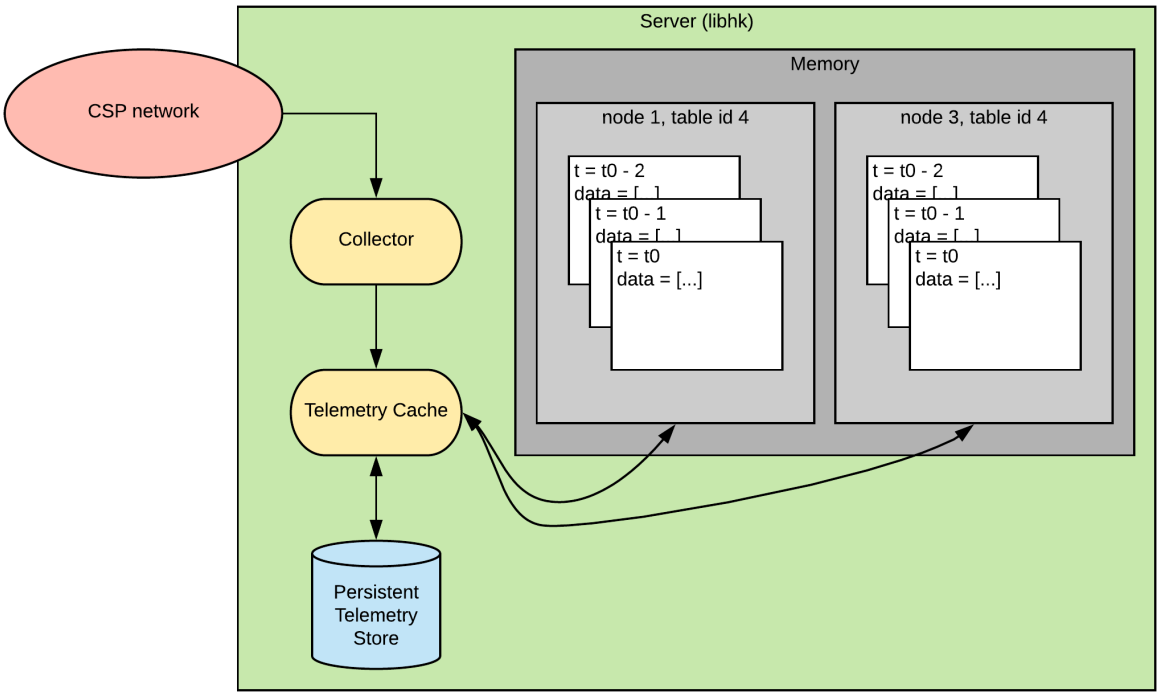


Fig. 3.6: Collector & Telemetry Cache.

3.11.5 Beacon Service

The Beacon Service serves two purposes;

Encoding and transmission of automatic Telemetry Beacons

Automatic telemetry beacons are generated and transmitted according to the *auto_beacon_policy* in the beacon specification. The *auto_beacon_policy* parameter controls whether to send the beacon, and the beacon is sent whenever tables are sampled - controlled by the *samplerate* setting. See *Autobeacon policies* and *Samplerates* for details. All automatic beacon transmissions can be temporarily disabled using the *bcn_atx_tpds* parameter in *Parameter Table 'hkcfg'*. The CSP destination for automatic telemetry beacons can be controlled by *bcn_atx_da* and *bcn_atx_dp*, both in *Parameter Table 'hkcfg'*.

Handling Telemetry Beacon requests

The Beacon Service also handles Telemetry Beacon requests from clients. There are two ways of getting Telemetry Beacons from the Beacon Service, via CSP packets and via files.

If Telemetry Beacons via CSP packets are requested the Beacon Service will transmit one CSP packet pr. requested timestamp, one CSP packet contains a single data for a single Beacon Configuration at a single timestamp.

If Telemetry Beacons via files are requested the Beacon Service will write all the requested timestamps into a single file on the local filesystem. The file can be downloaded after the Beacon Service is done writing it.

The recommended solution is to use Telemetry Beacons via CSP packets - that way partial results can still be downlinked on a link of low quality.

See *API* for details on how to send requests to the Beacon Service and *Data formats* for details on the actual data layout of Telemetry Beacons and files.

3.11.6 Beacon Parser

The Beacon Parser is a python module that is capable of parsing/decoding telemetry beacons from either a file or a CSP packet.

Please refer to the API documentation in the module itself for further details. The module can be found in *python/hk/beacon_parser.py* in the Housekeeping client library.

3.11.7 Data formats

A telemetry beacon is formated as described in the following tables:

Table 3.8: Telemetry Beacon

Field	Offset	Size	Type	Description
protocol_version	0	1	UINT8	Beacon protocol version, rest of data is only valid if this equals 1.
type	1	1	UINT8	The beacon type.
version	2	1	UINT8	The beacon version.
satid	3	2	UINT16	Identifier of the satellite from which the beacon originates.
data_element1	5	8+M		Data element 1, as described in *Telemetry Beacon Element*. M is the size of the data in this element.
data_element2	5+8+M	8+M		Data element 2, as described in *Telemetry Beacon Element*. M is the size of the data in this element.
...	...	...	...	...
data_elementN	5+N*(8+M)	8+M		Data element N, as described in *Telemetry Beacon Element*. M is the size of the data in this element.

Each data element is formatted like this:

Table 3.9: Telemetry Beacon Element

Field	Offset	Size	Type	Description
checksum	0	2	UINT16	Checksum of the table used in this element.
timestamp	2	4	UINT32	Timestamp (seconds since UNIX epoch) containing sampletime for the data in this element.
source	4	2	UINT16	Node from which this data originates.
data	8		BYTE[]	Beacon data as described by the beacon specification.

The beacon data file format is listed in the following table:

Table 3.10: Telemetry Beacon File

Field	Offset	Size	Type	Description
file_id	0	4	UINT32	File id, 'magic' file id. 0x62630001 indicates Big Endian byte order. 0x01006362 indicates Little Endian byte order.
beacon1	4	3+M	UINT32	Beacon record 1, beacon data size M.
beacon2	4+3+M	3+M	UINT32	Beacon record 2, beacon data size M.
...	...	...	...	...
beaconN	4+N*(3+M)	3+M	UINT32	Beacon record N, beacon data size M.

Each record in the beacon file has the following format:

Table 3.11: Telemetry Beacon File Record

Field	Offset	Size	Type	Description
rec_sz	0	2	UINT16	Size of beacon record
type	2	1	UINT8	Beacon type
beacon_data	3	rec_sz	BYTE []	The beacon data as described by *Telemetry Beacon*

3.11.8 Resource usage considerations

The Housekeeping library uses resources in a dynamic way and the end user is responsible for validating actual configurations and that their resource use is within the bounds of the mission. Below are some guidelines to help calculate the resource usage of a given configuration.

Telemetry Cache - memory consumption

Each table being sampled will consume an amount of memory in the Telemetry Cache dependent on the size of the table being sampled and the (*max_samples*) setting.

If the specification from *Beacon specification example 1.* is loaded and we assume that *table_id* 4 on *node_address* 1 has a size of 120 bytes, the total memory consumption the Telemetry Cache for this is ~105KB *((120 bytes * 900 samples) / 1024)*.

If multiple beacons use data from the same table it will only be sampled once - at the highest required rate and storing the largest *max_samples*. If the specification from *Beacon specification example 2.* is also loaded and we assume that *table_id* 4 on *node_address* 2 has a size of 150 bytes, the total memory consumption for the Telemetry Cache is ~316KB *((120 bytes * 1200 samples + 150 bytes * 1200 samples) / 1024)*.

The *hk table show* command can be used to check the number of samples and size of each sample for a collected table.

Telemetry Cache - persistent storage usage

If persisting is enabled, the amount of bytes used for the Telemetry Cache will also be used on the persistent storage.

Collector - CSP Bus utilization

Sampling a table from a remote CSP node will utilize the CSP bus. The amount of utilization is dependent on the size of the table being sampled and the *samplerate* used.

If the specification from *Beacon specification example 1.* is loaded *table_id* 4 on *node_address* 1 will be sampled at a rate 1/30 Hz (*samplerate=medium*). If we assume that the table has a size of 120 bytes, the CSP bus utilization averaged over a period will be ~4bytes/s (1/30 * 120).

It is important to be aware that sampling of tables is synchronized to whole seconds and therefore the utilization will not be evenly distributed.

Collector - MCU utilization

The collector tasks utilizes the MCU to an extent depedent on the number of tables being sampled, the size of these tables and the *samplerate*. A few examples of measured utilization on the Nanomind A3200 platform are given here to aid mission design.

Table 3.12: Collector MCU utilization

Number of tables	Samplerate	Average table size	Measured utilization
20	1/60Hz (*low*)	~100 bytes	<1%
20	1/10Hz (*high*)	~100 bytes	<2%
20	1Hz (*highest*)	~100 bytes	<20%

3.11.9 API

server

Example of how the Housekeeping System is started.

```
// Housekeeping Server initialization example

#include <gs/hk/config.h>
#include <gs/hk/collector.h>
#include <gs/hk/beacon_service.h>
#include <gs/hk/persister.h>
#include <gs/hk/command.h>

void start_hk_system()
{
    /* call this before any other gs_hk_* functions */
    gs_hk_config_init();

    /* start tasks/threads */
    gs_hk_create_config_init_thread();
    gs_hk_create_collector_threads();
    gs_hk_create_beacon_service_thread();
    gs_hk_create_persister_thread();

    /* register commands */
    gs_hk_srv_register_commands();
}
```

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
```

(continues on next page)

(continued from previous page)

```
                                    3,      // Beacon type to request
                                    10,     // 10 second interval
                                    10,     // 10 samples
                                    GS_HK_REQUEST_BEACON_FIRST_SAMPLE_NOW,
                                    NULL); // Request packets not file
}
```

3.11.10 Parameter Tables

The Housekeeping library exposes a single parameter table with overall options for the Housekeeping system.

Table 3.13: Parameter Table 'hkcfg'

Name	Addr.	Type	
bcn_atx_tpds	0x00	uint16	Temporarily disable all automatic beacon transmits for this number of seconds. Default: `0` Unit: `second`
bcn_atx_da	0x02	uint8	CSP destination address for automatic beacon transmits. Default: `10`
bcn_atx_dp	0x03	uint8	CSP destination port for automatic beacon transmits. Default: `30`
bcn_satid	0x04	uint16	Identifier to stamp into each beacon to show from which satellite the beacon originates. Default: `1`
persist_intv	0x08	uint32	Interval for persisting table samples. Default: `900` Unit: `second`
persist_en	0x0c	bool	Flag to completely enable/disable persisting of table samples. Default: `True`
data_path	0x0d	string	Filesystem path used by the housekeeping system to read and write data. Default: `/flash/hk`
hk_en	0x3f	bool	Flag to completely enable/disable the housekeeping system. Default: `True` Reboot Required: `Yes`

3.11.11 Commands

hk_srv

The *hk_srv* command group provides a number of commands that are useful as examples, for debugging and for simple operation.

Table 3.14: Command Table 'hk_srv'

Command	
hk_srv reload	Reload configuration from files.
hk_srv options show	Show current option values.
hk_srv beacon show	Show information on one or all loaded beacon specifications. Arguments: `[type]`: Beacon type, if omitted all known beacons specifications are briefly listed. `[verbose]`: Show verbose information about a specific beacon.
hk_srv beacon samplerate	Temporarily set the samplerate of a beacon (highest, high, medium, low, lowest), will revert on next reboot or reload. Arguments: `<type>`: Beacon type to set samplerate on. `<samplerate>`: Samplerate to set, can be highest, high, medium, low or lowest.

Continued on next page

Table 3.14 – continued from previous page

Command	
hk_srv beacon policy	Temporarily set the autobeacon policy (always, complete, off), will revert on next reboot or reload. Arguments: <type>: Beacon type to set policy on. <policy>: Policy to set, can be always, complete or off.
hk_srv table show	Show information on one or all sampled tables. Arguments: [node]: The source node of the sampled table. [id]: The source id of the sampled table. [verbose]: Show verbose information about a specific table.

hk

The *hk* command group provides the following commands:

Table 3.15: 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.

3.12 JSON parser (libjson)

3.12.1 Introduction

The JSON parser library provides an ultralightweight JSON parser written in ANSI C.

The parser is based on the open source parser: https://github.com/DaveGamble/cJSON, extended with some helper functions for validating values based on a specific type.

License:

```
Copyright (c) 2009-2017 Dave Gamble and cJSON contributors

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

(continues on next page)

(continued from previous page)

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

3.13 Log (liblog)

3.13.1 Introduction

The Log library contains Log appenders that plugs-into the Gomspace logging framework provided in by the Utility library.

