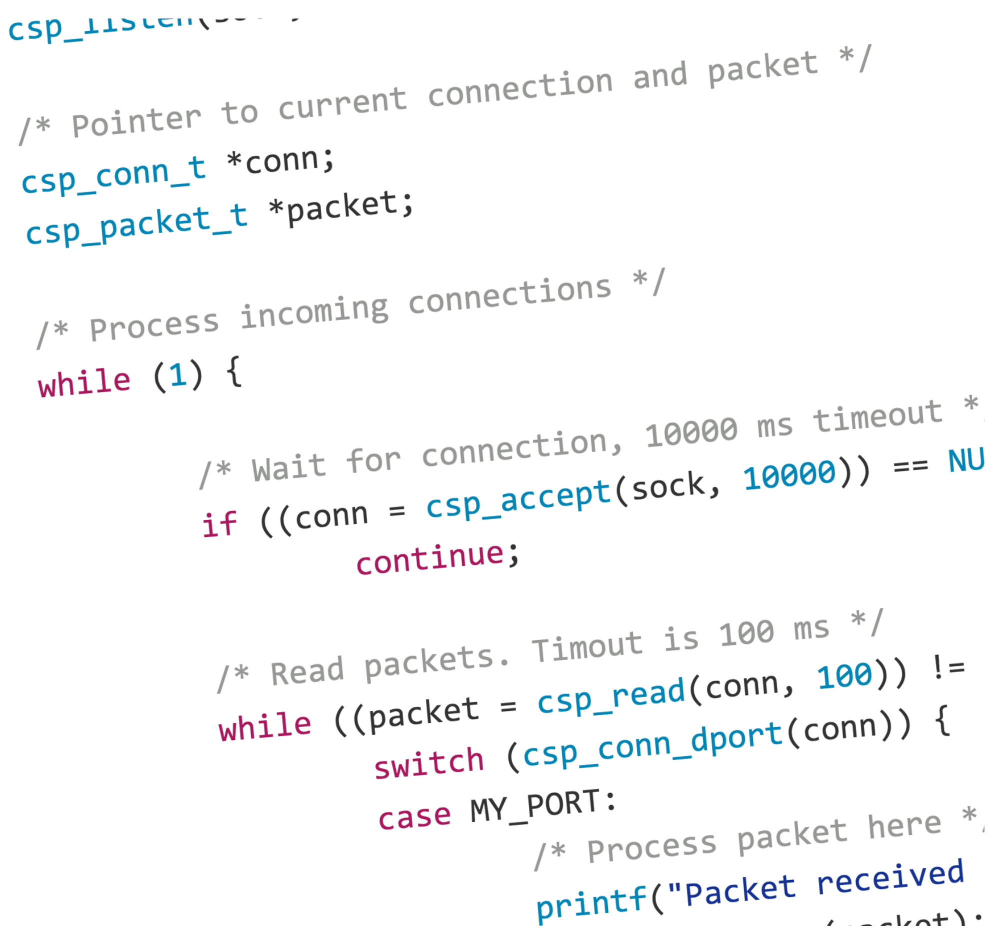

```
csp_listen(
/* Pointer to current connection and packet */
csp_conn_t *conn;
csp_packet_t *packet;

/* Process incoming connections */
while (1) {

        /* Wait for connection, 10000 ms timeout *
        if ((conn = csp_accept(sock, 10000)) == NU
                continue;

        /* Read packets. Timout is 100 ms */
        while ((packet = csp_read(conn, 100)) !=
                switch (csp_conn_dport(conn)) {
                case MY_PORT:
                        /* Process packet here *
                        printf("Packet received
```

GomSpace SDK for Linux

Manual

Software Documentation

Release v2.8-23-gd19fdd6

GomSpace ApS • Alfred Nobels Vej 21C • DK-9220 Aalborg East • Denmark
www.gomspace.com • info@gomspace.com • +45 96356111

Table of Contents

1. Introduction

This manual describes the SDK for Linux software package.

The SDK for Linux includes a set of advanced and feature rich modules that enable fast and efficient development of Linux applications for either Ground Station or for Linux based CubeSat computers. It can also be used as an example of how to develop GomSpace compatible applications for Linux.

The basis of the SDK for Linux is the individual modules and their functionalities. The following modules are included:

- **libcsp** Network library
- **libftp** File transfer
- **libgosh** Shell interface
- **libparam** Parameter system
- **liblog** Logging systems
- **libutil** Various utilities

Each library has it's own chapter later in this document

Installation and build instructions are described in the next sections.

1.1 SDK for Linux Installation and Build

1.1.1 Source Code Installation

Install Source Code:

1. Create a workspace folder for the source code, e.g. ~/workspace: *mkdir ~/workspace*
2. Change working directory to the workspace folder: *cd ~/workspace*
3. Copy source code tar ball (*linux-sdk-<version>.tar.bz2*) to ~/workspace
4. Unpack source code: *tar xfv linux-sdk-<version>.tar.bz2*

You now have the source code installed in *~/workspace/linux-sdk-<version>/*

1.1.2 Building the Source Code

Prerequisites: Recent Linux with python (example build with ubuntu 14.10)

Install requirements:

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

The executable build script is called *waf* and is placed in the root-folder of the software repository. Waf uses a recursive scheme to read the local *wscript* file(s), and submerge into the *lib/lib. . . ' folders and find their respective 'wscript* files. This enables an elaborate configuration of all modules included in the SDK for Linux.

To configure and build the source code:

1. Change directory to source code folder: *cd ~/workspace/linux-sdk-<version>*

2. Configure source code: *waf configure*. This will configure the source code and check that the tool chain (compiler, linker, ...) is available
3. Build source code: *waf build* (or just *waf*).

The *waf build* command will compile the source code, link object files and generate an executable in the build folder with the name *csp-term*.

1.1.3 Client Modules

To include all client modules from the clients sub-folder:

```
$ waf configure --with-clients=all
```

To include selected client modules (e.g. nanopower and nanocom-ax) from the clients sub-folder:

```
$ waf configure --with-clients=nanopower,nanocom-ax
```

Remember to re-build source after a *waf configure*:

```
$ waf build
```

Command line arguments

The SDK for Linux main application csp-term has the following command line arguments:

```
-a Address
-c CAN device
-d USART device
-b USART buad
-z ZMQHUB server
```

Example 1: Starting csp-term with address 10 and connecting to a ZMQ proxy on localhost:

```
$ ./build/csp-term -a 10 -z localhost
```

Example 2: Starting csp-term with address 10 using usart to /dev/ttyUSB0 at baudrate 500000:

```
$ ./build/csp-term -a 10 -d /dev/ttyUSB0 -b 500000
```

Example 3: Starting csp-term with address 10 using CAN interface can0:

```
$ ./build/csp-term -a 10 -c can0
```

2. CubeSat Space Protocol

2.1 The Cubesat Space Protocol

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

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

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

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

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

2.1.1 Features

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

- Promiscuous mode
- Encrypted packets with XTEA in CTR mode
- Truncated HMAC-SHA1 Authentication (RFC 2104)

2.1.2 LGPL Software license

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

2.2 History

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

2.2.1 Satellites using CSP

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

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

2.3 Structure

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

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

2.4 CSP Interfaces

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

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

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

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

2.4.1 Outgoing traffic

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

```
int csp_fifo_tx(csp_packet_t *packet, uint32_t timeout) {
    write(tx_channel, &packet->length, packet->length + sizeof(uint32_t) + sizeof(uint16_t));
    csp_buffer_free(packet);
    return 1;
}
```

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

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

2.4.2 Incoming traffic

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

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

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

```
void csp_new_packet(csp_packet_t *packet, csp_iface_t *interface, CSP_BASE_TYPE *pxTaskWoken);
```

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

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

2.4.3 Initialization

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

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

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

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

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

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

2.4.4 Building the example

The fifo examples can be compiled with:

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

The two named pipes are created with:

```
% mkfifo server_to_client client_to_server
```

2.5 How CSP uses memory

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

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

2.5.1 Tables

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

2.5.2 Buffer Pool

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

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

2.5.3 Interface list

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

2.6 The Protocol Stack

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

2.6.1 Layer 1: Drivers

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

2.6.2 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 supoorts specifying next-hop link-layer address directly. This avoids the need to implement an address resolution protocol to translate CSP addresses to I2C addresses.

2.6.3 Layer 3: Network Router

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

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

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

2.6.4 Layer 4: Transport Layer

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

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

UDP

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

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

While UDP is very simple, it also has some limitations. Normally a human in the loop is a good thing when operating the satellite over UDP. But when it comes to larger file transfers, the human becomes the bottleneck. When a high-speed file transfer is initiated data acknowledgement 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 Acknowdlegment

For more information on this, please refer to RFC908.

2.7 Network Topology

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

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

        Node 9          Node 10
```

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

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

2.8 Maximum Transfer Unit

There are two things limiting the MTU of CSP.

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

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

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

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

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

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

But, what if you just want 1000 bytes transmitted, and you don't care about the fragmentation unit, and also don't want the hassle of writing the fragmentation code yourself? - In this case, libcsp now featuers a new (still experimental) feature called SFP (small fragmentation protocol) designed to work on the application layer. For this purpose you will not use csp_send and csp_recv, but csp_sfp_send and csp_sfp_recv. This will split your data into chunks of a certain size, enummerate them and transfer over a given connetion. 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.

2.9 Client and server example

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

2.9.1 Initialization Sequence

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

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

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

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

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

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

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

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

2.9.2 Server

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

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

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

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

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

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

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

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

2.9.3 Client

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

```
int send_packet(void) {

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

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

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

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

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

    /* Close connection */
    csp_close(conn);

    return 0
}
```

3. GomSpace FTP Library

3.1 Introduction

The GomSpace FTP Library implements a generic File Transfer Protocol (FTP) running over CSP. FTP allows remote transfers and manipulation of files to and from the 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 e.g. used by NanoMind OBCs to uplaod new software images to the flash filesystem or directly to RAM for rapid testing of code.

The protocol implementation consists of a server and a client. The NanoMind runs a server that provides access to its local filesystems, but it can also be act as a client to other subsystems, e.g. to download pictures from a NanoCam. The figure below shows major components and call chains.

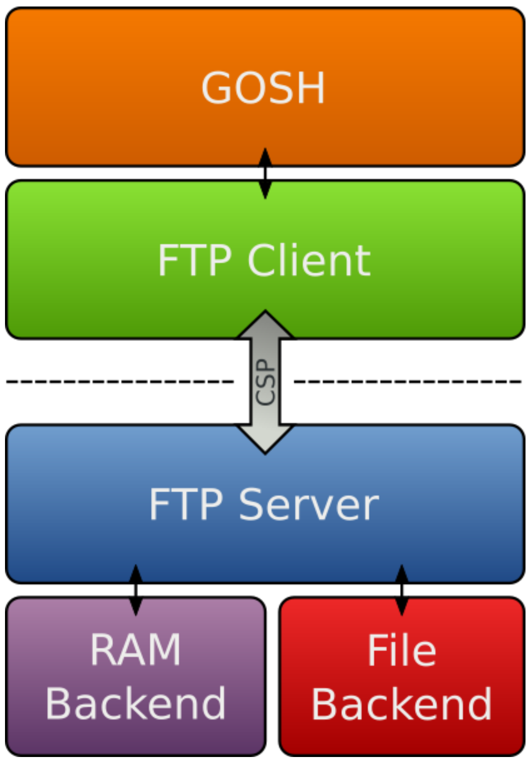

Fig. 3.1: File Transfer Protocol components.

The server component interfaces to two *backends*, one for accessing the filesystems and one for direct access to memory. A client library (see `libftp/src/ftp/ftp_client.c`) implements the client side of the FTP transfer. The `ftp` GOSH commands is implemented on top of this client interface.

3.2 Structure

The GomSpace FTP Library is structured as follows:

Folder	Description
libftp/include	GomSpace FTP Library API header files.
libftp/src/ftp	This folder contains the FTP implementation.
libftp/doc	The doc folder contains the source code for this documentation.

3.3 GOSH Commands

The GomSpace Shell includes a number of commands for using the FTP, all placed under the global `ftp` command. These commands are available in both CSP-term and on NanoMind systems:

Command	Description
ls	List files
rm	Remove files
mkfs	Make file system
mv	Move files
cp	Copy files
zip	Compress file
unzip	Decompress file
server	Set host and port
backend	Set filesystem backend
upload_file	Upload file
download_file	Download file
upload_mem	Upload to memory
download_mem	Download from memory
timeout	General timeout

To interface with a remote FTP server, you first need to specify its CSP address and port using the `ftp server <address> [port]` command. In the default configuration, NanoMind A3200 systems use address 1 and port 9 for FTP, and NanoMind A712D systems use address 1 and port 7. The remaining FTP commands will then use these variables when connecting to the server.

An example of transfer setup, listing, download and removal of the `/flash/test` file from a NanoMind A3200 is shown below:

```
csp-term # ftp server 1 9
csp-term # ftp ls /flash/
 97.7K test
csp-term # ftp download_file /flash/test
1447321375.671 ftp: Download begin: path /flash/test, type 3, addr 0x0, size 0, chunk size 185
File size is 100000
Checksum is 0xaacf4fc9
Transfer Status: 0 of 541 (0.00%)
1447321377.542 ftp: Transfer Status: 0 of 541 (0.00%)
100.0% [#########################################] (541/541) 25.2 kB/s
CRC Remote: 0xaacf4fc9, Local: 0xaacf4fc9
csp-term # ftp rm /flash/test
csp-term #
```

Refer to the GomSpace Storage Library documentation for a list of valid filesystem paths.

4. GOSH - GomSpace Shell Library

4.1 Introduction

One of the first things encountered when booting the NanoMind computer is the gomspace shell (GOSH) found on the serial port. This provides a simple but extensive debug interface to the NanoMind computer.

At the time of writing GOSH is running on most GomSpace products, including NanoMind A712, NanoMind A3200, NanoCom, NanoCam, NanoHub, and CSP-Term. GOSH includes a comprehensive console task and an extendable command parser with autocompletion and usage information.

4.2 Structure

The GOSH is structured as follows:

Table 4.1: GOSH structure

Folder	Description
libgssb/include	The inlcude folder contains the GOSH API header files
libgssb/src	The src folder contains the GOSH API source code files
libgssb/doc	The doc folder contains the source code for this documentation

4.3 Console

The console provides a text-interface to a given input/output stream such as a serial port. This provides a mean of typing a command in human readable text format, and passing this command to the command parser. The console uses a traditional keyboard shortcut layout for navigating history, line editing and includes command name tab completion and command usage information. Any user who have tried a text-interface before should feel right at home. The console task is written for both Linux and FreeRTOS and uses the getchar and usart_getc functions respectively, to get input from the user's console. The console assumes a standard VT102 terminal emulator, but features a few fixes for the quirks that minicom has. The console will greet the user with a hostname and promt. The hostname is set in the console_set_hostname function and the prompt is hardcoded in console.h as "033[1;30m # 033[0m". This special syntax gives a colored promt for the user. Here is an example promt:

The console will react to user input and commit to the command_run whenver is pushed and commit to command_complete when is pushed. This gives the same behaviour as for example a unix shell where entire commands are written as strings and comitted when enter is pushed. A list of key inputs are understood such as , to change the cursor position, and and to go through the command history. The complete list of command shortcuts are:

4.4 Command Parser

The command parser splits the typed command into tokens based on the space delimter. It searches for the first word in the list of root-commands. The list of root-commands can be displayed by the help command or pressing on an empty propmt. Here is a typical list of root-commands for CSP-Term running on Linux:

The list includes CSP network utilities, general utilities and subsystem commands. Some of the root- commands have sub-commands. A sub command is the next following word on the command line, so for example in:

The word ident is the sub command to the cmp root command. It's possible to make an arbitrary number of sub-commands. In order to get the list of sub commands for a given root command, type the sub command and or . For the cmp the list of sub commands are for example:

Each command that does not have a sub-command, wether its a root command or not, has an associated handler. This is a pointer to the function that will be called with the arguments. It also has a help text and a usage text. The help text is shown in the help output as shown above, and the usage text is shown when is pressed or when an invalid number of arguments are given to a command:

4.5 Defining new commands

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

```
command_t __root_command eps_commands[] = {
   {
      .name = "eps",
      .help = "EPS subsystem",
      .chain = INIT_CHAIN(eps_subcommands),
   }
};
```

Remember to include *<command/command.h>* to get the command_t declaration. In this example the root-command “eps” points to a sub-command using the INIT_CHAIN() macro, which does a compile-time count of the number of sub-commands and inserts in the root-command, so the command-parser knows the size of the array.

```
command_t __sub_command eps_subcommands[] = {
   {
      .name = "hk",
      .help = "Get HK",
      .handler = eps_hk,
   },{
      .name = "so",
      .help = "Set channel on/off, argument (channel) to (1 or 0)",
      .usage = "<channel> <mode> <delay>",
      .handler = eps_single_output,
   }
};
```

Here the example shows a two-subcommands from the examples above. It shows that calling eps hk will run the eps_hk() handler function. This command does not have any arguments, so there is no usage information associated. The so, command does have three arguments, so the usage info is specified. The array can be further extended with any number of sub commands.

4.6 Linker-optimizations

Instead of building the list of commands run-time using a linked list or similar datastructure, the root- commands are built by the linker using a special GCC-attribute to pack the command structs 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. It's therefore very important that all root-commands are tagged with the *__root_command* macro, and all sub-commands with the *__sub_command*. On X86 where these link time optimizations are impossible, the list of commands a built using a linked list and malloc.

4.7 Implementing a handler function

The handler function needs to follow the following prototype:

```
typedef int (*command_handler_t)(struct command_context * context);
```

An example of this is the eps_single_output handler:

```
int eps_single_output(struct command_context *ctx) {

    char * args = command_args(ctx);
    unsigned int channel;
    unsigned int mode;
    int delay;

    printf("Input channel, mode (0=off, 1=on), and delay\r\n");
    if (sscanf(args, "%u %u %d", &channel,&mode,&delay) != 3)
        return CMD_ERROR_SYNTAX;
    printf("Channel %d is set to %d with delay %d\r\n", channel,mode,delay);

    eps_set_single_output((uint8_t) channel, (uint8_t) mode, (int16_t) delay);

    return CMD_ERROR_NONE;
}
```

The functions can use the command_args() function to get the argument string and parse it using for example scanf. The alternative and preferred method is to use argc/argv syntax like this:

```
int my_cmd(struct command_context * ctx) {

    if (ctx->argc != 2)
        return CMD_ERROR_SYNTAX;

    uint8_t my_arg1 = atoi(ctx->argv[1]);

    return CMD_ERROR_NONE;
}
```

Remember to return with CMD_ERROR_NONE if the handler executed correctly, CMD_ERROR_SYNTAX if the arguments could not be parsed, or any of the other error codes found in *<command/command.h>*.

4.8 API

TBW

5. GomSpace Parameter Library

5.1 Introduction

GomSpace 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 datatypes: uint, int, float, double, string and hex
- Arrays of data
- Simple static parameter table and setup
- Multiple backends for non-volatile storage: FRAM and FILE
- Scratch memory and running memory configurations
- Remote client with full support for all commands
- Packed GET, SET queries with multiple parameters
- Simple data-set serialization and deserialization
- Supports both little and big-endian systems
- Gomspace Shell (GOSH) commands for both local and remote access
- Built-in CSP parameter server
- Service oriented Architecture (SoA) support with full download of parameter tables
- Autogeneration of client GUI
- Compile-time configuration of parameter system

5.1.1 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, object serialization, arrays, data locking and intuitive client interfaces. To best demonstrate this principle, let's take a simple example:

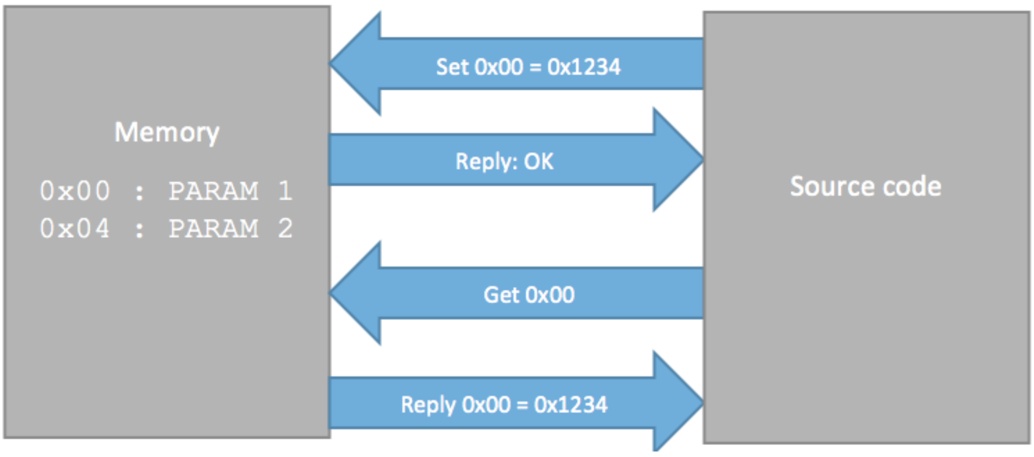

Fig. 5.1: Example of Set and Get commands

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:

5.1.2 Parameter tables

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

Table 5.1: 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.

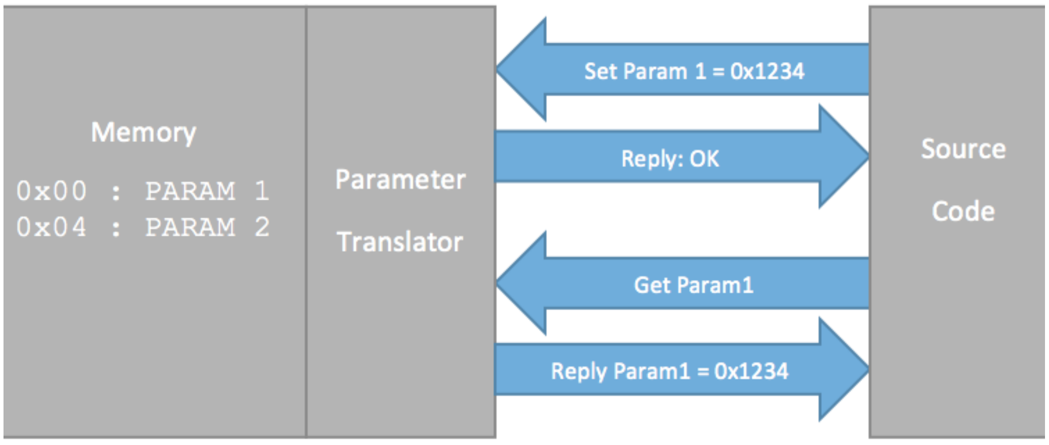

Fig. 5.2: Parameter Translator example

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.

This interface is known as the source API and is only available on a local system compiled with the parameter system library ‘libparam’. For more details on the source API, please consult the parameter system manual.

5.1.3 Network service

In order for the parameter system to work across multiple nodes, a client/server network API has been added also. 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.

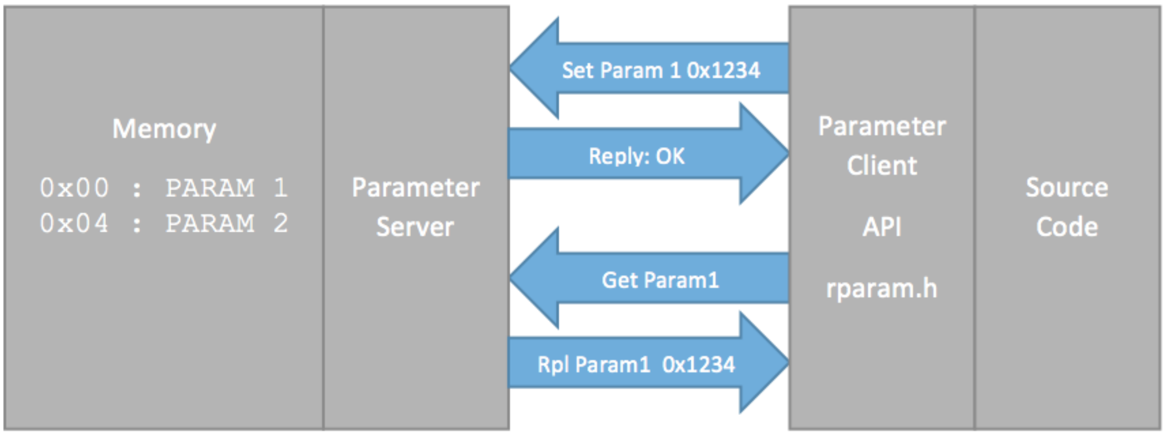

Fig. 5.3: Network Example

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

5.1.4 Parameter storage

The final thing that the parameter system needs is a method to store configurations into non-volatile memory. The parameter system is not limited to one or two types of memory, but for the most common setups there will be one volatile memory such as the RAM and another non-volatile memory such as FRAM or FLASH. This can be illustrated in the following way:

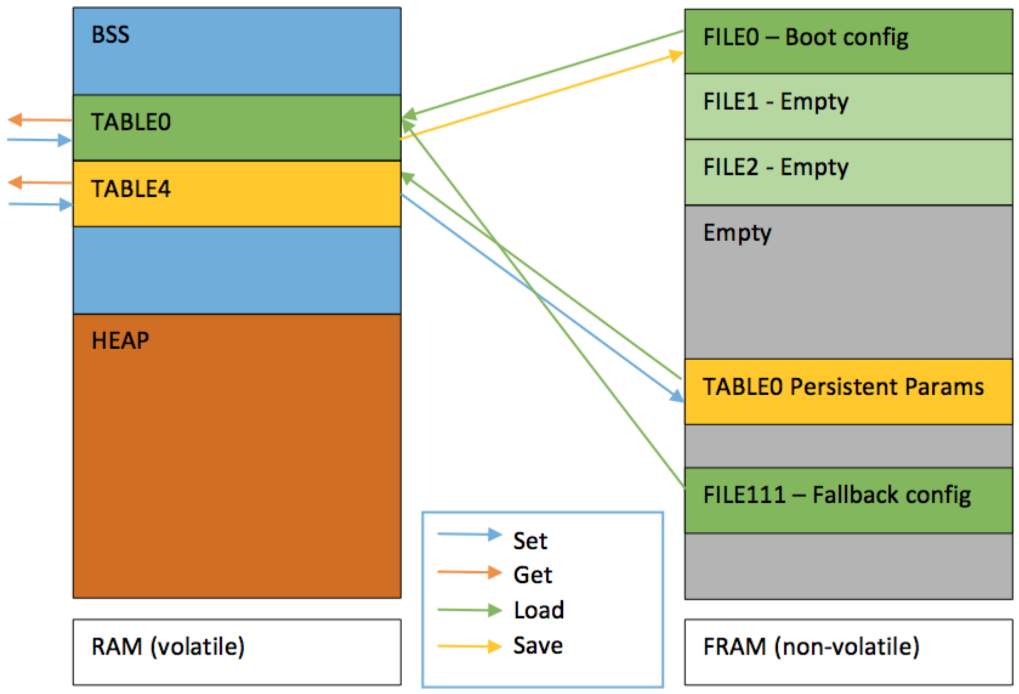

Fig. 5.4: Parameter Storage overview

The memory map for the volatile RAM memory is quite normal. Each set of parameters, each have their own reserved area in physical memory, and each logical parameter address then uniquely translates into a physical address by adding with an offset for the given parameter table. In this example the system have two tables, 0 which is used for configuration and 4 which is used for telemetry data. The storage system solves 2 issues for the parameter system:

Storing of multiple system configurations with fallback to factory configuration

The parameter system has a storage backend which allows storing copies of parameter tables to a file. In this case the FRAM has an extremely simple filesystem where every file is denoted with a number and that number directly translates into a memory address. In the example, file number zero is used as the default configuration that will be loaded into RAM during the system boot-up. This configuration can be loaded, saved or cleared at any time. If file number zero does not contain a valid configuration (verified by a checksum), then the system could continue to load from other file-id's in a certain order. In the example the system will try to load from file number 111 after failing at file number zero. This file number is special because the address resolves to an area of the FRAM chip which is write protected, so this area can be great to store a read-only fallback configuration.

Storage of persistent data (bootcounter, etc.)

The other thing that the FRAM chip is particularly good at, is storing rapidly changing persistent parameters. This could for example be a boot counter. If a parameter is marked with the PARAM_F_PERSIST in the parameter table and if a valid FRAM address has been initialized, 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 persistent 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 the value stored in FRAM. Due to the direct memory mapping there is no further checksum on each individual persistent parameter. This means that if the FRAM fails during startup, garbage could be loaded into the persistent parameters. Therefore these should not be used for critical configuration variables. The configuration files are however protected against corruption and a very good place to put critical variables. However updating the configuration and running a save on each variable change costs a bit more resources.

Using a custom file layout on the FRAM chip

If the standard layout of 32x1024B files does not match your task it is possible to setup a custom file layout. See *src/store/param_fram_mappers.h* for an example of a custom layout and see *tst/fram_store-UT/fram_store-UT.c* for how to use a custom layout in your task.

5.2 System overview

The parameter system is separated into three modules:

- **Parameter Host API**: This is the local interface used by the firmware on a subsystem
- **Parameter Service**: This module defines the network interfaces and implements a CSP service task to process incoming get/set requests.
- **Parameter Client API**: This module contains functions for sending get/set requests to a parameter service.

On top of these three modules lies a client layer where there is two shell-modules:

- **Local interface 'param'**: This is a set of commands that uses the Host API to modify parameters. This enables sending parameter commands directly to a system using the gomspace shell on the debug interface.
- **Remote client 'rparam'**: This is a set of commands that uses the Client API and the network 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:

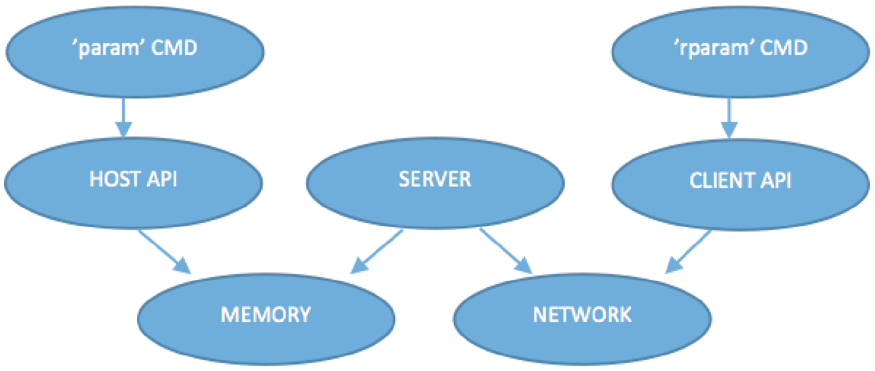

Fig. 5.5: Parameter system modules

5.3 GOSH: Local parameter interface

The local parameter interface can be used to directly modify the paramters of a system using the debug interface only. It's a quick and direct method of interfaceing to the parameter system which bypasses all network communication completely and goes directly to the memory of a running system. It can therefore be used as debugging tool on a running system, without having to bother with setup of a parameter service and client. Here is a list of sub-commands to the 'param' command:

```
nanocom-ax # param
'param' contains sub-commands:
  mem                   Set cmds working mem
  list                  List parameters
  set                   Set parameter
  setq                  Set parameter (quiet)
  get                   Get parameter
  copy                  Copy parameter sets from mem to mem
  load                  Load config from file/fram to mem
  save                  Save config from mem to file/fram
  clear                 Clear config from file/fram
```

The local client can be used by simply typing ‘param get <name>’ or ‘param set <name> <value>’, here is an example:

```
nanocom-ax # param get rx_freq
  GET rx_freq = 437250000
nanocom-ax # param set rx_freq 437555000
  SET rx_freq = 437555000 (4)
nanocom-ax # param get rx_freq
  GET rx_freq = 437555000
```

The example shows the ‘rx_freq’ parameter before and after setting the value using the ‘set’ command. Another example is the ‘param list’ command, which will return the parameter table, and the value of each parameter. Here is an example from the NanoCom AX100.

```
nanocom-ax # param list
  0x0000 rx_freq                U32 437555000
  0x0004 rx_baud                U32 4800
  0x000C rx_bw                  U32 15000
  0x0020 tx_freq                U32 437250000
  0x0024 tx_baud                U32 4800
  0x0028 tx_modindex            FLT 0
  0x002C preamb                 X8  0x55
  0x002D preamblen              U8  50
  0x002E preambflags            U8  56
  0x002F intfrm                 X8  0x7E
  0x0030 intfrmlen              U8  0
  0x0031 intfrmflags            U8  56
  0x0034 tx_guard               U16 50
  0x0032 tx_max_time            U16 10
  0x0036 tx_rssibusy            I16 -95
  0x0038 rx_idle_s              U16 3600
  0x003A mode                   U8  2
  0x003B fcs                    U8  1
  0x0040 pllrang_rx             U8  9
  0x0041 pllrang_tx             U8  9
  0x0044 max_temp               I16 60
  0x0050 csp_node               U8  5
  0x0060 reboot_in              U16 0
  0x0064 tx_inhibit             U32 0
  0x0070 bcn_interval           U32 10
  0x0074 bcn_holdoff            U32 4800
  0x0078 bcn_epsdata            U8  1
```

Another very important command is the ‘param mem <memid>’ command, which will change the currently selected memory for the interface. For example the AX100 has two parameter tables, 0 which is the running configuration, and 4 which is the telemetry table. Changing the param mem-id will change which memory area that the get/set/list commands operate on. The default parameter table selected is 0. Here is an example from the AX100 after selecting the telemetry table 4.

```
nanocom-ax # param mem 4
Using param mem 4
```

```
nanocom-ax # param list
  0x0000 temp_brd               I16 248
  0x0002 temp_pa                I16 257
  0x0004 last_rssi              I16 -103
  0x0006 last_rferr             I16 143
  0x0008 tx_count               U32 0
  0x000C rx_count               U32 0
  0x0010 tx_bytes               U32 0
  0x0014 rx_bytes               U32 0
  0x0020 boot_count             U16 32
  0x0024 boot_cause             X32 0x00000100
  0x0028 last_contact           U32 946855591
  0x0030 tot_tx_count           U32 32537
  0x0034 tot_rx_count           U32 172097
  0x0038 tot_tx_bytes           U32 4902966
  0x003C tot_rx_bytes           U32 35692697
```

The ‘save/load/copy/clear’ commands can be used to move entire tables between different memories. For example to load the default boot configuration which is stored in file number zero the command is:

```
param load 0 0
```

This tells the system to read the contents of file number zero into the parameter table with index zero. The same goes for saving a configuration back to the file

```
param save 0 0
```

Or to completely clear a file:

```
param clear 0
```

5.4 GOSH: Remote parameter client

The remote parameter system client can be used to remotely update paramters on other systems from a GOSH command interface. The ‘rparam’ client must first be initialized by downloading the list of remote parameters:

```
nanocom-ax # rparam download 5 1
Downloading RPARAM table for node 5 on port 7, with endian set big = 1
Checksum is: 0xB670
  0x0000 rx_freq
  0x0004 rx_baud
  0x000C rx_bw
  0x0020 tx_freq
  0x0024 tx_baud
  0x0028 tx_modindex
  0x002C preamb
  0x002D preamblen
  0x002E preambflags
  0x002F intfrm
  0x0030 intfrmlen
  0x0031 intfrmflags
  0x0034 tx_guard
  0x0032 tx_max_time
  0x0036 tx_rssibusy
  0x0038 rx_idle_s
  0x003A mode
  0x003B fcs
  0x0040 pllrang_rx
  0x0041 pllrang_tx
  0x0044 max_temp
  0x0050 csp_node
  0x0060 reboot_in
  0x0064 tx_inhibit
  0x0070 bcn_interval
  0x0074 bcn_holdoff
  0x0078 bcn_epsdata
```

Now that the ‘rparam’ client knows which parameters exists on the remote system, we can request or modify individual parameters:

```
nanocom-ax # rparam get preamblen
  REP preamblen = 50 (1)
nanocom-ax # rparam set preamblen 70
  INP preamblen = 70
Result: 1
nanocom-ax # rparam get preamblen
  REP preamblen = 70 (1)
```

We can also modify more than one parameter at once by disabling the ‘rparam autosend’ function and build up more than one get call for each rparam query:

```
nanocom-ax # rparam autosend 0
rparam autosend set to 0
nanocom-ax # rparam get rx_freq
nanocom-ax # rparam get tx_freq
nanocom-ax # rparam send
  REP rx_freq = 437555000 (4)
  REP tx_freq = 437250000 (4)
```

This function is very useful when changing both the uplink and downlink baudrate of a radio. Putting multiple set calls into one packet ensures that the system will never do a partial parameter update.

5.5 Network API

This section will document the following parameter commands:

Table 5.2: List of parameter commands

CMD	Name	Description
0x00	RPARAM_GET	Get one or more parameter values
0xFF	RPARAM_SET	Set one or more parameter values
0x55	RPARAM_REPLY	Packet containing parameters (from GET command)
0x44	RPARAM_TABLE	Download full parameter table
0x77	RPARAM_COPY	Copy from mem to mem
0x88	RPARAM_LOAD	Load from storage to mem
0x99	RPARAM_SAVE	Save from mem to storage
0xAA	RPARAM_CLEAR	Delete from storage

The parameter commands is sent in the following format:

5.5.1 Packet structure

The parameter server expects all requests to be in CSP format, and to contain the following data:

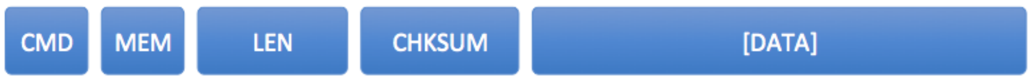

Fig. 5.6: Parameter Packet Format

The fields is described in the following table:

Table 5.3: Packet structure

Field	Length (bytes)	Description
CMD	1	Command ID from table above
MEM	1	Memory area to operate on
LEN	2	Length of DATA field
CHKSUM	2	Fletcher-16 checksum of parameter table with memory ID specified.
DATA	0 .. 180	Data field must be set according to command type

5.5.2 Parameter table checksum

The checksum is a part of all commands, and must match the on-board checksum of the parameter table. The checksum can be obtained by running a fletcher-16 checksum on the parameter table. The algorithm for fletcher-16 is:

```
static inline uint16_t fletcher16(uint8_t* data, int count) {
   uint16_t sum1 = 0;
   uint16_t sum2 = 0;
   int index;
   for (index = 0; index < count; ++index) {
      sum1 = (sum1 + data[index]) % 255;
      sum2 = (sum2 + sum1) % 255;
   }
   return (sum2 << 8) | sum1;
}
```

If you do not have access to the parameter table, you must download the table using RPARAM_TABLE, and run the checksum on the table itself. This check is done in order to verify that the request has been generated using the same parameter table definition as is used by the server. If this is not verified, it may be possible to GET or SET a parameter which does not exist (out of memory scope) and can lead to undefined behavior. In order to bypass the checksum feature, the CHKSUM field can be set to the following magic value: ‘0x0BB0’. This can be helpful while writing test code for the parameter client.

5.5.3 Data formats

The DATA field of the parameter packet is formatted differently for each command. Here is a description of how each command is packed:

```
typedef struct __attribute__ ((packed)) rparam_frame_s {
   rparam_action action;            //! Type is the action performed
   uint8_t mem;               //! Memory area to work on (0 = RUNNING, 1 = SCRATCH)
   uint16_t length;              //! Length of the payload in bytes
   uint16_t checksum;               //! Fletcher's checksum
   uint16_t seq;              //! Sequence number when split over multiple frames
   uint16_t total;            //! Total number of frames
   union {
      uint16_t addr[0];            //! action = PARAM_GET
      uint8_t packed[0];              //! action = PARAM_REPLY | PARAM_SET
      struct {
         uint8_t from;
         uint8_t to;
      } copy;               //! action = PARAM_SAVE | PARAM_LOAD |PARAM_CLEAR
      struct {
         uint8_t id;
      } clear;
   };
} rparam_query;
```

GET command

The GET command uses a packed 16-bit address array. Here you should put the list of parameters to download. A special case is an empty GET command, which will download all parameters in a single packet. Please note that this could is some cases exceed the MTU of the network system, in which case the message will be truncated to the maximum length. In this case you must split your request into two GET commands.

SET / REPLY ‘ Packet data

When setting, or getting a reply on a GET command, the data will come in a serialized packed format. This uses the parameter table to pack data in the following format: <addr><value>, <addr><value>, ‘ etc. The unpack algorithm is as follows:

1. Read 16-bit addr into memory (note that this is big-endian)
2. Find the parameter name, type and size using the parameter table
3. Read the next <size> number of bytes into memory
4. Convert endian from big-endian to host byte order
5. Save result into local memory using memcpy

The serializer and deserializer is the heart of the parameter system, making it possible to request and modify any arbitrary parameter. The source code for this requires a bit of programming experience, but can be obtained by contacting gomspace, or buying a subsystem containing the parameter system.

COPY

The copy data field contains two bytes containing two memory areas <from> and <to>.

SAVE / LOAD / CLEAR

The save/load/clear commands uses the mem-id from the parameter packet header, and an additional storage file-id to know which file to operate on.

5.5.4 Parameter Table

The table download requires no data in the DATA field of the parameter request packet. The response will be formatted as an array of the following data structures:

Table 5.4: Parameter table format

Field	Length (bytes)	Description
ADDR	2	Parameter Address
TYPE	1	Type of parameter (see table below)
SIZE	1	Size of the parameter
FLAGS	1	Reserved field
NAME	14	Name of parameter

5.5.5 Parameter Types

The parameter table describes the parameter types using a single byte according to the following table:

Table 5.5: List of parameter types

Type	Name	Description
0	PARAM_UINT8	8-bit unsigned
1	PARAM_UINT16	16-bit unsigned
2	PARAM_UINT32	32-bit unsigned
3	PARAM_UINT64	64-bit unsigned
4	PARAM_INT8	8-bit signed
5	PARAM_INT16	16-bit signed
6	PARAM_INT32	32-bit signed
7	PARAM_INT64	64-bit signed
8	PARAM_X8	8-bit unsigned (printed as hex)
9	PARAM_X16	16-bit unsigned (printed as hex)
10	PARAM_X32	32-bit unsigned (printed as hex)
11	PARAM_X64	64-bit unsigned (printed as hex)
12	PARAM_DOUBLE	Double
13	PARAM_FLOAT	Float
14	PARAM_STRING	Array of bytes, dynamic length, printed as string
15	PARAM_DATA	Array of bytes, dynamic length, printed as hex
16	PARAM_BOOL	Boolean

5.5.6 Parameter table download

When requesting the parameter table, the reply will come as CSP packets using the Simple Fragmentation Protocol (SFP). The reason is that the parameter table will normally span multiple CSP packets, and without any additional information, it is not possible to detect a missing packet. If you have the latest version of libcsp, SFP will come as standard, otherwise look at libcsp.org in order to get the source code for this protocol. The SFP protocol adds a 32-bit data_offset and a 32-bit total_length, field at the end of every CSP message. This can be used to know if any data has been missed during reception and do another request. SFP does not have automatic retransmission, for that you need to enable RDP. If you do not wish to use the SFP protocol, you can simply throw away the last 8-bytes of the message.

5.6 Structure

The GomSpace Parameter Library is structured as follows:

Table 5.6: GomSpace Parameter Library structure

Folder	Description
libparam/client	Client headers and code for GomSpace Parameter Library. See *API* below for further details
libparam/bindings/python	Python bindings for GomSpace Parameter Library
libparam/include/param	GomSpace Parameter Library server and host API header files
libparam/src	GomSpace Parameter Library server and host API source code files.
libparam/doc	The doc folder contains the source code for this documentation

5.7 API

The remote client API for GomSpace Parameter Library.

```
#ifndef _GS_PARAM_RPARAM_CLIENT_H_
#define _GS_PARAM_RPARAM_CLIENT_H_
/**
```

```
 * GomSpace Parameter System
 *
 * Copyright 2014 GomSpace ApS. All rights reserved.
 */

#include <param/param_types.h>

void cmd_rparam_setup(void);
int rparam_get_single(void * out, uint16_t addr, param_type_t type, int size, int mem_id, int node, i
int rparam_set_single(void * in, uint16_t addr, param_type_t type, int size, int mem_id, int node, in

/*******************************************************
 **                     Utils
 *******************************************************/

#define RPARAM_QUIET                                1
#define RPARAM_NOT_QUIET                        0


/**
 * Download all values of the tables in a table index
 * @param index pointer to table index for which we want to download values
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @returns 0 if table download succeeded, <0 otherwise
 */
int rparam_get_full_table(param_index_t* index, int node, int port, int index_id, int timeout);

/**
 * Set whether rparam API should print to stdout or not
 * @param quiet RPARAM_QUIET or RPARAM_NOT_QUIET
 */
void rparam_set_quiet(uint8_t quiet);

/**
 * Download a table specification from a remote node, store it in memory and save it to local file sy
 * @param fname name of the file to store the table specification in
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param index pointer table index memory to store the specification in
 * @param checksum pointer to memory to store the table specification checksum in
 * @returns CMD_ERROR_NONE on success, CMD_ERROR_FAIL otherwise
 */
int rparam_download_table_spec_from_remote_and_save_to_file(const char* fname, uint8_t node, uint8_t

/**
 * Load a table specification from a local file and store it in memory
 * @param fname name of the file to load table specification from
 * @param index pointer to table index memory to store the specification in
 * @param checksum pointer to memory to store the table specification checksum in
 * @returns CMD_ERROR_NONE on success, CMD_ERROR_INVALID or CMD_ERROR_NOMEM otherwise
 */
int rparam_load_table_spec_from_file(const char* fname, param_index_t* index, uint16_t* checksum);


/*******************************************************
 **            Memory and file manipulation
 *******************************************************/

/**
 * Copy from one memory area to another
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @param from memory area to copy from
 * @param to memory area to copy to
 * @returns 1 on success, 0 otherwise
 */
int rparam_copy(uint8_t node, uint8_t port, uint32_t timeout, uint8_t from, uint8_t to);

/**
 * Save contents of a memory area to persistent storage
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @param from memory area to save
 * @param to file id to save to
 * @returns 1 on success, 0 otherwise
 */
int rparam_save(uint8_t node, uint8_t port, uint32_t timeout, uint8_t from, uint8_t to);

/**
 * Load contents from persistent storage to memory
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @param from file id to load from
 * @param to memory area to load to
 * @returns 1 on success, 0 otherwise
 */
int rparam_load(uint8_t node, uint8_t port, uint32_t timeout, uint8_t from, uint8_t to);

/**
 * Clear a file on persistent storage
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @param id file id to clear
 * @returns 1 on success, 0 otherwise
 */
int rparam_clear(uint8_t node, uint8_t port, uint32_t timeout, uint8_t id);

/*****************************************************
 **                Query interface
 *****************************************************/

/**
 * Add a 'get' query to the current query, after a succesfull rparam_query_send()
 * the parameter value can be read using rparam_queury_get_value()
 * @param index pointer to the local table index index
 * @param param_name name of the parameter to get
 * @returns CMD_ERROR_NONE on succces, CMD_ERROR_FAIL or CMD_ERROR_INVALID otherwise
 * @see rparam_query_get_value(), rparam_query_send()
 */
int rparam_query_get(param_index_t* index, const char* param_name);

/**
 * Get a pointer to a queried value.
 * @param index pointer to the local table index
 * @param param_name name of the parameter
 * @param param_no parameter number, use 0 for non-array values
 * @returns NULL on error, a void* to the queried value otherwise
 */
int rparam_query_get_value(param_index_t* index, const char* param_name, uint16_t param_no, void* val
/**
 * Add a 'set' query to the current query. Use rparam_query_send() to execute the set query.
 * @param index pointer to the local table index
 * @param param_name name of the parameter to set
 * @param values array of values to set, multiple values can be set for array type parameters
 * @value_count number of elements in values
 * @returns CMD_ERROR_NONE on succces, CMD_ERROR_FAIL or CMD_ERROR_INVALID otherwise
 * @see rparam_query_send()
 */
int rparam_query_set(param_index_t* index, const char* param_name, char* values[], uint8_t value_cour

/**
 * Send the current query
 * @param index pointer to the local table index
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @checksum parameter table checksum
 * @returns CMD_ERROR_NONE on success, CMD_ERROR_FAIL otherwise
 */
int rparam_query_send(param_index_t* index, uint8_t node, uint8_t port, uint16_t checksum);

/**
 * Reset the current query
 */
void rparam_query_reset();

/**
 * Dump the current query to stdout
 * @param index pointer to local table index
 */
void rparam_query_print(param_index_t* index);



/******************************************************
 **               Getters and Setters
 ******************************************************/

/**
 * Remote getters for parameters.
 * @param out pointer to the output buffer/parameter
 * @param outlen length of the supplied out-buffer, only applies to rparam_get_string()
 * @param addr local address of the parameter
 * @param index_id id of the index in which the parameter table is located
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @returns <0 on error in which case the contents of out is invalid, >0 on success
 */
static inline int rparam_get_string(char *out, int outlen, uint16_t addr, int index_id, int node, int
        return rparam_get_single(out, addr, PARAM_STRING, outlen, index_id, node, port, timeout);
}

