Pubnub Contiki Library

Generated by Doxygen 1.8.6

Wed Feb 4 2015 22:53:28

Contents

Chapter 1

The ConTiki OS Pubnub client library

This is the Pubnub client library for the ConTiki OS. It is carefully designed for small footprint and to be a good fit for ConTiki OS way of multitasking with Protothreads and Contiki OS processes. You can have multiple pubnub contexts established; in each context, at most one Pubnub API call/transaction may be ongoing (typically a "publish" or a "subscribe").

It has less features than most Pubnub libraries for other OSes, as it is designed to be used in more constrained environments.

The most important differences from a full-fledged Pubnub API implementation are:

- The only available Pubnub APIs are: publish, subscribe, leave.
- Library itself doesn't handle timeouts other than TCP/IP timeout, which mostly comes down to loss of connection to the server. If you want to impose a timeout on transaction duration, use one of the several ConTiki OS timer interfaces yourself.
- You can't change the origin (URL) or several other parameters of connection to Pubnub.

Chapter 2

Pubnub client library for Contiki OS

Pubnub library for Contiki OS is designed for embedded/constrained devices. It consists of just two files - the header, which has the interface of the library, and the implementation (.c) file. Header is fully documented in Doxygen compatible comments.

There are no special requirements of the library, and it should be usable as-is on any platform that Contiki is ported to.

It has only the basic operations: publish, subscribe and "leave", and is designed with minimal amount of code in mind. It's data memory requirements can be tweaked by the user, but are by design static and brought down to minimum.

File Organization

The files of the library repository are:

- `pubnub.h` : the interface of the library, #include this in your code, and search the comments for documentation. You can also generate Doxygen documentation from it.
- `pubnub.c` : the implementation of the library, compile &link this with your code
- `pubnubDemo.c` : A simple demo of how the library should be used. Build this (with pubnub.c and Contiki) for a basic example of how stuff works.
- `pubnub.t.c` : The unit test. It uses the cgreen unit testing library and has "total" line coverage (only the lines that *can't* be executed - sanity checks - are not covered). You can look at it to see various ways to interact with the library, but as most test code, it's not very user-friendly. You can also build and run it yourself, if you wish.
- `Makefile` : basic Makefile to build the pubnubDemo "app" and pubnub.t unit test. Use are is, or look for clues on how to make one for yourself.
- `LICENSE` and this `README.md` should be self-explanatory.

Design considerations

The fundamental flow for working with the library is this:

0. Foremost, you should have a Contiki OS process to handle the outcome of your requests. You can work without them, but that would be somewhat clumsy. Of course, this can be done in an already existing Contiki OS process of yours.

1. Obtain a Pubnub "context" from the library. It is an opaque pointer. *Note*: you can't create contexts on your own. "All Pubnub contexz are belong to us." :)

2. Initialize the context, giving the subscribe and publish key.
3. Start a operation/transaction/Pubnub API call on the context. It will either fail or return an indication of "started". The outcome will be sent to your process, with event indicating the transaction type and data being the context on which the transaction was carried out.
4. On receipt of the process event, check the context for the result (success or indication of failure). If OK and it was a subscribe transaction, you can get the messages that were fetched (and, if available, the channels they pertain too).
5. When you're done processing the outcome event, you can start a new transaction in that context.

This is ilustrated in pubnubDemo.c, similar to this:

```
static pubnub_t *m_pb = pubnub_get_ctx(0);
pubnub_init(m_pb, pubkey, subkey);

etimer_set(&et, 3*CLOCK_SECOND);

while (1) {
    PROCESS_WAIT_EVENT();
    if (ev == PROCESS_EVENT_TIMER) {
        pubnub_publish(m_pb, channel, "\"ConTiki Pubnub voyager\"");
    }
    else if (ev == pubnub_publish_event) {
        pubnub_subscribe(m_pb, channel);
    }
    else if (ev == pubnub_subscribe_event) {
        for (;;) {
            char const *msg = pubnub_get(m_pb);
            if (NULL == msg) {
                break;
            }
            printf("Received message: %s\n", msg);
        }
        etimer_restart(&et);
    }
}
```