Provided appenders:

- *Memory Log Appender*

3.13.2 Memory Log Appender

The memory log appender implements a Log Appender that appends log entries to a user specified memory buffer. The memory buffer can be located in any directly mapped memory area, or an area that is mapped through the VMEM interface. The memory log appender is hence able to be used on RAM, FRAM, Flash, etc.

The memory log appender is a ring buffer appender, meaning that when the end of the buffer is reached the buffer will wrap, overwriting the oldest log entries.

The memory appender stores the data in the following way:

Index	Data

The first 4 or 8 bytes is used to store the ring-buffer index. The data field is a ring-buffer where each data element (log entry) is stored.

Len	Element (Log entry)	Len

Each packed element is stored with a length field before and after the data (log entry). This makes is easy to skip either forwards or backwards in the data. It also acts as a sanity check to the parser to help detect corruption in the data.

The following interfaces of the Log Appender API is implemented by the memory log appender:

API	X	Comments
init()	X	Initializes the memory buffer for logging
append()	X	Appends a log entry to the buffer
append_isr()		ISR logging not supported by the memory log appender
info()	X	Provides basic info on the memory log appender
hist()	X	Reads back log entries from the memory log appender

3.14 Nano Protobuf (libnanopb)

3.14.1 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.
```

3.15 Parameter System (libparam)

3.15.1 Introduction

The Parameter System is a light-weight parameter system designed for GomSpace satellite subsystems. It is based around a logical memory architecture, where every parameter is referenced directly by its logical address. A backend system takes care of translating addresses into physical addresses.

The features of this system includes:

- Direct memory access for quick parameter reads.
- Simple data types: uint, int, float, double, string.
- Arrays of simple data types.

- Supports multiple stores per table, e.g. FRAM, MCU flash, file (binary or text).
- Remote client with support for most features (rparam).
- Packed GET, SET queries, supporting multiple parameter set/get in a single request.
- Data serialization and deserialization.
- Supports both little and big-endian systems.
- Commands for both local (param) and remote access (rparam).
- Parameter server for remote access over CSP.
- Compile-time configuration of parameter system

Basic Principle of operation

The parameter system is a generic system that translates a memory area into network messages and vice-versa. It has two basic operations: Read or Write. This basic operation is how most memory works and is something that most programmers can understand. Therefore the parameter system have been designed to work just like you would expect a memory to work. However it also throws in some more added benefits such as parameter tables, serialization, arrays, data locking and intuitive client interfaces. To best demonstrate this principle, let's take a simple example:

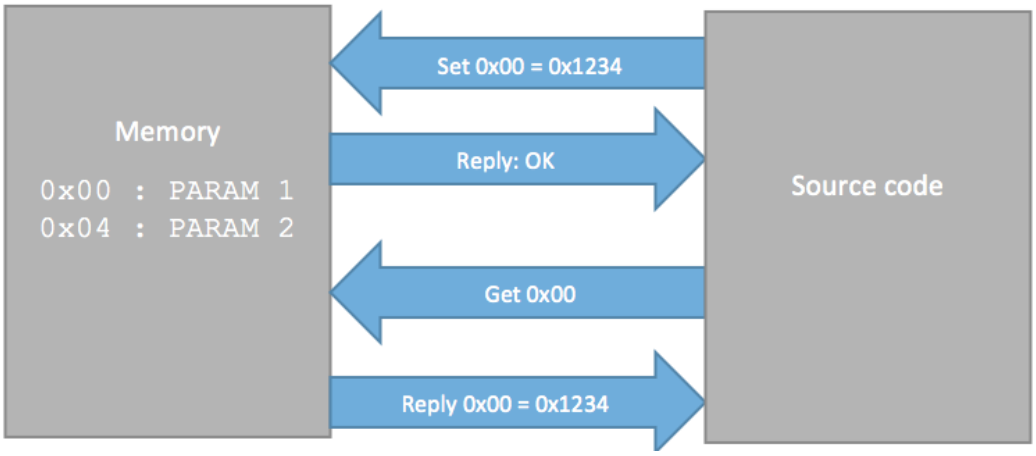


Fig. 3.7: Example of Set and Get commands

In this example, a set command is sent from the client to the memory, to update the parameter with address 0x00. The response is an OK message. Later the client issues a get command to the memory and receives the value that was stored. While this example shows the basic principle, it is of course pretty difficult for the programmer to be able to remember the address 0x00 of a given parameter. Therefore the parameter system only uses addresses internally. Externally to the programmer, only parameter names are used. The next example will show how parameter tables work:

Parameter tables

In order to translate from names to addresses, a parameter table is used:

Table 3.16: Parameter table example

Address	Name	Type	Description
0x00	param1	U32	Parameter 1
0x04	param2	U32	Parameter 2

The complexity of packing and unpacking data into messages is now handled by the parameter system based upon a pre-shared parameter-table. This means that a command such as ‘set param1 to 0x1234’ can easily be translated into a request by the client.

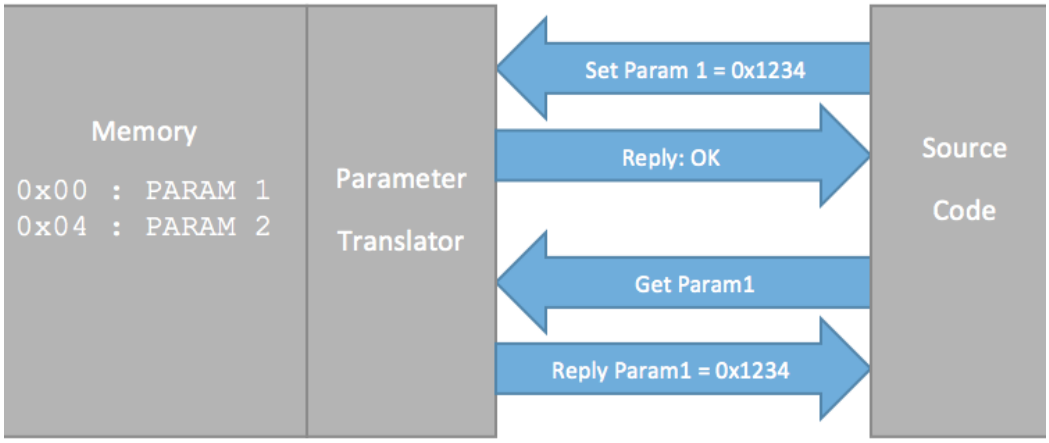


Fig. 3.8: Parameter Translator example

Here the first abstraction have been performed by the parameter translator. This now uses the address and type of the parameter ‘param1’ to generate a C-function that can be called by the source code. This C-function will ensure proper locking of the memory while the parameter is read and make sure that the data is also in the correct endian and datatype when returned.

Network service

In order for the parameter system to work across multiple nodes, a client/server network API has been added. The client and server shares the same serialization core, which is based on the parameter tables shown above. This means that a simple request can be sent over the network in a simple manner, without having to write any specific software client for a given command.

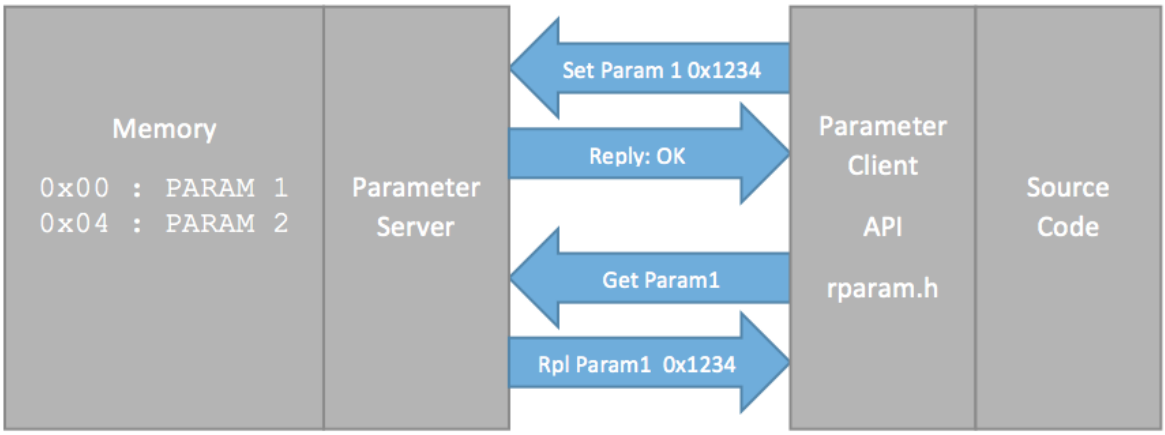


Fig. 3.9: Network Example

The parameter client can now use a pre-shared parameter table from a remote system to generate lists and change parameters on remote systems.

3.15.2 Table storage

The final thing that the parameter system needs is a method to store configurations into a persistent location, e.g. non-volatile memory. The parameter system is not limited to one or two types of stores, depending on the hardware available a number of table stores will be available.

Logical stores and initialization sources

To help describe schemes and recommendations for table storage, fallback and initialization we define a number of logical store types and initialization sources.

Table 3.17: Logical store types and initialization sources

Name	Description	Hardware
Volatile Store	Working copy of the parameter table.	Volatile memory / RAM.
Persistent Store	Persistent storage for saving parameter table values accross reboots.	FRAM without any write protection.
Protected Store	Persistent storage that is write protected by a hardware (or software) mechanism.	FRAM area with support for enabling/disabling write protection.
Backup Store	An alternative to Persistent Protected when no write protection mechanism exists. Basically another copy of the table.	FRAM without any write protection.
Read-only Store	Persistent storage that can only be read while 'in-flight'. It can be programmed/written on ground to set mission/satellite specific values.	Internal MCU flash. Provides backup if FRAM circuit fails completely.
Initializer Function	A C-function implemented by the application that sets default values in the table.	N/A
Auto Persist Store	Special storage area for parameters that are always written to persistent storage when set. It is only recomended to use this for telemetry as values are not protected by a checksum. See section below for details.	Dedicated area of an FRAM.

Fallback examples and recommendations

Connection configuration Configuration that affect how a module connects to the rest of the satellite bus is paramount. If the module is unable to communicate on the satellite bus using fallback configuration the module is lost in the case of corrupted configuration.

The recommended configuration stores scheme for such parameters is described by *Connection configuration initialization*. During initialization, if the table copy in 'Persistent Store' is corrupted initialization fails and an attempt is made to initialize from the hardware protected 'Protected Store'. If this also fails, e.g. FRAM circuit is defect, table initialization is done from the 'Read-only Store' as a final backup.

This scheme offers protection against corruption of unprotected FRAM and has default values defined by the mission in case of total FRAM failure.

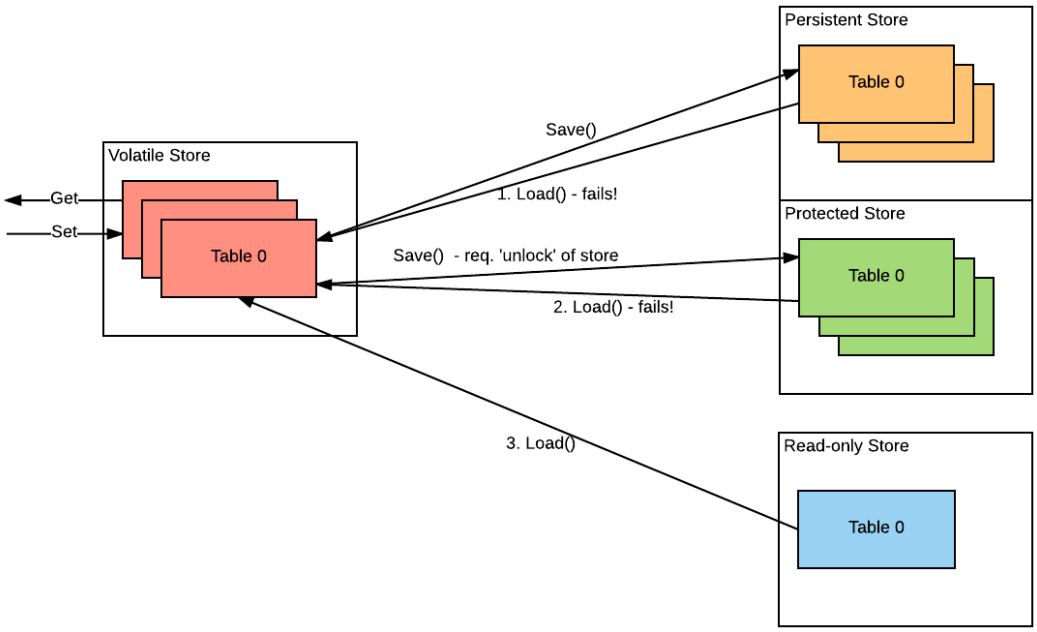


Fig. 3.10: Connection configuration initialization

Ground link configuration The GomSpace AX100 UHF/VHF radio protects ground link configuration using a ‘ground watchdog’ and a ‘Protected Store’. If the ‘ground watchdog’ is not maintained by a signal from the ground station the AX100 radio will invalidate configuration in the ‘Persistent Store’, reset and initialize from tables in the ‘Protected Store’.

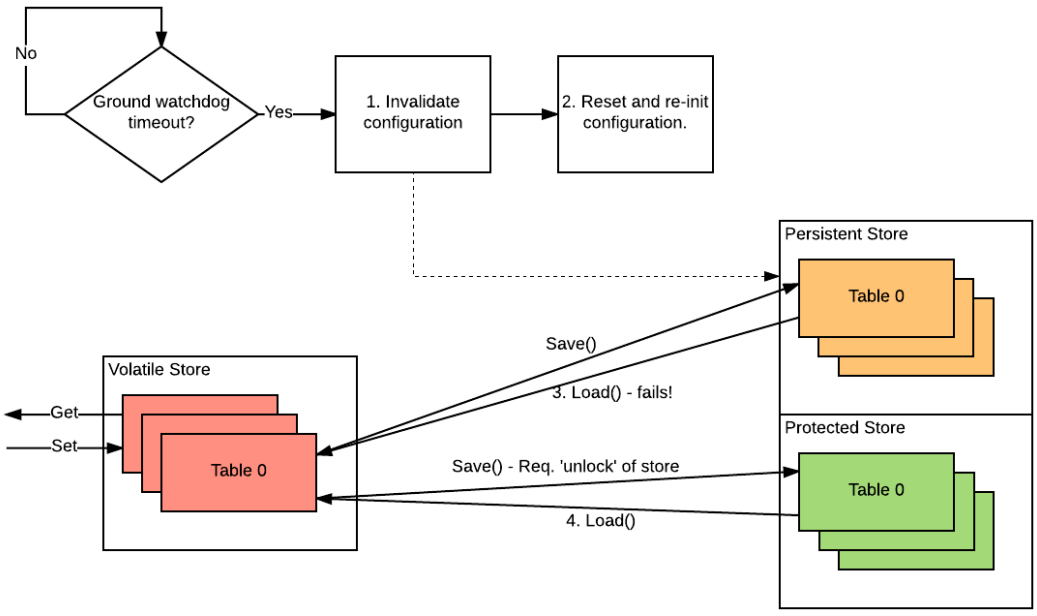


Fig. 3.11: Ground watchdog configuration fallback

Telemetry Store

The FRAM chip is particularly good at storing rapidly changing persistent parameters. This could for example be a boot counter. If a parameter is marked with the GS_PARAM_F_PERSIST flag in the parameter table and an auto-persist store has been setup for the table, every write operation to a persistent parameter will yield an update of the value stored in FRAM. After a reboot the FRAM will be checked for any stored auto-persist parameters and the value stored in RAM will be initialized from that. The advantage of using persistent parameters is that they are relatively cheap to update and does not require any checksums or similar. That means that they can be used for rapidly changing values without worrying about execution speed and storage wear. The disadvantage is that during initialization there is no checksum on auto-persist parameters and corrupted data may be loaded into the table. It is therefor **not** recommended to use auto-persist parameter for configuration.

3.15.3 System overview

The parameter system is separated into three components:

- **Parameter Host API (libparam)**: Local interface used by the firmware on a subsystem.
- **Parameter Service (libparam)**: Defines the network interfaces and implements a CSP service task to process incoming get/set requests.
- **Parameter Client API (libparam_client)**: Contains functions for sending get/set requests to a parameter service.

On top of these components lies a client layer where there is two GOSH interfaces:

- **Local interface 'param'**: This is a set of commands that uses the Host API to modify parameters. This provides access to changing parameters. save/load tables, wihtout going through the service component.
- **Remote client 'rparam'**: This is a set of commands that uses the Client API to send requests to a remote parameter service. This can be used to remotely operate another subsystem over a CSP link. For example the spacelink.

This can be illustrated in the following way:

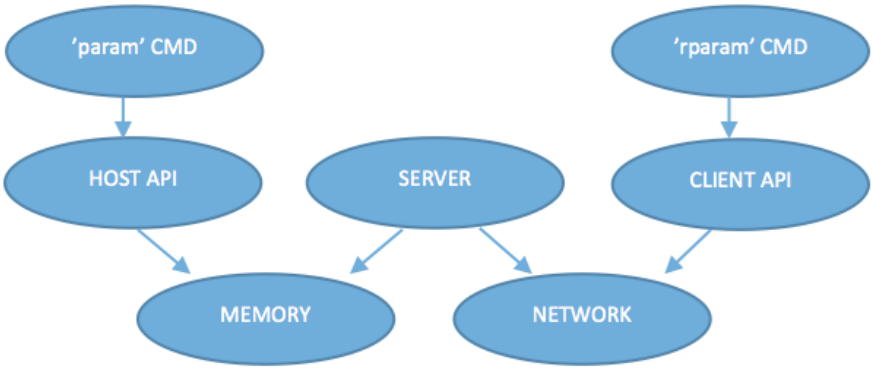


Fig. 3.12: Parameter system modules

3.15.4 Commands

param

The *param* commands can be used to modify the local paramters of a system, without any form of communication. The commands are grouped under *param* and provides following commands:

```
nanomind # param
Local Parameter System
  select              Select working table
  list                List all parameters
  tableinfo           Show table information
  export              Export parameters to stdout
  set                 Set parameter value
  get                 Get parameter value
  load                Load table
  save                Save table
  storeinfo           Show store information
  clear               Clear/invalidate store slot
  lock                Lock a store
  unlock              Unlock a store
```

To get and overview of all tables in the local system:

```
nanomind # param tableinfo
id  name
  0 board
  1 configuration
  4 telemetry
 21 hkcfg
```

To view an entire table, use the *list* command:

```
nanomind # param list configuration
Table configuration (1):
  0x0000 gssb1_pwr_en    BL  false
  0x0001 gssb2_pwr_en    BL  false
  0x0002 flash_pwr_en    BL  false
  0x0003 pwm_pwr_en      BL  false
  0x0004 swload_img1     STR ""
  0x0024 swload_cnt1     U16 0
  0x0026 swload_img2     STR ""
  0x0046 swload_cnt2     U16 0
  0x0048 fs_default      U8  0
  0x0049 fs_en_p0        BL  true
  0x004A fs_en_p1        BL  true
  0x004B log_fram_en     BL  false
  0x004C log_mask        STR ""
  0x008C tsync_node      U8  0
  0x008E tsync_intv      U16 60
  0x0090 spi0_cs_mode    U8  0
```

To modify a parameter, *select* the table and *set* the parameter:

```
nanomind # param select configuration
nanomind # param get swload_img1
  swload_img1 = ""
nanomind # param set swload_img1 /flash/nanomind_ram.bin
nanomind # param get swload_img1
  swload_img1 = "/flash/nanomind_ram.bin"
```

Once the table is configured, *save* the table:

```
nanomind # param save configuration
```

rparam

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

```
csp-client # rparam
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.

3.16 Remote GOSH (librgosh)

3.16.1 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*.

