Cordio® Profiles B50

Stack Profiles Developer's Guide

ARM-EPM-127303 3.0

Product status

The information in this document is for a product in development and is not final.

Web address

http://www.arm.com
http://www.arm.com/products/system-ip/cordio-radio-cores/index.php

Feedback

ARM limited welcomes feedback on both the product, and the documentation.

Feedback on this document

If you have any comments about this document, please send email to support-cordio-sw@arm.com giving:

- The document title
- The document's number
- The page number(s) to which your comments refer
- A concise explanation of your comments

General suggestion for additions and improvements are also welcome.

Support and Maintenance

Please contact support-cordio@arm.com regarding any issues with the installation, content or use of this release and a member of the ARM Product Support Group will log your query in the support database and respond as soon as possible. Note that Support for this release of the product is only provided by ARM to a recipient who has a current support and maintenance contract for the product.

Change history

Issue	Date	Release note part version
1.0	13-March-17	r2p0-10eac0
2.0	11-July-17	r2p1-01eac0
3.0	11-Sept-17	r2p3-00eac0

Contents

1 CORDIO STACK AND PROFILES

1.1 Introduction

The Cordio Stack, Cordio Profiles, and Wireless Software Foundation (WSF) IP together provide a fully compliant Bluetooth low energy solution complete with user documentation and sample application projects. This system solution is designed to work standalone, or with a proprietary or off-the-shelf embedded OS. The following section summarizes the documentation included as part of the Cordio Stack and Profiles IP deliverables. This Developer's Guide provides more in-depth information when developing a Bluetooth low energy application based on Cordio SW IP.

1.2 Documentation

The following documentation outlines the components, subsystems, API, and porting guidelines for Cordio Stack, Cordio Profiles, and the WSF.

1.2.1 WSF: Software Foundation API

The Software Foundation API document describes the functions provided by the WSF, a simple OS wrapper, porting layer, and general-purpose software service used by the Cordio embedded software system. This document is simply an API guide. Information discussing porting the WSF can be found in the Cordio Stack Porting Guide. Information on tuning and configuring the WSF in accordance with specific application requirements can be found in later sections of this document.

1.2.2 Stack: Software System

The Stack Software System document describes the building blocks and integration of the Cordio Stack and Profiles IP to build a compliant Bluetooth low energy product. Specifically this document describes the system architectural breakdown, interface and flow of data between components.

1.2.3 Stack: Porting Guide

The Stack Porting Guide provides detail for porting the WSF and HCI interfaces and services to a target system.

1.2.4 Stack: API Guides

The Cordio Stack deliverable includes API documentation on the following subsystem components: Device Manager (GAP), Security Manager (SMP), Attribute Protocol, L2CAP, and HCI.

1.2.5 Profiles: Sample Application User's Guide

The Sample App User's Guide provides an overview of the sample applications, including profiles and services, provided as part of the Cordio Profiles IP. It describes the interface to these applications, outlines the design model and configuration infrastructure for interacting with the Cordio Stack, and walks through example application code to better outline how everything fits together.

1.2.6 Profiles: API Guides

The Cordio Profiles deliverable includes API documentation on the following subsystem components: Application Framework, Profiles and Services.

2 DEVICE ROLES

The Cordio Stack can be configured to support GAP Master (central) and/or GAP Slave (peripheral) functionality. A device can also be configured to operate as a GATT Client and/or GATT Server. From the application perspective a device may operate in any combination of these roles, including supporting master and slave, and client and server simultaneously. The responsibility falls on the application to configure and manage whatever combination of roles and responsibilities are required by the device. This section provides some insight and guidelines as to how to configure device roles when building an application.

2.1 Stack Initialization

For all roles the stack must first initialize the needed stack components by setting up handlers and calling the initialization for each component that will be utilized. In our sample applications this occurs during the StackInit<app>(void) function call. Table 1 : Stack Initialization Functions defines the components that may need to be called during initialization.

