ARM® Cordio Stack

ARM-EPM-115880 1.0

Device Manager API

Confidential

ARM® Cordio Stack Device Manager API

Reference Manual

Release Information

The following changes have been made to this book:

Document History

Date	Issue	Confidentiality	Change
21 January 2016	-	Confidential	First Wicentric release for 1.6 as 2009-0008.
1 March 2016	A	Confidential	First ARM release for 1.6.
24 August 2016	A	Confidential	AUSPEX # / API Update

Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Contents

1 Preface

This preface introduces the *Cordio Stack Device Manager API Reference Manual*.

1.1 About this book

This document describes the *Device Manager* (DM) API and lists the API functions and their parameters.

1.1.1 Intended audience

This book is written for experienced software engineers who might or might not have experience with ARM products. Such engineers typically have experience of writing Bluetooth applications but might have limited experience of the Cordio software stack.

It is also assumed that the readers have access to all necessary tools.

1.1.2 Using this book

This book is organized into the following chapters:

- `Introduction`
  Read this for an overview of the API.
- `Main Interface`
  Read this for a list of common main interfaces used in the API.
- `Advertising and Device Visibility`
  Read this for a description advertising and device visibility functions.
- `Scanning and Device Discovery`
  Read this for a description of scanning and device discovery functions.
- `Connection Management`
  Read this for a description of connection management functions.
- `Local Device Management`
  Read this for a description of local device management functions.
- `Security Management`
  Read this for a description of security management functions.
- `Privacy`
  Read this for a description of privacy functions.
- `Scenarios`
  Read this for an overview of how APIs are used in different scenarios.
- `Revisions`
  Read this chapter for descriptions of the changes between document versions.

1.1.3 Terms and abbreviations

For a list of ARM terms, see the ARM glossary.

Terms specific to the Cordio software are listed below:

Term	Description
ACL	Asynchronous Connectionless data packet
AD	Advertising Data
AE	Advertising Extensions
ARQ	Automatic Repeat reQuest
ATT	Attribute Protocol, also attribute protocol software subsystem
ATTC	Attribute Protocol Client software subsystem
ATTS	Attribute Protocol Server software subsystem
CCC or CCCD	Client Characteristic Configuration Descriptor
CID	Connection Identifier
CSRK	Connection Signature Resolving Key
DM	Device Manager software subsystem
GAP	Generic Access Profile
GATT	Generic Attribute Profile
HCI	Host Controller Interface
IRK	Identity Resolving Key
JIT	Just In Time
L2C	L2CAP software subsystem
L2CAP	Logical Link Control Adaptation Protocol
LE	(Bluetooth) Low Energy
LL	Link Layer
LLPC	Link Layer Control Protocol
LTK	Long Term Key
MITM	Man In The Middle pairing (authenticated pairing)
OOB	Out Of Band data
SMP	Security Manager Protocol, also security manager protocol software subsystem
SMPI	Security Manager Protocol Initiator software subsystem
SMPR	Security Manager Protocol Responder software subsystem
STK	Short Term Key
WSF	Wireless Software Foundation software service and porting layer.

1.1.4 Conventions

The following table describes the typographical conventions:

Typographical conventions

Style	Purpose
Italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`MONOSPACE`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`MONO`SPACE	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace` *`italic`*	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

1.1.5 Additional reading

This section lists publications by ARM and by third parties.

See *Infocenter* for access to ARM documentation.

Other publications

This section lists relevant documents published by third parties:

- Bluetooth SIG, "*Specification of the Bluetooth System*", Version 4.2, December 2, 2015.

1.2 Feedback

ARM welcomes feedback on this product and its documentation.

1.2.1 Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- The title.
- The number, ARM-EPM-115146.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note: ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

2 Introduction

This document describes the API of the *Device Manager* (DM) subsystem. The device manager is responsible for many important operations of the protocol stack such as:

- Advertising and device visibility.
- Scanning and device discovery.
- Connection management.
- Security management.
- Local device management.

3 Main Interface

3.1 Constants and data types

3.1.1 Device Role

This parameter identifies the device role.

Table 1 Device role parameter

Name	Value	Description
DM_ROLE_MASTER	0	Role is master.
DM_ROLE_SLAVE	1	Role is slave.

3.1.2 Discoverability mode

This parameter sets the GAP discoverability mode.

Table 2 Discoverability Mode

Name	Value	Description
DM_DISC_MODE_NONE	0	GAP non-discoverable. Peer devices performing GAP discovery cannot discover this device.
DM_DISC_MODE_LIMITED	1	GAP limited discoverable mode. Peer devices performing GAP limited discovery can discover this device.
DM_DISC_MODE_GENERAL	2	GAP general discoverable mode. Peer devices performing GAP limited or general discovery can discover this device.

3.1.3 Advertising type

The advertising type indicates the connectable and discoverable nature of the advertising packets transmitted by a device.

Table 3 Advertising type

Name	Value	Description
DM_ADV_CONN_UNDIRECT	0	Connectable undirected advertising. Peer devices can scan and connect to this device.
DM_ADV_CONN_DIRECT	1	Connectable directed advertising. Only a specified peer device can connect to this device.
DM_ADV_SCAN_UNDIRECT	2	Scannable undirected advertising. Peer devices can scan this device but cannot connect.
DM_ADV_NONCONN_UNDIRECT	3	Non-connectable undirected advertising. Peer

		devices cannot scan or connect to this device.
DM_ADV_CONN_DIRECT_LO_DUTY	4	Connectable directed low duty cycle advertising.

3.1.4 Address type

The address type indicates whether an address is public or random.

Table 4 Address type

Name	Value	Description
DM_ADDR_PUBLIC	0	Public address.
DM_ADDR_RANDOM	1	Random address.
DM_ADDR_PUBLIC_IDENTITY	2	Public identity address
DM_ADDR_RANDOM_IDENTITY	3	Random (static) identity address
DM_ADDR_RANDOM_UNRESOLVED	0xFE	Random device address (controller can't resolve)
DM_ADDR_NONE	0xFF	No address provided (anonymous)

3.1.5 Advertising and scan intervals

Advertising and scan intervals in this API are specified in 0.625 ms units.

3.2 Functions

3.2.1 DmRegister()

Register a callback with DM for scan and advertising events.

Syntax:

```
void DmRegister(dmCback_t cback)
```

Where:

- cback: Client callback function. See 3.3.1.

3.2.2 DmFindAdType()

Find an advertising data element in the given advertising or scan response data.

Syntax:

```
uint8_t *DmFindAdType(uint8_t adType, uint16_t dataLen, uint8_t *pData)
```

Where:

- adType: Advertising data element type to find. See 4.2.
- dataLen: Data length.
- pData: Pointer to advertising or scan response data.

This function returns a pointer to the advertising data element byte array or NULL if not found.

3.3 Callback interface

3.3.1 (*dmCback_t)()

This callback function sends DM events to the client.

Syntax:

```
void (*dmCback_t)(dmEvt_t *pDmEvt)
```

Where:

- `pDmEvt`: Pointer to DM event structure.

3.3.2 Callback events

The following callback event values are passed in the DM event structure.

Table 5 Callback events

Name	Description
DM_RESET_CMPL_IND	Reset complete.
DM_ADV_START_IND	Advertising started.
DM_ADV_STOP_IND	Advertising stopped.
DM_ADV_NEW_ADDR_IND	New resolvable address has been generated.
DM_SCAN_START_IND	Scanning started.
DM_SCAN_STOP_IND	Scanning stopped.
DM_SCAN_REPORT_IND	Scan data received from peer device.
DM_CONN_OPEN_IND	Connection opened.
DM_CONN_CLOSE_IND	Connection closed.
DM_CONN_UPDATE_IND	Connection update complete.
DM_SEC_PAIR_CMPL_IND	Pairing completed successfully.
DM_SEC_PAIR_FAIL_IND	Pairing failed or other security failure.
DM_SEC_ENCRYPT_IND	Connection encrypted.
DM_SEC_ENCRYPT_FAIL_IND	Encryption failed.
DM_SEC_AUTH_REQ_IND	PIN or OOB data requested for pairing.
DM_SEC_KEY_IND	Security key indication.
DM_SEC_LTK_REQ_IND	LTK requested for encyption.

DM_SEC_PAIR_IND	Incoming pairing request from master.
DM_SEC_SLAVE_REQ_IND	Incoming security request from slave.
DM_SEC_CALC_OOB_IND	Result of OOB Confirm Calculation Generation.
DM_SEC_ECC_KEY_IND	Result of ECC Key Generation.
DM_SEC_COMPARE_IND	Result of Just Works/Numeric Comparison Compare Value calculation.
DM_SEC_KEYPRESS_IND	Keypress indication from peer in passkey security.
DM_PRIV_RESOLVED_ADDR_IND	Private address resolved.
DM_CONN_READ_RSSI_IND	Connection RSSI read.
DM_PRIV_ADD_DEV_TO_RES_LIST_IND	Device added to resolving list.
DM_PRIV_REM_DEV_FROM_RES_LIST_IND	Device removed from resolving list.
DM_PRIV_CLEAR_RES_LIST_IND	Resolving list cleared.
DM_PRIV_READ_PEER_RES_ADDR_IND	Peer resolving address read.
DM_PRIV_READ_LOCAL_RES_ADDR_IND	Local resolving address read.
DM_PRIV_SET_ADDR_RES_ENABLE_IND	Address resolving enable set.
DM_REM_CONN_PARAM_REQ_IND	Remote connection parameter requested.
DM_CONN_DATA_LEN_CHANGE_IND	Data length changed.
DM_CONN_WRITE_AUTH_TO_IND	Write authenticated payload complete.
DM_CONN_AUTH_TO_EXPIRED_IND	Authenticated payload timeout expired.
DM_PHY_READ_IND	Read PHY
DM_PHY_SET_DEF_IND	Set default PHY
DM_PHY_UPDATE_IND	PHY update
DM_ADV_SET_START_IND	Advertising set(s) started
DM_ADV_SET_STOP_IND	Advertising set(s) stopped
DM_SCAN_REQ_RCVD_IND	Scan request received
DM_EXT_SCAN_START_IND	Extended scanning started
DM_EXT_SCAN_STOP_IND	Extended scanning stopped

DM_EXT_SCAN_REPORT_IND	Extended scan data received from peer device
DM_ERROR_IND	General error.
DM_VENDOR_SPEC_IND	Vendor specific event.

4 Advertising and Device Visibility

The DM interface for advertising and device visibility configures, enables, and disables the advertising procedure. A device advertises when it wishes to connect to or be discovered by other devices. Devices may also advertise to simply broadcast data.

This interface can only be used when operating as a slave.

4.1 Constants and data types

4.1.1 Data Location

This parameter indicates whether data is located in the advertising data or the scan response data.

