CC254x/CC2540/CC2541 库函数速查

hci.h

```
//分配内存,应用程序不应该调用这个函数.
void *HCI_bm_alloc( uint16 size );
//检查连接时间参数和连接时间参数的组合是否有效
uint8 HCI_ValidConnTimeParams( uint16 connIntervalMin,
                               uint16 connIntervalMax,
                               uint16 connLatency,
                               uint16 connTimeout );
//HCI vendor specific registration for HCI Test Applicationvoid
HCI_TestAppTaskRegister( uint8 taskID );
// HCI vendor specific registration for Host GAP.
void HCI_GAPTaskRegister( uint8 taskID );
//HCI vendor specific registration for Host L2CAP.
void HCI_L2CAPTaskRegister( uint8 taskID );
//HCI vendor specific registration for Host SMP.
void HCI_SMPTaskRegister( uint8 taskID );
// HCI vendor specific registration for Host extended commands.
void HCI_ExtTaskRegister( uint8 taskID );
//发送一个 ACL 数据包
hciStatus_t HCI_SendDataPkt( uint16 connHandle,
                             uint8  pbFlag,
                             uint16 pktLen,
                             uint8  *pData );
//中断连接
hciStatus_t HCI_DisconnectCmd( uint16 connHandle,
                               uint8  reason );//请求得到版本信息
hciStatus_t HCI_ReadRemoteVersionInfoCmd( uint16 connHandle );
//设置消息蒙版,确定支持哪些消息
hciStatus_t HCI_SetEventMaskCmd( uint8 *pMask );
//重置连接层
hciStatus_t HCI_ResetCmd( void );
//读取发射功率
hciStatus_t HCI_ReadTransmitPowerLevelCmd( uint16 connHandle,
                                           uint8  txPwrType );
//主机用来开关流量控制(控制器法向主机的)
hciStatus_t HCI_SetControllerToHostFlowCtrlCmd( uint8 flowControlEnable );
//This BT API is used by the Host to notify the Controller of the maximum size ACL buffer
size the Controller can send to the Host.
```

```
hciStatus_t HCI_HostBufferSizeCmd( uint16 hostAclPktLen,
                                   uint8  hostSyncPktLen,
                                   uint16 hostTotalNumAclPkts,
                                   uint16 hostTotalNumSyncPkts );
//This BT API is used by the Host to notify the Controller of the number of HCI data packets that have been completed for each connection handle since this command was previously sent to the controller.
hciStatus_t HCI_HostNumCompletedPktCmd( uint8  numHandles,
                                        uint16 *connHandles,
                                        uint16 *numCompletedPkts );
//读取本地版本信息
hciStatus_t HCI_ReadLocalVersionInfoCmd( void );
//读取本地支持的命令
hciStatus_t HCI_ReadLocalSupportedCommandsCmd( void );
//读取本地支持的特性
hciStatus_t HCI_ReadLocalSupportedFeaturesCmd( void );
//读取设备的 BLE 地址
hciStatus_t HCI_ReadBDADDRCmd( void );
//读取最后一个包的 RSSI
hciStatus_t HCI_ReadRssiCmd( uint16 connHandle );
//设置 HCI LE 时间蒙版,确定支持哪些事件
hciStatus_t HCI_LE_SetEventMaskCmd( uint8 *pEventMask );
//取得最大 ACL 数据包大小
hciStatus_t HCI_LE_ReadBufSizeCmd( void );
//取得本地支持的特性
hciStatus_t HCI_LE_ReadLocalSupportedFeaturesCmd( void );
//设置设别随机地址
hciStatus_t HCI_LE_SetRandomAddressCmd( uint8 *pRandAddr );
//设置广播参数
hciStatus_t HCI_LE_SetAdvParamCmd( uint16 advIntervalMin,
                                   uint16 advIntervalMax,
                                   uint8  advType,
                                   uint8  ownAddrType,
                                   uint8  directAddrType,
                                   uint8  *directAddr,
                                   uint8  advChannelMap,
                                   uint8  advFilterPolicy );
//设置广播数据
hciStatus_t HCI_LE_SetAdvDataCmd( uint8 dataLen,
                                  uint8 *pData );
//设置广播搜索响应数据
hciStatus_t HCI_LE_SetScanRspDataCmd( uint8 dataLen,
                                      uint8 *pData );
//开关广播
```

```
hciStatus_t HCI_LE_SetAdvEnableCmd( uint8 advEnable );
//读取广播时的发射功率
hciStatus_t HCI_LE_ReadAdvChanTxPowerCmd( void );
//设置搜索参数
hciStatus_t HCI_LE_SetScanParamCmd( uint8  scanType,
