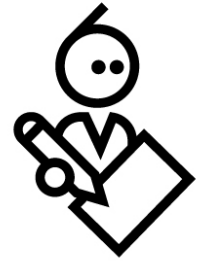




Nabto SDK

How to Write a uNabto Device Application

NABTO/001/TEN/023



Contents

1	Abstract	4
2	Bibliography	4
3	Nabto Platform Basics	5
3.1	Nabto Communication Patterns	6
4	uNabto Device Introduction	6
5	Application components	9
5.1	uNabto framework compile time configuration	9
5.1.1	unabto_config.h	9
5.2	uNabto initialization.....	9
5.3	uNabto Framework Runner	10
5.3.1	Enabling Cryptography	11
5.4	The client query handler	12
5.4.1	Synchronous and asynchronous operation modes.....	12
5.5	Synchronous query request handling	13
5.6	Processing the query.....	14
5.6.1	The query model.....	14
5.6.2	The query parameter types	14
5.6.3	The query list element.....	15
5.6.4	Working with the query model in the application.....	16
5.6.5	Integral types	16
5.6.6	The raw type	18
5.6.7	Lists	19

Abstract

5.7	Asynchronous request handling	21
5.8	General application development notes.....	25
5.9	Query request/response size platform configuration.....	25
6	Streaming	26
6.1	Stream Demo Application	26
6.2	Streaming Usage	27
6.2.1	New streams	27
6.2.2	Reading from a stream	27
6.2.3	Writing to a stream.....	28
6.2.4	Closing a stream	28
6.2.5	Releasing a stream.....	28
6.2.6	Stream Events.....	28
6.3	Stream Configuration.....	29
7	Access Control	29
7.1	Connection level access control.....	29
7.2	Query level access control	30
8	Special event handlers	31
8.1	Getting a UTC time stamp from the basestation	32
8.2	Monitoring the basestation connection status.....	32
9	The log printing framework.....	32
9.1	Printing.....	33
9.2	Controlling log printing by severity.....	33
10	uNabto helper modules.....	34
10.1	ACL	34
10.2	Configuration Store.....	34
11	uNabto platform adapters.....	34
11.1	Porting uNabto - Creating a new uNabto Platform Adapter.....	36
11.1.1	Overall Structure	36
11.1.2	Basic code.....	36
11.1.3	Implementing unabto_platform_types.h.....	38

Abstract

11.1.4	Implementing unabto_platform.h	39
11.1.5	Implementing a Network Adapter	39
11.1.6	Implementing a Time Adapter	40
11.1.7	Implementing a DNS Adapter.....	41
11.1.8	Implementing a Random Adapter	43
12	The uNabto framework source code.....	43
12.1	Where to download the source code	43
12.2	The structure of the source code.....	43
12.2.1	The uNabto framework core source code and header file directory	43
12.2.2	The uNabto platform adapter specific source directories.....	43
12.2.3	The feature module and platform adapter specific source directory	44
12.2.4	API include directories	44
12.3	Building the uNabto SDK with CMake.....	44
12.4	Device platform memory requirements	45
12.4.1	RAM usage	45
12.4.2	ROM usage	45
12.5	Summary of the referenced configuration parameters.....	46

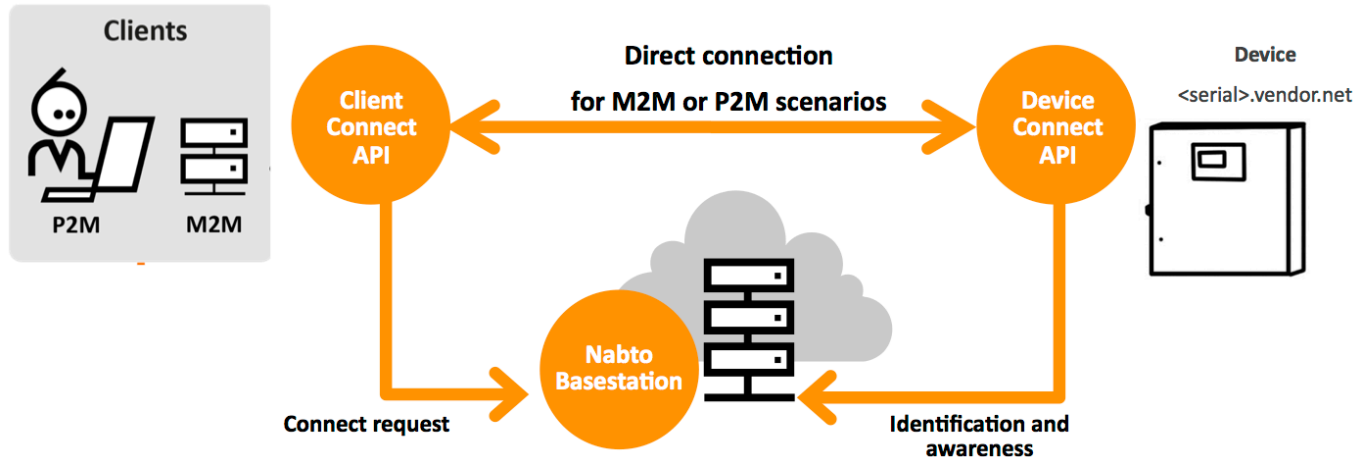
1 Abstract

This document describes the various steps in creating a uNabto Micro Server (a Nabto enabled device application). Application development guidelines, short descriptions of the target device requirements, and the uNabto framework interfaces are the main focus points. After reading this document an application developer should be able to create basic uNabto enabled device applications, written in the C programming language.

2 Bibliography

TEN025	Writing a Nabto Client SDK application
--------	--

3 Nabto Platform Basics



The Nabto platform consists of 3 components:

- Nabto **client**: Libraries supplied by Nabto, used by the customer's application
- Nabto **device**: The uNabto SDK - an open source framework supplied by Nabto, integrated with the customer's device application
- Nabto **basestation**: Services supplied by Nabto (Nabto- or self-hosted) that mediates connections between Nabto clients and devices.

The Nabto client initiates a direct, encrypted connection to the Nabto enabled device – the Nabto basestation mediates this direct connection: The device's unique name, e.g. <serial>.vendor.net, is mapped to the IP address of the Nabto basestation – this is where devices register when online and where clients look for available devices. After connection establishment, the client and device communicates directly with each other, the basestation is out of the loop – no data is stored on the basestation, it only knows about currently available Nabto enabled devices.




The client can also discover the device if located on the same LAN and communicate directly without the basestation – useful for bootstrap scenarios or for offline use.

Integrating Nabto on the customer's device is the topic of [TEN023].

Nabto client applications developed using the Nabto Client SDK described in [TEN025]. The Nabto Client SDK is the lowest level way of developing a Nabto application - several wrappers exist on top of this lowest level SDK to provide a more abstract experience, for instance for developing Cordova/Ionic or Xamarin hybrid apps or just simplify native Android and iOS app development.

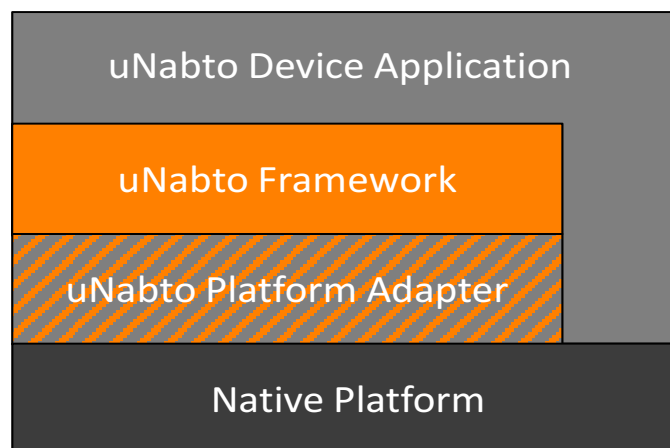
3.1 Nabto Communication Patterns

The Nabto platform supports 3 communication patterns that will be referenced throughout this document:

-  **RPC:** The Nabto P2P-RPC communication mechanism allows a client to securely invoke a remote function on a Nabto device. The device implements an interface definition shared between client and device, the client works with normal JSON documents, exchanged in a compact representation with the device.
-  **Streaming:** Nabto P2P-Streaming can be used for retrieving larger amounts of data from a device or sending e.g. a firmware update. With sufficient resources available on the device, Nabto P2P-Streaming can be used for high performance streaming suitable for video scenarios.
-  **Push:** Nabto Push is used for communication initiated by the device, for instance to implement mobile push notifications or to support big data scenarios where data is collected centrally for further analysis. Nabto Push can also trigger an M2M scenario using RPC or Streaming - e.g. when a certain condition is triggered, the device sends a Nabto Push message and a server function invokes an RPC function or streams data.