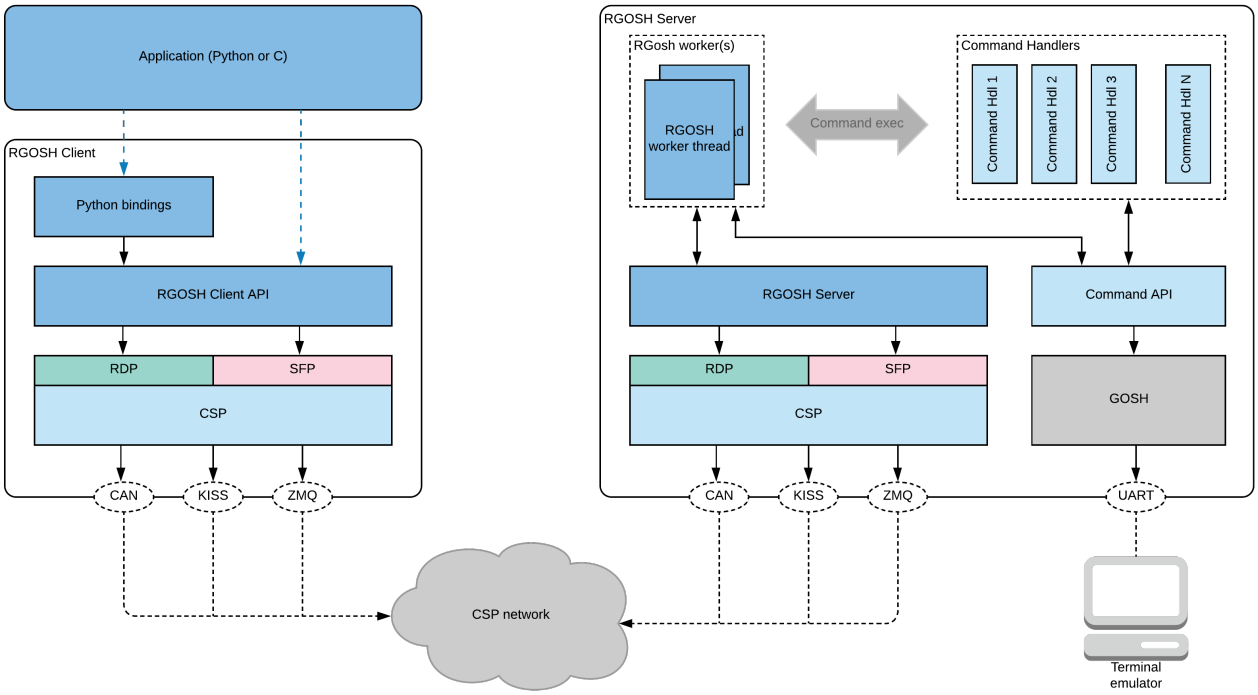


Fig. 3.13: Using Remote GOSH for commands execution & feedback

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

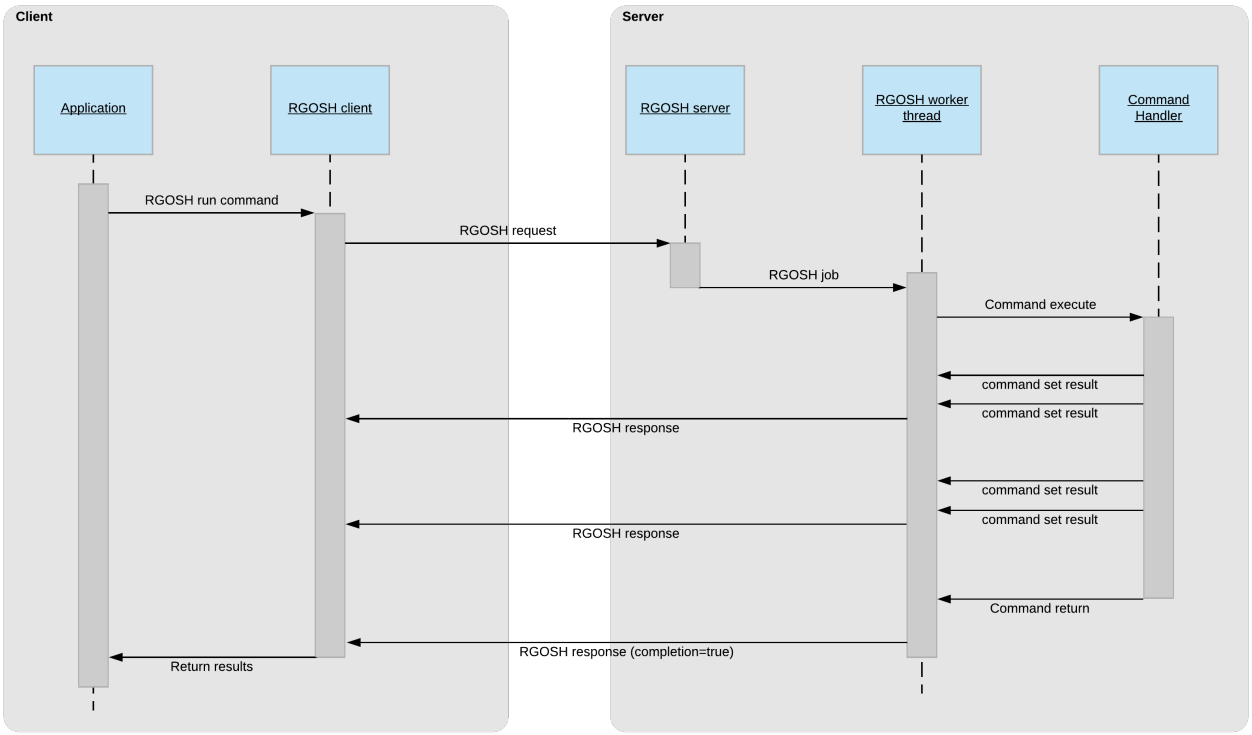


Fig. 3.14: Sequence diagram showing the use of Remote GOSH.

3.17 Storage (libstorage)

3.17.1 Introduction

The Storage library contains implementations of embedded file systems (currently UFFS) and a Virtual File System layer (VFS) that provides an abstraction of the physically file system (e.g. UFFS).

VFS - Virtual File Systemm

The VFS layer ensures that code using the file system can be written in a portable manner, using the standard C library file access primitives such as `fopen`, `fread`, and `fwrite`.

A number of commands provides access to the file system, see *Commands*.

The figure below shows the call chain from standard C library functions to the physical file system layer.

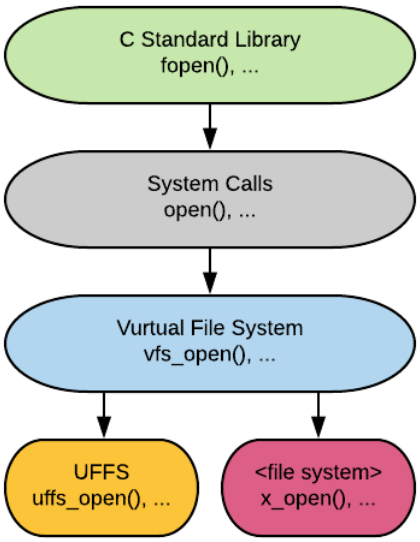


Fig. 3.15: File System interfaces

The VFS can be extended with multiple file systems, by adding additional partitions in the file: `src/vfs/vfs_partitions.c`. The function `vfs_mount` handles all initialisation and mounting of the file systems.

UFFS - Ultra low cost Flash File System

Ultra low cost Flash File system for embedded systems, designed for NAND and NOR flash called UFFS.

UFFS manages all the pages and blocks on the flash directly and will ensure an even load of the flash pages. This makes it much more robust and it can theoretically survive a power-cut at any time.

If a problem occurs during the writing of a flash page, UFFS will mark the entire block as a bad block, and not attempt writing to it again. The user must be aware that at some point, all blocks could be marked as bad and the file system will mount without any storage capacity. In such a case, user intervention is required in order to try to recover bad blocks. The easiest way to reset the bad blocks is to format the file system. In the event that all blocks are damaged, which will happen once the flash medium wears out, the file system will be unable to mount. In this case a reformat and reboot will put the system into a reboot-loop, and potentially render the OBC responseless. A safe mechanism to be able to execute without file system is therefore recommended.

In order to use the UFFS file system directly, use the `uffs_open`, `uffs_read` etc. API functions. You can also use the Newlib C-library as a wrapper (and cache) with the `fopen`, `fread` etc. calls. If you wish to keep your code portable, the POSIX API is the best choice. The POSIX api is in `stdio.h` and the UFFS API in `uffs/uffs_fd.h`

Note: The UFFS implementation has some limitations to the usage. The maximum number of open files are 10 and maximum dir handles 5. Furthermore the page cache of a file is kept per file handle, so two simultaneous file handles to the same file is not possible. Finally, if a file handle is shared between two tasks, external locking will be required since the UFFS is not thread safe when sharing handles. Generally sharing files should be avoided if possible.

Note: The UFFS cache will not automatically flush data to the file system, so to ensure your data is written use `fsync()` or `uffs_sync()` after writing. Closing the file will also trigger a cache flush.

3.17.2 Implementation

Initialization of the storage layer and mounting of file systems takes place during startup of the system. It is recommended to perform the mounting of the file system in a separate task with low priority, to avoid blocking the system while booting.

The time it takes to mount a file system depends on the usage of the partition, and can take several minutes if the device is full. Empty or sparsely used file systems usually mount in less than 30 seconds.

For further details, please see the respective SDK, e.g. *A3200 SDK*.

3.17.3 Commands

The Storage library includes a number of commands for debugging the file system. Most commands are named to match the corresponding Linux shell commands, e.g `ls`, `rm`, `cp`.

Command	Description
uffs	UFFS specific commands
testfs	Test R/W-speeds
ls	List files in directory
rm	Remove file
mv	Move or rename file
cp	Copy file
mkdir	Make directory
touch	Create empty file
cat	Show the contents of a file
append	Append data (basic file editor)
chksum	Calculate CRC32 checksum of a file
hexdump	Show the contents of a file in hex format
mkfs	Recreate file system
df	Show file system usage

Like all commands, the expected command arguments can be listed by entering the command name and pressing the `<tab>` key.

The `df` command is used for showing the status of the mounted file systems:

```
nanomind # df
File system    Size    Used   Avail  Use% Mounted on
uffs          62.0M  992.0K   61.0M    1% /flash0
uffs          62.0M    0.0B   62.0M    0% /flash1
uffs          62.0M  992.0K   61.0M    1% /flash -> flash0
```

The `mkfs` command can be used to recreate a file system. **NOTE: this will permanently erase all stored files**. To use it, simply run `mkfs <partition>`, where the partition string is the mountpoint without preceeding slash, i.e. `mkfs flash1` to recreate the UFFS file system on A3200. Be aware that recreating the file system can take up to 2 minutes:

```
nanomind # mkfs flash1 1
```

`testfs` can be used to verify the performance of the file system. This test is run on an A3200, just after the file system has been recreated using `mkfs flash1 1`:

```
nanomind # testfs /flash1/test1
Wrote 100000 bytes in 1156 mS (86 KBytes/sec)
Read 100000 bytes in 959 mS (104 KBytes/sec)

nanomind # ls /flash1
test1            97.7K
```

3.18 Thirdparty (libthirdparty)

3.18.1 Introduction

The Thirdparty library contains low-level drivers for 3rd components.

The Thirdparty library provides a C API for interfacing with the external component using SPI, I2C or UART (serial).

Features

Following drivers are supported:

- FRAM components
  - FM33256B, 256-Kbit (32 K × 8) Serial (SPI) F-RAM, with RTC and Watchdog.
  - CY15B102Q, 2-Mbit (256 K × 8) Serial (SPI) Automotive F-RAM.
- RM3100: Geomagnetic Sensor (magnetometer)
- Novatel_615: Dual-Frequency GNSS Receiver (GPS)
- AF_wheel: Astrofein Wheel
- HTPA: Heimann Thermophile Array HTPA32x32dR1L2
- Temperature sensor
  - Texas Instruments TMP100
  - LM71
- FLASH
  - S70FL01GSAGMFI011, 1 Gbit (128 Mbyte) 3.0V SPI Flash.
- Hyperion ST200 star tracker
- INA226
- ISIS: Antenna system

3.19 Utility (libutil)

3.19.1 Introduction

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

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

Features

- Time
  - Time get and set
  - Delay and sleep
  - Time conversions
  - Timestamp formatting and parsing

  - Real Time Clock (RTC) interface
- Threading
  - Queue
  - Mutex
  - Thread
  - Semaphore
  - Software watchdog
- String
  - Number to string
  - Other string parsing and creation
- Checksum/Hash
  - Fletcher16
  - CRC8
  - CRC32
- Command (GOSH)
  - Console
- Logging
  - Groups
  - Appenders
- Drivers
  - SPI
  - I2C
  - CAN
  - GPIO
- Zip file creation and extraction

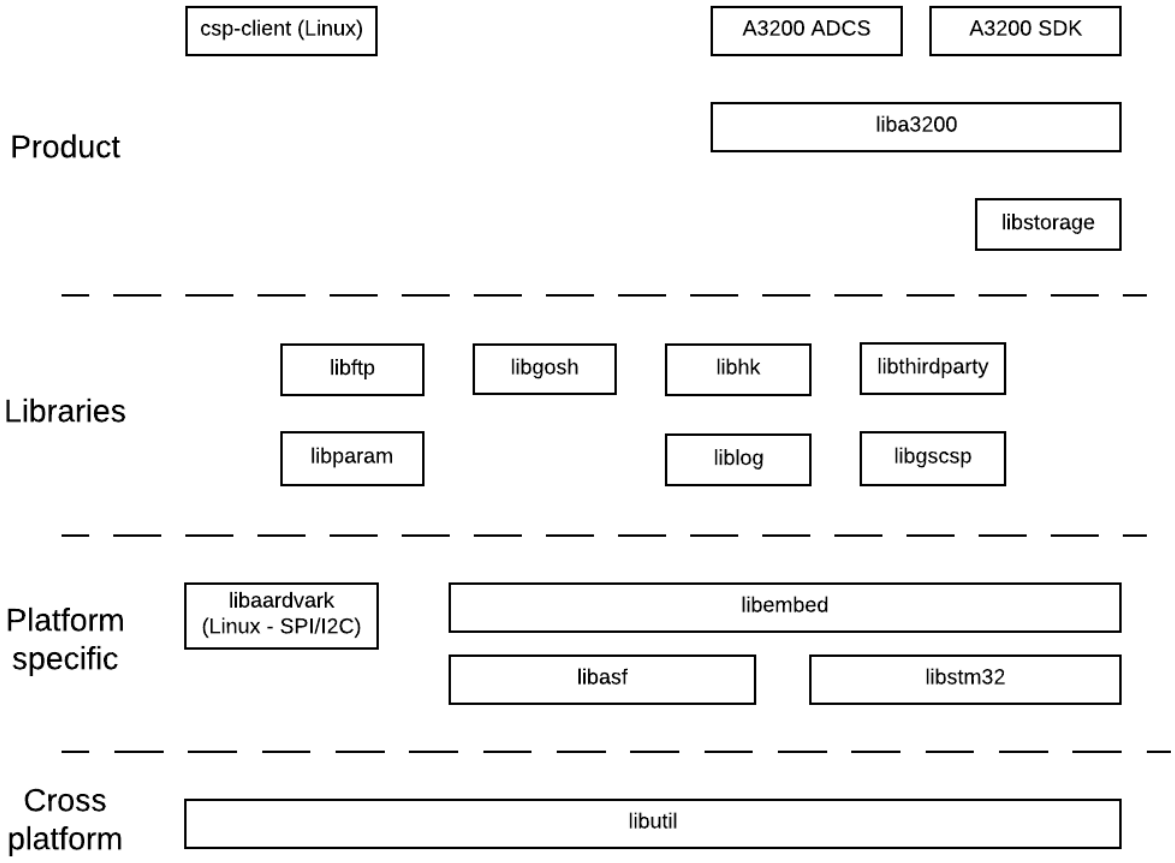
3.19.2 Software concepts and terminology