                                    uint16 scanInterval,
                                    uint16 scanWindow,
                                    uint8  ownAddrType,
                                    uint8  filterPolicy );
//开关搜索
hciStatus_t HCI_LE_SetScanEnableCmd( uint8 scanEnable,
                                     uint8 filterDuplicates );
//建立连接
hciStatus_t HCI_LE_CreateConnCmd( uint16 scanInterval,
                                  uint16 scanWindow,
                                  uint8  initFilterPolicy,
                                  uint8  addrTypePeer,
                                  uint8  *peerAddr,
                                  uint8  ownAddrType,
                                  uint16 connIntervalMin,
                                  uint16 connIntervalMax,
                                  uint16 connLatency,
                                  uint16 connTimeout,
                                  uint16 minLen,
                                  uint16 maxLen );
//取消创建连接
hciStatus_t HCI_LE_CreateConnCancelCmd( void );
//读取白名单
hciStatus_t HCI_LE_ReadWhiteListSizeCmd( void );
//清除白名单
hciStatus_t HCI_LE_ClearWhiteListCmd( void );
//添加一条白名单
hciStatus_t HCI_LE_AddWhiteListCmd( uint8 addrType,
                                    uint8 *devAddr );
//移除一条白名单
hciStatus_t HCI_LE_RemoveWhiteListCmd( uint8 addrType,
                                       uint8 *devAddr );
//更新连接参数
hciStatus_t HCI_LE_ConnUpdateCmd( uint16 connHandle,
                                  uint16 connIntervalMin,
                                  uint16 connIntervalMax,
                                  uint16 connLatency,
                                  uint16 connTimeout,
                                  uint16 minLen,
```

```
                                        uint16 maxLen );
//更新当前数据通道 MAP
hciStatus_t HCI_LE_SetHostChanClassificationCmd( uint8 *chanMap );
//读取连接数据通道 MAP
hciStatus_t HCI_LE_ReadChannelMapCmd( uint16 connHandle );
//读取远程设备用户特性
hciStatus_t HCI_LE_ReadRemoteUsedFeaturesCmd( uint16 connHandle );
//执行 AES128 加密
hciStatus_t HCI_LE_EncryptCmd( uint8 *key,
                                    uint8 *plainText );
//产生随机数
hciStatus_t HCI_LE_RandCmd( void );
//连接中开始加密
hciStatus_t HCI_LE_StartEncyptCmd( uint16 connHandle,
                                       uint8  *random,
                                       uint8  *encDiv,
                                       uint8  *ltk );
//主机向控制器发送一个 LTK 回应
hciStatus_t HCI_LE_LtkReqReplyCmd( uint16 connHandle,
                                       uint8  *ltk );
//This LE API is used by the Host to send to the Controller a negative LTK reply.
hciStatus_t HCI_LE_LtkReqNegReplyCmd( uint16 connHandle );
//读取控制器支持的状态
hciStatus_t HCI_LE_ReadSupportedStatesCmd( void );
// This LE API is used to start the receiver Direct Test Mode test.
hciStatus_t HCI_LE_ReceiverTestCmd( uint8 rxFreq );
//This LE API is used to start the transmit Direct Test Mode test.
hciStatus_t HCI_LE_TransmitterTestCmd( uint8 txFreq,
                                           uint8 dataLen,
                                           uint8 pktPayload );
//This LE API is used to end the Direct Test Mode test.
hciStatus_t HCI_LE_TestEndCmd( void );
//This HCI Extension API is used to set the receiver gain.
hciStatus_t HCI_EXT_SetRxGainCmd( uint8 rxGain );
//设置发射功率
hciStatus_t HCI_EXT_SetTxPowerCmd( uint8 txPower );
//设置是否连接中一个消息只能包含一个包
hciStatus_t HCI_EXT_OnePktPerEvtCmd( uint8 control );
//This HCI Extension API is used to set whether the system clock will be divided when
the MCU is halted.
hciStatus_t HCI_EXT_ClkDivOnHaltCmd( uint8 control );
//This HCI Extension API is used to indicate to the Controller whether or not the Host
will be using the NV memory during BLE operations.