4 uNabto Device Introduction

A uNabto device application consists of 3 parts, running on top of the native platform:



The *uNabto Device Application* receives queries from the client via the *uNabto Framework* and may use the *Native Platform* for data storage, IO etc. The *uNabto Device Application* component is created by the application developer and includes integration with e.g. an existing backend to retrieve data from. The uNabto SDK includes demo applications that may be suitable as starting points.

The *uNabto Framework* abstracts all network related communication away from the application. In fact, the application has no awareness of a network and is at the core just the implementation of a remote procedure call.

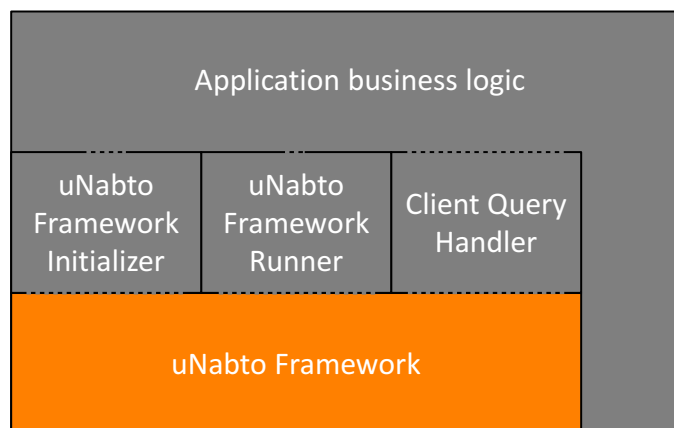
uNabto Device Introduction

This makes it very simple to make network enabled applications and port them to other uNabto enabled devices. Another aspect that makes it simple to develop applications, is that the uNabto API is very small, consisting of just a handful of generic functions. The uNabto SDK contains all the code for the *uNabto Framework*.

The *uNabto Platform Adapter* is a small component that abstracts the *Native Platforms* network and time functionality. This allows for a very simple port of the *uNabto Framework* to a given native platform. The uNabto SDK contains platform adapters for various hardware platforms, and for the Microsoft Windows and Linux operating systems. Details on how to create a *uNabto Platform Adapter* for a new platform is provided in section "Porting uNabto" below.

The uNabto Device Application will act as a pure data content provider for the remote client application and will in many simple designs simply be a passive part, just waiting for requests from the client. In a simple design the application specific code is just a few code lines and thereby enabling rapid application development.

The *uNabto Device Application* can be subdivided into the following components:



The *Application business logic* is the actual customer application running on the device, e.g. an existing application extended with remote access capabilities through Nabto.

The *uNabto Framework Initializer* component initializes the *uNabto Framework* and configures the uNabto device application's unique identifier.

The *uNabto Framework Runner* component is responsible for propelling the *uNabto Framework*. This and the initialization component are typical very small and in general very alike from application to application.

The *Client Query Handler* component is where all queries from the client are processed, typically by interacting with the application's business logic to create a response. The *Client Query Handler* component is in general unique from application to application.

– The dotted lines illustrate the level of interaction between the various components in a typical application.

uNabto Device Introduction

The following chapter will describe how to create each of the *uNabto Framework initializer*, *uNabto Framework Runner* and the *Client Query Handler* components.

5 Application components

This chapter will give an introduction to uNabto Framework compile time configuration and describe the three required sub components described in the previous section..

5.1 uNabto framework compile time configuration

The uNabto framework has a vast number of configuration parameters thereby enabling the application developer detailed compile time memory resource usage trimming and feature selection. The many configuration parameters and their interdependence are defined in `unabto_config_defaults.h` (https://github.com/nabto/unabto/blob/master/src/unabto/unabto_config_defaults.h), the most important are described in this document.

5.1.1 unabto_config.h

`unabto_config.h` is a mandatory and application developer created placeholder for all uNabto application specific configuration parameters. This file must be placed in the compiler's include search path. It is also a possibility to let some or all the parameters be defined as compiler defined options, but in any case the file must exist.

All the configuration parameters are configured by macro definitions in the configuration file. E.g.:

```
#define NABTO_RESPONSE_MAX_SIZE 1300
```

A minimal and in general compilable configuration file, for a platform without hard RAM resource limits, could look like this:

```
#ifndef _UNABTO_CONFIG_H_
#define _UNABTO_CONFIG_H_

#define NABTO_RESPONSE_MAX_SIZE 1300
#define NABTO_RESPONSE_MAX_SIZE 1300
#define NABTO_SET_TIME_FROM_ALIVE 0
#define NABTO_ENABLE_UCRYPTO 0
#define NABTO_ENABLE_LOGGING 0

#endif /* _UNABTO_CONFIG_H_ */
```

5.2 uNabto initialization

The uNabto framework has to be initialized explicitly by calling two functions in sequence; the first function initializes the uNabto framework context:

```
nabto_main_setup* unabto_init_context(void)
```

Application components

The returned `nabto_main_setup` structure is used to runtime configure the uNabto framework. The following fields in this structure are the most relevant for the application developer:

Name	Type	Default	Description
<code>id</code>	<code>const char*</code>	none	This must be set to a string containing the unique device id. The id must DNS resolve to the basestation. E.g.: 1234.mycompany.com
<code>url</code>	<code>const char*</code>	NULL	Deprecated: This may optionally be set to a string directing the client to an alternative public location for the now deprecated HTML device driver bundle.
<code>presharedKey</code>	<code>uint8_t[16]</code>	zero	The uNabto cryptographic engine uses symmetric-key cryptography. If <code>NABTO_ENABLE_UCRYPTO</code> is set and communication is to be secure, this 16 bytes array must be initialized with the secret shared key. If the field is left unchanged, the key will be the special zero cryptographic key. See <code>unabto_config_defaults.h</code> for details.

When the fields have been configured, the uNabto framework is initialized and configured with a call to:

```
bool unabto_init(void)
```

The function will return false if native platform's network layer fails initialization.

A usage example is shown in the next section.

5.3 uNabto Framework Runner

The uNabto framework is designed to run on native platforms without an operative system. On a platform without the concept of processes, the uNabto framework has to be actively propelled by the uNabto device application by calling this function:

```
void unabto_tick(void)
```

The function has to be called regularly to execute the internal interface polls and service functions. The call interval should not exceed 10ms to avoid retransmissions and a sluggish user experience.

Due to the simplicity, this method of propelling the uNabto framework is also often used on native platforms with a full operating system, like Unix.

A simple example for a main function in a uNabto device application with initialization:

Application components

```
int main()
{
    nabto_main_setup* nms;
    nms = unabto_init_context(void);
    /* replace <unique string> with a unique text string e.g.
       the Ethernet MAC address */
    nms->id = <unique string>".demo.u.nabto.net";
    if (!unabto_init(void))
    {
        /* handle error */
        ...
    }
    else
        for (;;)
        {
            unabto_tick();
            usleep(10); // sleep 10ms
        }
}
```

An application with just this main function and an empty client query request handler can be discovered by a Nabto client's discovery functionality on a local network. To supply data contents to the client, the functionality described in the next section must be implemented.