/**
 * Remote setters for parameters.
 * @param in pointer to the buffer/parameter value to set
 * @param inlen length of the supplied in-buffer, only applies to rparam_set_string()
 * @param addr local address of the parameter
 * @param index_id id of the index in which the parameter table is located
 * @param node CSP address of the node to query
 * @param port CSP port of the parameter server
 * @param timeout timeout on remote CSP calls
 * @returns <0 on error in which case the contents of out is invalid, >0 on success
 */
static inline int rparam_set_string(char *in, int inlen, uint16_t addr, int index_id, int node, int p
        return rparam_set_single(in, addr, PARAM_STRING, inlen, index_id, node, port, timeout);
}
static inline int rparam_get_uint8(uint8_t * out, uint16_t addr, int index_id, int node, int port, in
        return rparam_get_single(out, addr, PARAM_UINT8, sizeof(uint8_t), index_id, node, port, timeo
}
static inline int rparam_set_uint8(uint8_t * in, uint16_t addr, int index_id, int node, int port, int
        return rparam_set_single(in, addr, PARAM_UINT8, sizeof(uint8_t), index_id, node, port, timeou
}
static inline int rparam_get_uint16(uint16_t * out, uint16_t addr, int index_id, int node, int port,
        return rparam_get_single(out, addr, PARAM_UINT16, sizeof(uint16_t), index_id, node, port, tim
}
static inline int rparam_set_uint16(uint16_t * in, uint16_t addr, int index_id, int node, int port, i
        return rparam_set_single(in, addr, PARAM_UINT16, sizeof(uint16_t), index_id, node, port, time
}
static inline int rparam_get_uint32(uint32_t * out, uint16_t addr, int index_id, int node, int port,
        return rparam_get_single(out, addr, PARAM_UINT32, sizeof(uint32_t), index_id, node, port, tim
}
static inline int rparam_set_uint32(uint32_t * in, uint16_t addr, int index_id, int node, int port, i
        return rparam_set_single(in, addr, PARAM_UINT32, sizeof(uint32_t), index_id, node, port, time
}
static inline int rparam_get_uint64(uint64_t * out, uint16_t addr, int index_id, int node, int port,
        return rparam_get_single(out, addr, PARAM_UINT64, sizeof(uint64_t), index_id, node, port, tim
}
static inline int rparam_set_uint64(uint64_t * in, uint16_t addr, int index_id, int node, int port, i
        return rparam_set_single(in, addr, PARAM_UINT64, sizeof(uint64_t), index_id, node, port, time
}
static inline int rparam_get_int8(int8_t * out, uint16_t addr, int index_id, int node, int port, int
        return rparam_get_single(out, addr, PARAM_INT8, sizeof(int8_t), index_id, node, port, timeout
}
static inline int rparam_set_int8(int8_t * in, uint16_t addr, int index_id, int node, int port, int t
        return rparam_set_single(in, addr, PARAM_INT8, sizeof(int8_t), index_id, node, port, timeout)
}
static inline int rparam_get_int16(int16_t * out, uint16_t addr, int index_id, int node, int port, in
        return rparam_get_single(out, addr, PARAM_INT16, sizeof(int16_t), index_id, node, port, timeo
}
static inline int rparam_set_int16(int16_t * in, uint16_t addr, int index_id, int node, int port, int
        return rparam_set_single(in, addr, PARAM_INT16, sizeof(int16_t), index_id, node, port, timeou
}
static inline int rparam_get_int32(int32_t * out, uint16_t addr, int index_id, int node, int port, in
        return rparam_get_single(out, addr, PARAM_INT32, sizeof(int32_t), index_id, node, port, timeo
}
static inline int rparam_set_int32(int32_t * in, uint16_t addr, int index_id, int node, int port, int
        return rparam_set_single(in, addr, PARAM_INT32, sizeof(int32_t), index_id, node, port, timeou
}
static inline int rparam_get_int64(int64_t * out, uint16_t addr, int index_id, int node, int port, in
        return rparam_get_single(out, addr, PARAM_INT64, sizeof(int64_t), index_id, node, port, timeo
}
static inline int rparam_set_int64(int64_t * in, uint16_t addr, int index_id, int node, int port, int
        return rparam_set_single(in, addr, PARAM_INT64, sizeof(int64_t), index_id, node, port, timeou
}
static inline int rparam_get_float(float * out, uint16_t addr, int index_id, int node, int port, int
        return rparam_get_single(out, addr, PARAM_FLOAT, sizeof(float), index_id, node, port, timeout
}
static inline int rparam_set_float(float * in, uint16_t addr, int index_id, int node, int port, int t
        return rparam_set_single(in, addr, PARAM_FLOAT, sizeof(float), index_id, node, port, timeout)
}
static inline int rparam_get_double(double * out, uint16_t addr, int index_id, int node, int port, in
        return rparam_get_single(out, addr, PARAM_DOUBLE, sizeof(double), index_id, node, port, timeo
}
static inline int rparam_set_double(double * in, uint16_t addr, int index_id, int node, int port, int
        return rparam_set_single(in, addr, PARAM_DOUBLE, sizeof(double), index_id, node, port, timeou
}

#endif
```

Examples of how to use the remote client API of GomSpace Parameter Library.

6. GomSpace Log Library

6.1 Client Commands

Here is a list of client commands for liblog:

```
gosh # log <tab>
  print                Set printmask
  store                Set storemask
  list                 Show groups
  insert               Insert log message
  hist                 Show history
```

Furthermore there is an alias for ‘log print’:

```
nanocom-ax # help debug
  debug                log: Alias of 'log print'
```

6.1.1 print & store

The ‘log print’, ‘debug’ and ‘log store’ commands will change the logging levels for different groups, either for the console (print) target or the ram/fram (store) target. Here is an example of setting the debug flag for the ‘mac’ group:

```
gosh # log print mac
usage: print <group>[,group] <Err|Wrn|Inf|Dbg|Trc|Std|All|No>
gosh # log print mac debug
Set mask of group mac from 0x18 to 0x1E
```

6.1.2 list

The ‘list’ commands shows all groups and their ‘print’ and ‘store’ targets:

```
nanocom-ax # log list
Group           Print Store
axsem           EWIDT EW...
event           ..... .....
spidma          EWIDT EW...
...
```

6.1.3 insert

The insert command will create a new logging entry in the ‘user’ group of the logging system. This can be used to test the logging system and during testing to mark the begining or the result of a certain test:

```
gosh # log insert
usage: insert <level> <message>
gosh # log insert info "Beginning of some test"
3120366290.557 user: Beginning of some test
```

The accepted log level names are: ‘error’, ‘warn’, ‘info’, ‘debug’, trace’

6.1.4 hist

The hist command will show the latest stored log entries:

```
nanocom-ax # log insert error "Test error msg"
3120367702.941 user: Test error msg
nanocom-ax # log insert warn "Test warning"
3120367708.950 user: Test warning
nanocom-ax # log hist
3120367708.950395 user: Test warning
3120367702.941871 user: Test error msg
```

7. GomSpace Util Library

7.1 Introduction

The GomSpace Util Library contains a number of usefull utilities that can be used when developing A3200 applications.

The following sections desctibes the GomSpace Util Library structure and API.

7.2 Structure

The GomSpace Util Library is structured as follows:

Table 7.1: GomSpace Util Library structure

Folder	Description
libutil/include	The include folder contains the GomSpace Util Library API header files
libutil/src	The src folder contains the GomSpace Util Library API source code files. See *API* below for further details
libutil/doc	The doc folder contains the source code for this documentation

7.3 API

The GomSpace Util Library includes API functions for

- Time and clock functions
- Byte order handling
- Print and print formatting functions
- CRC and checksum functions
- Delay functions
- Compression & decompression functions (LZO)
- Hash, List, Linked List, and Array operation
- Linux specific functions

7.3.1 Time and clock functions

Clock

The clock module contains functions for getting and setting the Real Time Clock (RTC) and for getting the time since last boot.

The clock module uses the type `gs_timestamp_t` for exchanging timestamps:

```
typedef struct {
   uint32_t tv_sec;
   uint32_t tv_nsec;
} timestamp_t;
```

Getting the time as a local time timestamp. The elements tv_sec and tv_nsec represents the seconds and nano-seconds since 1970.

```
void gs_clock_get_time(timestamp_t * time);
```

Setting the time from an UTC timestamp.The elements tv_sec and tv_nsec must represent the seconds and nano-seconds since 1970 as an UTC timestamp.

```
void gs_clock_set_time(timestamp_t * utc_time);
```

Getting the time as a monotonic timestamp (i.e. seconds and nano-seconds since boot):

```
void gs_clock_get_monotonic(timestamp_t * time);
```

Getting the nano-seconds since boot as 64 bit unsigned integer:

```
uint64_t gs_clock_get_nsec(void);
```

Timestamp

The timestamp module contains function for added, subtracting, comparing and copying timestamps. The timestamps are of type `gs_timestamp_t` as described in *Clock*.

```
/** Add two timestamps, return result in base */
int timestamp_add(gs_timestamp_t * base, gs_timestamp_t * add);

/** Subtract two timestamps, return result in base */
int timestamp_diff(gs_timestamp_t * base, gs_timestamp_t * diff);

/** Compare two timestamps, return 1 if test > base, otherwise 0 */
int timestamp_ge(gs_timestamp_t * base, gs_timestamp_t * test);

/** Copy timestamp from 'from' to 'to' */
int timestamp_copy(gs_timestamp_t * from, gs_timestamp_t * to);
```

7.3.2 Byte order handling

Byteorder

The byte order module contains functions for converting numbers from host to network byte order and vice verca. The byte order module also contains functions for converting numbers from host to little/big endian byte order and vice verca.

Host - Network byte order:

```
/** Convert 16-bit integer from host byte order to network byte order
 * @param h16 Host byte order 16-bit integer*/
uint16_t util_htons(uint16_t h16);

/** Convert 16-bit integer from host byte order to host byte order
 * @param n16 Network byte order 16-bit integer */
uint16_t util_ntohs(uint16_t n16);

/** Convert 32-bit integer from host byte order to network byte order
 * @param h32 Host byte order 32-bit integer */
uint32_t util_htonl(uint32_t h32);

/** Convert 32-bit integer from host byte order to network byte order
 * @param h32 Host byte order 32-bit integer */
uint32_t util_ntohl(uint32_t n32);
```

```
/** Convert 16-bit integer from host byte order to network byte order
 * @param h16 Host byte order 16-bit integer */
uint16_t util_hton16(uint16_t h16);

/** Convert 16-bit integer from host byte order to host byte order
 * @param n16 Network byte order 16-bit integer */
uint16_t util_ntoh16(uint16_t n16);

/** Convert 32-bit integer from host byte order to network byte order
 * @param h32 Host byte order 32-bit integer */
uint32_t util_hton32(uint32_t h32);

/** Convert 32-bit integer from host byte order to host byte order
 * @param n32 Network byte order 32-bit integer */
uint32_t util_ntoh32(uint32_t n32);

/** Convert 64-bit integer from host byte order to network byte order
 * @param h64 Host byte order 64-bit integer */
uint64_t util_hton64(uint64_t h64);

/** Convert 64-bit integer from host byte order to host byte order
 * @param n64 Network byte order 64-bit integer */
uint64_t util_ntoh64(uint64_t n64);

/** Convert float from network to host byte order
 * @param float in host byte order */
float util_htonflt(float f);

/** Convert float from network to host byte order
 * @param float in network byte order */
float util_ntohflt(float f);

/** Convert double from network to host byte order
 * @param d double in network byte order */
double util_htondbl(double d);

/** Convert double from network to host byte order
 * @param d double in network byte order */
double util_ntohdbl(double d);
```

Host - Little/Big Endian byte order:

```
/** Convert 16-bit integer from host byte order to big endian byte order
 * @param h16 Host byte order 16-bit integer */
uint16_t util_htobe16(uint16_t h16);

/** Convert 16-bit integer from host byte order to little endian byte order
 * @param h16 Host byte order 16-bit integer */
uint16_t util_htole16(uint16_t h16);

/** Convert 16-bit integer from big endian byte order to little endian byte order
 * @param be16 Big endian byte order 16-bit integer */
uint16_t util_betoh16(uint16_t be16);

/** Convert 16-bit integer from little endian byte order to host byte order
 * @param le16 Little endian byte order 16-bit integer */
uint16_t util_letoh16(uint16_t le16);

/** Convert 32-bit integer from host byte order to big endian byte order
 * @param h32 Host byte order 32-bit integer */
uint32_t util_htobe32(uint32_t h32);

/** Convert 32-bit integer from little endian byte order to host byte order
```

```
 * @param h32 Host byte order 32-bit integer */
uint32_t util_htole32(uint32_t h32);

/** Convert 32-bit integer from big endian byte order to host byte order
 * @param be32 Big endian byte order 32-bit integer */
uint32_t util_betoh32(uint32_t be32);

/** Convert 32-bit integer from little endian byte order to host byte order
 * @param le32 Little endian byte order 32-bit integer */
uint32_t util_letoh32(uint32_t le32);

/** Convert 64-bit integer from host byte order to big endian byte order
 * @param h64 Host byte order 64-bit integer */
uint64_t util_htobe64(uint64_t h64);

/** Convert 64-bit integer from host byte order to little endian byte order
 * @param h64 Host byte order 64-bit integer */
uint64_t util_htole64(uint64_t h64);

/** Convert 64-bit integer from big endian byte order to host byte order
 * @param be64 Big endian byte order 64-bit integer */
uint64_t util_betoh64(uint64_t be64);

/** Convert 64-bit integer from little endian byte order to host byte order
 * @param le64 Little endian byte order 64-bit integer */
uint64_t util_letoh64(uint64_t le64);
```

7.3.3 Print and print formatting functions

Time

The time module contains function for formatting a timestamp:

```
int strtime(char *str, int64_t utime, int year_base);
```

The formatted timestamp will be returned in `str`, `utime` is the timestamp as a 64 bit signed integer representing milliseconds.

If the `year_base` is 1970, the formatted timestamp returned in `str` will be:

`day-month-year hour:minute:second.millisecond`,

Example `22-06-2015 12:34:56.234`.

If the `year_base` is not 1970, the formatted timestamp returned in `str` will treated as delta time and will just count the years, months, days, hours, minutes, seconds and milliseconds. If the number years is larger than zero, the formatted string will be:

`<years> years <months> months <days> days <hours> hours <minutes> minutes <seconds> seconds <milliseconds> milliseconds`

For each counter (years, months, days, ...), while the value is zero, the value will be omitted in the output, e.g. if years and months are zero, the format will be:

`<days> days <hours> hours <minutes> minutes <seconds> seconds <milliseconds> milliseconds`

If the `utime` parameter is negative, the formatted string will be prefixed with a '-'.

Byte size

Format a string with a byte size in the format <size> M/K/B, counting the number of 1048576/1024/1 byte blocks respectively.

