dmrlib Documentation

Release 0.99.3

Wijnand Modderman-Lenstra

September 03, 2016

Contents

CHAPTER 1

Overview

dmrlib is a multi-platform support library to build applications for Ham Radio applications of the Digital Mobile Radio (DMR) protocol.

CHAPTER 2

Documentation

2.1 bits: bit and byte manipulation

2.1.1 Data types

dmr_bit
Single bit values.

dmr_dibit
Dibit (or double bit) values.

dmr_tribit
Tribit (or triple bit) values.

2.1.2 API

Bit sizes

DMR_SLOT_TYPE_HALF

DMR_SLOT_TYPE_BITS

Byte manipulation

DMR_UINT16_LE(b0, b1)

DMR_UINT16_BE(b0, b1)

DMR_UINT16_LE_PACK(b, n)

DMR_UINT16_BE_PACK(b, n)

DMR_UINT32_LE(b0, b1, b2, b3)

DMR_UINT32_BE(b0, b1, b2, b3)

DMR_UINT32_BE_UNPACK(b)

DMR_UINT32_BE_UNPACK3(b)

DMR_UINT32_LE_PACK(b, n)

DMR_UINT32_LE_PACK3(b, n)

DMR_UINT32_BE_PACK(b, n)

DMR_UINT32_BE_PACK3(b, n)

char ***dmr_byte_to_binary**(uint8_t *byte*)

uint8_t **dmr_bits_to_byte**(bool *bits[8]*)

void **dmr_bits_to_bytes**(bool **bits*, size_t *bits_length*, uint8_t **bytes*, size_t *bytes_length*)

void **dmr_byte_to_bits**(uint8_t *byte*, bool *bits[8]*)

void **dmr_bytes_to_bits**(uint8_t **bytes*, size_t *bytes_length*, bool **bits*, size_t *bits_length*)

2.2 crc: cyclic redundancy checks

2.2.1 API

DMR_CRC8_MASK_VOICE_PI

DMR_CRC8_MASK_VOICE_LC

DMR_CRC8_MASK_TERMINATOR_WITH_LC

void **dmr_crc9**(uint16_t *, uint8_t, uint8_t)

void **dmr_crc9_finish**(uint16_t *, uint8_t)

void **dmr_crc16**(uint16_t *, uint8_t)

void **dmr_crc16_finish**(uint16_t *)

void **dmr_crc32**(uint32_t *, uint8_t)

void **dmr_crc32_finish**(uint32_t *)

2.3 fec: forward error correction

2.3.1 Block Product Turbo Code (196, 96)

2.3.2 Golay (20, 8)

2.3.3 Quadratic Residue (16, 7)

2.3.4 Reed-Solomon

2.3.5 Reed-Solomon (12, 9)

2.3.6 Trellis

2.3.7 Variable Block Product Turbo Code (16, 11)

2.4 log: logging

2.4.1 Data types

dmr_log_priority_t

DMR_LOG_PRIORITY_TRACE

DMR_LOG_PRIORITY_DEBUG

DMR_LOG_PRIORITY_INFO

DMR_LOG_PRIORITY_WARN

DMR_LOG_PRIORITY_ERROR

DMR_LOG_PRIORITY_CRITICAL

DMR_LOG_PRIORITIES

void **(*dmr_log_cb_t)** (void *, *dmr_log_priority_t*, const char *)

2.4.2 API

DMR_LOG_MESSAGE_MAX

DMR_LOG_TIME_FORMAT

DMR_LOG_BOOL (x)

bool **dmr_log_color** (void)

void **dmr_log_color_set** (bool)

dmr_log_priority_t **dmr_log_priority** (void)

void **dmr_log_priority_set** (*dmr_log_priority_t*)

void **dmr_log_priority_reset** (void)

const char ***dmr_log_prefix** (void)

void **dmr_log_prefix_set** (const char *)

void **dmr_log** (const char *, ...)

void **dmr_log_mutex** (const char *, ...)

void **dmr_log_trace** (const char *, ...)

void **dmr_log_debug** (const char *, ...)

void **dmr_log_info** (const char *, ...)

void **dmr_log_warn** (const char *, ...)

void **dmr_log_error** (const char *, ...)

void **dmr_log_errno** (const char *msg)

void **dmr_log_critical** (const char *, ...)