5.3.1 Enabling Cryptography

To enable cryptography for secure communication with the client and the basestation, enable the `NABTO_ENABLE_UCRYPTO` option in the `unabto_config.h` configuration file and add the following to the above initialization:

```
nms->id = "myname.demo.nab.to";
nms->secureAttach = true;
nms->secureData = true;
nms->cryptoSuite = CRYPT_W_AES_CBC_HMAC_SHA256;
if (gopt_arg(options, 'k', &preSharedKey)) {
    size_t i;
    size_t pskLen = strlen(preSharedKey);
    // read the pre shared key as a hexadecimal string.
    for (i = 0; i < pskLen/2 && i < 16; i++) {
        sscanf_s(preSharedKey+(2*i), "%02hx", &nms->presharedKey[i]);
    }
}
if (!unabto_init(void))
{
    ...
}
```

The additional lines enables cryptographic support, selects the AES+SHA256 suite and reads the preshared cryptographic key from somewhere (commandline in this example) into the `nms` struct.

Application components

The cryptographic key to use in the "*.demo.nab.to" domain can be obtained from the Nabto Developer Portal at <https://portal.nabto.com>. If you own your own basestation, you can configure a dummy key for development devices, typically consisting of just zeroes. Or create production cryptographic keys using tools available on the basestation.

5.4 The client query handler

When the uNabto framework receives a message with a query request from the client, the request is immediately handed over to the application's "Client query request handler". The handler will in general examine the query, execute the query relevant code and create a response. The client query request handler can be configured to operate in different modes, depending on how fast a query can be processed.

5.4.1 Synchronous and asynchronous operation modes

The client query request handler can operate in one of these non-blocking modes:

- **Synchronous**
The handler will return with a completed query response. This is the default mode and also the simplest to implement.
- **Asynchronous**
The handler initiates the query processing and returns with just a query request accepted status code. There is in principle no upper time bound on how long the query processing may take, but the various timing issues on client side should be taking into consideration. These issues are typical timeouts in the HTML render components (AJAX request timeout) and a possible impatient user waiting. Three additional functions have to be implemented in this mode as explained later.
- **Synchronous and asynchronous**
Depending on the query, the handler will return with a completed query response. Or the handler may initiate the query processing and return with a query request accepted status code. The combined handler shares the general properties of the pure asynchronous handler, but also handles queries where a fast query response is possible.

In all three cases, and for event handlers in general, they should return fast for the reasons mentioned in the `unabto_tick` section.

The "Client query request handler" callback invoked by the uNabto framework is called `application_event` and is used in all operation modes:

```
application_event_result application_event(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
```

Parameters:

Application components

`applicationRequest` – Pointer to a structure with the following relevant fields:

`uint32_t queryId` – The query identifier defined for the request in the query model. The query model is explained in The query model section.

`const char* clientId` – The user email address given to client. This value can be used to enforce access restrictions for specific requests. The usage is explained in the Access Control section.

`bool isLocal` – true if the request came from a client on the same local network as the device.

`queryRequest` – Buffer containing the client query request parameters (see below).

`queryResponse` – Buffer to place the applications query response parameters (see below).

The application handler shall return one of the following status codes:

AER_REQ_RESPONSE_READY – The query response is written in the `queryResponse` and ready to be sent to the client.

AER_REQ_ACCEPTED – Used in asynchronous mode. The query request is accepted and will be processed by the application.

AER_REQ_BUSY – Used in asynchronous mode. Signals that the application can not handle the request now. The client will see the query request as accepted, but the uNabto framework will repeatedly call the event handler with the query request data until a different result code is returned.

AER_REQ_NO_ACCESS – Used to reject a query from the client. The client application will receive this as an `ACCESS_DENIED` error code. It is up to the client to take proper action.

AER_REQ_INV_QUERY_ID – Used if the `queryId` is unknown to the application. The client application will receive this as a `DEVICE_ERR_UNKNOWN_QUERY_ID` error code. It is up to the client to take proper action.

AER_REQ_NOT_READY, AER_REQ_OUT_OF_RESOURCES – Used if the application for some reason can not handle the query. The client application will receive these as a `MICROSERVER_BUSY` error code. It is up to the client to take proper action.

See [TEN025] Section 5.2.3 on Nabto Error Codes to see how these error codes are propagated to the client.

5.5 Synchronous query request handling

As stated previously, the synchronous client request handler can be implemented when the query processing can be accomplished within 10 ms.

To build an application with synchronous query handling, follow these steps:

Application components

1. Set the macro definition `NABTO_APPLICATION_EVENT_MODEL_ASYNC` to 0 in the application header file `unabto_config.h` – this is the default setting.
2. Implement application logic.
3. Let the application client query request handler process the query and return `AER_REQ_RESPONSE_READY` to tell uNabto that the query response was generated successful.

5.6 Processing the query

How the query requests and responses are defined and related is normally defined in a project specific query model on the client side, essentially defining the public remote interface of the device. This model must be meticulously obeyed by the uNabto device application, especially with respect to returning a correctly formatted response. The next section provides a brief introduction to the query model and will be followed by concrete examples on how it is handled in the uNabto device application.

5.6.1 The query model

The query model is defined in XML as specified by the schema: http://www.nabto.com/unabto/query_model.xsd

The query model will briefly be explained through an example model. This model will also be used in the code examples presented later. Only the `query` element sections are shown:

```
<query name="get_switch_state.json" id="1">
  <request>
    <parameter name="switch_id" type="uint8"/>
  </request>
  <response>
    <parameter name="switch_state" type="uint8"/>
  </response>
</query>
```

For the uNabto device application developer, the relevant parts of the query elements are the following:

- The *query id* which identifies the query on the uNabto device. In this example the `id` is 1.
- The *parameters* name their types and their mutual ordering in the request and response elements. In this example the query request from the client will contain a switch identifier of type unsigned 8 bit integer and the device is expected to return a response giving the state of the switch in an unsigned 8 bit integer.

5.6.2 The query parameter types

The following parameter types are allowed:

Type name	Description
int8, int16, int32	Signed integer with the bit size specified. Equivalent to the stdint types int8_t, int16_t, int32_t
uint8, uint16, uint32	Unsigned integer with the bit size specified. Equivalent to the stdint types uint8_t, uint16_t and uint32_t
raw	A variable length binary string. The maximum length is limited by the uNabto framework configuration settings. On the device this type is represented by a list of uint8.

The client in our example may also set a descriptive string for the “switches” using a raw:

```
<query name="set_switch_description.json" id="2">
  <request>
    <parameter name="switch_id" type="uint8"/>
    <parameter name="switch_description" type="raw"/>
  </request>
  <response>
    <parameter name="status" type="uint8"/>
  </response>
</query>
```

In this example, the device is supposed to save the switch description for later retrieval.

The previous examples had a static number of request and response parameters in the received requests and in the returned responses. The list element opens up for more dynamical queries.

5.6.3 The query list element

A list can (in principle) consist of an unlimited number of parameters and even other lists. In practice the list length is limited by the uNabto framework configuration.

In this example, the client can send a list of multiple switch ids and the device can respond with a list with multiple switch states and descriptions. Remember that the lists need to be named just like the parameters:

Application components

```
<query name="get_multiple_switch_info.json" id="3">
  <request>
    <list name="switch_req_list">
      <parameter name="switch_id" type="uint8"/>
    </list>
  </request>
  <response>
    <list name="switch_resp_list">
      <parameter name="switch_state" type="uint8"/>
      <parameter name="switch_description" type="raw"/>
    </list>
  </response>
</query>
```

5.6.4 Working with the query model in the application

The uNabto framework passes two buffers to the client query request handler. The first buffer, the `queryRequest`, contains the query parameters in the order and with the type specified in the query request element in the model. The second buffer, the `queryResponse`, is for returning the query response parameters in the correct order and with the types specified by the query response element in the model.

The uNabto SDK has a utility function library to help in reading and writing the query parameters. The parameters are read and respectively written sequentially with these functions, and the functions will keep track of read/write positions and report under/overflow. It is highly recommended to use these function to ensure compatibility with the client and future updates to the uNabto SDK.