Remarks

- We said there are no requirements, because we assume the minimal Contiki OS. But, in fact, *we do* expect that:
  - you have both UDP and TCP enabled
  - you have IPv4 enabled
  - you have DNS enabled
- While you may have parallel transactions running in different contexts, a single context can handle only one transaction at a time.
- You have to start the `pubnub_process` at some point, preferrably before you initiate any transaction. It does not start automatically. This enables you to start it at your own convinience. The pubnubDemo starts it in the `AUTOSTART_PROCESSES` list, but you don't have to do it like that.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 pubnub Struct Reference

The Pubnub context.

Data Fields

- const char * **publish_key**
- const char * **subscribe_key**
- const char * **uuid**
- const char * **auth**
- char **timetoken** [64]
- struct process * initiator

  Process that started last transaction.
- enum pubnub_res last_result

  The result of the last Pubnub transaction.
- enum pubnub_state **state**
- enum pubnub_trans **trans**
- struct psock **psock**
- char **http_buf** [PUBNUB_BUF_MAXLEN]
- int **http_code**
- unsigned **http_buf_len**
- unsigned **http_content_len**
- bool **http_chunked**
- char **http_reply** [PUBNUB_REPLY_MAXLEN+1]
- unsigned short **msg_ofs**
- unsigned short **msg_end**
- unsigned short **chan_ofs**
- unsigned short **chan_end**

5.1.1 Detailed Description

The Pubnub context.

The documentation for this struct was generated from the following file:

- pubnub.c

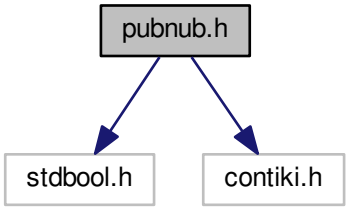
Chapter 6

File Documentation

6.1 pubnub.h File Reference

```
#include <stdbool.h>
#include "contiki.h"
```

Include dependency graph for pubnub.h:



Macros

- #define PUBNUB_CTX_MAX 2

  Maximum number of PubNub contexts.

- #define PUBNUB_BUF_MAXLEN 256

  Maximum length of the HTTP buffer.

- #define PUBNUB_REPLY_MAXLEN 512

  Maximum length of the HTTP reply.

- #define PUBNUB_MISSMSG_OK 1

  If defined, the PubNub implementation will not try to catch-up on messages it could miss while subscribe failed with an IO error or such.

- #define PUBNUB_ORIGIN "pubsub.pubnub.com"

  This is the URL of the Pubnub server.

- #define PUBNUB_USE_MDNS 1

  If `1`, the MDNS module will be used to handle the DNS resolving.

Typedefs

- typedef struct pubnub pubnub_t

  A pubnub context.

Enumerations

- enum pubnub_res {
  PNR_OK, PNR_TIMEOUT, PNR_ABORTED, PNR_IO_ERROR,
  PNR_HTTP_ERROR, PNR_FORMAT_ERROR, PNR_CANCELLED, PNR_STARTED,
  PNR_IN_PROGRESS, PNR_RX_BUFF_NOT_EMPTY, PNR_TX_BUFF_TOO_SMALL }

  Result codes for Pubnub functions and transactions.