void **dmr_log_message** (*dmr_log_priority_t*, const char *, ...)

void **dmr_log_messagev** (*dmr_log_priority_t*, const char *, va_list)

void **dmr_log_cb_get** (*dmr_log_cb_t* *, void **)

void **dmr_log_cb** (*dmr_log_cb_t*, void *)

2.5 malloc: memory allocation

2.5.1 API

dmr_palloc (void *, type)
Allocate a 0-initizialized structure with parent context.

dmr_palloc_size (void *, size)
Allocate a 0-initizialized untyped buffer with parent context.

dmr_malloc (type)
Allocate a 0-initizialized structure.

dmr_malloc_size (size)
Allocate a 0-initizialized untyped buffer.

dmr_realloc (void *, void*, type, size)
Resize an untyped buffer with parent context.

dmr_strdup (void *, char *)
Duplicate a string with parent context.

dmr_free (void *)
Free a previously allocated structure.

DMR_NULL_CHECK (expr)
Checks an expression for `NULL` returns, sets `ENOMEM`.

DMR_NULL_CHECK_FREE (expr, var)
Checks an expression for `NULL` returns, sets `ENOMEM` and frees `var`.

2.6 packet: DMR packet handling

2.6.1 Data types

dmr_color_code

dmr_data_type
DMR data type. Also includes pseudo data types for internal usage.

DMR_DATA_TYPE_VOICE_PI

DMR_DATA_TYPE_VOICE_LC

DMR_DATA_TYPE_TERMINATOR_WITH_LC

DMR_DATA_TYPE_CSBK

DMR_DATA_TYPE_MBC_HEADER

DMR_DATA_TYPE_MBC_CONTINUATION

DMR_DATA_TYPE_DATA_HEADER

DMR_DATA_TYPE_RATE12_DATA

DMR_DATA_TYPE_RATE34_DATA

DMR_DATA_TYPE_IDLE

DMR_DATA_TYPE_INVALID

DMR_DATA_TYPE_SYNC_VOICE

DMR_DATA_TYPE_VOICE_SYNC

DMR_DATA_TYPE_VOICE

dmr_id
DMR identifier.

dmr_fid
DMR function identifier.

DMR_FID_ETSI

DMR_FID_DMRA

DMR_FID_HYTERA

DMR_FID_MOTOROLA

dmr_flco
Full Link Control Opcode.

DMR_FLCO_GROUP

DMR_FLCO_PRIVATE

DMR_FLCO_INVALID

dmr_ts
DMR time slot.

DMR_TS1

DMR_TS2

Packets

dmr_packet
Raw DMR packet.

dmr_packet_data
Raw DMR packet data bits.

dmr_packet_data_block_bits
Raw DMR packet data block bits.

dmr_packet_data_block
Parsed DMR packet data block.

int **(*dmr_packet_cb)** (*dmr_packet*, void *)
Callback for raw DMR packets.

dmr_parsed_packet
Parsed DMR packet with meta data.

dmr_packet **dmr_parsed_packet.packet**

dmr_ts **dmr_parsed_packet.ts**

dmr_flco **dmr_parsed_packet.flco**

dmr_id **dmr_parsed_packet.src_id**
Source (originating) DMR ID.

dmr_id **dmr_parsed_packet.dst_id**
Target DMR ID.

dmr_id **dmr_parsed_packet.repeater_id**
DMR ID of the repeater or hotspot.

dmr_data_type **dmr_parsed_packet.data_type**

dmr_color_code **dmr_parsed_packet.color_code**

uint8_t **dmr_parsed_packet.sequence**
Sequential number for frames that belong to the same *dmr_parsed_packet.stream_id*.

uint32_t **dmr_parsed_packet.stream_id**
Unique identifier for the stream.

uint8_t **dmr_parsed_packet.voice_frame**
Voice frame index.

bool **dmr_parsed_packet.parsed**

int **(*dmr_parsed_packet_cb)** (*dmr_parsed_packet* *, void *)
Callback for parsed DMR packets.

2.6.2 API

DMR_PACKET_LEN
Size of a single DMR packet (frame) in bytes.

DMR_PACKET_BITS
Size of a single DMR packet (frame) in bits.

DMR_DATA_TYPE_COUNT
Number of valid data types, without pseudo data types.