Table 6 Callback events

Name	Value	Description
DM_DATA_LOC_ADV	0	Locate data in the advertising data.
DM_DATA_LOC_SCAN	1	Locate data in the scan response data.

4.2 Advertising data element types

This parameter indicates the type of advertising data element.

Table 7 Advertising data element types

Name	Description
DM_ADV_TYPE_FLAGS	Flag bits.
DM_ADV_TYPE_16_UUID_PART	Partial list of 16 bit UUIDs.
DM_ADV_TYPE_16_UUID	Complete list of 16 bit UUIDs.
DM_ADV_TYPE_32_UUID_PART	Partial list of 32 bit UUIDs.
DM_ADV_TYPE_32_UUID	Complete list of 32 bit UUIDs.
DM_ADV_TYPE_128_UUID_PART	Partial list of 128 bit UUIDs.
DM_ADV_TYPE_128_UUID	Complete list of 128 bit UUIDs.
DM_ADV_TYPE_SHORT_NAME	Shortened local name.
DM_ADV_TYPE_LOCAL_NAME	Complete local name.
DM_ADV_TYPE_TX_POWER	TX power level.
DM_ADV_TYPE_SM_TK_VALUE	Security manager TK value
DM_ADV_TYPE_SM_OOB_FLAGS	Security manager OOB flags

DM_ADV_TYPE_CONN_INTERVAL	Slave preferred connection interval.
DM_ADV_TYPE_SIGNED_DATA	Signed data.
DM_ADV_TYPE_16_SOLICIT	Service solicitation list of 16 bit UUIDs.
DM_ADV_TYPE_128_SOLICIT	Service solicitation list of 128 bit UUIDs.
DM_ADV_TYPE_SERVICE_DATA	Service data.
DM_ADV_TYPE_PUBLIC_TARGET	Public target address.
DM_ADV_TYPE_RANDOM_TARGET	Random target address.
DM_ADV_TYPE_APPEARANCE	Device appearance.
DM_ADV_TYPE_ADV_INTERVAL	Advertising interval
DM_ADV_TYPE_BD_ADDR	LE Bluetooth device address
DM_ADV_TYPE_ROLE	LE role
DM_ADV_TYPE_32_SOLICIT	Service solicitation list of 32 bit UUIDs
DM_ADV_TYPE_SVC_DATA_32	Service data – 32-bit UUID
DM_ADV_TYPE_SVC_DATA_128	Service data – 128-bit UUID
DM_ADV_TYPE_LESC_CONFIRM	LE secure connection confirm value
DM_ADV_TYPE_LESC_RANDOM	LE secure connection random value
DM_ADV_TYPE_URI	URI
DM_ADV_TYPE_MANUFACTURER	Manufacturer specific data.

4.3 Advertising channel map

This parameter indicates the advertising channel map.

Table 8 Advertising channel map

Name	Description
DM_ADV_CHAN_37	Advertising channel 37.
DM_ADV_CHAN_38	Advertising channel 38.
DM_ADV_CHAN_39	Advertising channel 39.
DM_ADV_CHAN_ALL	All advertising channels.

4.4 Functions

4.4.1 DmAdvInit()

Initialize DM advertising. This function is typically called once at system startup.

Syntax:

```
void DmAdvInit(void)
```

4.4.2 DmExtAdvInit()

Initialize DM extended advertising. This function is typically called once at system startup.

Syntax:

```
void DmExtAdvInit(void)
```

4.4.3 DmAdvStart()

This function is called to start advertising using the given advertising set and duration.

Syntax:

```
void DmAdvStart(uint8_t numSets, uint8_t *pAdvHandle, uint16_t *pDuration, uint8_t
*pMaxEaEvents)
```

Where:

- `numSets`: Number of advertising sets to enable.
- `pAdvHandle`: Advertising handle array.
- `pDuration`: Advertising duration (in milliseconds) array.
- `pMaxEaEvents`: Maximum number of extended advertising events array.

If advertising is started successfully the client's callback function is called with a `DM_ADV_START_IND` event. If advertising fails to start for any reason the client's callback function is called with a `DM_ADV_STOP_IND` event. The client's callback function is also called with a `DM_ADV_STOP_IND` event if the advertising duration expires or `DmAdvStop()` is called.

4.4.4 DmAdvStop()

This function is called to stop advertising. When advertising is stopped the client's callback function is called with a `DM_ADV_STOP_IND` event.

Syntax:

```
void DmAdvStop(uint8_t numSets, uint8_t *pAdvHandles)
```

Where:

- `numSets`: Number of advertising sets to enable.
- `pAdvHandles`: Advertising handles array.

4.4.5 DmAdvSetInterval()

This function sets the minimum and maximum advertising intervals. This function should only be called when advertising is stopped.

Syntax:

```
void DmAdvSetInterval(uint8_t advHandle, uint16_t intervalMin, uint16_t
        intervalMax)
```

Where:

- advHandle: Advertising handle
- intervalMin: Minimum advertising interval. See 3.1.4.
- intervalMax: Maximum advertising interval. See 3.1.4.

4.4.6 DmAdvSetChannelMap()

This function is used to include or exclude certain channels from the advertising channel map. This function should only be called when advertising is stopped.

Syntax:

```
void DmAdvSetChannelMap(uint8_t advHandle, uint8_t channelMap)
```

Where:

- advHandle: Advertising handle
- channelMap: Advertising channel map. See 4.3.

4.4.7 DmAdvSetData()

This function sets the advertising or scan response data to the given data. The data will replace any existing data already present with the same advertising data type.

Syntax:

```
void DmAdvSetData(uint8_t advHandle, uint8_t op, uint8_t location, uint8_t len,
        uint8_t *pData)
```

Where:

- advHandle: Advertising handle
- op: Data operation
- location: Data location. See 4.1.1.
- len: Length of the data. Maximum length is 31 bytes.
- pData: Pointer to the data.

4.4.8 DmAdvSetAddrType ()

Set the local address type used while advertising. This function can be used to configure advertising to use a random or private address.

Syntax:

```
void DmAdvSetAddrType(uint8_t addrType)
```

Where:

- addrType: Address type. See 3.1.4.

4.4.9 DmAdvSetAdValue()

Set the value of an advertising data element in the given advertising or scan response data. If the

element already exists in the data then it is replaced with the new value. If the element does not exist in the data it is appended to it, space permitting.

Syntax:

```
Bool DmAdvSetAdValue(uint8_t adType, uint8_t len, uint8_t *pValue, uint8_t
        *pAdvDataLen, uint8_t *pAdvData, uint16_t advDataBufLen)
```

Where:

- `adType`: Advertising data element type.
- `len`: Length of the value. Maximum length is 29 bytes.
- `pValue`: Pointer to the value.
- `pAdvDataLen`: Advertising or scan response data length. The new length is returned in this parameter.
- `pAdvData`: Pointer to advertising or scan response data.
- `advDataBufLen`: Length of the advertising or scan response data buffer maintained by the application.

Returns TRUE if the element was successfully added to the data, FALSE otherwise.

4.4.10 DmAdvSetName()

Set the device name in the given advertising or scan response data. If the name can only fit in the data if it is shortened, the name is shortened and the AD type is changed to `DM_ADV_TYPE_SHORT_NAME`.

Syntax:

```
Bool DmAdvSetName(uint8_t len, uint8_t *pValue, uint8_t *pAdvDataLen, uint8_t
        *pAdvData, uint16_t advDataBufLen)
```

Where:

- `len`: Length of the name. Maximum length is 29 bytes.
- `pValue`: Pointer to the name in UTF-8 format.
- `pAdvDataLen`: Advertising or scan response data length. The new length is returned in this parameter.
- `pAdvData`: Pointer to advertising or scan response data.
- `advDataBufLen`: Length of the advertising or scan response data buffer maintained by the application.

Returns TRUE if the element was successfully added to the data, FALSE otherwise.

4.4.11 DmAdvPrivInit()

Initialize private advertising. This function is typically called once at system startup to enable the use of advertising with a private resolvable address.

Syntax:

```
void DmAdvPrivInit(void)
```

4.4.12 DmAdvPrivStart()

Start using a private resolvable address and start periodic generation of a new address.

When a new address is generated the client's callback function is called with a `DM_ADV_NEW_ADDR_IND`

event. The application must wait to receive this event once before starting advertising.

To stop using a private resolvable address call function `DmAdvPrivStop()`.

This function should not be used when the device is operating as a master, as master devices are forbidden from using a private resolvable address.

Syntax:

```
void DmAdvPrivStart(uint16_t changeInterval)
```

Where:

- `changeInterval`: Interval between automatic address changes, in seconds.

4.4.13 DmAdvPrivStop()

Stop using a private resolvable address.

Syntax:

```
void DmAdvPrivStop(void)
```

4.5 Callback interface

4.5.1 DM_ADV_START_IND: Advertising started

Callback event for advertising started.

Table 9 Advertising started

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.

4.5.2 DM_ADV_STOP_IND: Advertising stopped

Callback event for advertising stopped.

Table 10 Advertising stopped

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.

4.5.3 DM_ADV_NEW_ADDR_IND: New resolvable address has been generated

Callback event for new resolvable address has been generated.

Table 11 New resolvable address

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`bdAddr_t`	`addr`	New resolvable private address.

bool_t	firstTime	TRUE when address is generated for the first time.

5 Scanning and Device Discovery

The DM scanning and device discovery interface configures, enables, and disables the scanning procedure. A device scans when it wishes to discover or connect to other devices. A device may also scan simply to receive broadcast advertisements.

This interface can only be used when operating as a master.

5.1 Constants and data types

5.1.1 Scan type

This parameter indicates the scan type. A passive scan only receives advertising packets. An active scan receives advertising packets and scan response packets.

Table 12 Scan type

Name	Value	Description
DM_SCAN_TYPE_PASSIVE	0	Passive scan.
DM_SCAN_TYPE_ACTIVE	1	Active scan.

5.2 Functions

5.2.1 DmScanInit()

Initialize DM scanning. This function is typically called once at system startup.

Syntax:

```
void DmScanInit(void)
```

5.2.2 DmExtScanInit()

Initialize DM AE scanning. This function is typically called once at system startup.

Syntax:

```
void DmExtScanInit(void)
```

5.2.3 DmScanStart()

This function is called to start scanning. A scan is performed using the given discoverability mode, scan type, and duration.

Syntax:

```
void DmScanStart(uint8_t scanPhys, uint8_t mode, uint8_t scanType, bool_t
         filterDup, uint16_t duration, uint16_t period)
```

Where:

- scanPhys: Scanner PHYs.
- mode: Discoverability mode. See 3.1.1.
- scanType: Scan type. See 5.1.1.

- filterDup: Filter duplicates. Set to TRUE to filter duplicate responses received from the same device. Set to FALSE to receive all responses.
- duration: The scan duration, in milliseconds. If set to zero, scanning will continue until DmScanStop() is called.
- period: Period (only applicable to AE).