```
void bytesize(char *buf, int len, long n);
```

Example:

```
char my_str[128];

bytesize(my_str, 128, 512);
bytesize(my_str, 128, 1024);
bytesize(my_str, 128, 1048576);
```

will return the result `512.0B`, `1.0K` and `1.0M` in `my_str`.

Color printf

Color printf allows printing string to the console with colors, using the standard terminal escape sequences for setting the color.

```
void color_printf(color_printf_t color_arg, const char * format, ...);
```

The following colors are defined:

```
COLOR_NONE
COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE
```

The following colors attributes are defined:

```
COLOR_BOLD
```

The colors and color attributes must be OR'ed together, e.g. to print a string in green bold:

```
color_printf(COLOR_GREEN | COLOR_BOLD, "Hello bold-green-world\r\n");
```

Base16

The base16 module contains functions for encoding and decoding base16 arrays to and from strings.

```
/** Base16-encode data
 *
 * The buffer must be the correct length for the encoded string.  Use
 * something like
 *
 *     char buf[ base16_encoded_len ( len ) + 1 ];
 *
 * (the +1 is for the terminating NUL) to provide a buffer of the
 * correct size.
 *
 * @param raw Raw data
 * @param len Length of raw data
 * @param encoded Buffer for encoded string */
```

```
void base16_encode(uint8_t *raw, size_t len, char *encoded);

/** Base16-decode data
 *
 * The buffer must be large enough to contain the decoded data.  Use
 * something like
 *
 *     char buf[ base16_decoded_max_len ( encoded ) ];
 *
 * to provide a buffer of the correct size.
 * @param encoded Encoded string
 * @param raw Raw data
 * @return Length of raw data, or negative error */
int base16_decode(const char *encoded, uint8_t *raw);
```

The base16 module also contains two utility functions for calculating required length of buffers for encoding and decoding:

```
/** Calculate length of base16-encoded data
 * @param raw_len Raw data length
 * @return Encoded string length (excluding NUL) */
static inline size_t base16_encoded_len(size_t raw_len) {
   return (2 * raw_len);
}

/** Calculate maximum length of base16-decoded string
 * @param encoded Encoded string
 * @return Maximum length of raw data */
static inline size_t base16_decoded_max_len(const char *encoded) {
   return ((strlen(encoded) + 1) / 2);
}
```

Hexdump

The hexdump module contains a function for dumping a piece of memory as hex numbers and ascii characters:

```
void hex_dump(void *src, int len);
```

Example:

```
char buf[32];

memset(buf, 0, 32);
strcpy(buf, "01234567");
hex_dump(buf, 32);
```

will print the following:

```
80010120 : 30 31 32 33 34 35 36 37  00 00 00 00 00 00 00 00 | 01234567........
80010130 : 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

where `80010120` is the address of `buf`. The dots to the right are printed if the corresponding byte does not respresent a printable character.

Error

The error module contains error code definitions and a function for getting the error code description as a string.

```
char *error_string(int code);
```

where `code` is one of:

```
#define E_NO_ERR -1
#define E_NO_DEVICE -2
#define E_MALLOC_FAIL -3
#define E_THREAD_FAIL -4
#define E_NO_QUEUE -5
#define E_INVALID_BUF_SIZE -6
#define E_INVALID_PARAM -7
#define E_NO_SS -8
#define E_GARBLED_BUFFER -9
#define E_FLASH_ERROR -10
#define E_BOOT_SER -13
#define E_BOOT_DEBUG -14
#define E_BOOT_FLASH -15
#define E_TIMEOUT -16
#define E_NO_BUFFER -17
#define E_OUT_OF_MEM -18
#define E_FAIL -19
```

7.3.4 CRC and checksum functions

Crc32

The crc32 module functions for calculating CRC32 from a sequence of bytes.

```
/** Calculate single step of crc32 */
uint32_t chksum_crc32_step(uint32_t crc, uint8_t byte);

/** Caluclate crc32 of a block */
uint32_t chksum_crc32(uint8_t *block, unsigned int length);
```

The function `chksum_crc32` takes a pointer to a data `block` together with a `length` parameter and then calculates the CRC32 from the data in `block`.

Normally the function `chksum_crc32_step` is not used outside the crc32 module, but it is possible to use it to manually do the CRC32 calculation. `chksum_crc32_step` takes the current CRC32 value `crc` and updates it with from the `byte` parameter.

7.3.5 Delay functions

Delay

The delay module contains a function for waiting (delaying execution) a number of micro seconds.

```
/** Init delay system
 * @param freq CPU frequency in Hz */
void delay_init(uint32_t freq);

/** Delay for number of microseconds
 * @param us Number of microseconds to wait */
void delay_us(uint32_t us);

/** Delay until condition is true or timeout_us microseconds have passed.
 * @return 0 if condition was met before the timeout, -1 otherwise */
#define delay_until(condition, timeout_us)
```

Before using the delay functions, the `delay_init` should be called with the cpu frequency as parameter, e.g. `delay_init(16000000)` to initialise the delay system with CPU frquency 16 MHz.

The function `delay_us` is called with the number of micro seconds to wait, e.g. `delay_us(1000)` to wait 1 millisecond.

The macro delay_until is called with a condition and the number of micro seconds to wait, e.g. delay_until(my_condition, 10000) to wait until my_condition is not zero or 10 milliseconds have elapsed.

7.3.6 Compression & decompression functions (LZO)

LZO

The LZO module contains function for compressing and decompressing data using the LZO algorithm.

Functions for compressing and decompressing data:

```
int lzo_compress_buffer(void * src, uint32_t src_len, void *out,
                        uint32_t *out_len, uint32_t out_size);
int lzo_decompress_buffer(uint8_t * src, uint32_t srclen, uint8_t * dst,
                          uint32_t dstlen, uint32_t * decomp_len_p);
```

Functions for reading and writing LZO header:

```
int lzo_write_header(lzo_header_t * head);
```

The function lzo_write_header will fill out a LZO header defined by lzo_header_t, including the header checksum:

```
typedef struct __attribute__((packed)) {
   uint8_t magic[9];
   uint16_t version;
   uint16_t libversion;
   uint16_t versionneeded;
   uint8_t method;
   uint8_t level;
   uint32_t flags;
   uint32_t mode;
   uint32_t mtime_low;
   uint32_t mtime_high;
   uint8_t fnamelen;
   uint32_t checksum_header;
   uint8_t data[];
} lzo_header_t;
```

Example code for compressing data from memory to memory

```
int compress_data(uint8_t *data, uint32_t length, uint8_t *comp, uint32_t max_len) {

   if (length > max_len) {
      log_error("Error compressing data - length too large");
      return -1;
   }
   /* Zip / compress */
   uint32_t comp_len = 0;
   if (lzo_compress_buffer(data, length, comp, &comp_len, max_len) != 0) {
      log_error("Error compressing data");
      return -1;
   }

   return comp_len;
}
```

Example code for decompressing data from memory to memory

```
int decompress_data(uint8_t *data, uint32_t length, uint8_t *decomp, uint32_t max_len) {

   if (length > max_len) {
      log_error("Error decompressing data - length too large");
      return -1;
   }
   /* Unip / decompress */
   uint32_t decomp_len = 0;
   if (lzo_decompress_buffer(data, length, decomp, max_len, &decomp_len) != 0) {
      log_error("Error decompressing data");
      return -1;
   }

   return decomp_len;
}
```

7.3.7 Hash, List, Linked List, and Array operation

The uthash module is a set of header files containing some rather clever macros. These macros include **uthash.h** for hash tables, **utlist.h** for linked lists and **utarray.h** for arrays.

The list macros support the basic linked list operations: adding and deleting elements, sorting them and iterating over them. It does so for both single linked list double linked lists and circular lists.

The dynamic array macros supports basic operations such as push, pop, and erase on the array elements. These array elements can be any simple datatype or structure. The array operations are based loosely on the C++ STL vector methods. Internally the dynamic array contains a contiguous memory region into which the elements are copied. This buffer is grown as needed using realloc to accomodate all the data that is pushed into it.

The hash tables provides a good alternative to linked lists for larger tables where scanning through the entire list is going to be slow. The overhead added is larger memory usage and the additional hash processing time, so for short sets of data linked lists are preferred.

7.3.8 Linux specific functions

The folder in src/linux contains different helper functions for linux like exporting and setting gpio pins.

Example code for setting a GPIO pin:

```
#include <util/gpio.h>

/* Valid GPIOs */
enum example_gpio {
   GPIO_EXAMPLE_PIN_1,
   GPIO_EXAMPLE_PIN_2,
   GPIO_EXAMPLE_PIN_3,
};

static struct sysfs_gpio gpios[] = {
   [GPIO_EXAMPLE_PIN_1]  = { "1", "/sys/class/gpio/gpio1/direction", "/sys/class/gpio/gpio1/value", f
   [GPIO_EXAMPLE_PIN_2]  = { "2", "/sys/class/gpio/gpio2/direction", "/sys/class/gpio/gpio2/value", f
   [GPIO_EXAMPLE_PIN_3]  = { "3", "/sys/class/gpio/gpio3/direction", "/sys/class/gpio/gpio3/value", f
};

/* Setting GPIO example pin 2 to high */
gpio_set(&gpios[GPIO_EXAMPLE_PIN_2], 1)
```

8. GomSpace House Keeping library

This chapter describes the GomSpace House Keeping library.

Note: The GomSpace House Keeping library is an add-on library that must be purchased seperately.

8.1 Introduction

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

8.2 System overview

The GomSpace House Keeping library (HK) system is separated into four modules:

- **HK Collect Service**: This is the local interface collecting and persisting telemetry data from different nodes. This module also transmits beacons with configurable intervals.
- **HK Beacon Service**: This module defines the network interfaces and implements a CSP service task to process incoming HK get requests.
- **HK Beacon Parser**: This module handles the decoding and parsing of beacons. Typically used on the ground station.
- **HK Client API**: This module contains functions for sending HK get requests to the HK Service.

This can be illustrated in the following way:

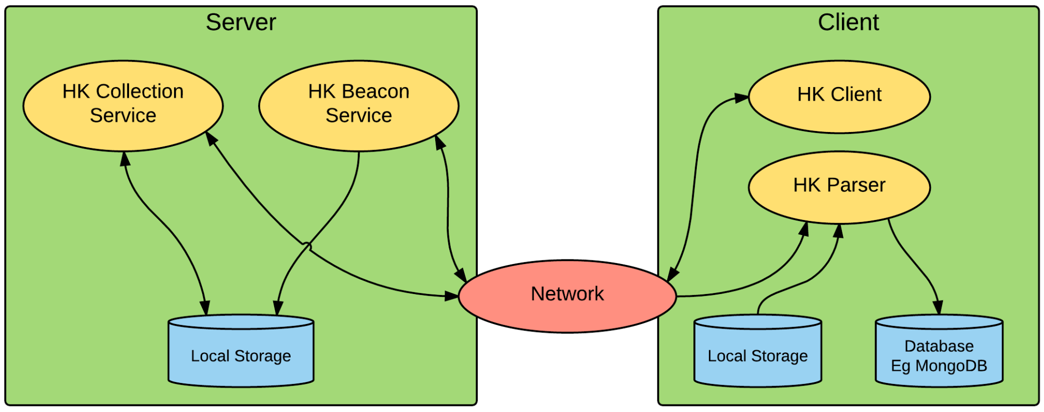

Fig. 8.1: HK system modules

Normally the HK Server modules (HK Collection and HK Beacon services) will run on the OBC (and ADCS) nodes on the satellite, and the HK Client modules will run in csp-term on the Ground Station.

8.3 HK Collection and Beacon Service

A single parameter table is used to setup the HK Collection and HK Beacon Services, see section *HK Collection and HK Beacon Services configuration*.

When all this is in place, the HK Collection Service will collect telemetry from the configured nodes and the HK Beacon Service will encode and transmit beacons from the configured beacon definitions.

The HK Collection Service will collect telemetry from the configured nodes given in the collection table. The collection is controlled by the collection parameters. The collection parameters is a set of time intervals that controls if collection should be active and how often the collection should be done for each entry in the collection table. There is also a global collection enable/disable parameter that can be used to turn collection on/off without changing the individual telemetry collection intervals.

The HK Beacon Service will encode and transmit beacons periodically, controlled by the beacon transmit parameters. The beacon transmit parameters is a set of time intervals that controls if each individual beacon transmit should be active and how often the beacon should be transmitted.