Note: The *dmr_dump_packet()* and *dmr_dump_parsed_packet()* functions are not ABI stable.

void **dmr_dump_packet** (*dmr_packet*)
Debug function that decodes a raw DMR packet.

void **dmr_dump_parsed_packet** (*dmr_parsed_packet* *)

dmr_parsed_packet ***dmr_packet_decode** (*dmr_packet*)

char ***dmr_flco_name** (*dmr_flco*)

char ***dmr_ts_name** (*dmr_ts*)

char ***dmr_fid_name** (*dmr_fid*)

char ***dmr_data_type_name** (*dmr_data_type*)

char ***dmr_data_type_name_short** (*dmr_data_type*)

int **dmr_payload_bits** (*dmr_packet*, void *)

int **dmr_slot_type_decode** (*dmr_packet*, *dmr_color_code* *, *dmr_data_type* *)

int **dmr_slot_type_encode** (*dmr_packet*, *dmr_color_code*, *dmr_data_type*)

2.7 packetq: DMR packet queue

2.7.1 Data types

dmr_packetq_entry

dmr_packetq_entry.parsed

dmr_packetq_entry.entries

dmr_packetq

dmr_packetq.head

2.7.2 API

dmr_packetq * **dmr_packetq_new** ()
Allocates a new packet queue. Free with *dmr_free()*.

int **dmr_packetq_add** (*dmr_packetq* *, *dmr_parsed_packet* *)

int **dmr_packetq_add_packet** (*dmr_packetq* *, *dmr_packet*)

int **dmr_packetq_shift** (*dmr_packetq* *, *dmr_parsed_packet* **)

int **dmr_packetq_shift_packet** (*dmr_packetq* *, *dmr_packet*)

int **dmr_packetq_foreach** (*dmr_packetq* *, *dmr_parsed_packet_cb*, void *)

int **dmr_packetq_foreach_packet** (*dmr_packetq* *, *dmr_packet_cb*, void *)

2.8 protocol: Site Connect protocols

2.8.1 Homebrew IP Site Connect

Data types

dmr_homebrew_state

DMR_HOMEBREW_AUTH_NONE

DMR_HOMEBREW_AUTH_INIT

DMR_HOMEBREW_AUTH_CONFIG

DMR_HOMEBREW_AUTH_DONE

DMR_HOMEBREW_AUTH_FAILED

dmr_homebrew

struct **dmr_homebrew.config**

char ***dmr_homebrew.config.call**

dmr_id **dmr_homebrew.config.repeater_id**

uint16_t **dmr_homebrew.config.rx_freq**

uint16_t **dmr_homebrew.config.tx_freq**

uint8_t **dmr_homebrew.config.tx_power**

dmr_color_code **dmr_homebrew.config.color_code**

double **dmr_homebrew.config.latitude**

double **dmr_homebrew.config.longitude**

uint16_t **dmr_homebrew.config.altitude**

char ***dmr_homebrew.config.location**

char ***dmr_homebrew.config.description**

char ***dmr_homebrew.config.url**

char ***dmr_homebrew.config.software_id**

char ***dmr_homebrew.config.package_id**

char ***dmr_homebrew.id**
Protocol identificaiton string.

void ***dmr_homebrew.sock**

uint8_t dmr_homebrew.peer_ip[16]

uint16_t **dmr_homebrew.peer_port**

uint8_t dmr_homebrew.bind_ip[16]

uint16_t **dmr_homebrew.bind_port**

uint8_t **dmr_homebrew.state**

char ***dmr_homebrew.secret**
Authentication secret set by *dmr_homebrew_auth()*.

uint8_t dmr_homebrew.nonce[8]
Nonce sent by repeater.

dmr_packetq ***dmr_homebrew.rxq**

dmr_packetq ***dmr_homebrew.txq**

dmr_rawq ***dmr_homebrew.rrq**

dmr_rawq ***dmr_homebrew.trq**

struct timeval **dmr_homebrew.last_ping**

struct timeval dmr_homebrew.last_pong /* last pong received */

API

DMR_HOMEBREW_PORT
Default Homebrew protocol UDP/IP port.

dmr_homebrew * **dmr_homebrew_new**(*dmr_id repeater_id*, uint8_t *peer_ip[16]*, uint16_t *peer_port*, uint8_t *bind_ip[16]*, uint16_t *bind_port*)
Setup a new Homebrew instance.