If scanning is started successfully the client's callback function is called with a DM_SCAN_START_IND event. If scanning fails to start for any reason the client's callback function is called with a DM_SCAN_STOP_IND event. The client's callback function is also called with a DM_SCAN_STOP_IND event if the scan duration expires or DmScanStop() is called.

Example for GAP limited discovery:

```
DmScanStart(HCI_SCAN_PHY_LE_1M_BIT, DM_DISC_MODE_LIMITED,
DM_SCAN_TYPE_ACTIVE, TRUE, 10240, 0);
```

Example for GAP general discovery:

```
DmScanStart(HCI_SCAN_PHY_LE_1M_BIT, DM_DISC_MODE_GENERAL,
DM_SCAN_TYPE_ACTIVE, TRUE, 10240, 0);
```

Example for GAP observe procedure:

```
DmScanStart(HCI_SCAN_PHY_LE_1M_BIT, DM_DISC_MODE_NONE,
DM_SCAN_TYPE_PASSIVE, FALSE, 0, 0);
```

5.2.4 DmScanStop()

This function is called to stop scanning. When scanning is stopped the client's callback function is called with a DM_SCAN_STOP_IND event.

Syntax:

```
void DmScanStop(void)
```

5.2.5 DmScanSetInterval()

This function sets the scan interval and window. This function should only be called when scanning is stopped.

Syntax:

```
void DmScanSetInterval(uint8_t scanPhy, uint16_t scanInterval, uint16_t
        scanWindow)
```

Where:

- scanPhy: Scanning PHY.
- scanInterval: The scan interval. See 3.1.4.
- scanWindow: The scan window. See 3.1.4.

5.2.6 DmScanSetAddrType ()

Set the local address type used while scanning. This function can be used to configure scanning to use

a random or private address.

Syntax:

```
void DmScanSetAddrType (uint8_t addrType)
```

Where:

- `addrType`: Address type. See 3.1.4.

5.3 Callback interface

5.3.1 DM_SCAN_START_IND: Scanning started

Callback event for scanning started.

Table 13 Scanning started

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.

5.3.2 DM_SCAN_STOP_IND: Scanning stopped

Callback event for scanning stopped.

Table 14 Scanning stopped

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.

5.3.3 DM_SCAN_REPORT_IND: Scan report

Callback event for scan report. This event uses type `hciLeAdvReportEvt_t` defined in *ARM Cordio Stack API Reference Manual*.

Table 15 Scan report

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`uint8_t *`	`pData`	Pointer to received data.
`uint8_t`	`len`	Data length.
`int8_t`	`rssi`	RSSI of received packet.
`uint8_t`	`eventType`	Scan report event type. See 3.1.3.
`uint8_t`	`addrType`	Peer address type.
`bdAddr_t`	`addr`	Peer address.

6 Connection Management

The DM connection management interface is used to open, accept, configure, and close connections. It is also used to read connection-related information such as the RSSI, channel map, and remote device information.

6.1 Constants and data types

6.1.1 Client ID

The client ID parameter to function `DmConnRegister()` identifies the client to the DM connection manager. The possible values are shown below.

Table 16 Client ID

Name	Description
`DM_CLIENT_ID_ATT`	Identifier for attribute protocol. For internal use only.
`DM_CLIENT_ID_SMP`	Identifier for security manager protocol. For internal use only.
`DM_CLIENT_ID_DM`	Identifier for device manager. For internal use only.
`DM_CLIENT_ID_APP`	Identifier for the application.
`DM_CLIENT_ID_L2C`	Identifier for L2CAP.

6.1.2 dmConnId_t

This data type is used for the connection identifier. The connection identifier uniquely identifies the connection.

6.1.3 Connection busy/idle state

The connection busy/idle state indicates when the connection is busy with a stack protocol procedure, such as pairing or service discovery. The application can use this state to decide whether or not to perform certain connection operations such as a connection parameter update.

Table 17 Connection busy/idle state

Name	Description
`DM_CONN_IDLE`	Connection is idle.
`DM_CONN_BUSY`	Connection is busy.

6.1.4 Busy/Idle state bitmask

The connection busy/idle bitmask indicates which stack protocol procedure or application procedure is busy.

Table 18 Busy/idle state bitmask

Name	Description
DM_IDLE_SMP_PAIR	SMP pairing in progress.
DM_IDLE_DM_ENC	DM Encryption setup in progress.
DM_IDLE_ATTS_DISC	ATTS service discovery in progress.
DM_IDLE_APP_DISC	App framework service discovery in progress.
DM_IDLE_USER_1	For use by user application.
DM_IDLE_USER_2	For use by user application.
DM_IDLE_USER_3	For use by user application.
DM_IDLE_USER_4	For use by user application.

6.2 Functions

6.2.1 DmConnInit()

Initialize DM connection manager. This function is typically called once at system startup.

Syntax:

```
void DmConnInit(void)
```

6.2.2 DmConnMasterInit()

Initialize DM connection manager for operation as master. This function is typically called once at system startup.

Syntax:

```
void DmConnMasterInit(void)
```

6.2.3 DmExtConnMasterInit()

Initialize DM connection manager for operation as AE master. This function is typically called once at system startup.

Syntax:

```
void DmExtConnMasterInit(void)
```

6.2.4 DmConnSlaveInit()

Initialize DM connection manager for operation as slave. This function is typically called once at system startup.

Syntax:

```
void DmConnSlaveInit(void)
```

6.2.5 DmExtConnSlaveInit()

Initialize DM connection manager for operation as AE slave. This function is typically called once at system startup.

Syntax:

```
void DmExtConnSlaveInit(void)
```

6.2.6 DmConnRegister()

This function is called by a client to register with the DM connection manager. After registering the client can call other functions in the API to open, close, update or accept a connection. The client will also receive DM connection events via its callback for all connections, whether or not initiated by the client.

Syntax:

```
void DmConnRegister(uint8_t clientId, dmCback_t cback)
```

Where:

- `clientId`: The client identifier. See 6.1.1.
- `cback`: Client callback function. See 3.3.1.

6.2.7 DmConnOpen()

This function opens a connection to a peer device with the given address. This function can only be called when operating as a master.

Syntax:

```
dmConnId_t dmConnId_t DmConnOpen(uint8_t clientId, uint8_t initPhys, uint8_t
         addrType, uint8_t *pAddr)
```

Where:

- `clientId`: The client identifier. See 6.1.1.
- `initPhys`: PHYs initialized for use.
- `addrType`: Address type. See 3.1.4.
- `pAddr`: Peer device address.

This function returns a connection identifier. When the connection is opened the client's callback function is called with a `DM_CONN_OPEN_IND` event. If the connection fails for any reason the client's callback function is called with a `DM_CONN_CLOSE_IND` event.

6.2.8 DmConnClose()

This function closes the connection with the give connection identifier. This function can be called when operating as a master or slave.

Syntax:

```
void DmConnClose(uint8_t clientId, dmConnId_t connId, uint8_t reason)
```

Where:

- `clientId`: The client identifier. See 6.1.1.

- connId: Connection identifier. See 6.1.2.
- reason: Reason connection is being closed.

When the connection is closed the client's callback function is called with a DM_CONN_CLOSE_IND event.

6.2.9 DmConnAccept()

This function accepts a connection from the given peer device by initiating directed advertising. This function can only be called when operating as a slave.

Syntax:

```
dmConnId_t DmConnAccept(uint8_t clientId, uint8_t advHandle, uint8_t advType,
         uint16_t duration, uint8_t maxEaEvents, uint8_t addrType, uint8_t
         *pAddr)
```

Where:

- clientId: The client identifier. See 6.1.1.
- advHandle: Advertising handle.
- advType: Advertising type. See 3.1.3.
- duration: Advertising duration.
- maxEaEvents: Maximum number of extended advertising events.
- addrType: Address type. See 3.1.4.
- pAddr: Peer device address.

This function returns a connection identifier. When the connection is opened the client's callback function is called with a DM_CONN_OPEN_IND event. If the connection fails for any reason or if the connection is not opened within 1.28 seconds the client's callback function is called with a DM_CONN_CLOSE_IND event.

6.2.10 DmConnUpdate()

This function updates the connection parameters of an open connection. This function can be called when operating as a master or a slave.

Syntax:

```
void DmConnUpdate(dmConnId_t connId, hciConnSpec_t *pConnSpec)
```

Where:

- connId: Connection identifier. See 6.1.2.
- pConnSpec: Connection specification. See the *ARM Cordio Stack API Reference Manual*.

6.2.11 DmConnSetScanInterval()

This function sets the scan interval and window for created connections created with DmConnOpen(). This function must be called before calling DmConnOpen() for the parameters to be in effect.

Syntax:

```
void DmConnSetScanInterval(uint8_t initPhy, uint16_t scanInterval, uint16_t
         scanWindow)
```

Where:

- `initPhy`: The initiator PHY.
- `scanInterval`: The scan interval. See 3.1.4.
- `scanWindow`: The scan window. See 3.1.4.

6.2.12 DmConnSetConnSpec()

This function sets the connection specification parameters for connections created with `DmConnOpen()`. This function must be called before calling `DmConnOpen()` for the parameters to be in effect.

Syntax:

```
void DmConnSetConnSpec(hciConnSpec_t *pConnSpec)
```

Where:

- `pConnSpec`: Connection specification. See the *ARM Cordio Stack API Reference Manual.*

6.2.13 DmConnReadRssi()

This function reads RSSI of a given connection.

Syntax:

```
void DmConnReadRssi(dmConnId_t connId)
```

Where:

- `connId`: Connection identifier. See 6.1.2.

6.2.14 DmRemoteConnParamReqReply()

Reply to the HCI remote connection parameter request event. This command is used to indicate that the Host has accepted the remote device's request to change connection parameters.

Syntax:

```
void DmRemoteConnParamReqReply(dmConnId_t connId , hciConnSpec_t *pConnSpec)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `pConnSpec`: Connection specification. See the *ARM Cordio Stack API Reference Manual*.

6.2.15 DmRemoteConnParamReqNegReply()

Negative reply to the HCI remote connection parameter request event. This command is used to indicate that the Host has rejected the remote device's request to change connection parameters.

Syntax:

```
void DmRemoteConnParamReqNegReply(dmConnId_t connId , uint8_t reason)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `reason`: Reason for rejection.

6.2.16 DmConnSetDataLen()

This function sets the data length for a given connection.

Syntax:

```
void DmConnSetDataLen(dmConnId_t connId , uint16_t txOctets, uint16_t txTime)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `txOctets`: Maximum number of payload octets for a Data PDU.
- `txTime`: Maximum number of microseconds for a Data PDU.

6.2.17 DmWriteAuthPayloadTimeout()

This function sets authenticated payload timeout for a given connection.

Syntax:

```
void DmWriteAuthPayloadTimeout(dmConnId_t connId, uint16_t timeout)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `timeout`: Timeout period in units of 10ms.