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

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

Software Stack

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



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

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

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

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

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

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

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

Client / Server

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

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

Directory Structure

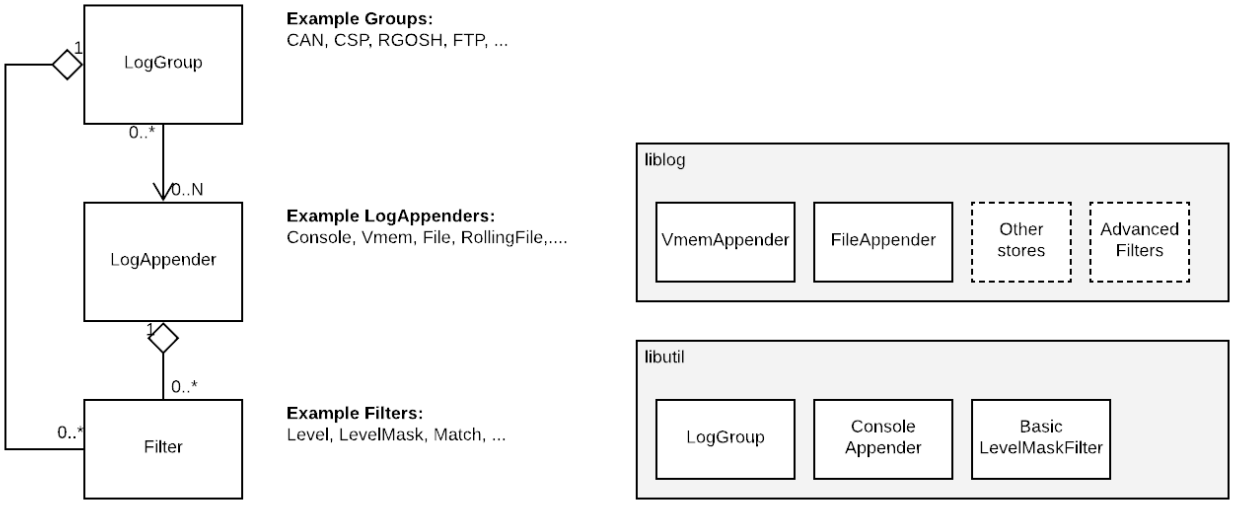
Modules are structured in the same way, which makes it easy to find the relevant files.

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

3.19.3 Logging framework

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

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



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

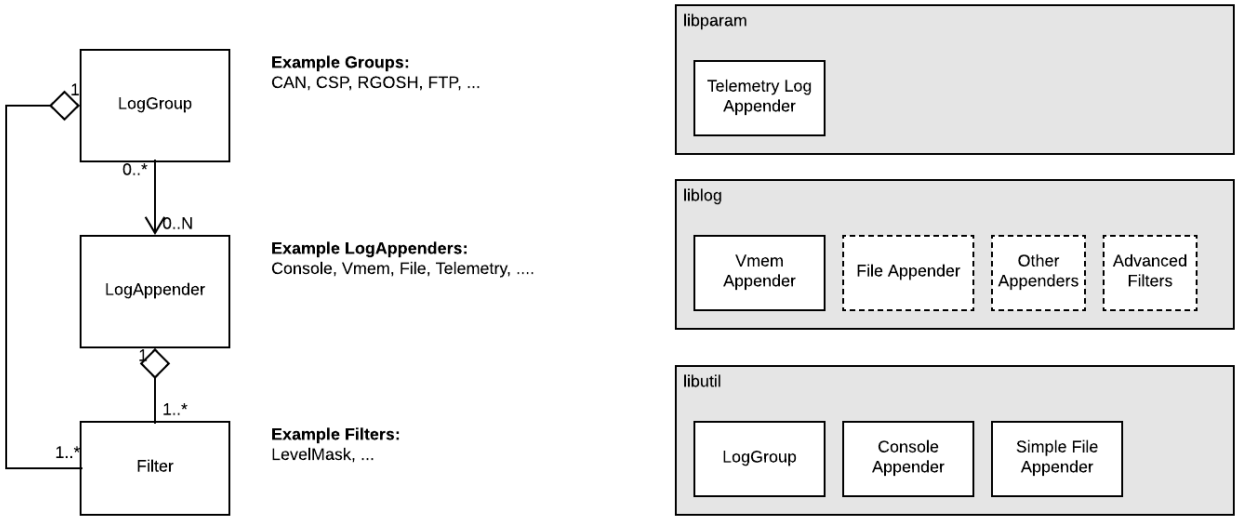
Each LogGroup can reference any number of LogAppenders (stores). The LogAppenders is responsible for writing the actual log statement to the “store”. The LogAppenders can also choose to filter the log statement depending on the associated filter. Currently the only supported filter of the LogAppender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

Logging hierarchy

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

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

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



The above figure shows three categories of LogAppenders:

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

Logging example application

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

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

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

(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;
}
```

3.19.4 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.

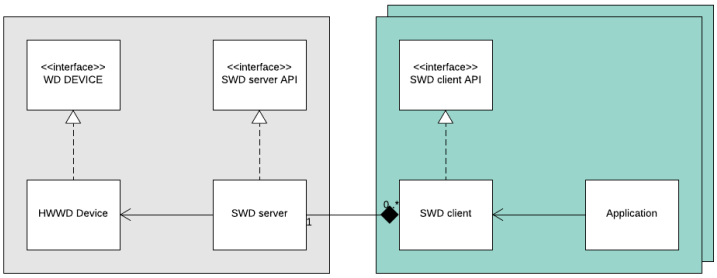


Fig. 3.16: Software Watchdog Client/Server modules

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.

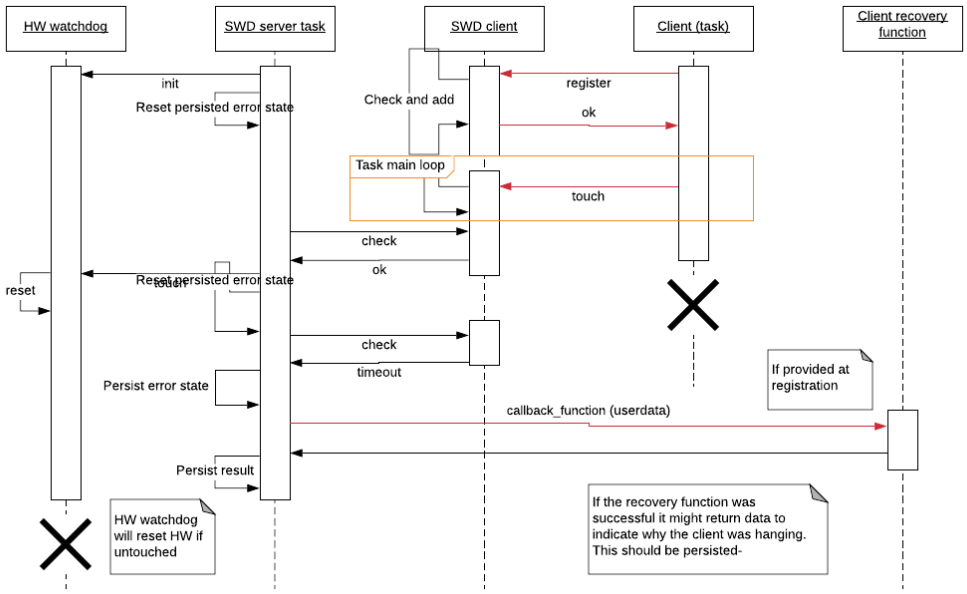


Fig. 3.17: Software Watchdog Client/Server sequence diagram

3.19.5 Command (GOSH)

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

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

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

Console

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

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

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

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

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

Table 3.18: 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:

```
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)

(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;
}
```

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*.

3.19.6 Commands

Miscellaneous commands

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

Table 3.19: 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 3.20: 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

Table 3.20 – 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 3.21: 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.

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)
```

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

5. Appendix

5.1 Software Changelogs

5.1.1 A3200 SDK

2.12.0 (2020-06-12)

- Improvement: Updated libcsp to new 1.6.0 (libgscsp 2.x->3.x)
- Improvement: Updated libraries liba3200, libasf, libftp, libhk
- Feature: house-keeping tables for BPX, eps etc default to invalid values if read fails

2.11.2 (2020-04-22)

- Improvement: Updated libs (libasf, libftp and more)
- Feature: Mission-lib is always included in SDK (No longer referred)

2.11.1 (2020-03-20)

- Improvement: Changed size summary to sysv format to get details on section sizes.
- Bug: BPX temperatures are in degrees celcius (℃)
- Improvement: Updated libraries.

2.10.1 (2019-10-18)

- Limitation: if enabling FreeRTOS configUSE_NEWLIB_REENTRANT (default), use AVR32 toolchain: 3.4.2_gs1 or later - otherwise crash occurs, when a task terminates (libembed).
- Limitation: Disabled FreeRTOS runtime stats until implemented for FreeRTOS 10.2.0.
- Improvement: Implement retargetable locks for newlib file operations. This allows for multi-threaded file operations. Requires toolchain 3.4.2_gs1 or later (libembed).
- Feature: Added generic scheduler for collecting telemetry from nodes, not supporting rparam internface.
- Bug: FreeRTOS: Enabled re-entrant per task/thread (configUSE_NEWLIB_REENTRANT=1 in FreeRTOSConfig.h) to prevent crashes when printing floats.
- Bug: Updated toolchain with patched newlib, that supports file-locking, prevent possible file corruption if multiple tasks accesses the file system.
- Improvement: Updated libraries and documentation.
- Breaking: Removed Health System (libhealth).

2.9.1 (2019-07-03)

- Bug: Updated to FreeRTOS 10.2.0 to fix crash issue, when running from RAM image.
- Limitation: Disabled FreeRTOS runtime stats until implemented for FreeRTOS 10.2.0.
- Improvement: Updated libraries and documentation.

2.8.2 (2019-06-12)

- Improvement: Updated libraries and documentation.

2.8.1 (2019-05-20)

- Improvement: Updated libraries and documentation.

2.7.1 (2019-04-11)

- Improvement: Updated libraries and documentation.

2.6.1 (2019-02-27)

- Improvement: Updated libraries and documentation.
- Feature: Added technology preview of new Health System (libhealth).

2.5.1 (2019-01-30)

- Improvement: Updated libraries and documentation. Log groups must be registered in order to change level.

2.4.2 (2018-12-20)

- Improvement: Updated libraries and documentation.

2.4.1 (2018-11-23)

- Bug: Missing avrisp.rules file was added to dist. tarball.
- Feature: Moved initialization of flight-planner to after file system mounted.
- Feature: Updated libraries.

2.3.1 (2018-08-14)

- Feature: Updated libraries.
- Feature: Use gs_csp_service_dispatcher(), instead of server task in liba3200.

2.2.2 (2018-05-31)

- Feature: Updated libraries.

2.1.1 (2018-04-18)

- Feature: Changed param 4 store configuration to use stores defined in liba3200, libhk.
- Feature: Added Doxygen generated API documentation to html manual.
- Feature: Added default clients.
- Breaking: The format used to store paramters in FRAM has been changed, preventing load of existing data.

2.0.1 (2018-03-14)

- Breaking: Removed libclients, replacing it very limited set of clients added.
- Breaking: updated all libraries to latest version, e.g. libparam version 4.

2.0.0 (2017-09-06)

- Feature: Use gsbuildtools for building and generate documentation.

1.1 (2017-03-08)

1.0 (2017-01-10)

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)

- 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)

5.0.4 (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.

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.

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”
- 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)

- 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

A3200 (liba3200)

4.0.1 (2020-06-02)

- Breaking: Updated to libgscsp 3.0.