This function sets up the internal structure as well as the communication sockets, the caller must provide data to the internal config struct before *dmr_homebrew_auth()* is called.

int **dmr_homebrew_auth**(*dmr_homebrew* *homebrew, char *secret*)
Initiate authentication with the repeater.

int **dmr_homebrew_close**(*dmr_homebrew* *homebrew*)
Close the link with the repeater.

int **dmr_homebrew_read**(*dmr_homebrew* *homebrew, *dmr_parsed_packet* **parsed_out*)
Read a packet from the repeater.

This also processes communications with the Homebrew repeater, if the received frame does not contain a DMR packet, the function will set the destination packet pointer to *NULL*.

int **dmr_homebrew_send**(*dmr_homebrew* *homebrew, *dmr_parsed_packet* *parsed*)
Send a DMR frame to the repeater.

int **dmr_homebrew_send_buf**(*dmr_homebrew* *homebrew, uint8_t *buf, size_t *len*)
Send a raw buffer to the repeater.

int **dmr_homebrew_send_raw**(*dmr_homebrew* *homebrew, *dmr_raw* *raw*)
Send a raw packet to the repeater.

int **dmr_homebrew_parse_dmrd**(*dmr_homebrew* *homebrew, *dmr_raw* *raw, *dmr_parsed_packet* **parsed_out*)
Parse a Homebrew protocol DMR data frame.

dmr_protocol **dmr_homebrew_protocol**
Protocol specification.

2.8.2 Multi Mode Digital Voice Modem

Supported hardware

The following MMDVM variants are supported by dmrlib:

- G4KLX MMDVM

- PE1PLM AMBE3000
- PE1PLM DVMEGA

2.9 queue: Berkeley queues

This file defines five types of data structures: singly-linked lists, lists, simple queues, tail queues and XOR simple queues.

2.9.1 API

Singly-linked list

A singly-linked list is headed by a single forward pointer. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of $\mathcal{O}(n)$ removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. Elements being removed from the head of the list should use the explicit macro for this purpose for optimum efficiency. A singly-linked list may only be traversed in the forward direction. Singly-linked lists are ideal for applications with large datasets and few or no removals or for implementing a LIFO queue.

DMR_SLIST_HEAD (name, type)

DMR_SLIST_HEAD_INITIALIZER (head)

DMR_SLIST_ENTRY (type)

DMR_SLIST_FIRST (head)

DMR_SLIST_END (head)

DMR_SLIST_EMPTY (head)

DMR_SLIST_NEXT (elm, field)

DMR_SLIST_FOREACH (var, head, field)

DMR_SLIST_FOREACH_SAFE (var, head, field, tvar)

DMR_SLIST_INIT (head)

DMR_SLIST_INSERT_AFTER (slistelm, elm, field)

DMR_SLIST_INSERT_HEAD (head, elm, field)

DMR_SLIST_REMOVE_AFTER (elm, field)

DMR_SLIST_REMOVE_HEAD (head, field)

DMR_SLIST_REMOVE (head, elm, type, field)

List

A list is headed by a single forward pointer (or an array of forward pointers for a hash table header). The elements are doubly linked so that an arbitrary element can be removed without a need to traverse the list. New elements can be added to the list before or after an existing element or at the head of the list. A list may only be traversed in the forward direction.

DMR_LIST_HEAD (name, type)

DMR_LIST_HEAD_INITIALIZER (head)

DMR_LIST_ENTRY (type)

DMR_LIST_FIRST (head)

DMR_LIST_END (head)

DMR_LIST_EMPTY (head)

DMR_LIST_NEXT (elm, field)

DMR_LIST_FOREACH (var, head, field)

DMR_LIST_FOREACH_SAFE (var, head, field, tvar)

DMR_LIST_INIT (head)

DMR_LIST_INSERT_AFTER (listelm, elm, field)

DMR_LIST_INSERT_BEFORE (listelm, elm, field)

DMR_LIST_INSERT_HEAD (head, elm, field)

DMR_LIST_REMOVE (elm, field)

DMR_LIST_REPLACE (elm, elm2, field)

Simple queue

A simple queue is headed by a pair of pointers, one to the head of the list and the other to the tail of the list. The elements are singly linked to save space, so elements can only be removed from the head of the list. New elements can be added to the list before or after an existing element, at the head of the list, or at the end of the list. A simple queue may only be traversed in the forward direction.