6.2.18 DmConnSecLevel()

Return the security level of the connection.

Syntax:

```
uint8_t DmConnSecLevel(dmConnId_t connId)
```

Where:

- `connId`: Connection identifier. See 6.1.2.

6.2.19 DmConnSetAddrType ()

Set the local address type used for connections created with `DmConnOpen()`. This function can be used to create connections using a random or private address.

Syntax:

```
void DmConnSetAddrType (uint8_t addrType)
```

Where:

- `addrType`: Address type. See 3.1.4.

6.2.20 DmConnSetIdle()

Configure a bit in the connection idle state mask as busy or idle.

Syntax:

```
void DmConnSetIdle(dmConnId_t connId, uint16_t idleMask, uint8_t idle)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `idleMask`: Bit in the idle state mask to configure. See 6.1.4.
- `idle`: `DM_CONN_BUSY` or `DM_CONN_IDLE`. See 6.1.3.

6.2.21 DmConnCheckIdle()

Check if a connection is idle.

Syntax:

```
uint16_t DmConnCheckIdle(dmConnId_t connId)
```

Where:

- `connId`: Connection identifier. See 6.1.2.

This function returns zero if the connection is idle or nonzero if busy.

6.3 Callback interface

6.3.1 DM_CONN_OPEN_IND: Connection opened

Callback event for connection opened. This event uses type hciLeConnCmplEvt_t defined in *ARM Cordio Stack API Reference Manual*.

Table 19 Connection opened

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`wsfMsgHdr_t`	`hdr.param`	Connection identifier.
`uint8_t`	`Status`	Connection status
`uint16_t`	`handle`	Connection handle.
`uint8_t`	`role`	Connection role.
`uint8_t`	`addrType`	Address type.
`bdAddr_t`	`peerAddr`	Peer address.
`uint16_t`	`connInterval`	Connection interval.
`uint16_t`	`connLatency`	Connection latency.
`uint16_t`	`supTimeout`	Connection supervision timeout.
`uint8_t`	`clockAccuracy`	Peer clock accuracy.

6.3.2 DM_CONN_CLOSE_IND: Connection closed

Callback event for connection closed. This event uses type `hciDisconnectCmplEvt_t` defined in the *ARM Cordio Stack API Reference Manual*.

Table 20 Connection closed

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`wsfMsgHdr_t`	`hdr.param`	Connection identifier.
`uint8_t`	`status`	Connection status
`uint16_t`	`handle`	Connection handle.
`uint8_t`	`reason`	Disconnect reason.

6.3.3 DM_CONN_UPDATE_IND: Connection update

Callback event for connection update complete. This event uses type `hciLeConnUpdateCmplEvt_t` defined in the *ARM Cordio Stack API Reference Manual*.

Table 21 Connection update

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`wsfMsgHdr_t`	`hdr.param`	Connection identifier.
`uint8_t`	`status`	Status of connection update procedure.
`uint16_t`	`handle`	Connection handle.
`uint16_t`	`connInterval`	Connection interval.
`uint16_t`	`connLatency`	Connection latency.
`uint16_t`	`supTimeout`	Supervision timeout.

6.3.4 DM_CONN_READ_RSSI_IND: connection RSSI read

Callback event for reading connection RSSI. This event uses type `hciReadRssiCmdCmplEvt_t` defined in the *ARM Cordio Stack API Reference Manual*.

Table 22 RSSI read

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.
`wsfMsgHdr_t`	`hdr.param`	Connection identifier.
`uint8_t`	`status`	Status of procedure.

uint16_t	handle	Connection handle.
uint8_t	rssi	RSSI.

6.3.5 DM_REM_CONN_PARAM_REQ_IND: Remote connection parameter requested

Callback event for remote connection parameter requested. This event uses type hciLeRemConnParamReqEvt_t defined in the *ARM Cordio Stack API Reference Manual*.

Table 23 Remote connection parameter requested

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint16_t	handle	Connection handle.
uint16_t	intervalMin	Minimum value of the connection interval requested by the remote device.
uint16_t	intervalMax	Maximum value of the connection interval requested by the remote device.
uint16_t	latency	Maximum allowed slave latency for the connection specified as the number of connection events requested by the remote device.
uint16_t	timeout	Supervision timeout for the connection requested by the remote device.

6.3.6 DM_CONN_DATA_LEN_CHANGE_IND: Connection data length changed

Callback event for data length changed. This event uses type hciLeDataLenChangeEvt_t defined in the *ARM Cordio Stack API Reference Manual.*

Table 24 Connection data length changed

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint16_t	handle	Connection handle.
uint16_t	maxTxOctets	The maximum number of payload octets in a Link Layer Data Channel PDU that the local Controller will send on this connection.
uint16_t	maxTxTime	The maximum time that the local Controller will take to send a Link Layer Data Channel PDU on this

		connection.
uint16_t	maxRxOctets	The maximum number of payload octets in a Link Layer Data Channel PDU that the local controller expects to receive on this connection.
uint16_t	maxRxTime	The maximum time that the local Controller expects to take to receive a Link Layer Data Channel PDU on this connection.

6.3.7 DM_CONN_WRITE_AUTH_TO_IND: Write authenticated payload timeout complete

Callback event for write authenticated payload timeout complete. This event uses type hciWriteAuthPayloadToCmdCmplEvt_t defined in the *ARM Cordio Stack API Reference Manual*.

Table 25 Write authenticated payload timeout complete

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	hdr.status	Status of procedure.
uint8_t	status	Status of procedure.
uint16_t	handle	Connection handle.

6.3.8 DM_CONN_AUTH_TO_EXPIRED_IND: Authenticated payload timeout expired

Callback event for authenticated payload timeout expired. This event uses type hciAuthPayloadToExpiredEvt_t defined in the *ARM Cordio Stack API Reference Manual*.

Table 26 Authenticated payload timeout expired

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	hdr.status	Status of procedure.
uint16_t	handle	Connection handle.

7 Local Device Management

The DM local device management interface is used for initialization and reset, setting local parameters, sending vendor-specific commands, and LE GAP attribute management.

7.1 Functions

7.1.1 DmDevReset()

This function initiates the HCI reset sequence. When the reset sequence is complete the client's callback function is called with a `DM_RESET_CMPL_IND` event.

Syntax:

```
void DmDevReset(void)
```

7.1.2 DmDevRole()

This function returns the device role indicating master or slave. See 3.1.1.

Syntax:

```
uint8_t DmDevRole(void)
```

7.1.3 DmDevSetRandAddr()

Set the random address to be used by the local device.

Syntax:

```
void DmDevSetRandAddr(uint8_t *pAddr)
```

Where:

- `pAddr`: Random address.

7.1.4 DmDevWhiteListAdd()

Add a peer device to the white list. Note that this function cannot be called while advertising, scanning, or connecting with white list filtering active.

Syntax:

```
void DmDevWhiteListAdd(uint8_t addrType, uint8_t *pAddr)
```

Where:

- `addrType`: Address type. See 3.1.4.
- `pAddr`: Peer device address.

7.1.5 DmDevWhiteListRemove()

Remove a peer device from the white list. Note that this function cannot be called while advertising, scanning, or connecting with white list filtering active.

Syntax:

```
void DmDevWhiteListRemove(uint8_t addrType, uint8_t *pAddr)
```

Where:

- `addrType`: Address type. See 3.1.4.
- `pAddr`: Peer device address.

7.1.6 DmDevWhiteListClear()

Clear the white list. Note that this function cannot be called while advertising, scanning, or connecting with white list filtering active.

Syntax:

```
void DmDevWhiteListClear(void)
```

7.2 Callback interface

7.2.1 DM_RESET_CMPL_IND: Reset complete

Callback event for reset complete.

Table 27 Reset complete

Type	Name	Description
`wsfMsgHdr_t`	`hdr.event`	Callback event.

8 Security Management

The DM security management interface is used for pairing, authentication, and encryption.

8.1 Constants and data types

8.1.1 Authentication flags

This parameter contains the authentication flags of a procedure or its associated data.

Table 28 Authentication flags

Name	Value	Description
DM_AUTH_BOND_FLAG	0x01	Bonding requested.
DM_AUTH_MITM_FLAG	0x04	MITM (authenticated pairing) requested.
DM_AUTH_SC_FLAG	0x08	LE Secure Connections requested
DM_AUTH_KP_FLAG	0x10	Keypress notification requested

8.1.2 Key distribution

This parameter contains a bit mask of the keys distributed during the pairing procedure.

Table 29 Key distribution

Name	Value	Description
DM_KEY_DIST_LTK	0x01	Distribute LTK used for encryption.
DM_KEY_DIST_IRK	0x02	Distribute IRK used for privacy.
DM_KEY_DIST_CSRK	0x04	Distribute CSRK used for signed data.

8.1.3 Key type

This parameter indicates the key type used in DM_SEC_KEY_IND.

Table 30 Key type

Name	Description
DM_KEY_LOCAL_LTK	LTK generated locally for this device.
DM_KEY_PEER_LTK	LTK received from peer device.
DM_KEY_IRK	IRK and identity info of peer device.
DM_KEY_CSRK	CSRK of peer device.

8.1.4 Security level

This parameter indicates the security level of a connection.

Table 31 Security level

Name	Description
DM_SEC_LEVEL_NONE	Connection has no security.
DM_SEC_LEVEL_ENC	Connection is encrypted with unauthenticated key.
DM_SEC_LEVEL_ENC_AUTH	Connection is encrypted with authenticated key.
DM_SEC_LEVEL_ENC_LESC	Connection is encrypted with LE Secure Connections.

8.1.5 Security error codes

These error codes can be used in the status parameter of security functions and callback event structures.

Table 32 Security error codes

Name	Value	Description
SMP_ERR_PASSKEY_ENTRY	0x01	User input of passkey failed.
SMP_ERR_OOB	0x02	OOB data is not available.
SMP_ERR_AUTH_REQ	0x03	Authentication requirements cannot be met.
SMP_ERR_CONFIRM_VALUE	0x04	Confirm value does not match.
SMP_ERR_PAIRING_NOT_SUP	0x05	Pairing is not supported by the device.
SMP_ERR_ENC_KEY_SIZE	0x06	Insufficient encryption key size.
SMP_ERR_COMMAND_NOT_SUP	0x07	Command not supported.
SMP_ERR_UNSPECIFIED	0x08	Unspecified reason.
SMP_ERR_ATTEMPTS	0x09	Repeated attempts.
SMP_ERR_INVALID_PARAM	0x0A	Invalid parameter or command length.
SMP_ERR_DH_KEY_CHECK	0x0B	DH Key check did not match
SMP_ERR_NUMERIC_COMPARISON	0x0C	Numeric comparison did not match
SMP_ERR_BR_EDR_IN_PROGRESS	0x0D	BR/EDR in progress
SMP_ERR_CROSS_TRANSPORT	0x0E	BR/EDR Cross transport key generation not allowed
SMP_ERR_MEMORY	0xE0	Out of memory.

SMP_ERR_TIMEOUT	0xE1	Transaction timeout.

8.1.6 Keypress types

These values are used in to notify the peer of a keypress event types.

Table 33 Keypress types

Name	Value	Description
SMP_PASSKEY_ENTRY_STARTED	0x00	Passkey entry started keypress type.
SMP_PASSKEY_DIGIT_ENTERED	0x01	Passkey digit entered keypress type
SMP_PASSKEY_DIGIT_ERASED	0x02	Passkey digit erased keypress type
SMP_PASSKEY_CLEARED	0x03	Passkey cleared keypress type
SMP_PASSKEY_ENTRY_COMPLETED	0x04	Passkey entry complete keypress type

8.1.7 dmSecLtk_t

This data structure is the LTK data type.

Table 34 LTK data type

Type	Name	Description
uint8_t	key[SMP_KEY_LEN]	Key.
uint8_t *	rand[SMP_RAND8_LEN]	Random identifier for key.
uint16_t	ediv	Diversifier for key.

8.1.8 dmSecIrk_t

This data structure is the IRK data type.

Table 35 IRK data type

Type	Name	Description
uint8_t	key[SMP_KEY_LEN]	Key.
bdAddr_t	bdAddr	Peer device address.
uint8_t	addrType	Peer device address type.

8.1.9 dmSecCsrk_t

This data structure is the CSRK data type.

Table 36 CSRK data type

Type	Name	Description
uint8_t	key[SMP_KEY_LEN]	Key.

8.1.10 dmSecKey_t

This data structure is a union of key types.

Table 37 dmSecKey_t type

Type	Name	Description
dmSecLtk_t	ltk	LTK.
dmSecIrk_t	irk	IRK.
dmSecCsrk_t	csrk	CSRK.

8.2 Function interface

8.2.1 DmSecInit()

Initialize DM security manager. This function is typically called once at system startup.

Syntax:

```
void DmSecInit(void)
```

8.2.2 DmSecPairReq()

This function is called by a master device to initiate pairing.

Syntax:

```
void DmSecPairReq(dmConnId_t connId, bool_t oob, uint8_t auth, uint8_t iKeyDist,
        uint8_t rKeyDist)
```

Where:

- connId: Connection identifier. See 6.1.2.
- oob: Out-of-band pairing data present or not present.
- auth: Authentication and bonding flags. See 8.1.1.
- iKeyDist: Initiator key distribution flags. See 8.1.2.
- rKeyDist: Responder key distribution flags. See 8.1.2.

When the pairing procedure is complete the client's callback function is called with a DM_SEC_PAIR_CMPL_IND event if successful or a DM_SEC_PAIR_FAIL_IND if failure.

8.2.3 DmSecPairRsp()

This function is called by a slave device to proceed with pairing after a DM_SEC_PAIR_IND event is received. This function must be called within 30 seconds of receiving the event otherwise the procedure will time out.

Syntax:

```
void DmSecPairRsp(dmConnId_t connId, bool_t oob, uint8_t auth, uint8_t iKeyDist,
        uint8_t rKeyDist)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `oob`: Out-of-band pairing data present or not present.
- `auth`: Authentication and bonding flags. See 8.1.1.
- `iKeyDist`: Initiator key distribution flags. See 8.1.2.
- `rKeyDist`: Responder key distribution flags. See 8.1.2.

When the pairing procedure is complete the client's callback function is called with a `DM_SEC_PAIR_CMPL_IND` event if successful or a `DM_SEC_PAIR_FAIL_IND` if failure.

8.2.4 DmSecCancelReq()

This function is called to cancel the pairing process.

Syntax:

```
void DmSecCancelReq(dmConnId_t connId, uint8_t reason)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `reason`: Failure reason. See 8.1.5.

8.2.5 DmSecAuthRsp()

This function is called in response to a `DM_SEC_AUTH_REQ_IND` event to provide PIN or OOB data during pairing.

Syntax:

```
void DmSecAuthRsp(dmConnId_t connId, uint8_t authDataLen, uint8_t *pAuthData)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `authDataLen`: Length of PIN or OOB data. Set to 3 if PIN is used or 16 if OOB data is used.
- `pAuthData`: Pointer to PIN or OOB data. If PIN is used, this points to a byte array containing a 24-bit integer in little endian format.

8.2.6 DmSecSlaveReq()

This function is called by a slave device to request that the master initiates pairing or link encryption.

Syntax:

```
void DmSecSlaveReq(dmConnId_t connId, uint8_t auth)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `auth`: Authentication and bonding flags. See 8.1.1.

8.2.7 DmSecEncryptReq()

This function is called by a master device to initiate link encryption.

Syntax:

```
void DmSecEncryptReq(dmConnId_t connId, uint8_t secLevel, dmSecLtk_t *pLtk)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `secLevel`: Security level of pairing when LTK was exchanged. See 8.1.4.
- `pLtk`: Pointer to LTK parameter structure.

When the encryption procedure is complete the client's callback function is called with a `DM_ENCRYPT_IND` event if successful or a `DM_ENCRYPT_FAIL_IND` if failure.

8.2.8 DmSecLtkRsp()

This function is called by a slave in response to a `DM_SEC_LTK_REQ_IND` event to provide the long term key used for encryption.

Syntax:

```
void DmSecLtkRsp(dmConnId_t connId, bool_t keyFound, uint8_t secLevel, uint8_t
        *pKey)
```

Where:

- `connId`: Connection identifier. See 6.1.2.
- `keyFound`: TRUE if key found.
- `secLevel`: Security level of pairing when LTK was exchanged. See 8.1.4.
- `pKey`: Pointer to the key, if found.

8.2.9 DmSecSetLocalCsrk()

This function sets the local CSRK used by the device.

Syntax:

```
void DmSecSetLocalCsrk(uint8_t *pCsrk)
```

Where:

- `pCsrk`: Pointer to CSRK.

8.2.10 DmSecSetLocalIrk()

This function sets the local IRK used by the device.

Syntax:

```
void DmSecSetLocalIrk(uint8_t *pIrk)
```

Where:

- `pIrk`: Pointer to IRK.

8.2.11 DmSecLescInit()

This function is called to initialize the LE Secure Connections subsystem.

Syntax:

```
void DmSecLescInit(void)
```

8.2.12 DmSecKeypressReq()

This function can be used to send a keypress request command to the peer device during LE Secure Connections Passkey Security.

Syntax:

```
void DmSecKeypressReq(dmConnId_t connId, uint8_t keypressType)
```

Where:

- `ConnId`: Connection identifier. See 6.1.2.
- `KeypressType`: Type of keypress reported to peer. See 8.1.6.

8.2.13 DmSecGenerateEccKeyReq()

This function is called to generate an ECC Key for use in LE Secure Connections. The application is notified of the result of the generate ECC key operation via the `DM_SEC_ECC_KEY_IND` event.

Syntax:

```
void DmSecGenerateEccKeyReq(void)
```

8.2.14 DmSecSetEccKey()

This function is called to set the ECC key used in LE Secure Connections.

Syntax:

```
void DmSecSetEccKey(wsfSecEccKey_t *pKey)
```

Where:

- `pKey`: Pointer to the ECC key.

8.2.15 DmSecSetDebugEccKey()

This function is called to set the ECC key used in LE Secure Connections.

Syntax:

```
void DmSecSetDebugEccKey(void)
```

8.2.16 DmSecSetOob()

This function is called to set the Out of Band configuration containing the local and remote confirm and random values for LE Secure Connections Security.

Syntax:

```
void DmSecSetOob(dmConnId_t connId, dmSecLescOobCfg_t *pConfig)
```

Where:

- ConnId: Connection identifier. See 6.1.2.
- pConfig: The OOB configuration.

8.2.17 DmSecCalcOobReq()

This function is used to calculate the local confirm value used in Out of Band LE Secure Connections Security.

Syntax:

```
void DmSecCalcOobReq(uint8_t *pRand, uint8_t *pPubKeyX)
```

Where:

- pRand: A 128-bit random value.
- pPubKeyX: The X component of the ECC public key.

8.2.18 DmSecCompareRsp()

This function is used to indicate the LE Secure Connections Numeric Comparison value is valid or invalid. It is typically called in response to a DM_SEC_COMPARE_IND event.

Syntax:

```
void DmSecCompareRsp(dmConnId_t connId, bool_t valid)
```

Where:

- ConnId: Connection identifier. See 6.1.2.
- valid: TRUE if the compare value is correct, else FALSE.

8.3 Callback interface

8.3.1 DM_SEC_PAIR_CMPL_IND: Pairing complete

Callback event for pairing complete. This event uses type dmSecPairCmplIndEvt_t.

Table 38 Pairing complete

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	auth	Authentication and bonding flags. See 8.1.1.

8.3.2 DM_SEC_PAIR_FAIL_IND: Pairing failed

Callback event for pairing failed. This event uses type wsfMsgHdr_t.

Table 39 Pairing failed

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
wsfMsgHdr_t	hdr.status	Pairing failure status. See 8.1.5.

8.3.3 DM_SEC_ENCRYPT_IND: Connection encrypted

Callback event for connection encrypted. This event uses type dmSecEncryptIndEvt_t.

Table 40 Connection encrypted

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
bool_t	usingLtk	TRUE if connection encrypted with LTK.

8.3.4 DM_SEC_ENCRYPT_FAIL_IND: Encryption failed

Callback event for encryption failed. This event uses type wsfMsgHdr_t.

Table 41 Encryption failed

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
wsfMsgHdr_t	hdr.status	Encryption failure status. See 8.1.5.

8.3.5 DM_SEC_AUTH_REQ_IND: Authentication requested

Callback event for PIN or OOB data requested for pairing. This event uses type dmSecAuthReqIndEvt_t.

Table 42 Authentication requested

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
bool_t	oob	Out-of-band data requested.

bool_t	display	TRUE if PIN is to be displayed.

If OOB is TRUE, the client should call DmSecAuthRsp() with OOB data, if available. If display is TRUE, the client will typically generate and display a random PIN and call DmSecAuthRsp() with this PIN. If display is FALSE, the client will typically prompt the user to enter a PIN and call DmSecAuthRsp() with this PIN.

8.3.6 DM_SEC_KEY_IND: Key data

Callback event for key data indication. This event uses data type dmSecKeyIndEvt_t.

Table 43 Key data

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
dmSecKey_t	keyData	Key data.
uint8_t	type	Key type. See 8.1.3.
uint8_t	secLevel	Security level of pairing when key was exchanged. See 8.1.4.
uint8_t	encKeyLen	Length of encryption key used when data was transferred.