Function	Role	Description
SecInit()	All Connectable Roles	Initialize general security
SecAesInit()	All Connectable Roles	Required when AES security is utilized
SecCmacInit()	All Connectable Roles	Required when CMAC operations utilized
SecEccInit()	All Connectable Roles	Required when ECC security is utilized
DmAdvInit()	GAP Slave	Initialize advertising device
DmDevPrivInit()	GAP Slave / Master	Initialize device privacy module
DmExtAdvInit()	5.0: GAP Slave	Initialize DM extended advertising
DmExtConnMasterInit()	5.0: GAP Master	Initialize DM connection manager for operation as extended master
DmExtConnSlaveInit()	5.0: GAP Slave	Initialize DM connection manager for operation as extended slave
DmExtScanInit()	5.0: GAP Master	Initialize DM extended scanning
DmScanInit()	GAP Master	Initialize scanning device
DmConnInit()	GAP Connectable	Initialize connectable device
DmConnSlaveInit()	GAP Connectable Slave	Initialize connectable slave
DmConnMasterInit()	GAP Connectable Master	Initialize connectable master
DmSecInit()	GAP Security	Initialize legacy security
DmSecLescInit()	GAP Enhanced Security	Initialize LESC security
DmPrivInit()	GAP Privacy	Initialize RPA security
DmPhyInit()	5.0: GAP Phy	Initialize PHY interface
AttsInit()	GATT Server	Initialize ATT manager for Server
AttsIndInit()	GATT Server	Initialize indications and notifications
AttsSignInit()	GATT Server	Initialize data signing
AttcInit()	GATT Client	Initialize ATT manager for Client
AttcSignInit()	GATT Client	Initialize data signing

SmpiScInit()	SMP Initiator	Initialize SMP manager for Initiator
SmprScInit()	SMP Responder	Initialize SMP manager for Responder
L2cInit()	ALL	Initialize L2CAP
L2cSlaveInit()	GAP Slave	Initialize L2CAP Slave side
L2cMasterInit()	GAP Master	Initialize L2CAP Master side
L2cCocInit()	L2CAP COC	Initialize connection oriented channel

Table 1 : Stack Initialization Functions

2.2 GAP Central / Master Role

A Central / Master device is responsible for scanning for advertisements (Observer role) and when appropriate, initiating a connection, pairing, and potentially bonding with a Peripheral device.

To configure the stack to support functionality required by the Central role, the <app>_main.c must define (discussed in Sample App User's Guide) and setup the master configuration pointers (below) and initialize the application subsystem by calling the appropriate stack initialization functions from Table 1 above.

```
 /* Set configuration pointers */
 pAppMasterCfg = (appMasterCfg_t *) &datcMasterCfg;
 pAppSecCfg = (appSecCfg_t *) &datcSecCfg;
 pAppDiscCfg = (appDiscCfg_t *) &datcDiscCfg;
 pAppCfg = (appCfg_t *)&datcAppCfg;
 pSmpCfg = (smpCfg_t *) &datcSmpCfg;

 /* Initialize application framework */
 AppMasterInit();
 AppDiscInit();
```

The application must also wire up the master application callback and then in the application handler utilize the AppMaster* API defined in app_api.h (see App Framework API) to process connection, security related, and discovery events.

```
/* process DM messages */
else if (pMsg->event <= DM_CBACK_END)
{
 /* process advertising and connection-related messages */
 AppMasterProcDmMsg((dmEvt_t *) pMsg);

 /* process security-related messages */
 AppMasterSecProcDmMsg((dmEvt_t *) pMsg);
```

When operating as a Central it is the responsibility of the application to initiate scanning.

```
 AppScanStart(datcMasterCfg.discMode, datcMasterCfg.scanType,
              datcMasterCfg.scanDuration);
```

If desired the application can initiate a connection based on the received scan report. Note the application must stop scanning before opening a connection (our sample code will use the scan report to save information on the device it wishes to connect with and then stop scanning, using the DM_SCAN_STOP_IND to trigger the connection procedure).

```
case DM_SCAN_REPORT_IND:
  datcScanReport(pMsg);
  break;

case DM_SCAN_STOP_IND:
  datcScanStop(pMsg);
  uiEvent = APP_UI_SCAN_STOP;
  break;
```

The application support layer provies an interface for initiating a connection as a Master. This api returns the dmConnId_t structure (uint8_t) used to store the connection ID that is used for subsequent communication on this connection.

```
dmConnId_t AppConnOpen(uint8_t addrType, uint8_t *pAddr, appDbHdl_t dbHdl);
```