DMR_SIMPLEQ_HEAD (name, type)

DMR_SIMPLEQ_HEAD_INITIALIZER (head)

DMR_SIMPLEQ_ENTRY (type)

DMR_SIMPLEQ_FIRST (head)

DMR_SIMPLEQ_END (head)

DMR_SIMPLEQ_EMPTY (head)

DMR_SIMPLEQ_NEXT (elm, field)

DMR_SIMPLEQ_FOREACH (var, head, field)

DMR_SIMPLEQ_FOREACH_SAFE (var, head, field, tvar)

DMR_SIMPLEQ_INIT (head)

DMR_SIMPLEQ_INSERT_HEAD (head, elm, field)

DMR_SIMPLEQ_INSERT_TAIL (head, elm, field)

DMR_SIMPLEQ_INSERT_AFTER (head, listelm, elm, field)

DMR_SIMPLEQ_REMOVE_HEAD (head, field)

DMR_SIMPLEQ_REMOVE_AFTER (head, elm, field)

DMR_SIMPLEQ_CONCAT (head1, head2)

XOR simple queue

An XOR simple queue is used in the same way as a regular simple queue. The difference is that the head structure also includes a "cookie" that is XOR'd with the queue pointer (first, last or next) to generate the real pointer value.

DMR_XSIMPLEQ_HEAD (name, type)

DMR_XSIMPLEQ_ENTRY (type)

DMR_XSIMPLEQ_XOR (head, ptr)

DMR_XSIMPLEQ_FIRST (head)

DMR_XSIMPLEQ_END (head)

DMR_XSIMPLEQ_EMPTY (head)

DMR_XSIMPLEQ_NEXT (head, elm, field)

DMR_XSIMPLEQ_FOREACH (var, head, field)

DMR_XSIMPLEQ_FOREACH_SAFE (var, head, field, tvar)

DMR_XSIMPLEQ_INIT (head)

DMR_XSIMPLEQ_INSERT_HEAD (head, elm, field)

DMR_XSIMPLEQ_INSERT_TAIL (head, elm, field)

DMR_XSIMPLEQ_INSERT_AFTER (head, listelm, elm, field)

DMR_XSIMPLEQ_REMOVE_HEAD (head, field)

DMR_XSIMPLEQ_REMOVE_AFTER (head, elm, field)

Tail queue

A tail queue is headed by a pair of pointers, one to the head of the list and the other to the tail of the list. The elements are doubly linked so that an arbitrary element can be removed without a need to traverse the list. New elements can be added to the list before or after an existing element, at the head of the list, or at the end of the list. A tail queue may be traversed in either direction.

DMR_TAILQ_HEAD (name, type)

DMR_TAILQ_HEAD_INITIALIZER (head)

DMR_TAILQ_ENTRY (type)

DMR_TAILQ_FIRST (head)

DMR_TAILQ_END (head)

DMR_TAILQ_NEXT (elm, field)

DMR_TAILQ_LAST (head, headname)

DMR_TAILQ_PREV (elm, headname, field)

DMR_TAILQ_EMPTY (head)

DMR_TAILQ_FOREACH (var, head, field)

DMR_TAILQ_FOREACH_SAFE (var, head, field, tvar)

DMR_TAILQ_FOREACH_REVERSE (var, head, headname, field)

DMR_TAILQ_FOREACH_REVERSE_SAFE (var, head, headname, field, tvar)

DMR_TAILQ_INIT (head)

DMR_TAILQ_INSERT_HEAD (head, elm, field)

DMR_TAILQ_INSERT_TAIL (head, elm, field)

DMR_TAILQ_INSERT_AFTER (head, listelm, elm, field)

DMR_TAILQ_INSERT_BEFORE (listelm, elm, field)

DMR_TAILQ_REMOVE (head, elm, field)

DMR_TAILQ_REPLACE (head, elm, elm2, field)

DMR_TAILQ_CONCAT (head1, head2, field)

2.10 raw: raw byte buffers

2.10.1 Data types

dmr_raw
Raw byte buffer with preallocated size.

uint8_t ***dmr_raw.buf**
The actual buffer.

uint64_t **dmr_raw.len**
Number of bytes stored in the buffer.

int64_t **dmr_raw.allocated**
Number of bytes allocated in the buffer.