The usage of some of these functions will be demonstrated in the following examples. A detailed description of all the functions can be found in the library header file: `unabto/src/unabto/unabto_query_rw.h`

5.6.5 Integral types

The uNabto SDK supports all the signed and unsigned integers in the query model.

The utility functions reading integers query request parameters are:

- `unabto_query_read_int8`
- `unabto_query_read_uint8`
- `unabto_query_read_int16`
- `unabto_query_read_uint16`
- `unabto_query_read_int32`
- `unabto_query_read_uint32`

Application components

The utility functions writing integers query response parameters are:

- `unabto_query_write_int8`
- `unabto_query_write_uint8`
- `unabto_query_write_int16`
- `unabto_query_write_uint16`
- `unabto_query_write_int32`
- `unabto_query_write_uint32`

Since the semantics is very similar, only the `int8` variant will be described briefly here:

```
bool unabto_query_read_int8(unabto_query_request *queryRequest, int8_t *num)
```

Copies an 8 bit integer from the current read position in the query request buffer, to a position given by the supplied integer pointer.

E.g.:

```
int8_t number;
if (unabto_query_read_int8(queryRequest, &number))
{
    printf("Hello my number is %d", number);
}
```

If the value for some reason is unwanted, the passing of a NULL pointer as integer pointer argument will skip the parameter.

Writing is similar simple using the `int8` write function:

```
bool unabto_query_write_int8(unabto_query_response *queryResponse, int8_t
num)
```

Copies an 8 bit integer into the query response buffer at the current write position.

E.g.:

```
if (!unabto_query_write_int8(queryResponse, 42))
    ... /* error handling */
```

Adding the pieces together, a simple synchronous event handler for the first query in our example model (without error handling) could look like this:

Application components

```

application_event_result application_event(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
{
    application_event_result result = AER_REQ_NOT_READY;
    switch (applicationRequest->queryId)
    {
        case 1 :
        {
            uint8_t switchId, switchtState;
            unabto_query_read_uint8(queryRequest, &switchId);
            // Call some application function to get the state
            switchState = getSwitchState(switchtId);
            unabto_query_write_uint8(queryResponse, switchState);
            result = AER_REQ_RESPONSE_READY;
            break;
        }
        case 2 :
            ...
    }
    return result;
}

```

5.6.6 The raw type

On the device, the Raw type from the query model is handled as a list of `uint8_t` elements (lists are described in detail below). The maximum length depends on the value of the `NABTO_RESPONSE_MAX_SIZE` and request ditto size configuration macros – see the section on calculating the query request/response size.

Passing floating point numbers as text strings via a list of `uint8_t` is the recommended platform independent way to do so.

If strings are received from the client and perceived as so on the device, it is up to the application to convert the list to a C sting. If the raw is not going to be saved prior to generation of the response, a simple trick is to waste a little bandwidth and let the client add the terminating zero to the raw, thereby avoiding allocation of space for the the `uin8_t` list and the terminating zero.

For reading a `uint8_t` list from the query request buffer use the function:

```

bool unabto_query_read_uint8_list(unabto_query_request *queryRequest,
    uint8_t **listData, uint16_t *listLength)

```

The function does not copy the “listData”, but just passes a pointer to them. If the “list data” is not wanted, it can just be skipped by passing a NULL pointer. The length is returned in the `listLength` parameter.

E.g.:

Application components

```
uint8_t *listData;
uint16_t listLength;
if (unabto_query_read_uint8_list(queryRequest, &listData, &listLength))
{
    ... /* do something with data */
}
```

Example with parameter skip:

```
unabto_query_read_uint8_list(queryRequest, NULL, NULL);
```

The function for writing a uint8_t list works similar to the integral writing functions:

```
bool unabto_query_write_uint8_list(unabto_query_response *queryResponse,
    uint8_t *listData, uint16_t listLength)
```

The “list data” given by listData and with a length of listLength is written at the current position in the query response buffer.

E.g.:

```
char *data = "The red road rabbit rode rattlesnakes risking retribution";
uint16_t len;
if (!unabto_query_write_uint8_list(queryResponse, (uint8_t*)data,
    sizeof(data)-1))
{
    ... /* error handling */
}
```

A list of uint8 requires at least two bytes for the length, plus the space for the number of bytes in the raw.

5.6.7 Lists

By using lists, the query request/response structure becomes more dynamic. The equivalent of the list in C is a variable length array of structs. A list element can be any sequence of the other parameter types and other lists. The list length is in practice restricted by the request/response buffer size configuration parameters and the MTU of the packet carrier.

In contrary to the other query buffer read/write functions, the functions for list handling are made asymmetric. This was required since lists can be nested and the length should be allowed to be unknown until all list elements have been written.

Reading a list is straight forward:

1. Start by reading the number of elements
2. For each element use the other query processing functions to process it.

The list length is read by the following function:

```
bool unabto_query_read_list_length(unabto_query_request *queryRequest,
    uint16_t *list_length)
```

Application components

E.g.:

```
uint16_t len;
if (unabto_query_read_list_length(queryRequest, &len))
    printf("The length of the list is: %d\n", len);
```

A list in a query response is generated by following these steps:

1. Initialize a list context and save it for the finalization of the list.
2. Write each element by using the other query response writing functions.
3. Finalize the list.

For initializing a list, the following function is used:

```
bool unabto_query_write_list_start(unabto_query_response *queryResponse,
                                   unabto_list_ctx_t *list_ctx)
```

The returned `list_ctx` are saved for the finalization of the list and passed to the following function after all list elements have been written:

```
bool unabto_query_write_list_end(unabto_query_response *queryResponse
                                 unabto_list_ctx_t *list_ctx,
                                 uint16_t list_length)
```

The number of elements written is passed to the `list_length` parameter.

Simple list example from the example model without error handling:

```

...
case 3 :
{
    uint16_t i, listLength;
    if (unabto_query_read_list_lenght(&listLength))
    {
        uint8_t ids[listLength]; // Unsafe and requires C99!
        unabto_list_ctx_t listCtx;
        for (i = 0; i < listLength; i++)
            unabto_query_read_uint8(queryRequest, &ids[i]);

        unabto_query_write_list_start(queryResponse, &listCtx);
        for (i=0; i < listLength; i++)
        {
            char *desc;
            uint8_t switchtState;
            switchState = getSwitchState(ids[i]);
            desc = getSwitchDescription(ids[i]);
            unabto_query_write_uint8(queryResponse, switchStatus);
            unabto_query_write_uint8_list(queryResponse, (uint8_t*)desc,
                strlen(desc));
        }
        unabto_query_write_list_end(queryResponse, &listCtx, listLength);
        result = AER_REQ_RESPONSE_READY;
    }
    break;
}
...

```

A List requires at least two bytes for the length, plus the space required for all the list elements.

The AppMyProduct Heat Pump stub¹ contains an example of list handling, see the functions `copy_string` and `write_string` for reading and writing lists, respectively in `unabto_application.c`.

5.7 Asynchronous request handling

When the query processing is expecting to last for more than ten milliseconds, the application event handler must operate in asynchronous mode.

Operating in this mode, the application event handler returns with an `AER_REQ_ACCEPTED`, where after the uNabto framework repeatedly will call the application implemented function: `application_poll_query`. When this function returns true, the uNabto framework will retrieve the result by calling the application implemented function `application_poll`.

¹ <https://github.com/nabto/appmyproduct-device-stub>

Application components

The uNabto Framework may drop the request for various reasons by calling the application implemented function `application_poll_drop`.

To build an application with asynchronous query handling, follow these steps:

1. Set the macro definition `NABTO_APPLICATION_EVENT_MODEL_ASYNC` to 1 in the application header file `unabto_config.h`.
2. Implement application logic.
3. Let the application event handler initiate processing and return `AER_REQ_ACCEPTED` to tell the client that the query processing has started.
4. Implement the following three handlers:
 - a. `application_poll_query`
 - b. `application_poll`
 - c. `application_poll_drop`

Note! Multiple client queries can be active concurrently. For this reason, the application event handler must hold the `applicationRequest` parameter, passed on invocation, as a reference to the request and pass it on to the application poll query function.

