# RW-BLE Host Software

Functional Specification

RW-BLE-HOST-SW-FS

*Version 8.01b*

2015-03-10

RivieraWaves
http://www.rivierawaves.com/

# Revision History

| Version | Date | Revision Description | Author |
|---|---|---|---|
| 1.00 | 2011-12-21 | Update to 1.0 | KY/SLH/CI/SB |
| 1.01 | 2012-11-28 | Profile update, minor changes | KY |
| 1.02 | 2012-12-13 | Add more GAP information on privacy, security<br>GAP attributes, discovery | KY |
| 7.00 | 2014-06-30 | BLE 4.1 update | FBE |
| 7.01 | 2014-07-02 | Modification of Update Parameters Algo | FBE |
| 8.00b | 2015-01-23 | BLE 4.2 update | FBE |
| 8.01b | 2015-03-10 | Fix issue in L2CAP: SDU length not part of MTU | FBE |

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms and Abbreviations

| Acronym or abbreviation | Writing out in full / Meaning |
|---|---|
| AD_TYPE | Advertising Data Type (https://www.bluetooth.org/en-us/specification/assigned-numbers/generic-access-profile) |
| ATTC | Attribute Protocol Client toolbox. |
| ATTM | Attribute Manager and Generic Attribute toolbox. |
| ATTS | Attribute Protocol Server toolbox. |
| BLE | Bluetooth Low Energy. Also termed LE. |
| CSRK | Connection Signature Resolving Key |
| FS | Functional Specification |
| FW | Firmware |
| GAP | Generic Access Profile |
| GAPC | Generic Access Profile Controller |
| GAPM | Generic Access Profile Manager |
| GATT | Generic Attribute Profile |
| GATTC | Generic Attribute Profile Controller |
| GATTM | Generic Attribute Profile Manager |
| GTL | Generic Transport Layer |
| HCI | Host Controller Interface |
| IRK | Identity Resolving Key |
| L2CAP | Logical Link Control and Adaptation Protocol |
| L2CC | L2CAP Controller |
| L2CM | L2CAP Manager |
| LL | Lower Layer (Link Manager and Link Controller) |
| LTK | Long Term Key |
| MITM | Man-In-The-Middle Attack |
| MPS | Maximum Packet Size |
| MTU | Maximum Transmission Unit |
| OOB | Out Of Band |
| PDU | Protocol Data Unit |
| PRP | Proximity Profile |
| PSM | Protocol/Service Multiplexer |
| RAL | Resolving Address List |
| RO | Read Only |
| SC | Secure Connection |
| SDU | Service Data Unit |
| SIG | Special Interest Group |
| SM | Security Manager |

| | |
|---|---|
| SMP | Security Manager Protocol. BLE block responsible for security. |
| SMPC | SMP Controller toolbox. |
| SMPM | SMP Manager toolbox. |
| STK | Short Term Key |
| TBC | To Be Checked |
| UUID | Universally Unique Identifier |

# 1 Overview

## 1.1 Document Overview

This document describes the functionalities of the different blocks present in the RW-BLE software host stack.

## 1.2 RW BLE Host Overview

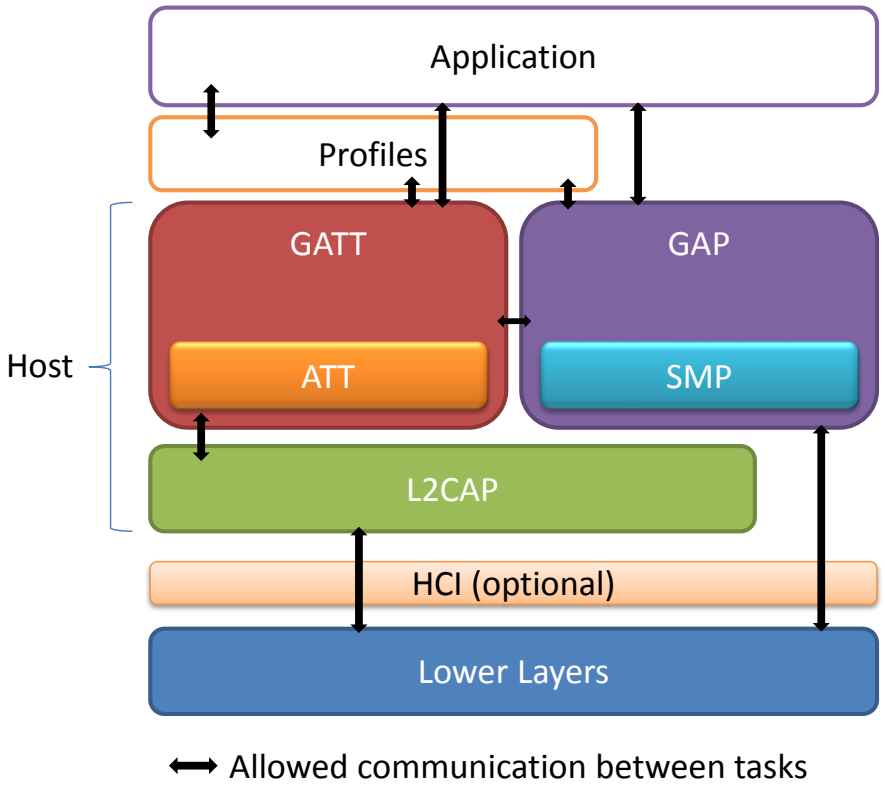



**Figure 1-1: RW BLE Host (Arrows shows allowed communication between tasks)**

## 1.3 Product Overview

The RW-BLE SW stack is available in three different partitioning types:

- ✓ **Split partition:** The link layer and the host run on two different CPUs, communicating together through the Host Controller Interface (HCI).
- ✓ **Fully-hosted partition**: The lower layers, higher layers, profiles and application run on the same CPU.
- ✓ **Fully-embedded partition**: The link layer and host protocols and profiles run on one CPU and the application runs in separate CPU, and these two components communicate via HCI.

### 1.3.1 Split Partition

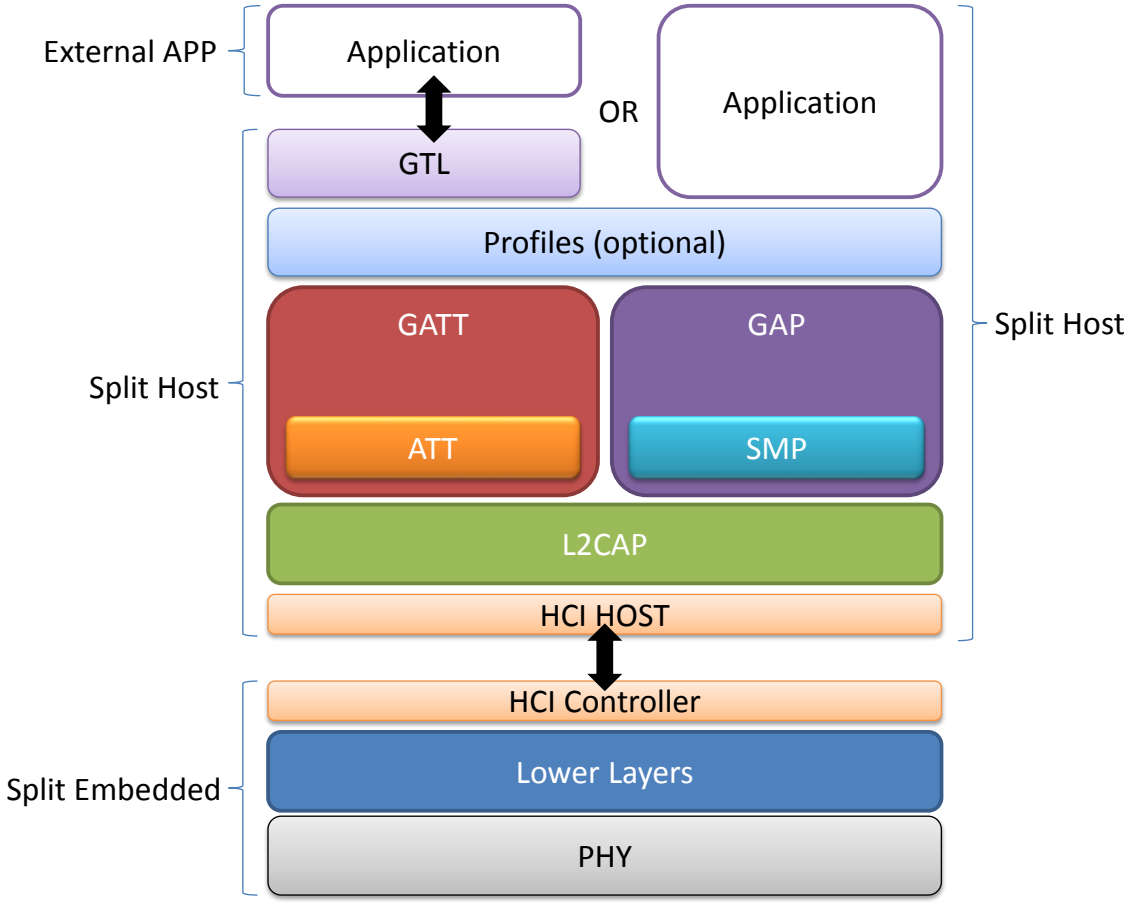



**Figure 1-2: Split Partitioning**

**Note**: Split Host can integrate application or provide (like full-embedded partitioning) an interface allowing an external application to use host stack through a transport layer.

### 1.3.2 Fully-Embedded Partition

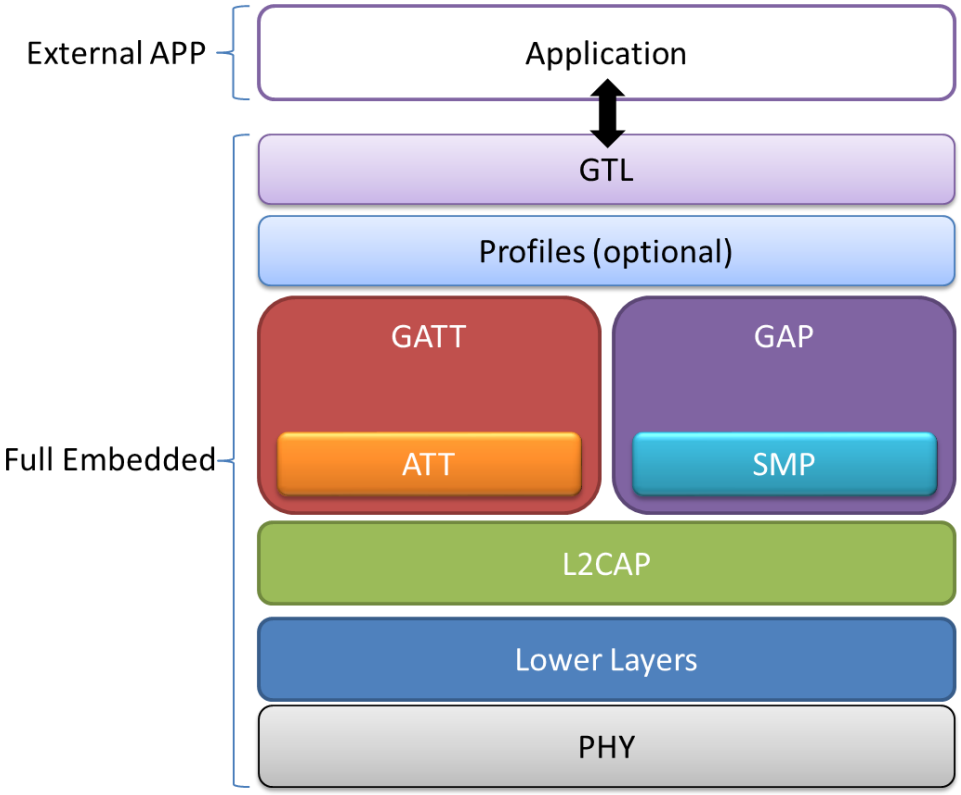



**Figure 1-3: Fully-embedded partitioning**

### 1.3.3 Fully-Hosted Partition

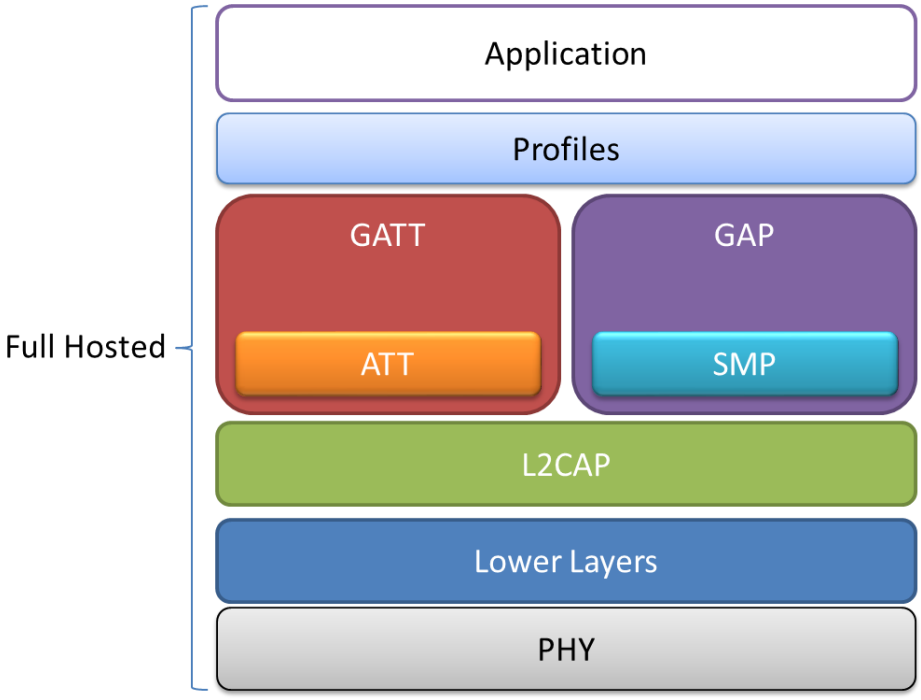



**Figure 1-4: Fully-hosted partitioning**

# 2 Host Controller Interface Functionalities (HCI)

See Host Controller Interface FS [4].

# 3 Logical Link Control and Adaptation Protocol Functionalities

L2CAP lies above the Link Layer (or HCI) and interfaces with higher layer protocols.

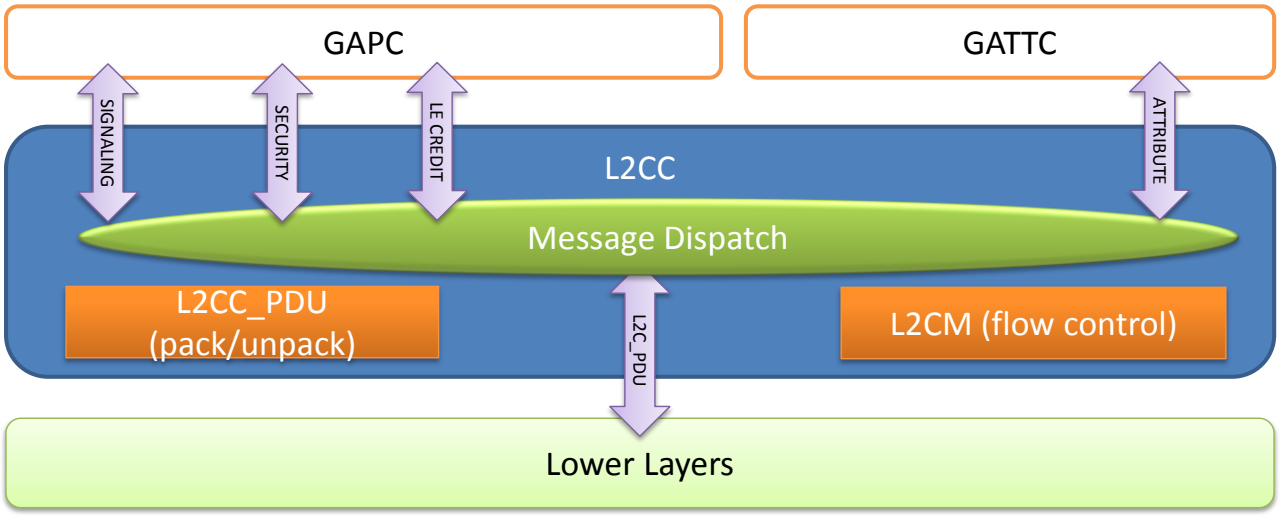



**Figure 3-1: L2CAP Architecture**

L2CAP in BLE operates only in Basic Mode and uses fixed channels plus dynamically allocated channels (see Table 3-11). In the fixed channel type, only BLE Signaling, Security Management protocol and Attribute protocol channels are used. Connection requests to device without utilizing allowed LE channels will cause an issuance of command reject response code with “0x0002 – Invalid CID in request” as reason of rejection.

In BLE, L2CAP Service Data Unit (**SDU**) is transmitted in one Protocol Data Unit (**PDU**).

The default Maximum Transmission Unit (**MTU**) size is set to 23 octets. This could be negotiated during the link for ATT and LE Credit Based data channels.

The maximum size of payload data (**MPS**) that can be accepted is 65535 octets. In our implementation it cannot exceed negotiated **MTU**.

**Note:** Software implementation supports a maximum MTU and so maximum MPS of 2048 bytes.

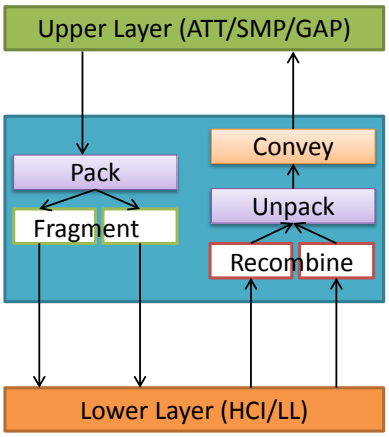



**Figure 3-2: L2CC Block Overview**

## 3.1 Packet Format

L2CAP is based around the concept of *channels*. Each endpoint of an L2CAP channel is referred to by a *channel identifier (CID).*

### 3.1.1 Basic Information Frame (B-Frame)

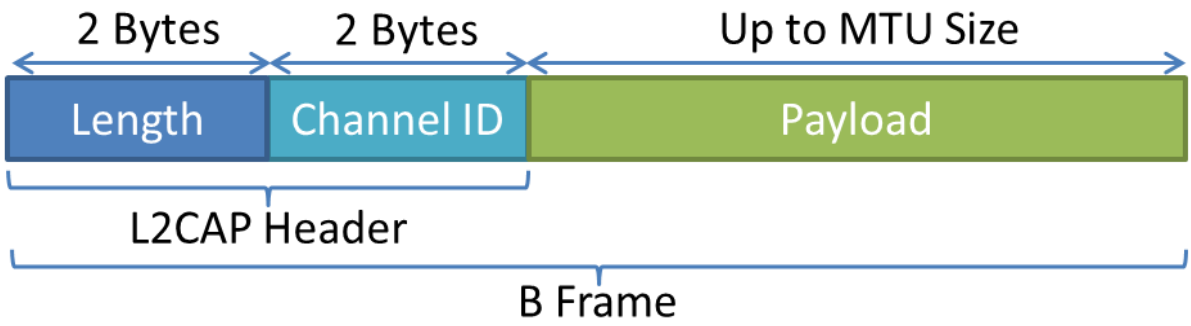



**Figure 3-3: L2CAP B-Frame**

#### 3.1.1.1 Length

Length indicates the size of the information payload in octets, without the L2CAP header size.

All L2CAP implementations supports the reception of B-frames with a payload length that does not exceed the negotiated **MTU**.

#### 3.1.1.2 Channel Identifier

The null identifier (0x0000) is an illegal identifier and is never used as a destination endpoint.

The use of fixed channel is mandatory for BLE L2CAP implementation. The table below shows the defined fixed channel identifiers for BLE. These channels are immediately available when a link layer logical transport is established between two entities.

| Channel ID | Channel type | Task Handler |
|---|---|---|
| 0x0004 | Attribute Protocol | **GATTC** (using Attribute toolbox) |
| 0x0005 | BLE Signaling Channel | **GAPC** |
| 0x0006 | Security Management Protocol | **GAPC** (using SMP toolbox) |
| 0x0040-0x007F | Dynamically allocated (LE Connected Data Channel) | **GAPC** (using Connection Oriented Channel toolbox) |

**Table 3-1: L2CAP Channel Identifiers**

#### 3.1.1.3 Information Payload

This contains the payload received from the upper layer protocol or delivered to the upper layer protocol.

## 3.2 Interfaces

### 3.2.1 Interface with upper layers

The L2CAP provides an API to the upper layers to allow them to transmit and/or receive data. The interface is based on RW kernel messages that are exchanged between the upper layer modules (GAP, GATT) and the L2CAP.

Message format that can be exchange with L2CAP interface are specified by L2CAP PDU module (see 3.2.3).

### 3.2.2 Interface with lower layers

In order to send/receive data to/from the lower layers, the L2CAP makes use of the Link Layer API. This API is based on the HCI messages defined in the Bluetooth standard, and implemented as RW kernel messages. The partitioning of the stack (split or fully-embedded) is totally transparent for the L2CAP, as Link Layer API messages are automatically converted to HCI commands and vice-versa in case of split partitioning.

### 3.2.3 PDU module

L2CAP PDU module is in charge of packet packing, unpacking, fragmentation and recombination.

This module provides structures describing packet supported by L2CAP.

To reduce code size of packer/unpacker, this module uses a kind of regular expression plus PDU structure format (closed to PDU format) in order to perform packing / unpacking.

**Packet description allowed field:**

- **B**: Byte value
- **W**: Word value
- **L**: Word value which contain length
- **c**: Next parameter can be used as a conditional value
- **w[cond]**: optional word value with following conditional expected value, not present condition not ok.
- **A**: 6 Bytes Bd Address value
- **K**: 16 Bytes Key value
- **R**: 8 Bytes random value
- **l[type]**: Length information about following array of specified type

**Examples**:

| Pkt_id | Len | ITV Min | ITV Max | Latency | Timeout |
|---|---|---|---|---|---|
| B | L | W | W | W | W |

**Table 3-2: Connection Parameter Update Request**

| sHdl | eHdl | AttType | AttVal |
|---|---|---|---|
| W | W | W | lB |

**Table 3-3: Find By Type Value Request**

## 3.3 Segmentation and Reassembly

L2CAP Manager is able to segment a Service Data Unit (SDU) to several PDUs. Segmentation is performed automatically by message handler used for sending packets, and only on Connection Oriented Channel Channels (see Chapter 5.5). When sending a SDU message with a size greater than negotiated MPS, PDU module already in charge of fragmentation will also cut packet into segment of MPS size.

**Note:** For ATT messages, MPS is equal to negotiated MTU, so ATT message are never segmented but can be fragmented.

**Note:** LE Signaling and Security Manager PDU size are less than 23 bytes if secure connection isn't supported, else less than 65 byes. Those PDUs are never segmented and SDU length field is not present.

**Note:** SDU size cannot exceed MTU and PDU size cannot exceed MPS.

When a PDU on a Connection Oriented Channel is received, the PDU is put at end of segment list. Then, a procedure is in charge of browsing list of segment, and if a full SDU is available into this list, the SDU is reassembled into an **l2cc_pdu_recv_ind** message and conveyed to destination task.

**Note:** If MTU or MPS size are exceed on the Connection Oriented Channel, then the channel connection is closed (disconnected) and an error response should be send with respectively invalid MTU or invalid Packet size error code.

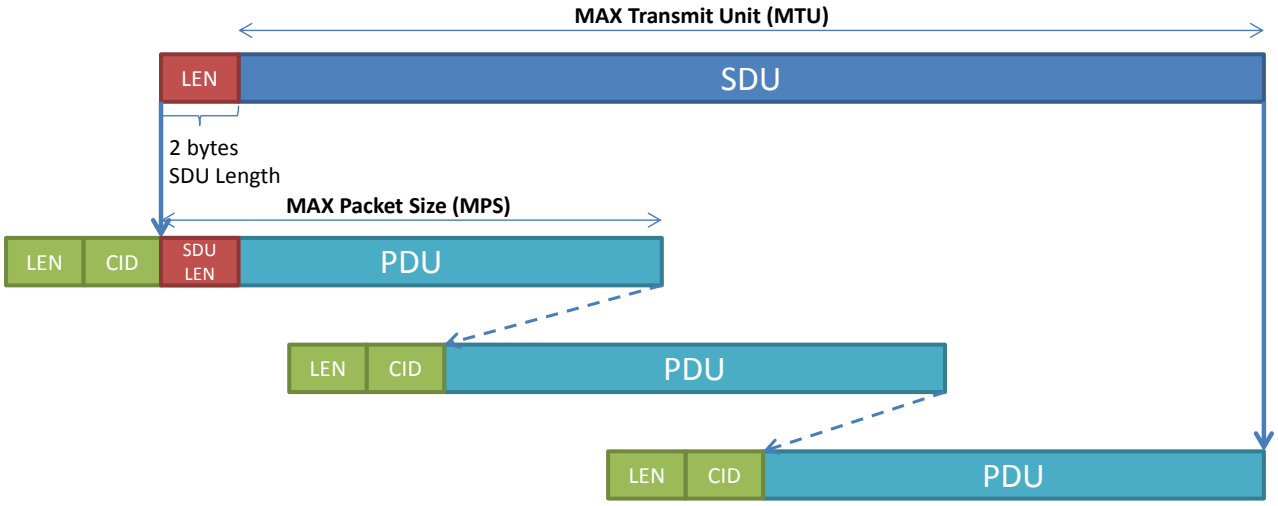



**Figure 3-4: L2CAP SDU Segmentation**

**Note:** When a Connection Oriented Channel is closed, corresponding packets into segment list are dropped.

## 3.4 Limitations

The following features are not supported in RW L2CAP BLE implementation:

- L2CAP PDU retransmission and flow control.

## 3.5 Environment Variables

| Type | Value | Comment |
|---|---|---|
| **l2cm_buf_mon** | buf_mon | L2CAP available buffers management |

**Table 3-4: L2CAP manager Environment variables**

| Type | Value | Comment |
|---|---|---|
| **l2cc_pdu_send_req*** | p_send_req | Pointer to the PDU which is currently sent (fragmented) |
| **l2cc_pdu_recv_ind*** | p_recv_ind | Pointer to the PDU which is currently received (fragmented) |
| **struct co_list** | segments | List of received PDU segment that should be reassembled |

**Table 3-5: L2CAP controller Environment variables**

# 4 Generic Attribute Profile

The Generic Attribute Profile (GATT) is the gateway in using the Attribute Protocol to discover, read, write and obtain indications of the attributes present in the server attribute, and to configure the broadcasting of attributes. The GATT lies above the Attribute Protocol and communicates with Generic Access Profile, higher layer profiles and application.

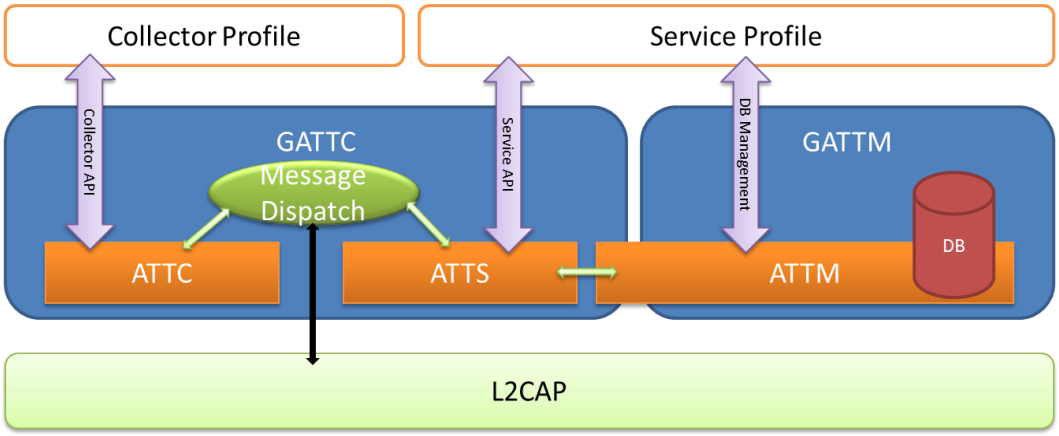



**Figure 4-1: GATT Architecture**

## 4.1 Fundamentals

### 4.1.1 Roles

GATT Client is the device that initiates commands and requests towards the GATT Server and can receive responses, indications and notifications from the GATT Server.

GATT Server is the device that accepts incoming commands and requests from GATT Client and sends responses, indications and notifications to GATT Client.

The roles are not fixed to the device, and the device's affiliation to the role is stopped as soon as the role-specific procedure ends. A device can act in both roles simultaneously.

### 4.1.2 Security Features

Encryption in GATT depends on the type of Physical link. On an LE Physical link, security features are optional, while it is the reverse on a BR/EDR Physical link.

### 4.1.3 Attribute Grouping

GATT defines grouping of Attribute to improve attribute discovery and access manipulation. The three groups are defined below.

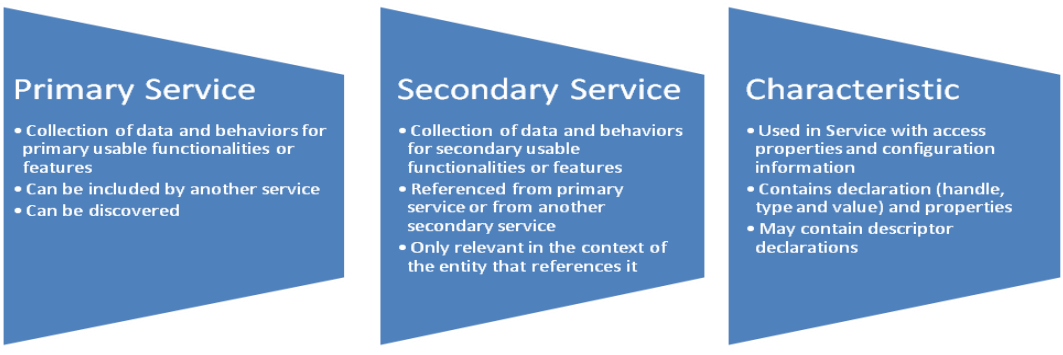



**Figure 4-2: ATT Grouping**

#### 4.1.3.1 Service

The Service definition contains a service declaration and may contain include and characteristic definitions. The Service declaration is an Attribute with the Attribute Type set to UUID for Primary or Secondary Service.

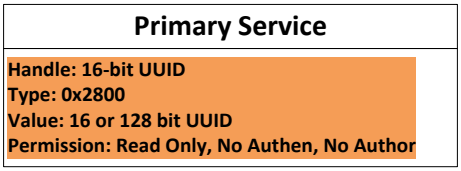



**Figure 4-3: Primary Service Declaration**

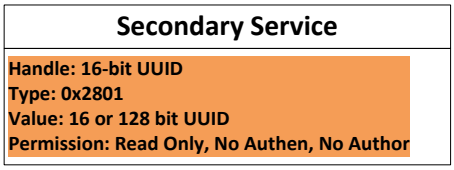



**Figure 4-4: Secondary Service Declaration**

When multiple services exist, definitions of the services must be grouped together, according to the Bluetooth UUID type (2, 4 or 16 octets).

#### 4.1.3.2 Included Service

Include definition contains only one include declaration.

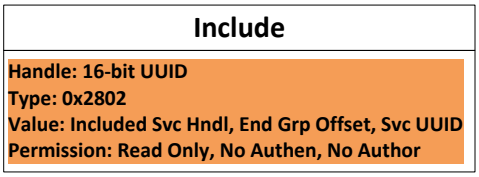



**Figure 4-5: Include Declaration**

Include declaration is an Attribute with Attribute type set to 0x2802 and the value set to Attribute handle, End group offset and UUID for the Service (2, 4 or 16 octets).

If the Attribute Client detects a circular reference or nested include declarations to a greater level than it expects, it should terminate the ATT Bearer.

#### 4.1.3.3 Characteristic

The Characteristic definition contains a characteristic declaration, Value and may contain characteristic descriptor declaration.

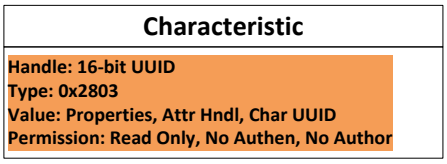



**Figure 4-6: Characteristic Declaration**

The Characteristic declaration is an Attribute with the Attribute UUID Type set to 0x2803, Attribute Value set to the Characteristic Properties, Value Attribute Handle and Value UUID (2, 4 or 16 octets).

#### 4.1.3.3.1 Characteristic Extended Properties (CEP)

Characteristic descriptors are used to contain related information about the Characteristic Value, identified by the Characteristic descriptor UUID. The access permissions are profile or implementation defined.

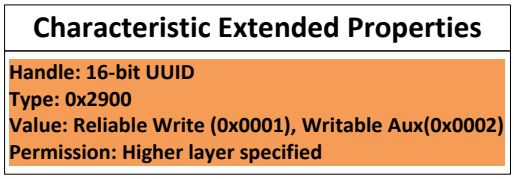



**Figure 4-7: characteristic Extended Properties Declaration**

The Characteristic Extended Properties declaration is a descriptor that gives more Characteristic information. The descriptor is an Attribute with type set to 0x2900, and the Attribute Value equal to a set Characteristic Extended Properties Bit field.

#### 4.1.3.3.2 Characteristic User Description

The Characteristic User Description declaration is an optional characteristic descriptor of UTF-8 string of variable sized textual description of the Characteristic Value.

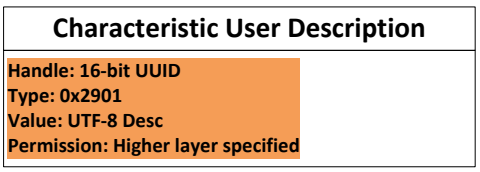



**Figure 4-8: Characteristic User Description Declaration**

The descriptor is an Attribute with type set to 0x2901, and the value set to user description UTF-8 format.

#### 4.1.3.3.3 Client Characteristic Configuration (CCC)

An Attribute Client may write a pre-configured descriptor to control the configuration of a characteristic on the server for the client.

The declaration of the Client Characteristic Configuration is readable and writable.

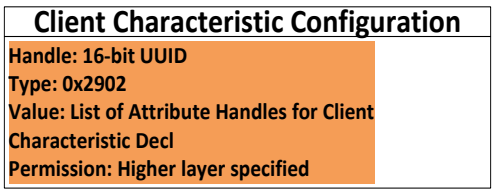



**Figure 4-9: Client Characteristic Configuration Declaration**

The descriptor is an Attribute with type set to 0x2902, and the value set to characteristic descriptor value.

#### 4.1.3.3.4 Server Characteristic Configuration (SCC)

An Attribute Client may write a pre-configured descriptor to control the configuration of a characteristic on the server for all Attribute Clients.

The declaration of the Server Characteristic Configuration is readable and writable.

| Server Characteristic Configuration |
| --- |
| **Handle: 16-bit UUID**<br>**Type: 0x2903**<br>**Value: List of Attribute Handles for Server Characteristic Decl**<br>**Permission: Higher layer specified** |

**Figure 4-10: Server Characteristic Configuration Declaration**

The descriptor is an Attribute with type set to 0x2903, and the value set to characteristic descriptor value.

**Note:** Service data in advertising data is managed by application using GAP interface.

#### 4.1.3.3.5 Characteristic Presentation Format

The Characteristic Presentation Format declaration is an optional characteristic descriptor that describes the Characteristic Value format. The value is composed of five parts: format, exponent, unit, name space and description.

| Characteristic Format |
| --- |
| **Handle: 16-bit UUID**<br>**Type: 0x2904**<br>**Value: Format, Exponent, Unit, Name Space, Desc**<br>**Permission: Higher layer specified** |

**Figure 4-11: Characteristic Format Declaration**

The access permissions are profile or implementation defined. The bit ordering is little-endian.

| Size | Component | Description |
| --- | --- | --- |
| 1 | Format | Format of the value |
| 1 | Exponent | Another representation for integer format types |
| 2 | Unit | Unit of the characteristic |
| 1 | Name Space | Identify the organization |
| 2 | Description | Depiction of the organization defined by Name space |

**Figure 4-12: Format Components**

#### 4.1.3.3.6 Characteristic Aggregate Format

The Characteristic Aggregate Format declaration is an optional characteristic descriptor that defines the format of an aggregated Characteristic Value, composed of a list of Attribute Handles of Characteristic Format declarations.

| Characteristic Aggregate Format |
| --- |
| **Handle: 16-bit UUID**<br>**Type: 0x2905**<br>**Value: List of Attribute Handles for Format Decl**<br>**Permission: Higher layer specified** |

**Figure 4-13: Characteristic Aggregate Format Declaration**

The Attribute Value is a list of Attribute handles, which is the concatenation of multiple 16-bit Attribute handle values. The list contains at least two Attribute handle for Characteristic Presentation format declaration.

### 4.1.4 L2Cap Requirements

| Parameter | Value | Description |
|---|---|---|
| L2CAP Channel ID | 0x0004 | Channel ID is fixed. |
| Maximum Transmission Unit | Mini 23 | GATT Client and Server is greater or equals 23 |
| Flush Time Out | 0xFFFF (Infinite) | PDUs shall be reliably sent, and not flushed |
| Flow specification | Best Effort | No defined QOS |
| Mode | Basic Mode | Mode of the L2CAP, No retransmission |

**Table 4-1: GATT Requirements for L2CAP**

## 4.2 Attribute Protocol Toolbox

The Attribute Protocol is used to read and write values from database of a peer device, called the *attribute server*. To do this, first the list of attributes in the database on the server are discovered. Once the attributes have been found, they can be read and written as required by the client.

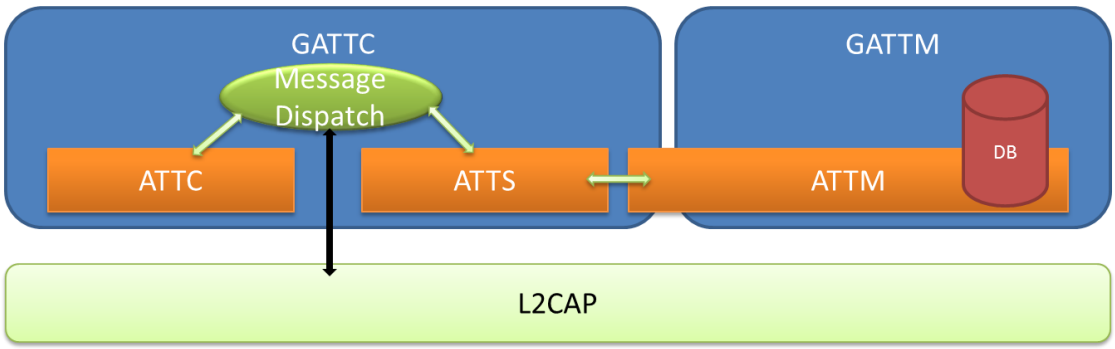



**Figure 4-14: Attribute module toolbox overview.**

The Attribute Block is composed of three entities: attribute server, attribute client and attribute manager.

The **attribute server** (ATTS) handles the server-based request messages and prepares responses for the received requests.

The **attribute client** (ATTM) handles the client-related request messages for attribute server.

The **attribute manager** (ATTM) is responsible of storing the attribute database of the device.

**Note:** Attribute Toolbox is not task oriented and can only be used by Generic Attribute Tasks.

### 4.2.1 Basic Attribute Concepts

#### 4.2.1.1 Attribute

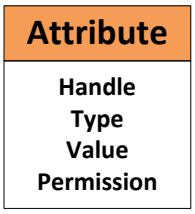



**Figure 4-15: Attribute Component**

An attribute is the basic block of the Attribute Protocol. It is composed of four items: attribute handle, type, value and permission property. The access permission to the attribute is defined by the higher layer, and is not accessible through Attribute Protocol.

| Element | Information |
|---|---|
| **Handle** | The attribute handle is a 16-bit value that is assigned by each server to its own attributes to allow a client to reference those attributes. The attribute handle on any given server shall have unique, non-zero values. |
| **Type** | An attribute type identified by a UUID specifies what the attribute represents. This is for the client to understand the meaning of the attributes exposed by a server. The UUID that identifies the attribute is considered unique over all space and time.<br>UUID is 128-bits in size, and for efficiency's sake, UUIDs may be shortened to 16-bits or 32-bits.<br>**Note:** 16-bits and 32-bits UUIDs are assigned by Bluetooth SIG. 32-bits UUIDs are reserved for proprietary profiles. 128-bits UUIDs can be used for any proprietary profiles without any fees. |
| **Value** | The attribute value is an octet array that may be either fixed or variable length. This is the actual value of the attribute and may contain a value that is too large to fit in a single PDU, which will be transmitted using multiple PDUs. |
| **Permission** | An attribute can have a set of permission values associated with it.<br>1. Read, Write Access Permission<br>2. Indications or notifications permission<br>3. Security Access Requirement: Authentication required or not |

**Table 4-2: Attribute description**

#### 4.2.1.2 Protocol Methods

Examples of protocol method are request, response, command, notification, indication and confirmation. These are used by the Attribute Protocol to find, read, write, notify and indicate attributes.