hciStatus_t HCI_EXT_DeclareNvUsageCmd( uint8 mode );
```

```
//使用 AES128 解密
hciStatus_t HCI_EXT_DecryptCmd( uint8 *key,
                                uint8 *encText );
//设置支持的特性
hciStatus_t HCI_EXT_SetLocalSupportedFeaturesCmd( uint8 *localFeatures );
//设置尽快发送数据
hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd( uint8 control );
//This HCI Extension API is used to to enable or disable suspending slave latency.
hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd( uint8 control );
//This API is used start a continuous transmitter modem test, using either a modulated or unmodulated carrier wave tone, at the frequency that corresponds to the specified RF channel. Use HCI_EXT_EndModemTest command to end the test.
hciStatus_t HCI_EXT_ModemTestTxCmd( uint8 cwMode,
                                    uint8 txFreq );
//This API is used to start a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37 byte packet of Pseudo-Random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0..39) every 625us. Use HCI_EXT_EndModemTest command to end the test.
hciStatus_t HCI_EXT_ModemHopTestTxCmd( void );
//This API is used to start a continuous receiver modem test using a modulated carrier wave tone, at the frequency that corresponds to the specific RF channel. Any received data is discarded. Receiver gain may be adjusted using the HCI_EXT_SetRxGain command. RSSI may be read during this test by using the HCI_ReadRssi command. Use HCI_EXT_EndModemTest command to end the test.
hciStatus_t HCI_EXT_ModemTestRxCmd( uint8 rxFreq );
//This API is used to shutdown a modem test. A complete Controller reset will take place.
hciStatus_t HCI_EXT_EndModemTestCmd( void );
//设置设备的 BLE 地址
hciStatus_t HCI_EXT_SetBDADDRCmd( uint8 *bdAddr );
//设置设备的睡眠时钟精度
hciStatus_t HCI_EXT_SetSCACmd( uint16 scaInPPM );
//This HCI Extension API is used to enable Production Test Mode.
hciStatus_t HCI_EXT_EnablePTMCmd( void );
//This HCI Extension API is used to set the frequency tuning up or down. Setting the mode up/down decreases/increases the amount of capacitance on the external crystal oscillator.
hciStatus_t HCI_EXT_SetFreqTuneCmd( uint8 step );
//保存频率调谐值到 Flash
hciStatus_t HCI_EXT_SaveFreqTuneCmd( void );
//This HCI Extension API is used to set the maximum transmit output power for Direct Test Mode.
hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd( uint8 txPower );

llStatus_t HCI_EXT_MapPmIoPortCmd( uint8 ioPort, uint8 ioPin );
//立即断开连接
```

```
hciStatus_t HCI_EXT_DisconnectImmedCmd( uint16 connHandle );
//读取或复位包错误率计数器
hciStatus_t HCI_EXT_PacketErrorRateCmd( uint16 connHandle, uint8 command );
//开始或结束包错误率计数
hciStatus_t HCI_EXT_PERbyChanCmd( uint16 connHandle, perByChan_t *perByChan );
//This HCI Extension API is used to Extend Rf Range using the TI CC2590 2.4 GHz RF Front End device
hciStatus_t HCI_EXT_ExtendRfRangeCmd( void );
//This HCI Extension API is used to enable or disable halting the CPU during RF. The system defaults to enabled.
hciStatus_t HCI_EXT_HaltDuringRfCmd( uint8 mode );
//This HCI Extension API is used to enable or disable a notification to the specified task using the specified task event whenever a Adv event ends. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent is taken to be "disable".
hciStatus_t HCI_EXT_AdvEventNoticeCmd( uint8 taskID, uint16 taskEvent );
//This HCI Extension API is used to enable or disable a notification to the specified task using the specified task event whenever a Connection event ends. A non-zero taskEvent value is taken to be "enable", while a zero valued taskEvent taken to be "disable".
