csp-term Documentation

Release 2.0

GomSpace ApS

May 08, 2015

CONTENTS

Csp-term is a complete linux application using the GomSpace SDK. It can be used as an example of how to develop gomspace compatible applications for linux.

The basis of the GomSpace SDK is the libraries and their individual functionalities. Csp-term includes the following libraries:

- **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

CHAPTER ONE

GETTING STARTED

1.1 How to compile csp-term

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

Install requirements:

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

Configure source:

```
$ waf configure
```

Build source:

```
$ waf build
```

1.2 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
```

CHAPTER

TWO

GOMSPACE UTIL LIBRARY

2.1 Introduction

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

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

2.2 Structure

The GomSpace Util Library is structured as follows:

libutil/include	The src 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.

2.3 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
- Hash, List, Linked List, and Array operation

2.3.1 Time and clock functions

clock.h

Functions

void **clock_uptime**(timestamp_t **time*)

void **clock_get_time**(timestamp_t *)

void **clock_set_time**(timestamp_t *)

void **clock_get_monotonic**(timestamp_t **time*)

uint64_t **clock_get_nsec**(void)

void **clock_restore_time_from_rtc**(void)

Restore the system time from an external RTC clock.

This clock's precision is typically less than a second and should not be called other than from the system bootup.

uint32_t **clock_get_subtick_nsec**(void)

Calculate the currect tick's fractional value, that is how close are we to get another tick. Returns a number between 0 and 1, this includes zero if the timercounter was just reset, but the value 1 should never be obtained since the counter should automatically reset when it reaches its maximum value.

The function is not included in the standard util library, but must be provided by a platform specific function with direct access to the tick timer. If you don't support this feature in your timer, just make a dummy function that returns zero

Todo: there may be an off by one error here.

Return

Tick timer's fractional value converted to nsecs

2.3.2 Byte order handling

byteorder.h

Functions

uint16_t **util_htons**(uint16_t *h16*)

Convert 16-bit integer from host byte order to network byte order

Parameters

- `h16` - Host byte order 16-bit integer

uint16_t **util_ntohs**(uint16_t *n16*)

Convert 16-bit integer from host byte order to host byte order

Parameters

- `n16` - Network byte order 16-bit integer

uint32_t **util_htonl**(uint32_t *h32*)

Convert 32-bit integer from host byte order to network byte order

Parameters

- `h32` - Host byte order 32-bit integer

uint32_t **`util_ntohl`**(uint32_t *n32*)

Convert 32-bit integer from host byte order to network byte order

Parameters

- `h32` - Host byte order 32-bit integer

uint16_t **`util_hton16`**(uint16_t *h16*)

Convert 16-bit integer from host byte order to network byte order

Parameters

- `h16` - Host byte order 16-bit integer

uint16_t **`util_ntoh16`**(uint16_t *n16*)

Convert 16-bit integer from host byte order to host byte order

Parameters

- `n16` - Network byte order 16-bit integer

uint32_t **`util_hton32`**(uint32_t *h32*)

Convert 32-bit integer from host byte order to network byte order

Parameters

- `h32` - Host byte order 32-bit integer

uint32_t **`util_ntoh32`**(uint32_t *n32*)

Convert 32-bit integer from host byte order to host byte order

Parameters

- `n32` - Network byte order 32-bit integer

uint64_t **`util_hton64`**(uint64_t *h64*)

Convert 64-bit integer from host byte order to network byte order

Parameters

- `h64` - Host byte order 64-bit integer

uint64_t **`util_ntoh64`**(uint64_t *n64*)

Convert 64-bit integer from host byte order to host byte order

Parameters

- `n64` - Network byte order 64-bit integer

float **`util_htonflt`**(float *f*)

Convert float from network to host byte order

Parameters

- `float` - in host byte order

float **`util_ntohflt`**(float *f*)

Convert float from network to host byte order

Parameters

- `float` - in network byte order

double **`util_htondbl`**(double *d*)

Convert double from network to host byte order

Parameters

