GOMSPACE

CubeSat Space Protocol (CSP)

Network-Layer delivery protocol for CubeSats and embedded systems.

Feature Overview

- Routed Network Protocol
- Both Connection Oriented and Connection Less Programming Model
- API similar to BSD/Posix Sockets.
- Small Footprint (2700 lines, including comments)
- Designed for very small 8-bit processors
- Dynamic and/or fully Static Buffer handling
- Fully Zero-copy buffer / queue system
- Modular Network Interface System
- Modular OS interface: FreeRTOS, Posix
- Compiles on AVR-8, AVR-32, ARM, PC
- Supported network interfaces: I2C, CAN, RS.232, Loopback.
- Thread and Interrupt Safe Calls.
- Implements RDP (Reliable Datagram Protocol) for

Applications

- Embedded Networks
- Distributed Systems
- Bus-networks (CAN, I2C)
- Serial Communication
- Radio-links for satellites.

Compatibility

- All GomSpace Products:
    - NanoMind, NanoPower, NanoCom, TNC1 and NanoCam.
- CSP-Term (Ground Support)
- ISIS and ClydeSpace subsystems using I2C slave mode.

Functional Description

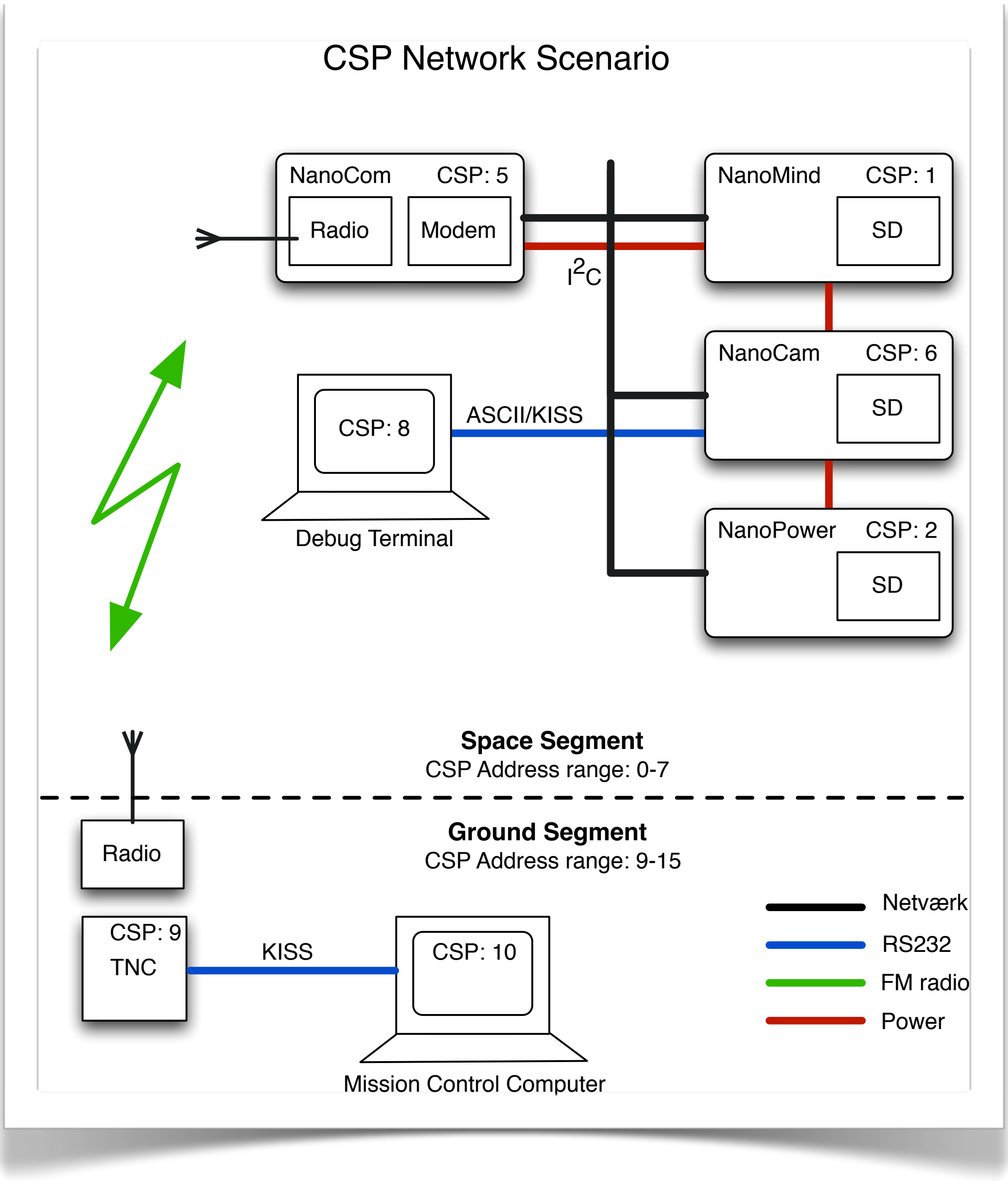
Cubesat Space Protocol (CSP) is a small network-layer delivery protocol designed for Cubesats. CSP enables distributed embedded systems to have a service oriented network topology. The layering of CSP corresponds to the same layers as the TCP/IP model. The implementation consists of a connection oriented transport protocol (Layer 4), a router-core (Layer 3), and several network-interfaces (Layer 1-2). A service oriented topology eases the design of satellite subsystems, since the communication bus itself is the interface to other subsystems. This means that each subsystem developer only needs to think about defining a service-contract, and a set of port-numbers his system will be responding on. Furthermore sub-system inter-dependencies are reduced, and redundancy is easily added by adding multiple similar nodes to the communication bus.