DMR_TAILQ_ENTRY(dmr_raw) dmr_raw.entries
If this raw buffer is part of a c:type:*dmr_rawq*, this stores pointers to the sibling entries.

dmr_rawq
Zero or more *dmr_raw* buffers in a tail queue.

DMR_TAILQ_HEAD **dmr_rawq.head**
Head of the tail queue.

size_t **dmr_rawq.limit**
Maximum number of buffers in the tail queue.

void **(*dmr_raw_cb)** (*dmr_raw* *, void *)

2.10.2 API

dmr_raw ***dmr_raw_new** (uint64_t *len*)
Setup a new raw buffer.

void **dmr_raw_free** (*dmr_raw* **raw*)
Destroy a raw buffer.

int **dmr_raw_add** (*dmr_raw* **raw*, const void **buf*, size_t *len*)
Add to the raw buffer.

int **dmr_raw_add_hex** (*dmr_raw* **raw*, const void **buf*, size_t *len*)
Add to the raw buffer and hex encode.

int **dmr_raw_add_uint8** (*dmr_raw* *raw, uint8_t *in*)
Add an uint8_t to the buffer.

int **dmr_raw_add_uint16** (*dmr_raw* *raw, uint16_t *in*)
Add an uint16_t to the buffer.

int **dmr_raw_add_uint24** (*dmr_raw* *raw, uint24_t *in*)
Add an uint24_t to the buffer.

int **dmr_raw_add_uint32** (*dmr_raw* *raw, uint32_t *in*)
Add an uint32_t to the buffer.

int **dmr_raw_add_uint32_le** (*dmr_raw* *raw, uint32_t *in*)
Add an uint32_t to the buffer (little endian).

int **dmr_raw_add_uint64** (*dmr_raw* *raw, uint64_t *in*)
Add an uint64_t to the buffer.

int **dmr_raw_add_xuint8** (*dmr_raw* *raw, uint8_t *in*)
Add a hex encoded uint8_t to the buffer.

int **dmr_raw_add_xuint16** (*dmr_raw* *raw, uint16_t *in*)
Add a hex encoded uint16_t to the buffer.

int **dmr_raw_add_xuint24** (*dmr_raw* *raw, uint24_t *in*)
Add a hex encoded uint24_t to the buffer.

int **dmr_raw_add_xuint32** (*dmr_raw* *raw, uint32_t *in*)
Add a hex encoded uint32_t to the buffer.

int **dmr_raw_add_xuint32_le** (*dmr_raw* *raw, uint32_t *in*)
Add a hex encoded uint32_t to the buffer (little endian).

int **dmr_raw_add_xuint64** (*dmr_raw* *raw, uint64_t *in*)
Add a hex encoded uint64_t to the buffer.

int **dmr_raw_addf** (*dmr_raw* *raw, size_t *len*, const char *fmt, ...)
Add a formatted string to the buffer.

int **dmr_raw_grow** (*dmr_raw* *raw, uint64_t *len*)
Resize a raw buffer.

int **dmr_raw_grow_add** (*dmr_raw* *raw, uint64_t *add*)
Resize a raw buffer if *add* number of bytes can't be added.

int **dmr_raw_reset** (*dmr_raw* *raw)
Reset a raw buffer.

int **dmr_raw_zero** (*dmr_raw* *raw)
Zero a raw buffer, maitaining the size.

dmr_rawq ***dmr_rawq_new** (size_t *limit*)
Setup a new raw queue.

A limit of 0 means unlimited queue size.

void **dmr_rawq_free** (*dmr_rawq* *rawq)
Destroy a raw queue.

int **dmr_rawq_add** (*dmr_rawq* *rawq, *dmr_raw* *raw)
Add a buffered element to the raw queue.

int **dmr_rawq_addb** (*dmr_rawq* *rawq, uint8_t *buf, size_t *len*)
Add a buffer to the raw queue.

int **dmr_rawq_each** (*dmr_rawq* *rawq, *dmr_raw_cb* cb, void *userdata)
Iterate over all the items in a raw queue.

bool **dmr_rawq_empty** (*dmr_rawq* *rawq)
Check if the rawq is empty.

size_t **dmr_rawq_size** (*dmr_rawq* *rawq)
Size of the rawq.

dmr_raw ***dmr_rawq_shift** (*dmr_rawq* *rawq)
Shift the first element from the raw queue.

int **dmr_rawq_unshift** (*dmr_rawq* *rawq, *dmr_raw* *raw)
Prepend a raw buffer to the raw queue.