8.3.7 DM_SEC_LTK_REQ_IND: LTK requested

Callback event for LTK requested. This event uses data type hciLeLtkReqEvt_t.

Table 44 LTK requested

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint16_t	handle	Connection handle
uint8_t	randNum[HCI_RAND_LEN]	Random number associated with key
uint16_t	encDiversifier	Encryption diversifier associated with key

8.3.8 DM_SEC_PAIR_IND: Incoming pairing request

Callback event for incoming pairing request. This event uses type dmSecPairIndEvt_t.

Table 45 Incoming pairing requested

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	auth	Authentication and bonding flags. See 8.1.1.
bool_t	oob	Out-of-band pairing data present or not present.
uint8_t	iKeyDist	Initiator key distribution flags. See 8.1.2.
uint8_t	rKeyDist	Responder key distribution flags. See 8.1.2.

8.3.9 DM_SEC_SLAVE_REQ_IND: Incoming slave security request

Callback event for incoming slave security request. This event uses type dmSecPairIndEvt_t.

Table 46 Incoming slave security request

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	auth	Authentication and bonding flags. See 8.1.1.

8.3.10 DM_SEC_CALC_OOB_IND: Out of band confirm

Callback with the result of an Out Of Band confirm calculation. This event uses type dmSecOobCalcIndEvt_t.

Table 47 Out of band confirm

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Encryption failure status. See 8.1.5.
uint8_t	confirm[SMP_CONFIRM_LEN]	Local confirm value.
uint8_t	random[SMP_RANDOM_LEN]	Local random value.

8.3.11 DM_SEC_ECC_KEY_IND: ECC key generation

Callback with the result of an ECC Key generation. This event uses type wsfSecEccMsg_t.

Table 48 ECC key generation

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
wsfMsgHdr_t	hdr.status	Key generation status.
uint8_t	pubKey_x [WSF_ECC_KEY_LEN]	X component of the public key.
uint8_t	pubKey_y [WSF_ECC_KEY_LEN]	Y component of the public key.
uint8_t	privKey[WSF_ECC_KEY_LEN]	Private key.

8.3.12 DM_SEC_COMPARE_IND: Confirm comparison pairing

Callback with the confirm value during Numeric Comparison LE Secure Connections pairing. This event uses type dmSecCnfIndEvt_t.

Table 49 Numeric comparison pairing

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	confirm[SMP_CONFIRM_LEN]	Local confirm value.

8.3.13 DM_SEC_KEYPRESS_IND: Keypress from peer

Callback when peer receives a keypress command from the peer during LE Secure Connections passkey pairing. This event uses type dmSecKeypressIndEvt_t.

Table 50 Keypress indication

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.param	Connection identifier.
uint8_t	notificationType	Type of keypress.

9 Privacy

The DM Privacy interface is used by a master or slave device for private address resolution.

9.1 Function interface

9.1.1 DmPrivInit()

Initialize DM privacy module. This function is typically called once at system startup.

Syntax:

```
void DmPrivInit(void)
```

9.1.2 DmPrivResolveAddr()

Resolve a private resolvable address. When complete the client's callback function is called with a `DM_PRIV_RESOLVED_ADDR_IND` event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivResolveAddr(uint8_t *pAddr, uint8_t *pIrk, uint16_t param)
```

Where:

- `pAddr`: Peer device address.
- `pIrk`: The peer's identity resolving key.
- `Param`: Client-defined parameter returned with callback event.

9.1.3 DmPrivAddDevToResList()

Add device to resolving list. When complete the client's callback function is called with a `DM_PRIV_ADD_DEV_TO_RES_LIST_IND` event. The client must wait to receive this event before executing this function again.

If the local or peer IRK associated with the peer Identity Address is all zeros then the LL will use or accept the local or peer Identity Address.

Note: `enableLlPriv` should be set to TRUE when the last device is being added to resolving list.

Syntax:

```
void DmPrivAddDevToResList(uint8_t addrType, const uint8_t *pIdentityAddr,
         uint8_t *pPeerIrk, uint8_t *pLocalIrk, bool_t enableLlPriv, uint16_t
         param)
```

Where:

- `addrType`: Peer identity address type.
- `pIdentityAddr`: Peer identity address.
- `pPeerIrk`: The peer's identity resolving key.
- `pLocalIrk`: The local identity resolving key.
- `enableLlPriv`: Set to TRUE to enable address resolution in LL.
- `param`: Client-defined parameter returned with callback event.

9.1.4 DmPrivRemDevFromResList()

Remove device from resolving list. When complete the client's callback function is called with a DM_PRIV_REM_DEV_FROM_RES_LIST_IND event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivRemDevFromResList(uint8_t addrType, const uint8_t *pIdentityAddr,
        uint16_t param)
```

Where:

- addrTypePeer: Identity address type.
- pIdentityAddr: Peer identity address.
- param: Client-defined parameter returned with callback event.

9.1.5 DmPrivClearResList()

Clear resolving list. When complete the client's callback function is called with DM_PRIV_CLEAR_RES_LIST_IND event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivClearResList(void)
```

9.1.6 DmPrivReadPeerResolvableAddr()

Read peer resolvable address. When complete the client's callback function is called with a DM_PRIV_READ_PEER_RES_ADDR_IND event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivReadPeerResolvableAddr (uint8_t addrType, const uint8_t
        *pIdentityAddr)
```

Where:

- addrTypePeer: Identity address type.
- pIdentityAddr: Peer identity address.

9.1.7 DmPrivReadLocalResolvableAddr ()

Read local resolvable address. When complete the client's callback function is called with a DM_PRIV_READ_LOCAL_RES_ADDR_IND event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivReadLocalResolvableAddr (uint8_t addrType, const uint8_t
        *pIdentityAddr)
```

Where:

- addrTypePeer: Identity address type.
- pIdentityAddr: Peer identity address.

9.1.8 DmPrivSetAddrResEnable()

Enable or disable address resolution in LL. When complete the client's callback function is called with a DM_PRIV_SET_ADDR_RES_ENABLE_IND event. The client must wait to receive this event before executing this function again.

Syntax:

```
void DmPrivSetAddrResEnable(bool_t enable)
```

Where:

- enable: Set to TRUE to enable address resolution or FALSE to disable it.

9.1.9 DmPrivSetResolvablePrivateAddrTimeout ()

Set resolvable private address timeout.

Syntax:

```
void DmPrivSetResolvablePrivateAddrTimeout (uint16_t rpaTimeout)
```

Where:

- rpaTimeout: Timeout measured in seconds.

9.2 Callback interface

9.2.1 DM_PRIV_RESOLVED_ADDR_IND: Private address resolved

Callback event for private address resolved. This event uses type wsfMsgHdr_t. If address resolution is successful hdr.status is set to HCI_SUCCESS, otherwise it is set to HCI_ERR_AUTH_FAILURE.

Table 51 Private address resolved

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.
wsfMsgHdr_t	hdr.param	Client-defined parameter passed to DmPrivResolveAddr().

9.2.2 DM_PRIV_ADD_DEV_TO_RES_LIST_IND: Device added to resolving list

Callback event for adding a device to the resolving list. This event uses type hciLeAddDevToResListCmdCmplEvt_t.

Table 52 Device added to resolving list

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.

wsfMsgHdr_t	hdr.param	Client-defined parameter passed to DmPrivAddDevToResList().
uint8_t	status	Command status

9.2.3 DM_PRIV_REM_DEV_FROM_RES_LIST_IND: Device removed from resolving list

Callback event for removing a device from the resolving list. This event uses type hciLeRemDevFromResListCmdCmplEvt_t.

Table 53 Device removed from resolving list

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.
wsfMsgHdr_t	hdr.param	Client-defined parameter passed to DmPrivRemDevToResList().
uint8_t	status	Command status

9.2.4 DM_PRIV_CLEAR_RES_LIST_IND: Resolving list cleared

Callback event for clearing the resolving list. This event uses type hciLeClearResListCmdCmplEvt_t.

Table 54 Resolving list cleared

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.

9.2.5 DM_PRIV_READ_PEER_RES_ADDR_IND: Peer resolving address read

Callback event for returning the peer resolving address. This event uses type hciLeReadPeerResAddrCmdCmplEvt_t.

Table 55 Read peer resolving address

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.
uint8_t	peerRpa[BDA_ADDR_LEN]	Peer resolving address.

9.2.6 DM_PRIV_READ_LOCAL_RES_ADDR_IND: Local resolving address read

Callback event for returning the local resolving address. This event uses type

hciLeReadLocalResAddrCmdCmplEvt_t.

Table 56 Read local resolving address

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.
uint8_t	localRpa[BDA_ADDR_LEN]	Local resolving address.

9.2.7 DM_PRIV_SET_ADDR_RES_ENABLE_IND: Address resolving enable set

Callback event for enabling address resolution. This event uses type hciLeSetAddrResEnableCmdCmplEvt_t.

Table 57 Set address event

Type	Name	Description
wsfMsgHdr_t	hdr.event	Callback event.
wsfMsgHdr_t	hdr.status	Status.

10 Scenarios

10.1 Advertising and scanning

Figure 1 shows a master device performing a scan and a slave device advertising. The slave application first configures the advertising parameters by calling `DmAdvSetInterval()` to set the advertising interval and then `DmAdvSetData()` twice to set the advertising data and the scan response data. Then it calls `DmAdvStart()` to start advertising.

The master application configures the scan interval and then calls `DmScanStart()` to begin scanning. When advertisements are received the stack sends `DM_SCAN_REPORT_IND` events to the application. The master application stops scanning by calling `DmScanStop()`. The slave application stops advertising by calling `DmAdvStop()`.

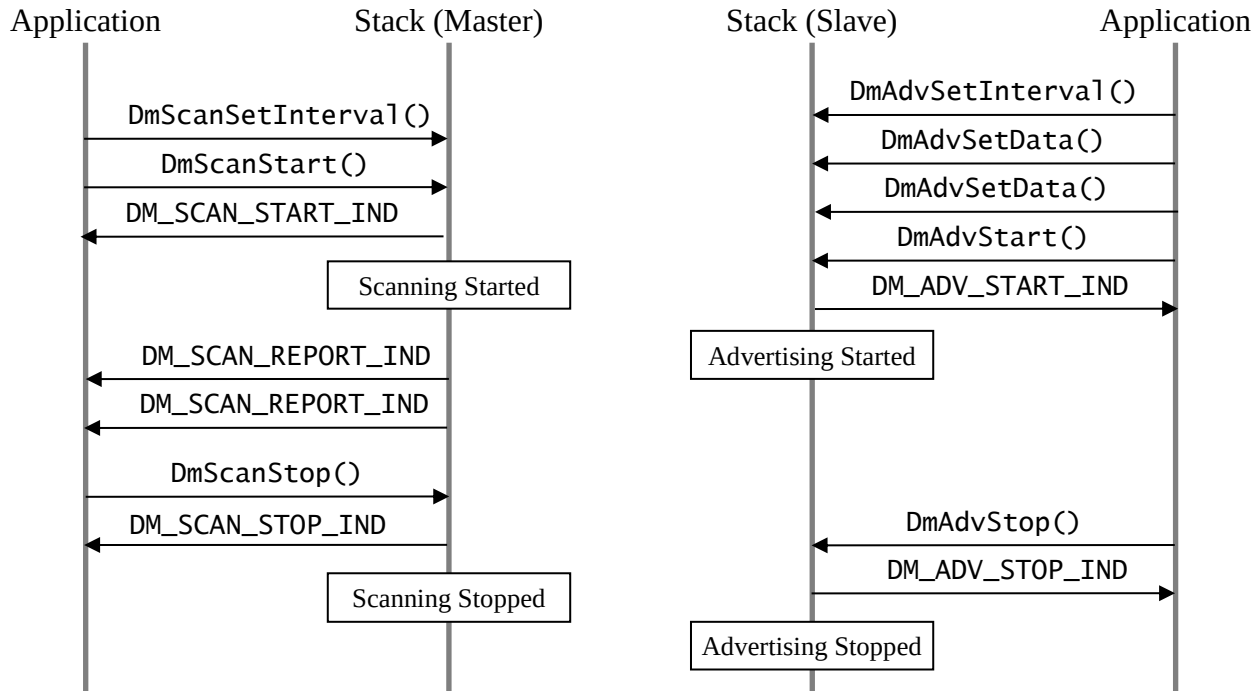

Figure 1. Advertising and scanning.

10.2 Connection open and close

Figure 2 shows connection procedures between two devices. The scenario starts with the slave device advertising and the master device already having the address of the slave. The master application calls `DmConnOpen()` to initiate a connection. A connection is established and a `DM_CONN_OPEN_IND` is sent to the application from the stack on each device.

Next, the master performs a connection update by calling `DmConnUpdate()`. When the connection update is complete a `DM_CONN_UPDATE_IND` is sent to the application from the stack on each device.

Next, the slave closes the connection by calling `DmConnClose()`. A DM_CONN_CLOSE_IND event is sent from the stack on each device when the connection is closed.

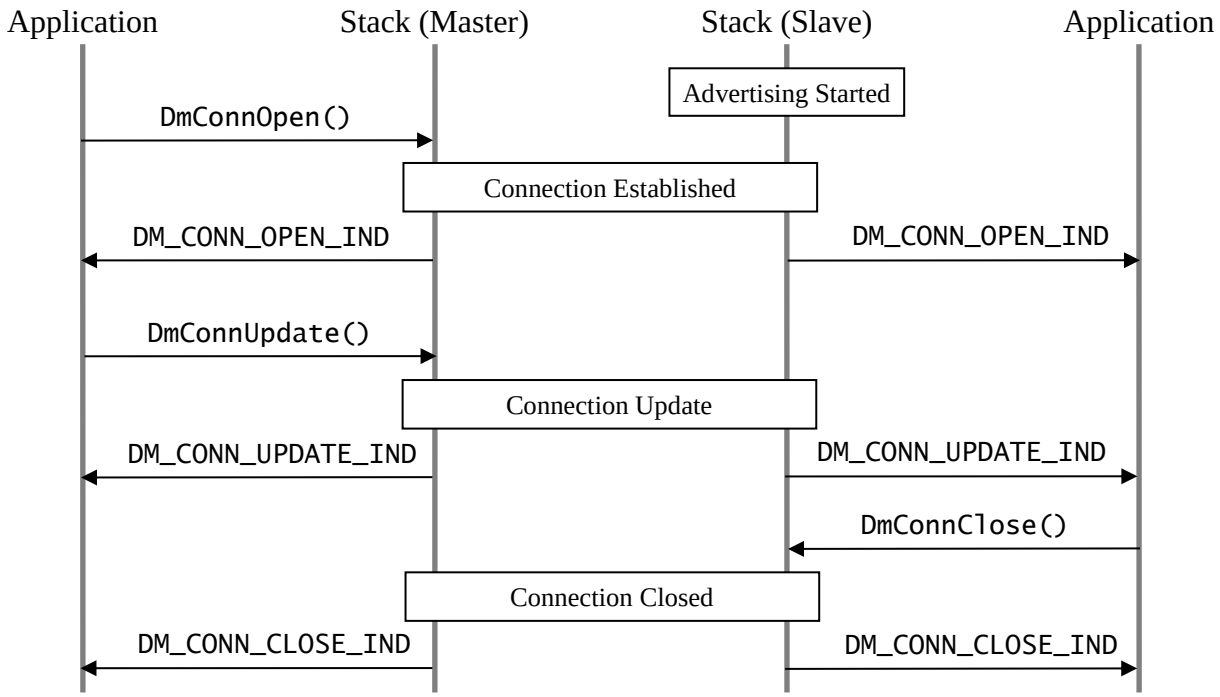

Figure 2. Connection open and close.

10.3 Pairing

Figure 3 shows a pairing procedure between two devices. A connection is established between the two devices and the master application initiates pairing by calling `DmSecPairReq()`. The slave application receives a `DM_SEC_PAIR_IND` and calls `DmSecPairRsp()` to proceed with pairing. In this example a PIN is used and a `DM_SEC_AUTH_REQ_IND` is sent to the application on each device to request a PIN. Each application responds with the PIN by calling `DmSecAuthRsp()`.