Finally when operating as a GAP Central a device will also assume the role of the SMP Initiator. This requires calling the initialization interface and setting the appropriate SMP configuration parameters.

```
/*! SMP security parameter configuration */
static const smpCfg_t tagSmpCfg =
{
  3000,                                   /*! 'Repeated attempts' timeout in msec */
  SMP_IO_NO_IN_NO_OUT,                    /*! I/O Capability */
  7,                                      /*! Minimum encryption key length */
  16,                                     /*! Maximum encryption key length */
  3,                                      /*! Attempts to trigger 'repeated attempts' timeout */
  0                                       /*! Device authentication requirements */
};
```

2.3 GAP Peripheral / Slave Role

A Peripheral / Slave device is responsible for advertising (Broadcaster role) and handling a connection attempt from a Central / Master device. To configure the stack to support functionality required by the Peripheral role, the <app>_main.c must define (discussed in Sample App User's Guide) and setup the slave configuration pointers (below) and initialize the application subsystem by calling the appropriate stack initialization functions listed in Table 1 above.

```
 /* Set configuration pointers */
 pAppAdvCfg = (appAdvCfg_t *) &fitAdvCfg;
 pAppSlaveCfg = (appSlaveCfg_t *) &fitSlaveCfg;
 pAppSecCfg = (appSecCfg_t *) &fitSecCfg;
 pAppUpdateCfg = (appUpdateCfg_t *) &fitUpdateCfg;

 /* Initialize application framework */
 AppSlaveInit();
```

The application must also wire up the slave application callback and then in the application handler utilize the AppSlave* API defined in app_api.h (see App Framework API) to process connection, security related and discovery events.

```
  /* process DM messages */
  else if (pMsg->event <= DM_CBACK_END)
  {
    /* process advertising and connection-related messages */
    AppSlaveProcDmMsg((dmEvt_t *) pMsg);

    /* process security-related messages */
    AppSlaveSecProcDmMsg((dmEvt_t *) pMsg);
    /* process discovery-related messages */
    AppDiscProcDmMsg((dmEvt_t *) pMsg);
  }
```

When operating as a Peripheral it is the responsibility of the application to initiate advertising.

```
  /* start advertising; automatically set connectable/discoverable mode and bondable mode */
  AppAdvStart(APP_MODE_AUTO_INIT);
```

The application must also update the peer address information when a connection is established. Note the DM_CONN_OPEN_IND message contains the connection ID (dmConnId_t) in the messages hdr-param field which can be used for subsequent communication on this connection.

```
   case DM_CONN_OPEN_IND:
    tagOpen(pMsg);
    uiEvent = APP_UI_CONN_OPEN;
    break;
```

Finally when operating as a GAP Peripheral a device will also assume the role of the SMP Responder. This requires calling the initialization interface and setting the appropriate SMP configuration parameters.

```
/*! SMP security parameter configuration */
static const smpCfg_t tagSmpCfg =
{
  3000,                                   /*! 'Repeated attempts' timeout in msec */
  SMP_IO_NO_IN_NO_OUT,                    /*! I/O Capability */
  7,                                      /*! Minimum encryption key length */
  16,                                     /*! Maximum encryption key length */
  3,                                      /*! Attempts to trigger 'repeated attempts' timeout */
  0                                       /*! Device authentication requirements */
};
```

2.4 GATT Client

To act as a GATT Client for GATT Discovery the application must enumerate a list of services to be discovered and define a handle list for maintaining the handles of discovered service attributes.

```
/*! Discovery states:  enumeration of services to be discovered */
enum
{
  DATC_DISC_GATT_SVC,     /*! GATT service */
  DATC_DISC_GAP_SVC,      /*! GAP service */
  DATC_DISC_WP_SVC,       /*! Wicentric proprietary service */
  DATC_DISC_SVC_MAX       /*! Discovery complete */
};

/*! Pointers into handle list for each service's handles */
static uint16_t *pDatcGattHdlList = &datcCb.hdlList[DATC_DISC_GATT_START];
static uint16_t *pDatcGapHdlList = &datcCb.hdlList[DATC_DISC_GAP_START];
static uint16_t *pDatcWpHdlList = &datcCb.hdlList[DATC_DISC_WP_START];
```