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.

Network Terminology

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



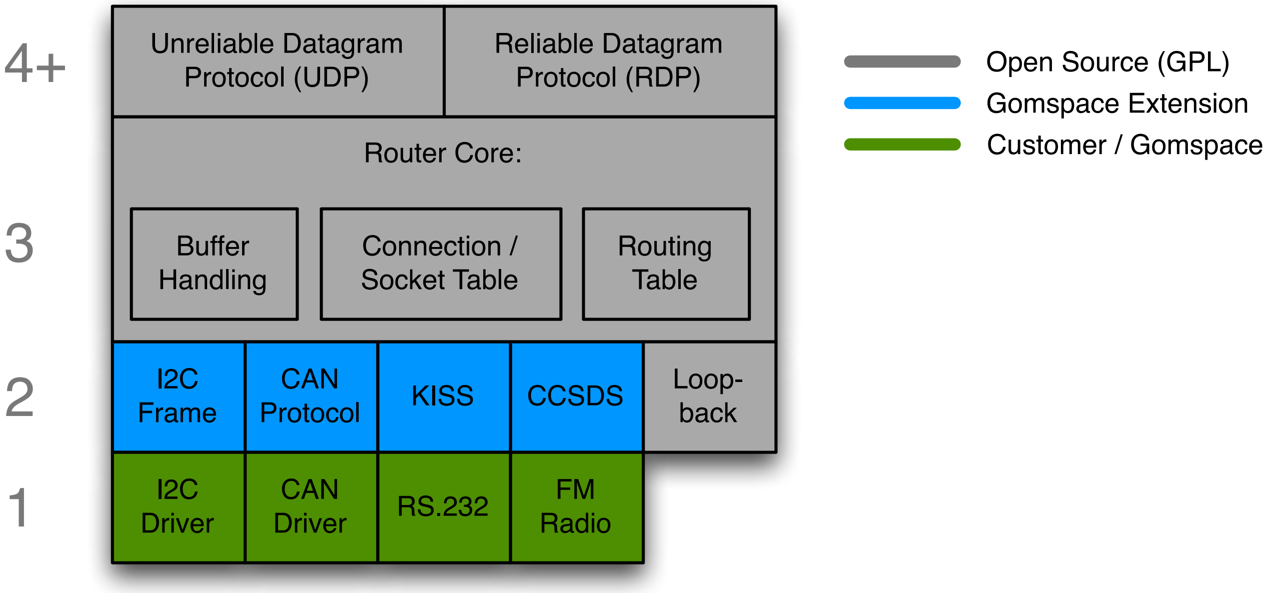
The address range, from 0 to 15, has been segmented into two equal size segments. This allows for easy routing in the network. All addresses starting with binary 1 is on the ground-segment, and all addresses starting with 0 is on the space segment. From CSP 1.0 the address space has been increased to 32 addresses, 0 to 31. But for legacy purposes, the old 0 to 15 is still used in most GomSpace products.