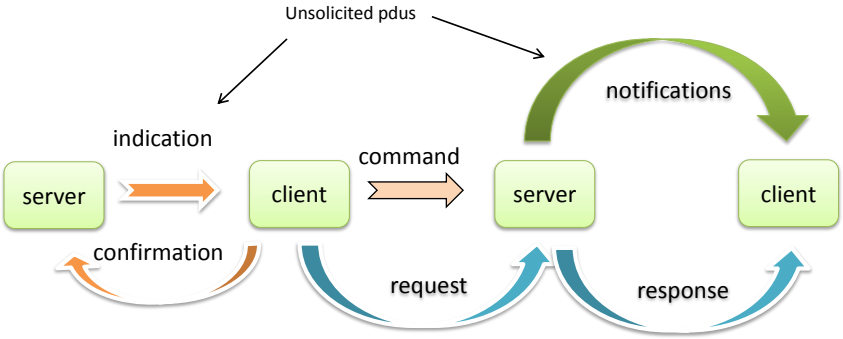



**Figure 4-16: Overview of ATT protocol messages**

### 4.2.2 Attribute Protocol Packet Data Unit Format

All Attribute Protocol messages in L2CAP are transmitted using a fixed channel ID (0x0004).

There are 6 types of Attribute Protocol PDUs:

1. Requests –PDUs which are sent to a server by a client and invoke responses.
2. Responses –PDUs which are sent in reply to an attribute client's requests.
3. Commands – PDUs which are sent to a server by a client.
4. Notifications – PDUs which are unsolicited sent to a client by a server.
5. Indications – PDUs which are unsolicited sent to a client by a server, and invoke confirmations.

6. Confirmation – PDUs which are sent to a server to confirm receipt of an indication to a client.

Multi-octet fields within the attribute protocol are transmitted the least significant octet first (little endian).

Attribute PDUs may or may not contain signature.

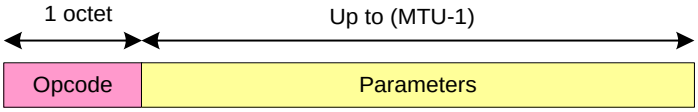



**Figure 4-17: ATT PDU if without signature**

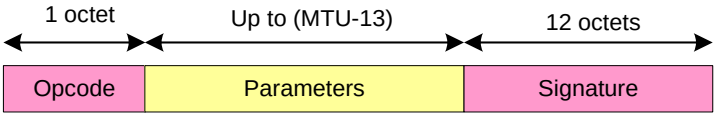



**Figure 4-18: ATT PDU if with signature**

L2CAP Attribute Protocol PDU messages are described in Core Specification (see [1] Volume 3, Part F, Chap. 3.4).

### 4.2.3 Attribute Protocol Operations

#### 4.2.3.1 Atomic Operations

Each command sent by the client is atomic in nature, and are treated by the server as one command, unaffected by another client sending a command simultaneously.

#### 4.2.3.2 Flow Control

Once a command has been sent to an attribute server, no other commands are sent to the same attribute server until a response message has been received.

It is possible for an attribute server to receive a command, send an indication back, and then the response to the original command. The flow control of commands is not affected by the transmission of the indication.

#### 4.2.3.3 Transaction

An Attribute Protocol command and response pair is considered as a single transaction.

A transaction starts when the request is sent by the attribute client. A transaction is completed when the response is received by the attribute client.

Similarly, a transaction starts when an indication is sent by the attribute server. A transaction is completed when the confirmation is received by the attribute server.

A transaction have to be completed within **30 seconds**, else it is considered has timed out. If a transaction has not completed before it times out, then this transaction is considered to have failed and the local higher layers are informed of this failure. No more ATT transaction accepted for the link.

### 4.2.4 Attribute Protocol Module Interfaces

#### 4.2.4.1 Interface with upper layers

The Attribute Protocol Module provides an API to the upper layers to allow them the following operations:

- ✓ reading/writing attributes, and receive notifications and indications (client side)
- ✓ sending notifications/indications, be notified when a client reads or writes an attribute (server side)

This API is implemented as functions available for GATT Modules (see [6]).

#### 4.2.4.2 Interface with L2CAP

The interface with L2CAP is handled by GATT task, It then use attribute toolbox to process them (see 3.2.1).

### 4.2.5 Attribute Manager (Database owner)

Managed by Attribute Manager module, the attribute database is composed into a list of services dynamically allocated.

A service is a memory block allocated from Kernel attribute heap and available for attribute manager into a list of services sorted by handles.

Attribute manager provide a function API available for GATTM (see [6]) in order to allocate new service with a specific start handle. If not set, start handle is dynamically allocated.

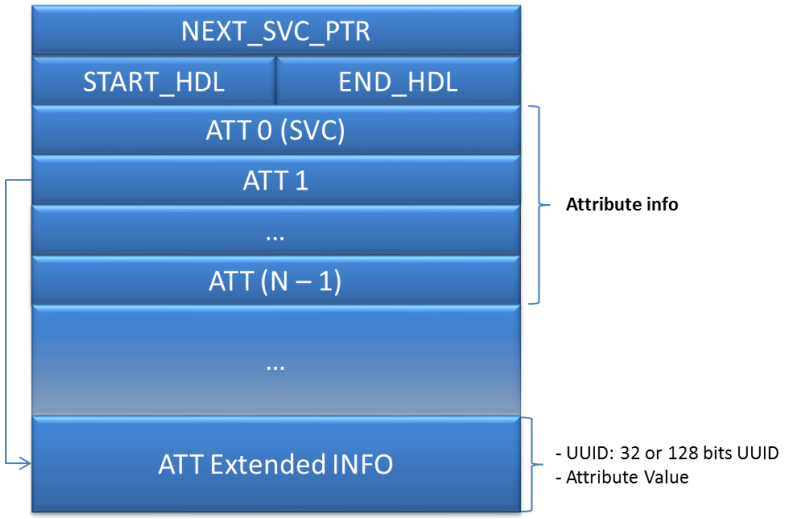



**Figure 4-19: Service Description block of ATT database.**

This memory block contains a pointer to next service (NEXT_SVC_PTR), its start handle and the last handle value. Then it contains an array of attributes definition (see 4.2.5.1).

First attribute into service describes the services (see 4.2.5.3). It is used to know services permission and number of attributes present into the service. It's forbidden to have several services attribute into a service memory block.

Finally end of memory block is used to retrieve 32-bits or 128-bits UUIDs and attribute values that can be read from the database.

**Note:** Attribute handle are unique, services handles have to be exclusive.

**Note:** Services handles mapping should be fixed in order to prevent collector to perform discovery at each connection.

#### 4.2.5.1 Attribute definition

An attribute is a 6 bytes field used to describe its UUID, its permissions and some extended information such as:

- Service Task ID
- Pointed handle
- Maximum Attribute Length
- Value
- Value offset.

**Note**: if Attribute UUID is a 32 or 128 bits UUID, UUID value contains offset in the service block where it can be found.

**Note**: Figure 4-20 describes Attribute types specified by Core Specification (see [1] Volume 3, Part G, Chap 3)

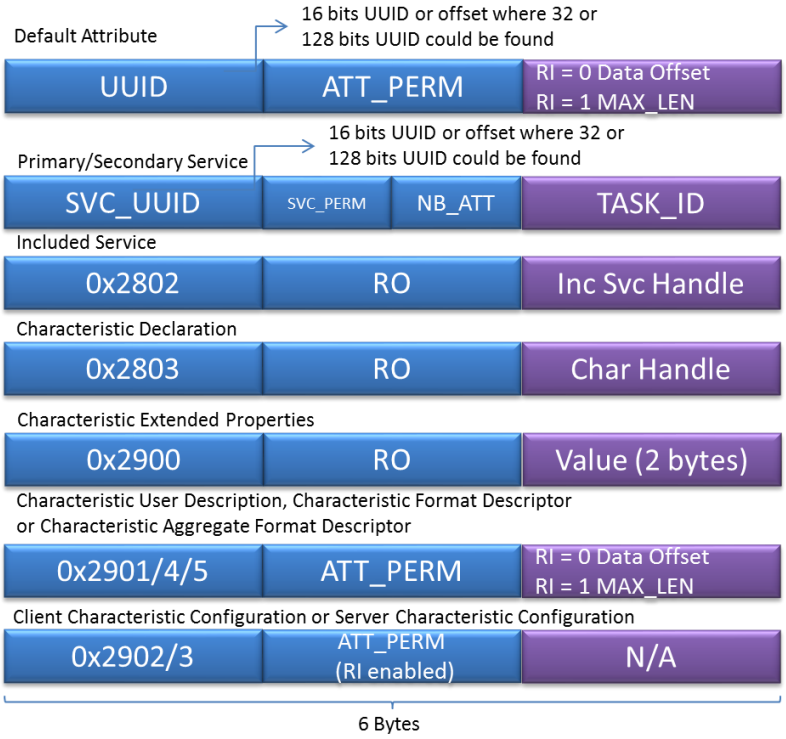



**Figure 4-20: Attributes types.**

### 4.2.5.2 Service definition

A service is described with a 6 bytes field:

- Task Handler
- Service permissions
- Number of attributes in service
- Service UUID

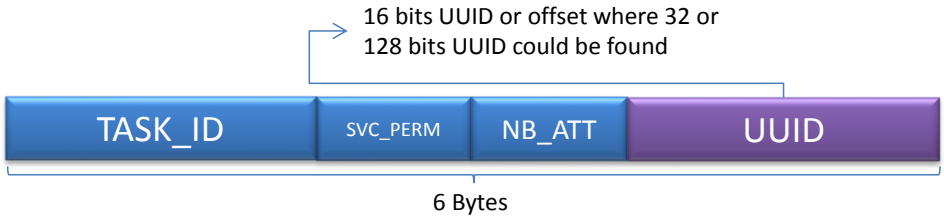



**Figure 4-21: Service definition.**

**Note**: if Service UUID is a 32 or 128 bits UUID, UUID value contains offset in the service block where it can be found.

#### 4.2.5.3 Service Permission Field

Service permission is an 8 bits field used to describe Service

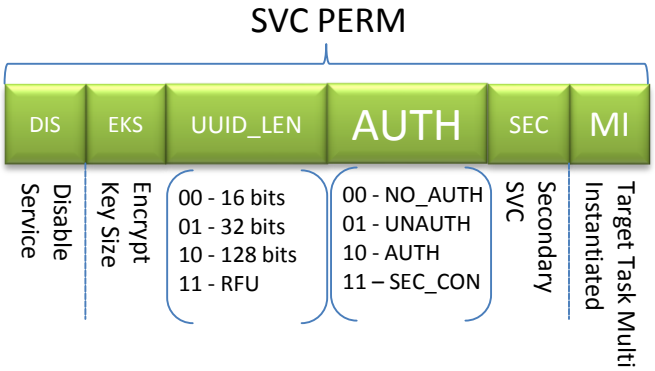



**Figure 4-22: Service Permission field**

128 bits, the UUID field contains offset pointer.

- **P**: Used to know if service is a primary or a secondary service.
- **UUID_LEN**: get Service UUID length (16, 32 or 128 bits) If length is 32 or
- **AUTH**: Force a level of authentication for attributes present into the service. Note, this has no impact on attributes which are Read-Only mandatory.
- **EKS**: Required an encryption key of 16 bits for attribute requiring an authentication level.
- **MI**: Used to know if Task that manages service is multi-instantiated or mono-instantiated.

#### 4.2.5.4 Attribute Permission Field

Attribute permission is a 16 bits field:

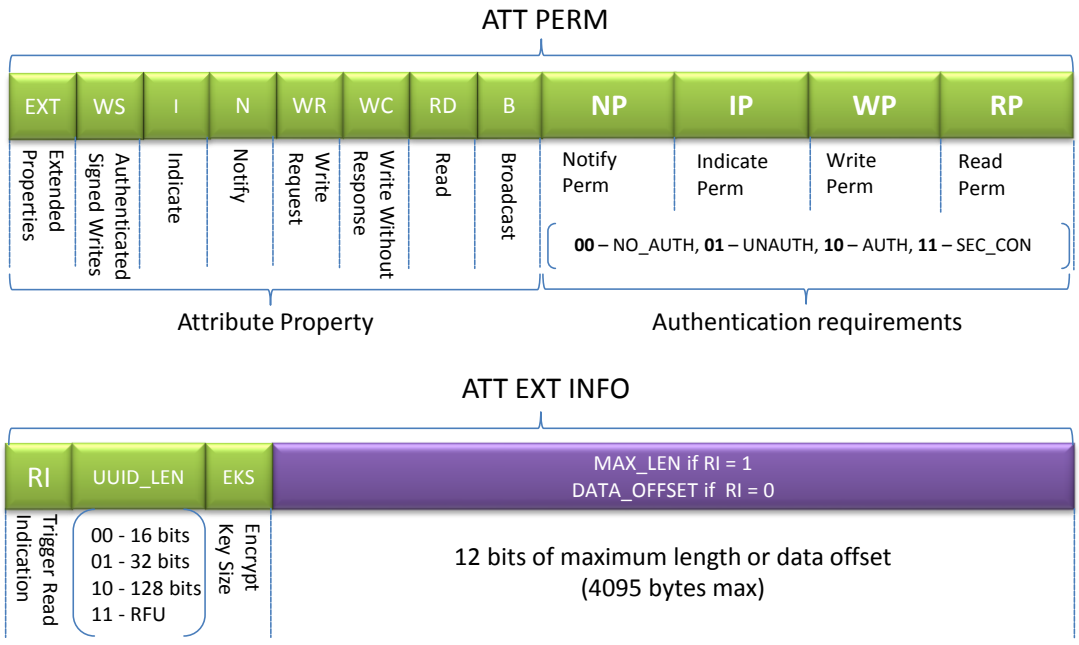



**Figure 4-23: Attribute Permission field**

**Following field used to generate Characteristic declaration properties value:**

- **RD:** read attribute allowed
- **WR:** Write Request allowed on current attribute
- **WS:** Write Signed allowed on current attribute
- **WC:** Write without response allowed on current attribute
- **N**: Notification event allowed
- **I**: Indication event allowed
- **B**: Attribute value can be broadcasted using advertising data *(SCC descriptor shall follow)*
- **EXT**: Extended property field present *(CEP descriptor shall follow)*

**Attribute Authentication requirements**

- **WP:** Write permission allowed with a certain level of authentication
- **RP:** Read permission allowed with a certain level of authentication
- **NP**: Notification allowed with a specific level of authentication *(CCC descriptor shall follow)*
- **IP**: Indication allowed with a specific level of authentication *(CCC descriptor shall follow)*

**Note**: For an attribute value, permissions are used to generate Characteristic Description property value.

**Extended Attribute information**

- **RI**: trigger a request to profile when a read is requested by peer collector
- **UUID_LEN**: Attribute UUID length (16, 32 or 128 bits). If length is 32 or 128 bits, the UUID field contains offset pointer.
- **EKS**: Required an encryption key of 16 bytes for attribute requiring an authentication level.
- **MAX_LEN**: Maximum length of the attribute that can be written (valid only if RI = 1)

  **or**
- **DATA_OFFSET**: Data offset of the attribute value in service memory block (valid only if RI = 0)

#### 4.2.5.5 Data Caching

In order to ease database browsing, since several search can be performed on a same service, keeping pointer to last search service in environment variable speeds-up service and attribute discovery.

#### 4.2.5.6 Attribute Database Example

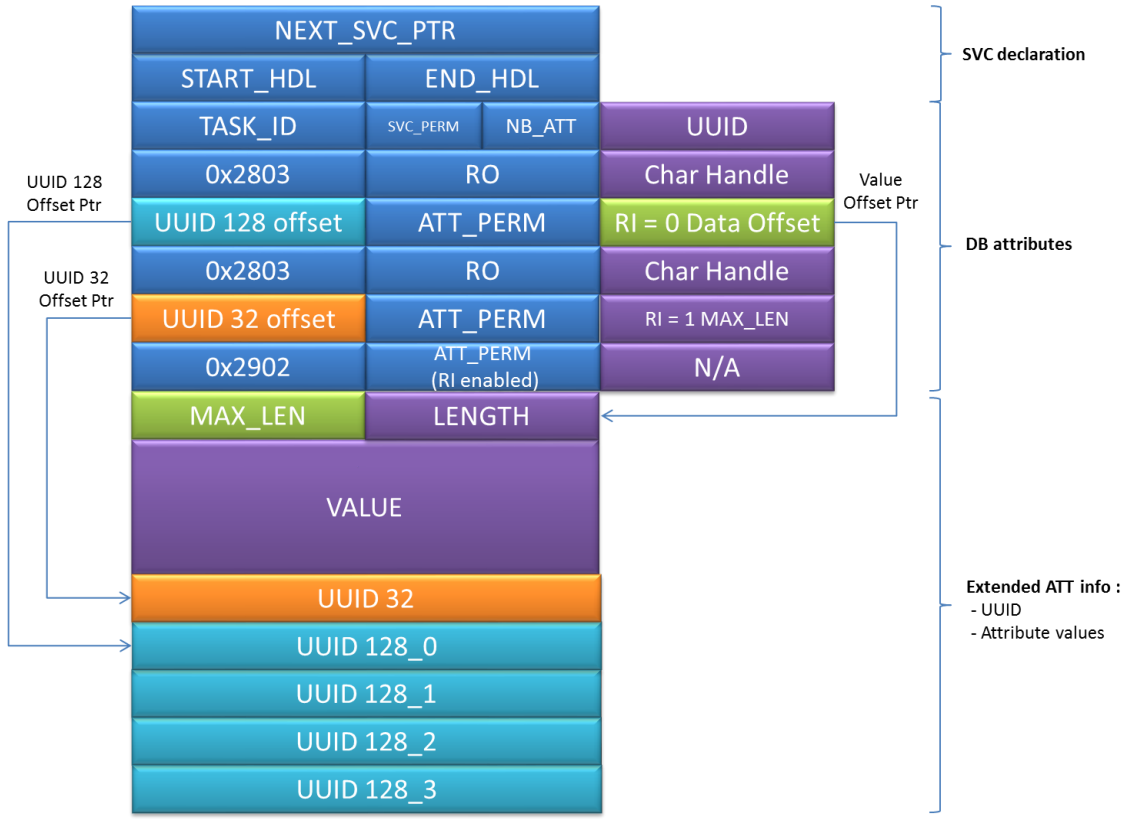



**Figure 4-24: Attribute database example**

### 4.2.6 Attribute Server

Attribute server have direct (function call) interface with Attribute database present in attribute manager. It uses this interface to browse services and read characteristics values.

#### 4.2.6.1 Attribute Discovery / Read

Attribute discovery procedures, or reading procedures can encounter those issues:

- Total length of the response exceeds MTU
- Attribute to read is not present into the database.

The discovery procedure is rescheduled in kernel several times until its completion. Incomplete response is kept into a cache variable and is fulfilled at end of search, or if MTU is exceed.

This procedure also uses data caching of ATTS in order to accelerate read (see 4.2.6.4).

Search algorithm is described in Figure 4-25.

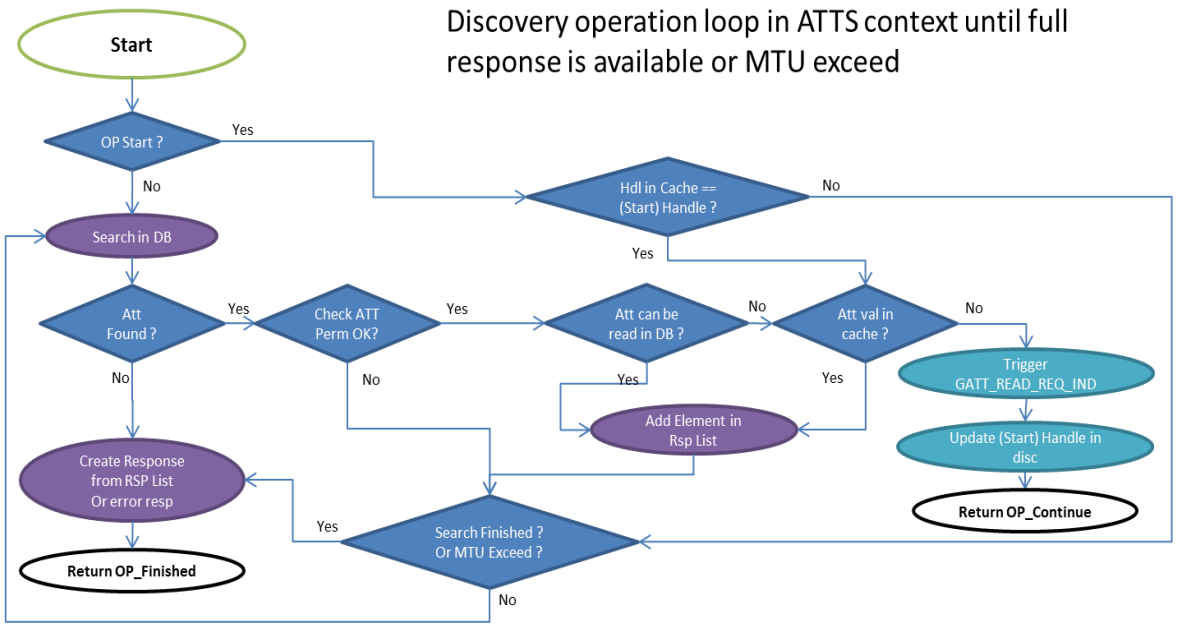



**Figure 4-25: Attribute discovery state machine**

### 4.2.6.2 Attribute Write

Following figure describes different type of write procedure in ATT

- **Write Request**

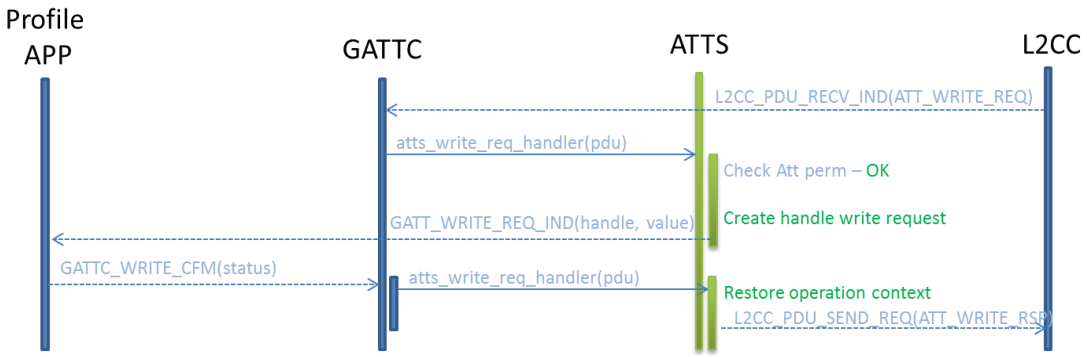



**Figure 4-26: Write Request MSC**

- **Write Command**

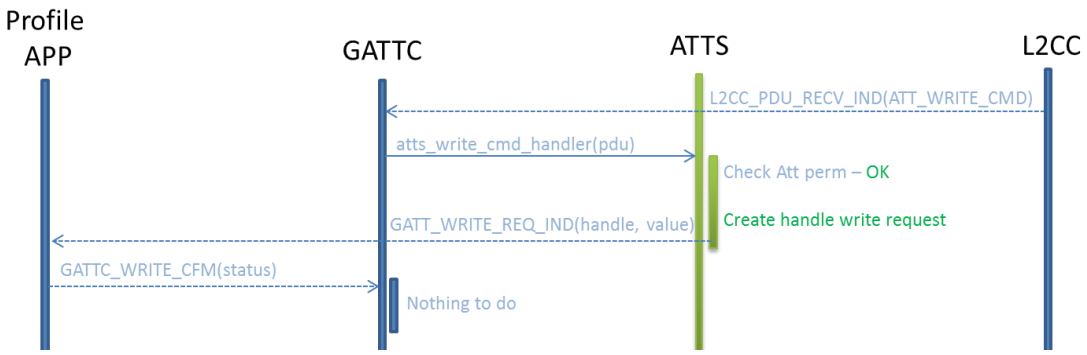



**Figure 4-27: Write Request MSC**

- Write Signed

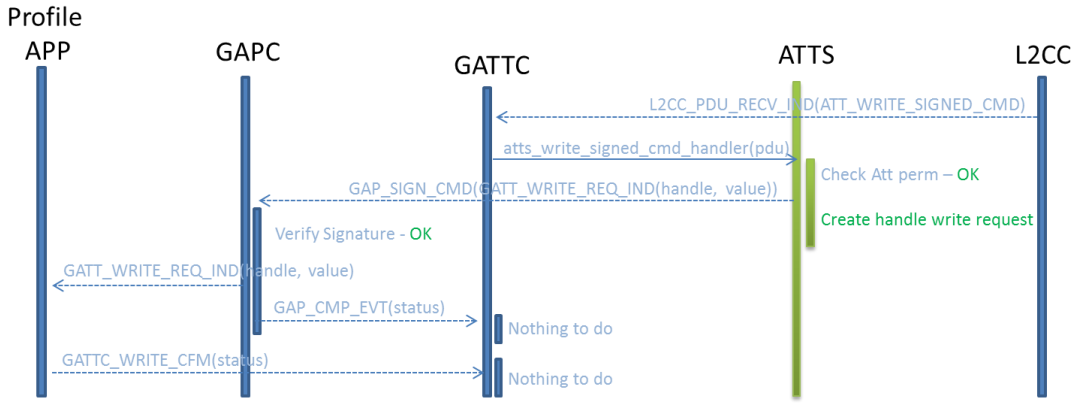



**Figure 4-28: Write Signed MSC**

- Write Long/Multiple

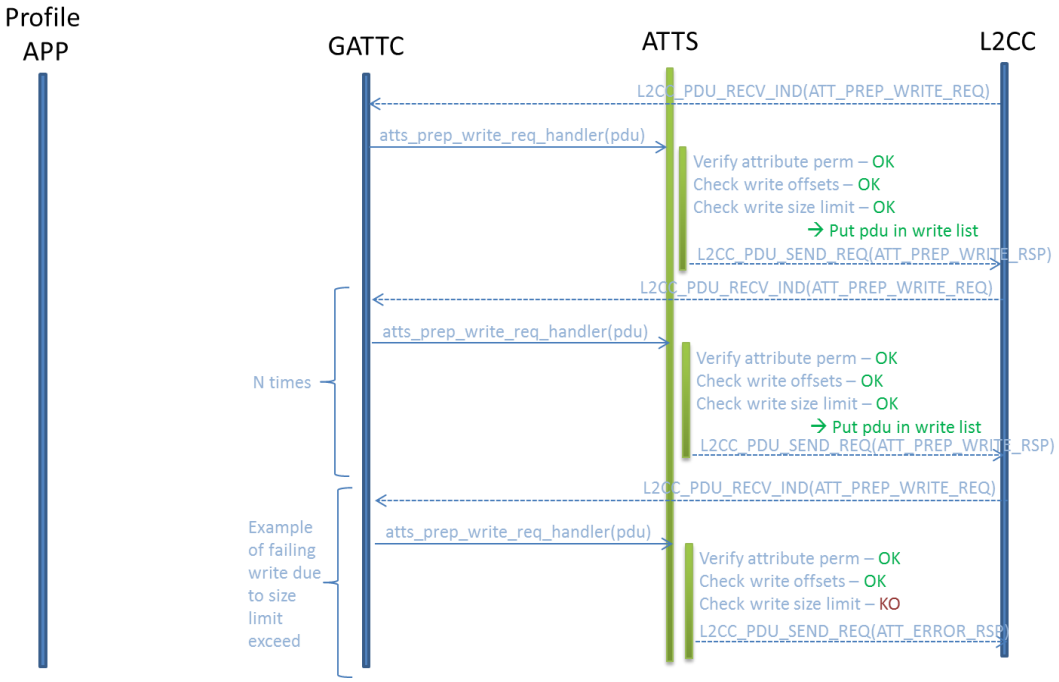



**Figure 4-29: Multiple prepare write MSC**

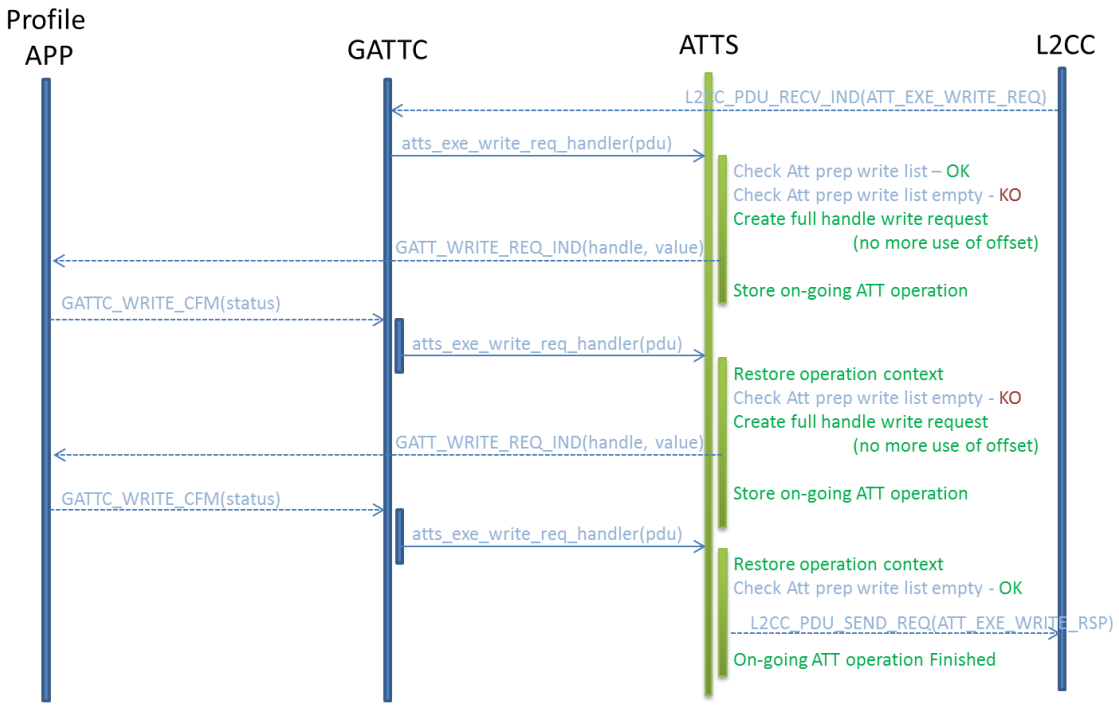



**Figure 4-30: Execute write MSC**

**Note**: Write request is always send to profile that manage handle, it requires a confirmation of write event if a message is triggered to peer device or not.

#### 4.2.6.3 Server Initiated events

Attribute server can be used to trigger some indication or notifications:

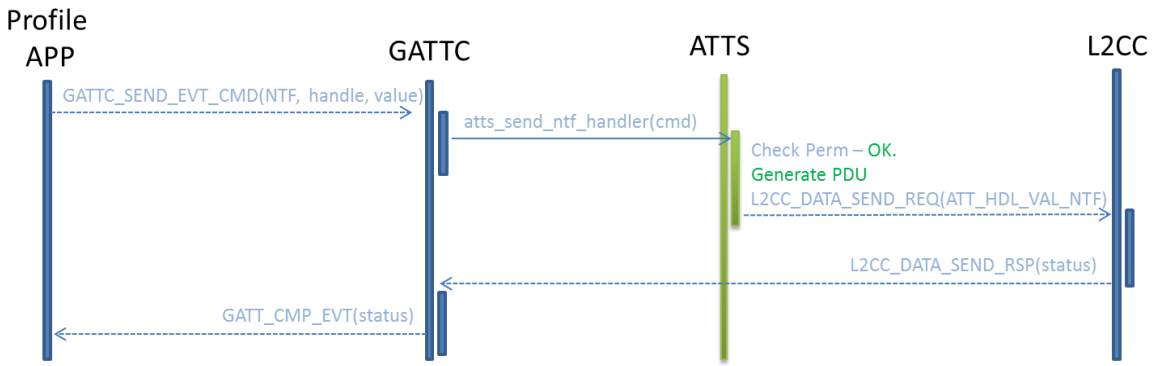



**Figure 4-31: Trigger notification MSC**

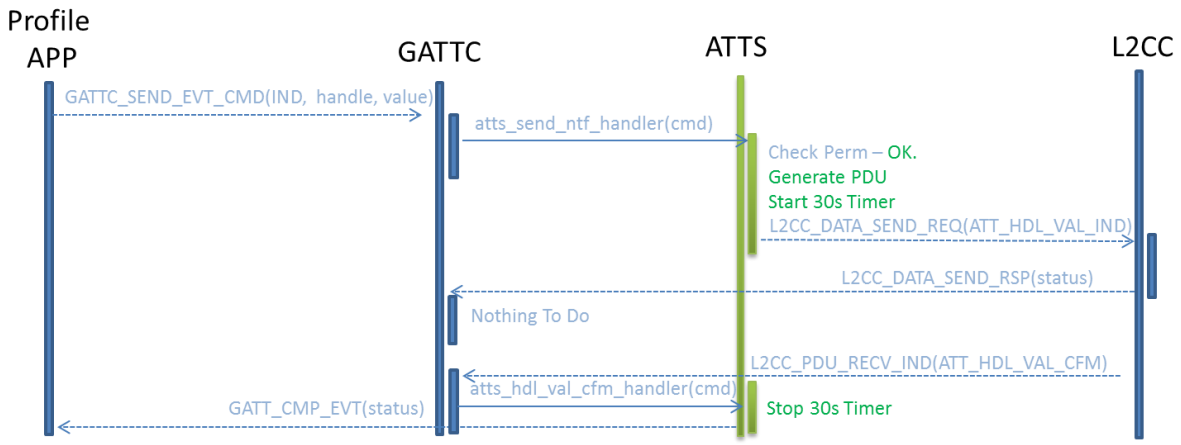



**Figure 4-32: Trigger indication MSC**

**Note**: Notification/Indication data is present into the event message. This event message can be used to update database value (If attribute value present in database).

#### 4.2.6.4 Data Caching

**Ongoing procedure**:

Ongoing procedure pointer (L2CAP message) is kept in order to be reschedule by kernel until its operation is finished and in order to perform flow control on the requests

**Response cache:**

Until executed procedure is finished, partial procedure response is stored into attribute server environment.

**Prepare write cache:**

For non-atomic write, a cache is required. This cache is feed by prepare write and flushed during execute write request.

**Read attribute Cache** (see Figure 4-33)

In Attribute database, in order to speed-up read procedures, value of latest attribute read is kept until:

- A write request is accepted on this attribute
- Notification/indication is triggered for this attribute
- Attribute fully read by peer device.
- Disconnection

**Note**: Attribute value is put in cache if value not present in attribute database.

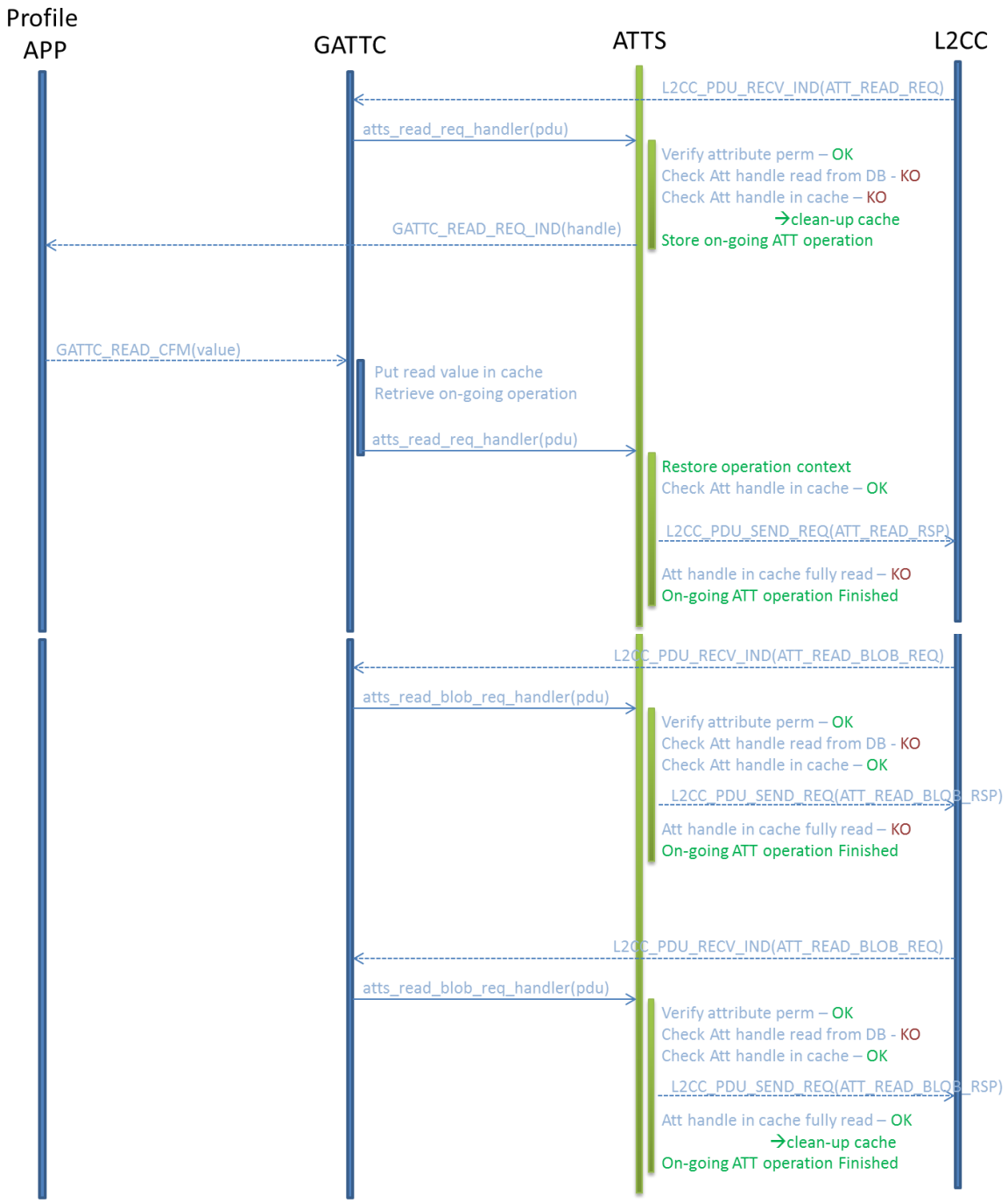



**Figure 4-33: Data caching of latest read attribute MSC**

### 4.2.7 Attribute Client

Attribute client role is very simple; it conveys requests from GATT client to L2CAP, Manage transaction atomicity and maximum duration using a timer.