During discovery these handle lists are populated so a Client can read/write attributes.

```
/* discover proprietary data service */
WpcP1Discover(connId, pDatcWpHdlList);
```

To receive indications / notifications the application must configure the appropriate CCCD upon completing discovery.

```
/* start configuration */
AppDiscConfigure(connId, APP_DISC_CFG_START, DATC_DISC_CFG_LIST_LEN,
                 (attcDiscCfg_t *) datcDiscCfgList, DATC_DISC_HDL_LIST_LEN, datcCb.hdlList);
```

2.5 GATT Server

To configure the GATT Server for GATT Discovery a device's application in general will share a list of services supported as part of its advertisememt and will register the CCCDs for supported services with ATT.

```
AttsCccRegister(FIT_NUM_CCC_IDX, (attsCccSet_t *) fitCccSet, fitCccCback);
```

A Server application must also handle the case where CCCD values are changed to ensure that attributes are notified / indicated as appropriate.

```
static void fitProcCccState(fitMsg_t *pMsg)
{
  APP_TRACE_INFO3("ccc state ind value:%d handle:%d idx:%d", pMsg->ccc.value, pMsg->ccc.handle, pMsg->ccc.idx);

 /* handle heart rate measurement CCC */
 if (pMsg->ccc.idx == FIT_HRS_HRM_CCC_IDX)
 {
  if (pMsg->ccc.value == ATT_CLIENT_CFG_NOTIFY)
  {
```

The application must register its ATT services with the GATT server:

```
  /* Initialize attribute server database */
  SvcCoreAddGroup();
  SvcHrsCbackRegister(NULL, HrpsWriteCback);
  SvcHrsAddGroup();
  SvcDisAddGroup();
  SvcBattCbackRegister(BasReadCback, NULL);
  SvcBattAddGroup();
  SvcRscsAddGroup();
```

2.6 Advertising Only

When implementing a minimal advertising only device the subsystem initialization for connections and GATT/ATT are not required. In addition you will not require code for security, key and device management, or services and the discovery process. To minimize code size you will need to ensure that you are not initializing or calling any unnecessary initialization functions to avoid compiling in those libraries.

When calling `AppSetAdvType(uint8_t advType)` the type should be set to `DM_ADV_NONCONN_UNDIRECT.`

3 BUFFER / POOL MANAGEMENT

The WSF buffer management service is a pool-based dynamic memory allocation service. The buffer service interface is defined in file wsf_buf.h. You can learn more about the buffer management system in the Software Foundation API.

3.1 Configuring Buffer Pools

The buffer pools are configured as part of the application system configuration.

```
/*! \brief      Number of WSF buffer pools. */
#define WSF_BUF_POOLS         4

/*! \brief      Buffer size. */
#define ACL_BUF_SIZE          256

/*! \brief      Total number of buffers. */
#define NUM_BUFS              8

/*! \brief      Default pool descriptor. */
static wsfBufPoolDesc_t mainPoolDesc[WSF_BUF_POOLS] =
{
  { 16,   8 },
  { 32,   8 },
  { 64,   8 },
  { ACL_BUF_SIZE, NUM_BUFS }
```

Buffer allocation is a system configuration and the WSF buffer interface supports allocating one or more buffer pools for use by the application and protocol stack as desired. However in general we recommend choosing the size and number of buffer pools to coincide with application requirements. Below you will find additional information on collecting pool statistics while running your application to better tune buffer pool settings. We also note that while the buffer memory does not need to be a power of 2, it does need to be word aligned (4 bytes for ARM Cortex M).

3.2 Pool Analysis and Optimization

When the stack or application requests a buffer the WSF buffer allocation attempts to locate an available buffer from the smallest pool that meets the minimum size criteria. If no buffers are available from the ideal pool the allocation mechanism will then allocate from larger pools based on availability. As a result an improperly sized system is likely to waste valuable RAM if the buffer pools are not configured in accordance with application and system requirements. To address this issue the WSF IP includes diagnostic functions (see Software Foundation API) to monitor and tune buffer configuration in accordance with application requirements.