The network is configured using static routing tables programmed into the source-code of the different sub-systems. Each node has a full address table including next-hop interface and packet counters. This means that any given configuration can be made, using any given setup of network interfaces. Looking at the network topology again, it is shown that node 8 is different from the other ground-nodes. Each satellite subsystem is configures to use the NanoCom radio to contact nodes 9-15, and to use it's own RS.232 port to contact node 8.

This means that the debugging terminal can be connected to any subsystem directly and still use CSP for communication. This makes it possible to test and develop the software and protocol for any subsystem without having the fully integrated satellite and radio-link running.

Protocol Stack

The protocol stack is built up of different elements. The author of CSP has chosen to make the core of the stack public available as an open-source and LGPL project. So it is free for anybody to use and adapt to their mission. After being hired at Gomspace, several more advanced extensions have been added. These are therefore only available at Gomspace. The figure below shows how the stack is built.



Layer 1: Drivers

The drivers for the network device is of course very important in order to obtain good stability and performance. GomSpace has already developed drivers to support all GomSpace products. If your mission features another processor or bus-transceiver, a driver must be written for CSP to work on this system. All good drivers use DMA and Interrupt driven drivers where possible. Putting this together with the queueing system of CSP gives a very good performance, low latency and low CPU usage.

Layer 2: MAC-layer protocols

The layer 2 protocol software defines a frame-format that is suitable for the media. The interface can be easily extended with implementations for even more links. For example on the space-link, Gomspace uses the CCSDS framing format with forward error correction, scrambling and a 32-bit attached sync marker. The purpose of the MAC-layer protocol is to remove all this extra information, decode the frame correctly and deliver the arrived data to the router core.

Layer 3: Router Core

The router core is the backbone of the CSP implementation. Not only does it take care of the basic routing function, it also holds some utility code for buffer handling, which must be used by all the drivers. The router works by looking at a 32-bit CSP header which contains the usual delivery and source address together with port numbers for the connection.

There is no routing protocol for automatic route discovery, all routing tables are pre-programmed into the subsystems. This means that the overall topology must be decided before putting subsystems together. An example will be shown later.

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. Since all elements are of the same size, the search algorithm is extremely quick and requires no hard-locking of the processor while running. It can even run from both interrupt and task context at the same time.

The routing table is implemented as a full 1-to-1 map of destination addresses and next-hop interfaces. The connection table can hold a pre-defined number of connection handles at a time, and all handles must be free'd after use, just like buffers.

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.

More examples of how to use the router and how to listen for incoming traffic is shown later.

Layer 4: Transport Extensions

As known from the TCP/IP or the OSI model, layer 3 only takes care of packet delivery, nothing else. In the TCP/IP model, the data is handed to the user-space application after the TCP protocol has ensured the packets are correctly ordered, and there are no data missing. This is very practical when using networks like the internet where traffic might take different routes in the network and packets may be lost. However for a small satellite, the internal data-bus is very reliable and delivers information in the correct order, and the space-link is too slow for a 3-way handshake protocol.

Data packets are therefore handed to the user-space application just as-is, when they entered the node. This resembles the well known UDP (user datagram protocol). 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.