The HK Beacon Service will transmit the beacons to CSP destination address 10, destination port 30. In the current version this cannot be changed. To receive the beacons form the HK Beacon Service, the ground station must include an application with CSP address 10 that receives the beacon data on port 30.

8.3.1 HK Collection and HK Beacon Services configuration

The default table id of the HK configuration table is 19. To change the table id for the HK Collection Parameter table, use the `waf configure` command to specify another table id:

```
waf configure --paramtbl-libhk=<table id>
```

The HK Collection Parameter table is shown below:

Table 8.1: Parameter table 19: HK Collection Parameters

Name	Address	Type	Index	Default	Comment
col_en	0x0000	U8		0	Enable/disable HK collection
store_en	0x0004	U8		0	Enable/disable telemetry caching
store_intv	0x0008	U32		600	How often to persist telemetry cache
store_max	0x000C	U16		2000	Maximum number of telemtry samples ot cache

8.3.2 HK Sampling and Beacon Parameter table

The default table id of the HK Sampling and Beacon Parameter table is 20. To change the table id for the HK Sampling and Beacon Parameter table, use the `waf configure` command to specify another table id:

```
waf configure --paramtbl-libhk-settings=<table id>
```

The HK Sampling and Beacon Parameter table must be defined by the mission specific code, as it is dependant on the mission specific telemetry collection table and beacon definition table. The HK Sampling and Beacon Parameter table contains the telemetry collection intervals for each node in the satellite and the beacon transmit interval for each beacon in the Beacon defintion table.

An example HK Sampling and Beacon Parameter table is shown below:

Table 8.2: Parameter table 20: HK Sampling and Beacon Parameters

Name	Address	Type	Index	Default	Comment
col_obc	0x0000	U8		10	Telemetry collection interval for OBC
col_eps	0x0001	U8		10	Telemetry collection interval for EPS
col_com	0x0002	U8		10	Telemetry collection interval for COM
bcn_interval	0x0004	U16	[0 .. 2]	60 60 60	Beacon transmit interval for each beacon type

If telemetry collection is required for additional nodes, these should be added in the HK Sampling and Beacon Parameter table, one new element similar to eg `col_com` should be added to table 20.

If more than 3 beacon types is required, the `bcn_interval` parameter array size should be increased to match the number of beacon types.

8.3.3 Local caching and storage of collected telemetry

The telemetry collected by the HK Collection Service is by default cached by the HK Collection Service. The number of telemetry samples to cache is configurable. The HK Collection Service can also persist the cached telemetry samples to the local file system. It is possible to turn the persistence to local file system on and off and it is also possible to configure how often the telemetry sample cache is persisted to local file system. In case of a reboot, the telemetry cache is re-initialised from the persisted telemetry samples. This is all controlled by the parameters in the HK Collection Parameter table (table 19), see section libhk_collection_param.

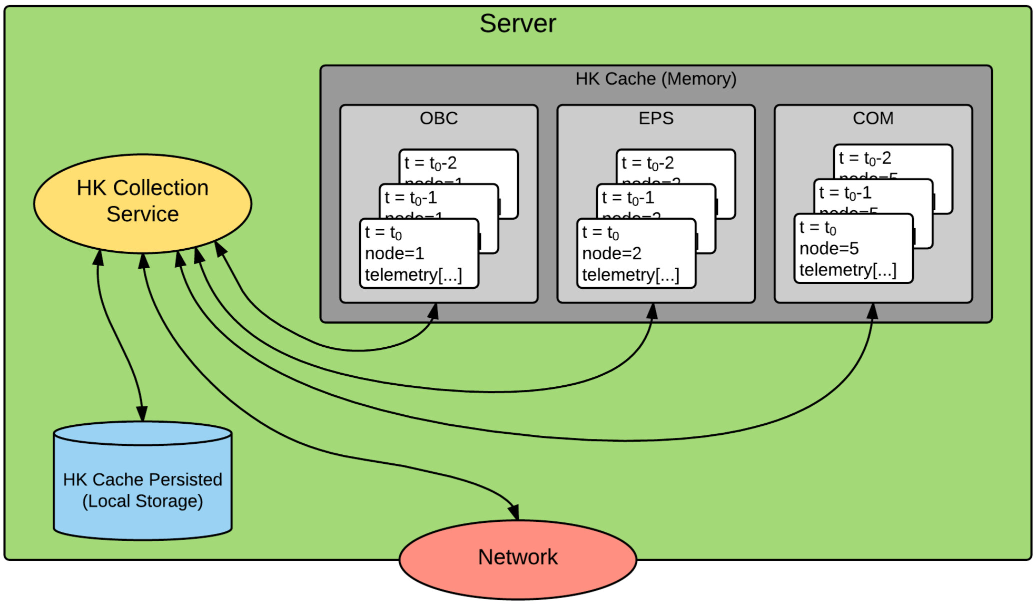

Fig. 8.2: HK Collection Service, caching and persistence

8.3.4 Defining the collection tables

The HK collection tables defines which nodes to collect telemetry from and how to collect it. Collection can be done either via the parameter interface (using the remote parameter interface) or via callback functions that can handle nodes that do not support the parameter interface.

The example code below shows a list of HK collection tables with entries for OBC, EPS and COM. The OBC and COM nodes support the parameter system, but the EPS does not. This is why the `hk_eps2_collect` function has been setup for the EPS node in the second entry. An example `hk_eps2_collect` is provided in section *Example code for collection function*.

```
hk_table_t obc_hk_tables[] = {
  {
      .node = CSP_ADDR_OBC,
      .tableid = A3200_BOARD,
        .memid = PARAM_INDEX_OBC_BOARD,
        .table = a3200_config,
        .count = A3200_CONFIG_COUNT,
        .size = A3200_MEM_SIZE,
        .nodename = "obc",
        .param_interval = MISSION_HK_INTV_OBC,
    },{
        .node = CSP_ADDR_EPS,
        .tableid = EPS_HK,
        .memid = PARAM_INDEX_EPS,
        .table = eps_hk,
        .count = EPS_HK_COUNT,
        .size = EPS_HK_SIZE,
        .nodename = "eps",
        .param_interval = MISSION_HK_INTV_EPS,
        .collect_fct = hk_eps2_collect,
    },{
        .node = CSP_ADDR_COM,
        .tableid = AX100_PARAM_TELEM,
        .memid = PARAM_INDEX_COM,
        .table = ax100_telem,
        .count = AX100_TELEM_COUNT,
        .size = AX100_TELEM_SIZE,
        .nodename = "com",
        .param_interval = MISSION_HK_INTV_COM,
    },
    /* .... */
};
```

The constants in the table above must be defined in a header file:

```
#define PARAM_HK_SETTING        20
#define PARAM_INDEX_OBC_BOARD   21
#define PARAM_INDEX_EPS         22
#define PARAM_INDEX_COM         25

#define CSP_ADDR_OBC            1
#define CSP_ADDR_EPS            2
#define CSP_ADDR_COM            5
```

8.3.5 Defining the beacon packet format

The beacon packets are defined from a list of parameter table addresses and parameter elements. The parameter table address is encoded by the setting the MSB of the entry, eg `0x8000 | PARAM_INDEX_EPS` to indicate parameter table `PARAM_INDEX_EPS`. After the parameter table address follows list of individial parameters from that parameter table.

```
/* Power and Temp: This beacon has voltages, currents and temperatures */
static const uint16_t hk_beacon_format_0[] = {
    /** EPS */
    0x8000 | PARAM_INDEX_EPS,
    EPS_HK_VBOOST(0),
    EPS_HK_VBOOST(1),
    EPS_HK_VBOOST(2),
    EPS_HK_VBATT,
    EPS_HK_CUROUT(0),
    EPS_HK_CUROUT(1),
    EPS_HK_CUROUT(2),
    EPS_HK_CUROUT(3),
    EPS_HK_CUROUT(4),
    EPS_HK_CUROUT(5),
    EPS_HK_CUROUT(6),
    EPS_HK_CURIN(0),
    EPS_HK_CURIN(1),
    EPS_HK_CURIN(2),
    EPS_HK_CURSUN,
    EPS_HK_CURSYS,
    EPS_HK_TEMP(0),
    EPS_HK_TEMP(1),
    EPS_HK_TEMP(2),
    EPS_HK_TEMP(3),
    EPS_HK_TEMP(4),
    EPS_HK_TEMP(5),
    EPS_HK_BATTMODE,
    /** COM */
    0x8000 | PARAM_INDEX_COM,
    AX100_TELEM_TEMP_BRD,
    AX100_TELEM_TEMP_PA,
    AX100_TELEM_LAST_RSSI,
    AX100_TELEM_LAST_RFERR,
    AX100_TELEM_BGND_RSSI,
    /** OBC */
    0x8000 | PARAM_INDEX_OBC_BOARD,
    A3200_CUR_GSSB1,
    A3200_CUR_GSSB2,
    A3200_CUR_FLASH,
    A3200_CUR_PWM,
    A3200_CUR_GPS,
    A3200_CUR_WDE,
    A3200_TEMP_A,
    A3200_TEMP_B,
};
```

8.3.6 Initialize collection parameters and beacon packets

```
/* .... */

#define OBC_HK_NO_TABLES        3
#define OBC_HK_NO_BEACONS       2

hk_table_t hk_tables[OBC_HK_NO_TABLES];
int hk_tables_count = OBC_HK_NO_TABLES;

hk_beacon_t hk_beacons[OBC_HK_NO_BEACONS];
int hk_beacons_count = OBC_HK_NO_BEACONS;

static void mission_fallback(void) {
    /* Default fallback values */
    param_set_uint8(MISSION_HK_INTV_OBC, PARAM_HK_SETTING, 10);
    param_set_uint8(MISSION_HK_INTV_EPS, PARAM_HK_SETTING, 10);
    param_set_uint8(MISSION_HK_INTV_COM, PARAM_HK_SETTING, 10);
    param_set_uint16(MISSION_HK_BCN_INTERVAL(0), PARAM_HK_SETTING, 60);
    param_set_uint16(MISSION_HK_BCN_INTERVAL(1), PARAM_HK_SETTING, 60);
}

static void mission_hk_init(void)
{
    /* Init HK system's own parameters first */
    param_index_set(PARAM_HK_SETTING, (param_index_t) {.table = mission_hk_param, \
         .count = MISSION_HK_PARAM_COUNT, .physaddr = calloc(1, MISSION_HK_SIZE), \
         .size = MISSION_HK_SIZE});
    param_load_fallback(param_ptr(PARAM_HK_SETTING), PARAM_HK_SETTING, PARAM_HK_SETTING,
         &mission_fallback, "HKSET");

    /* Initialize hk tables */
    for (int i = 0; i < OBC_HK_NO_TABLES; i++) {
        hk_tables[i] = obc_hk_tables[i];
    }
    /* Initialize hk beacon definitions */
    for (int i = 0; i < OBC_HK_NO_BEACONS; i++) {
        hk_beacons[i] = obc_hk_beacons[i];
    }
}
```

8.3.7 Initializing the HK Services

The following initialise and creates all the HK Services to run on the HK Server node (eg the OBC).

```
void mission_init(void) {
    /* .... */

    /* Initialize parameters, hk tables and beacon definitions */
    mission_hk_init();

    /* Start hk collect service */
    void hk_collect(void *);
    xTaskCreate(hk_collect, "HKCOL", 4000, NULL, 1, NULL);

    /* Start hk beacon service */
    void hk_service(void *);
    xTaskCreate(hk_service, "HKSRV", 4000, NULL, 1, NULL);

    /* Start hk persist service */
    void hk_persist(void * param);
    xTaskCreate(hk_persist, "HKPERS", 4000, NULL, 1, NULL);
}
```