To enable buffer statistics set WSF_BUF_STATS to TRUE. This will enable statistics on the buffer pool to be collected while a device is operational. Statistics collected include the number of buffers currently allocated, the maximum allocations from a buffer pool, and the maximum size buffer allocated from each pool. These statistics can then be accessed using the WsfBufGetPoolStats(…); interface. After running your application and executing as much potential functionality as the device supports, these buffer pool statistics can be accessed and analyzed. If it is found that the number of allocations is maxing out within a buffer pool than the system is likely utilizing buffers from a larger pool unnecessarily. To address this you should adjust the number of buffers for a given pool and re-run the application. If it is found that the maximum required buffer from a pool is less than the size of the buffers in that pool than you can reduce the buffer sizes for that pool accordingly as the extra memory set aside is not being utilized.

In addition you can enable WSF_BUF_STATS_HIST to enable a histogram of buffer pool statistics. This enables tracking buffer allocations through a 128 byte array indexed by length of the allocated buffer. Collected data can be accessed through

```
uint8_t *WsfBufGetAllocStats(void)
```

4 SECURITY

Bluetooth low energy supports data encryption used to prevent passive and active man-in-the-middle (MITM) eavesdropping attacks on a link. In order to establish shared keys for encryption Bluetooth low energy devices must pair and exchange various keys depending on security requirements. Pairing is carried out in three phases:

1. Two connected devices announce their input and output capabilities and from that information determine a suitable method for the next phase.
2. A Short Term Key (STK) is generated and used in the final phase to secure long term key distribution.
3. Up to three keys are distributed using the STK. This includes the Long Term Key (LTK) used for Link Layer encryption and authentication; the Connection Signature Resolving Key (CSRK) used for data signing at the ATT layer; the Identity Resolving Key (IRK) used to generate and resolve Private Random Addresses (RPAs).

Bluetooth low energy also supports Out Of Band (OOB) Pairing, where instead of the previously mentioned 3 phase key exchanges the required secret keys are exchanged using another communication medium such as Near Field Communication (NFC). For more information on security see the latest Bluetooth core specification.

4.1 Security Manager Configuration

STK key generation and exchange depends on a device's input and output capabilities. To configure I/O capabilities for an application set the appropriate I/O capability in the smpCfg_t parameter set. Supported I/O capabilities are:

```
/*! I/O capabilities */
#define SMP_IO_DISP_ONLY              0x00    /*! DisplayOnly */
#define SMP_IO_DISP_YES_NO            0x01    /*! DisplayYesNo */
#define SMP_IO_KEY_ONLY               0x02    /*! KeyboardOnly */
#define SMP_IO_NO_IN_NO_OUT           0x03    /*! NoInputNoOutput */
#define SMP_IO_KEY_DISP               0x04    /*! KeyboardDisplay */
```

To enable security upon connection set the initiateSec parameter of appSecCfg_t to TRUE and set the authentication and bonding flags (auth).

```
/*! Pairing authentication/security properties bit mask */
#define DM_AUTH_BOND_FLAG            SMP_AUTH_BOND_FLAG    /*! Bonding requested */
#define DM_AUTH_MITM_FLAG            SMP_AUTH_MITM_FLAG    /*! MITM (authenticated pairing) requested */
#define DM_AUTH_SC_FLAG              SMP_AUTH_SC_FLAG      /*! LE Secure Connections requested */
#define DM_AUTH_KP_FLAG              SMP_AUTH_KP_FLAG      /*! Keypress notifications requested */
```

Also set the initiator (master) key distribution (iKeyDist) and the responder (slave) key distribution (rKeyDist) flags.

```
/*! Key distribution bit mask */
#define DM_KEY_DIST_LTK         SMP_KEY_DIST_ENC      /*! Distribute LTK used for encryption */
#define DM_KEY_DIST_IRK         SMP_KEY_DIST_ID       /*! Distribute IRK used for privacy */
#define DM_KEY_DIST_CSRK        SMP_KEY_DIST_SIGN     /*! Distribute CSRK used for signed data */
```