hciStatus_t HCI_EXT_ConnEventNoticeCmd( uint8 taskID, uint16 taskEvent );
//设置用户版本号
hciStatus_t HCI_EXT_BuildRevisionCmd( uint8 mode, uint16 userRevNum );
```

lacap.h

```
//初始化 L2CAP 层 void L2CAP_Init( uint8 taskId );
//L2CAP 任务时间处理函数
uint16 L2CAP_ProcessEvent( uint8 taskId, uint16 events );
//为协议或程序注册一个 L2CAP 通道
bStatus_t L2CAP_RegisterApp( uint8 taskId, uint16 CID );
//发送 L2CAP 数据包
bStatus_t L2CAP_SendData( uint16 connHandle, l2capPacket_t *pPkt );
//发送拒绝命令
bStatus_t L2CAP_CmdReject( uint16 connHandle, uint8 id, l2capCmdReject_t *pCmdReject );
//建立拒绝命令
uint16 L2CAP_BuildCmdReject( uint8 *pBuf, uint8 *pCmd );
//发送 L2CAP Echo 请求
bStatus_t L2CAP_EchoReq( uint16 connHandle, l2capEchoReq_t *pEchoReq, uint8 taskId );
//发送 L2CAP 信息请求
bStatus_t L2CAP_InfoReq( uint16 connHandle, l2capInfoReq_t *pInfoReq, uint8 taskId );
//建立信息响应
uint16 L2CAP_BuildInfoRsp( uint8 *pBuf, uint8 *pCmd );
//解析信息请求
```

```
bStatus_t L2CAP_ParseInfoReq( l2capSignalCmd_t *pCmd, uint8 *pData, uint16 len );
//发送 L2CAP 连接参数更新请求
bStatus_t L2CAP_ConnParamUpdateReq( uint16 connHandle, l2capParamUpdateReq_t
*pUpdateReq, uint8 taskId );
//解析连接参数更新请求
bStatus_t L2CAP_ParseParamUpdateReq( l2capSignalCmd_t *pCmd, uint8 *pData, uint16
len );
//发送连接参数更新响应
bStatus_t L2CAP_ConnParamUpdateRsp( uint16 connHandle, uint8 id, l2capParamUpdateRsp_t
*pUpdateRsp );
//建立连接参数更新响应
uint16 L2CAP_BuildParamUpdateRsp( uint8 *pBuf, uint8 *pData );
//在 L2CAP 层分配内存 void *L2CAP_bm_alloc( uint16 size );
```

gatt.h

```
//初始化 GATT 客户端
bStatus_t GATT_InitClient(void);
//注册接收 ATT 的 Indications 或 Notifications 属性值 void GATT_RegisterForInd(uint8
taskId);
//准备写请求用于请求服务器准备写一个属性的值
bStatus_t GATT_PrepareWriteReq(uint16 connHandle, attPrepareWriteReq_t *pReq, uint8
taskId);
//执行写请求
bStatus_t GATT_ExecuteWriteReq(uint16 connHandle, attExecuteWriteReq_t *pReq, uint8
taskId);
//初始化 GATT 服务器
bStatus_t GATT_InitServer(void);
//为 GATT 服务器注册服务属性列表
bStatus_t GATT_RegisterService(gattService_t *pService);
//为 GATT 服务器注销一个属性列表
bStatus_t GATT_DeregisterService(uint16 handle, gattService_t *pService);
//注册接收 ATT 请求 void GATT_RegisterForReq(uint8 taskId);
//验证属性的读取权限
bStatus_t GATT_VerifyReadPermissions(uint16 connHandle, uint8 permissions);
//验证属性的写权限
bStatus_t GATT_VerifyWritePermissions(uint16 connHandle, uint8 permissions,
attWriteReq_t *pReq);
//发送服务改变 Indication
uint8 GATT_ServiceChangedInd(uint16 connHandle, uint8 taskId);
//通过 UUID 找到属性记录
gattAttribute_t *GATT_FindHandleUUID(uint16 startHandle, uint16 endHandle, const uint8
```

```
*pUUID, uint16 len, uint16 *pHandle);
//通过句柄找属性记录
gattAttribute_t *GATT_FindHandle(uint16 handle, uint16 *pHandle);
//找给定的属性相同类型的下一个属性
gattAttribute_t *GATT_FindNextAttr(gattAttribute_t *pAttr, uint16 endHandle, uint16
service, uint16 *pLastHandle);
//取得服务的属性数
uint16 GATT_ServiceNumAttrs(uint16 handle);
//发送 Indication
bStatus_t GATT_Indication(uint16 connHandle, attHandleValueInd_t *pInd, uint8
authenticated, uint8 taskId);
//发送 Notification
bStatus_t GATT_Notification(uint16 connHandle, attHandleValueNoti_t *pNoti, uint8
authenticated);
//客户端设置 ATT_MTU 最大值
bStatus_t GATT_ExchangeMTU(uint16 connHandle, attExchangeMTUReq_t *pReq, uint8 taskId);
//客户端用来发现服务器的所有主要服务
bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId);
//客户端通过 UUID 发现服务器的特定服务
bStatus_t GATT_DiscPrimaryServiceByUUID(uint16 connHandle, uint8 *pValue, uint8 len,
uint8 taskId);
//This sub-procedure is used by a client to find include service declarations within a
service definition on a server. The service specified is identified by the service handle
range.