8.3.8 Example code for collection function

This section contains an example collection function for a node that does not support the parameter interface, in this example the EPS. The concept is to collect the telemetry from the EPS using the eps hk client interface (`eps_hk_get()` to collect all the telemetry parameters and then copy them to a local parameter table that will act as a proxy parameter table for the EPS. In this the HK Collection Service and the HK Sampling and Beacon service does not need to know that the EPS does not support the parameter interface and can handle the EPS in the same way as all other nodes.

```
#include <stdlib.h>
#include <util/log.h>
#include <param/param.h>

int hk_eps2_collect(void) {
   eps_hk_t hk = {};
   if (eps_hk_get(&hk) < 0) {
      return -1;
   }
   int i;
   for (i = 0; i < 3; i++)
      param_set_uint16(EPS_HK_VBOOST(i), PARAM_INDEX_EPS, hk.vboost[i]);
   param_set_uint16(EPS_HK_VBATT, PARAM_INDEX_EPS, hk.vbatt);
   for (i = 0; i < 6; i++)
      param_set_uint16(EPS_HK_CUROUT(i), PARAM_INDEX_EPS, hk.curout[i]);
   for (i = 0; i < 3; i++)
      param_set_uint16(EPS_HK_CURIN(i), PARAM_INDEX_EPS, hk.curin[i]);
   param_set_uint16(EPS_HK_CURSUN, PARAM_INDEX_EPS, hk.cursun);
   param_set_uint16(EPS_HK_CURSYS, PARAM_INDEX_EPS, hk.cursys);
   for (i = 0; i < 6; i++)
      param_set_int16(EPS_HK_TEMP(i), PARAM_INDEX_EPS, hk.temp[i]);
   for (i = 0; i < 8; i++)
      param_set_uint8(EPS_HK_OUT_VAL(i), PARAM_INDEX_EPS, hk.output[i]);
   param_set_uint32(EPS_HK_WDT_I2C_S, PARAM_INDEX_EPS, hk.wdt_i2c_time_left);
   param_set_uint32(EPS_HK_WDT_GND_S, PARAM_INDEX_EPS, hk.wdt_gnd_time_left);
   param_set_uint32(EPS_HK_CNT_BOOT, PARAM_INDEX_EPS, hk.counter_boot);
   param_set_uint32(EPS_HK_CNT_WDTI2C, PARAM_INDEX_EPS, hk.counter_wdt_i2c);
   param_set_uint32(EPS_HK_CNT_WDTGND, PARAM_INDEX_EPS, hk.counter_wdt_gnd);
   for (i = 0; i < 2; i++)
      param_set_uint32(EPS_HK_CNT_WDTCSP(i), PARAM_INDEX_EPS, hk.counter_wdt_csp[i]);
   for (i = 0; i < 2; i++)
      param_set_uint8(EPS_HK_CNT_WDTCSP(i), PARAM_INDEX_EPS, 0);
   param_set_uint8(EPS_HK_BOOTCAUSE, PARAM_INDEX_EPS, hk.bootcause);
   for (i = 0; i < 6; i++)
      param_set_uint16(EPS_HK_LATCHUPS(i), PARAM_INDEX_EPS, hk.latchup[i]);
   param_set_uint8(EPS_HK_BATTMODE, PARAM_INDEX_EPS, hk.battmode);
   param_set_uint8(EPS_HK_PPTMODE, PARAM_INDEX_EPS, hk.pptmode);

   return 0;
}
```

The above code will requires a parameter table defintion for the EPS. The following parameter table will match the above collection function for collection EPS house keeping parameters:

```
#include <stddef.h>
#include <stdint.h>
#include <param/param_types.h>

typedef struct hk_s {
   uint16_t vboost[3];
   uint16_t vbatt;
   uint16_t curout[6];
   uint16_t curin[3];
   uint16_t cursun;
   uint16_t cursys;
   int16_t temp[6];
   uint8_t output[8];
   uint16_t output_on_delta[8];
   uint16_t output_off_delta[8];
   uint32_t wdt_i2c_time_left;
   uint32_t wdt_gnd_time_left;
   uint32_t counter_boot;
   uint32_t counter_wdt_i2c;
   uint32_t counter_wdt_gnd;
   uint32_t counter_wdt_csp[2];
   uint8_t  wdt_csp_pings_left[2];
   uint8_t  bootcause;
   uint16_t latchup[6];
   uint8_t battmode;
   uint8_t pptmode;
} __attribute__((packed)) hk_t;

#define EPS_HK_VBOOST(i)       (offsetof(hk_t, vboost) + sizeof(uint16_t) * i)
#define EPS_HK_VBATT           (offsetof(hk_t, vbatt))
#define EPS_HK_CUROUT(i)       (offsetof(hk_t, curout) + sizeof(uint16_t) * i)
#define EPS_HK_CURIN(i)        (offsetof(hk_t, curin) + sizeof(uint16_t) * i)
#define EPS_HK_CURSUN          (offsetof(hk_t, cursun))
#define EPS_HK_CURSYS          (offsetof(hk_t, cursys))
```

```
#define EPS_HK_TEMP(i)          (offsetof(hk_t, temp) + sizeof(int16_t) * i)
#define EPS_HK_OUT_VAL(i)       (offsetof(hk_t, output) + sizeof(uint8_t) * i)
#define EPS_HK_WDT_I2C_S        (offsetof(hk_t, wdt_i2c_time_left))
#define EPS_HK_WDT_GND_S        (offsetof(hk_t, wdt_gnd_time_left))
#define EPS_HK_CNT_BOOT         (offsetof(hk_t, counter_boot))
#define EPS_HK_CNT_WDTI2C       (offsetof(hk_t, counter_wdt_i2c))
#define EPS_HK_CNT_WDTGND       (offsetof(hk_t, counter_wdt_gnd))
#define EPS_HK_CNT_WDTCSP(i)    (offsetof(hk_t, counter_wdt_csp) + sizeof(uint32_t) * i)
#define EPS_HK_WDT_CSP_C(i)     (offsetof(hk_t, wdt_csp_pings_left) + sizeof(uint8_t) * i)
#define EPS_HK_BOOTCAUSE        (offsetof(hk_t, bootcause))
#define EPS_HK_LATCHUPS(i)      (offsetof(hk_t, latchup) + sizeof(uint16_t) * i)
#define EPS_HK_BATTMODE         (offsetof(hk_t, battmode))
#define EPS_HK_PPTMODE          (offsetof(hk_t, pptmode))

#define EPS_HK_SIZE             sizeof(hk_t)

static const param_table_t eps_hk[] = {
   {.name = "vboost",    .addr = EPS_HK_VBOOST(0),      .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "vbatt",     .addr = EPS_HK_VBATT,          .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "curout",    .addr = EPS_HK_CUROUT(0),      .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "curin",     .addr = EPS_HK_CURIN(0),       .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "cursun",    .addr = EPS_HK_CURSUN,         .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "cursys",    .addr = EPS_HK_CURSYS,         .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "temp",      .addr = EPS_HK_TEMP(0),        .type = PARAM_INT16,  .size = sizeof(int16_t),
   {.name = "out_val",   .addr = EPS_HK_OUT_VAL(0),     .type = PARAM_UINT8,  .size = sizeof(uint8_t),
   {.name = "wdtI2cS",   .addr = EPS_HK_WDT_I2C_S,      .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "wdtGndS",   .addr = EPS_HK_WDT_GND_S,      .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "cntBoot",   .addr = EPS_HK_CNT_BOOT,       .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "cntWdtI2c", .addr = EPS_HK_CNT_WDTI2C,     .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "cntWdtGnd", .addr = EPS_HK_CNT_WDTGND,     .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "cntWdtCsp", .addr = EPS_HK_CNT_WDTCSP(0),  .type = PARAM_UINT32, .size = sizeof(uint32_t)
   {.name = "wdtCspC",   .addr = EPS_HK_WDT_CSP_C(0),   .type = PARAM_UINT8,  .size = sizeof(uint8_t),
   {.name = "bootcause", .addr = EPS_HK_BOOTCAUSE,      .type = PARAM_UINT8,  .size = sizeof(uint8_t)}
   {.name = "latchups",  .addr = EPS_HK_LATCHUPS(0),    .type = PARAM_UINT16, .size = sizeof(uint16_t)
   {.name = "battmode",  .addr = EPS_HK_BATTMODE,       .type = PARAM_UINT8,  .size = sizeof(uint8_t)}
   {.name = "pptmode",   .addr = EPS_HK_PPTMODE,        .type = PARAM_UINT8,  .size = sizeof(uint8_t)}
};
```

8.4 GOSH: HK commands

The HK Client interface can be used to directly request beacons from the ground station. It is possible to specify beacon type, beacon sample interval, number of beacons and timestamp for first beacon sample. By default the requested beacons are transmitted directly to ground station, but if a filename is specified, the beacons are stored in a file instead. This file can then be downloaded later using the ftp interface.

```
csp-term # hk
'hk' contains sub-commands:
  server              Setup hk server
  get                 Request housekeeping
  load                Load beacons from file (only available on ground station)
```

8.4.1 hk server

The `hk server` command configures the csp address and csp port of the HK Service to request beacons from.

Syntax: `hk server <node> <port>`

`<node>`	CSP address of HK Service. Default CSP address is 1
`<port>`	CSP port of HK Service. Default CSP port is 21

8.4.2 hk get

The `hk get` command requests beacon samples from the HK Service configured with the `hk server` command.

Syntax: `hk get <type> [<interval>] [<count>] [<t0>] [<path>]`

`<type>`	The beacon type to request.
`<interval>`	Interval in seconds. Optional, default 60 seconds.
`<count>`	Number of beacon samples. Optional, default 10.
`<t0>`	Timestamp of first beacon sample as UTC epoch. Use 0 for now. Optional, default is 0.
`<path>`	Filename (full path) for storing beacon samples in file instead of transmitting directly. Optional, default is “”.

8.4.3 hk load

The `hk load` command reads beacon samples from a file.

Syntax: `hk load <path> [<offset>] [<count>]`

`<path>`	Filename (full path) to read beacon samples from.
`<offset>`	Number of beacon samples to skip initally. Optional, default is 0.
`<count>`	Number of beacon samples to read after skipping `<offset>` samples. Optional, default is all beacon samples.

9. GomSpace Flight Planner Library

9.1 Introduction

This chapter describes the GomSpace Flight Planner Library.

Note: The GomSpace Flight Planner Library is an add-on library that must be purchased seperately.

The GomSpace Flight Planner Library implements a service to execute GOSH commands at certain timestamps, either absolute or relative to the time of creation. The flight planner service is configured via CSP, so entries can be manipulated with GOSH commands directly on the NanoMind or from CSP-term.

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 GOSH command can run at a time, but multiple entries can be scheduled for future execution. A client library (see `libfp/src/client/ftp_client.c`) implements the client side of the flight planner protocol. The `fp` GOSH commands is implemented on top of this client interface.

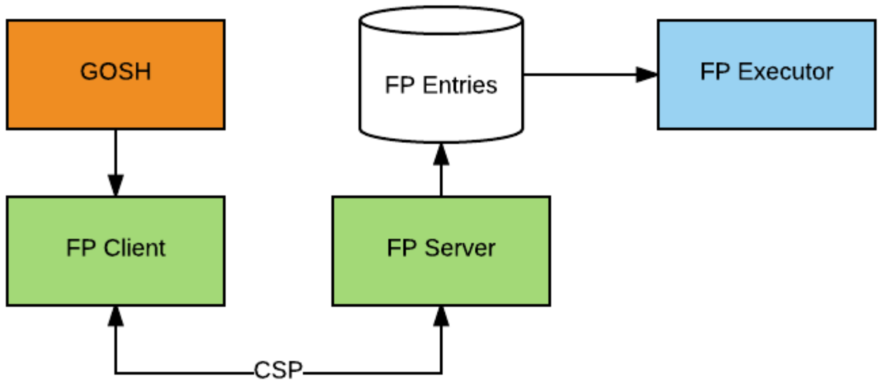

Fig. 9.1: Flight Planner components.

The flight planner operates on entries, each containing a single GOSH command to execute. Entries are tagged with an exectution 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.

9.2 Structure

The GomSpace Flight Planner Library source code is structured as follows:

Folder	Description
libfp/include	GomSpace Flight Planner Library API header files.
libfp/src/server	This folder contains the server implementation.
libfp/src/client	This folder contains the client and GOSH commands.
libfp/doc	The doc folder contains the source code for this documentation.

9.3 GOSH Commands

The GomSpace Shell includes a number of commands for creating and editing flight planner entries, all placed under the global `fp` command. These commands are available in both CSP-term and on NanoMind systems:

Command	Description
server	Set FP server
flush	Fligh current flight plan
load	Load flight plan from file
store	Store current flight plan to file
create	Create a new flight plan entry
delete	Delete flight planner entry
active	Set entry active
allactive	Set all entries active
dormant	Set entry dormant
repeat	Set repeat count
time	Set exection time
list	List current flight planner entries

To interface with a remote flight planner server, you first need to specify its CSP address and port using the `fp server <address> <port>` command. In the default configuration, systems use port 18 for FTP. The remaining FP commands will then use these variables 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 18
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 <name>` or deleted with `fp delete <name>`.

Using the `fp store <path>` and `fp load <path>` commands, it is possible to store and load flight planner entries to the file system. Entries are stored as lines of ASCII text in the following comma-separated format:

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

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

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

Note that only absolute time-tagged entries can be loaded in the current configuration.

9.4 Implementation

To include flight planner support in your image, you must start tasks for the server and execution thread ("FPTIMER"). The recommended place to do so is in the `hook_init();` function in the `src/hook.c` file.

```
void hook_init(void)
{
        ...
        /* FP */
        fp_init();
        fp_server_start(4000, A3200_PORT_FP);
        void fp_timer_task(void * param);
        xTaskCreate(fp_timer_task, "FPTIMER", 4000, NULL, 1, NULL);
        ...
}
```