Resolvable Public Addresses (RPAs) are used when an IRK is present. If a device enables IRK key distribution it must also configure the stack with a specified IRK. This is accomplished by first defining a 16 byte IRK

```
/*! local IRK */
static uint8_t localIrk[] =
{
  0xA6, 0xD9, 0xFF, 0x70, 0xD6, 0x1E, 0xF0, 0xA4, 0x46, 0x5F, 0x8D, 0x68, 0x19, 0xF3, 0xB4, 0x96
};
```

and during system initialization registering the IRK with the Device Manager (DM).

```
/* Set IRK for the local device */
DmSecSetLocalIrk(localIrk);
```

As part of Privacy 1.2, the LL can also generate and resolve RPA's. If the RPA is not resolved by the Controller, it is then the responsibility of the application to utilize a peer device's IRK to resolve the RPA when appropriate

```
/* if the peer device uses an RPA */
else if (DM_RAND_ADDR_RPA(pMsg->scanReport.addr, pMsg->scanReport.addrType))
{
  /* reslove advertiser's RPA to see if we already have a bond with this device */
  AppMasterResolveAddr(pMsg, APP_DB_HDL_NONE, APP_RESOLVE_ADV_RPA);
}
```

When configuring a device to use a Resolve Private Address (RPA) the application should call `DmAdvPrivStart(uint16_t changeInterval)` with the changeInterval in seconds. This will automatically enable the use of LL privacy, if supported.

5 DATA AND THROUGHPUT

This section discussed how to maximize data throughput with the Cordio Stack and Profiles.

5.1 MTU Payload Size

The permitted MTU size is going to depend on the capabilities of both master and slave. During pairing the MTU exchange will determine the maximum size supported, which is the lesser of the supported MTU sizes of both devices. To configure the MTU payload size supported by the Cordio Stack set the <app>AttCfg's 2nd parameter, the desired ATT MTU size. Note: If the application does not contain a <app>AttCfg structure then one should be created.

```
const attCfg_t tagAttCfg = { … }
```

This structure must then be configured during application initialization.

```
pAttCfg = (attCfg_t *) &tagAttCfg;
```

In addition since the stack does not allow us to send packets into the HCI unless we are also capable of receiving a packet of the same size (minus overhead), we also need to configure the HCI on the read side to be able to handle a packet of the configured size + 4 bytes of ATT overhead.

```
HciSetMaxRxAclLen(size);
```

Once this is done the application is now capable of sending payloads of the configured size to the controller to be sent. Please note that other considerations that must be taken into consideration are the supported maximum MTU size specified by the version of the stack being utilized, the Controller's maximum supported buffer sizes for receiving packets over HCI, as well as fragmentation features in L2CAP, HCI, and in the Controller itself. For example the HCI_OPCODE_LE_READ_BUF_SIZE is used to determine the maximum size packet the controller can receive over HCI and any packet larger than this size will be discarded by the Controller.

5.2 Negotiated MTU Size

While connecting peer devices go through a negotiation of the maximum supported MTU size. In the case that this negotiated size is greater than the set default value the stack will issue a callback notifying the application of this negotiated maximum. The handler for application callbacks (<app>ProcMsg(…)) requires a case for the `ATT_MTU_UPDATE_IND` to handle this event. The mtu size information can be extracted from the attEvt_t structure's mtu field (pMsg->mtu).

5.3 Maximize Throughput

There are many considerations when attempting to maximize throughput at the application layer including the supported version and features of the Controller on both sides of the link as well as buffer allocation and HCI configuration limitations. What follows is a general guideline for maximizing application throughput.

In general to maximize throughput at the application layer we want to fill up the controller's buffers as quickly as possible and keep them as full as possible so the utilization of the time allocated on the PHY for communication between the two devices is maximized. We want to maximize the payload size to reduce overhead as much as possible. Note that the limit here is going to be the maximum allowable packet size supported by and negotiated by both devices (see previous section). Finally at the application layer we want to package data into maximum packets (which will get fragmented at the HCI layer and in the Controller). However these packets can't be so big that they create memory problems or overrun the stacks ability to receive a similar size packet.

One influential configuration parameter negotiated between the Master and Peripheral devices is the connection interval. On the one hand the Controller can increase the connection interval to a maximum value allowing more unacknowledged packets (Notifications and Write Commands) to be sent per connection interval, thereby minimizing time “wasted” in sleep and PHY management overhead. On the other hand if you want to have the ability to maximize throughput of packets such as Indications and Write Requests then you want to set the connection interval to a minimum since the protocol limits the sendings of these packet types to a single handshake exchange per connection interval.