bStatus_t GATT_FindIncludedServices(uint16 connHandle, uint16 startHandle, uint16
endHandle, uint8 taskId);
//找到所有特性
bStatus_t GATT_DiscAllChars(uint16 connHandle, uint16 startHandle, uint16 endHandle,
uint8 taskId);
//通过 UUID 找到特性
bStatus_t GATT_DiscCharsByUUID(uint16 connHandle, attReadByTypeReq_t *pReq, uint8
taskId);
//找到所有特性描述
bStatus_t GATT_DiscAllCharDescs(uint16 connHandle, uint16 startHandle, uint16 endHandle,
uint8 taskId);
//读取特性值
bStatus_t GATT_ReadCharValue(uint16 connHandle, attReadReq_t *pReq, uint8 taskId);
        bleTimeout: Previous transaction timed out.<BR>
//通过 UUID 读取特性值
bStatus_t GATT_ReadUsingCharUUID(uint16 connHandle, attReadByTypeReq_t *pReq, uint8
taskId);
//读取长特性值
bStatus_t GATT_ReadLongCharValue(uint16 connHandle, attReadBlobReq_t *pReq, uint8
taskId);
```

```
//读取多个特性值
bStatus_t GATT_ReadMultiCharValues(uint16 connHandle, attReadMultiReq_t *pReq, uint8
taskId);
//写特性值,不需要回应
bStatus_t GATT_WriteNoRsp(uint16 connHandle, attWriteReq_t *pReq);

bStatus_t GATT_SignedWriteNoRsp(uint16 connHandle, attWriteReq_t *pReq);
//写特性值
bStatus_t GATT_WriteCharValue(uint16 connHandle, attWriteReq_t *pReq, uint8 taskId);
//写长特性值
bStatus_t GATT_WriteLongCharValue(uint16 connHandle, gattPrepareWriteReq_t *pReq,
uint8 taskId);

bStatus_t GATT_ReliableWrites(uint16 connHandle, attPrepareWriteReq_t *pReqs, uint8
numReqs, uint8 flags, uint8 taskId);
//读取特性描述
bStatus_t GATT_ReadCharDesc(uint16 connHandle, attReadReq_t *pReq, uint8 taskId);
//读取长特性描述
bStatus_t GATT_ReadLongCharDesc(uint16 connHandle, attReadBlobReq_t *pReq, uint8
taskId);
//写特性描述
bStatus_t GATT_WriteCharDesc(uint16 connHandle, attWriteReq_t *pReq, uint8 taskId);
//写长特性描述
bStatus_t GATT_WriteLongCharDesc(uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8
taskId);
```

gap.h

```
//设备初始化
bStatus_t GAP_DeviceInit(  uint8 taskID,
                           uint8 profileRole,
                           uint8 maxScanResponses,
                           uint8 *pIRK,
                           uint8 *pSRK,
                           uint32 *pSignCounter );
//设置 GAP 广播搜索响应数据
bStatus_t GAP_SetAdvToken( gapAdvDataToken_t *pToken );
//读取 GAP 广播响应数据
gapAdvDataToken_t *GAP_GetAdvToken( uint8 adType );
//移除 GAP 广播响应数据
gapAdvDataToken_t *GAP_RemoveAdvToken( uint8 adType );
//重建加载广播响应数据
```

```
bStatus_t GAP_UpdateAdvTokens( void );
//设置 GAP 参数
bStatus_t GAP_SetParamValue( gapParamIDs_t paramID, uint16 paramValue );
//取得 GAP 参数
uint16 GAP_GetParamValue( gapParamIDs_t paramID );
//设置设备地址类型
bStatus_t GAP_ConfigDeviceAddr( uint8 addrType, uint8 *pStaticAddr );
//注册任务 IDvoid GAP_RegisterForHCIMsgs( uint8 taskID );
//开始搜索
bStatus_t GAP_DeviceDiscoveryRequest( gapDevDiscReq_t *pParams );
//取得发现任务
bStatus_t GAP_DeviceDiscoveryCancel( uint8 taskID );
//设置改变开始广播
bStatus_t GAP_MakeDiscoverable( uint8 taskID, gapAdvertisingParams_t *pParams );
//设置改变搜索响应数据
bStatus_t GAP_UpdateAdvertisingData( uint8 taskID, uint8 adType,
      uint8 dataLen, uint8 *pAdvertData );
//停止广播
bStatus_t GAP_EndDiscoverable( uint8 taskID );
//Resolves a private address against an IRK.