Functions

- pubnub_t * pubnub_get_ctx (unsigned index)

  Returns a context for the given index.
- void pubnub_init (pubnub_t *p, const char *publish_key, const char *subscribe_key)

  Initialize a given pubnub context `p` to the `publish_key` and `subscribe_key`.
- void pubnub_done (pubnub_t *p)

  Deinitialize a given pubnub context, freeing all its associated resources.
- void pubnub_set_uuid (pubnub_t *p, const char *uuid)

  Set the UUID identification of PubNub client context `p` to `uuid`.
- void pubnub_set_auth (pubnub_t *p, const char *auth)

  Set the authentication information of PubNub client context `p`.
- void pubnub_cancel (pubnub_t *p)

  Cancel an ongoing API transaction.
- enum pubnub_res pubnub_publish (pubnub_t *p, const char *channel, const char *message)

  Publish the `message` (in JSON format) on `p` channel, using the `p` context.
- char const * pubnub_get (pubnub_t *p)

  Returns a pointer to an arrived message.
- char const * pubnub_get_channel (pubnub_t *pb)

  Returns a pointer to an fetched transaction's next channel.
- enum pubnub_res pubnub_subscribe (pubnub_t *p, const char *channel)

  Subscribe to `channel`.
- enum pubnub_res pubnub_leave (pubnub_t *p, const char *channel)

  Leave the `channel`.
- enum pubnub_res pubnub_last_result (pubnub_t const *p)

  Returns the result of the last transaction in the `p` context.
- int pubnub_last_http_code (pubnub_t const *p)

  Returns the HTTP reply code of the last transaction in the `p` context.
- **PROCESS_NAME** (pubnub_process)

Variables

- process_event_t pubnub_publish_event

  The ID of the Pubnub Publish event.
- process_event_t pubnub_subscribe_event

  The ID of the Pubnub Subscribe event.
- process_event_t pubnub_leave_event

  The ID of the Pubnub Leave event.

6.1.1 Macro Definition Documentation

6.1.1.1 #define PUBNUB_BUF_MAXLEN 256

Maximum length of the HTTP buffer.

This is a major component of the memory size of the whole pubnub context, but it is also an upper bound on URL-encoded form of published message, so if you need to construct big messages, you may need to raise this.

6.1.1.2 #define PUBNUB_CTX_MAX 2

Maximum number of PubNub contexts.

A context is used to publish messages or subscribe (get) them.

Doesn't make much sense to have less than 1. :) OTOH, don't put too many, as each context takes (for our purposes) a significant amount of memory - app. 128 + PUBNUB_BUF_MAXLEN + PUBNUB_REPLY_MAXLEN bytes.

A typical configuration may consist of a single pubnub context for channel subscription and another pubnub context that will periodically publish messages about device status (with timeout lower than message generation frequency).

Another typical setup may have a single subscription context and maintain a pool of contexts for each publish call triggered by an external event (e.g. a button push).

Of course, there is nothing wrong with having just one context, but you can't publish and subscribe at the same time on the same context. This isn't as bad as it sounds, but may be a source of headaches (lost messages, etc).

6.1.1.3 #define PUBNUB_MISSMSG_OK 1

If defined, the PubNub implementation will not try to catch-up on messages it could miss while subscribe failed with an IO error or such.

Use this if missing some messages is not a problem.

Note

messages may sometimes still be lost due to potential PUBNUB_REPLY_MAXLEN overrun issue

6.1.1.4 #define PUBNUB_ORIGIN "pubsub.pubnub.com"

This is the URL of the Pubnub server.

Change only for testing purposes.

6.1.1.5 #define PUBNUB_REPLY_MAXLEN 512

Maximum length of the HTTP reply.

The other major component of the memory size of the PubNub context, beside PUBNUB_BUF_MAXLEN. Replies of API calls longer than this will be discarded and instead, PNR_FORMAT_ERROR will be reported. Specifically, this may cause lost messages returned by subscribe if too many too large messages got queued on the Pubnub server.

6.1.1.6 #define PUBNUB_USE_MDNS 1

If `1`, the MDNS module will be used to handle the DNS resolving.

If 0 the "resolv" module will be used. This is a temporary solution, it is expected that ConTiki will unify those two modules.

6.1.2 Typedef Documentation

6.1.2.1 typedef struct pubnub pubnub_t

A pubnub context.

An opaque data structure that holds all the data needed for a context.

6.1.3 Enumeration Type Documentation

6.1.3.1 enum pubnub_res

Result codes for Pubnub functions and transactions.