In order to control data integrity when using an unreliable protocol, extra software is needed. GomSpace provides a transport layer extension to simple UDP function called RDP (reliable datagram protocol). RDP is an implementation of the RFC 908 and RFC 1151 directly, which adds automatic retransmission and packet re-ordering on top of UDP. Another way of implementing retransmission is to take care of it on application layer, or using a combination of both application layer and transport layer (RDP), which is how the File Transfer Protocol works.

The next sections will give some code examples.

Hello World Example

Server

```
csp_conn_t * conn;
csp_packet_t * packet;

csp_socket_t * socket = csp_socket(0);
csp_bind(socket, PORT_4);
csp_listen(socket, MAX_CONNS_IN_Q);

while(1) {
  conn = csp_accept(socket, TIMEOUT_MAX);
  packet = csp_read(conn, TIMEOUT_NONE);

  printf(“%S\r\n”, packet->data);

  csp_buffer_free(packet);
  csp_close(conn);
}
```

Client

```
csp_conn_t * conn;
csp_packet_t * packet;

conn = csp_connect(PRIO_NORM, SERVER, PORT_4,
TIMEOUT, 0);
packet = csp_buffer_new(sizeof(csp_packet_t));

sprintf(packet->data, “Hello World”);
packet->length = strlen(“Hello World”);

csp_send(conn, packet, TIMEOUT_NONE);

csp_close(conn);
```

This example shows the entire source-code needed by both the server and client in order to make the server output a “hello world” string, sent from the client. (Note: This code is simplified, normally return value checks must be performed). Lets start the description with the server first:

The server will be using two pointers, or handles, one for the connection and one for the packet being processed. These are declared in the beginning. Then a socket is opened and configured to be able to hold a certain number of incoming connections using the listen call. Then the socket is bound to a specific port-number, in this case 4. Binding to port number PORT_ANY will cause your socket to receive ALL incoming traffic. When the socket is configured, the network is correctly initialized and the main body of the server task can begin. This involves a while(1) loop in order to return to the waiting state when a connection has been handled. The accept call, blocks and waits for an infinite amount of time until a new connection arrives from a client. When a connection arrives, the read call is used to fetch the first incoming packet in this connection. The string contents of this packet will be printed to the screen of the server. By convention, the server now knows that there will be no more data on this connection, and chooses to close it. Before closing the server ensures to free the packet-buffer to ensure all resources are returned to the pool.

The Client is a bit more simple, the only thing needed in order to obtain a connection handle is a call to the connect function. This takes the priority, destination and port number as arguments. Since CSP does not have a transport and handshake protocol, nothing actually happen apart from a new connection resource is reserved. After having obtained the connection handle, a packet buffer is obtained from the buffer pool. The packet is filled with some information, in this case the

“Hello World” string, and before sending it, it is ensured that the packet length field is also set correctly. Then the send function is called. This function accepts the packet handle, and passes it to the correct lower layer device, depending on the destination of the packet. The packet buffer is automatically returned to the pool by the driver once the data has been physically transmitted on the media. Finally, after transmitting the string, it is known by convention that the server will not accept more strings, so the connection is closed.