- `d` - double in network byte order

double **`util_ntohdbl`**(double *d*)
Convert double from network to host byte order

Parameters

- `d` - double in network byte order

uint16_t **`util_htobe16`**(uint16_t *h16*)
Convert 16-bit integer from host byte order to big endian byte order

Parameters

- `h16` - Host byte order 16-bit integer

uint16_t **`util_htole16`**(uint16_t *h16*)
Convert 16-bit integer from host byte order to little endian byte order

Parameters

- `h16` - Host byte order 16-bit integer

uint16_t **`util_betoh16`**(uint16_t *be16*)
Convert 16-bit integer from big endian byte order to little endian byte order

Parameters

- `be16` - Big endian byte order 16-bit integer

uint16_t **`util_letoh16`**(uint16_t *le16*)
Convert 16-bit integer from little endian byte order to host byte order

Parameters

- `le16` - Little endian byte order 16-bit integer

uint32_t **`util_htobe32`**(uint32_t *h32*)
Convert 32-bit integer from host byte order to big endian byte order

Parameters

- `h32` - Host byte order 32-bit integer

uint32_t **`util_htole32`**(uint32_t *h32*)
Convert 32-bit integer from little endian byte order to host byte order

Parameters

- `h32` - Host byte order 32-bit integer

uint32_t **`util_betoh32`**(uint32_t *be32*)
Convert 32-bit integer from big endian byte order to host byte order

Parameters

- `be32` - Big endian byte order 32-bit integer

uint32_t **`util_letoh32`**(uint32_t *le32*)
Convert 32-bit integer from little endian byte order to host byte order

Parameters

- `le32` - Little endian byte order 32-bit integer uint64_t **util_htobe64** (uint64_t *h64*)

Convert 64-bit integer from host byte order to big endian byte order

Parameters

- `h64` - Host byte order 64-bit integer

uint64_t **util_htole64** (uint64_t *h64*)

Convert 64-bit integer from host byte order to little endian byte order

Parameters

- `h64` - Host byte order 64-bit integer

uint64_t **util_betoh64** (uint64_t *be64*)

Convert 64-bit integer from big endian byte order to host byte order

Parameters

- `be64` - Big endian byte order 64-bit integer

uint64_t **util_letoh64** (uint64_t *le64*)

Convert 64-bit integer from little endian byte order to host byte order

Parameters

- `le64` - Little endian byte order 64-bit integer

2.3.3 Print and print formatting functions

color_printf.h

Typedefs

typedef enum color_printf_e color_printf_t

Enums

enum color_printf_e

Values:

COLOR_NONE

COLOR_RED

COLOR_YELLOW

COLOR_BLUE

COLOR_GREEN

Functions

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

base16.h

Functions

static size_t **base16_encoded_len** (size_t *raw_len*)

Calculate length of base16-encoded data

Return

Encoded string length (excluding NUL)

Parameters

- `raw_len` - Raw data length

static size_t **base16_decoded_max_len** (**const** char **encoded*)

Calculate maximum length of base16-decoded string

Return

Maximum length of raw data

Parameters

- `encoded` - Encoded string

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

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.

Parameters

- `raw` - Raw data
- `len` - Length of raw data
- `encoded` - Buffer for encoded string

int **base16_decode** (**const** char **encoded*, uint8_t **raw*)

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.

Return

Length of raw data, or negative error

Parameters

- `encoded` - Encoded string
- `raw` - Raw data

hexdump.h

A simple way of dumping memory to a hex output.

Functions

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

Dump memory to debugging output.

Dumps a chunk of memory to the screen

error.h

Error codes.

Defines

E_NO_ERR

E_NO_DEVICE

E_MALLOC_FAIL

E_THREAD_FAIL

E_NO_QUEUE

E_INVALID_BUF_SIZE

E_INVALID_PARAM

E_NO_SS

E_GARBLED_BUFFER

E_FLASH_ERROR

E_BOOT_SER

E_BOOT_DEBUG

E_BOOT_FLASH

E_TIMEOUT

E_NO_BUFFER

E_OUT_OF_MEM

E_FAIL

Functions

char ***error_string**(int *code*)

Prototypes.

Prototypes.

Error code string format Provide fancy debugging/AL error string

Return

pointer to error string

Parameters

- `code` - error code

2.3.4 CRC and checksum functions

crc32.h

Functions

uint32_t **`chksum_crc32_step`** (uint32_t *crc*, uint8_t *byte*)

Calculate single step of crc32

uint32_t **`chksum_crc32`** (uint8_t **block*, unsigned int *length*)

Caluclate crc32 of a block

2.3.5 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.