**Note**: Discovery and read procedure using UUID as input supports 16, 32 and 128 bits UUIDs.

#### 4.2.7.1 Discovery Command

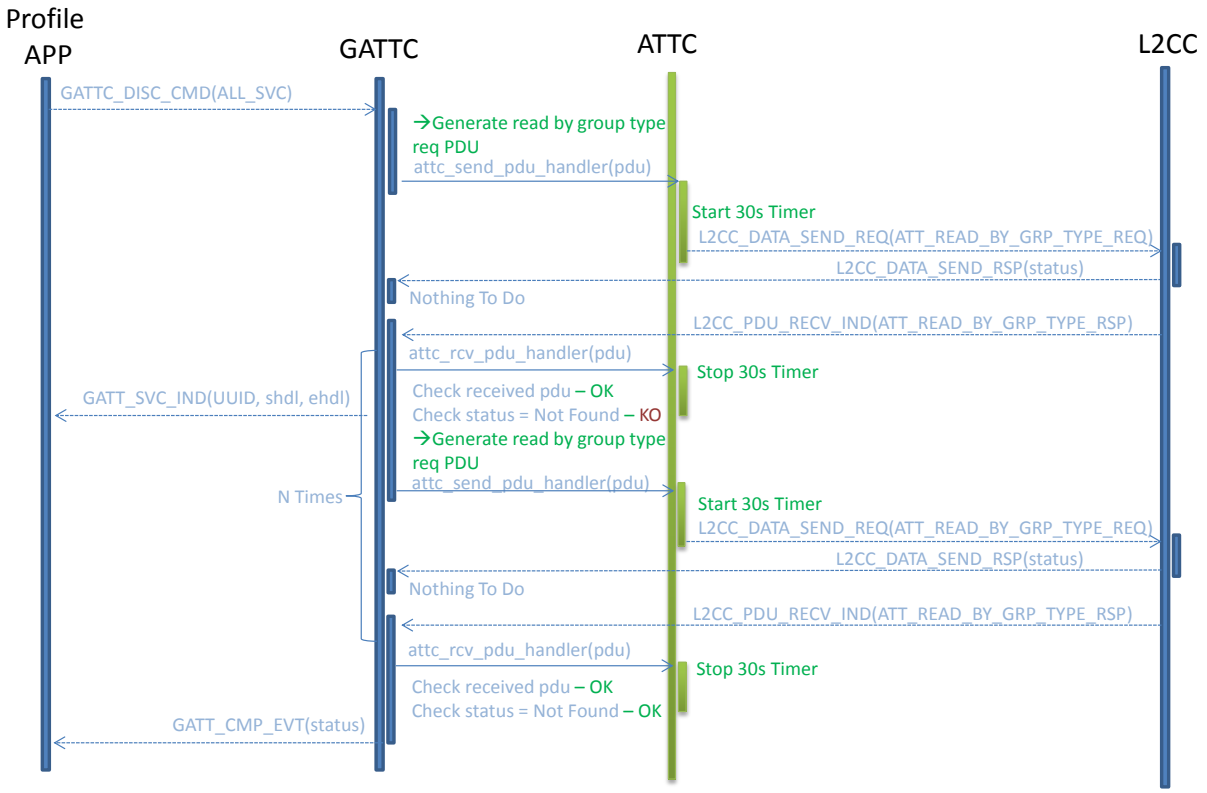



**Figure 4-34: Discover all peer services MSC**

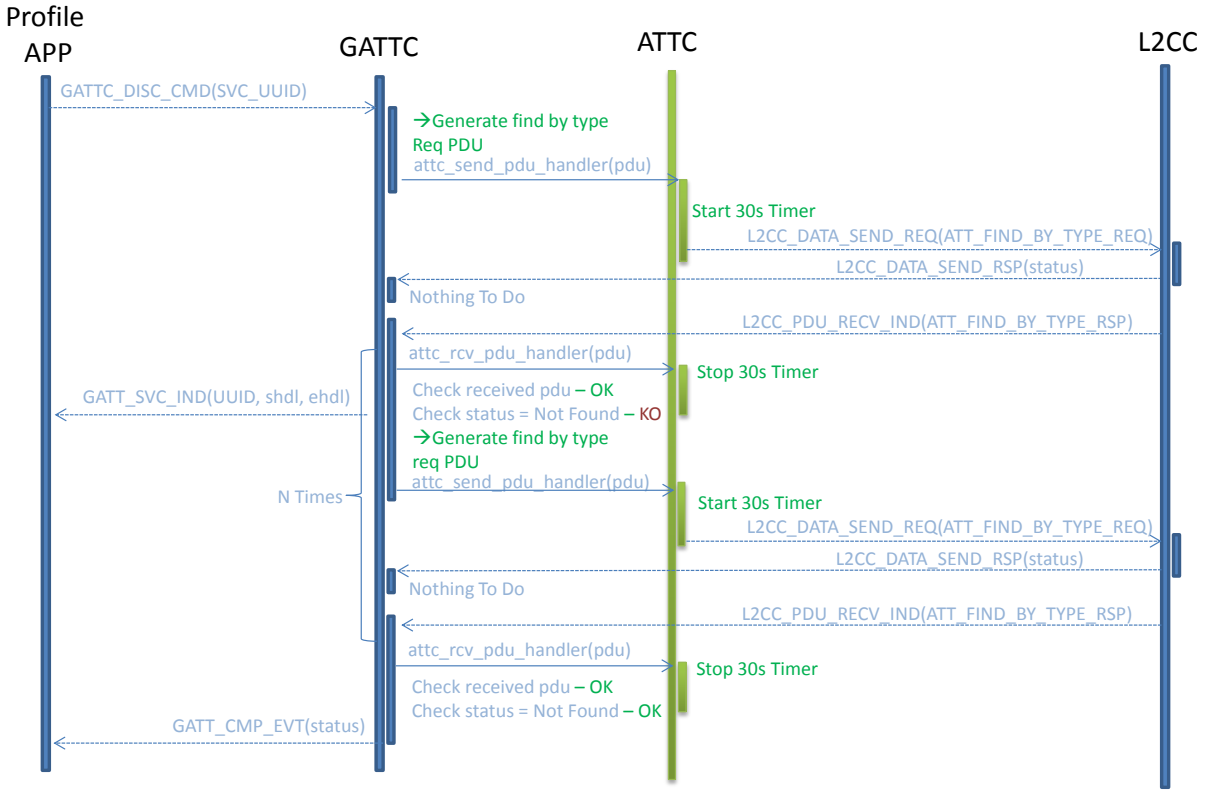



**Figure 4-35: Discover peer services with specific UUID MSC**

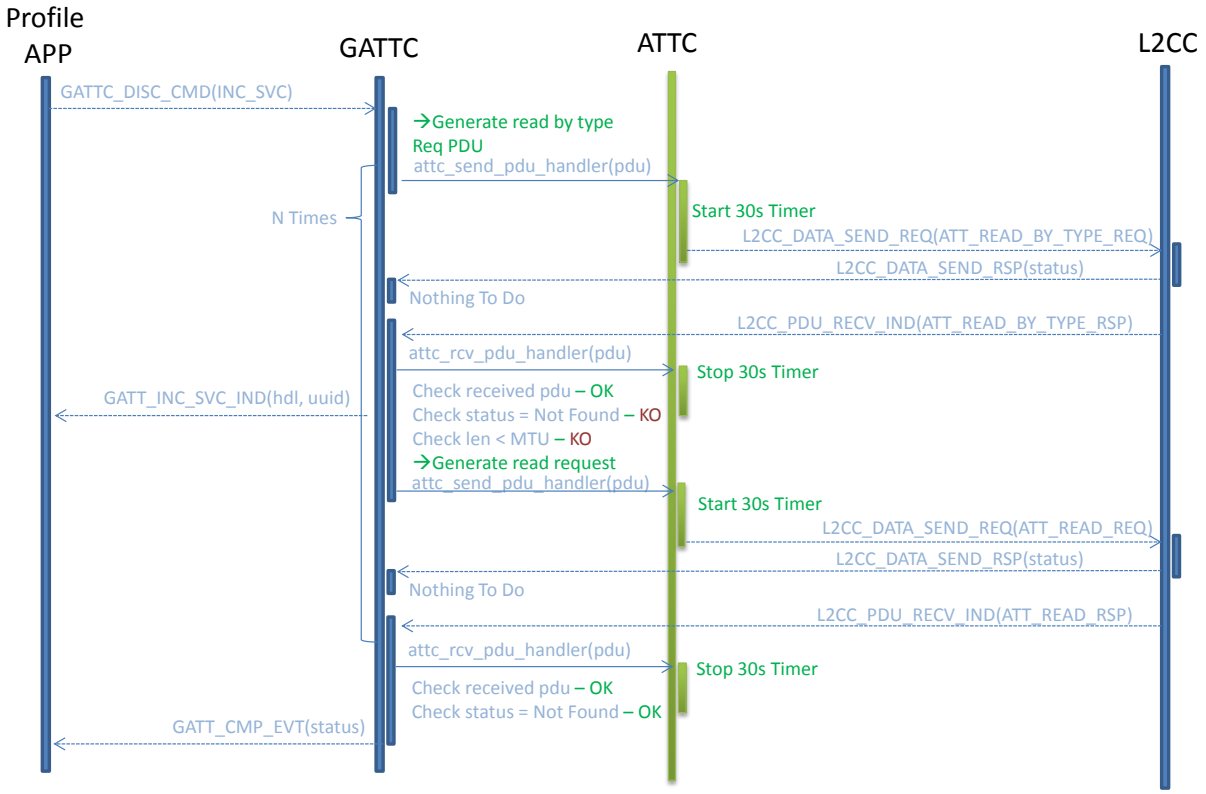



**Figure 4-36: Discover peer included services UUID MSC**

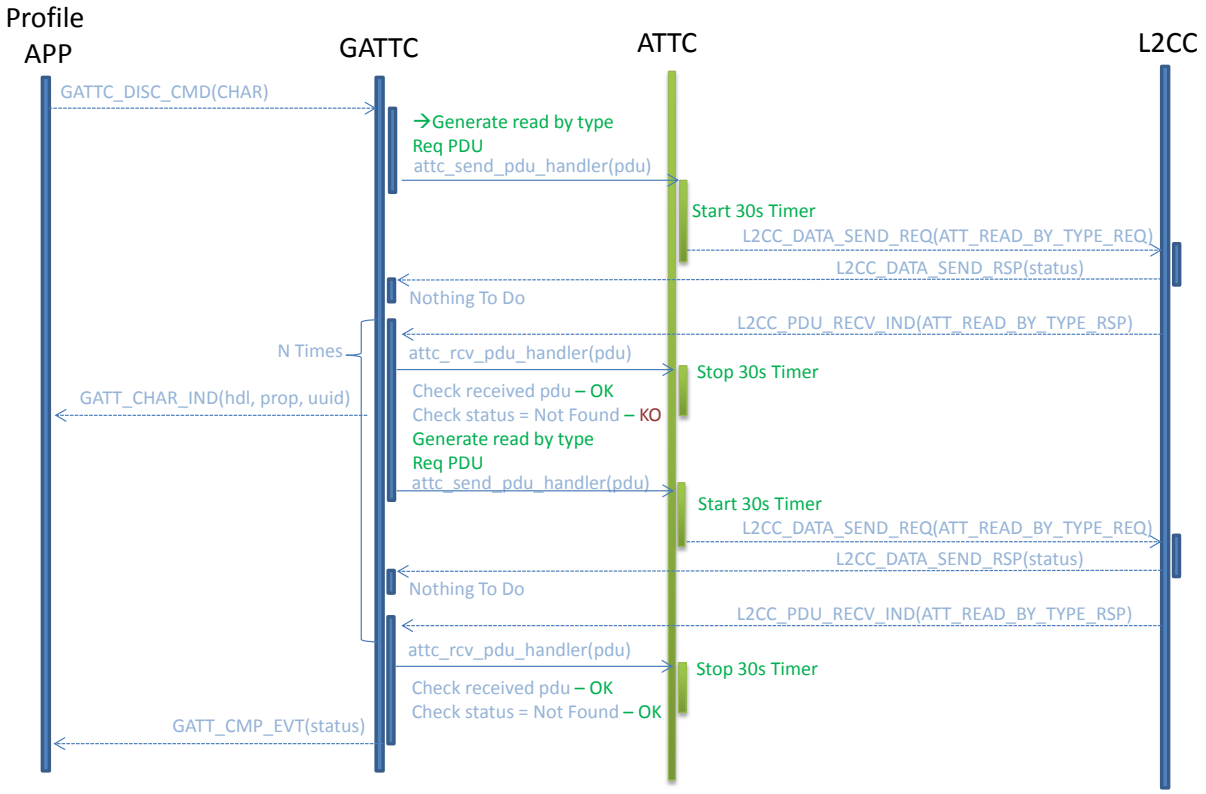



**Figure 4-37: Discover peer characteristics (all or with specific UUID) MSC**

**Note**: Same procedure is used to discover all characteristics or with a specific UUID. The filtering of UUID is performed by client side and not by service side.

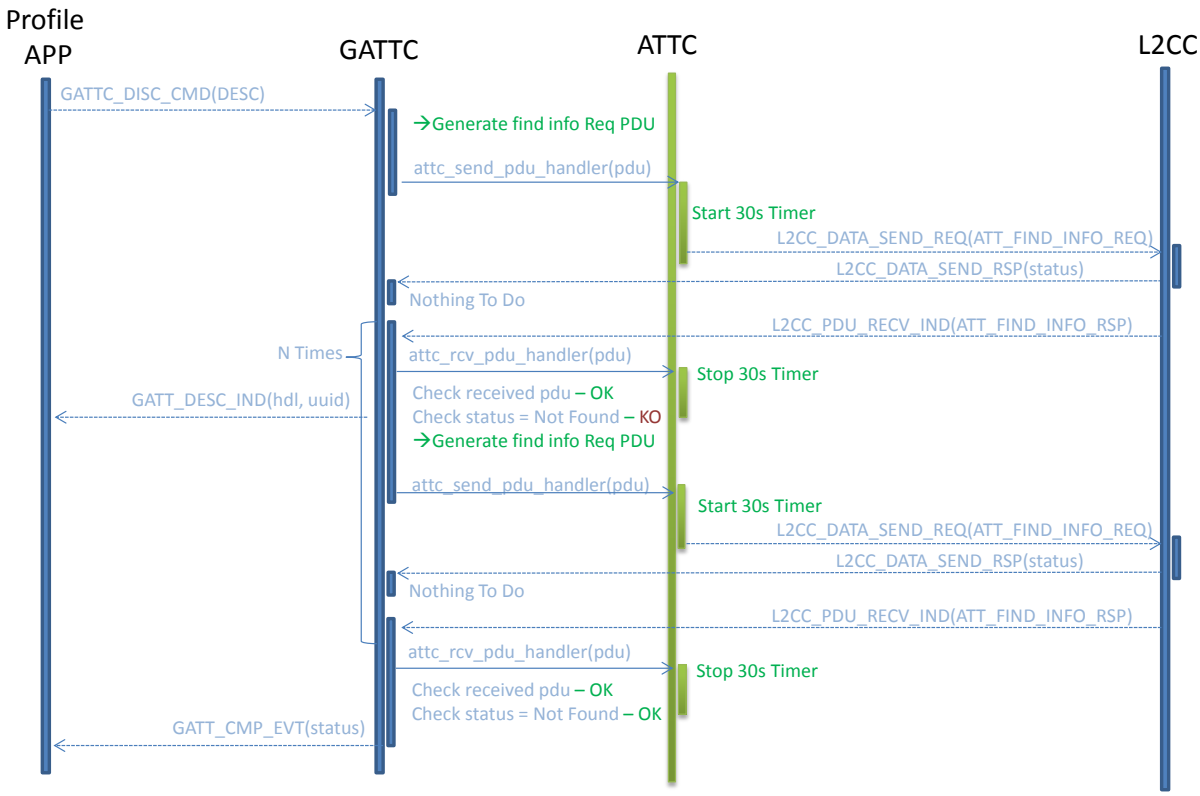



**Figure 4-38: Discover peer descriptors MSC**

#### 4.2.7.2 Read Command

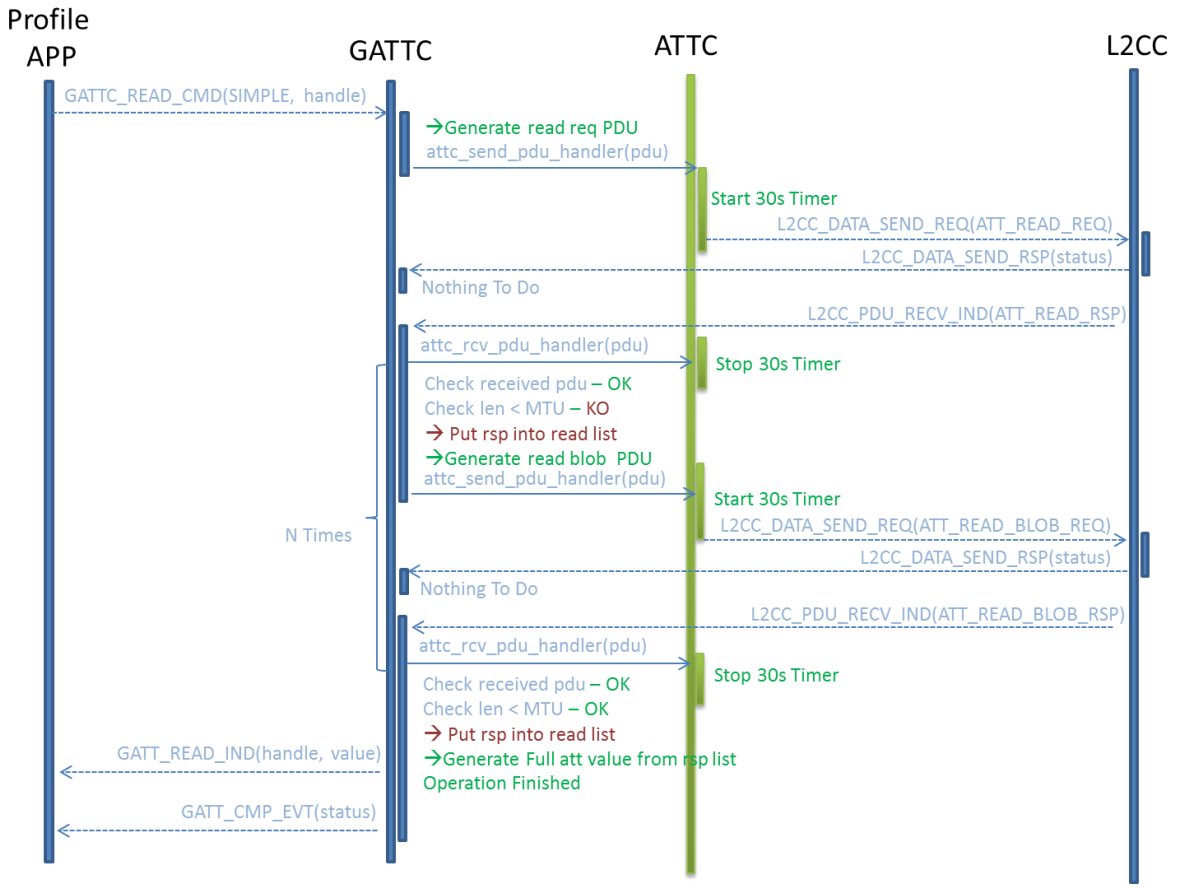



**Figure 4-39: Read Simple Request MSC**

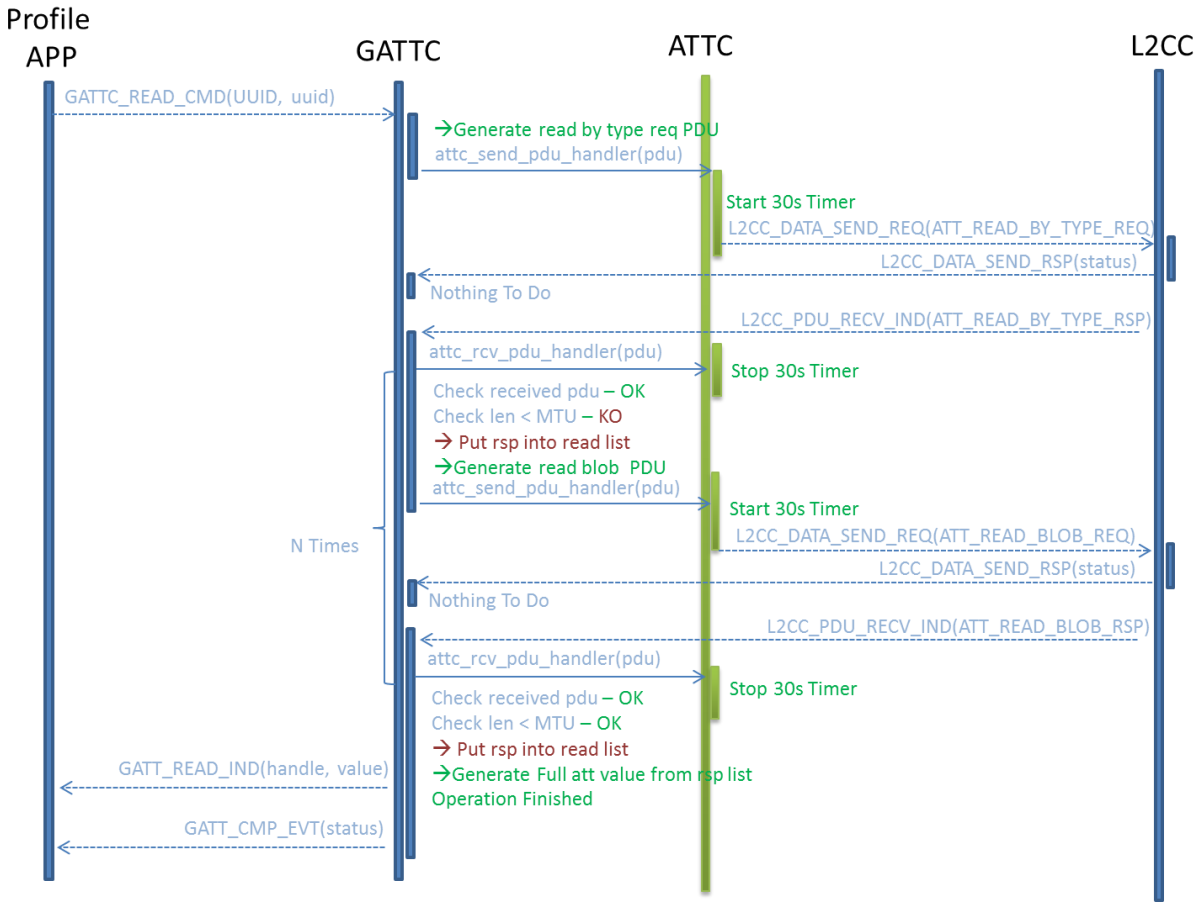



**Figure 4-40: Read By UUID Request MSC**

### 4.2.7.3 Write Command

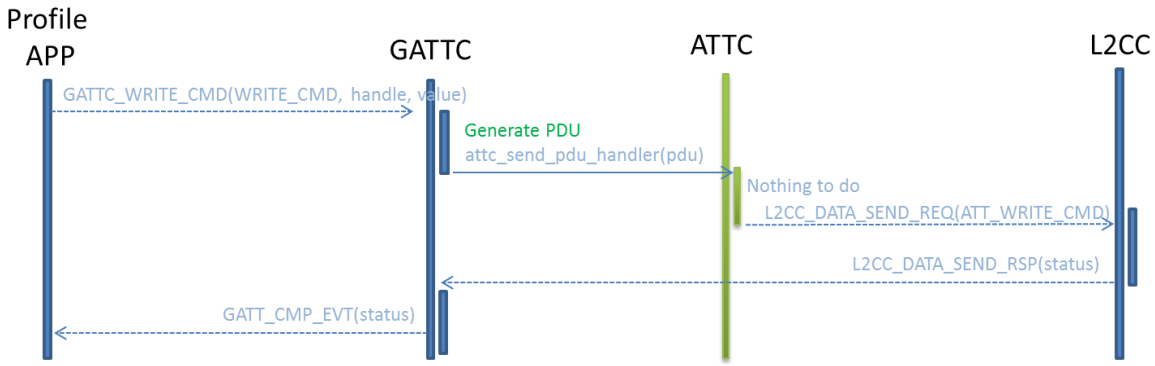



**Figure 4-41: Write Command MSC**

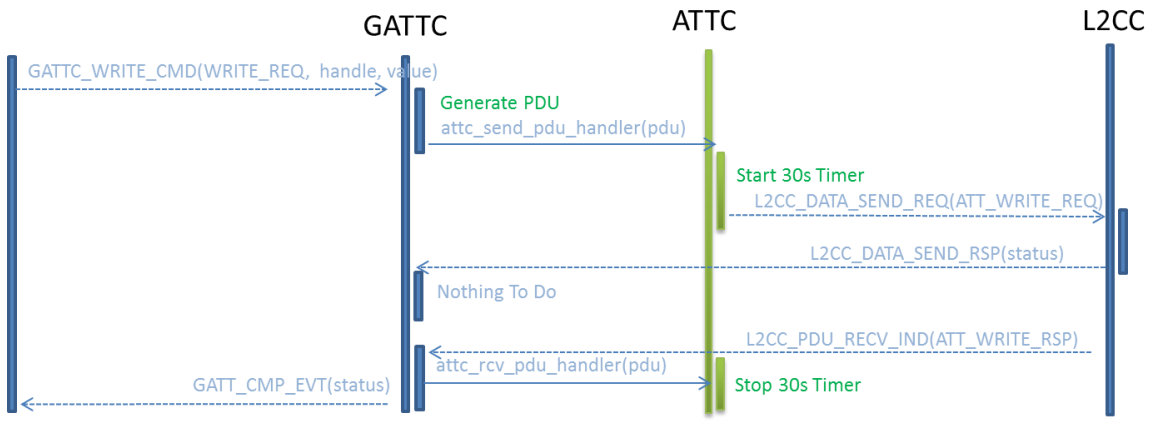



**Figure 4-42: Write Request MSC**

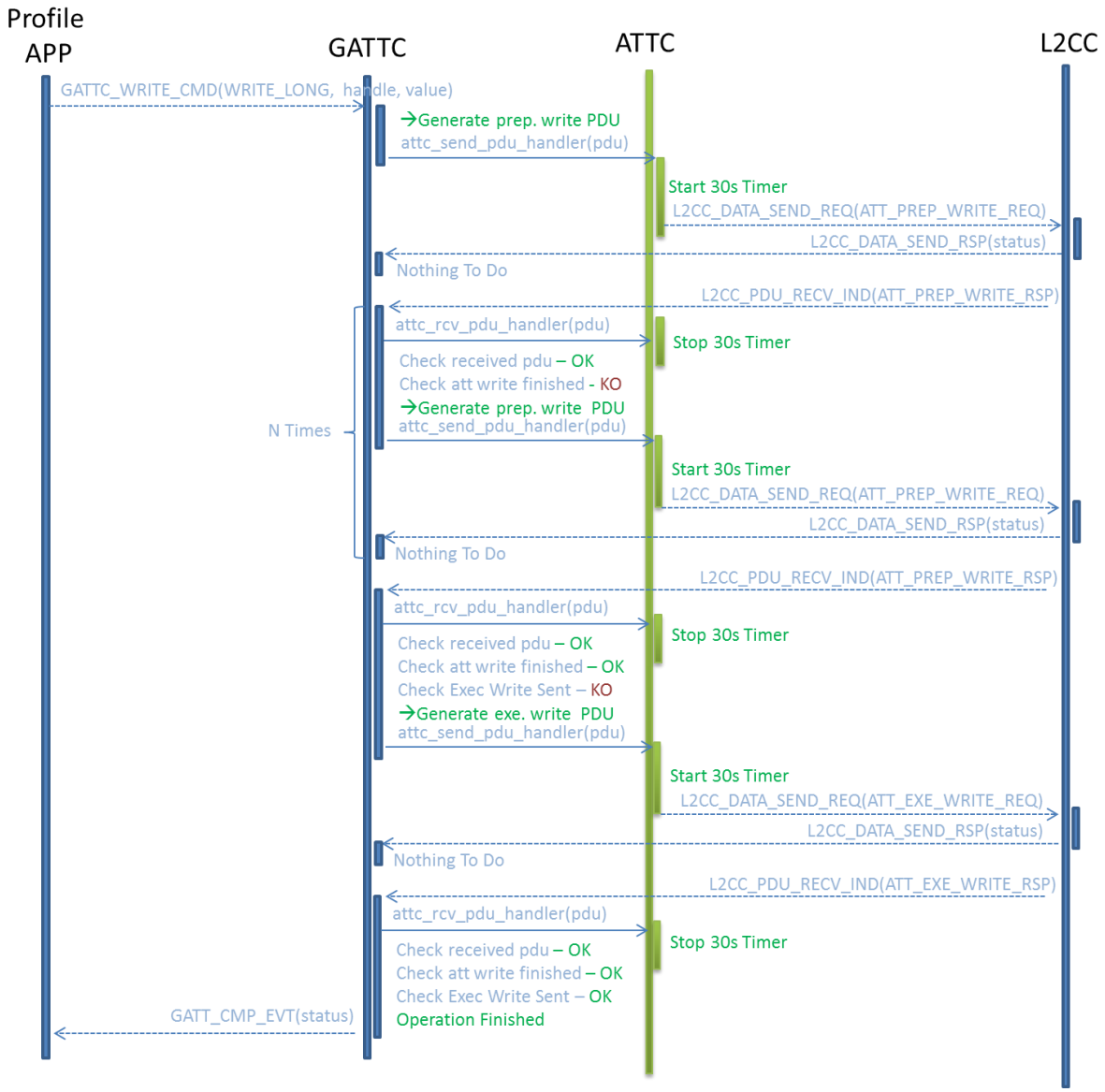



**Figure 4-43: Write Long/Multiple MSC**

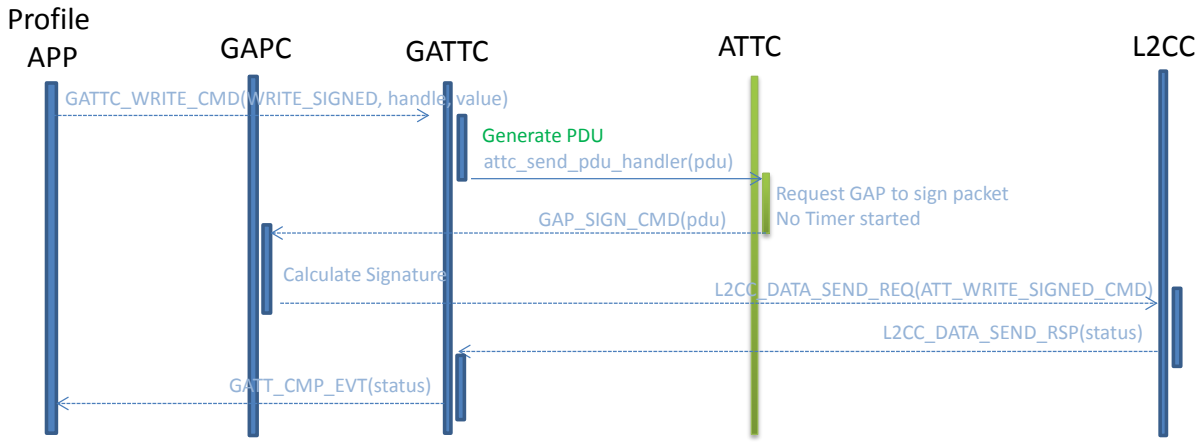



**Figure 4-44: Write Signed MSC**

#### 4.2.7.4 Reception of Notification or Indications

In order to allow a profile to receive notification or indication of a peer device, it has to register to service events. This can be done by provided peer service handle range.

**Note**: By default application if informed of any received events if no registration has been performed.

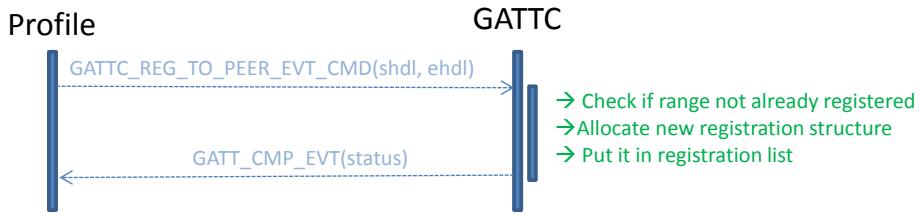



**Figure 4-45: Event handle range registration MSC**

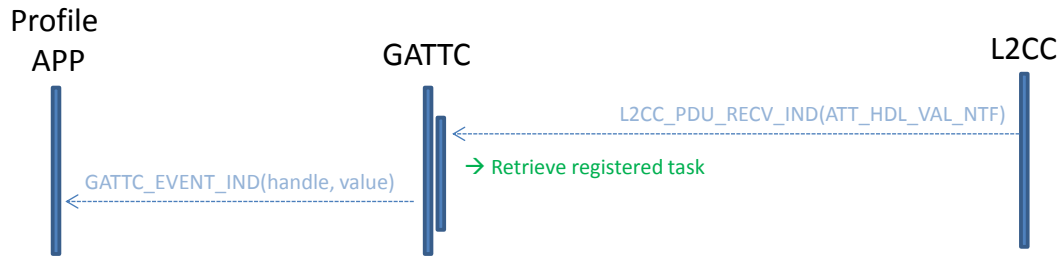



**Figure 4-46: Reception of a notification from peer device MSC**

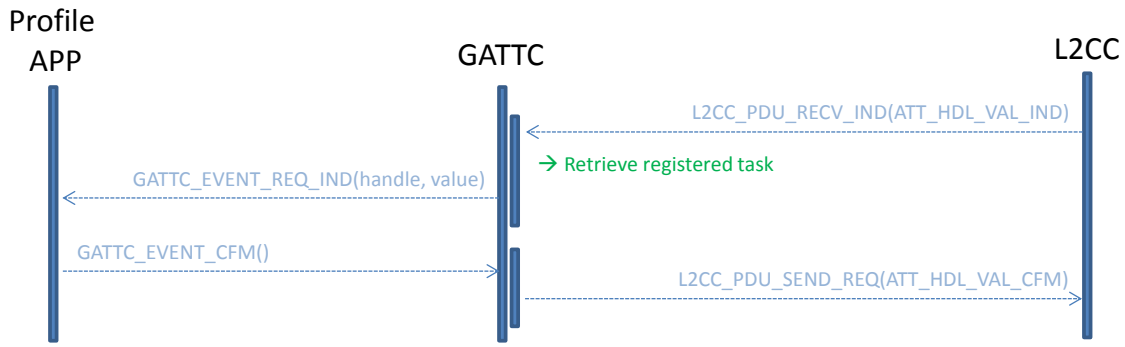
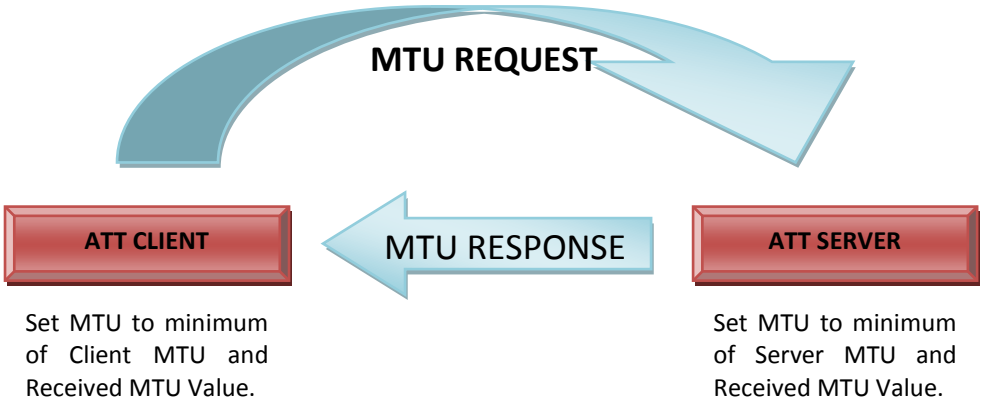



**Figure 4-47: Reception of an indication from peer device MSC**

## 4.3 Features and Functions

### 4.3.1 Attribute Packet Size Negotiation




The MTU exchange procedure is a sub-procedure of server configuration. This is launch by the Attribute Client to configure the Attribute Protocol.

At the end of the exchanges, both the Attribute Client and Server will have common set MTU, which is the minimum value exchanged.

### 4.3.2 Primary Service Discovery

The procedure is used by an Attribute Client to discovery primary services on a server. Once these services are discovered, additional information like characteristics and secondary services may be retrieved.

There are two sub-procedures for primary service discovery.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Discover All Primary Services | Read By Group Type Request<br>Read By Group Type Response<br>Error Response |
| Discover Primary Services By Service UUID | Find By Type Request<br>Find By Type Response<br>Error Response |

**Table 4-3: Primary Service Discovery Sub-Procedures**

The “Discover All Primary Services” is used by the client to discover all the primary services on a server.

The “Discover Primary Services by Service UUID” is used by the client to discover a specific primary service on a server when only the Service UUID is identified.

The functions are completed in two ways: either the application receives an error code (Attribute not found) or the application cancels the search (in case the desired primary service is already found).

Insufficient authentication error or read not permitted error shall not occur (Service declaration is readable and requires no authentication or authorization).

### 4.3.3 Relationship Discovery

The procedure is used by an Attribute Client to discover service relationships to other services.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Find Included Services | Read By Type Request<br>Read By Type Response<br>Error Response |

**Table 4-4: Relationship Discovery Sub-Procedure**

The “Find Included Services” is used by the client to find include service declarations in the Attribute server database.

The function is completed in two ways: either the application receives an error code (Attribute not found) or the Read by Type Response has enough unused data to contain another result indicating that no further results exist.

Insufficient authentication error or read not permitted error shall not occur (Include declaration is readable and requires no authentication or authorization).

### 4.3.4 Characteristic Discovery

The procedure is used by an Attribute Client to discover service characteristics present on the Attribute Server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Discover All Characteristic of a Service | Read By Type Request<br>Read By Type Response<br>Error Response |
| Discover Characteristic by UUID | Read By Type Request<br>Read By Type Response<br>Error Response |

**Table 4-5: Characteristic Discovery Sub-Procedures**

The “Discover All Characteristic of a Service” is used to find all the characteristic declarations within a service definition on a server when only the service handle range is known.

The “Discover Characteristic by UUID” is used to discover service characteristics on the Attribute Server when only the service handle ranges and characteristic UUID are known.

The functions are completed in two ways: either the application receives an error code (Attribute not found) or the application cancels the search (in case the desired characteristic is already found).

Insufficient authentication error or read not permitted error shall not occur (Characteristic declaration is readable and requires no authentication or authorization).

### 4.3.5 Characteristic Descriptor Discovery

The procedure is used by an Attribute Client to discover characteristic descriptors of a characteristic.

| Sub-Procedure | ATT Operation Code |
|---|---|

| | |
|---|---|
| Discover All Characteristic Descriptors | Find Information Request<br>Find Information Response<br>Error Response |

**Table 4-6: Characteristic Descriptor Discovery Sub-Procedure**

The “Discover All Characteristic Descriptors” is used by a client to find all the attribute handles and types of the characteristic descriptor within the characteristic definition, and only when the handle range is known.

The function is completed in two ways: either the application receives an error code (Attribute not found) or the application cancels the search (in case the desired characteristic descriptor is already found).

### 4.3.6 Characteristic Value Read

The procedure is used by an Attribute Client to read a Characteristic Value from a server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Read Characteristic Value | Read Request<br>Read Response<br>Error Response |
| Read Using Characteristic UUID | Read By Type Request<br>Read By Type Response<br>Error Response |
| Read Long Characteristic Values | Read Blob Request<br>Read Blob Response<br>Error Response |
| Read Multiple Characteristic Value | Read Multiple Request<br>Read Multiple Response<br>Error Response |

**Table 4-7: Characteristic Value Read Sub-Procedures**

The “Read Characteristic Value” is used to read a Characteristic Value from a server when the client knows the Characteristic Value Handle.

The “Read Using Characteristic UUID” is used to read a Characteristic Value from a server when the client only knows the Characteristic UUID and does not know the handle of the characteristic.

The “Read Long Characteristic Values” is used to read a Characteristic Value from a server when the client knows the Characteristic Value Handle and the length of the Characteristic Value is longer than can be sent in a single Read Response Attribute Protocol message.

The “Read Multiple Characteristic Values” is used to read multiple characteristic values from an Attribute Server when the client knows the Characteristic Value Handles.

**Note**: Read Blob means reading a specific part of an attribute, starting from an offset and until end of attribute value or MTU size.

### 4.3.7 Characteristic Value Write

The procedure is used by the Client to write a Characteristic Value to an Attribute Server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Write Without Response | Write Command |
| Signed Write Without Response | Write Command |
| Write Characteristic Value | Write Request<br>Write Response<br>Error Response |
| Write Long Characteristic Values | Prepare Write Request<br>Prepare Write Response<br>Execute Write Request<br>Execute Write Response<br>Error Response |
| Characteristic Value Reliable Writes | Prepare Write Request<br>Prepare Write Response<br>Execute Write Request<br>Execute Write Response<br>Error Response |

**Table 4-8: Characteristic Value Write Sub-Procedures**

The “Write Without Response” is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the client does not need an acknowledgement that the write was successfully done.

The “Signed Write without Response” is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the ATT Bearer is not encrypted.

The “Write Characteristic Value” is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle.

The “Write Long Characteristic Values” is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle but the length of the Characteristic Value is longer than can be sent in a single Write Request Attribute Protocol message.

The “Characteristic Value Reliable Writes” is used to write a Characteristic Value to an Attribute Server when the client knows the Characteristic Value Handle, and assurance is required that the correct Characteristic Value is going to be written by transferring the Characteristic Value to be written in both directions before the write is performed.

### 4.3.8 Characteristic Value Notification

The procedure is used to notify a client of the value of a Characteristic Value from a server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Notifications | Handle Value Notification |

**Table 4-9: Characteristic Value Notification**

The “Notifications” sub-procedure is used when a server is configured to notify a Characteristic Value to a client without expecting any Attribute Protocol layer acknowledgement that the notification was successfully received.

### 4.3.9 Characteristic Value Indication

The procedure is used to indicate the Characteristic Value from a server to a client.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Indications | Handle Value Indication |
| | Handle Value Confirmation |

**Table 4-10: Characteristic Value Indication Sub-Procedure**

The “Indications” sub-procedure is used when a server is configured to indicate a Characteristic Value to a client and expects an Attribute Protocol layer acknowledgement that the indication was successfully received.