Simplifying client code

The example from before was quite simplified and no function return values were checked. In real life, there are several things that can fail and must be checked for. To exemplify this, the source code of the csp_transaction function is shown below - in font size 6 :)

```
/**
 * Perform an entire request/reply transaction
 * Copies both input buffer and reply to output buffeer.
 * @param prio CSP Prio
 * @param dest CSP Dest
 * @param port CSP Port
 * @param outbuf pointer to outgoing data buffer
 * @param outlen length of request to send
 * @param inbuf pointer to incoming data buffer
 * @param inlen length of expected reply, -1 for unknown size (note inbuf MUST be large enough)
 * @return Return 1 or reply size if successful, 0 if error or incoming length does not match or -1 if timeout was reached
 */
int csp_transaction(uint8_t prio, uint8_t dest, uint8_t port, int timeout, void * outbuf, int outlen, void * inbuf, int inlen) {

        csp_conn_t * conn = csp_connect(prio, dest, port);
        if (conn == NULL)
                return 0;

        csp_packet_t * packet = csp_buffer_get(sizeof(csp_packet_t));
        if (packet == NULL) {
                csp_close(conn);
                return 0;
        }

        /* Copy the request */
        if (outlen > 0 && outbuf != NULL)
                memcpy(packet->data, outbuf, outlen);
        packet->length = outlen;

        if (!csp_send(conn, packet, 0)) {
                printf("Send failed\r\n");
                csp_buffer_free(packet);
                csp_close(conn);
                return 0;
        }

        /* If no reply is expected, return now */
        if (inlen == 0) {
                csp_close(conn);
                return 1;
        }

        packet = csp_read(conn, timeout);
        if (packet == NULL) {
                printf("Read failed\r\n");
                csp_close(conn);
                return -1;
        }

        if ((inlen != -1) && (packet->length != inlen)) {
                printf("Reply length %u expected %u\r\n", packet->length, inlen);
                csp_buffer_free(packet);
                csp_close(conn);
                return 0;
        }

        memcpy(inbuf, packet->data, packet->length);
        int length = packet->length;
        csp_buffer_free(packet);
        csp_close(conn);
        return length;

}
```

As shown, there are quite some code that needs to be executed just to create a simple request reply. This funciton has therefore been implemented in order to make the programmers life easier. Taking the example from before with the “Hello world” client, this could also be written like this:

```
csp_transaction(PRIO_NORM, SERVER, PORT_4, TIMEOUT_NONE, “Hello World”, 12, NULL, 0);
```

Now this call is entirely secure, we are sure there will be no leakage or breakage, even if the CSP core runs out of buffer space. The function simply returns the transmitted length back to the user for verification that the data was sent.

Writing a good Subsystem API

The previous example showed how the hello-world example could be simplified and improved upon. The improvement was that with csp_transaction, all the error checks and detailed function calls could be implemented in one function, thereby hiding the complexity for the programmer. However this can be even better yet!

So far the example has only shown a simple string being transferred. But what if an entire and more complex data-structure must be transmitted? The solution is to define a programmers interface to your sub-system on a higher-level than network-level. The interface definition is done in a header file as shown below:

subsystem.h

```
#include <stdint.h>

typedef struct __attribute__((packed)) {
  uint8_t byte;
  uint16_t word;
  uint32_t int;
} subsystem_data_type;

int subsystem_get_data(subsystem_data_type * outptr);
```

This header has one data-type, and one function prototype. The idea is that any user-space application can call the funciton “subsystem_get_data()” in order to get a data-structure of the type subsystem_data_type to work with.

Now this functionality or interface must be implemented on both the server and in the client-library. This is shown below:

Server

```
#include <subsystem.h>
#include <csp/csp.h>
extern subsystem_data_type local_data;
...
while(1) {
  conn = csp_accept(socket, TIMEOUT_MAX);
  packet = csp_read(conn, TIMEOUT_NONE);

  /* Cast packet-data to datatype */
  subsystem_data_type * data =
(subsystem_data_type *) packet->data;

  /* Fill in data in correct byte order */
  data->byte = local_data.byte;
  data->word = csp_htons(local_data.word);
  data->int = csp_htonl(local_data.int);

  csp_send(conn, packet, 0);
  csp_close(conn);
}
```

Client-library

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

int subsystem_get_data(subsystem_data_type * p)
{
  /* Retrieve data from server */
  int status = csp_transaction(PRIO_NORM,
SERVER, PORT, TIMEOUT, NULL, 0, p, sizeof(p));

  /* Convert to correct byte order */
  p->word = csp_ntohs(p->word);
  p->int = csp_ntohl(p->int);