3.8.2 (2020-03-20)

- Improvement: Optimized waiting on file system to be mounted (using a semaphore).

3.8.1 (2019-12-17)

- Feature: Added commands for onboard power switches.
- Bug: hmc5843 selftest was dependent on prior command sent to hmc5843.

3.7.1 (2019-10-16)

- Improvement: Increase delay between read in gyro self test to allow gyro to stabilize.
- Improvement: Fail gyro self test if FT value is zero.
- Improvement: Added hmc5843 selftest
- Improvement: Added configUSE_NEWLIB_REENTRANT=1 in conf/FreeRTOSConfig.h to prevent crash from multiple tasks/thread sprintf floats.
- Improvement: Added gs_a3200_init_set_heap(), which controls location of heap (internal or external).
- Feature: Added commands for reading LM71.
- Improvement: Added gs_a3200_get_default_stack_size() as weak function, to allow stack size tuning in order to run in internal memory.
- Improvement: Generalized gs_a3200_init_twi() and gs_a3200_init_can() and made them public.

3.6.1 (2019-07-03)

- Feature: Append “(RAM)” to model, when compiling RAM image.
- Limitation: Disabled FreeRTOS runtime stats until implemented for FreeRTOS 10.2.0.
- Improvement: Improved FileSystem logging during boot/mount (logs goes to FRAM if enabled).

- Breaking: Removed gs_s3200_driver_register_commands(), replaced by libembed gs_embed_register_driver_commands().

3.5.2 (2019-06-12)

- Feature: Only initialize KISS UART, if KISS is enabled (in the board table).
- Improvement: Always build commands.

3.5.1 (2019-04-09)

- Feature: Added support for configuring log groups/appenders, using the “log_mask” parameter in the “configuration” table.

3.4.1 (2019-02-26)

- Breaking: Renamed LM70 to LM71.

3.3.6 (2018-01-30)

- Breaking: Removed A3200_DOCK_TABLE from A3200 protected storage.
- Improvement: Re-arranged boot order to enable persistent logging earlier.

3.3.5 (2018-12-19)

- Improvement: Updated documentation.

3.3.4 (2018-11-21)

- Improvement: Reflect actual Flash mounting status in telemetry.

3.3.3 (2018-09-21)

- Feature: Integrated latest liba3200dock and libthirdparty.

3.3.1 (2018-08-13)

- Breaking: dock_type in board table change from uint8_t to string.
- Improvement: spn_fl512s_read_data clears return buffer in case of SPI timeout instead of returning random data.
- Breaking: Cleaned up options and re-organized code to better support BSP and SDK builds (minor code changes).
- Breaking: Moved linker-scripts to conf (standard).
- Breaking: Removed server task - moved to a3200-sdk.
- Feature: Removed KEEP from linker-scrips..
- Breaking: Restructured - moved fl512s driver to libthirdparty and moved UFFS mounting procedure to libstorage.

3.2.1 (2018-05-30)

- Breaking: Refactored hmc5843 API, prefixed with *gs* and return gs_error_t error codes.
- Breaking: In the telemetry table boot_count and boot_cause is renamed to bootcount and bootcause to adhere to standard
- Breaking: In the configuration table fs_disable flag is removed, this invalidates FRAM copies of this table.
- Breaking: Changed spn_fl512s return codes to gs_error_t.

3.1.1 (2018-04-18)

- Breaking: Changed uptime in telemetry from mS to seconds.
- Feature: Added param 4 store configuration.

3.0.1 (2018-03-14)

- Breaking: Changed table layout.

- Breaking: Changed hook interface - provide callbacks before and after file system is mounted.
- Feature: Use gsbuildtools.

A3200 (liba3200_client) 3.5.3 (2019-12-17)

- Improvement: Updated documentation.

3.5.2 (2019-10-16)

- Improvement: Updated documentation.

3.5.1 (2019-04-09)

- Improvement: Updated dependencies.

3.3.7 (2019-02-26)

- Breaking: Renamed LM70 to LM71.
- Feature: configuration table, changed default value for logging to FRAM to true.

3.3.6 (2019-01-30)

- Improvement: Updated API documentation.

3.3.5 (2018-12-19)

- Feature: Added support for disabling mounting of Flash 0 and 1.

3.3.4 (2018-11-21)

- Feature: Added support for disabling mounting of Flash 0 and 1.

3.3.2 (2018-09-20)

- Feature: Added SPI slave ids for LM71 (1) and extern gyro.

3.3.1 (2018-08-13)

- Breaking: Renamed tables: conf -> configuration, telem -> telemetry.
- Breaking: Removed CSP port definitions, use defines in libgscsp.
- Breaking: Restructured - moved fl512s driver to libthirdparty.

3.2.1 (2018-05-30)

- Improvement: added logical SPI slave definition for A3200 boards.
- Breaking: spn_fl512s - changed return codes to gs_error_t.

A3200 DOCK (liba3200dock)

2.5.2 (2019-05-19)

- Improvement: Include source code in distribution.

2.5.1 (2019-04-09)

- Improvement: Updated dependencies.

2.4.4 (2019-01-29)

- Breaking: Removed protected store of dock table for ADCS3, ADCS6 and ADCS8.
- Improvement: Register all used log groups.

2.4.3 (2018-12-19)

- Improvement: Updated documentation.

2.4.2 (2018-11-21)

- Improvement: Updated API documentation.

2.4.1 (2018-09-20)

- Breaking: Changed initialization function to accept dock name, instead of reading it from liba3200 'board' table.

2.3.1 (2018-08-13)

- Feature: Added support for ADCS8 dock.

ASF (libasf)

3.6.3 (2020-04-20)

- Bug: Fix race condition in freertos port regarding systemcall handling.

3.6.2 (2020-03-20)

- Improvement: AVR32 TWIM driver, reduced number of interrupts to transfer data.
- Bug: AVR32 TWIM driver, stop sending data on bus, after no address/data acknowledge.
- Bug: AVR32 TWIM driver, fixed potential wrong status from twim_pdc_transfer() when accessed by multiple tasks.
- Bug: AVR32 Fixed possible stack corruption, by disabling optimization of FreeRTOS port.
- Improvement: AVR32 Added twim_pdc_transfer_with_timeout() for supporting user specified timeout.

3.6.1 (2019-10-16)

- Improvement: AVR32 TWIM/I2C: improved logging.
- Breaking: Removed newlib port/stubs for avr32 and samc (moved to libembed).

3.5.1 (2019-07-03)

- Feature: Added FreeRTOS 10.2.0.

3.4.1 (2019-04-08)

- Improvement: Updated tools dependencies.

3.3.6 (2019-02-26)

- Improvement: Removed/reduced configuration used for compile check's, not used on real targets.

3.3.5 (2019-01-28)

- Feature: Changed log group for twim/twis to "i2c".

3.3.4 (2018-12-19)

- Improvement: TWIM NACK warning moved to debug.
- Improvement: Updated documentation.

3.3.3 (2018-11-21)

- Improvement: Updated default log settings.

3.3.2 (2018-09-10)

- Bug: Removed log warnings from ISR, readded warnings on TWI NACK and ARB, enable idle interrupt on commandready

3.3.1 (2018-08-13)

- Improvement: Ensure twim_pdc_transfer release locks in case of error.

3.2.1 (2018-05-30)

- Breaking: AVR32: Moved exception handler to libembed.

- Improvement: Changed SPI driver use standard SPI log group.

3.1.1 (2018-04-17)

- Feature: SAMC21: Added Invalidate Cache command, missing in ASF implementation.

3.0.1 (2018-03-06)

- Breaking: Removed libcsp dependency
- Breaking: Moved GomSpace code to libembed.
- Feature: Added support fo AT32UC3C1512C.

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.
- 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.
- 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)
- new: Autogenerated python bindings using ctypesgen
- new: Task-less operation with router invocation from external scheduler function
- api: Refactor route_if_add to csp_iflist_add
- api: Refactor route_set and friends to rtable_set
- api: Refactor csp_fifo_qos to csp_qfifo
- api: Added defined to be backwards compatible with 1.x
- interfaces: Drop packets on LOOP interface not for own address (blackhole)
- interfaces: New ZMQHUB interface (using zeroMQ over TCP)
- other: Increase stack size from 250 to 1100 for csp_can_rx_task
- other: Cleanup in csp_route.c
- other: Show incoming interface name in debug message
- other: Remove newlines from debug calls
- improvement: Reduce debug hook function complexity with valist passing
- fix: csp_sleep_ms did not work

libcsp 1.3, 07-05-2015

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

libcsp 1.2, 25-10-2013

- Feature release
- New: CMP service for peek and poke of memory
- New: CMP interface statistics struct is now packed
- New: Faster O(1) buffer system with reference counting and automatic alignment
- New: Thread safe KISS driver with support for multiple interfaces
- New: CSP interface struct now holds an opaque pointer to driver handle
- New: removed TXBUF from KISS driver entirely to minimize stack usage, added TX lock instead
- New: Pre-calculated CRC table .romem or PROGMEM on __avr__
- New: Added buffer overflow protection to KISS interface
- New: Allow posting null pointers on conn RX queues
- New: Lower memory usage on AVR8
- New: csp_route_save and csp_route_load functions
- New: option –disable-verbose to disable filenames and linenumber on debug
- Protocol: KISS uses csp_crc32 instead of it own embedded crc32
- Improvement: Use buffer clone function to copy promisc packets
- Bugfix: Fix pointer size (32/16bit) in cmp_peek/poke
- Bugfix: Issue with double free in KISS fixed
- Bugfix: Change rdp_send timeout from packet to connection timeout to make sending task block longer
- Bugfix: Fix conn pool leak when using security check and discarding new packets
- Bugfix: Add packet too short check for CRC32
- Bugfix: Accept CRC32 responses from nodes without CRC support
- Bugfix: Ensure csp_ping works for packets > 256 bytes
- Bugfix: Cleanup printf inside ISR functions
- Bugfix: Do not add forwarded packets to promisc queue twice
- Bugfix: Fix return value bug of csp_buffer_get when out of buffers
- Bugfix: Always post null pointer with lowest priority, not highest
- Bugfix: Add check on debug level before calling do_csp_debug, fixes #35
- Other: Export csp/arch include files
- Other: Remove the use of bool type from csp_debug
- Other: Moved csp debug functions to csp_debug.h instead of csp.h
- Other: Ensure assignment of id happens using the uint32_t .ext value of the union, quenches warning

libcsp 1.1, 24-08-2012

- Feature release
- Defacto stable since Feb 2012
- New: I2C interface
- New: KISS interface
- New: USART drivers for Linux, Mac and Windows

- New: Windows/MinGW support
- New: MacOSX support
- New: Interface register function
- New: Interface search function
- New: CMP service for remote route updating
- New: CMP service for interface statistics
- Improvement: Better QoS support
- Improvement: Send RDP control messages with high priority
- Improvement: WAF distcheck now works
- Improvement: Automatic endian discovery
- Improvement: Accept packets with CRC32 checksum if compiled without CRC32 support
- Improvement: Do not wake the router task if RDP is not enabled
- Improvement: Save 102 bytes of RAM by packing route entries
- Cleanup: Simplify CAN configuration
- Cleanup: Move architecture specific code to src/arch
- Bugfix: CSP_MEMFREE gives wrong answer on freertos AVR due to truncation
- Bugfix: Fixed wrong 64-bit size_t in csp_service_handler
- Bugfix: Fixed problem in csp_if_kiss when out of buffers
- Bigfix: Handle bus-off CAN IRQ for AT90CAN128

libcsp 1.0.1, 30-10-2011

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

libcsp 1.0, 24-10-2011

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

Embed (libembed)

1.9.1 (2020-03-20)

- Breaking: ASF/AVR32: Removed deprecated/unused gs_spi_{read, write, setup_chip, select_chip, unselect_chip}.
- Bug: STM32: Fixed UART device 3 callback to make it usable.
- Improvement: AVR32 Locate can_mob_ram in named section, so it can be placed at HSB_SRAM.
- Improvement: AVR32: Use the timeout given by user for SPI / I2C
- Improvement: STM32: Optimized UART driver.
- Breaking: AVR32: Removed deprecated/unused gs_spi_set_cs_timeout().