### 4.3.10 Characteristic Descriptor Value Read

The procedure is used to read characteristic descriptor on a server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Read Characteristic Descriptors | Read Request |
| | Read Response |
| | Error Response |
| Read Long Characteristic Descriptors | Read Blob Request |
| | Read Blob Response |
| | Error Response |

**Table 4-11: Characteristic Descriptor Value Read Sub-Procedures**

The “Read Characteristic Descriptor Value Read” sub-procedure is to read a characteristic descriptor from a server when the client knows the Attribute handle of the Characteristic declaration.

The “Read Long Characteristic Descriptors” sub-procedure is used to read a characteristic descriptor from a server when the client knows the Attribute handle of the characteristic descriptor declaration and the length of the characteristic Value is more than fit for a single Read Response Attribute Protocol message.

### 4.3.11 Characteristic Descriptor Value Write

The procedure is used to write characteristic descriptor on a server.

| Sub-Procedure | ATT Operation Code |
|---|---|
| Write Characteristic Descriptors | Read Request |
| | Read Response |
| | Error Response |
| Write Long Characteristic Descriptors | Read Blob Request |
| | Read Blob Response |
| | Error Response |

**Table 4-12: Characteristic Descriptor Value Write Sub-Procedures**

The “Write Characteristic Descriptors” sub-procedure is used to write a characteristic descriptor value to a server when the client knows the characteristic descriptor handle.

The "Write Long Characteristic Descriptors" sub-procedure is used to write a characteristic descriptor value to a server when the clients knows the characteristic descriptor handle of the characteristic descriptor declaration and the length of the characteristic Value is more than fit for a single Write Response Attribute Protocol message.

## 4.4 Service Discovery Procedure

The service discovery should be generic feature used by client profiles in order to discover peer device database. By using a generic method of service discovery it prevent from code duplication in client profiles.

This discovery should be performed for all services type or only for some of them.

With this feature, application can decide if discovery is performed by client profiles or by application itself.

For each discovered services, this procedure is in charge of finding included services, characteristics and descriptors.

When a full service is discovered, this operation triggers a message containing all information.

**Note**: Since this procedure can be very long, it could be aborted by application through a Cancel API.

**Note**: The client profile should verify if peer device service can be used by its implementation.


**Figure 4-48: Service Discovery procedure.**

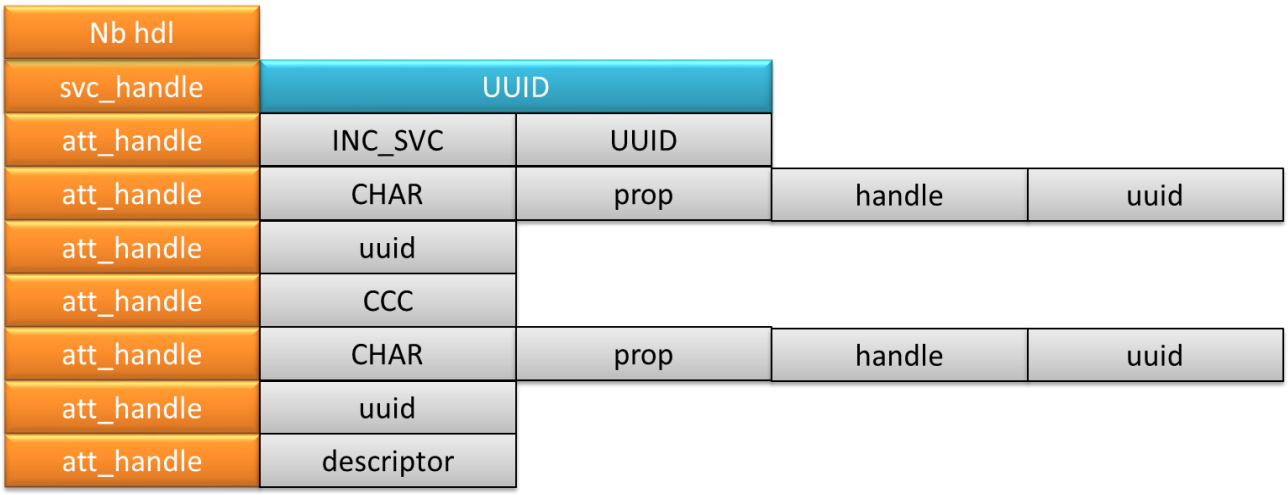



**Figure 4-49: Overview of information present in discovered service.**

## 4.5 GATT Profile Service

The GATT Profile service is a single-instantiated primary service which exposed on a GATT server.

The Profile Service has Service changed characteristic.

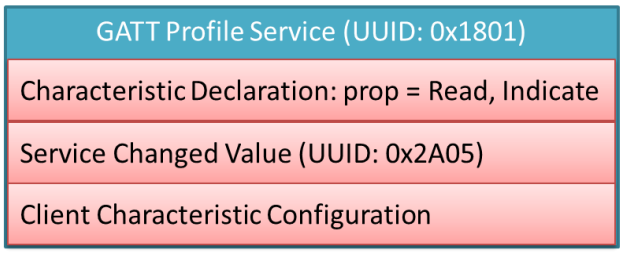



**Figure 4-50: GATT Profile Service**

The “Service Changed” Characteristic is a control point attribute that is used to notify connected devices that GATT services have been changed. The Value cannot be read nor written but can be notified at any time.

## 4.6 GATT Environment Variables

### 4.6.1 GATT Manager Environment

| Type | Value | Comment |
|---|---|---|
| **uint16_t** | svc_start_hdl | GATT service start handle |
| **uint16_t** | max_mtu | Maximum device MTU size |
| **attm_svc_db*** | db | Attribute database pointer |

| | | |
|---|---|---|
| **attm_svc_db*** | last_svc | Last attribute service searched. |

**Table 4-13: GATTM Environment variables**

### 4.6.2 GATT Controller Environment

| Type | Value | Comment |
|---|---|---|
| **ke_msg *** | Client Operation | Client Initiated operation |
| **ke_msg *** | Service Operation | Service Initiated operation (notification indication) |
| **ke_msg *** | SDP Operation | Operation used for Service Discovery Procedure |
| **uint16_t** | mtu_size | Size of attribute protocol MTU |
| **co_list** | cli_reg_evt | List that contains task to inform when an event is trigger on a specific attribute handle range |
| **co_list** | cli_rsp_list | List of messages received used to generate response indication |
| **l2cc_pdu *** | srv_req | Request that service is currently processing |
| **co_list** | srv_prep_wr_list | List of prepare write message received from peer client |
| **gattc_read_cfm *** | srv_read_cache | Structure is used to store in cache latest attribute read value |
| **co_list** | srv_rsp_list | List of values used to create response |
| **co_list** | SDP Data | List that contains service discovery procedure data |

**Table 4-14: GATTC Environment variables**

# 5 Generic Access Profile Functionalities

This profile states the requirements on names, values and coding schemes used for names of parameters and procedures experienced on the user interface level.

This profile describes the general procedures that can be used for establishing connections to other Bluetooth devices that are able to accept connections and service requests.

GAP defines two parties (A and B) in establishing Bluetooth communication.

> A-Party: the device that is either scanning in the link layer scanning state or initiating in the link layer initiating state.
>
> B-Party: the device that is either advertising in the link layer advertising state or accepting the link request.

The GAP allows minimal functionality in absence of other profiles and provides an API when other profiles are present (see [5]).

## 5.1 Modes and Profile Roles

GAP introduces three device types based on supported Core Configurations.

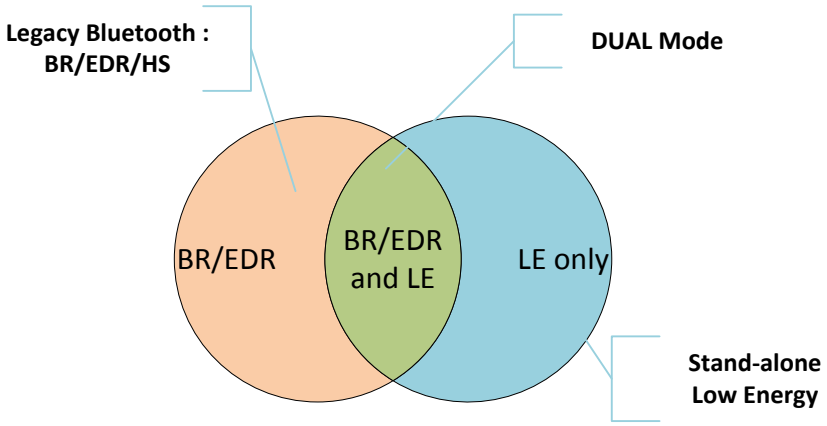



**Figure 5-1: Device Types**

Devices of type LE-only and BR/EDR/LE are capable of operating over an LE physical channel.

**Note: Our implementation of Generic Access Profile supports only LE Only mode.**

| | Non Discoverable | Discoverable | |
|---|---|---|---|
| | | Limited Discoverable | General Discoverable |
| **Non Connectable** | Not advertising | Non connectable limited advertising | Non connectable general advertising |
| **Connectable** | Connectable directed advertising | Connectable limited advertising | Connectable general advertising |

**Table 5-1: Discoverability and Connectability Modes to Advertising Capability**

Moreover, GAP defines different modes of operation which are generic and can be used by profiles and by devices implementing multiple profiles.

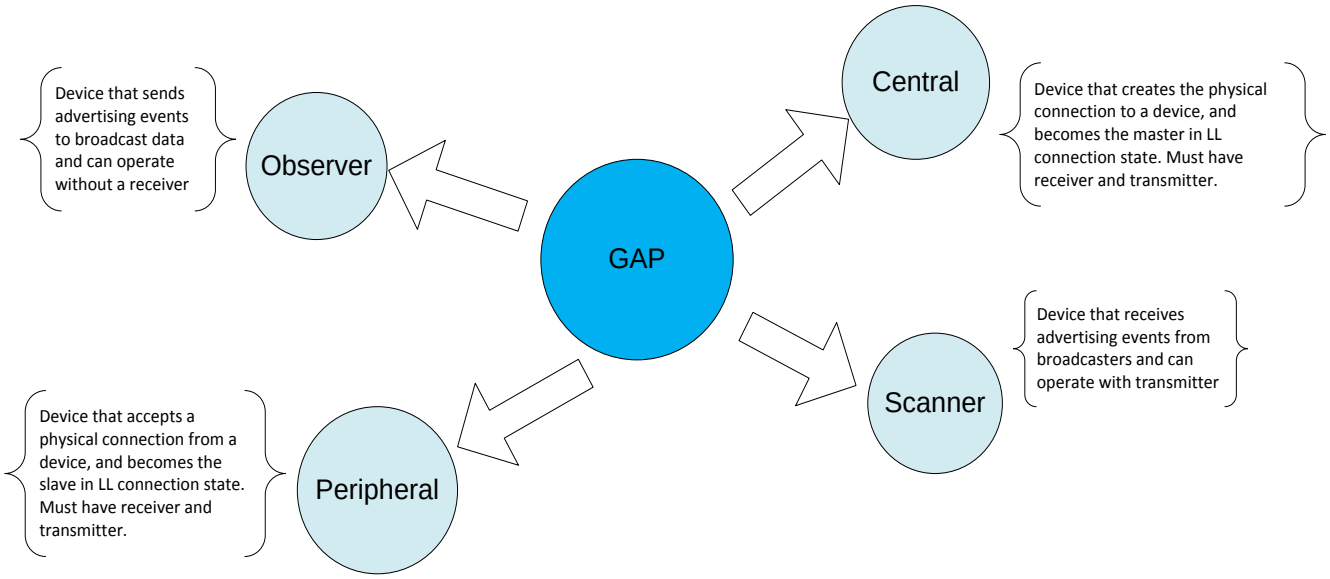



**Figure 5-2: GAP Roles**

In addition of Figure 5-2, Peripheral is able to broadcast data, and a Central is able to enter in observable mode.

A device can support all roles in same time, so that it can act both as a central (scan + master of a link) and peripheral (advertise + slave of a link).

A device supporting all modes cannot start two non-connected operation (such as advertising, scanning or connection init) in same time.

## 5.2 General LE Procedures

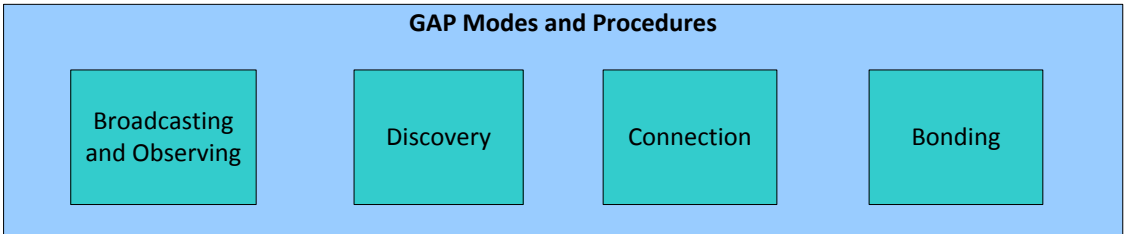



**Figure 5-3: LE Operational Modes**

GAP defines the general procedures that can be used for discovering identities, names and basic capabilities of other BLE devices that are discoverable. It also describes the ability of a device to be connected and discovered by another device.

### 5.2.1 Broadcasting and Observing

The broadcast and observe modes allow two devices to communicate in a unidirectional and connectionless manner using advertising events.

#### 5.2.1.1 Conditions

A broadcaster is a device operating in broadcast mode. It sends data in either non-connectable undirected or discoverable undirected advertising events. All data sent by a broadcaster is considered unreliable since there is no acknowledgement from any device that may have received data. No support for encryption.

An observer is a device operating in scan mode. It uses either passive or active scanning in receiving advertising events. No support for encryption.

### 5.2.2 Advertising modes

A device can perform an Advertising procedure in a connectable or non-connectable mode. White list can be used to filter device that can receive scan response and initiate a connection.

**Note**: When this operation is on-going application can modify advertise and scan response data to update ongoing broadcasted data.

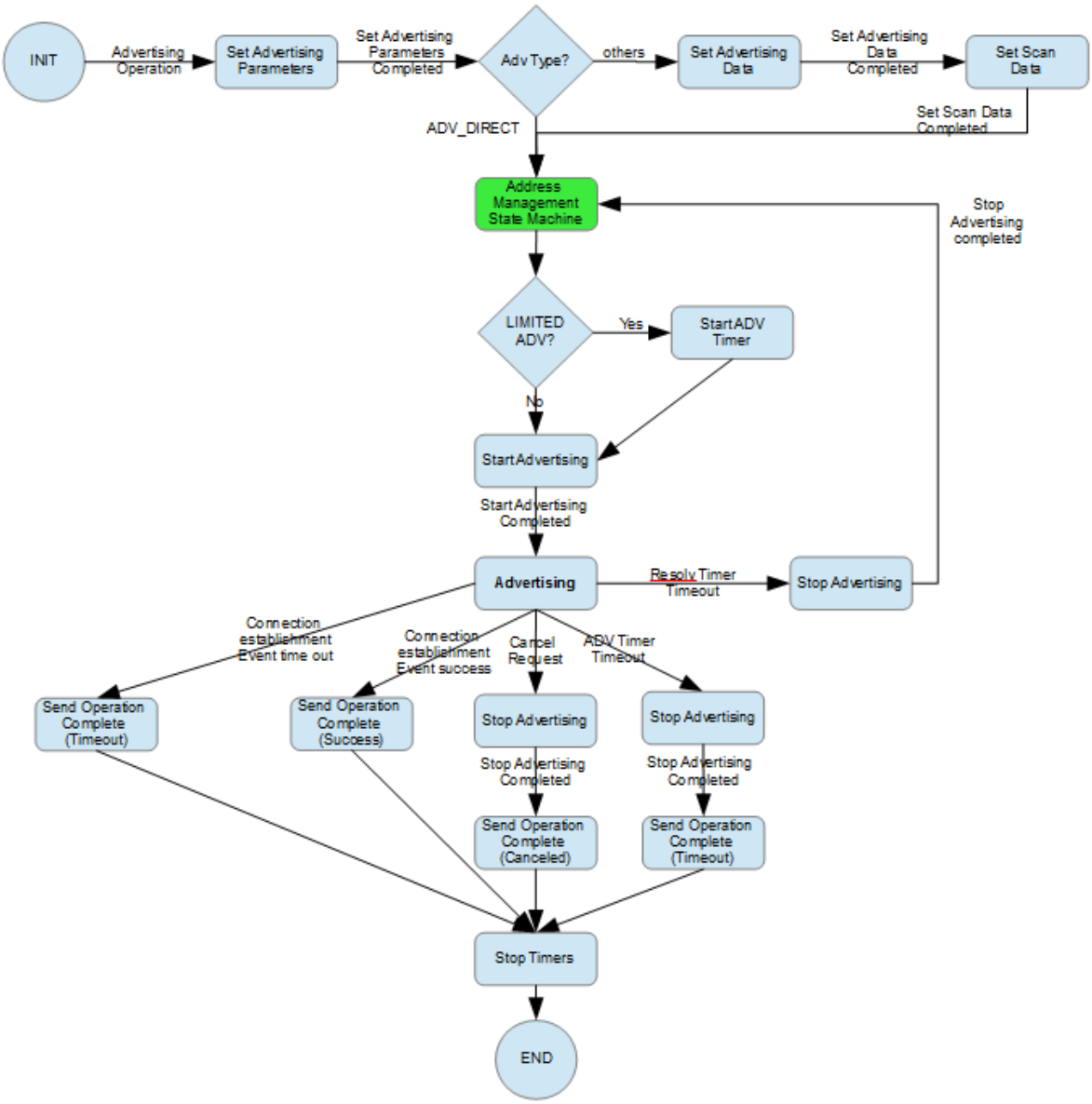



**Figure 5-4: Advertise Air Operation State Machine.**

#### 5.2.2.1 Broadcast Mode

The broadcast mode is like non-discoverable mode. In AD_TYPE flag of advertising data, LE General and LE Limited discoverable flag are set to zero.

**Note**: This is the only mode that can be used by a Broadcaster device.

#### 5.2.2.2 Non Discoverable Mode

Non discoverable mode is a connectable or non-connectable procedure without duration limitation.

In AD_TYPE flag of advertising data, LE General and LE Limited discoverable flag are set to zero

#### 5.2.2.3 General Discoverable

General discoverable mode is a connectable or non-connectable procedure without duration limitation.

In AD_TYPE flag of advertising data, LE General is set to 1 and LE Limited discoverable flag is set to zero

#### 5.2.2.4 Limited Discoverable

Limited discoverable mode is a connectable or non-connectable procedure with a limited duration.

In AD_TYPE flag of advertising data, LE General is set to zero and LE Limited discoverable flag is set to 1

#### 5.2.2.5 Direct Mode

Direct mode is used to perform a direct connection. Advertising data contains only the targeted device.

Advertising data cannot be dynamically changed in this mode.

### 5.2.3 Scan modes

#### 5.2.3.1 Device Discovery

The device discovery has two parts: **procedures and modes**.

A device that is searching for other devices performs one of the discovery procedures. A device that is the target of the search is operating in one of the discoverable modes. A device in the Non-Discoverable mode is configured to not be discovered. All devices are in either Non-Discoverable mode or one of the Discoverable modes (General and Limited).

A typical example of a device that need not be in discoverable mode is an observer. Device that operates in an observer profile role requires no transmitter.

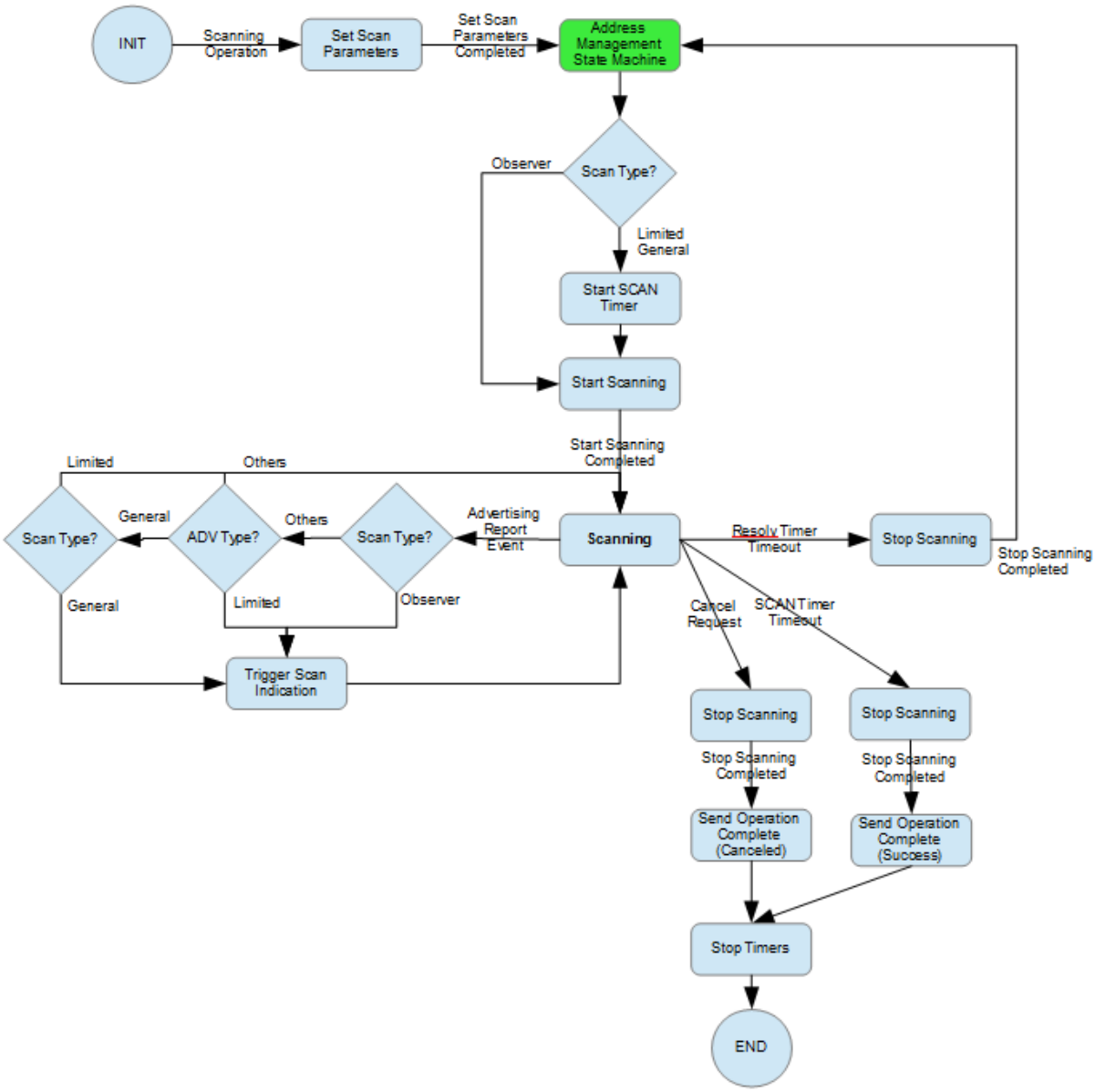



**Figure 5-5: Scan Air Operation State Machine.**

#### 5.2.3.2 Observer Mode

The Observer mode is a passive or an active scan procedure with non-limited duration. In this mode, notify application of any type of advertising data.

**Note**: This is the only mode that can be used by an Observer device.

#### 5.2.3.3 General Discovery

The General discovery is a passive or an active scan procedure with a limited duration. In this mode, device is able to discover advertiser that broadcast data in limited or general discoverable mode.

#### 5.2.3.4 Limited discovery

The Limited discovery is a passive or an active scan procedure with a limited duration. In this mode, device is able to discover advertiser that broadcast data in limited discoverable mode.

#### 5.2.3.5 Name Discovery

Other aspect of discoverability is device name discovery, wherein the user-friendly name of the remote device is retrieved. This is performed by a device that can scan remote connectable devices – a Central.

The discovery procedure involves three fundamental steps:

1. Search and connect to a connectable device (Advertising device)
2. Perform Read by Characteristic UUID (Device Name: 0x2A00)
3. Terminate the link

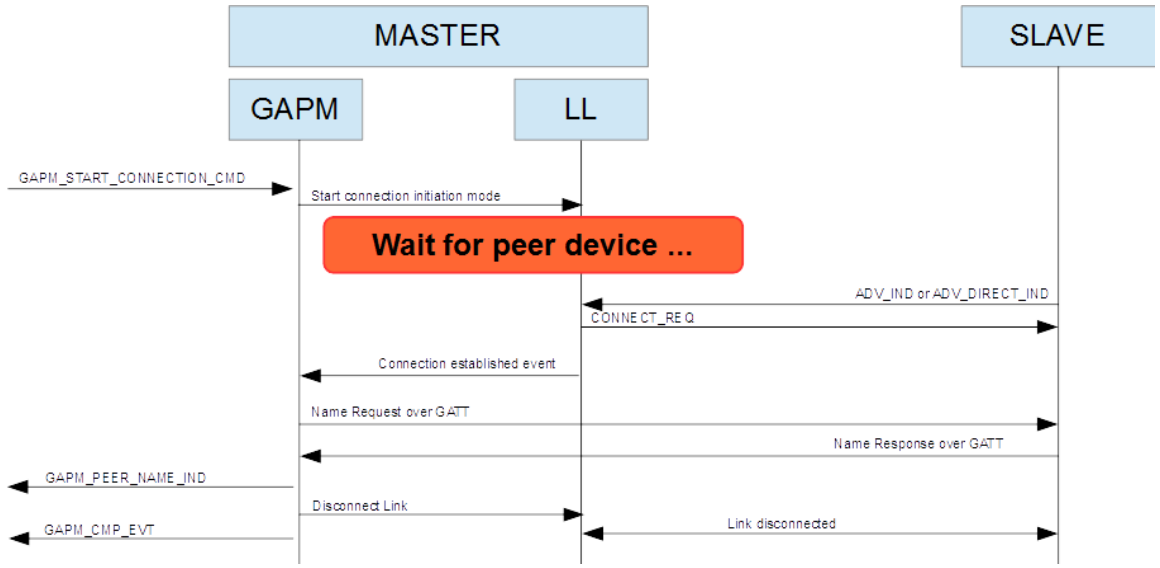



**Figure 5-6: Name Request procedure**

### 5.2.4 Connection

There are two modes of LE connections defined in GAP.

Connectable: permits a device to make connections to or accept connections from another device.

Non-connectable: prohibits a device accepting connections from another device.

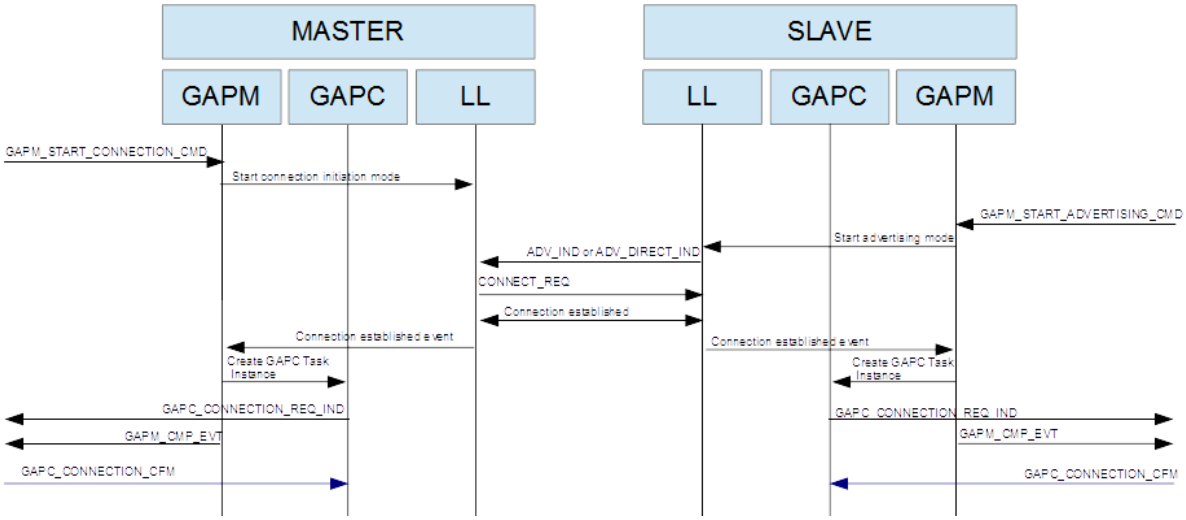



**Figure 5-7: Connection establishement overview**

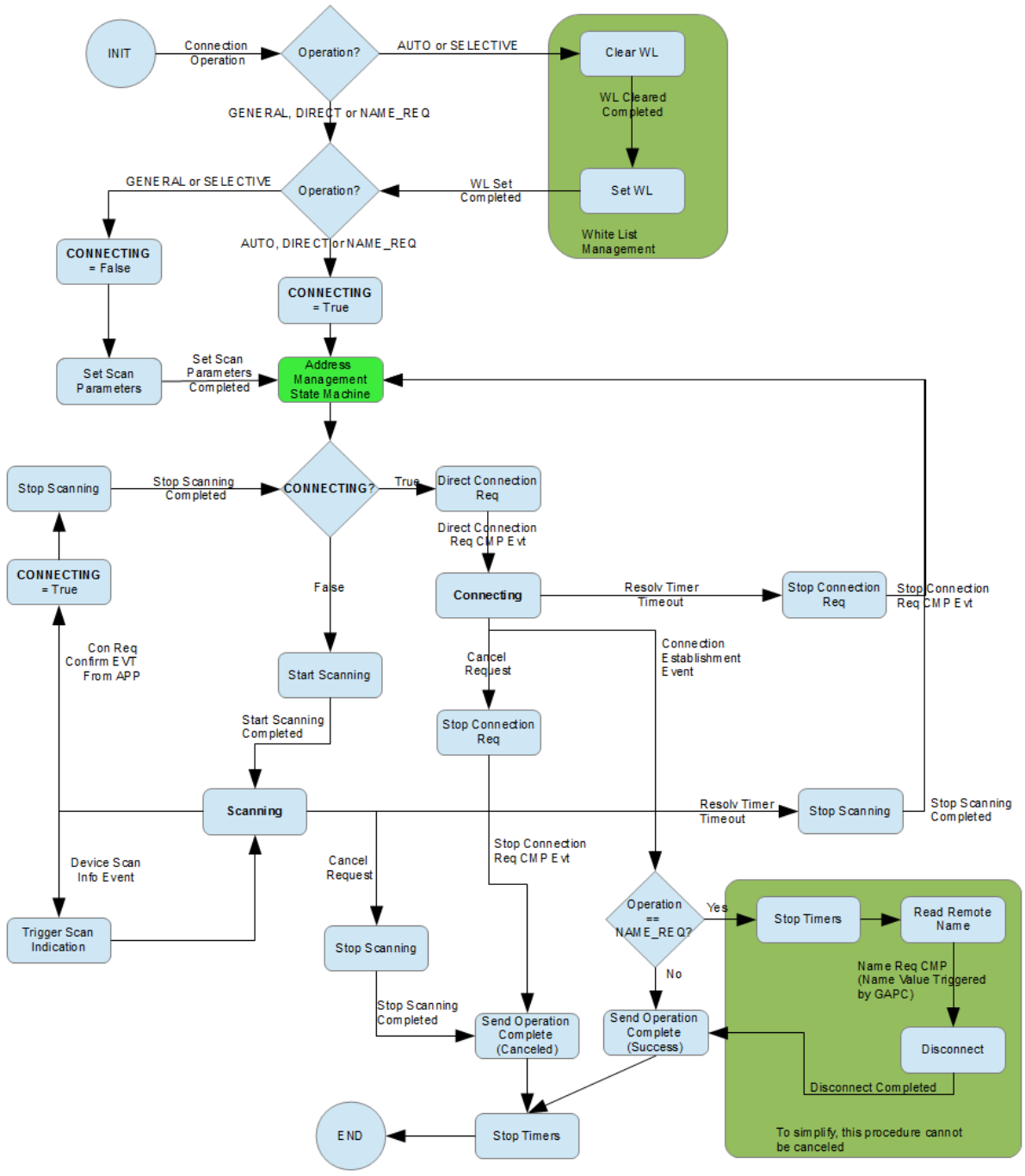



**Figure 5-8: Connection establishement state machine**

#### 5.2.4.1 Direct Connection Establishment

To be able to establish link between devices, one device must be in connectable mode, and the other device would be performing connection establishment procedure.

**Figure 5-9: Direct connection procedure**

#### 5.2.4.2 General Connection Establishment

Use General discovery procedure and then direct connection establishment procedure to perform a general connection establishment.

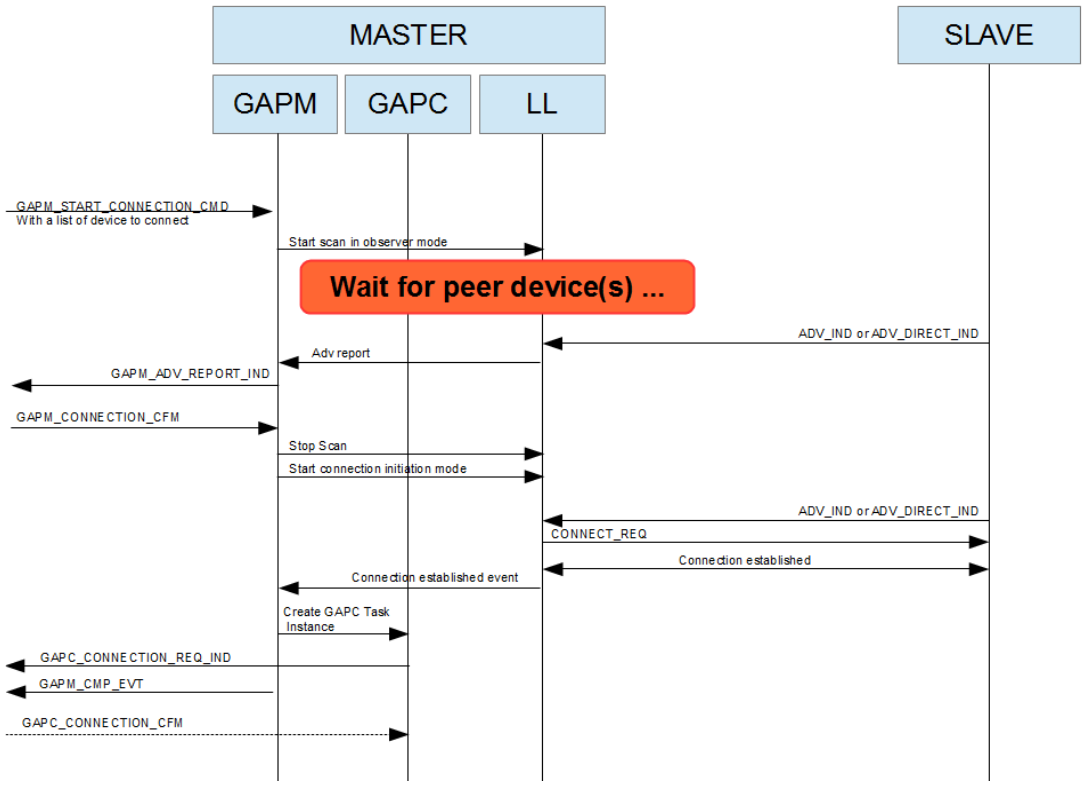



**Figure 5-10: General connection procedure**

#### 5.2.4.3 Automatic Connection Establishment

Automatic connection establishment procedure uses white list in connection mode in order to find any known device.

As soon as a known device is found, it uses a direct connection in order to connect to the peer device.

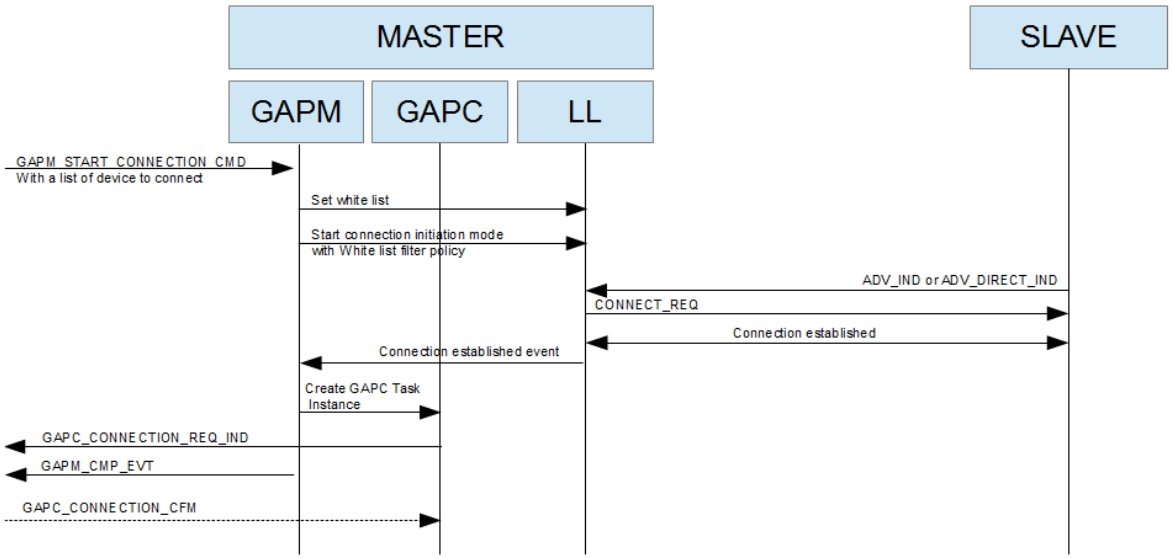



**Figure 5-11: Automatic connection procedure**

**Note:** When device is in this mode, it's not possible to modify white list.

#### 5.2.4.4 Selective Connection Establishment

Automatic connection establishment procedure uses white list in order and observer mode in order to find any known device.

As soon as a known device is found, application is notified and should reply with some connection parameters to use with this device,

Finally, it uses a direct connection in order to connect to the peer device.

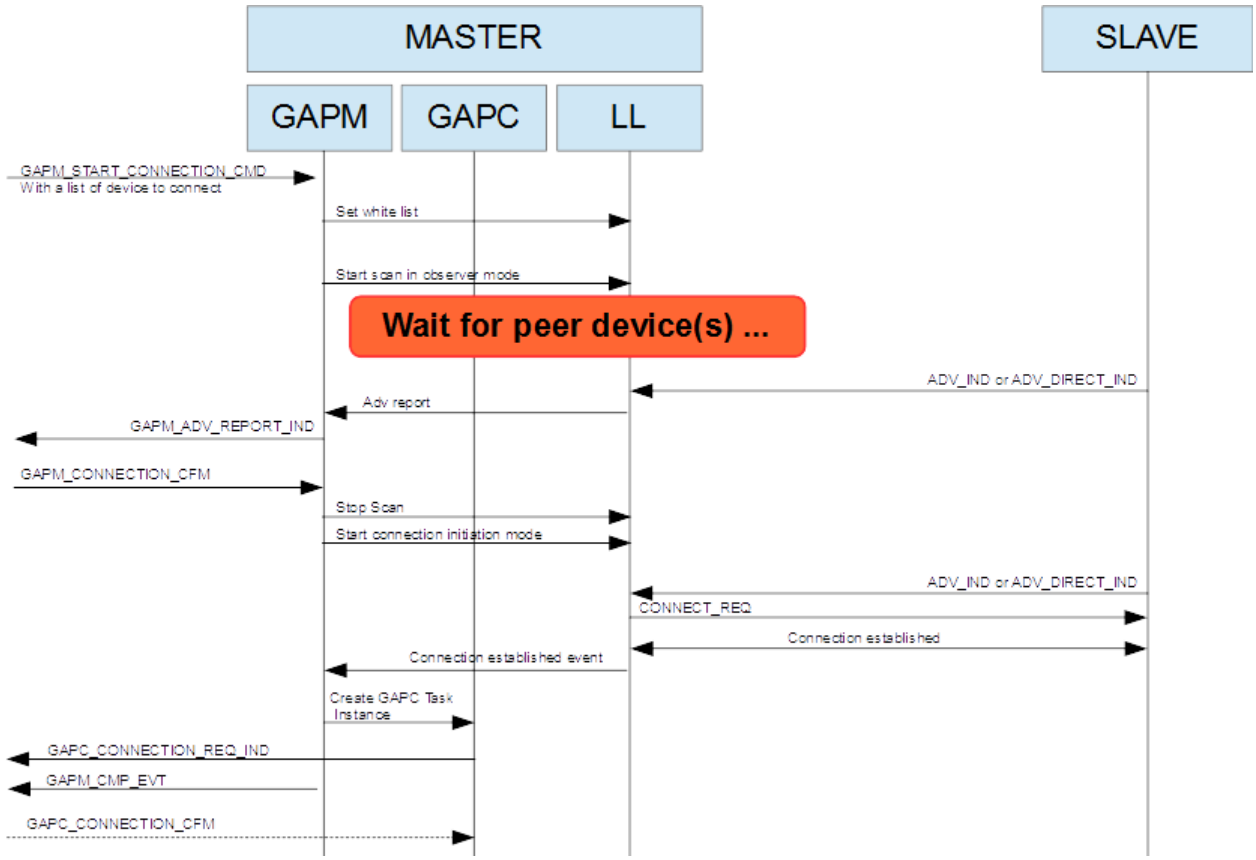



**Figure 5-12: Selective connection procedure**

**Note:** When device is in this mode, it's not possible to modify white list.

#### 5.2.4.5 Update connection Parameters

Operation that can be started over a LE link, Parameter update procedure is used to update link parameters.