Enumerator

PNR_OK Success. Transaction finished successfully.

PNR_TIMEOUT Time out before the request has completed.

PNR_ABORTED Connection to Pubnub aborted (in most cases, a TCP reset was received)

PNR_IO_ERROR Communication error (network or HTTP response format).

PNR_HTTP_ERROR HTTP error. Call pubnub_last_http_code() to get the error code.

PNR_FORMAT_ERROR Unexpected input in received JSON.

PNR_CANCELLED Request cancelled by user.

PNR_STARTED Transaction started. Await the outcome via process message.

PNR_IN_PROGRESS Transaction (already) ongoing. Can't start a new transaction while the old one is in progress.

PNR_RX_BUFF_NOT_EMPTY Receive buffer (from previous transaction) not read, new subscription not allowed.

PNR_TX_BUFF_TOO_SMALL The buffer is to small. Increase PUBNUB_BUF_MAXLEN.

6.1.4 Function Documentation

6.1.4.1 void pubnub_cancel (pubnub_t * p)

Cancel an ongoing API transaction.

The outcome of the transaction in progress will be PNR_CANCELLED.

6.1.4.2 void pubnub_done (pubnub_t * p)

Deinitialize a given pubnub context, freeing all its associated resources.

Needs to be called only if you manage multiple pubnub contexts dynamically.

6.1.4.3 char const* pubnub_get (pubnub_t * p)

Returns a pointer to an arrived message.

Message(s) arrive on finish of a subscribe transaction. Subsequent call to this function will return the next message (if any). All messages are from the channel(s) the last subscription was for.

Note

Context doesn't keep track of the channel(s) you subscribed to. This is a memory saving design decision, as most users won't change the channel(s) they subscribe too.

Parameters

p	The Pubnub context. Can't be NULL.

Returns

Pointer to the message, NULL on error

See Also

pubnub_subscribe

6.1.4.4 char const* pubnub_get_channel (pubnub_t * *pb*)

Returns a pointer to an fetched transaction's next channel.

Each transaction may hold a list of channels, and this functions provides a way to read them. Subsequent call to this function will return the next channel (if any).

Note

You don't have to read all (or any) of the channels before you start a new subscribe transaction.

Parameters

pb	The Pubnub context. Can't be NULL.

Returns

Pointer to the channel, NULL on error

See Also

pubnub_subscribe
pubnub_get

6.1.4.5 pubnub_t* pubnub_get_ctx (unsigned *index*)

Returns a context for the given index.

Contexts are statically allocated by the Pubnub library and this is the only way to get a pointer to one of them.

Parameters

index	The index of the context

Precondition

(index >= 0) && (index < PUBNUB_CTX_MAX)

Returns

Context pointer on success

6.1.4.6 void pubnub_init (pubnub_t * p, const char * publish_key, const char * subscribe_key)

Initialize a given pubnub context `p` to the `publish_key` and `subscribe_key`.

You can customize other parameters of the context by the configuration function calls below.

Note

The `publish_key` and `subscribe` key are expected to be valid (ASCIIZ string) pointers throughout the use of context `p`, that is, until either you call pubnub_done(), or the otherwise stop using it (like when the whole software/ firmware stops working). So, the contents of these keys are not copied to the Pubnub context `p`.

Precondition

Call this after TCP initialization.

Parameters

p	The Context to initialize (use pubnub_get_ctx() to obtain it)
publish_key	The string of the key to use when publishing messages
subscribe_key	The string of the key to use when subscribing to messages

6.1.4.7 int pubnub_last_http_code (pubnub_t const * p)

Returns the HTTP reply code of the last transaction in the `p` context.

6.1.4.8 enum pubnub_res pubnub_last_result (pubnub_t const * p)

Returns the result of the last transaction in the `p` context.

6.1.4.9 enum pubnub_res pubnub_leave (pubnub_t * p, const char * channel)

Leave the `channel`.