bStatus_t GAP_ResolvePrivateAddr( uint8 *pIRK, uint8 *pAddr );
//建立一个连接到从设备
bStatus_t GAP_EstablishLinkReq( gapEstLinkReq_t *pParams );
//中断连接
bStatus_t GAP_TerminateLinkReq( uint8 taskID, uint16 connectionHandle );
//更新连接参数到从设备
bStatus_t GAP_UpdateLinkParamReq( gapUpdateLinkParamReq_t *pParams );
//返回活跃连接数
uint8 GAP_NumActiveConnections( void );
//启动认证流程
bStatus_t GAP_Authenticate( gapAuthParams_t *pParams, gapPairingReq_t *pPairReq );
//发送配对失败消息
bStatus_t GAP_TerminateAuth( uint16 connectionHandle, uint8 reason );
//字符串格式的密钥更新
bStatus_t GAP_PasskeyUpdate( uint8 *pPasskey, uint16 connectionHandle );
//数字形式的密钥更新
bStatus_t GAP_PasscodeUpdate( uint32 passcode, uint16 connectionHandle );
//产生一个从机请求的安全消息到主机
bStatus_t GAP_SendSlaveSecurityRequest( uint16 connectionHandle, uint8 authReq );
//Set up the connection to accept signed data.
bStatus_t GAP_Signable( uint16 connectionHandle, uint8 authenticated, smSigningInfo_t
*pParams );
//设置连接的绑定参数
```

```
bStatus_t GAP_Bond( uint16 connectionHandle, uint8 authenticated,
      smSecurityInfo_t *pParams, uint8 startEncryption );
```

att.h

```
//解析 ATT 包
uint8 ATT_ParsePacket(l2capDataEvent_t *pL2capMsg, attPacket_t *pPkt);
//比较 UUID
uint8 ATT_CompareUUID(const uint8 *pUUID1, uint16 len1, const uint8 *pUUID2, uint16
len2);
//转换 16bit 的 UUID 到 128bit
uint8 ATT_ConvertUUIDto128(const uint8 *pUUID16, uint8 *pUUID128);
//转换 128bit 的 UUID 到 16bit
uint8 ATT_ConvertUUIDto16(const uint8 *pUUID128, uint8 *pUUID16);
//构建错误响应
uint16 ATT_BuildErrorRsp(uint8 *pBuf, uint8 *pMsg);
//解析错误响应
bStatus_t ATT_ParseErrorRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建交换 MTU 请求
uint16 ATT_BuildExchangeMTUReq(uint8 *pBuf, uint8 *pMsg);
//构建交换 MTU 响应
uint16 ATT_BuildExchangeMTURsp(uint8 *pBuf, uint8 *pMsg);
//解析 MTU 响应
bStatus_t ATT_ParseExchangeMTURsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建找信息请求
uint16 ATT_BuildFindInfoReq(uint8 *pBuf, uint8 *pMsg);
//解析找信息响应
bStatus_t ATT_ParseFindInfoRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建找信息响应
uint16 ATT_BuildFindInfoRsp(uint8 *pBuf, uint8 *pMsg);
//构建通过类型值找请求
uint16 ATT_BuildFindByTypeValueReq(uint8 *pBuf, uint8 *pMsg);
//构建通过类型值找响应
uint16 ATT_BuildFindByTypeValueRsp(uint8 *pBuf, uint8 *pMsg);
//解析通过类型值找响应
bStatus_t ATT_ParseFindByTypeValueRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建通过类型值读请求
uint16 ATT_BuildReadByTypeReq(uint8 *pBuf, uint8 *pMsg);
//构建通过类型值读响应
uint16 ATT_BuildReadByTypeRsp(uint8 *pBuf, uint8 *pMsg);
//解析通过类型值读响应
```

```
bStatus_t ATT_ParseReadByTypeRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建读请求
uint16 ATT_BuildReadReq(uint8 *pBuf, uint8 *pMsg);
//构建读响应
uint16 ATT_BuildReadRsp(uint8 *pBuf, uint8 *pMsg);
//解析读响应
bStatus_t ATT_ParseReadRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建场数据读取请求
uint16 ATT_BuildReadBlobReq(uint8 *pBuf, uint8 *pMsg);
//构建长数据读取响应
uint16 ATT_BuildReadBlobRsp(uint8 *pBuf, uint8 *pMsg);
//解析大数据读取响应
bStatus_t ATT_ParseReadBlobRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//构建多数据读取请求
uint16 ATT_BuildReadMultiReq(uint8 *pBuf, uint8 *pMsg);
//构建多数据读取响应
uint16 ATT_BuildReadMultiRsp(uint8 *pBuf, uint8 *pMsg);
//解析多数据读取响应
bStatus_t ATT_ParseReadMultiRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//Build Read By Group Type Response.