If operation initiated by master on 4.0 (Legacy) devices (see Figure 5-13) there is no link negotiation and new link parameter are automatically applied.

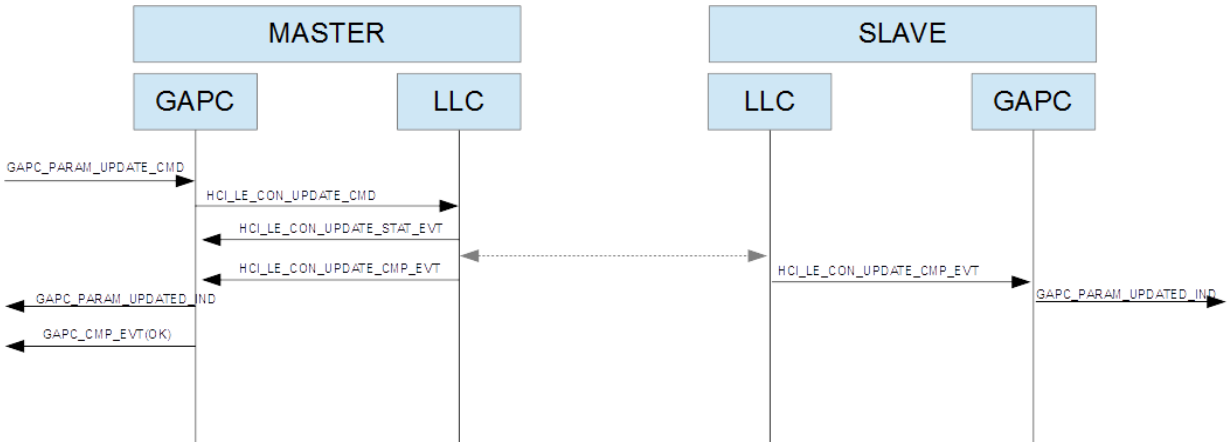



**Figure 5-13: Parameter Update initiated by Master**

When a slave initiate a connection parameter update without knowing remote features, first start a parameter update through HCI and if it fails due to following reason, use legacy negociation over L2CAP (see Figure 5-14):

- Unknown HCI Command
- Command Disallowed
- Unsupported Command
- Unknown LMP PDU
- Unsupported Remote feature
- LMP Pdu Not Allowed

**Note:** Operation completion message for a legacy parameter update initiated by slave (on slave device) have to be triggered before GAPM_PARAM_UPDATE_IND to ensure that master device did not use connection param request.

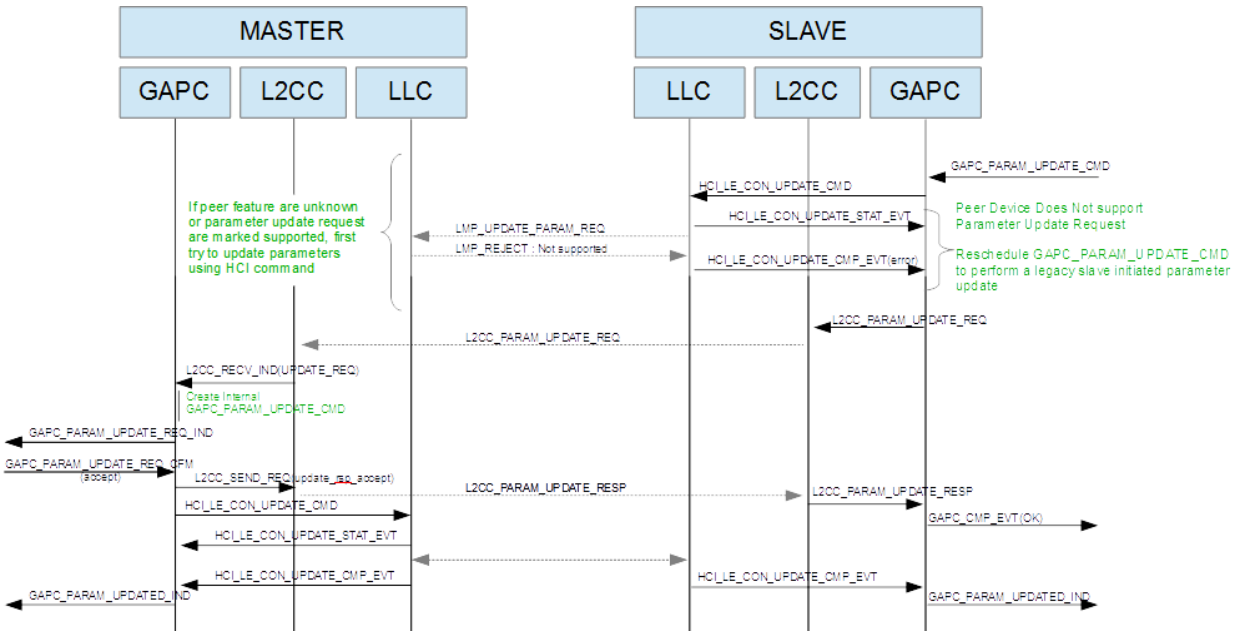



**Figure 5-14: Legacy Parameter Update initiated by Slave**

Figure 5-15 shows legacy parameter update negotiation initiated by slave and rejected by master.

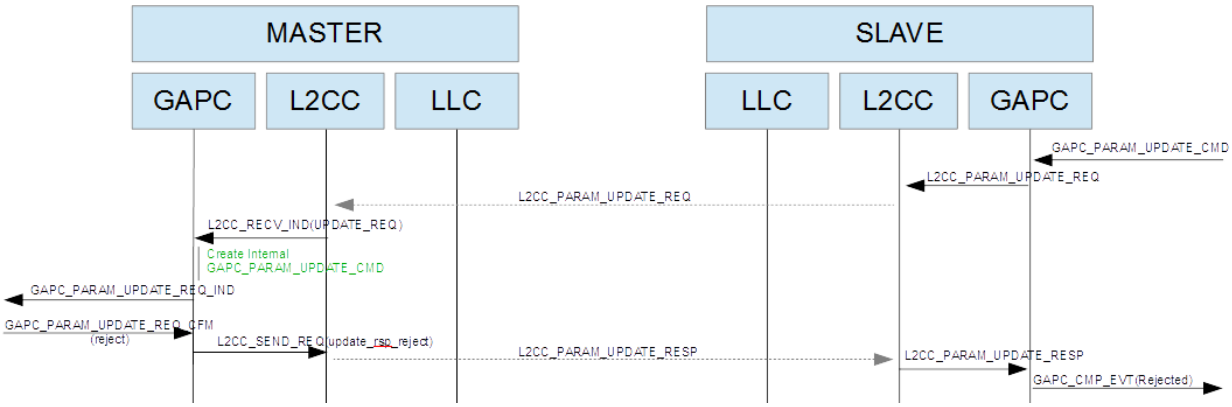



**Figure 5-15: Legacy Parameter Update initiated by Slave, rejected by Master**

If Parameter update request is supported by peer device, Connection parameter initiated by master or slave is a little bit different. It is requested for peer device to accept or reject new parameters. This Parameter update is only performed through LLCP even if it's initiated by slave or master of the link.

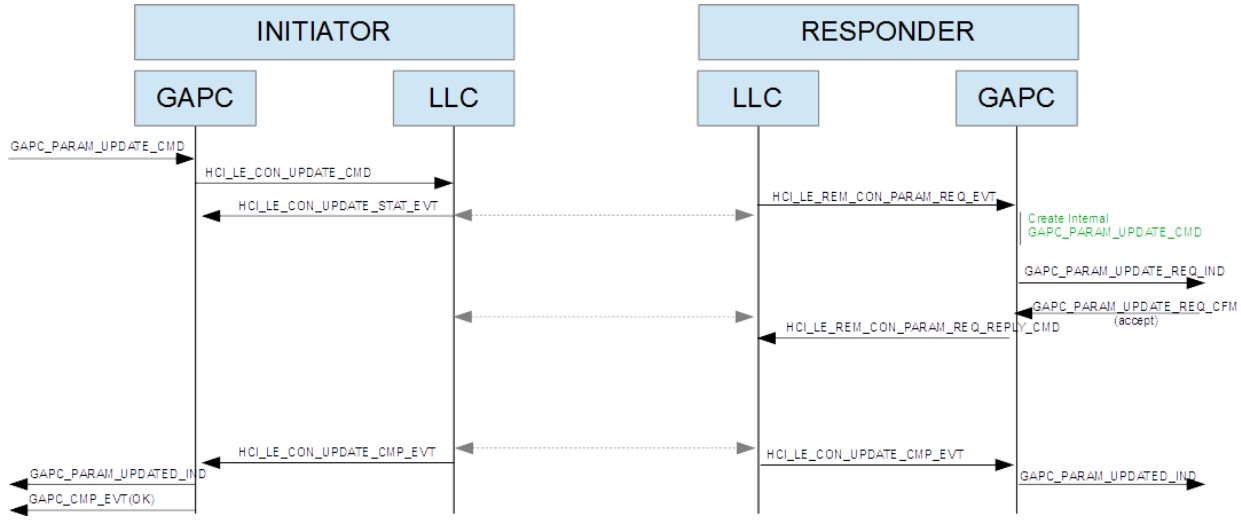



**Figure 5-16: Parameter Update with remote update request support accepted by responder**

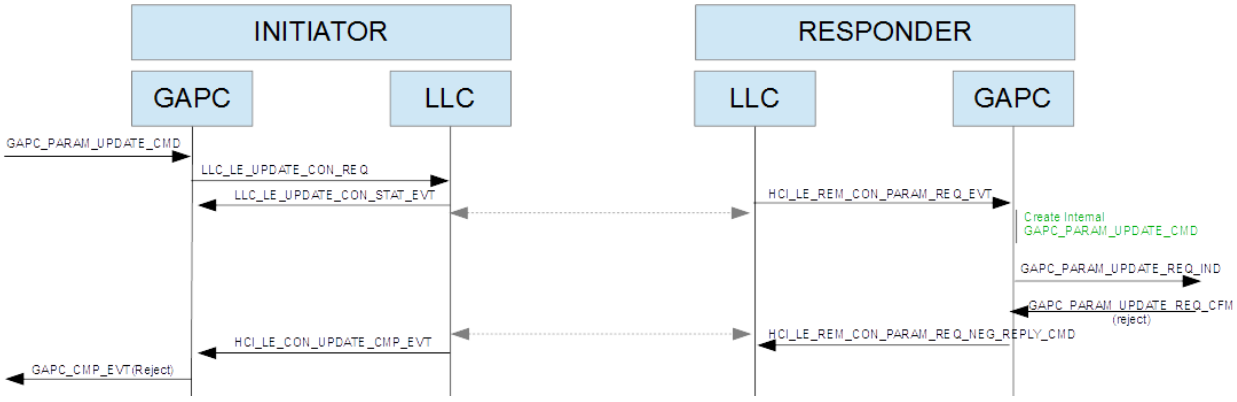



**Figure 5-17: Parameter Update with remote update request support rejected by responder**

### 5.2.5 Bonding

Bonding is the function where devices exchange and store security and identity information to create a secure relationship. It occurs at the first connection between devices or the first service that requires security or authorization.

Two types of bonding procedures are defined:

- Dedicated Bonding is when the user initiates SM pairing with the explicit purpose of creating a bond (i.e., a secure relationship) between two devices.
- General Bonding is when the user is requested to pair before accessing a service since the devices did not share a bond beforehand.

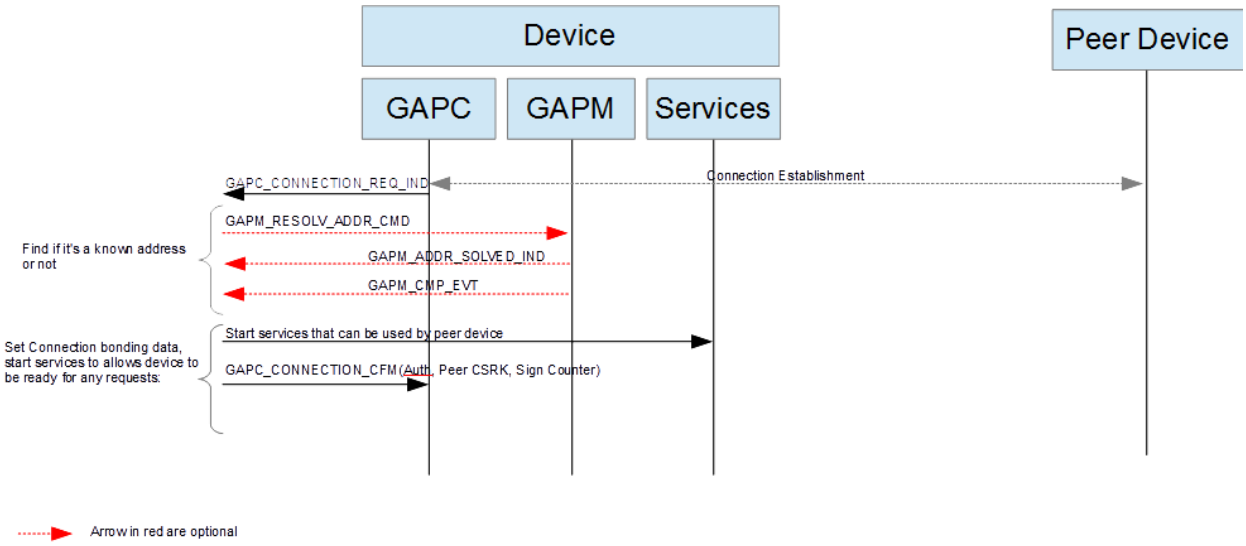



**Figure 5-18: Connection establishment with a known device (recover bond data)**

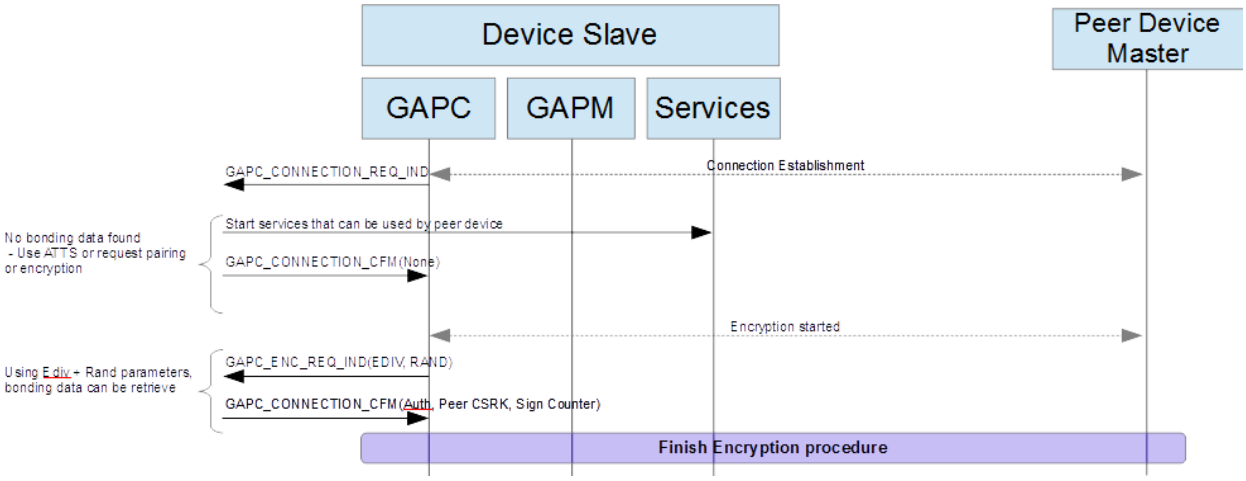



**Figure 5-19: Recover bond data of a peer device with a random address.**

## 5.3 Low Energy Security

Security mode and level defines the safety requirements of a device or access to services offered by the device.

### 5.3.1 Security Modes

The LE security is expressed in modes and levels. There are two security modes: Sec 1 (encryption) and Sec 2 (signing).

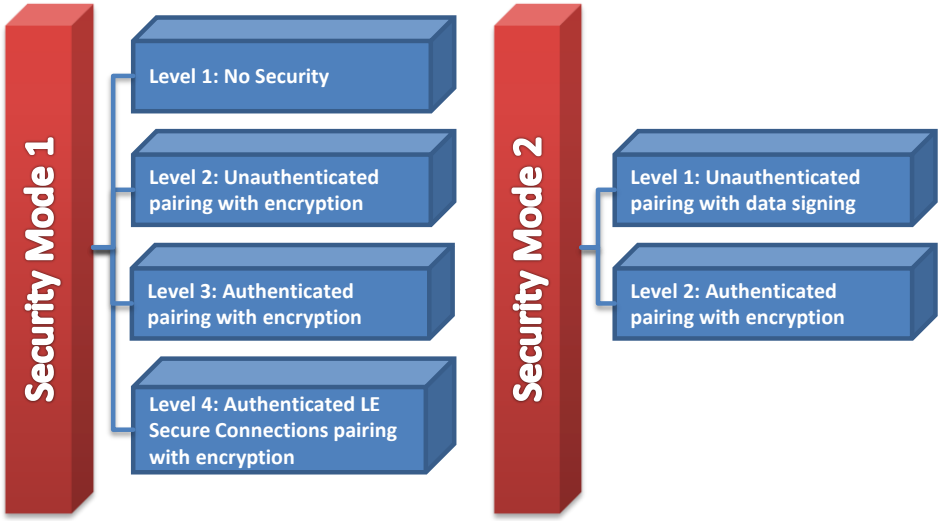



**Figure 5-20: LE Security Modes**

### 5.3.2 Authentication Procedure

The procedure pertains to satisfying the security requirements of the connecting devices when service request is initiated on either side. The authentication procedure is only valid after establishing LE link.

There are two types of pairing.

- Authenticated Pairing – Perform pairing procedure with authentication set to MITM protection
- Unauthenticated Pairing – Perform pairing procedure with authentication set to No MITM protection

### 5.3.3 Authorization Procedure

The procedure allows the continuation of service access by remote device. This is a confirmation by the user to continue with the procedure. Authorization may be granted after successful authentication.

### 5.3.4 Data Signing

The procedure is used to transfer authenticated data between two devices in an unencrypted connection. This is used by services that require fast connection setup and data transfer. If this is used, Security mode 2 is a must.

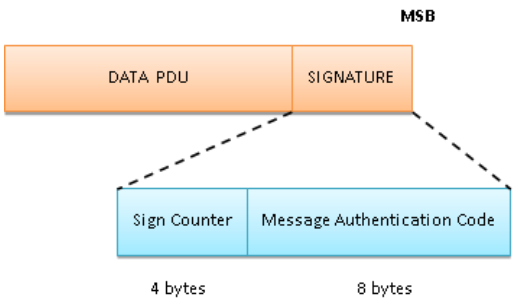



**Figure 5-21: Packet Signature**

### 5.3.5 Privacy

#### 5.3.5.1 Host managed Privacy (1.1)

The host managed privacy feature provides specific level of security from an attacker to track an LE device over a certain period of time. This is an optional feature for all GAP roles.

| | Broadcast | Observer | Central | Peripheral |
|---|---|---|---|---|
| **Privacy Off** | Public or Static | Public or Static | Public or Static | Public or Static |
| **Privacy On** - Connectable | N/A | N/A | Resolvable | Resolvable |
| **Privacy On** - Non Connectable | Resolvable or Non-Resolvable | Resolvable or Non-Resolvable | Resolvable or Non-Resolvable | Resolvable or Non-Resolvable |

**Table 5-2: Device address type according to privacy configuration**

**Note:** For passive scan, Privacy feature is ignored

**Note:** If device has ***all roles***, it cannot use both Resolvable address for an air activity and Non-Resolvable address for another air activity. A privacy error will be triggered in that case.

#### 5.3.5.2 Controller managed Privacy (1.2)

With controller managed privacy, the application should set the resolving address list (RAL) (see [1] Vol. 1 Part. A Chapter 5.4.5) using GAPM_RAL_MGMT_CMD command. This resolving address list can be managed like a white list and is used in complement of the white list.

When the controller managed privacy is enabled, scan, advertise and initiating parameters are set to use the resolving list.

If this feature is enabled when setting device configuration (see 5.11.2) then Central Address Resolution characteristic becomes present in the GAP service (see 5.9)

#### 5.3.5.3 LE Address

There are two types of Bluetooth LE addresses

1. Static Address
    - Two most significant bits are equal to 1
    - All the other bits are not “all 0s” nor “all 1s”
2. Private Address
    1. Non-resolvable Address
        - Two most significant bits are equal to 0
        - All the other bits are not “all 0s” nor “all 1s”
    2. Resolvable Address
        - Two most significant bits are equal to 01
        - 22 remaining bits of prand are not “all 0s” nor “all 1s”
        - 24 bits hash part is derived from IRK, prand and ah func

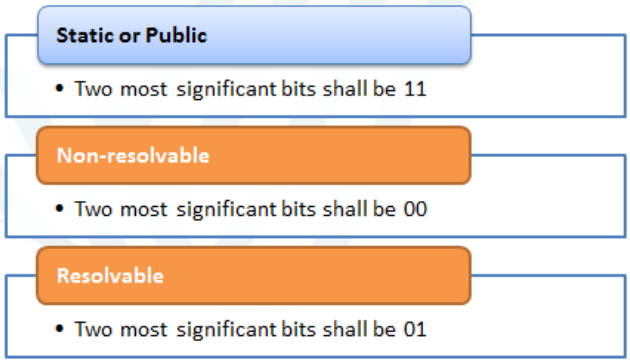



**Figure 5-22: LE Address**

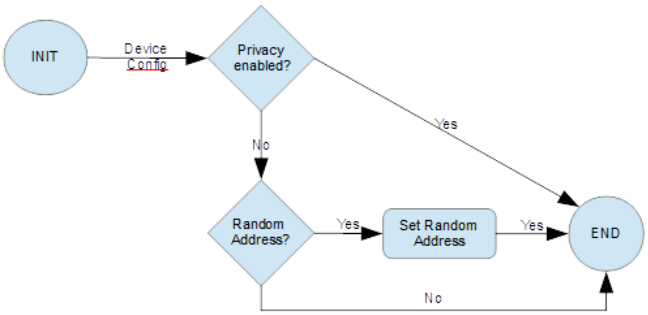



**Figure 5-23: Initialize device address FW state machine**

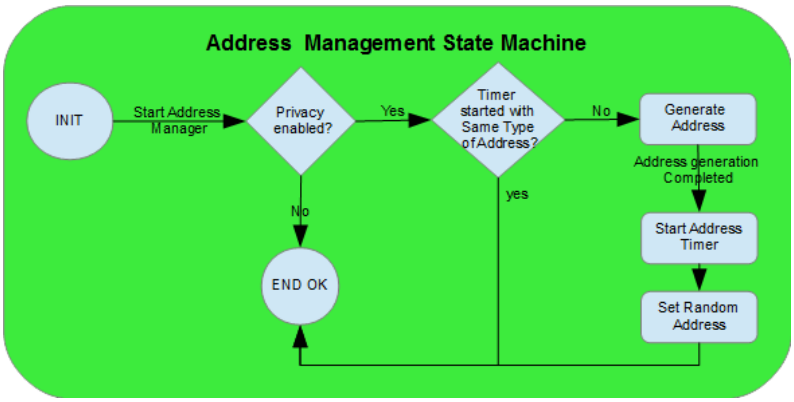



**Figure 5-24: Air operation address management FW state machine**

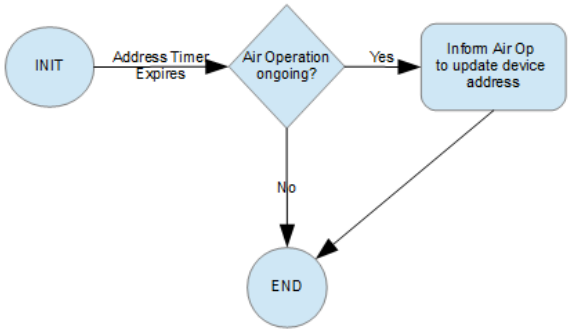



**Figure 5-25: Privacy address management FW state machine**

## 5.4 Security Manager Toolbox

The Bluetooth Low Energy Security Manager allows two devices to setup a secure relationship either by encrypting a link, by bonding (exchanging information about each other) or signature use over a plain link.

Please refer to the Bluetooth Core 4.0 specification [1] for the SM requirements and protocol methods.

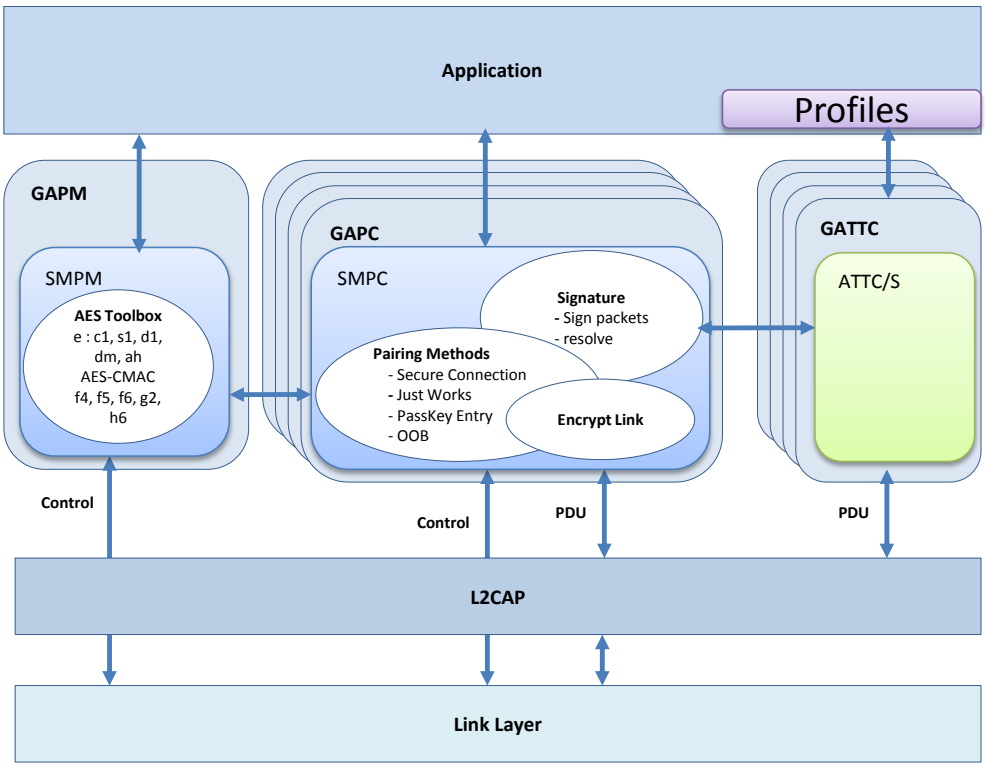



**Figure 5-26: SMP Block Overview**

A few key concepts must be presented for a clearer understanding of the SM:

- **Pairing**: this procedure allows two devices to agree upon features that will allow them to establish a certain level of security.
- **Bonding**: this procedure involves at least one device sending some sort of identification or security information to the other device, to be used in future connections. This may be an encryption key, signature key, identification resolution key. If both devices are bondable, the Transport Key Distribution Phase following pairing will occur. Otherwise, no bonding information will be exchanged and if any is sent, it is a violation of protocol. Pairing may occur without necessarily bonding, but the features exchanged during pairing are essential to the existence of a bonding stage. If not both devices are bondable, no information about the peer should be stored (not even BD address or other kind of non-security related information).
- **Unauthentication/Authentication**: Unauthentication is NOT lack of any security, but an intermediary level between no security and authenticated security level. The relationship between two devices is said to be (un)authenticated when the key(s) being used for their link encryption/signing/etc. have a security property that confirms (un)authentication. This security property is bestowed on a key during pairing, in function of the STK method generation used. For Either Passkey Entry and OOB Methods, all keys generated and exchanged afterwards have Authenticated (MITM) property (a pin key/ larger OOB key was used, this

enforces security). If the Just Works method was used, all keys will have Unauthenticated (No MITM) property. There may also be the No security property, which applies when the link is plain.

- **LE Secure Connection**: This pairing method allows having a greater security level than the normal pairing method. It uses Private/Public keys (P-256 elliptic curve) security algorithm in order to prevent any man in the middle attack. This secure connection is a fully new pairing method that can be used for just work pairing, OOB or pin code entry. With this method, except just work pairing, the security level of the link is considered as "Secure Connection Authenticated" link.

The Security Manager (SM) Toolbox is in charge of BLE secure communication issues: encrypted links, identity or private addresses resolution and signed unencrypted messages. The functionalities of the SM are enforced by clearly specified pairing and key distribution methods, and the protocol that is to be respected for their correct implementation. An additional cryptographic toolbox of functions based on the AES-128 algorithm supports key generation, private address generation and resolution and message signing and signature resolution.

The architecture decided for the implementation of the Security Manager is visible in Core Spec 4.1 Vol. 3 Part C, Chap. 10 [1]. Since the different functionalities may be required simultaneously for several connections a device may have, those functionalities have been implemented in the toolbox called SMPC: the Security Manager Protocol Controller. SMPC toolbox is available only using GAPC API.

However, certain higher and lower layer modules have an unique instance, handled By GAPM task through its API, SMPM Toolbox – Security Manager Protocol Manager – which will monitor SMPC's requests and responses without overloading those modules.

The dialogue between SMPM and SMPC's through GAPM and GAPC's API is limited to a few basic requests and responses. The communication between SMPCs, Higher and lower layers is much richer and also allows a device to proceed with link encrypting procedures at different stages with the different peers it possesses.

### 5.4.1 Keys Definition

There are several important types of keys in BLE security.

| Key Type | Description |
|---|---|
| Identity Root (IR) | ✓ 128-bit key generated for LE device<br>✓ Only for devices that support encryption or use random addresses<br>✓ Device can have multiple IR keys, but will only use one per connection<br>✓ Used to generate IRK and DHK |
| Encryption Root (ER) | ✓ 128-bit random generated<br>✓ Used to generate CSRK and LTK. |
| Identity Resolving Key (IRK) | ✓ 128-bit key<br>✓ Used to resolve random addresses |
| Diversifier Hiding Key (DHK) | ✓ 128-bit key<br>✓ Used to encrypt DIV during encryption connection setup |
| Connection Signature Resolving Key (CSRK) | ✓ 128-bit key<br>✓ Used to sign and verify signatures on the receiving device |
| Long Term Key (LTK) | ✓ 128-bit key, used partially depending on agreed key size<br>✓ Used to generate contributory session key for an encrypted connection |
| Diversifier (DIV) | ✓ 128-bit stored value, used to calculate LTK<br>✓ A new DIV is generated each time a unique LTK is distributed. |

| | |
|---|---|
| | ✓ The DIV value is masked to the 2 octet EDIV distributed value. |
| Short Term Key (STK) | ✓ Generated at the end of Phase 2 using TK<br>✓ Used to encrypt link after Phase 2 (according to agreed key size) |
| Temporary Key (TK) | ✓ Either 0, Pass Key or OOB depending on STK generation method<br>✓ Used to calculate STK |

**Table 5-3: BLE Keys**

### 5.4.2 Cryptographic Algorithms Overview

This part gives a fast overview about the cryptographic algorithms used in the SMP layer.

The understanding of theses algorithms is required to better understand what is done in the SMP layer.

***Notations***:

- $0^s$ is the bit string that consists of s '0' bits.
- The concatenation on bit strings is denoted ||.
- The function $LSB_s(X)$ returns the s least significant bits of X.

#### 5.4.2.1 Encryption Function e

The encryption function generates a 128-bit encrypted data from a 128-bit key and a 128-bit data using the AES-128-bit block implemented in the controller.

*encryptedData = e(key, data)*

Each time this function is supposed to be used in the SMP, a LLM_LE_ENC_CMD message is sent to the controller. The implementation of the stack implies that the answer LLM_LE_ENC_CMP_EVT is sent to SMPM.

#### 5.4.2.2 Keys Generation

The keys used in SMP are distributory, which means that the keys for distribution are made by the devices that dispense them.

The following security functions are used to generate SMP keys:

1. Encryption function ***e*** generates 128-bit *encryptedData* from a 128-bit key and 128-bit *plaintextData* using the AES-128 bit block cipher as defined in NIST Publication FIPS-197. This function is not implemented in the Host stack, as it makes use of the Controller feature allowing the encryption of a block of data by a specified key (HCI LE Encrypt command)

   *encryptedData = e* (*key*, *plaintextData*)

2. The diversification function d1 generates diversified keys (IRK, CSRK, LTK, DHK) from base keys such as IR and ER and makes use of the encryption function e:

   *d1 (k, d, r) =* ***e*** *(k, d')*  with d'= padding||r||d,  r and d 16 bits long.

3. Random addresses are generated by combining a random number and a hash of that random number. The random address hash function *ah* is used to generate the hash.

   *ah (k, r) =* ***e****(k, r') mod* $2^{24}$, r'= padding || r with r 24 bits long

4. DIV is masked during encryption session setup using the output of the DIV mask generation function *dm* to generate the distributed EDIV value.

   *dm (k, r) =* ***e****(k, r') mod* $2^{16}$,  r'= padding || r,  with r 64 bits long

5. During the pairing process confirm values are exchange. This confirm value generation function *c1* is used to generate the confirm values.

   *c1 (k, r, preq, pres, iat, rat, ia, ra) =* ***e****(k,* ***e****(k, r XOR pres||preq||rat'||iat') XOR pad||ia||ra)*

   with :

   - *pres* = pairing result PDU
   - *preq* = pairing request pdu
   - *rat'* = responder address type padded to 8 bits
   - *iat'* = initiator address type padded to 8 bits
   - *ra* = responder address
   - *ia* = initiator address

6. The key generation function *s1* is used to generate the STK during the pairing process.

   *s1 (k, r1, r2) =* ***e****(k, r')* , r1 = Srand, r2 = Mrand and r'= r1[127:64] || r2[127:64]

#### 5.4.2.3 Resolvable Private Address Hash Part Generation

A resolvable private address is the concatenation of two 24-bit parts: the hash part and the random part. The hash allows an address to be resolved using an Identity Resolution Key (IRK) and the random part.

*resolv_priv_addr = hash || prand*

The function allowing to generate the hash part is called ah.

*hash = ah(IRK, prand) =* $LSB_{24}$*(e(IRK, prand'))*

*with prand' =* $0^{104}$ *|| prand*

#### 5.4.2.4 Confirm Value Generation

The confirm value is exchanged during a pairing procedure.

The function used to generate a confirm value is called c1. This function takes values as parameters:

- Temporary Key (TK), 128 bits
- A random number (R), 128 bits
- The pairing request command (PREQ), 56 bits
- The pairing response command (PRES), 56 bits
- The initiating device address type (IAT), 1 bit
- The initiating device address (IA), 48 bits
- The responding device address type (RAT), 1 bit
- The responding device address (RA), 48 bits

The following steps have to be followed within the c1 function:

1. *Let* $p_1$ *= PRES || PREQ ||* $0^7$ *|| RAT ||* $0^7$ *|| IAT*
2. *Let* $p_2 = 0^{32}$ *|| IA || RA*
3. *Let* $e_1 = e(TK, R \text{ XOR } p_1)$

4. *confirm = e(TK, $e_1$ XOR $p_2$)*

#### 5.4.2.5 Short Term Key (STK) Generation

The STK is used during the pairing procedure to encrypt the link before the device keys are exchanged. It is generated through the s1 function.

The following steps have to be followed within the s1 function:

1. *Let Srand' = $LSB_{64}$(Srand)*
2. *Let Mrand' = $LSB_{64}$(Mrand)*
3. *Let r = Srand' || Mrand'*
4. *STK = e(TK, r)*

#### 5.4.2.6 AES-CMAC Algorithm

RFC-4493 1 defines the Cipher-based Message Authentication Code (CMAC) that uses AES-128 as the block cipher function, also known as AES-CMAC.

The inputs to AES-CMAC are:

- M is the variable length data to be authenticated
- K is the 128-bit key

The AES-CMAC requires generation of two subkeys which are deduced from input key K.

Generation of two subkeys is made using the following algorithm:

1. *Let L = e(CSRK, $0^{128}$)*
2. *If MSB(L) == 0, $K_1$ = (L << 1) else $K_1$ = (L << 1) XOR $R_{128}$, with $R_{128}$ = $0^{120}$10000111*
3. *If MSB($K_1$) == 0, $K_2$ = $K_1$ << 1 else $K_2$ = ($K_1$ << 1) XOR $R_{128}$*

AES-CMAC is performed using the algorithm describes below:

1. *Produce $K_1$, $K_2$ = subkey_gen(K) (see 0)*
2. *Let n = ceil($M_{len}$/128), $M_{len}$ = len(M), M = Data PDU*
3. *Let M = $M_1$ || $M_2$ || … || $M_n$* where $M_1$, …, $M_{n-1}$ are blocks of size 128*
4. *If len($M_n$*) == 128, $M_n$ = $K_1$ XOR $M_n$* else $M_n$ = $K_2$ XOR ($M_n$* || **1**$0^j$), with j = 128 – len($M_n$*) - 1*
5. *Let $C_0$ = $0^{128}$*
6. *For i = 1 to n, let $C_i$ = AES_128(K, $C_{i-1}$ XOR $M_i$)*

#### 5.4.2.7 Data Signing (MAC Generation)

The Data Signing procedure is used to authenticate a data PDU sent over a non-encrypted link.

The algorithm used for data signing in presented in [2].

The inputs to MAC are:

- *M is the variable length data to be authenticated*

- *CSRK is the 128-bit key*

The MAC generation uses AES-CMAC algorithm like described below:

- $MAC = MSB_{64}(AES\text{-}CMAC_{CSRK}(M))$

### 5.4.3 LE Secure Connections Confirm Value Generation Function f4

During the LE Secure Connections pairing process, confirm values are exchanged. These confirm values are computed using the confirm value generation function f4 (see [1] Vol 3 Part H Chapter 2.2.6).

This confirm value generation function makes use of the MAC function AES-CMAC X , with a 128-bit key X.

The inputs to function f4 are:

- *U is 256 bits*
- *V is 256 bits*
- *X is 128 bits*
- *Z is 8 bits*

Algorithm:

- $f4(U, V, X, Z) = AES\text{-}CMAC_X(U \,||\, V \,||\, Z)$

| Numeric Comparison / Just Works | Out-Of-Band | Passkey Entry |
|---|---|---|
| **Ca = f4 (PKax, PKbx, Na, 0)** | Ca = f4 (PKax, PKax, Rara, 0) | Cai = f4 (PKax, PKbx, Nai, rai) |
| **Cb = f4 (PKbx, PKax, Nb, 0)** | Cb = f4 (PKbx, PKbx, Rbrb, 0) | Cbi = f4 (PKbx, PKax, Nbi, rbi) |

**Table 5-4 Inputs to f4 for the different protocols**

### 5.4.4 LE Secure Connections Key Generation Function f5

The LE Secure Connections key generation function f5 is used to generate derived keying material in order to create the LTK and keys for the commitment function f6 during the LE Secure Connections pairing process (see [1] Vol 3 Part H Chapter 2.2.7).

The inputs to function f5 are:

- *W is 256 bits*
- *N1 is 128 bits*
- *N2 is 128 bits*
- *A1 is 56 bits*
- *A2 is 56 bits*

The key (T) is computed as follows:

- *SALT = 0x6C88 8391 AAF5 A538 6037 0BDB 5A60 83BE*
- $T = AES\text{-}CMAC_{SALT}(W)$

The string “btle” is mapped into keyID using extended ASCII as follows:

- *keyID = 0x62746c65*

The output of the key generation function f5 is as follows:

- *f5(W, N1, N2, A1, A2) = AES-CMAC$_T$(0x00||keyID||N1||N2||A1||A2||0x0100)*
  *|| AES-CMAC$_T$(0x01||keyID||N1||N2||A1||A2||0x0100)*

The LTK and MacKey are calculated as:

- *MacKey || LTK = f5(DHKey, N_master, N_slave, BD_ADDR_master, BD_ADDR_slave)*

### 5.4.5 LE Secure Connections Check Value Generation Function f6

The LE Secure Connections check value generation function f6 is used to generate check values during authentication stage 2 of the LE Secure Connections pairing process (see [1] Vol 3 Part H Chapter 2.2.8).