2.11 tree: data trees

This module defines data structures for different types of trees: splay trees and red-black trees.

2.11.1 API

Splay tree

A splay tree is a self-organizing data structure. Every operation on the tree causes a splay to happen. The splay moves the requested node to the root of the tree and partly rebalances it.

This has the benefit that request locality causes faster lookups as the requested nodes move to the top of the tree. On the other hand, every lookup causes memory writes.

The Balance Theorem bounds the total access time for m operations and n inserts on an initially empty tree as $\mathcal{O}((m+n)\log n)$. The amortized cost for a sequence of m accesses to a splay tree is $\mathcal{O}(\log n)$.

DMR_SPLAY_HEAD (name, type)

DMR_SPLAY_INITIALIZER (root)

DMR_SPLAY_INIT (root)

DMR_SPLAY_ENTRY (type)

DMR_SPLAY_LEFT (elm, field)

DMR_SPLAY_RIGHT (elm, field)

DMR_SPLAY_ROOT (head)

DMR_SPLAY_EMPTY (head)

DMR_SPLAY_ROTATE_RIGHT (head, tmp, field)

DMR_SPLAY_LINKLEFT (head, tmp, field)

DMR_SPLAY_LINKRIGHT (head, tmp, field)

DMR_SPLAY_ASSEMBLE (head, node, left, right, field)

DMR_SPLAY_GENERATE (name, type, field, cmp)

DMR_SPLAY_NEGINF

DMR_SPLAY_INF

DMR_SPLAY_INSERT(name, x, y)

DMR_SPLAY_REMOVE(name, x, y)

DMR_SPLAY_FIND(name, x, y)

DMR_SPLAY_NEXT(name, x, y)

DMR_SPLAY_MIN(name, x)

DMR_SPLAY_MAX(name, x)

DMR_SPLAY_FOREACH(x, name, head)

Red-black tree

A red-black tree is a binary search tree with the node color as an extra attribute. It fulfills a set of conditions:

- every search path from the root to a leaf consists of the same number of black nodes,
- each red node (except for the root) has a black parent,
- each leaf node is black.

Every operation on a red-black tree is bounded as $\mathcal{O}(\log n)$. The maximum height of a red-black tree is $2 \log (n + 1)$.

DMR_RB_BLACK

DMR_RB_RED

DMR_RB_NEGINF

DMR_RB_INF

DMR_RB_HEAD(name, type)

DMR_RB_INITIALIZER(root)

DMR_RB_INIT(root)

DMR_RB_ENTRY(type)

DMR_RB_LEFT(elm, field)

DMR_RB_RIGHT(elm, field)

DMR_RB_PARENT(elm, field)

DMR_RB_COLOR(elm, field)

DMR_RB_ROOT(head)

DMR_RB_EMPTY(head)

DMR_RB_SET(elm, parent, field)

DMR_RB_SET_BLACKRED(black, red, field)

DMR_RB_AUGMENT(x)

DMR_RB_ROTATE_LEFT(head, elm, tmp, field)

DMR_RB_ROTATE_RIGHT(head, elm, tmp, field)

DMR_RB_PROTOTYPE(name, type, field, cmp)

DMR_RB_PROTOTYPE_STATIC(name, type, field, cmp)

DMR_RB_GENERATE(name, type, field, cmp)

DMR_RB_GENERATE_STATIC(name, type, field, cmp)

DMR_RB_INSERT(name, x, y)

DMR_RB_REMOVE(name, x, y)

DMR_RB_FIND(name, x, y)

DMR_RB_NFIND(name, x, y)

DMR_RB_NEXT(name, x, y)

DMR_RB_PREV(name, x, y)

DMR_RB_MIN(name, x)

DMR_RB_MAX(name, x)

DMR_RB_FOREACH(x, name, head)

DMR_RB_FOREACH_FROM(x, name, y)

DMR_RB_FOREACH_SAFE(x, name, head, y)

DMR_RB_FOREACH_REVERSE(x, name, head)

DMR_RB_FOREACH_REVERSE_FROM(x, name, y)

DMR_RB_FOREACH_REVERSE_SAFE(x, name, head, y)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

D