In the following code examples, four application functions, for retrieving the switch state from a slow device, will be referenced: `sendSwitchStateRead`, `receiveSwitchStateReading`, `isSwitchStateReadingReceived`, `dropSwitchStateProcessing`; for respectively sending, receiving, probing and dropping the IO request.

An asynchronous query request handler for the first query in the example query model may look like this:

Application components

```

/* Only a single request is allowed to execute in these examples */
static application_request* currentApplicationRequest=NULL;

application_event_result application_event(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
{
    application_event_result result = AER_REQ_NO_QUERY_ID;

    if (NULL != currentApplicationRequest)
        return AER_REQ_BUSY; /* EXIT */

    switch (applicationRequest->queryId)
    {
        case 1 :
        {
            uint8_t switchId;
            unabto_query_read_uint8(queryRequest, &switchId);
            sendSwitchStateRead(switchId);

            /* Block for other requests while executing this */
            currentApplicationRequest = applicationRequest;

            result = AER_REQ_ACCEPTED;
            break;
        }
        case 2 :
            ...
    }
    return result;
}

```

The two things to really differentiate this handler from the previous example with the synchronous handler, is the absence of query response buffer write statements and the handler's return value `AER_REQ_ACCEPTED`.

The uNabto framework will regularly probe the query response processing progress by calling the application implemented function:

```
bool application_poll_query(application_request** applicationRequest)
```

Parameters:

applicationRequest – Set this to the applicationRequest obtained from the application_event handler for which the application wants to convey progress status.

The application poll query handler returns true if the query response is ready to be delivered to the client or false otherwise.

Asynchronous example continued:

Application components

```
bool application_poll_query(application_request** applicationRequest)
{
    /* Tell the uNabto framework for which request the status is returned.
       Here we only got one possibility - the last pending.*/
    *applicationRequest = currentApplicationRequest;
    return isSwitchStateReadingReceived();
}
```

When the `application_poll_query` returns true, the uNabto framework will retrieve the query response by calling the application implemented function:

```
application_event_result application_poll(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
```

Parameters:

`applicationRequest` – The `applicationRequest` returned by the application `application_poll_query` function.

`queryRequest` – Buffer containing the client query request parameters. The usage is similar to the `application_event` handler.

`queryResponse` – Buffer to place the applications query response parameters. The usage is similar to the `application_event` handler.

The application poll handler returns the same type of status code as the `application_event` handler does. In normal cases this will be the `AER_REQ_RESPONSE_READY` return code.

Asynchronous example continued:

```
application_event_result application_poll(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
{
    uint8_t switchtState;
    /* here we assume that:
       applicationRequest == currentApplicationRequest
    */

    switchtState = receiveSwitchStateReading();
    unabto_query_write_uint8(queryResponse, switchStatus);
    /* Allow new requests to be processed*/
    currentApplicationRequest = NULL;
    return AER_REQ_RESPONSE_READY;
}
```

Whenever an error occurs within the uNabto framework during processing of a request in the application, the uNabto framework will call the following application implemented function to drop further actions and let the application clean up any internal state:

Application components

```
void application_poll_drop(application_request* applicationRequest)
```

Parameters:

`applicationRequest` – The `applicationRequest` that is dropped.

A the typical reason for this function to be called, is the loss of connection to the client.

Asynchronous example continued:

```
void application_poll_drop(application_request* applicationRequest)
{
    /* Call an application function to drop the switch state request
       and clean up */
    dropSwitchStateProcessing();
    /* Prepare for a new request */
    currentApplicationRequest = NULL;
}
```

5.8 General application development notes

There are three important issues to observe before starting on the application development:

1. For memory space efficiency the `queryRequest` and `queryResponse` shares the same memory location. It is therefore in general a good idea to process all query request parameters before writing the query response parameters.
2. All parameters specified by the query model's query response section must be written in the response. Likewise the client is required to write all the parameters specified by the query model's query request section.
3. All parameters in the largest query response must fit with in the buffer space defined by the `NABTO_RESPONSE_MAX_SIZE` macro. The parameters are packed in the various buffers, and by using the space requirement given for each parameter type, the space requirement is simple to calculate.

5.9 Query request/response size platform configuration

Both a query request and a query response message must be contained within a single transfer unit on the data carrier layer i.e. the MTU for the data carrier must be respected. Furthermore the various headers for transport, control and encryption will require approx. 100 bytes.

For most devices connected to an Ethernet, a useful query request/response payload size is about 1310 bytes.

Calculating the size of a query request/response is quite simple since all query parameters are packed and the size of each parameter size is easily found.

Streaming

The **integral** parameter type requires the space indicated by the parameter type suffix. E.g. a uint32 parameter requires four bytes. Four uint32 requires each 32 bits giving a total space requirement of 128 bits i.e. 16 bytes.

The **raw** parameter type requires at least two bytes (the length) plus the size of the raw data.

The **list** parameter type requires at least two bytes (the element count) plus the size of each list element.

E.g.:

Given a list element with two uint32, a list with ten elements will require:

$$2 + 10 * (4 + 4) = 82 \text{ bytes.}$$

With a `NABTO_REQUEST_MAX_SIZE` set to 1310 bytes, the client can send:

$$(1310 - 2) / (4 + 4) \approx 163 \text{ elements of this type.}$$

When the calculations for the requests and responses and has been completed do the following:

- Set the `NABTO_REQUEST_MAX_SIZE` macro to the calculated maximum query request size.
- Set the `NABTO_RESPONSE_MAX_SIZE` macro to the calculated maximum query response size.

6 Streaming

The uNabto streaming implementation can be used to create a reliable sequential stream of bytes between a Nabto Client application and a uNabto device. The implementation mimicks TCP behavior so it is possible to think of uNabto streaming as TCP/Nabto instead of e.g. TCP/IP.

The Nabto Streaming implementation features selective acknowledge and congestion control which together gives the ability for high performance streaming over congested network connections. This could e.g. be HD video via an ADSL connection.

6.1 Stream Demo Application

We provide a stream echo demo application. It can be found in the uNabtio SDK under `unabto/demo/stream_echo`. This simple demo application can be used to get a quick overview of uNabto streaming. An echo client can be found in Nabto Toolbox which can be downloaded from nabto.com. The program `nterm.exe` in the .NET Nabto CLI Utils x can be used to connect to the echo server. `nterm` can be used with these parameters: `nterm.exe <echo server name> -s echo -e`, it will open a stream to the echo server with the name `<echo server name>`.

6.2 Streaming Usage

Streaming can be enabled and disabled in `unabto_config.h` by the configuration option `NABTO_ENABLE_STREAM`.

When a new stream is opened the framework will call the function `unabto_stream_accept` which has to be implemented by the customer application. After a stream has been accepted it is possible to read and write from it. The application owns the stream. Hence, the application is required to release the stream when finished using it.

Lifecycle of a uNabto stream:

- `unabto_stream_accept` is called and the stream is accepted.
- `unabto_stream_read`, `unabto_stream_ack` and `unabto_stream_write` are called multiple times.
- After reading and writing has ended, the stream is closed using `unabto_stream_close`.
- When the stream is closed `unabto_stream_release` is called, such that the stream resource is freed.

6.2.1 New streams

When the framework has a new stream ready for the application the function `unabto_stream_accept(unabto_stream* stream)` is called. This function shall be implemented by the application developer. The application is required to handle all streams even if they do not intent to read or write to them, it still has to close and release them. When a new stream is seen it can immediately be read from and written on.

6.2.2 Reading from a stream

The function `size_t unabto_stream_read(unabto_stream* stream, uint8_t** buf, unabto_stream_hint* hint)` is used to read from a stream. It returns the number of bytes available.

If it returns 0 the status hint can be read to get the reason why nothing is read. If the hint is `UNABTO_STREAM_HINT_OK` it simply means that there was no bytes to be read.