The inputs to function f6 are:

- *W is 128 bits*
- *N1 is 128 bits*
- *N2 is 128 bits*
- *R is 128 bits*
- *IOcap is 24 bits*
- *A1 is 56 bits*
- *A2 is 56 bits*

The output of the check value generation function f6 is as follows:

- *f6(W, N1, N2, R, IOcap, A1, A2) = AES-CMAC$_W$(N1||N2||R||IOcap||A1||A2)*

The inputs to f6 are different depending on different association models:

| Numeric Comparison / Just Works | Out-Of-Band | Passkey Entry |
|---|---|---|
| **Ea = f6 (MacKey, Na, Nb, 0,IOcapA, A, B)** | Ea = f6 (MacKey, Na, Nb, rb, IOcapA, A, B) | Ea = f6 (MacKey, Na20, Nb20, rb, IOcapA, A, B) |
| **Eb = f6 (MacKey, Nb, Na, 0, IOcapB, B, A)** | Eb = f6 (MacKey, Nb, Na, ra, IOcapB, B, A) | Eb = f6 (MacKey, Nb20, Na20, ra, IOcapB, B, A) |

**Table 5-5 Inputs to f6 for the different protocols**

In Passkey Entry, ra and rb are 6-digit passkey values expressed as a 128-bit integer.

### 5.4.6 LE Secure Connections Numeric Comparison Value Generation Function g2

The LE Secure Connections numeric comparison value generation function g2 is used to generate the numeric comparison values during authentication stage 1 of the LE Secure Connections pairing process (see [1] Vol 3 Part H Chapter 2.2.9).

The inputs to function g2 are:

- *U is 256 bits*
- *V is 256 bits*
- *X is 128 bits*
- *Y is 128 bits*

The output of the numeric comparison value generation function g2 is as follows:

- *g2 (U, V, X, Y) = AES-CMAC$_X$(U||V||Y) mod $2^{32}$*

### 5.4.7 Link Key Conversion Function h6

The function h6 is used to convert keys of a given size from one key type to another key type with equivalent strength (see [1] Vol 3 Part H Chapter 2.2.10).

The inputs to function h6 are:

- *W is 128 bits*
- *keyID is 32 bits*

The output of h6 is as follows:

- *h6 (W, keyID) = AES-CMAC$_W$(keyID)*

### 5.4.8 Identity Root Generation

The Identity Root (IR) can be created in two ways. It can be assigned by a value or generated in random. If by arbitrary creation, it will follow the requirements of random generation defined in Volume 2, Part H Section 2 of Bluetooth Core Specification.

#### 5.4.8.1 Identity Resolving Key Generation

The Identity Resolving Key (IRK) is used for random address construction and resolution. It is created through the diversification function d1, using the IR as parameter k and 0x0001 as parameter d. In case a hierarchy method is not used, IRK may be directly assigned as a random 16 octet value to the device (per connection).

#### 5.4.8.2 Diversifier Hiding Key Generation

The Diversifier Hiding Key (DHK) is used to mask DIV during the encrypted session setup. It is created through the diversification function d1, using the IR as parameter k and 0x0002 as parameter d. If the hierarchy method is not used, it can also be randomly generated.

#### 5.4.8.3 Connection Signature Resolving Key Generation

The Connection Signature Resolving Key (CSRK) is used to sign data and resolve signature of received messages. It can be assigned or randomly generated. If by arbitrary creation, it will follow the requirements of random generation defined in Volume 2, Part H Section 2 of Bluetooth Core Specification.

#### 5.4.8.4 Long Term Key and Diversifier Generation

Devices supporting encrypted links in the slave role are capable of generating unique LTK and DIV values.

The DIV is used by the slave device to regenerate a previously shared LTK in order to start an encrypted connection with a previously paired master device.

Any method of generation of LTK can be used as it is not visible outside the slave device. New values of LTK and DIV are generated each time they are distributed.

#### 5.4.8.5 Encrypted Session Setup

Establishing an encrypted link requires that both devices use the same Key, which has either been generated on both devices using the same base parameters (reference to STK) or previously distributed. Both devices always use the Slave distributed LTK if the link is to be encrypted using LTK.

The host of the master provides the Link Layer with the Long Term Key to use when setting up the encrypted session, together with the EDIV and RAND number that correspond to it. The EDIV and RAND are two 'identifiers' for the LTK and they allow retrieval of the same key on both devices without actually exchanging it. During the encryption session setup the master device sends the *EDIV* and the random number to the slave device. The host of the slave receives the *EDIV* and *Rand* values and provides the corresponding Long Term Key to the slave's Link Layer to use when setting up the encrypted link.

The encrypted session can be setup either by using STK or LTK. The procedure is the same, the only difference being that when using STK, EDIV=RAND=0.

#### 5.4.8.6 Signing Algorithm

An LE device can send signed data without having to establish an encrypted session with a peer device. Data is signed using *CSRK*.

The signing algorithm is used in two situations:

- ✓ Signing own data with own CSRK in view of transmission to peer which is supposed to have received the CSRK during phase 3, and would thus interpret the received message.
- ✓ Verification of received signed messages, using CSRK received from peer during previous Phase 3. The same algorithm is used to generate the signature of the received message and check it against the received signature.

#### 5.4.8.7 Slave Initiated Security

There are three manners in which the master handles the security request from slave.

- ✓ No LTK is available for this connection, or the existing security information does not have the requested security properties => pairing must be initiated.
- ✓ An LTK is available for this connection, with security properties matching the request => start encrypting link directly without pairing.
- ✓ Send slave Pairing Failed PDU that Master does not support pairing at that moment.

### 5.4.9 Procedure Details

This part presents the messages that are exchanged between the layers of the RW-BLE stack during the different procedure that are supported by the SMP. The SMP API messages are described in the next part.

#### 5.4.9.1 Random Address Generation

A device may use a random address. This random address may be of either of the following types:

- Static address
- Private address

A private address may be either of the following types:

- Non-resolvable private address

- Resolvable private address

The three figures below give the structure of each kind of private address.

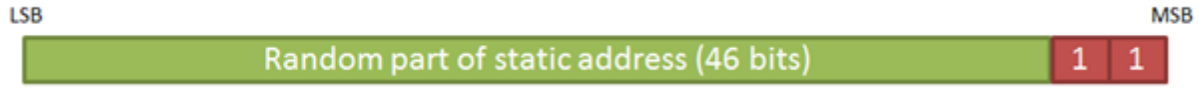



**Figure 5-27: Static random address structure**


**Figure 5-28: Private non-resolvable random address structure**

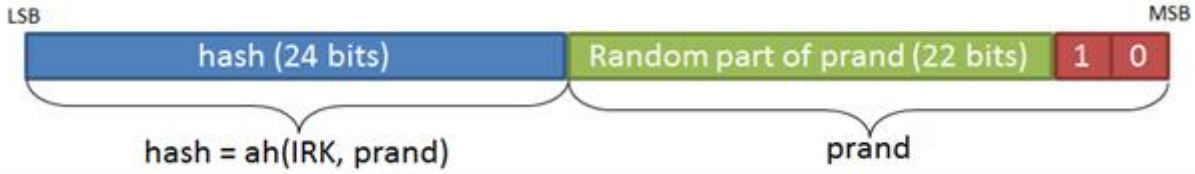



**Figure 5-29: Private resolvable random address structure**

The random address generation procedure will be the same whatever the kind of random address which is requested. However in the case of a resolvable private address, the IRK used to generate the address shall be kept by the higher layers so that it can be distributed to a peer device.

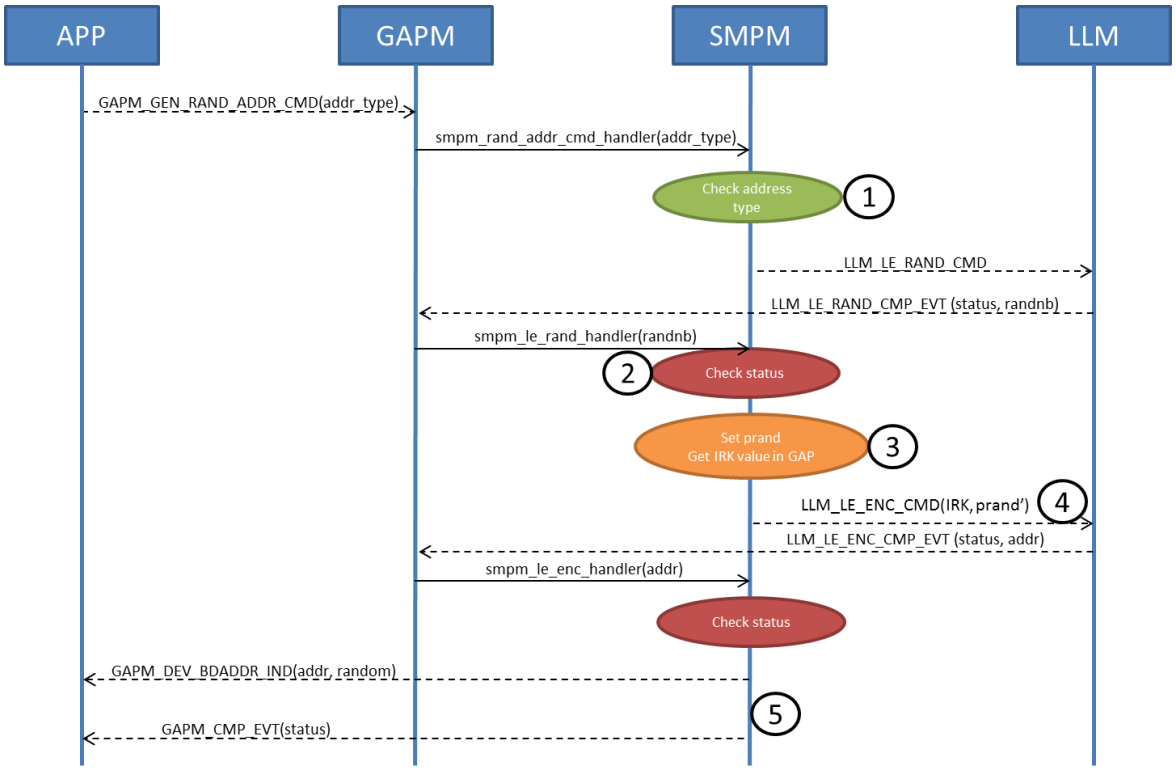



**Figure 5-30: Random Address Generation Procedure**

1. If the address type is not valid, a GAPM_CMP_EVT message with a GAP_ERR_INVALID_PARAM status error is sent.
2. If an error status is returned by the controller, a GAPM_CMP_EVT message with a GAP_ERR_LL_ERROR status error is sent.
3. prand = $LSB_{22}$(randnb) || $LSB_{2}$(addr_type)
4. prand' = $0^{104}$ || prand
5. If an error status is returned by the controller, a GAPM_CMP_EVT message with a GAP_ERR_LL_ERROR status error is sent else the status will be GAP_ERR_NO_ERROR.

#### 5.4.9.2 Address Resolution

The address resolution procedure is used to identify a device which would use a resolvable private random address. The structure of this kind of address is defined in 5.4.9.1.

The GAP provides several IRK for a same address. The hash part of this address is regenerated using the IRK and the prand part of the address. If the generated hash part is the same as the hash part of the provided address, the address is considered as resolved, else another IRK shall be sent.

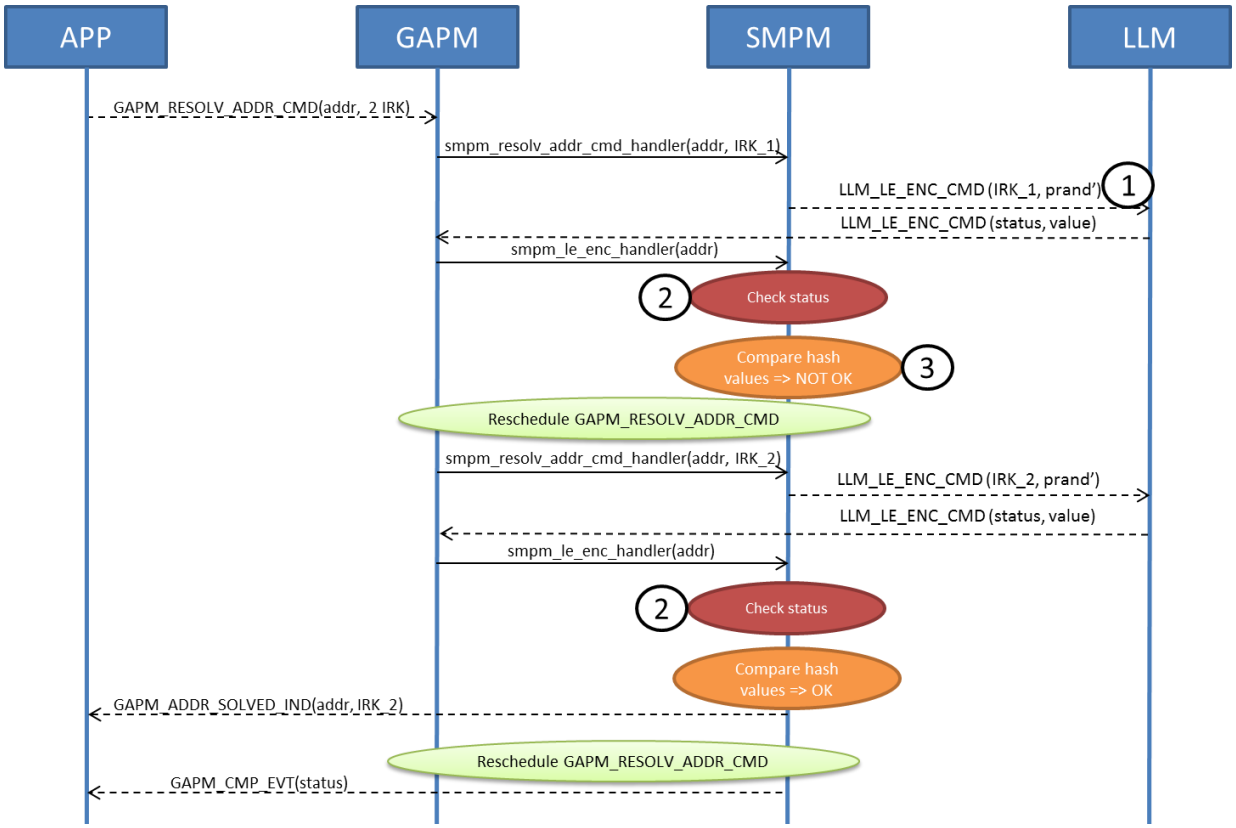



**Figure 5-31: Address Resolution Procedure**

1. prand' = $0^{104}$ || prand = $0^{104}$ || addr[0:23]
2. If an error status is returned by the controller, a GAPM_CMP_EVT message with a GAP_ERR_LL_ERROR status error is sent.
3. hash = value[0:23]

### 5.4.9.3 Encryption Toolbox Access

The encryption toolbox access provides a way a host layer to use the hardware encryption block. This block can be access using the LLM API.

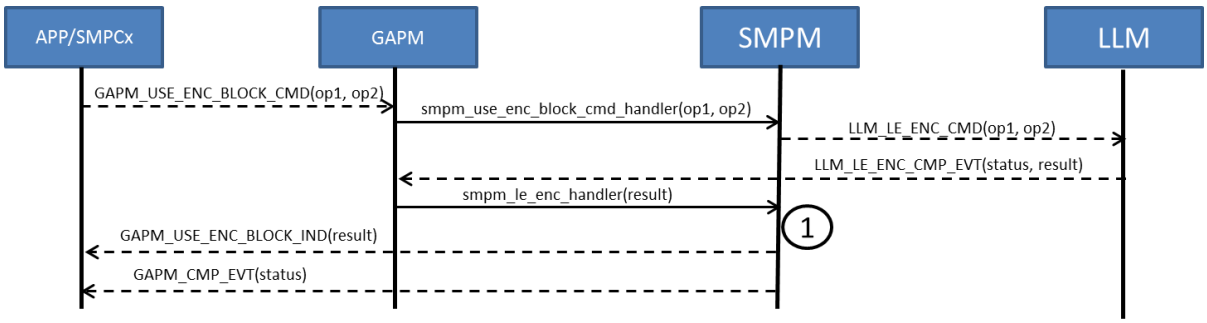



**Figure 5-32: Encryption Toolbox Access**

1. If an error status is received from the controller, the GAPM_CMP_EVT message with a GAP_ERR_LL_ERROR status is directly sent to the requested layer.

#### 5.4.9.4 Pairing

The pairing procedure between two devices can be divided into several phases:

- Phase 1 – Pairing Feature Exchange: It is used to exchange IO capabilities, OOB authentication data, authentication requirements and which keys to distribute.
- Legacy Phase 2 – Authentication and Encryption: Information exchanged during the phase 1 are used to determine which method will be used to encrypt the link (Just Works, Passkey Entry, Out Of Band).
- LE Secure Connections Phase 2: – Authentication and Encryption: Information exchanged during the phase 1 are used to determine which method will be used to encrypt the link (Just Works, Numeric Comparison, Passkey Entry, Out Of Band). The outcome of this pairing is Long Term Key (LTK) Generation
- Phase 3 – Transport Keys Distribution: This phase is optional and depends on the key distribution features shared during phase 1.

##### 5.4.9.4.1 Phase 1: Pairing Feature Exchange (Initiated by Master)

The pairing is always iniated by the master device by sending a pairing request message.

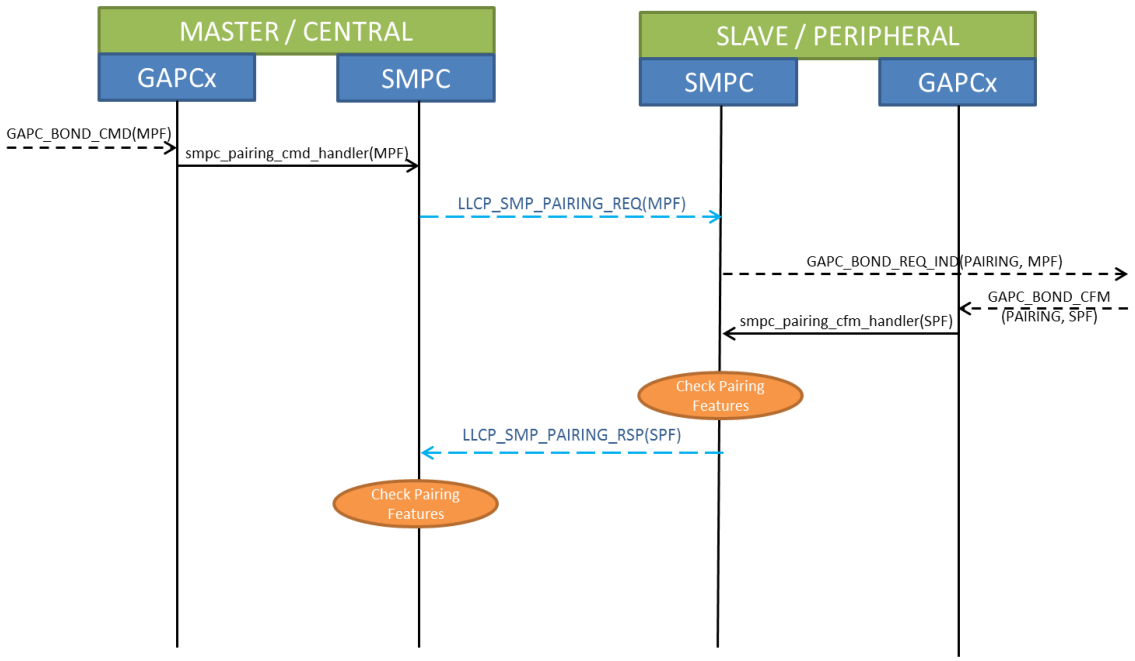



**Figure 5-33: Pairing Phase 1: Pairing Features Exchange (Initiated by Master)**

If the slave device doesn't support pairing, it responds using the Pairing Failed message with the error code "Pairing Not Supported" upon reception of a Pairing Request message.

If a device receives a command with invalid parameters, it responds with a Pairing Failed command with the error code "Invalid Parameters".

The minimum and maximum encryption key length parameters shall be between 7 bytes and 16 bytes in 1 byte step. The smaller value of the initiating and the responding devices maximum encryption key length will be used as the encryption key size. If the resultant encryption key size is smaller than the minimum key size parameter, the device responds with Pairing Failed command with the error code "Encryption Key Size".

#### 5.4.9.4.2 Phase 1: Pairing Feature Exchange (Initiated by Slave)

A slave device may require that the master initiates a pairing produre by sending a security request.

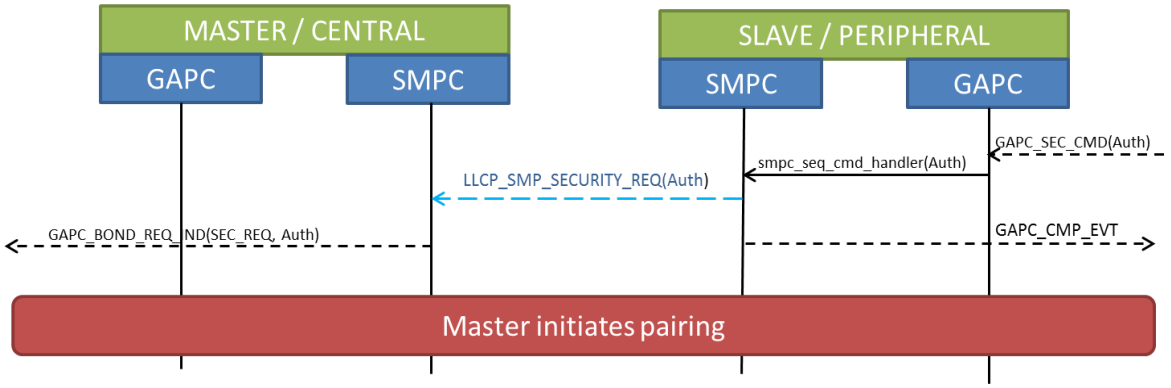



**Figure 5-34: Pairing Phase 1: Pairing Features Exchange (Initiated by Slave)**

#### 5.4.9.4.3 Legacy Phase 2: Authentication and Encryption

The information exchanged in Phase 1 is used to select which STK generation method is used in Phase 2.

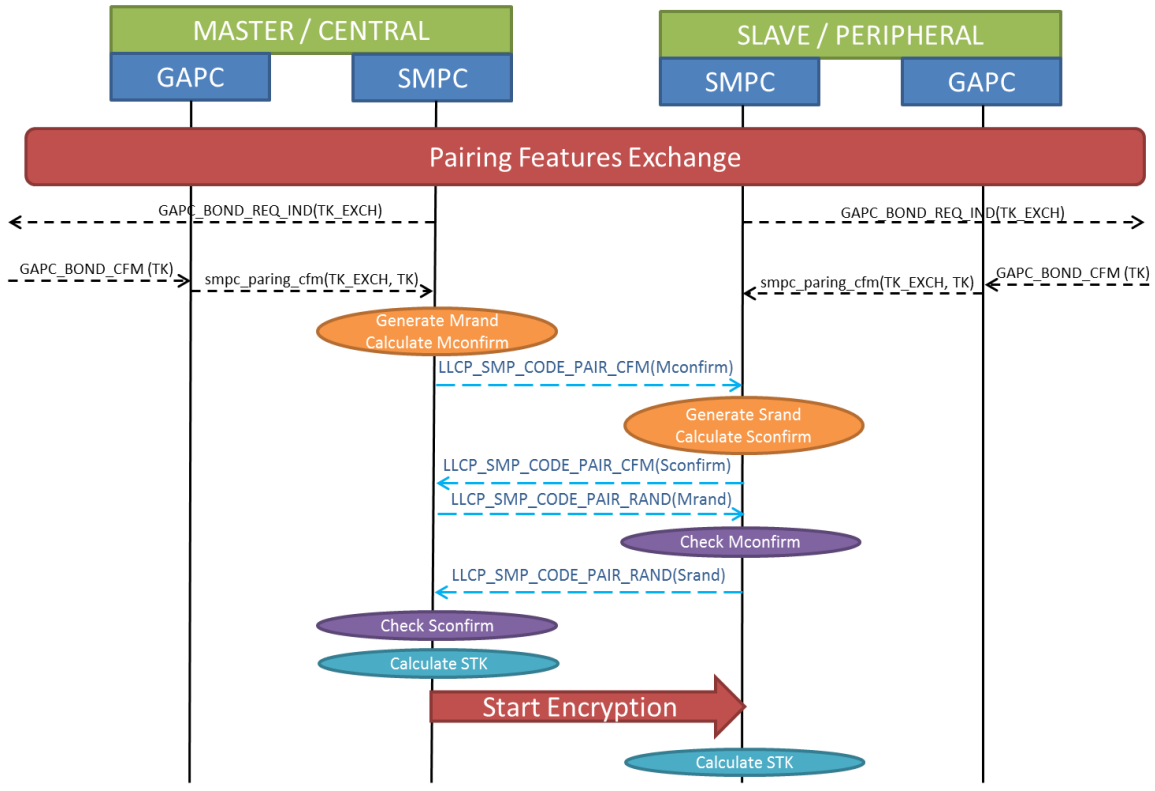



**Figure 5-35: Phase 2: Authentication and Encryption**

If the just works method is used, no TK will be required (0 is used) from the application.

If a generated Sconfirm or Mconfirm value doesn't match with the received confirm value from the peer device, the device aborts the the pairing procedure by sending a Pairing Failed message with a "Confirm Value Failed" error code.

#### 5.4.9.4.4 LE Secure Connection Phase 2: Authentication and Encryption

##### 5.4.9.4.4.1 Authentication Stage 1: Just Work Method

If not possible to enter passkey or do a numeric comparison, this method applies:

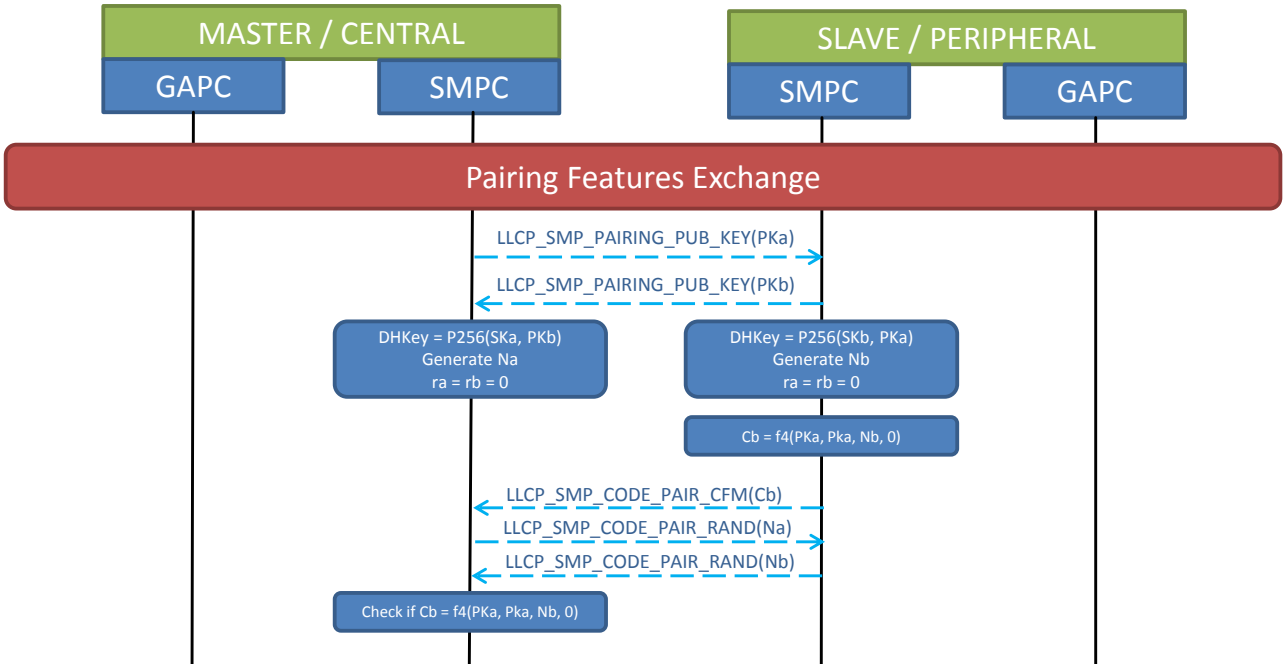



**Figure 5-36: Phase 2: LE Secure Connection Just Work Pairing**

At the end of the pairing, link is considered Unauthenticated.

#### 5.4.9.4.4.2 Authentication Stage 1: Numeric Comparison Method

If both devices have display capability, Numeric comparison should be chosen

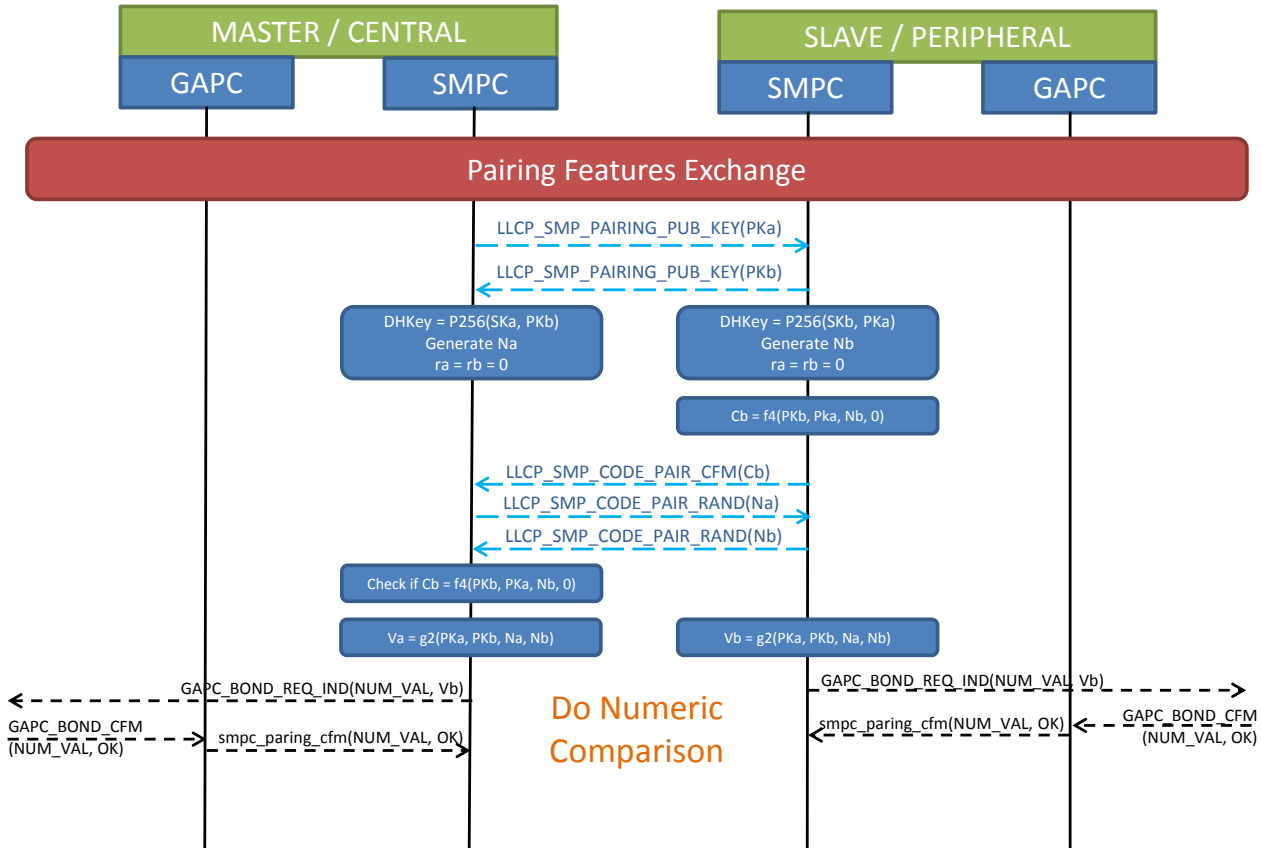



**Figure 5-37: Phase 2: LE Secure Connection Numeric Comparison Pairing**

At the end of the pairing, link is considered Secure Connection Authenticated.

#### 5.4.9.4.4.3 Authentication Stage 1: Passkey Entry Method

If both devices pin code entry is possible, passkey entry is chosen:

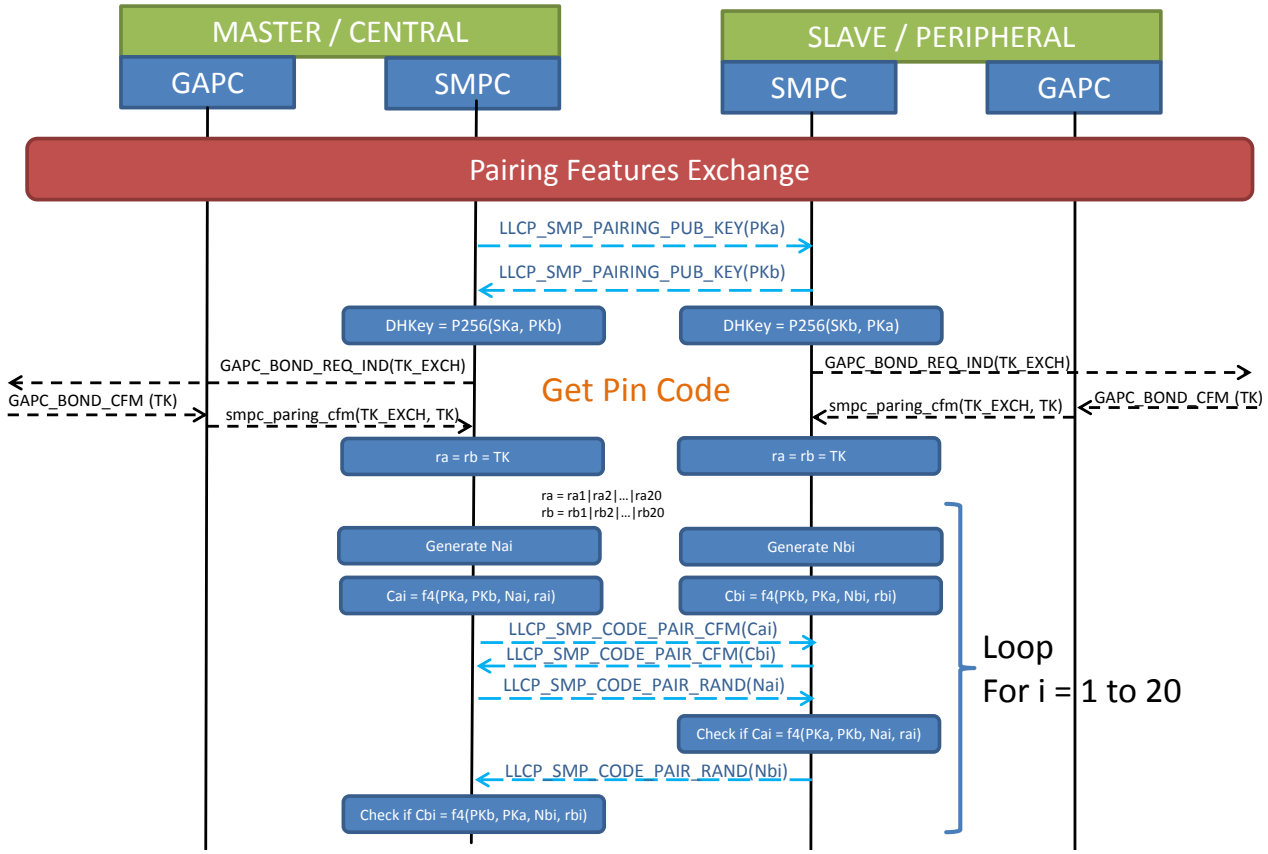



**Figure 5-38: Phase 2: LE Secure Connection Passkey Entry pairing**

During Passkey entry, *LLCP_SMP_PASS_KEY_ENTRY* message can be sent to peer device using *GAPC_BOND_CFM(PASSKEY_ENTRY)* message in order to inform peer device that user is entering password.

At the end of the pairing, link is considered Secure Connection Authenticated.

#### 5.4.9.4.4.4 Authentication Stage 1: Passkey Entry Method

If OOB Data can be send by one or both devices, Out Of Band pairing method is chosen:

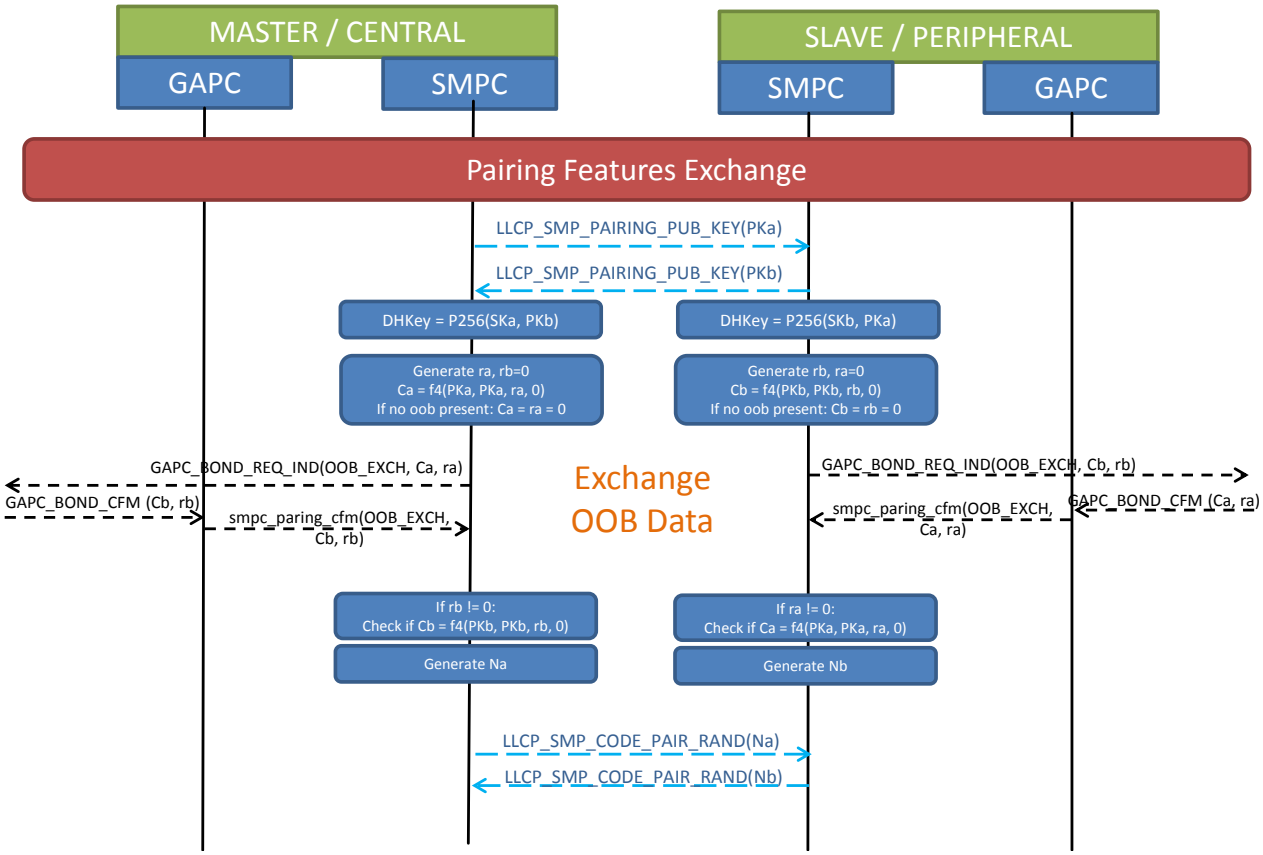



**Figure 5-39: Phase 2: LE Secure Connection Out Of Band pairing**

At the end of the pairing, link is considered Secure Connection Authenticated.

#### 5.4.9.4.4.5 Authentication Stage 2: Generation of LTK

After the LE Secure Connection Authentication Stage2, LTK is generated according to pairing information. Then link is encrypted.

If encryption succeeds and BOND bit present in pairing feature exchange, the generated LTK is provided to upper application.