In the next phase of pairing the connection is encrypted and a `DM_SEC_ENCRYPT_IND` event is sent to the application on each device. Then key exchange begins. According to the Bluetooth specification, the slave device always distributes keys first. In this example, the slave distributes two keys and the master device distributes one. The slave sends its key data to the master. Note that when the slave sends its LTK, the slave application receives a `DM_SEC_KEY_IND` containing its own LTK. Then the master sends its key data to the slave. When the key exchange is completed successfully, a `DM_SEC_PAIR_CMPL_IND` event is sent to the application on each device.

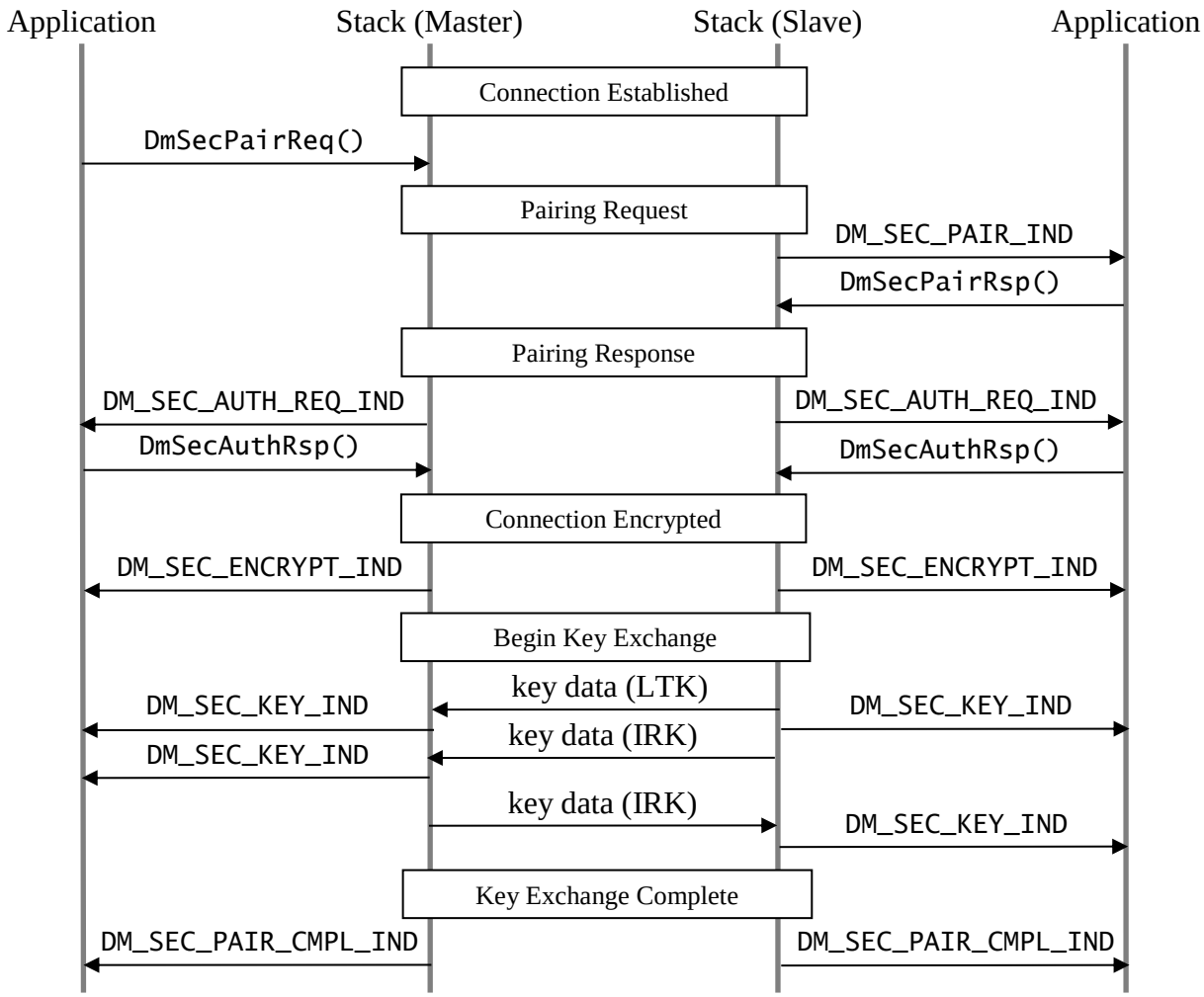

Figure 3. Pairing

10.4 Encryption

Figure 4 shows an encryption procedure. In this example the slave device requests security by calling `DmSecSlaveReq()` to sends a slave security request message to the master. The stack on the master sends a `DM_SEC_SLAVE_REQ_IND` to the application. Upon receiving the event the master application determines that this is a bonded device and its LTK is available, so it calls `DmSecEncryptReq()` to enable encryption.

After the encryption procedure is initiated the slave application receives a `DM_SEC_LTK_REQ_IND`, requesting the LTK used with this master device. The application finds the key and calls `DmSecLtkRsp()`. The encryption procedure completes and a `DM_SEC_ENCRYPT_IND` event is sent to the application on each device.

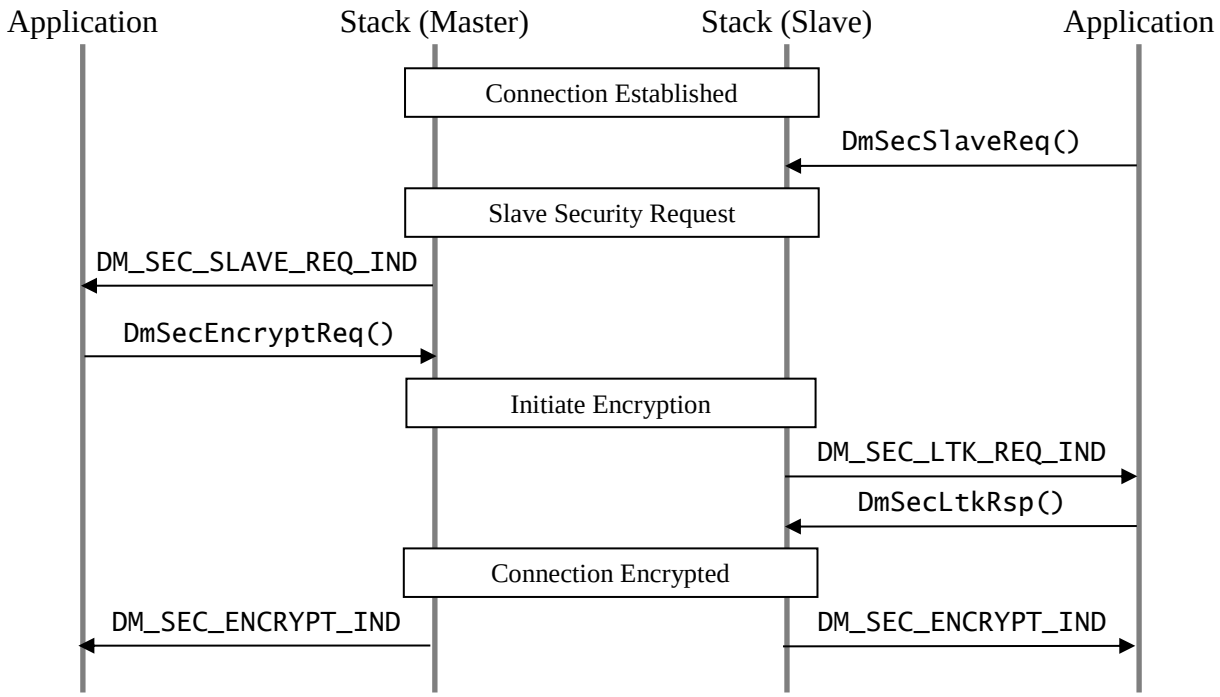

Figure 4. Encryption

10.5 Privacy

Figure 5 shows a master device performing a scan and a slave device advertising with a private resolvable address. Before a master device can resolve a slave's address the devices must have paired and the master must have received the slave's IRK during pairing.

The slave application first enables use of a private resolvable address by calling `DmAdvPrivStart()`. If this is the first time since device reset that `DmAdvPrivStart()` has been called, the application must wait for a `DM_ADV_NEW_ADDR_IND` before it starts advertising. Then it calls `DmAdvStart()` to start advertising.

The master application calls `DmScanStart()` to begin scanning. When advertisements are received the stack sends `DM_SCAN_REPORT_IND` events to the application. The master application calls `DmPrivResolveAddr()` with the address and address type from the scan report to resolve the address with the IRK it had received previously.

After the slave application stops advertising it can call `DmAdvPrivStop()` to stop using a private resolvable address.

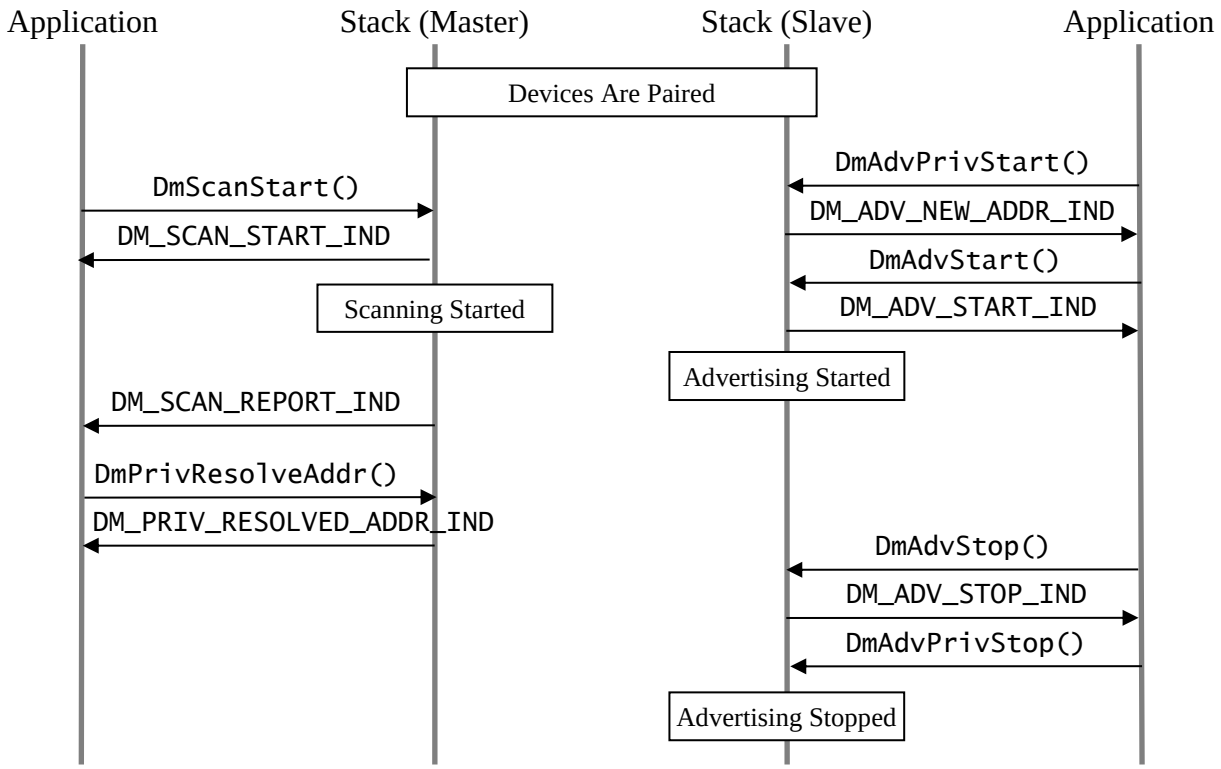

Figure 5. Privacy

10.6 ECC key generation

An ECC Key must be stored in the Device Manager prior to use of LE Secure Connections pairing. The Device Manager can generate an ECC, Elliptic Curve Cryptography, key, or the application can store an ECC Key in Non-Volatile storage. An ECC key cannot be generated until after the Device Manager reset is complete.

To generate an ECC Key, call the `DmSecGenerateEccKeyReq()` function after receiving the `DM_RESET_CMPL_IND` event. The `DM_SEC_ECC_KEY_IND` event will be called after the ECC Key generation is complete. The ECC Key can then be stored into the DM using the `DmSecSetEccKey()` function.

Note: For some applications, it may be desirable to skip ECC Key Generation and store an ECC key in Non Volatile storage. In these situations, the ECC key can be written to the Device Manager with `DmSecSetEccKey` any time after the DM is reset, and before pairing begins.

Note: The Device Manager makes use of the WSF ECC subsystem to generate and validate ECC keys. The WSF ECC subsystem may need to be ported to an application's target hardware or software framework for LE Secure Connections to operate properly.

The following figure shows the ECC Key generation scenario:

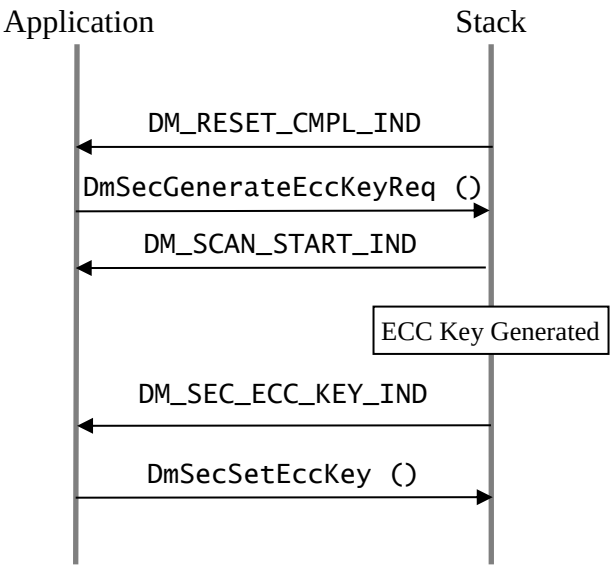

Figure 6: ECC key generation

10.7 Out of Band confirm calculation

When using Out-of-Band (OOB) LE Secure Connections pairing, devices must generate random and confirm values. Furthermore, the devices must exchange random and confirm values through an out-of-band mechanism. At which point, the local and peer random and confirm values must be stored in the Device Manager prior to OOB pairing.

The OOB confirm calculation can be performed with `DmSecCalcOobReq()`, and requires an ECC, Elliptic Curve Cryptography, key. Therefore, on receipt of the ECC key indication event, `DM_SEC_ECC_KEY_IND`, an application may call the `DmSecCalcOobReq()` function to calculate an OOB confirm value. The result of the confirm calculation will be returned via the `DM_SEC_CALC_OOB_IND` event.

After an application exchanges random and confirm values via an out-of-band mechanism with a peer, the application must store the local random and confirm values in the device manager. This can be performed with the `DmSecSetOob()` function. This must happen prior to initiating LE Secure Connections OOB Pairing.

The following figure shows the OOB confirm calculation scenario:

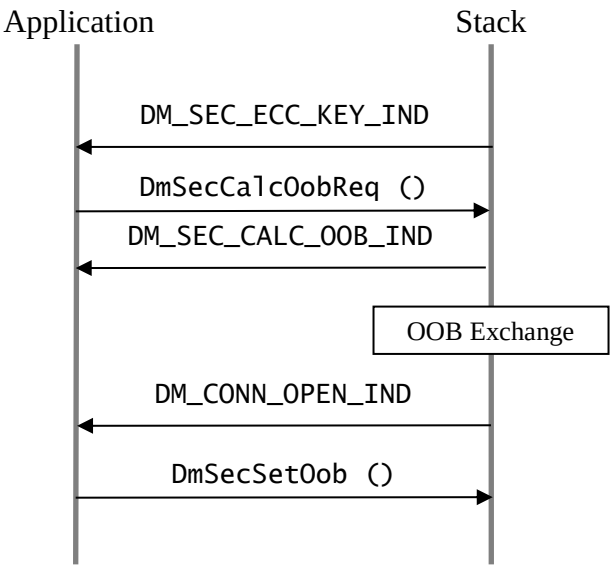

Figure 7: OOB confirm generation