When bytes has been consumed the application should call `bool unabto_stream_ack(unabto_stream* stream, const uint8_t* buf, size_t used,`

Streaming

`unabto_stream_hint* hint`) to acknowledge towards the framework that the bytes has been consumed and they now can be freed.

It is not necessary to acknowledge all the bytes `unabto_stream_read` returns, but they have to be acknowledged sequentially. If not all bytes have been acknowledged, the rest of the bytes will be available the next time `unabto_stream_read` is called.

6.2.3 Writing to a stream

The function `size_t unabto_stream_write(unabto_stream* stream, uint8_t* buf, size_t size, unabto_stream_hint* hint)` writes data to a stream.

If 0 is returned `hint` tells why it was zero. If `hint` is 0 and it returns 0 it simply means that no more data can be written to the stream at the moment. Else it returns the number of bytes which was written to the stream.

6.2.4 Closing a stream

A stream is closed with the function `bool unabto_stream_close(unabto_stream* stream)`. If this function returns false it means that the stream has not yet been closed and you have to poll it later until it returns true. When `unabto_stream_close` returns true the stream can be safely released.

6.2.5 Releasing a stream

The last thing to do with a stream is releasing its resources. After `void unabto_stream_release(unabto_stream* stream)` has been called on a stream no further processing will be done for this stream.

6.2.6 Stream Events

Every time an event happens on a stream the function `void unabto_stream_event(unabto_stream* stream, unabto_stream_event_type event)` is called, this way the application can be notified when the state of a stream changes.

If you do not want this behavior the configuration option `NABTO_ENABLE_STREAM_EVENTS` can be set to 0 such that you do not have to implement a function to handle these events.

6.3 Stream Configuration

In `unabto_config_defaults.h` several streaming related configuration options are described. They can be used to tweak the performance and memory requirements for the streaming. Generally better performance and more concurrent streams requires more memory so for resource constrained devices it makes sense to use some time to tweak these options for the specific application.

To use the Streaming implementation, define `NABTO_ENABLE_MICRO_STREAM` to 1.

7 Access Control

Access control can be used to restrict access to a device in general or for specific queries. The identity of the client (authenticated by the basestation), is passed to the uNabto framework in the connection request – typically this is an email address. If the client is allowed by the uNabto device application to connect, the uNabto framework will pass the user identity in every succeeding call to the client query request handler, this allows for query level access restriction.

It is up to the application to store and manage identities (email addresses) and to impose and manage access rules. The uNabto SDK provides a utility module called ACL in the module directory to help managing the email addresses.

7.1 Connection level access control

When a client tries to connect to a device, the application has the ability to reject the connection request directly. The client will then receive an access error notification and any related resources in uNabto framework will be released.

To implement this first level of access control, follow these steps:

- Implement functionality to handle the allowed email addresses. Special queries may be implemented to handle email addresses storage/removal etc.
- Set the macro: `NABTO_ENABLE_CONNECTION_ESTABLISHMENT_ACL_CHECK` to 1 in `unabto_config.h` :
- Implement the client access event handler function: `allow_client_access`

The client access handler to implement by the application:

```
bool allow_client_access(nabto_connect* connection)
```

Parameters:

Access Control

connection – Structure where the following two fields are relevant for the topic:

char* **clientId** – The user email address given to client.

bool **isLocal** – True if the request came from a client on the same local network as the device. This enables restrictions based whether the client is “local” or not.

The handler should return true if the email address passed from the client are acceptable for the uNabto device application, false otherwise.

Simple connection access control example:

```
const char* secretId = "secret@mycompany.com";
bool allow_client_access(nabto_connect* connection)
{
    return 0 == strcmp(connection->clientId, secretId);
}
```

7.2 Query level access control

When the application event handler is invoked by the uNabto framework, the user email address, on the client side, is passed to the client query request handler. This allow for very fine grained access control on the device. To list up a few access restriction possibilities:

- Specific queries for specific users.
- Based on the supplied query parameters, specific queries for specific device resources etc.
- Specific queries for specific non local users.

To implement this level of access control, follow these basic steps:

- Implement functionality to handle the allowed email addresses. Special queries may be implemented to handle email addresses storage/removal etc.
- Examine the two fields: clientId and isLocal in the applicationRequest struct parameter passed to the client query request handler by the uNabto framework.
- Let the client query request handler return `AER_REQ_NO_ACCESS` if access restrictions are violated.

Simple query access control example:

```

static const char* unwantedId = "hacker@anonymous.org";

application_event_result application_event(
    application_request* applicationRequest,
    unabto_query_request* queryRequest,
    unabto_query_response* queryResponse)
{
    application_event_result result;

    switch (applicationRequest->queryId)
    {
        case 1 :
        {
            uint8_t switchId, switchtState;
            unabto_query_read_uint8(queryRequest, &switchId);
            /* We don't want everybody to know the state
               of the switch with id 42 */
            if (42 == switchId &&
                0 == strcmp(applicationRequest->clientId, unwantedId))
                result = AER_REQ_NO_ACCESS
            else
            {
                /* Call an application function to get the state */
                switchState = getSwitchState(switchtId);
                unabto_query_write_uint8(queryResponse, switchState);
                result = AER_REQ_RESPONSE_READY;
            }
            break;
        }
        case 2 : /* with query level access control */
        {
            uint16_t i, listLength;
            if (0 == strcmp(applicationRequest->clientId, unwantedId))
                result = AER_REQ_NO_ACCESS
            else
            {
                /* process the query ... */
                result = AER_REQ_RESPONSE_READY;
            }
        }
    }
    return result;
}

```

8 Special event handlers

In addition to the client query request and allow_client_access events the uNabto framework may be configured to notify the application for other event types.

8.1 Getting a UTC time stamp from the basestation

When the uNabto framework and the basestation exchanges awareness messages, the current time on the basestation is passed to the uNabto framework from the basestation. In a standard basestation setup, this message exchange occurs approximately every 10 seconds

The time stamp can be read by implementing the setTimeFromGSP event handler:

```
void setTimeFromGSP(uint32_t stamp);
```

Parameters:

stamp – UTC time stamp from basestation.

Due to basestation and network load the time will lag with respect to the time source on the basestation. Use other time sources if high time precision is required, e.g. NTP.

Note! By default the handler has to be implemented. If the functionality is unwanted, the `NABTO_SET_TIME_FROM_ALIVE` macro must be defined to 0 in the `unabto_config.h` configuration file.

8.2 Monitoring the basestation connection status

The uNabto framework on the device will by default have a connection with the basestation, this allows for remote clients to connect to the device. If this feature is important, it is possible for the application to monitor the connection status, and for example signal the status via a LED, by implementing the function:

```
void unabto_attach_state_changed(nabto_state state);
```

Parameters:

state – The most interesting state for the application is `NABTO_AS_ATTACHED`. Every other state indicates that the device is not connected with the basestation.

Enabled by defining the macro: `NABTO_ENABLE_STATUS_CALLBACKS` to 1.

9 The log printing framework

The uNabto framework provides a fine grained log printing framework, where log printing can be controlled by compile time “severity” level configuration settings. The uNabto framework itself uses the log printing framework extensively for logging various status messages.

The log printing framework

Not all ports of uNabto supports log printing, to figure out whether log printing is supported or not, the macro `NABTO_LOG_BASIC_PRINT` definition could be examined. On simple platforms this macro is often defined with an empty replacement token.

Log printing is by default enabled by the application configurable macro `NABTO_ENABLE_LOGGING` definition.

The following subsections will give a brief overview of the capabilities – consult the document: “uNabto SDK – Configuring and compiling” for more details.

9.1 Printing

All the log commands have a suffix describing the “severity” level:

- `NABTO_LOG_FATAL`
- `NABTO_LOG_ERROR`
- `NABTO_LOG_WARN`
- `NABTO_LOG_INFO`
- `NABTO_LOG_DEBUG`
- `NABTO_LOG_TRACE`

The parameters to the log commands are compatible with the `stdio printf` function. To obtain simple multiple compiler support, by avoiding variadic macro definitions, the parameter list must be enclosed in parentheses.