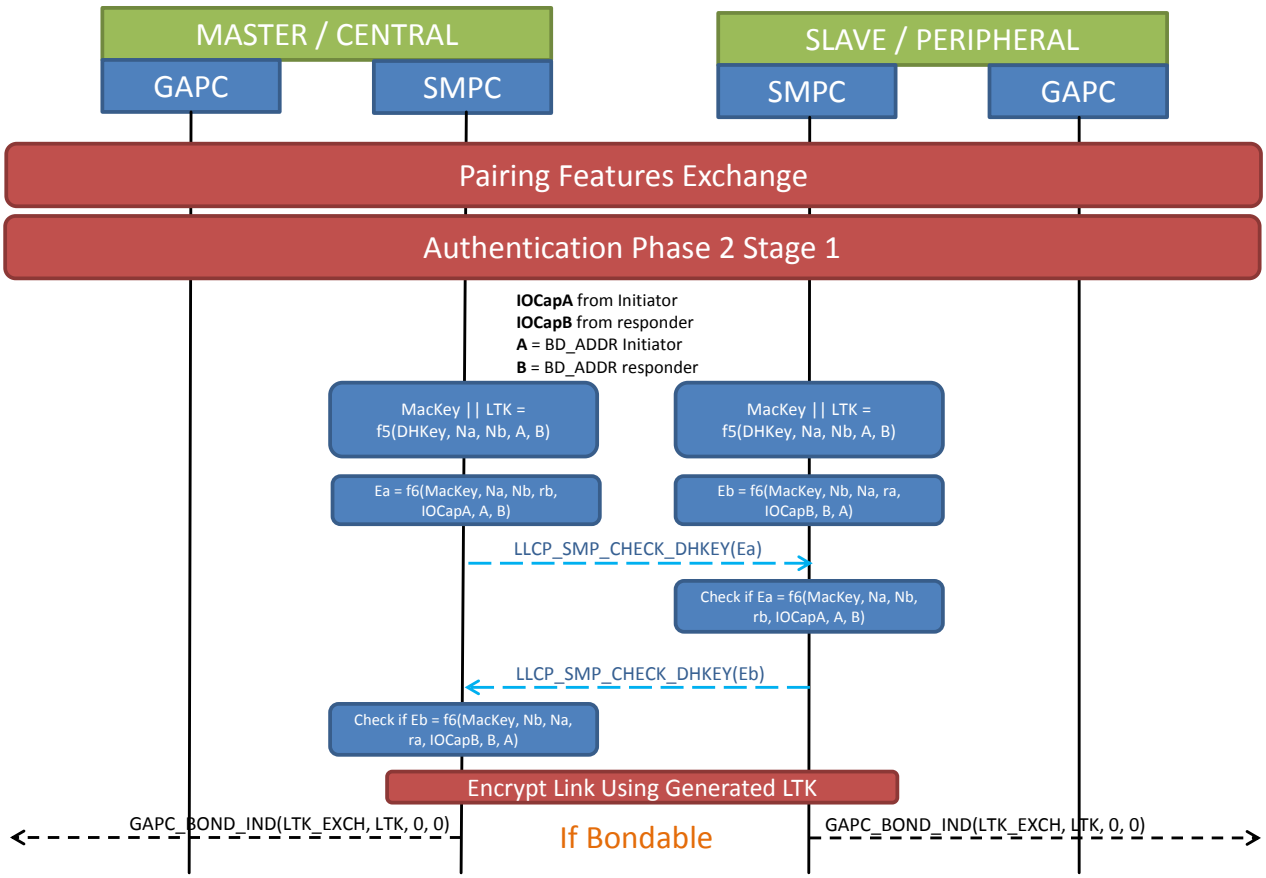



**Figure 5-40: Phase 2: LE Secure Connection LTK Generation**

#### 5.4.9.4.5 Phase 3: Transport Keys Distribution

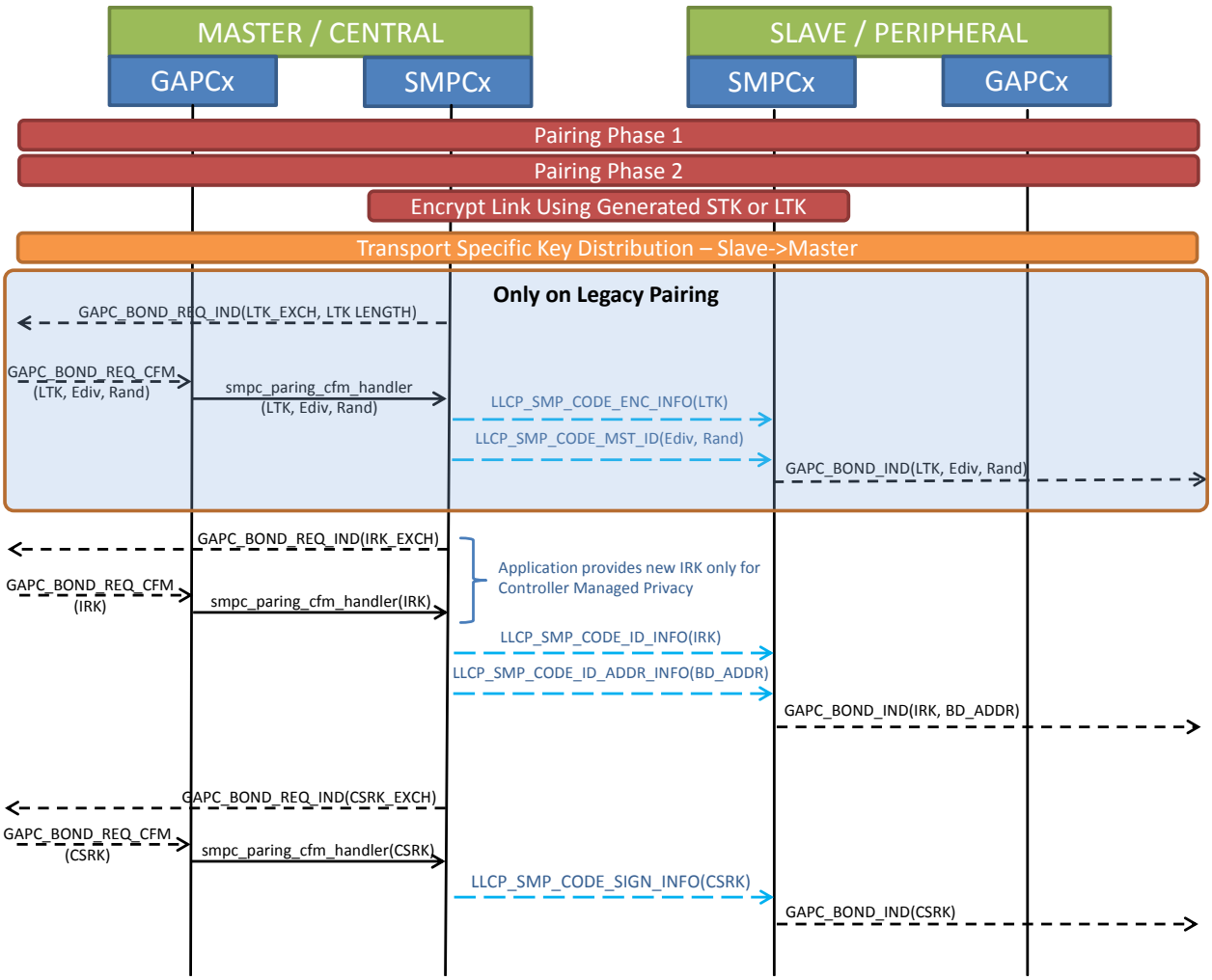



**Figure 5-41: Phase 3: Transport Keys Distribution**

When Privacy is managed by host (privacy 1.1), IRK value is already set in the GAP environment. But for a controller managed privacy (privacy 1.2), IRK should be unique for each bonded device, and so the new IRK should be generated and retrieved from the application.

The LTK and the CSRK needs to be retrieved from the application.

On legacy paring, application is responsible for generating the transport keys (CSRK, IRK, LTK, Ediv, Rand) by any means. The GAPM_USE_ENC_BLOCK_CMD message can be used through the GAP API.

On secure Connection pairing, application is responsible for generating only CSRK, IRK, the LTK is generated by the pairing algorithm

#### 5.4.9.4.6 End of Pairing Procedure

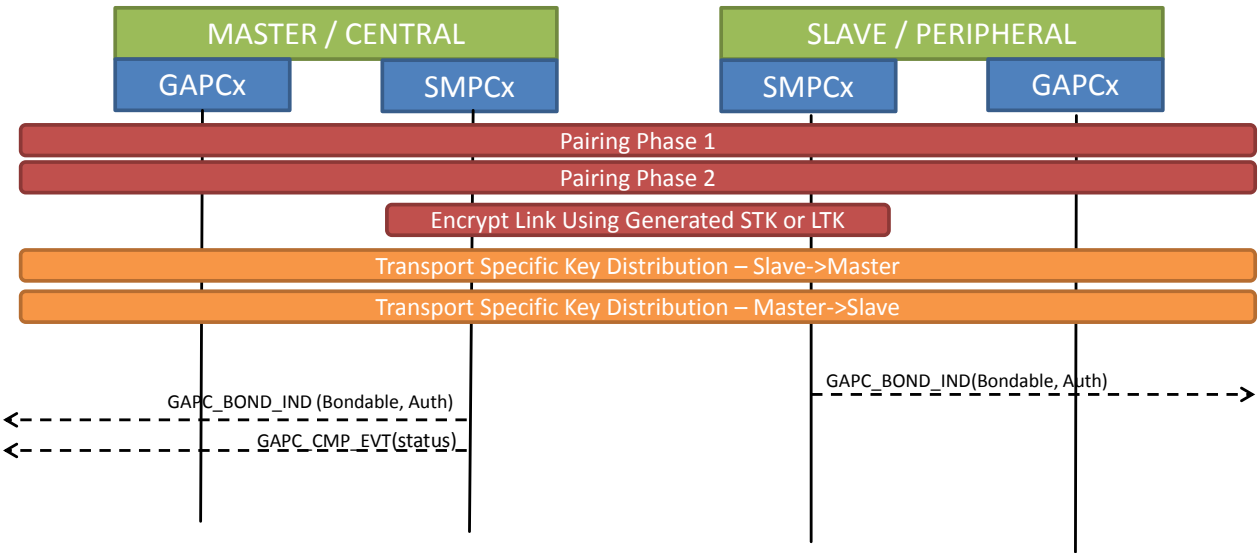



**Figure 5-42: End of Pairing Procedure**

The pairing procedure is considered as over in the following case:

- A Pairing Failed message has been received or generated.
- Phase 2 is over and no keys need to be distributed.
- All required keys have been distributed during Phase 3.

### 5.4.9.5 Encryption

The master device must have the security information (LTK, EDIV, and Rand) distributed by the slave device to setup an encrypted session. An encrypted session is always initiated by the master.

#### 5.4.9.5.1 Case 1: Both devices have LTK

If a master already knows the encryption keys of the slave device it is connected with, it can initiate the creation of an encrypted link.

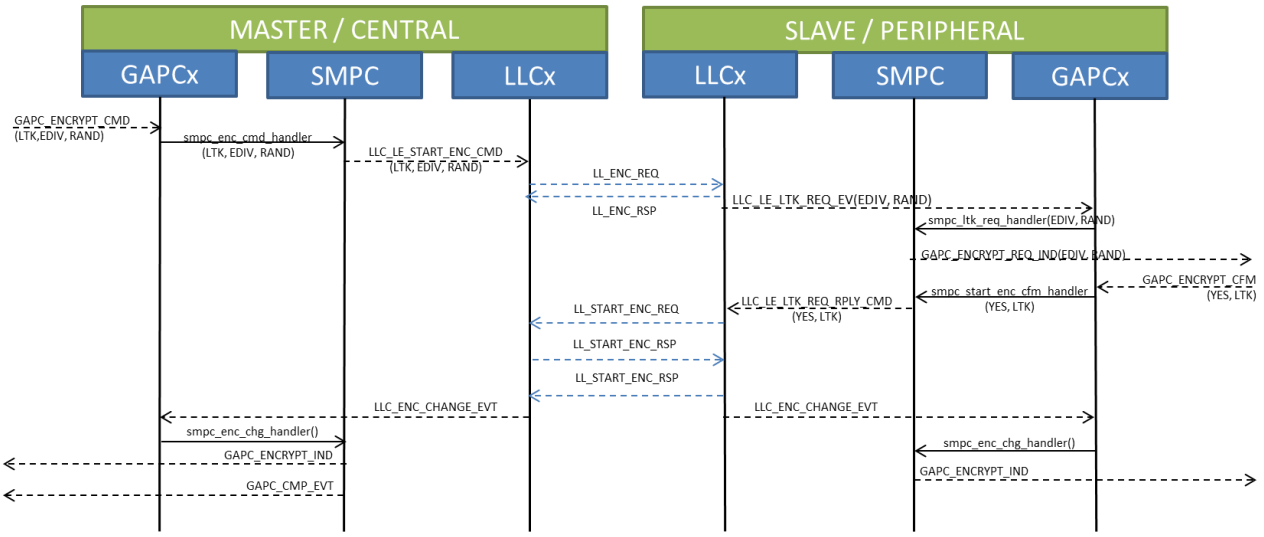


**Figure 5-43: Start Encryption Procedure (Both devices have keys)**

The slave may require establishment of an encrypted session by sending a security request. Upon reception of this request, the master device will check if it can retrieve the LTK distributed by the device. If a key is found, the master will start the encryption procedure else it will start a pairing procedure.

#### 5.4.9.5.2 Case 2: Slave forgot the LTK

If the slave forgot the LTK distributed by the master device during a previous bonding procedure, it will reject the encryption request with a "PIN KEY missing" error.

Upon reception of this error, the master can initiate a new pairing procedure with the slave device.

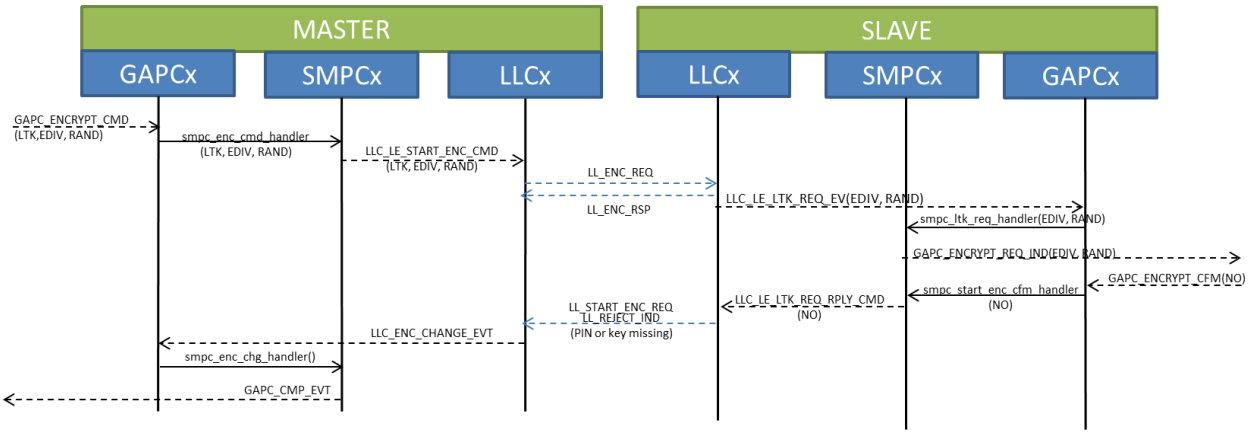



**Figure 5-44: Start Encryption Procedure (Slave forgot keys)**

#### 5.4.9.5.3 Case 3: Slave doesn't support encryption

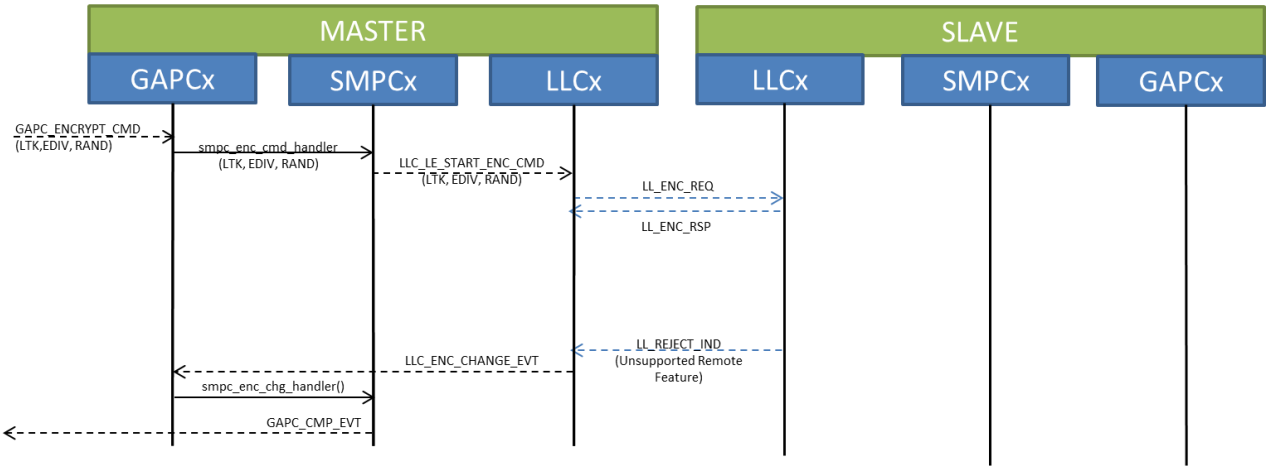



**Figure 5-45: Start Encryption Procedure (Slave doesn't support encryption)**

### 5.4.9.6 Data Signing

The Data Signing procedure is used to authenticate a data PDU sent over a non-encrypted link.

More details about the generation of the signature can be found in 5.4.2.6.

#### 5.4.9.6.1 Subkeys Generation

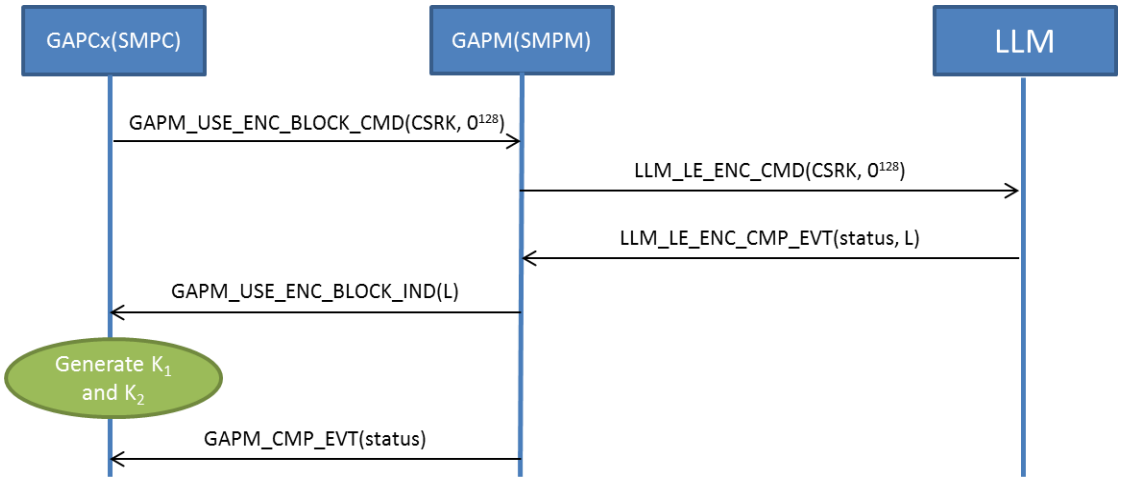



**Figure 5-46: Data Signing: Subkeys Generation**

#### 5.4.9.6.2 MAC Generation

The data to be signed in the concatenation of the data PDU and the SignCounter value.

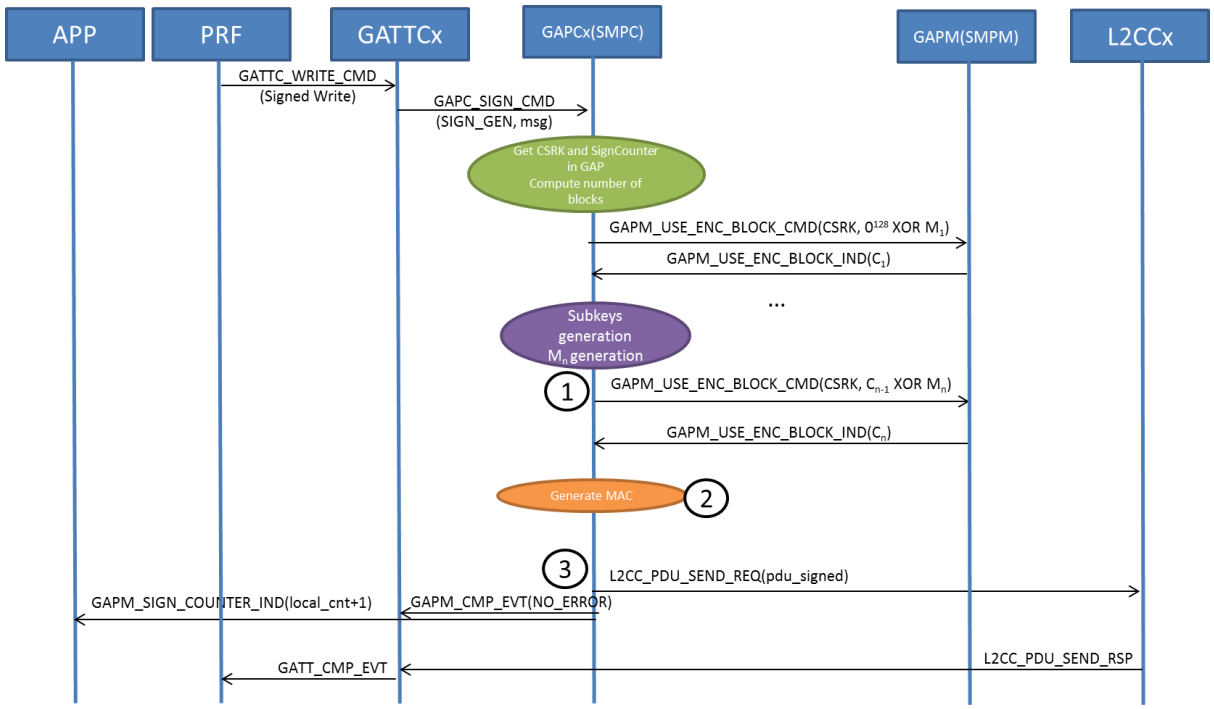



**Figure 5-47: Data Signing: MAC Generation**

1. SMPC module receive a PDU message to sign from GATTC, it uses SMPM Encryption block through GAPM API.
2. After using several times encryption block, it generate MAC signature and append it to the PDU.
3. The signed PDU message is conveyed to L2CC with GATTC task as source ID in order to prevent a kernel reschedule. Application is also informed that sign counter has been increased.

#### 5.4.9.6.3 MAC Verification

The verification of the received MAC is done by generating a MAC value based on the received data PDU and SignCounter values. If the generated MAC value matches with the received one, the signature is accepted.

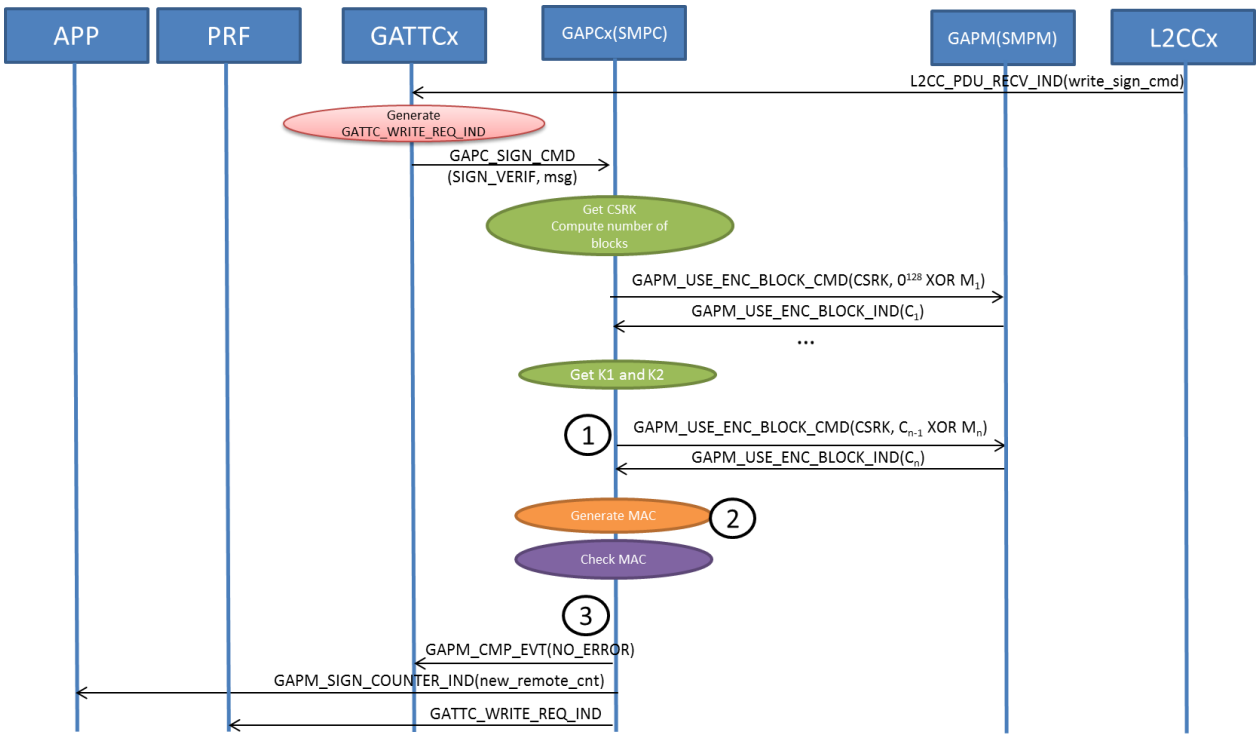



**Figure 5-48: Data Signing: MAC Verification**

1. SMPC module receives a GATTC_WRITE_REQ_IND message with as signature to verify from GATTC, it uses SMPM Encryption block through GAPM API.
2. After using several times encryption block, it generate MAC signature and compare to the provided signature.
3. The GATTC_WRITE_REQ_IND message is sent to the targeted profile GATTC task as source ID in order to prevent a kernel reschedule. Application is also informed that remote sign counter has been increased. If an error occurs during signature, GATTC_WRITE_REQ_IND message is dropped and GATTC is inform that signature verification has failed.

#### 5.4.9.7 Pairing Repeated Attempts

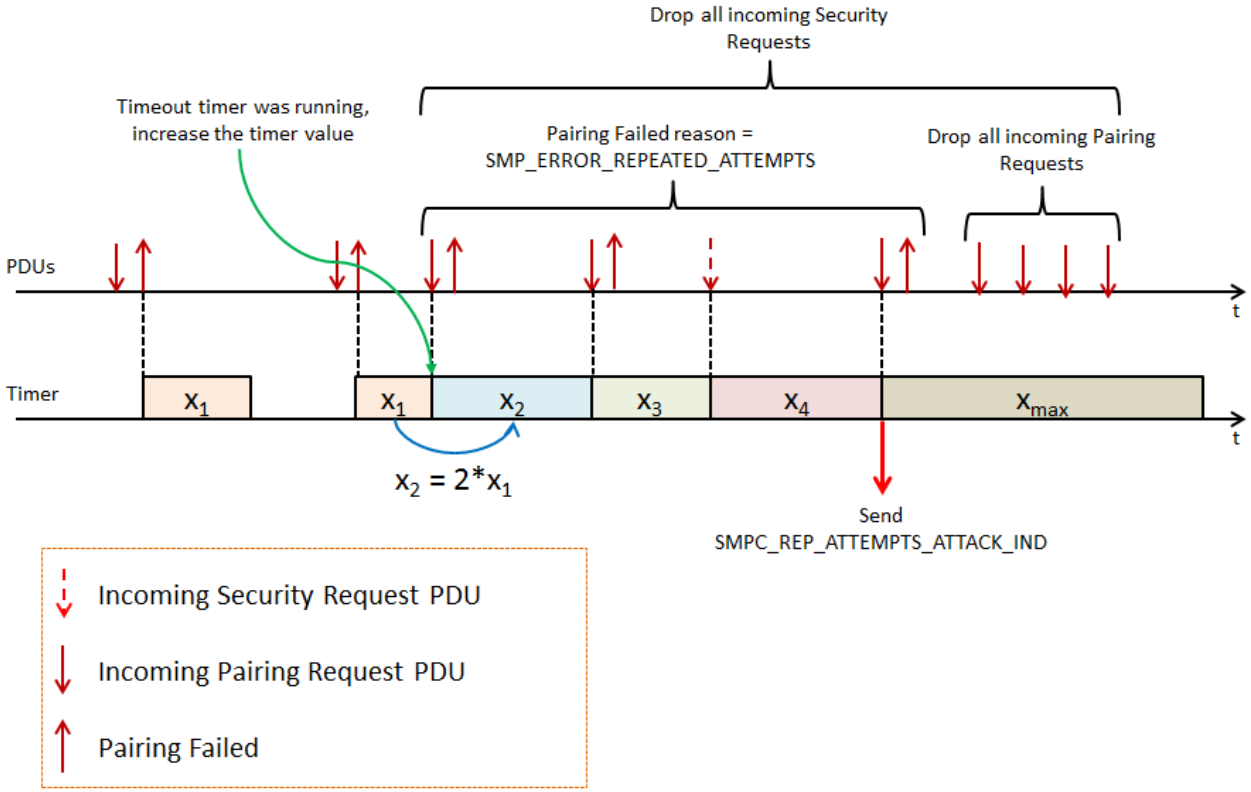



**Figure 5-49: Repeated Attempts Protection**

The Bluetooth specification [1] required the implementation of a mechanism

*"When a pairing procedure fails a waiting interval shall pass before the verifier will initiate a new Pairing Request command or Security Request command to the same claimant, or before it will respond to a Pairing Request command or Security Request command initiated by a device claiming the same identity as the failed device. For each subsequent failure, the waiting interval shall be increased exponentially."*

Figure 5-49 presents the mechanism implemented to rapidly detect an attack from a malicious device.

The minimal interval value is set to 2s and the maximal interval value is set to 30s. Thus, according to the procedure described in the Figure 5-49, a repeated attempt attack will be detected after 5 attempts.

### 5.4.10 Security Manager Protocol Data Unit Format

All SMP commands are transmitted over L2CAP using fixed channel with CID 0x0006 in Basic L2CAP mode. SMP has a fixed L2CAP MTU size of 23 octets. Only a single SMP command is sent per L2CAP frame.

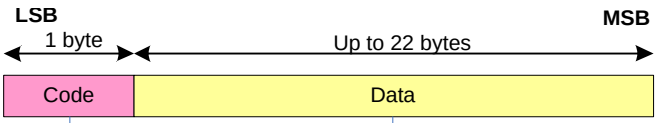



**Figure 5-50: SMP Command PDU**

#### 5.4.10.1 SMP PDU Codes

The table below specifies the SMP codes. A packet with a code not included in the list below is ignored.

| Code | Description |
|---|---|
| 0x00 | Reserved |
| 0x01 | Pairing Request |
| 0x02 | Pairing Response |
| 0x03 | Pairing Confirm |
| 0x04 | Pairing Random |
| 0x05 | Pairing Failed |
| 0x06 | Encryption Information |
| 0x07 | Master Identification |
| 0x08 | Identity Information |
| 0x09 | Identity Address Information |
| 0x0A | Signing Information |
| 0x0B | Security Request |
| 0x0C | Public Key |
| 0x0D | DHKey Check |
| 0x0E | Keypress Notification |

**Table 5-6: SMP Codes**

To ensure there is no lag during the procedure, an SM Timer is implemented allowing maximum 30s of delay between PDU transmissions on a device. This timer is reset and started upon transmission or reception of Pairing Request command. It is reset every time a command is queued for transmission. If the timer expires, failure is indicating to the Host and no more SMP exchanges are allowed. A new SM procedure may start once the physical link has been re-established.

## 5.5 LE Credit Based Channel

The LE credit based Connection, Also Called Connection Oriented Channel (**COC**) is an L2CAP feature managed by GAP. It allows a LE Service to create a dedicated channel on a specific link.

Peer service client should connect to this LE Credit Based Connection before exchanging any packets

GAPM manage list of LE Credit Based Channels created by a profile service. (A peer device cannot connect to a LE Credit Based Channel if it doesn't exist on manager).


**Figure 5-51: LE Credit manager environment structure.**

The manager environment of variable that manages an LE Credit Based Channel is allocated in **ATT_DB heap**.

It contains:

- The LE_PSM (LE Protocol/Service Multiplexer)
- TASK Identifier that manages the Channel.

- Security Level Requirement (Authentication level and Encryption Key Size)

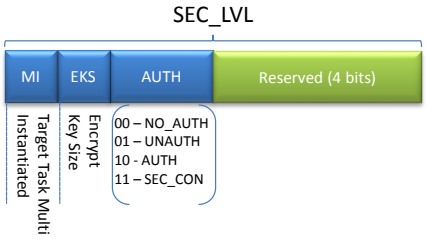



**Figure 5-52: LE Credit Connection security bit field**

GAPC manages LE Credit based channels connection using a list.


**Figure 5-53: LE Credit connection environment structure.**

The environment of variable that manages an LE Credit Based Connection is allocated in **ATT_DB heap**.

It contains:

- The LE_PSM (LE Protocol/Service Multiplexer)
- TASK Identifier that manages reception of packet form peer device.
- Source and destination channel ID
- MTU negotiated for this channel (minimum of MTU of each devices)
- MPS negotiated for this channel (minimum of MPS of each devices)
- Credit allocated by local and peer device for the LE Credit Based Connection.
- Status of the current connection

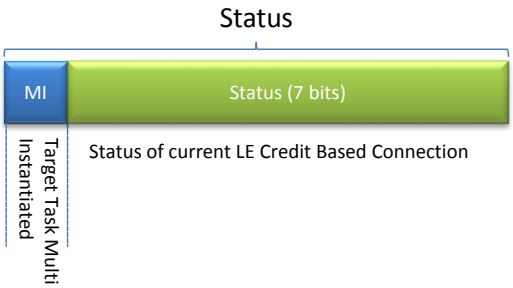



**Figure 5-54: LE Credit Connection status bit field**

### 5.5.1 Connection Creation

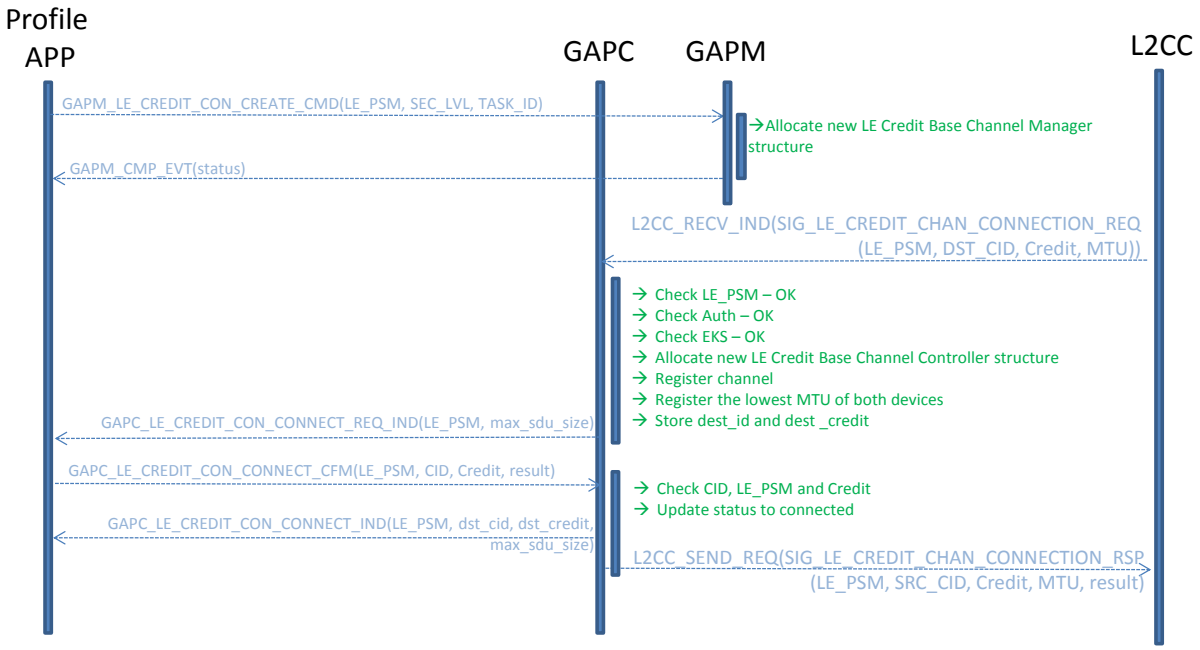



**Figure 5-55: Service view of LE Credit Connection.**

Figure 5-55 shows different step of LE Credit Based Connection creation on service side.

- First a service requests GAP to create a LE Credit Based connection on a specific LE PSM with a specific channel ID and certain number of credit.
- Then peer client establish the LE Credit Based connection using same LE PSM, its channel ID and a credit count.
- Device receive connection request and have to confirm connection with specific result code:
    - Accept
    - Reject Insufficient Resources
    - Reject Not Authorized

**Note:** When Le Credit Based Connection is established, application is informed about the maximum SDU size allowed (MTU - 2)

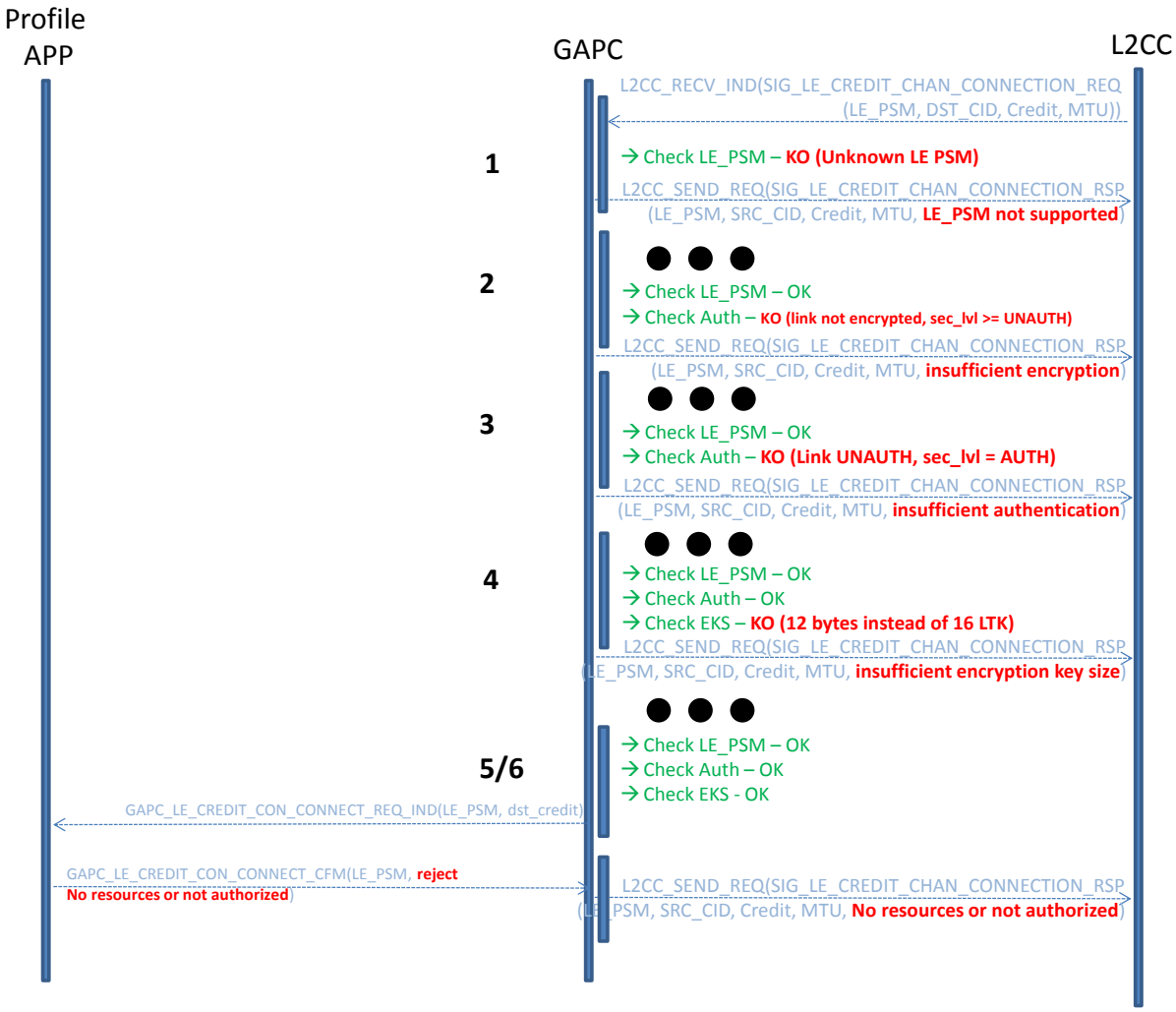



**Figure 5-56: Service Reject Connection creation.**

Figure 5-56 shows possible connection creation errors on service side:

1. LE PSM is Unknown
2. Security Level is set to Unauthenticated or Authenticated but link isn't encrypted
3. Link is encrypted with unsufficient authentication level
4. Link is encrypted with LTK key < 16 bytes and connection requires a 16 bytes LTK
5. Application cannot accept link connection due to insufficient resources.
6. Application cannot accept link connection due because peer device is not authorized.