In configuring the stack for maximum throughput we also want to understand the maximum packet that can be received by the Controller over HCI. To do this we read the LE_READ_BUF_SIZE during system reset. This provides a limit on the size that the Controller can accept, so anything larger passed to HCI will have to be fragmented first. We then need to configure the application to maximize the supported MTU size in correspondence with the maximum packet size supported by the Controller and the number of supported buffers. For example if the Controller supports a maximum packet of 256 bytes and there are 4 buffers available, then the application could send a payload of (256 x 4 – 4 [payload overhead]) 1020 bytes which would get fragmented at the HCI layer and passed to the Controller in 4 separate payloads. This would essentially keep the Controller buffers full and maximize use of the channel. See Section 5.1 for more information on configuring MTU size.

Once this is done the application is now configured to send the maximum payload size capable of being buffered in the Controller. In general to maximize throughput you will need to use Notifications when sending data, as Indications require acklowledgement before another message can be sent and therefore add significant overhead and delay.

6 BLUETOOTH 5.0 SUPPORT

Licensees of Cordio BT5 Stack and Cordio BT5 Profiles software IP can enable Bluetooth 5 features within the protocol stack. Enhanced features include support for enabling the 2 Mbps and Long Range (coded PHY) modes of operation, enabling use of Advertising Extensions including support for periodic advertising and scanning, and more. This section discusses these included features including how to enable them.

6.1 Advertising Extensions

To enable advertising extensions for an application configuration currently utilizing legacy advertising on the Slave side simply replace the DmAdvInit() initialization call with one to DmExtAdvInit() and replace the DmConnMasterInit() call with one to DmExtConnMasterInit(). Similarly enable advertising extensions on the Master side by replacing DmScanInit() with DmExtScanInit() and DmConnSlaveInit() with DmExtConnSlaveInit() during system startup. When configuring the advertising type through AppSetAdvType(advType) you can also use the Extended Advertising type's defined in dm_api.h.

Name	Value	Description
DM_EXT_ADV_CONN_UNDIRECT	5	Connectable undirected advertising.
DM_EXT_ADV_NONCONN_DIRECT	6	Non-connectable and non-scannable directed advertising
DM_EXT_ADV_SCAN_DIRECT	7	Scannable directed advertising

In addition to the legacy DM API's, the following API can be utilized for an advertising device utilizing AE features:

```
void DmPerAdvConfig(uint8_t advHandle)
void DmPerAdvSetData(uint8_t advHandle, uint8_t op, uint8_t len, uint8_t *pData)
void DmPerAdvStart(uint8_t advHandle)
void DmPerAdvStop(uint8_t advHandle)
void DmAdvSetRandAddr(uint8_t advHandle, const uint8_t *pAddr)
void DmAdvUseLegacyPdu(uint8_t advHandle, bool_t useLegacyPdu)
void DmAdvOmitAdvAddr(uint8_t advHandle, bool_t omitAdvAddr)
void DmAdvIncTxPwr(uint8_t advHandle, bool_t incTxPwr, int8_t advTxPwr)
void DmAdvSetPhyParam(uint8_t advHandle, uint8_t priAdvPhy, uint8_t secAdvMaxSkip, uint8_t secAdvPhy)
void DmAdvScanReqNotifEnable(uint8_t advHandle, bool_t scanReqNotifEna)
void DmAdvSetFragPref(uint8_t advHandle, uint8_t fragPref)
void DmPerAdvSetInterval(uint8_t advHandle, uint16_t intervalMin, uint16_t intervalMax)
void DmPerAdvIncTxPwr(uint8_t advHandle, bool_t incTxPwr)
bool_t DmAdvModeExt(void)
```

In addition to the legacy DM API's, the following API can also be utilized by a scanning device utilizing AE features:

```
void DmSyncStart(uint8_t advSid, uint8_t advAddrType, const uint8_t *pAdvAddr, uint16_t skip, uint16_t syncTimeout)
void DmSyncStop(uint16_t syncId)
void DmAddDeviceToPerAdvList(uint8_t advAddrType, const uint8_t *pAdvAddr, uint8_t advSid)
void DmRemoveDeviceFromPerAdvList(uint8_t advAddrType, const uint8_t *pAdvAddr, uint8_t advSid)
void DmClearPerAdvList(void)
bool_t DmScanModeExt(void)
```

For more information on these function please reference the Device Manager API Guide.