  /* Do some more here, perhaps */

  return status;
}
```

Now lets start with the server: The server first includes the public interface description in subsystem.h, then it declares that somewhere externally in the program, an instance of the subsystem data type is stored in a variable with the name local_data. Then the server application listens and accepts an incoming connection. Now the data must be prepared for transmission. This means that it must be copied from the local_data storage into the packet buffer. An easy way to do this is to use the function memcpy() however, this function retains the byte-order of the microprocessor. So the data inserted into the packet could potentially be little-endian, whereas all CSP should be big-endian. Therefore a set of conversion functions are called (htons = Host TO Network Short), these functions ensured that the information is in big-endian, also known as network-byte-order. This is an important step in order to make PC's (little-endian) communicate with ARM/AVR32 big-endian systems.

When the server is implemented, the API function subsystem_get_data() must be implemented in a common library. This code relies on the simplification shown before, namely the csp_transaction function. The funciton could also do some more complex streaming, (de)fragmentation or data-processing, but for the sake of simplicity, the only thing it does is return the data structure. Again the data structure must be converted from network-byte-order to host-byte-order. In this example this is done in-place as shown in the code.

Now that both the server and client-api code is implemented, a user could use the API to do something like this:

some-user-application.c

```
#include <subsystem.h>

int main(void) {

  ...

  subsystem_data_type remote_data;
  if (subsystem_get_data(&remote_data)) {

     printf(“Data was received from subsystem\r\n”);
     printf(“Byte %u\r\n”, remote_data.byte);
     printf(“Word %u\r\n”, remote_data.word);
     printf(“Int &u\r\n”, remote_data.int);

  }

  ...

}
```

This user-application now does not need to care about the network at all. Everything is handled by first the client library, then the CSP transport extension and the CSP router core, the MAC-layer drivers and the physical interface drivers.

Ultimately this gives complete transparency in your code. You can execute this function on any node in the CSP network, even on the local host, a local debugging PC over RS.232 or after your satellite has been launched by the ground-station PC.

This complements the idea of having a service-oriented-architecture, where all the interfaces are defined in files like “subsystem.h”, and the implementation in a common-library.

CSP Initialization

In order to start up CSP on your micro-controller or PC you need to initialize the internal data-pools, setup the routing table and attach some hardware-interfaces. An example is shown below from the NanoMind OBC:

some-user-application.c

```
#include <csp/csp.h>
#include <csp/csp_if_kiss.h>
#include <csp/csp_if_i2c.h>

int main(void) {
  ...
  csp_buffer_init(300, 324) // Allocate 300 buffers of 324 bytes each
  csp_init(NODE_ADDRESS);   // Init internal bufers, and remember the node id.
  csp_i2c_init(NODE_ADDRESS, 0) // Layer 2 init
  csp_kiss_init(F_CPU, USART_BAUD); // Layer 2 init
  csp_route_set(“KISS”, 8, csp_kiss_tx, CSP_HOST_MAC);
  csp_route_set(“DEFAULT”, 16, csp_i2c_tx, CSP_HOST_MAC);
  csp_route_start_task(500, 1);
  ...
}
```

This code first initializes the buffer system, which in this case has been compiled for dynamic one-time allication. Which in this case will allocate 300 buffers of 324 bytes each. Then the core of CSP is initialized, this includes the connection buffer pool and clears the routing table. After this, the two network interfaces on the NanoMind OBC is setup, this is the I2C interface for the satellite bus, and the KISS interface for the serial debugging line. Finally the routing table is setup to use the KISS interface to reach node 8 and the I2C interface to reach everybody else. (Destination node 16 = default route).

CSP API Reference

The most recent API description is always available in the include file csp.h found here:
http://code.google.com/p/cubesat-space-protocol/source/browse/libCSP/trunk/include/csp/csp.h

In order to avoid duplicate API references, the primary resource should always be the inline doxygen comments in this header file. This also makes it easy to use programming environments such as the Eclipse C/C++ IDE. This automatically recognizes the doxygen documentation, and checks datatypes and autocompletes your code.

Contact information

For any questions about CSP and the use hereof, please contact:

Gomspace ApS
Niels Jernes Vej 10
9220 Aalborg East
Denmark

johan@gomspace.com
+45 9635 6113