For example:

```
NABTO_LOG_INFO(("The meaning of life is: %d", 42 ));
```

9.2 Controlling log printing by severity

The various severity logging statements can be enabled by defining the `NABTO_LOG_SEVERITY_FILTER` mask macro definition to be the or'ing of the severity macro definition for each log severity. The macro names for these are predefined with the `NABTO_LOG_SEVERITY_` prefix followed by the aforementioned suffixes.

For example the following definition will ensure that only the `NABTO_LOG_FATAL` and `NABTO_LOG_ERROR` statements are executed:

```
#define NABTO_LOG_SEVERITY_FILTER (NABTO_LOG_SEVERITY_FATAL \  
                                  | NABTO_LOG_SEVERITY_ERROR)
```

Example for info logging only:

```
#define NABTO_LOG_SEVERITY_FILTER NABTO_LOG_SEVERITY_INFO
```

uNabto helper modules

Logging may also be specified by setting to a “severity” level. The logging severity level macros are defined likewise the other macros, beginning with a NABTO_LOG_SEVERITY_LEVEL_ prefix followed by the aforementioned suffixes.

E.g.:

```
NABTO_LOG_SEVERITY_LEVEL_WARN
```

Each level will include all the other levels in this increasing verbosity order: FATAL, ERROR, WARN, INFO, DEBUG, TRACE i.e. TRACE includes all the other levels.

Note! A short cut to log all is to define the symbol: NABTO_LOG_ALL

10 uNabto helper modules

Nabto provides a set of small application level modules to be used in uNabto device applications, described below. The modules can be found in the uNabto SDK folder: `/unabto/src/modules`

10.1 ACL

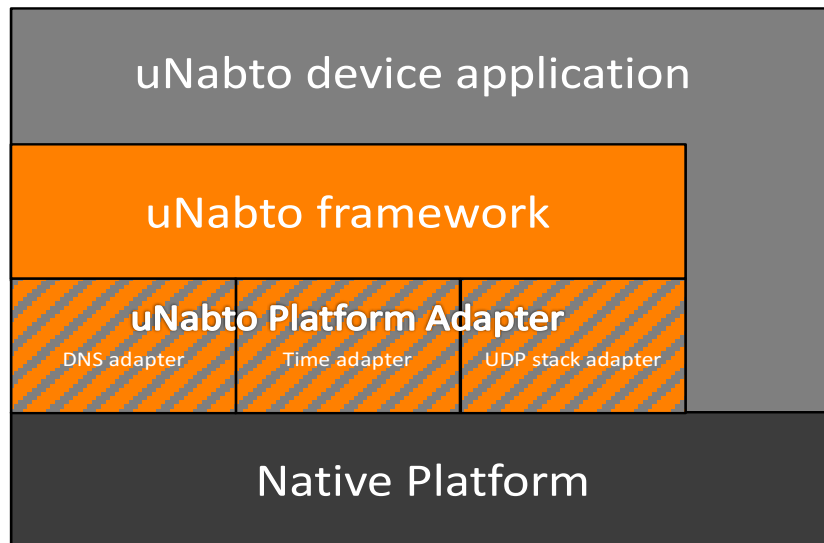
This module is primarily intended for maintaining an Access Control List. The module helps in the administration of a user list by providing the following functionality: add/remove/lookup a user, traversal of the user list and obtaining the number of users in the list. A 32 bit integer is associated with each user to aid in the creation of access right levels. The module uses the Configuration Store module for data persistence.

10.2 Configuration Store

This module provides a uniform interface to a persisted storage, where data like configuration data may be maintained. The module does not ensure transactional storage operation but offers data integrity validation. The module has abstraction for Microship PIC18 internal flash, POSIX compatible file IO and RAM (for testing). The ACL module depends on this module.

11 uNabto platform adapters

A *uNabto platform adapter* is made up of the three shaded components shown in this figure:



The three adapters create a uniform interface for the uNabto framework to the Native Platform: A DNS resolver, time functionality and a UDP network layer.

The uNabto SDK contains adapter source code for the components various native platforms:

Arduino, Microship PIC 18, Freescale MQX, Unix, Microsoft Windows.

These are a good starting point for new, customer adapters if porting to an unsupported platform.

11.1 Porting uNabto - Creating a new uNabto Platform Adapter

To explain how to port to a new platform, a new uNabto platform adapter will be created for the Linux platform, the code can be found in `unabto/demo/porting`.

A uNabto platform adapter is the code the uNabto Core uses for network communication, time handling, DNS resolving, logging and randomness. The standard uNabto SDK already provides modules for many platforms, located in the `modules` folder in the source tree.

All the functionality in this description is also available as uNabto modules and the implementation using modules can be found under the official Unix platform in the `platforms/unix` folder in the uNabto source.

11.1.1 Overall Structure

The overall structure of a uNabto platform adapter consists of 2 header files and several source files.

The two header files are `unabto_platform_types.h` and `unabto_platform.h`. The former header file describes all the types necessary for uNabto to run while the latter describes all the remaining platform dependencies, for example macro definitions.

A uNabto platform adapter is added to a uNabto project by adding the uNabto platform adapter to the compiler's include path, then uNabto will include the respective `unabto_platform.h` and `unabto_platform_types.h`.

Individual configuration macros is defined in `unabto_config_defaults.h`
(https://github.com/nabto/unabto/blob/master/src/unabto/unabto_config_defaults.h).

11.1.2 Basic code

The first step is to create a main program which can run uNabto with the platform adapter being developed. And secondly a uNabto configuration file which sets the outline for the implementation.

The main file below is just a simple runner which sets up uNabto and calls the tick function.

`unabto/demo/porting/src/main.c`:

uNabto platform adapters

```
#include <unabto/unabto_common_main.h>
#include <unabto/unabto_app.h>
int main() {
    nabto_main_setup* nms = unabto_init_context();
    nms->id = "myid.example.net";
    unabto_init();
    while(true) {
        unabto_tick();
    }
}

application_event_result application_event(application_request* request, buffer_read_t*
read_buffer, buffer_write_t* write_buffer) {
    return AER_REQ_INV_QUERY_ID;
}
```

The uNabto configuration file looks as follows (unabto/demo/porting/src/unabto_config.h):

```
#define UNABTO_PLATFORM_CUSTOM 1
#define NABTO_SET_TIME_FROM_ALIVE 0
```

11.1.3 Implementing unabto_platform_types.h

Next the unabto_platform_types.h file is created, it should define all necessary types for uNabto. That is, at least timestamps, intX_t, uintX_t, booleans and sockets (unabto/demo/porting/src/unabto_platform_types.h):

```
#include <stdint.h>
#include <stdbool.h>
#include <time.h>

typedef uint64_t nabto_stamp_t;
typedef int64_t nabto_stamp_diff_t;
typedef int nabto_socket_t;
```

uNabto platform adapters

11.1.4 Implementing unabto_platform.h

With all the basic types defined, the unabto_platform.h must be implemented. This can be used to implement all the adhoc functions which are not necessarily specified as a link time dependency in unabto_external_environment.h.

```
#include "unabto_platform_types.h"
#include <platforms/unabto_common_types.h>

#define NABTO_INVALID_SOCKET -1
#define nabtoMsec2Stamp(stamp) (stamp)
#define NABTO_LOG_BASIC_PRINT(severity, message)
```