This actually means "initiate a leave transaction". You should leave a channel when you want to subscribe to another in the same context to avoid loosing messages.

The outcome is sent to you via a process event, which you are free to ignore, but is a good place to start subscribe to another channel, via pubnub_get().

You can't leave if a transaction is in progress on the context.

Parameters

p	The Pubnub context. Can't be NULL.
channel	The string with the channel name (or comma-delimited list of channel names) to subscribe to.

Returns

PNR_STARTED on success, an error otherwise

6.1.4.10 enum pubnub_res pubnub_publish (pubnub_t * p, const char * channel, const char * message)

Publish the `message` (in JSON format) on `p` channel, using the `p` context.

This actually means "initiate a publish transaction". The outcome is sent to the process that starts the transaction via process event pubnub_publish_event. You don't have to do any special processing of said event - use it at your own convinience (you may retry on failure, for example).

You can't publish if a transaction is in progress in p context.

Parameters

p	The pubnub context. Can't be NULL
channel	The string with the channel (or comma-delimited list of channels) to publish to.
message	The message to publish, expected to be in JSON format

Returns

PNR_STARTED on success, an error otherwise

6.1.4.11 void pubnub_set_auth (pubnub_t * *p*, const char * *auth*)

Set the authentication information of PubNub client context `p`.

Pass NULL to unset.

Note

The `uuid` is expected to be valid (ASCIIZ string) pointers throughout the use of context `p`, that is, until either you call pubnub_done() on `p`, or the otherwise stop using it (like when the whole software/ firmware stops working). So, the contents of the uuid string is not copied to the Pubnub context `p`.

6.1.4.12 void pubnub_set_uuid (pubnub_t * *p*, const char * *uuid*)

Set the UUID identification of PubNub client context `p` to `uuid`.

Pass NULL to unset.

Note

The `uuid` is expected to be valid (ASCIIZ string) pointers throughout the use of context `p`, that is, until either you call pubnub_done() on `p`, or the otherwise stop using it (like when the whole software/ firmware stops working). So, the contents of the `uuid` string is not copied to the Pubnub context `p`.

6.1.4.13 enum pubnub_res pubnub_subscribe (pubnub_t * *p*, const char * *channel*)

Subscribe to `channel`.

This actually means "initiate a subscribe transaction". The outcome is sent to the process that starts the transaction via process event pubnub_publish_event, which is a good place to start reading the fetched message(s), via pubnub_get().

Messages published on `p` channel since the last subscribe transaction will be fetched.

The `channel` string may contain multiple comma-separated channel names, so only one call is needed to fetch messages from multiple channels.

You can't subscribe if a transaction is in progress on the context.

Also, you can't subscribe if there are unread messages in the context (you read messages with pubnub_get()).

Note

Some of the subscribed messages may be lost when calling publish() after a subscribe() on the same context or subscribe() on different channels in turn on the same context. But typically, you will want two separate contexts for publish and subscribe anyway. If you are changing the set of channels you subscribe to, you should first call pubnub_leave() on the old set.

Parameters

p	The pubnub context. Can't be NULL
channel	The string with the channel name (or comma-delimited list of channel names) to subscribe to.

Returns

PNR_STARTED on success, an error otherwise

See Also

pubnub_get

6.1.5 Variable Documentation

6.1.5.1 process_event_t pubnub_leave_event

The ID of the Pubnub Leave event.

Event carries the context pointer on which the leave transaction finished. Use pubnub_last_result() to read the outcome of the transaction.

6.1.5.2 process_event_t pubnub_publish_event

The ID of the Pubnub Publish event.

Event carries the context pointer on which the publish transaction finished. Use pubnub_last_result() to read the outcome of the transaction.

6.1.5.3 process_event_t pubnub_subscribe_event

The ID of the Pubnub Subscribe event.

Event carries the context pointer on which the subscribe transaction finished. Use pubnub_last_result() to read the outcome of the transaction.

Index