1.8.1 (2019-10-16)

- Limitation: if enabling FreeRTOS configUSE_NEWLIB_REENTRANT (default), use AVR32 toolchain: 3.4.2_gs1 or later - otherwise crash occurs, when a task terminates.
- Improvement: Updated dependencies and documentation.
- Improvement: Close stdin/stdout/stderr to prevent memory leak in gs_thread_exit() (if configUSE_NEWLIB_REENTRANT)
- Breaking: AVR32: gs_mcu_flash API: Disable write to internal flash, if not running from MCU flash image.
- Improvement: AVR32: Changed gs_delay_us() to use cpu_delay_us()
- Improvement: Implement retargetable locks for newlib file operations. This allows for multi-threaded file operations. Requires toolchain 3.4.2_gs1 or later.
- Breaking: Consolidated newlib stubs for STM32, AVR32 and samc -> src/newlib/syscalls.

1.7.4 (2019-07-03)

- Feature: Added support for FreeRTOS 10.
- Bug: fixed minor issues in AVR32 CAN driver, e.g. return codes, ISR logging.
- Improvement: AV32 CAN driver statistics, command: `driver can <device>`.

1.7.3 (2019-06-13)

- Bug: Disabling and clearing GPIO interrupts before booting RAM image

1.7.2 (2019-05-23)

- Bug: Fixed bug in UART RX queue handling, which potentially could result in lost characters (present in all versions). Not present if using callbacks.

1.7.1 (2019-05-16)

- Improvement: STM32: Rewritten I2C driver to make it work under higher load on the MCU.
- Improvement: Include source code in distribution.

1.6.1 (2019-04-08)

- Improvement: Updated dependencies and documentation.

1.5.5 (2019-03-21)

- Bug: STM32: Fixed race condition in CAN driver.
- Improvement: STM32: SPI driver handles slaves faster and does not init GPIO when using chip select callback.

1.5.4 (2019-02-26)

- Bug: Removed AVR32 GPIO definitions due to some clashes with gpio.h in libasf.

- Improvement: STM32 linker-scripts: Removed KEEP for log groups.

1.5.3 (2019-01-29)

- Improvement: Register all used log group.
- Feature: Replaced log group “reset” with “default” group.
- Feature: SAMC: set default minimum stack size to 2000 bytes.
- Bug: STM32: Made I2C slave driver capable of receiving pure master-writes.

1.5.2 (2018-12-19)

- Improvement: Updated documentation.

1.5.1 (2018-11-21)

- Bug: Made gs_time_uptime() thread safe (added call in gs_thread_create() for initialization of mutex).
- Breaking: Removed gs_time_rel_s() and gs_time_rel_s_isr().
- Improvement: STM32: Made support for 20 SPI slaves instead of 8.

1.4.1 (2018-09-20)

- Feature: Add gs_thread_create_with_stack to manually place stack buffer.
- Feature: avr32: Add support for using GPIO as SPI chipselect.

1.3.1 (2018-08-13)

- Feature: avr32: added support for gs_getcwd().
- Feature: avr32: added avr32_reset_all_interrupt_settings to use when booting from RAM.
- Improvement: avr32: Optimized gs_spi_master_transactions().

1.2.1 (2018-05-30)

- Breaking: Updated reset cause and system reset API.
- Feature: Added support for microsecond delays on STM32
- Improvement: Use standard log groups for I2C, SPI and CAN.
- Feature: split “free” (memory) comamnd into intern/extern for improved error handling.
- Breaking: SAMC21 - refactored APIs to comply with standard APIs.

1.1.2 (2018-04-17)

- Feature: Support for different SPI clock polarity/phase.
- Feature: Adde gs_freertos_convert_ms_to_ticks().
- Breaking: Changed I2C API to comply with standard.
- Feature: Added VMEM driver for MCU flash.
- Feature: SAMC21 changed UART driver from polled to interrupt.

1.0.1 (2018-03-09)

AVRx emulation (libemul)

1.3.2 (2019-10-16)

- Improvement: Changed some functions to weak.

1.3.1 (2019-04-09)

- Improvement: Updated dependencies.

1.2.3 (2018-12-19)

- Improvement: Updated documentation.

1.2.2 (2018-11-22)

- Breaking: ‘reset’ - relies on libutil::console to register console cleanup.

1.2.1 (2018-08-13)

- Breaking: Changed module to a ‘regular’ library, but passive (no code compiled unless enabled).

Flight Planner (libfp)

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

3.6.3 (2019-10-16)

- Improvement: Updated client.

3.6.2 (2019-05-19)

- Improvement: Include source code in distribution.

3.6.1 (2019-04-09)

- Improvement: Updated dependencies.

3.5.1 (2019-03-26)

- Bug: Fixed bug in listing FP entries.
- Improvement: Improved load and processing of flight plans.
- Breaking: Removed “internal” headers from public scope.
- Improvement: Updated documentation.

3.4.1 (2019-02-26)

- Improvement: Improved error handling.

3.3.3 (2019-01-29)

- Bug: Removed duplicated header file gs/fp/internal/types.h.
- Improvement: Register log groups.

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.

3.2.2 (2018-08-13)

- Feature: Added command registration API.

3.2.1 (2018-05-30)

- Breaking: Moved “private” headers internally. Changed return codes to gs_error_t (from negative -1 / 0).

3.1.1 (2018-04-17)

2.0.0 (2017-05-04)

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.
- 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)

5.2.4 (2020-05-18)

- Bug: Fixed issue resuming a file upload.

5.2.3 (2020-04-17)

- Bug: issue with temporary file upload if ftp done is lost - for now not using temp file.

5.2.2 (2020-03-20)

- Bug: fixed issue with resume of a file transfer, causing the overall transfer to fail. Occurred if resume was done too quickly.
- Feature: ftp transfer is now done to a temporary file, and renamed on successful transfer.

- Improvement: Reduced ftp server timeout (CSP read) from 180 to 60 seconds.

5.2.1 (2019-10-16)

- Improvement: Updated client and dependencies.

5.1.3 (2019-07-03)

- Bug: mkdir - always create directory on server with permission 0777 (fixes endianess problem between client and server, which causes invalid/none permission on created directory).

5.1.2 (2019-05-19)

- Improvement: Include source code in distribution.

5.1.1 (2019-04-09)

- Improvement: Updated dependencies.

5.0.5 (2019-02-26)

- Feature: Added gs_ftp_csp_service_handler_set_stack_size().

5.0.4 (2019-01-29)

- Improvement: Register all used log group.

5.0.3 (2018-12-19)

- Bug: Updating client
- Improvement: Updated documentation.

5.0.2 (2018-11-29)

- Feature: Updated client

5.0.1 (2018-11-21)

- Breaking: Updated client

4.3.3 (2018-09-21)

- Improvement: Updated client and dependencies.

4.3.1 (2018-08-13)

- Improvement: Updated client and dependencies.

4.2.1 (2018-05-30)

- Improvement: Updated client and dependencies.

4.1.1 (2018-04-18)

- Improvement: Removed dependency to libgosh

4.0.1 (2018-03-06)

3.0.0 (2017-07-02)

2.0 (2015-05-07)

- Feature: 32-bit size field
- Feature: Added ftp zip/unzip
- Improvement: Do not sync for every chunk on linux for faster transfers

1.0 (2015-03-09)

- Feature: FTP client work on nanomind a3200
- Feature: FTP server works on linux too
- Feature: Updated newlib backend

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)

- 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)

3.5.4 (2019-12-03)

- Bug: Fixed race-condition in gs_gosh_redirect_stdio(), causing random failure in redirecting stdio correctly.
- Improvement: Updated dependencies.

3.5.3 (2019-10-16)

- Improvement: Updated documentation.

3.5.2 (2019-05-16)

- Improvement: Included source code in distribution.

3.5.1 (2019-04-09)

- Bug: Fixed bug in remote server, failed on 32 bit platform

3.4.3 (2018-01-29)

- Improvement: Added support for stopping stdio redirect thread.
- Bug: Fixed file handle leaks in stdio redirect.
- Feature: Added initialization functons: gs_gosh_remote_init(), gs_gscript_init().

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).

3.3.1 (2018-08-13)

- Feature: added support for CSP service handler interface.

3.2.1 (2018-05-30)

- Breaking: Moved internal struct/defines out of public headers (used between host and client side).

3.1.1 (2018-04-17)

- Breaking: deprecated headers to libutil.

3.0.1 (2018-03-14)

- Breaking: Moved command and console APIs to libutil.
- Improvement: Improved g-script loading, avoid blocking calling thread and cause watchdog reset.

1.0 (2015-05-08)

- Feature: Changed to generic console interface (stdin/stdout)
- Feature: Remove need for command_setup calls on all platforms (apart from matlab)
- Feature: Remove need for USART_CONSOLE
- Feature: Support runtime setup of console
- Feature: Added commands for vmem
- Improvement: CMP clock printout
- Feature: Clock debug command
- Feature: Simplify toolchain configuration
- Feature: Clear screen on <CTRL+L>
- Feature: Automatic check for time.h
- Feature: Network redirection (telnet like)
- Feature: Moved code from libgomspace into libgosh
- Breaking: Deleted old csp_if_sia header
- Feature: Added console_messages_waiting

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)

4.4.3 (2020-03-20)

- Improvement: Updated client.

4.4.2 (2019-07-03)

- Improvement: Updated client.

4.4.1 (2019-05-19)

- Improvement: Include source code in distribution.

4.3.1 (2019-04-10)

- Bug: Fixed csp service ant6 status handling.

4.2.6 (2019-02-26)

- Improvement: Updated client.

4.2.5 (2019-01-28)

- Improvement: Updated client.

4.2.4 (2018-12-19)

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

4.2.3 (2018-11-21)

- Improvement: Removed locking from sunsensor sample/get functions (not required).
- Feature: Added extern cplusplus to headers (support C++ usage).

4.2.2 (2018-08-13)

- Feature: Added command register API.

4.2.1 (2018-05-30)

4.1.1 (2018-04-17)

- Improvement: Removed libgosh dependency.

4.0.1 (2018-03-06)

3.0.0 (2017-06-22)

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.

4.2.1 (2018-05-30)

4.0.1 (2018-03-01)

Housekeeping System (libhk)

4.5.3 (2020-05-11)

- Bug: Fixed possible memory leak if hk persist failed opening file (LIBHK-120).
- Improvement: Updated dependencies.

4.5.2 (2019-10-16)

- Bug: Fixed that the feature for temporarily disabling auto-beacons did not countdown as expected (LIBHK-119).

4.5.1 (2019-07-01)

- Feature: When defining a beacon it is now possible to select a single index in an array parameter (LIBHK-117).
- Feature: Added function in API to set priority on all beacons
- Bug: Fixing issue where it was not possible to generate beacons after lowering priority (LIBHK-114).
- Bug: Fixing issue where collector threads where temporarily locked out during configration loading (LIBHK-116).
- Improvement: Improved logging when generating beacons.

4.4.1 (2019-05-20)

- Improvement: Added note in documentation about having mulitple beacons using the same table.
- Bug: Fixed an issue that could cause a crash if the ‘hk_srv’ command was called before initialization was done.

- Improvement: Include source code in distribution.

4.3.1 (2019-04-10)

- Improvement: Updated dependencies.

4.2.3 (2019-04-10)

- Bug: Fixed issue that could cause unexpected behaviour when collected tables where removed.
- Bug: The TTL used to signal when to clean up tables not being collected was not reset correctly.
- Feature: Default table download timeout has increased from 100 milliseconds to 200 milliseconds.
- Feature: It is now possible to tweak the table download timeout pr. table being collected.

4.2.2 (2019-04-08)

- Bug: Fixed issue that could cause a crash if collected tables where removed.

4.2.1 (2019-02-26)

- Feature: Added host commands (moved from libhk_client).

4.1.4 (2019-01-29)

- Improvement: Register all used log groups.

4.1.3 (2018-12-20)

- Improvement: Updated documentation.