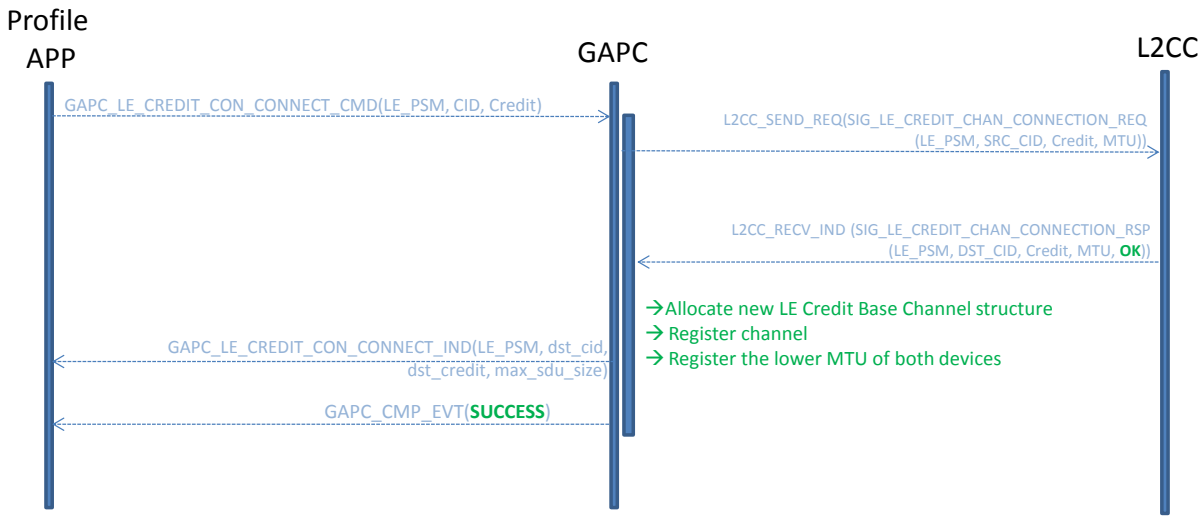



**Figure 5-57: Client view of LE Credit Connection.**

Figure 5-57 shows different step of LE Credit Based Connection creation on client side.

- Using LE PSM, its channel ID and a credit count, a client establishes a LE Credit Based connection created by peer service device.

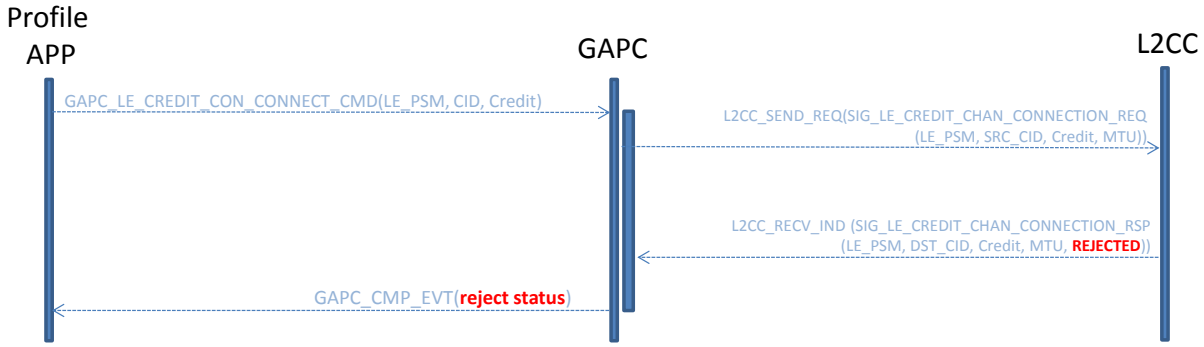



**Figure 5-58: Client LE Credit Connection rejected by peer device.**

### 5.5.2 Disconnection

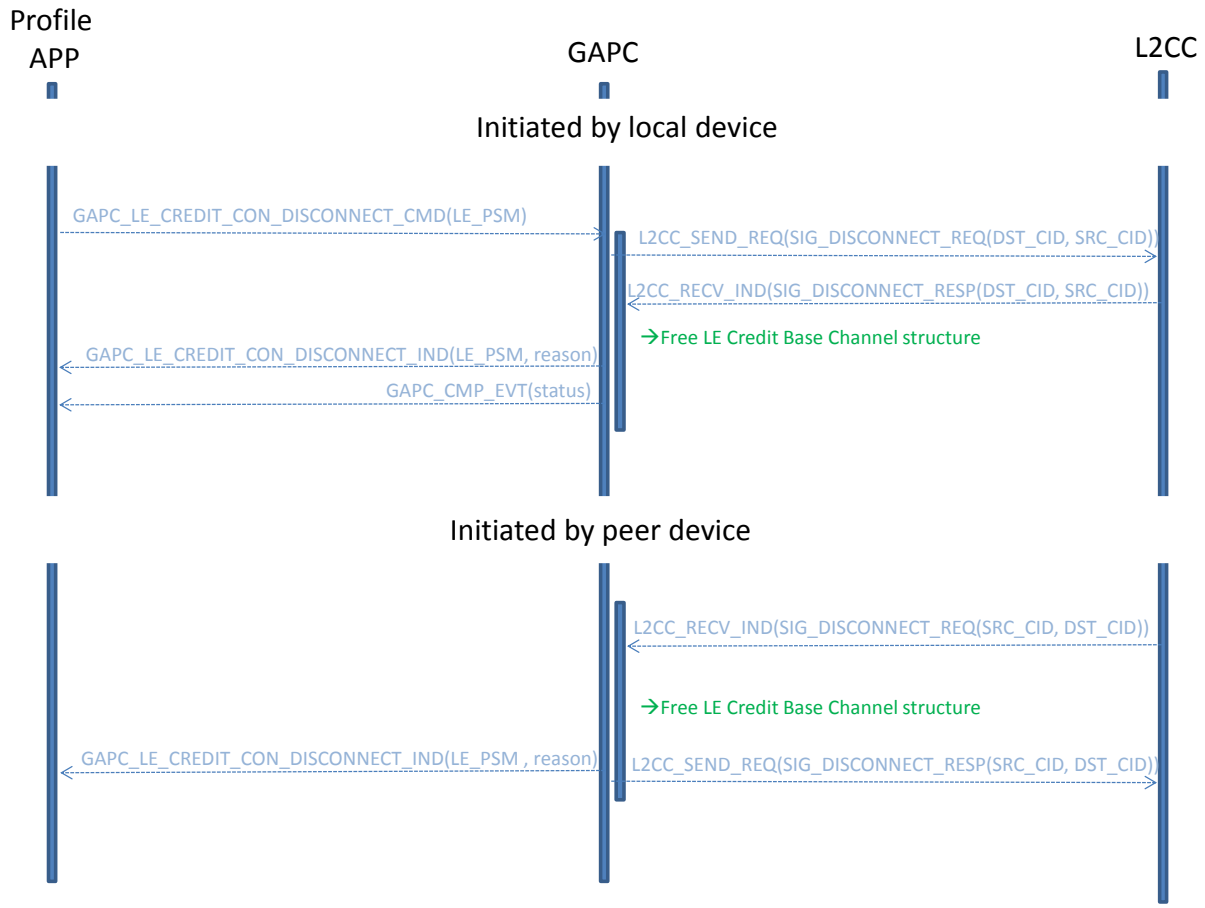



**Figure 5-59: Disconnection overview.**

Figure 5-59 shows how LE Credit Base connection can be stopped from any devices. When disconnection is performed, corresponding environment variables are free and no more data can be sent or received on this channel.

**Note:** Reason of disconnection is provided to Upper Layer such as:

- Local device Initiates Disconnection (no error)
- Remote device Initiates Disconnection
- No More Credit Available
- Invalid MTU (MTU Exceed)
- Invalid Packet Size (MPS exceed)

### 5.5.3 Data Exchange

Data exchange over a LE Credit Based channel is performed directly over L2CC Task.

- Packet Transmission

When sending a packet, L2CC Send procedure verifies with GAPC module (using native functions) if there is still available credit on destination device, and if negotiated MTU is not exceeded. In same time available number of credit is decremented.

Then L2CC task fragment and sent packet to lower layers and notify that packet has been sent with number of credit still available.

- Packet Reception

When receiving a packet, L2CC receive procedure verifies with GAPC module (using native functions) if channel us allocated and there is still available credit on local device. Number of credit is decremented.

Finally L2CC task inform the destination task (used to create the LE Credit Based Channel) that a packet has been received and how many credits are still available.

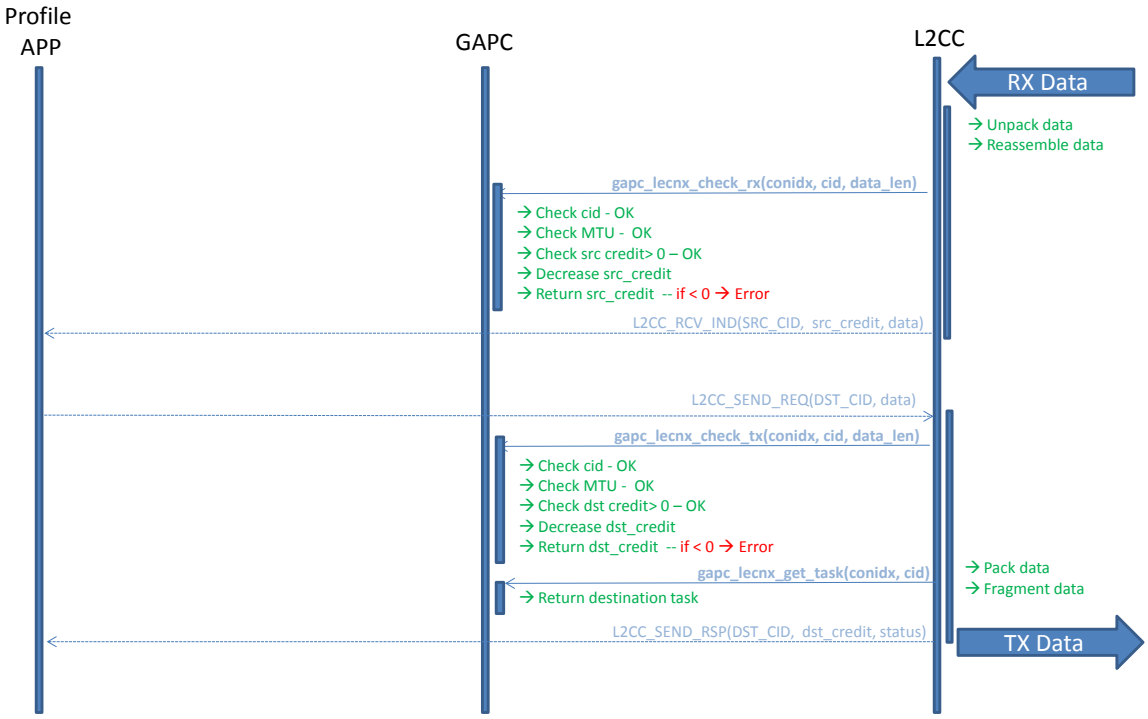



**Figure 5-60: Data exchange between each devices overview.**

### 5.5.4 Credit Management

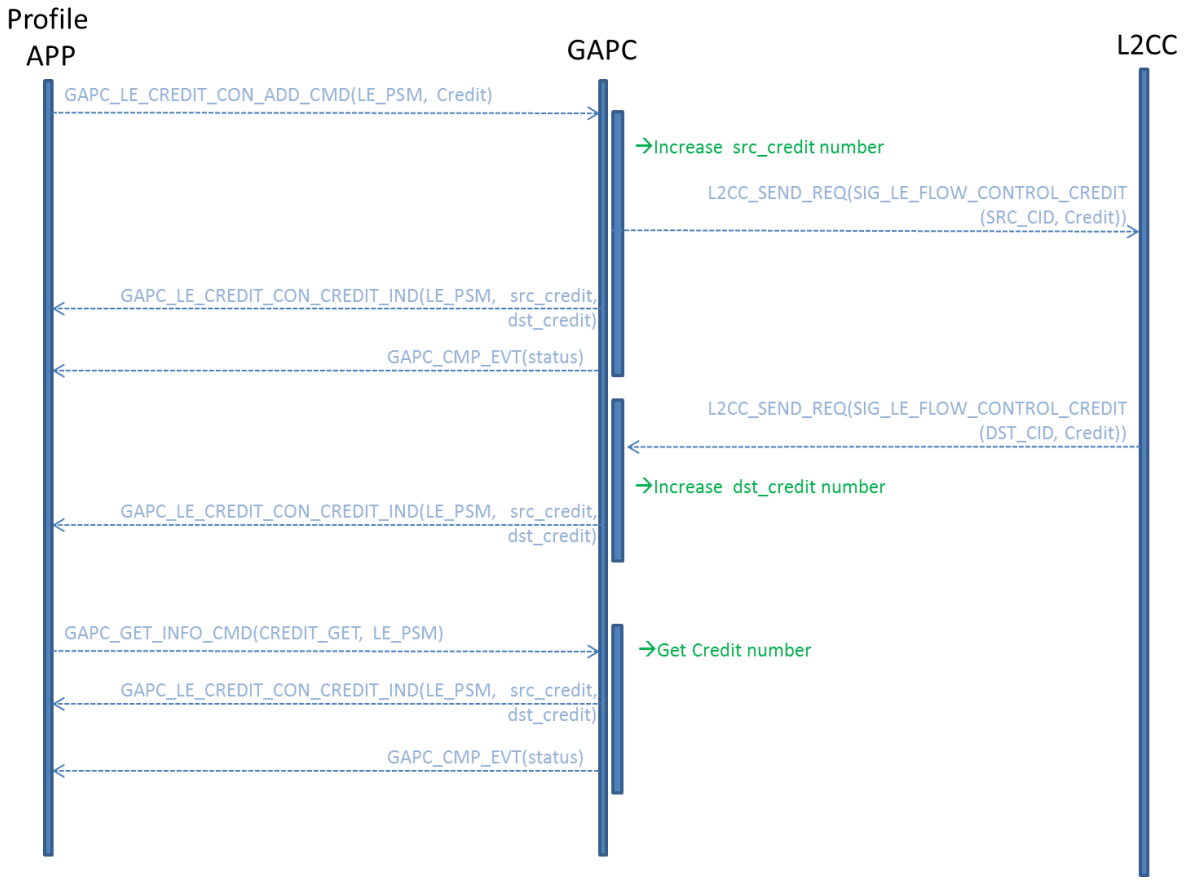



**Figure 5-61: LE Credit management.**

Figure 5-61 shows how to manage credit on a LE Credit Based connection:

- One of devices can increase its local number of credit, by doing this peer device will be informed that credit number has been updated.
- When peer device update its credit count, local device increase destination credit count and task that manage the LE Credit based connection is informed of current source and destination credit count.
- At any time, a task can request information about number of credit available for a specific LE_PSM.

## 5.6 LE Ping

LE Ping Feature is handled by Lower Layers and is described into BLE SW FS (see [7] Chapter 9.6.9). Application can configure or retrieve the Authenticated payload timeout (10ms step) per through GAP interface.

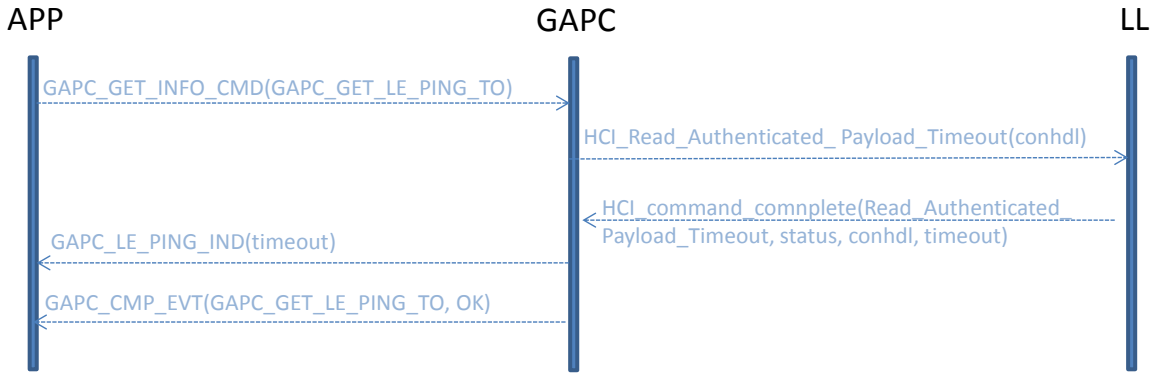



**Figure 5-62: Retrieve LE Ping Authenticated payload timeout from LL.**

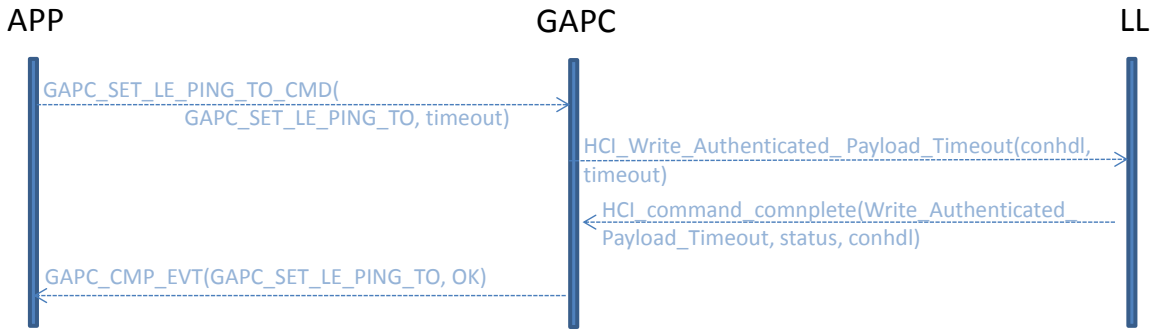



**Figure 5-63: Modify LE Ping Authenticated payload timeout used by LL.**

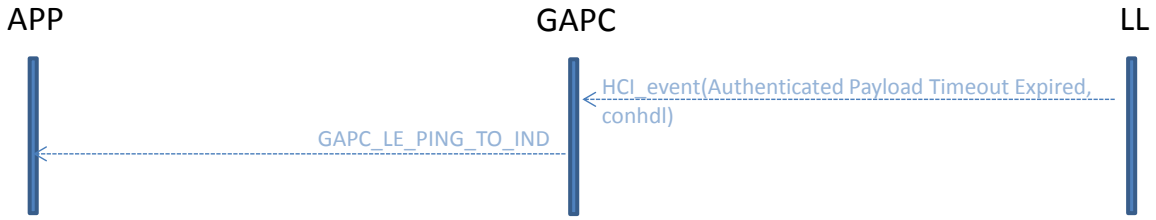



**Figure 5-64: Inform Application about LE Ping Authenticated payload timeout expiration.**

## 5.7 LE Data Packet Length Extension

Size of LE data packets can be negotiated over BLE Link. The preferred LE data packet size is set by application when setting device configuration (see 5.11.2).

Although, when link is established, application can try to (re)negotiate LE data packet size using GAPC_SET_LE_PKT_SIZE_CMD.

When Link size is updated, GAPC_LE_PKT_SIZE_IND is trigged. It doesn't change the fragmentation mechanism in L2CAP since it will use as much as possible fragmentation mechanism provided by lower layers.

## 5.8 Profile Management

Our stack implementation supports a large amount of profiles; for each profiles, a minimum of two tasks is implemented, one for the profile, one for the client. Those tasks should support multiple connections.

In a normal use case, an application should not support all profile and services in same time; number of profile should be limited to a certain amount of profile tasks. To do so, an Array in Generic Access Profile environment variable is used to manage profile tasks. This array contains the task descriptor and a pointer to environment heap.

At start-up application decides profiles that can be started (both client and services tasks). For services task, it means that corresponding attribute database will be loaded, and a minimum authentication level is selected:

- No Authentication required
- Unauthenticated link required
- Authenticated link required
- Secure Connection link required

Profile manage allocation of its task state array, and its environment memory (static and for each links).

Number of profile tasks managed by Generic Access Profile is managed by a compilation flag.

Note: For integration purpose, the customer should allow this to be runtime configurable.

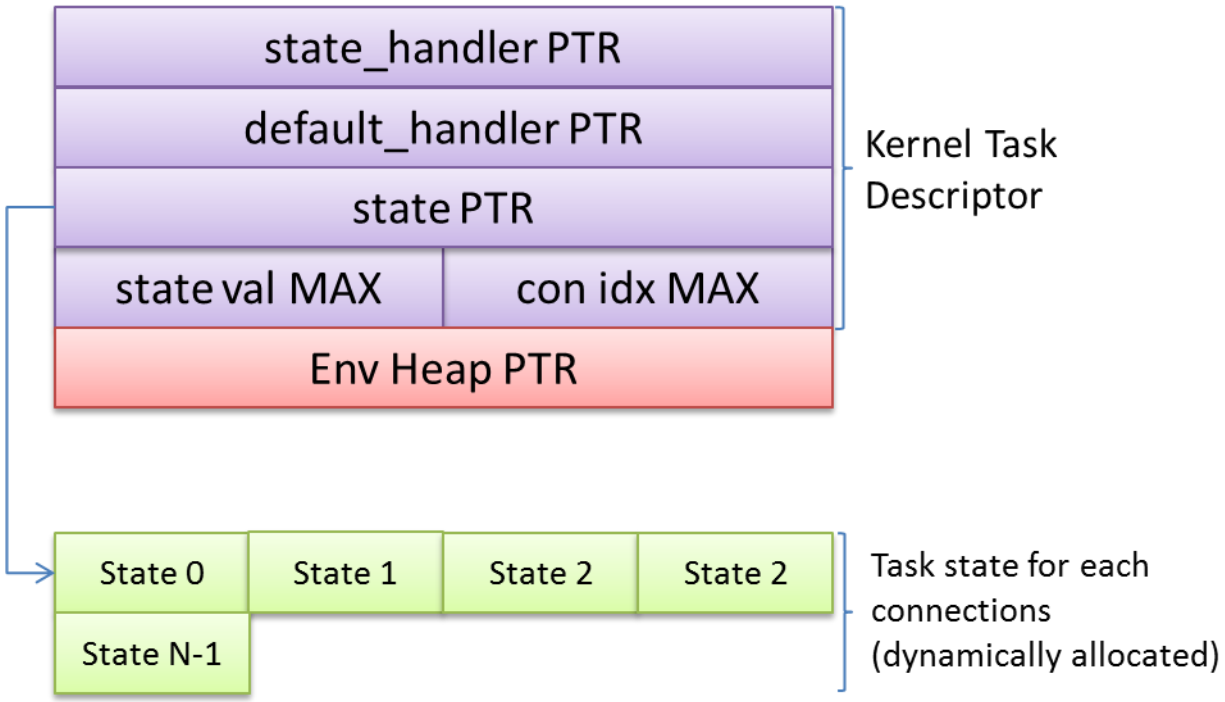



**Figure 5-65: Overview of a Profile Task descriptor in GAP profile task array.**

**Note**: When all profile tasks has been affected, an application requesting to use another profile will receive an OUT of Memory error.

In order to fix profile API, instead of using a task number, a profile id (statically set) is used. This ID should be unique and not be used by another task.

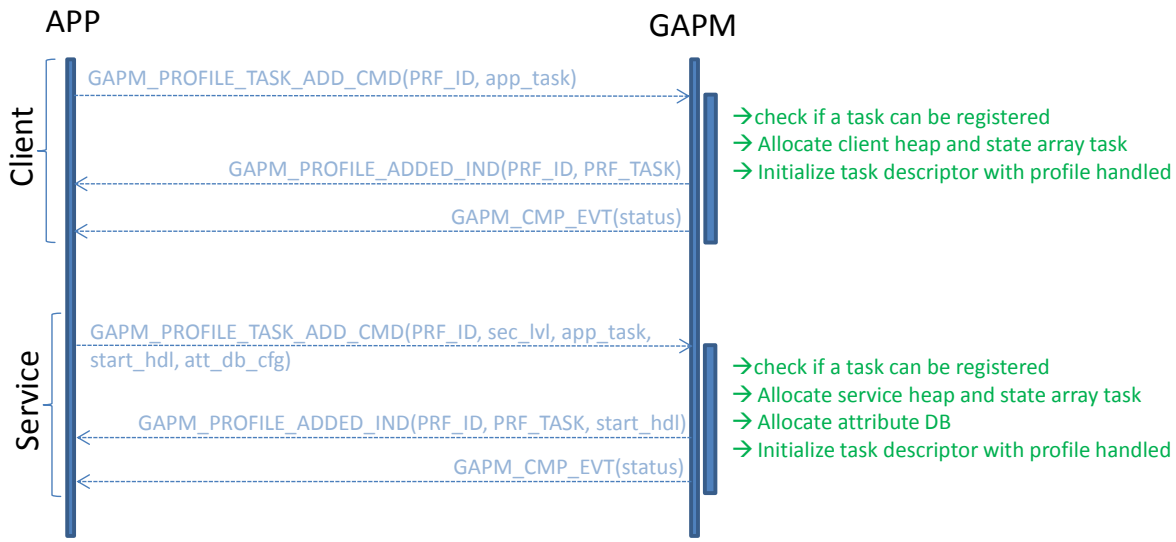



**Figure 5-66: Profile Task registration.**

For the GTL, in GAP environment, a specific array is used to retrieve correspondence between profile identifier and corresponding task id. GAP also provides a native API to retrieve profile id from task id or task id from profile id.

When a profile is registered, it is natively informed about link establishment (to allocate environment) and termination.

**Note**: When system is reset, all registered task are remove and profiles are cleaned-up.

By default Profile task descriptors are initialized without any handler and without any task id.

This ensures that when task is not registered, any message kernel to this task will be ignored.

**Note**: if GTL receive a message on a non-registered profile identifier, it should answer with a generic error message.

When an application has to communicate with a profile task, it has to request its task identifier to gap through its native API.

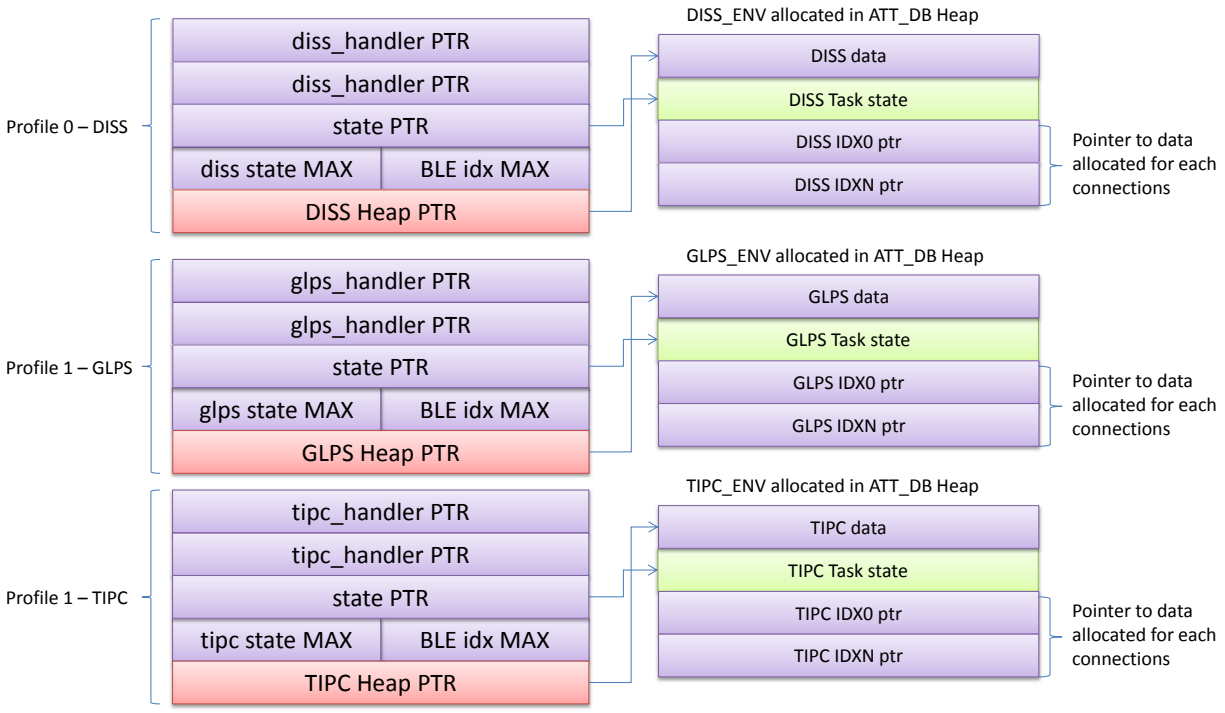



**Figure 5-67: Example of Profile Task registration.**

## 5.9 GAP service database

GAP service (UUID = 0x1800) will be represented as an attribute service in the Attribute database. Depending on the role of the device, certain attribute characteristics are required in the service definition.

| CHARACTERISTICS | GAP ROLE | | | DESCRIPTION |
|---|---|---|---|---|
| | CT | PH | BC/OB | |
| (0x2A00) Device name | m | m | x | Name of the device in UTF-8 format (write optional) |
| (0x2A01) Appearance | m | m | x | Representation of the LE device (write optional) |
| (0x2A04) Preferred conn par | x | o | x | Set of conn parameters preferred by the device |
| (0x2A06) Central Addr Resolution | o | o | x | Central Address Resolution characteristic defines whether the device supports privacy with address resolution |

**Table 5-7: GAP Characteristics**

Those characteristics value are not present into the database. If peer device try to read or write those values, a request will be sent to application. It allows application to manage memory position of those fields.

## 5.10 GAP Environment Variables

### 5.10.1 GAP Manager Environment

| Type | Value | Comment |
|---|---|---|
| **ke_msg*** | CFG operation | Operation used to configure System, use encryption block, get system information |
| **ke_msg*** | AIR operation | Operation used to perform advertising, scanning or connection init activity |

| | | |
|---|---|---|
| **uint16_t** | Start_hdl | Gap Service start handle |
| **gap_sec_key** | IRK | IRK used for resolvable random BD address generation |
| **bd_addr** | addr | Current BD address (private or public) |
| **gap_bdaddr*** | scan_filter | Scan filtering Array |
| **co_list** | LECB channels | List that contains list of LE Credit Based channels |
| **uint8_t** | role | Current device role |
| **uint8_t** | nb_mst_con | Number of master connections |
| **uint8_t** | nb_slave_con | Number of slave connections |
| **uint16_t** | renew_dur | Duration of resolvable address before regenerate it. |
| **uint8_t** | Flags | Flag field for:<br>- Addr is private or public<br>- Host Privacy Enabled<br>- Controller Privacy Enabled<br>- Use resolvable/non resolvable address<br>- Slave preferred param present<br>- Address Renew timer started |

**Table 5-8: GAPM Environment variables**

### 5.10.2 GAP Controller Environment

| Type | Value | Comment |
|---|---|---|
| **ke_msg*** | Link Info operation | Operation used to manage Link info (get link and peer info) |
| **ke_msg*** | Link Param operation | Operation used to manage Link parameters (update parameters) |
| **ke_msg*** | SMP operation | Operation used to manage SMP |
| **ke_msg*** | LECBC operation | Operation used for LE Credit Based Connection |
| **ke_task_id_t** | disc_requester | Task id requested disconnection |
| **uint16_t** | conhdl | Connection handle |
| **gap_sec_key[]** | csrk | CSRK values (Local and remote) |
| **uint32_t[]** | sign_counter | signature counter values (Local and remote) |
| **uint8_t** | key_size | Encryption key size |
| **gap_bdaddr[]** | src | BD Address used for the link that should be kept |
| **smpc_pair_info/ smpc_sign_info** | pair_info/ sign_info | Pairing Information or sign info according to ongoing SMP procedure |
| **uint8_t** | SMP state | State of the current SMP procedure |
| **co_list** | LECB connections | List that contains list of LE Credit Based connections |
| **uint8_t** | fields | Configuration fields:<br>- Link Authorization level<br>- Encrypted Link<br>- Role<br>- Is SMP Timeout Timer running<br>- Is Repeated Attempt Timer running<br>- Has task reached a SMP Timeout |

**Table 5-9: GAPC Environment variables**

### 5.10.3 GAP Profiles Environment

| Type | Value | Comment |
|---|---|---|
| **prf_tasks_env[]** | prf | Array of Profile tasks environment descriptor |

**Table 5-10: GAP Profiles Environment variables**

## 5.11 Device initialization

### 5.11.1 Software Reset

At system start-up, to initialize Software state machines, a SW reset command shall be sent.

This command also initialize attribute database, after a SW reset, device attribute database is empty, Device configuration and Profile configuration shall be performed

### 5.11.2 Device Configuration

At system start-up, after sending software reset command, the device shall be set-up using the Set device configuration command.

Configuration of device can be updated only if there is no on-going connection.

- **Role**: Five role allowed

**Note**: The device can support all roles simultaneously, it means that it can be both master and slave of connection, and start scan and advertising activity in same time.

| Roles | Scan | Advertise | Master Connect | Slave Connect |
|---|---|---|---|---|
| Observer | X | O | O | O |
| Broadcaster | O | X | O | O |
| Peripheral | O | X | O | X |
| Central | X | O | X | O |
| All | X | X | X | X |

**Table 5-11: Device roles**

- **Device Privacy:**
  - Device IRK: Used to generate random address (only valid for Host Privacy 1.1)
  - Privacy managed by host (privacy 1.1), by controller (privacy 1.2) or disabled
  - Renew address timer duration
- **Device Address**: (if privacy disabled or managed by controller)
  - Device address type
  - Device static address (if address type is random)
- **Packet Size:** Maximum MTU allowed by device (mini = 23 bytes, max = 2048)
- **GAP DB configuration:**
  - GAP DB start handle (0x0000 – dynamically allocated)
  - Appearance Write Permissions
  - Device Name Write Permissions + Device name max length
  - Peripheral Preferred Connection Parameters present + Read Permissions
- **GATT DB configuration:**
  - GATT DB start handle (0x0000 – dynamically allocated)
  - Service changed characteristic present.

**Note:** Set device configuration command recreate GAP and GATT database.

# 6 Profiles Functionalities

Bluetooth LE profiles reside on top of the host protocols and generic profiles (GAP and GATT).

Support of an LE profile depends on its specification availability, from Bluetooth SIG.

The FS of these profile implementations are beyond the scope of this document.

Some guidelines for profile implementation:

- Due to BLE Topology, Client and Server Profiles are multi-instantiated tasks.
- Profiles have to manage environment memory by allocating it in ATT Heap. Memory for dedicated link and general configuration.
- If not enabled by GAP, the profile RAM footprint should be equals to zero.
- Service profile should be ready by default, enable message should be used to restore bond data of a known device.
- A Profile is not aware of its task identifier, in message handler, destination id should be used to retrieve its task identifier, or eventually request it to GAP through the native API.
- It's recommended to use operation mechanism (see 7.2) in order to optimize profile memory usage.

**Note**: Profile should be only on top of GATT API. Management of connection and advertising data should be handled by the application.

# 7 Memory Optimization

BLE Host software memory is optimized allowing system to shut down some memory block when sleeping between BLE events. This feature could be used thanks to kernel memory heap segmentation (see RW Kernel FS [3])

## 7.1 Connection Oriented Task

Environment variable for tasks related to a connection are allocated at connection and remove as soon as connection is stopped.

Those environment variable, should not contain value used only during specific operations such as pairing or connection update

These variables should be allocated in kernel environment heap.

## 7.2 Operation Model

An operation is a command that should be executed by a task. This command contains parameters that should be used during its execution. Instead of copying the parameter in task environment, Command message is stored until its execution is finished.

Thanks to this model:

- Command parameters could be easily reused
- Operation pointer can be used for command flow control
- Command message handler can be implemented as a state machine by rescheduling command in the kernel.

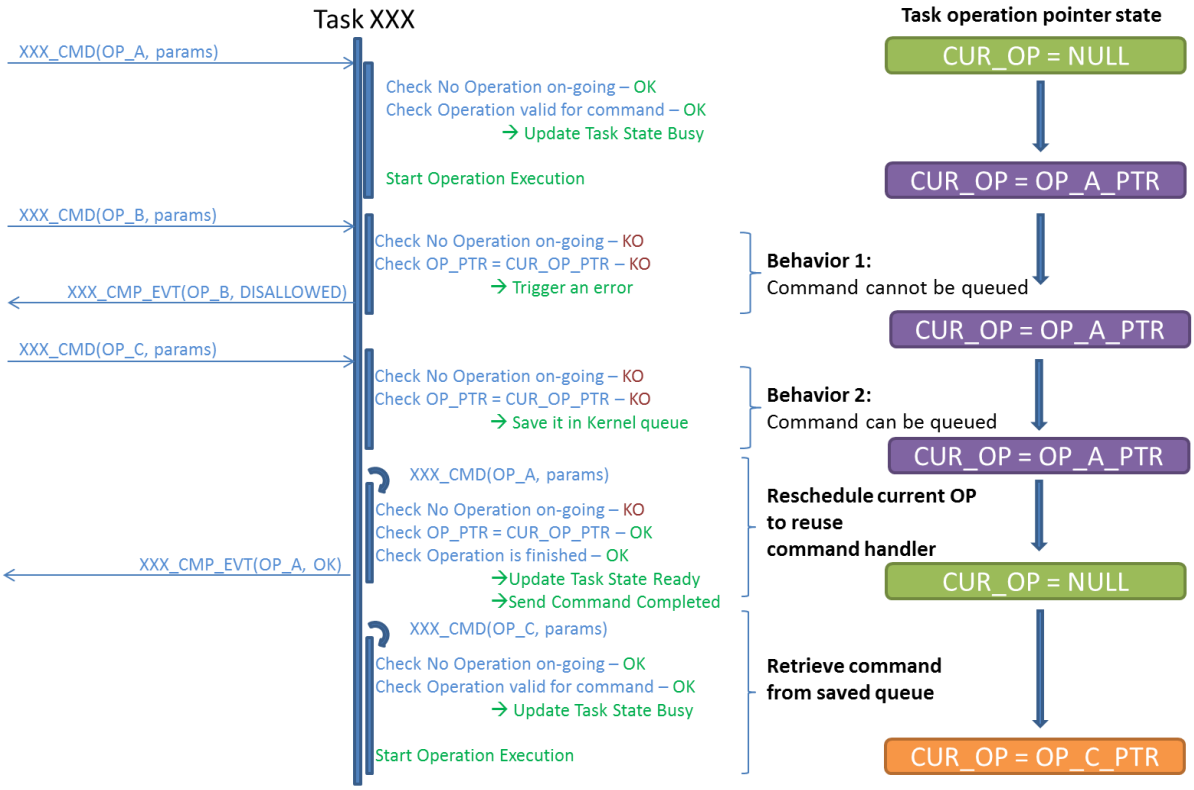



**Figure 7-1: Operation Life cycle**

# References


| | | | | |
|---|---|---|---|---|
| **[1]** | **Title** | Specification of the Bluetooth System v4.2 | | |
| | **Reference** | Bluetooth Specification LE LL and BR/EDR | | |
| | **Version** | V4.1 | **Date** | December 2, 2014 |
| | **Source** | Bluetooth SIG | | |

| | | | | |
|---|---|---|---|---|
| **[2]** | **Title** | Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication | | |
| | **Reference** | SP_800-38B | | |
| | **Version** | | **Date** | 2005-05 |
| | **Source** | National Institute of Standards and Technology | | |

| | | | | |
|---|---|---|---|---|
| **[3]** | **Title** | RivieraWaves Kernel | | |
| | **Reference** | RW-BT-KERNEL-SW-FS | | |
| | **Version** | 1.1 | **Date** | 2012-05-23 |
| | **Source** | RivieraWaves | | |

| | | | | |
|---|---|---|---|---|
| **[4]** | **Title** | RW HCI Software | | |
| | **Reference** | RW-HCI-SW-FS | | |
| | **Version** | 0.1 | **Date** | 2014-03-05 |
| | **Source** | RivieraWaves | | |

| | | | | |
|---|---|---|---|---|
| **[5]** | **Title** | GAP Interface Specification | | |
| | **Reference** | RW-BLE-GAP-IS | | |
| | **Version** | 8.00 | **Date** | TBD |
| | **Source** | RivieraWaves | | |

| | | | | |
|---|---|---|---|---|
| **[6]** | **Title** | GATT Interface Specification | | |
| | **Reference** | RW-BLE-GATT-IS | | |
| | **Version** | 8.00 | **Date** | TBD |
| | **Source** | RivieraWaves | | |

| | | | | |
|---|---|---|---|---|
| **[7]** | **Title** | RW-BLE Link Layer Software | | |
| | **Reference** | RW-BLE-LL-SW-FS | | |
| | **Version** | 8.0 | **Date** | TBD |
| | **Source** | RivieraWaves | | |