11.1.5 Implementing a Network Adapter

The network adapter implements the functions which are necessary for creating and using communication sockets. This is init, close, read and write functionality for network data. A simple unix implementation is seen below (greatly simplified - initialization and error checking omitted from here (see full source in unabto/demo/src/porting/network_adapter.c)).

```

bool nabto_init_socket(uint32_t localAddr, uint16_t* localPort, nabto_socket_t* sock) {
    nabto_socket_t sd;
    *sock = socket(AF_INET, SOCK_DGRAM, 0);
    *localPort = htons(sao.sin_port);
    return true;
}

void nabto_close_socket(nabto_socket_t* sock) {
    if (sock && *sock != NABTO_INVALID_SOCKET) {
        close(*sock);
        *sock = NABTO_INVALID_SOCKET;
    }
}

ssize_t nabto_read(nabto_socket_t sock,
                  uint8_t* buf,
                  size_t len,
                  uint32_t* addr,
                  uint16_t* port)
{
    int res;
    // ... (initialization omitted)
    res = recvfrom(sock, (char*) buf, (int)len, 0, (struct sockaddr*)&sa, &salen);
    // ... (skip error checking)
    return res;
}

ssize_t nabto_write(nabto_socket_t sock,
                   const uint8_t* buf,
                   size_t len,
                   uint32_t* addr,
                   uint16_t* port)
{
    int res;
    // ... (initialization omitted)
    res = sendto(sock, buf, (int)len, 0, (struct sockaddr*)&sa, sizeof(sa));
    // ... (skip error checking)
    return res;
}

```

11.1.6 Implementing a Time Adapter

The uNabto core has several timers which handle timed events, like reconnecting or retransmitting a packet. In order to facilitate this, some time functions must be available. The timing needed does not need to be absolute, it should just be monotonically increasing in some arbitrary time period.

unabto/demo/src/porting/time_adapter.c:


```
#include <unabto/unabto_external_environment.h>

nabto_stamp_t nabtoGetStamp() {
    struct timespec res;
    clock_gettime(CLOCK_MONOTONIC, &res);
    return res.tv_sec+(res.tv_nsec/1000000);
}

bool nabtoIsStampPassed(nabto_stamp_t* stamp) {
    nabto_stamp_t now = nabtoGetStamp();
    return *stamp <= now;
}

nabto_stamp_diff_t nabtoStampDiff(nabto_stamp_t * newest, nabto_stamp_t * oldest) {
    return newest - oldest;
}

int nabtoStampDiff2ms(nabto_stamp_diff_t diff) {
    return (int) diff;
}
```

11.1.7 Implementing a DNS Adapter

The DNS adapter must be able to do DNS resolving asynchronously as sketched below (the full source can be found in unabto/demo/src/porting/dns_adapter.c):

```
#include <unabto/unabto_external_environment.h>

void* resolver_thread(void* ctx) {
    resolver_state_t* state = (resolver_state_t*)ctx;
    struct hostent* he = gethostbyname(state->id);
    // ...
    resolver_is_running = false;
    return NULL;
}

void nabto_dns_resolve(const char* id) {
    uint32_t addr = inet_addr(id);
    if (addr != INADDR_NONE) {
        resolver_state.resolved_addr = htonl(addr);
        resolver_state.status = NABTO_DNS_OK;
    } else {
        // ...
        if (pthread_create(&thread, &attr, resolver_thread, &resolver_state) != 0) {
            return -1;
        }
        // ...
    }
}

nabto_dns_status_t nabto_dns_is_resolved(const char *id, uint32_t* v4addr) {
    if (resolver_is_running) {
        return NABTO_DNS_NOT_FINISHED;
    }

    if (resolver_state.status == NABTO_DNS_OK) {
        *v4addr = resolver_state.resolved_addr;
        return NABTO_DNS_OK;
    }

    return NABTO_DNS_ERROR;
}
```

The uNabto framework source code

11.1.8 Implementing a Random Adapter

```
#include <unabto/unabto_external_environment.h>
#include <openssl/rand.h>

void nabto_random(uint8_t* buf, size_t len) {
    if (!RAND_bytes((unsigned char*)buf, len)) {
        NABTO_LOG_FATAL(("RAND_bytes failed."));
    }
}
```

12 The uNabto framework source code

The uNabto SDK (the framework source, example platform adapters and application level modules) is written in C89 compliant C, making it straight forward to build with most compilers and giving a compact code size.

12.1 Where to download the source code

The uNabto SDK can be downloaded at <https://nabto.com/devices-download>.

12.2 The structure of the source code

The root directory of the uNabto SDK source tree in the uNabto SDK archive is named `unabto`. In every subdirectory below, the source and the associated header files are for the most part located in the same directory. The compiler include and source paths are therefore identical.

The `unabto` root directory includes the following directories of interest for the scope of this document:

12.2.1 The uNabto framework core source code and header file directory

`unabto/src/unabto`

This directory contains the uNabto framework and also the API header file for inclusion in application projects. Even though the source is open, it is highly recommended to refrain from making any change to the uNabto framework core source code. Support from Nabto will be difficult and future releases might break the patched code.

12.2.2 The uNabto platform adapter specific source directories

The source files for each of the native platforms adapters are for the most part placed in the directory:

`unabto/src/platforms`

The uNabto framework source code

But for some native platforms, the component source is placed in the modules directory – see the next section.

12.2.3 The feature module and platform adapter specific source directory

`unabto/src/modules`

This directory contains source code for feature extension modules, but also native platform specific adapter components. The feature extension modules and adapter components are used by Nabto in various projects and made public.

A short description some of the feature extension modules:

acl – module to help implementing device access control.

crypto – a required module if a cryptographic secured communication is wanted. - CHECK

random/dummy – If the cryptographic engine in uNabto is enabled, a function to create random strings is required. This module provides a dummy, but in a cryptographic sense useless, function.

A short description of some the platform adapter specific modules:

network/bsd – a uNabto Platform Adapter for the UNIX network interface.

network/winsock – a uNabto Platform Adapter for the Microsoft Windows network interface.

timers/unix – a uNabto Platform Adapter for the Unix time interface.

12.2.4 API include directories

All uNabto public header files are located in:

`unabto/src/unabto`

In an application it is sufficient to include the header file:

`unabto.h`

The targets uNabto framework Adapter contains a header file called `unabto_platform.h`, this file must in the compilers include search path.

12.3 Building the uNabto SDK with CMake

Nabto has provided a cmake script file to build the uNabto SDK. This file is located in the uNabto source package:

`unabto/build/cmake/unabto_files.cmake`

This script is to be included in an application CMake project file (see existing demos in the apps directory).

12.4 Device platform memory requirements

Configured to an absolute minimum the uNabto framework and uNabto platform adapter alone, i.e. without network stack etc., requires in general:

12.4.1 RAM usage

- 500 Bytes for buffers etc.
- 500 - 700 Bytes of stack memory.

12.4.2 ROM usage

This is very dependent on the target and the quality of the used compiler. From the experience gained by the porting of uNabto to various platforms a minimum of 10KB and up to 40KB (e.g. the Microchip PIC18) is in the expected ROM usage range for the uNabto framework alone. The network stack will use additional memory.

The uNabto request/response protocol can be implemented in less than 1 kB flash if hand-coding the implementation and bypassing the abstractions provided by the SDK.

12.5 Summary of the referenced configuration parameters

The following table summarize the configuration parameters mentioned in this document and also presents the minimum set a developer must know about to get the uNabto framework configured correctly.

Name	Default	Description
NABTO_RESPONSE_MAX_SIZE	200	Define to the maximum expected query response sent to the client.
NABTO_REQUEST_MAX_SIZE	200	Define to the maximum expected query request sent from the client.
NABTO_CLIENT_ID_MAX_SIZE	64	Maximum allowed length of the email passed in the credentials to the client. If very long email addresses are expected this must be adjusted accordingly, otherwise device access restriction policies will fail.
NABTO_APPLICATION_EVENT_MODEL_ASYNC	0	Set to 1 if the device may spend more than 50ms before returning the response to a client's request. Further details in section: Synchronous and asynchronous operation modes
NABTO_SET_TIME_FROM_ALIVE	1	Set 0 if the application won't need the current time in UTC from the basestation. If this option is set the application must implement the <code>setTimeFromGSP</code> function.
NABTO_ENABLE_UCRYPTO	1	Set to 0 if the communication between client and device does not require encrypted.
NABTO_ENABLE_LOGGING	1	Set to 0 if the log messages from the uNabto framework aren't wanted.