4.1.2 (2018-12-03)

- Improvement: Fixed commands documentation.
- Bug: Requesting a beacon with tables not being correctly collected could lead to a crash.
- Bug: Running a 'hk_srv table show' command could cause a crash.
- Bug: Timestamp in requested beacons would be set to the requested time.

4.1.1 (2018-11-22)

- Feature: Added flag to completely enable/disable the housekeeping system.
- Bug: Fixed a memory leak that would occur if a remote table became unavailable.
- Breaking: Renamed root Command to better match overall scheme (hk_client -> hk, hk -> hk_srv).
- Feature: Added support for configuration being reloaded from client via CSP.
- Bug: Fixed issue where server could crash if .json was malformed.

4.0.1 (2017-10-17)

- Breaking: major rework of configuration of the housekeeping system.

3.3.1 (2018-08-13)

- Feature: Updated dependencies to latest version.

3.2.1 (2018-05-30)

- Improvement: To limit risk of running out of memory the collector store_intv is capped at 14400s/4 hours. If store_intv is 0, tables are truncated every 600s, but persisting is omitted.
- Bug: Fixed an issue where the node would crash when trying to persist housekeeping data.

3.1.2 (2018-04-18)

- Feature: Added param 4 store configuration.

3.0.2 (2018-03-11)

2.0.0 (2018-06-22)

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)

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

JSON parser (libjson)

1.1.1 (2019-04-08)

- Improvement: Updated dependencies.

1.0.2 (2019-02-26)

- Improvement: Updated to cJSON version 1.7.10

1.0.1 (2018-12-19)

- Improvement: Updated documenation.

1.0.0 (2018-11-22)

- Feature: First release.

0.0.1 (2018-08-20)

- Feature: cJSON moved in.
- Feature: Type-checking json functions added.

Log stores (liblog)

3.6.1 (2019-10-16)

- Improvement: Updated tests.

3.5.2 (2019-05-16)

- Improvement: Include source code in distribution.

3.5.1 (2019-04-09)

- Improvement: Optimized memory logger (reduced number of reads/writes).
- Improvement: Added locking to prevent corruption.
- Feature: Added support for ‘clear’.

3.4.2 (2018-12-19)

- Improvement: Updated documentation.

3.4.1 (2018-11-21)

- Breaking: The Remote log service has been removed. Replaced by libutil Log appender + Rgosh
- Breaking: Re-factored the vmem log store to plug into the new Log Appender Framework.
- Feature: New memory Log appender which can be live in multiple instances.

3.3.2 (2018-09-20)

- Improvement: Prepared commands for remote-gosh.

3.3.1 (2018-08-13)

- Feature: Build shared library.

3.2.1 (2018-05-30)

- Breaking: Moved log API (front-end) to libutil, moved csp log hook to libgscsp.

3.1.1 (2018-04-17)

- Improvement: Removed dependency to libgosh

3.0.1 (2018-03-06)

1.0 (2015-05-08)

- Feature: Log output to console/stdout
- Feature: Store logs in FRAM (using VMEM interface).
- Feature: Adjustable log levels (ala gstreamer)
- Feature: Support for libcsp debug hook
- Feature: Very low memory footprint (1 ptr + 2 bytes per log domain)
- Feature: Replaces driver debug, csp debug and the former cdh log system

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)

4.8.1 (2020-03-19)

- Improvement: Updated client and tests.

4.7.3 (2019-12-03)

- Bug: filestore auto-persist, if auto persist file already existed, later 'update' would fail because file was opened in Read mode.
- Improvement: Updated client and dependencies.

4.7.2 (2019-10-16)

- Improvement: Improved check on VMEM store size.
- Improvement: Added validation of clear of VMEM stores.

4.7.1 (2019-05-16)

- Improvement: Include source code in distribution.

4.6.1 (2019-04-09)

- Breaking: Updated telemetry appender to new log appender interface.

4.5.4 (2019-02-26)

- Improvement: Updated client.

4.5.3 (2019-01-29)

- Improvement: Register all used log group.

4.5.2 (2018-12-19)

- Feature: Added Log Telemetry appender: Provides telemetry of Log errors/warnings in Param.
- Feature: added support for new rparam load/save from/to named stores.
- Improvement: Updated documentation.

4.5.1 (2018-11-21)

- Breaking: removed ‘rparam clear’ functionality.
- Bug: Table 0 was not always read correctly through the Python API.

4.4.1 (2018-09-20)

- Feature: Allow for tables larger than 1K in vmem stores.
- Bug: prevent stripping leading spaces, when setting string parameters.
- Feature: Added support for anonymous table, see gs_param_table_init().

4.3.1 (2018-08-13)

- Feature: table create: added support for default data values as a data struct.

4.2.2 (2018-05-30)

- Breaking: Renamed Parameter IO API to Parameter Protocol (PP).
- Bug: fixed missing auto-persist of parameter changed from GOSH.
- Breaking: removed support for “all” in “param save <table> all”.

4.1.1 (2018-04-17)

- Breaking: All new table stores, store values in network order.
- Feature: Added param commands: tableinfo, storeinfo.
- Feature: Added support for protected/locked tables in FRAM.
- Bug: Call callback function, when loading tables.
- Feature: Added binary file store.
- Feature: Re-introduced some of the old structures, in order to support “old” clients.
- Bug: Fixed auto-persist paramtres (value was not persisted).

4.0.1 (2018-03-12)

- Breaking: Number of APIs have been broken, especially regarding creating tables and setting up stores.
- Feature: Configure multiple stores per table, e.g. FRAM, FRAM (locked), File, etc.
- Feature: Variable store size, i.e. improving usage of storage media.
- Improvement: Genereal error handling and logging.

2.0 (2015-05-08)

- Breaking: Allow full table download to be split into multiple frames

- Bug: Init persist arrays.
- Feature: Added param_save and param_load to param_file backend
- Breaking: Serializer API, split API and introduced flags.
- Feature: Make deserializer support packet data format
- Bug: Add safety check on infinite serializer bug
- Feature: Added 'param tables' command

1.0 (2015-03-16)

- Feature: Supports RAM, FRAM, File system.
- Feature: Local host 'param' commands
- Feature: CSP Client/Server 'rparam' commands
- Feature: Arrays of parameters
- Feature: Table locking
- Feature: Serializer / Deserializer

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)

- 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)

1.2.3 (2020-03-24)

- Bug: Free resources in case of failure, i.e. connections, memory.
- Improvement: Increased timeout on CSP receive, to compensate for retransmissions on RDP layer.

1.2.2 (2020-03-18)

- Improvement: Updated client.

1.2.1 (2020-01-23)

- 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: Include source in distribution.

1.1.1 (2019-04-10)

- Improvement: Updated dependencies.

1.0.3 (2019-01-28)

- Improvement: Register all used log groups.
- Improvement: Updated API doc.

1.0.2 (2018-12-21)

- Bug: z7000 file seek was incompatible with rgosh

1.0.1 (2018-12-19)

- Improvement: Updated documentation.

1.0.0 (2018-11-21)

- Breaking: Updated client and client protocol messages
- Improvement: Send multiple RGOSH responses when long messages are transfered.
- Improvement: Logging cleaned-up
- Bug: Handle NULL values for Group/Key/Value pairs.
- Improvement: Balance workload better between RGOSH worker threads.
- Bug: Support RDP connections - reduce the MTU

0.1.2 (2018-08-07)

- Feature: Support for running GOSH commands remotely.

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.

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.

Embedded File Storage (libstorage)

3.7.1 (2020-03-18)

- Improvement: Added recursive remove to VFS API and as option to GOSH command.
- Bug: Fixed undefined behavior when file paths exceeded 50 characters.

3.6.1 (2019-10-16)

- Improvement: Updated dependencies.
- Breaking: Re-factored code for mounting, making vfs_mount() the actual mounting function.
- Improvement: Use endian definition from libutil.
- Improvement: Added VFS required define for newlib ('HAVE_OPENDIR=1'), moved from a3200-sdk.
- Breaking: Changed filesystem syscalls to implement re-entrant versions - requires libembed 1.8.

3.5.2 (2019-07-03)

- Improvement: Changed command/gosh definitions to const.

3.5.1 (2019-04-08)

- Improvement: Updated dependencies.

3.4.4 (2019-01-29)

- Improvement: Register all used log groups.

3.4.3 (2018-12-19)

- Improvement: Updated documentation.

3.4.2 (2018-11-21)

- Improvement: Improved output from commands (gosh).

3.4.1 (2018-10-08)

- Breaking: gs_fl512s_uffs_mount() now supports mounting two flash dies.

3.3.2 (2018-09-20)

- Improvement: Print output to stream, instead of stdout.

3.3.1 (2018-08-13)

- Breaking: Removed support for FAT (SD), and removed unused drivers: FAT (SD), UFFS.
- Breaking: Re-structured include and source, divided into 3rd part and GomSpace code.
- Breaking: Re-structured by moving mounting procedure to this lib from liba3200.
- Breaking: Re-structured - including driver for fl512s from libthirdparty instead of liba3200.
- Feature: Added support for mounting both dies of the fl512s.

3.2.1 (2018-05-30)

- Breaking: Updated internal use of reset API from libembed, which now takes a reset cause.
- Breaking: updated to new spn_fl512s API (changed return codes to gs_error_t).

3.1.1 (2018-04-17)

- Improvement: Removed libgosh and libgscsp dependencies.

3.0.1 (2018-03-06)

2.0.0 (2018-05-04)

Thirdparty (libthirdparty)

2.10.2 (2020-03-26)

- Improvement: Updated driver for ST200 Star Tracker

2.10.1 (2020-03-18)

- Improvement: Added M-G3xx driver (Epson Gyro)
- Improvement: Add vmem info to cy15b102q fram driver.

2.9.1 (2019-10-16)

- Bug: tps382x watchdog service interval outside spec
- Improvement: spn_fl512 API: Error handling in both API and commands. Commands now supports both dies.
- Breaking: spn_fl512s_read_status_register() return error code and status register by reference.

2.8.4 (2019-07-19)

- Improvement: Cleanup of HTPA driver and optimised wait time between sensor readouts

2.8.3 (2019-05-29)

- Improvement: Added gps log groupe - the log groupe is now resigterd from the gps task

2.8.2 (2019-05-16)

- Improvement: Added the capability to initialise the rm3100 with different I2C addresses than the default.
- Improvement: Added source code to distribution.

2.8.1 (2019-04-08)

- Improvement: Added 'info' API to FRAM interfaces, for getting detailed information.
- Improvement: Updated dependencies.

2.7.1 (2019-02-26)

- Breaking: Renamed LM70 API to LM71.
- Bug: LM71 (former LM70) failed to read negative temperatures - bug introduced in 2.4.1 (when driver was added).
- Feature: Added support for calibration and current/power measurements for INA226

2.6.1 (2019-01-30)

- Feature: Added driver for ISIS antenna system
- Improvement: FRAM read/write: added check on address and length. Added read back check in commands for erase and write.
- Feature: Added VARF command for Novatel 615
- Improvement: Register all used log groups.

2.5.2 (2018-12-19)

- Improvement: Updated documentation.

2.5.1 (2018-11-21)

- Feature: Added driver for Hyperion ST200 star tracker
- Breaking: Rework of AF Wheel driver with better error handling

2.4.1 (2018-09-20)

- Feature: Added driver for LM70 temperature sensor.

- Feature: Added driver for INA226-Q1 supply monitor.

2.3.1 (2018-08-13)

- Feature: Added driver for FL512S (moved from liba3200).

2.2.2 (2018-05-30)

- Bug: Fixed read command in Novatel 615.
- Improvement: unit-test - updated to use I2C plugin driver from libutil.
- Feature: Added driver for TSP382x watchdog.

2.1.2 (2018-04-17)

- Feature: TI TMP100 temperature sensor driver

2.0.1 (2018-03-06)

- Feature: CY15B102Q driver (including VMEM).
- Feature: HTPA32 driver

1.0.0 (2017-06-06)

- Feature: Novatel_615 driver

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.
- 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.

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

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
- 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

6. Bibliography

Bibliography

[1] GomSpace. NanoMind A3200 Board Support Package. gs-man-nanomind-a3200-board-support-package-2.4.1.pdf. IFS: 1019598.

7. 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.