6.2 2Mbps PHY

The BT5 Stack inherently supports 2Mbps mode. 2Mpbs mode can be enabled utilizing the DmSetPhy() call (See the Device Manager API Guide for more information). See the dats sample application for an example configuration that support 2Mbps operation.

7 SYSTEM CONFIGURATION

7.1 Stack Configuration Options

The following configurable parameters are defined in cfg_stack.h (./ble-host/sources/stack/cfg/cfg_stack.h)

7.1.1 HCI_TX_DATA_TAILROOM

For single CPU solutions using Cordio Stack IP we support a zero-copy interface which allows for optimal performance by eliminating unnecessary data copying. Zero-copy is used when a packet sent across the HCI interface from the Host to the Controller does not require fragmentation. In this scenario extra space must be allocated in the buffer for Link Layer operations and is defined by this parameter. Please note that when the zero-copy API is utilized the LL is required to free the HCI buffer.

7.1.2 DM_CONN_MAX

DM_CONN_MAX specifies the maximum number of connections that the stack can simultaneously support. This parameter is set to 3 by default with the only limitation on DM_CONN_MAX's configured value being the available memory of the system.

7.1.3 DM_SYNC_MAX

DM_SYNC_MAX defines the maximum number of periodic synchronized advertisements supported by the host stack by the scanner / observer. This parameter is specific to Bluetooth 5.0 and later versions of the specification. The default value for this parameter is set to 1.

7.1.4 DM_NUM_ADV_SETS

DM_NUM_ADV_SETS defines the number of supported advertising sets as defined by the Bluetooth 5.0 specification. Note that this parameter must be set to 1 for legacy advertising.

7.1.5 DM_NUM_PHYS

DM_NUM_PHYS defines the number of scanner and initiator PHYs (LE 1M, LE 2M, LE Coded) as defined by the Bluetooth 5.0 specification. Note that this parameter must be set to 1 for legacy scanner and initiator functionality to work.

7.2 HCI Flow Control

The Cordio Stack implements a flow control mechanism to prevent overwhelming the Controller with more data than it can handle. The flow control mechanism operates as follows:

A call to an ATT function (e.g. AttsHandleValueNtf) that results in an over-the-air packet from the application results in copying the application data into a WSF buffer (if available) and scheduling the packet for delivery to L2CAP. The message is then handled by the L2CAP process and passed down to the HciSendAclData() function where it is sent out over HCI if buffers are available, after which a Confirm status is propogated back up to the application so the application knows the packet has gone out (over HCI – not over the air).

To manage flow control for the case that the application / stack tries to overwhelm the HCI with too many packets, a flow control mechanism is configured per connection. The pConn->flowDisabled flag manages whether or not packet flow into HCI has been disabled.

The flow control mechanism kicks in when all of the HCI buffers are full. This occurs when the number of queued buffers for a connection exceeds the configurable hi water mark (HCI_ACL_QUEUE_HI). When this occurs the HCI connections (hciCoreConn_t) flowDisabled flag is set to prevent further buffering of data. Flow control is later re-enabled after enough HCI Number of Completed Packet Events are processed by the host such that the outgoing (queued) buffer list drops to or below the configured low water mark (HCI_ACL_QUEUE_LO). The high and low water marks for flow control are defined in ./ble-host/sources/hci/common/hci-core.c.

The HCI flow control mechanism is wired up through L2CAP to the ATT and SMP modules to simultaneously enable and disable flow control higher up in the stack. For the ATT server the callbacks are wired up to the attsIndCtrlCback (note: the callbacks are not wired up for the ATT client) when flow control is re-enabled.

8 PERSISTENT MEMORY

This section describes structures and information that should be saved in persistent storage.

8.1 Application Database

The application database (app_db.c) stores records for bonding with peer devices and should be stored in persistent memory. Note that our out-of-the box implementation of the stack saves all data in non-persistent memory, so modifications must be made based on the host CPU.

```
/*! Database */
static appDb_t appDb;
```