uint16 ATT_BuildReadByGrpTypeRsp(uint8 *pBuf, uint8 *pMsg);
// Parse Read By Group Type Response.
bStatus_t ATT_ParseReadByGrpTypeRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
// Build Write Request.
uint16 ATT_BuildWriteReq(uint8 *pBuf, uint8 *pMsg);
//Parse Write Response.
bStatus_t ATT_ParseWriteRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
// Build Prepare Write Request.
uint16 ATT_BuildPrepareWriteReq(uint8 *pBuf, uint8 *pMsg);
//Build Prepare Write Response.
uint16 ATT_BuildPrepareWriteRsp(uint8 *pBuf, uint8 *pMsg);
//Parse Prepare Write Response.
bStatus_t ATT_ParsePrepareWriteRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
// Build Execute Write Request.
uint16 ATT_BuildExecuteWriteReq(uint8 *pBuf, uint8 *pMsg);
// Parse Execute Write Response.
bStatus_t ATT_ParseExecuteWriteRsp(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//Build Handle Value Indication.
uint16 ATT_BuildHandleValueInd(uint8 *pBuf, uint8 *pMsg);
//Parse Handle Value Indication.
bStatus_t ATT_ParseHandleValueInd(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
// Parse Exchange MTU Request.
bStatus_t ATT_ParseExchangeMTUReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
```

```
attMsg_t *pMsg);
//Parse Find Information Request.
bStatus_t ATT_ParseFindInfoReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
// Parse Find By Type Value Request.
bStatus_t ATT_ParseFindByTypeValueReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
// Parse Read By Type Request.
bStatus_t ATT_ParseReadByTypeReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
//Parse Read Request.
bStatus_t ATT_ParseReadReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len, attMsg_t
*pMsg);
//Parse Write Blob Request.
bStatus_t ATT_ParseReadBlobReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
//Parse Read Multiple Request.
bStatus_t ATT_ParseReadMultiReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
//Parse Write Request.
bStatus_t ATT_ParseWriteReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len, attMsg_t
*pMsg);
//Parse Execute Write Request.
bStatus_t ATT_ParseExecuteWriteReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
//Parse Prepare Write Request.
bStatus_t ATT_ParsePrepareWriteReq(uint8 sig, uint8 cmd, uint8 *pParams, uint16 len,
attMsg_t *pMsg);
// Parse Handle Value Confirmation.
bStatus_t ATT_ParseHandleValueCfm(uint8 *pParams, uint16 len, attMsg_t *pMsg);
//发送交换 MTU 请求
bStatus_t ATT_ExchangeMTUReq(uint16 connHandle, attExchangeMTUReq_t *pReq);
//Send Find Information Request.
bStatus_t ATT_FindInfoReq(uint16 connHandle, attFindInfoReq_t *pReq);
// Send Find By Type Value Request.
bStatus_t ATT_FindByTypeValueReq(uint16 connHandle, attFindByTypeValueReq_t *pReq);
//Send Read By Type Request.
bStatus_t ATT_ReadByTypeReq(uint16 connHandle, attReadByTypeReq_t *pReq);
// Send Read Request.
bStatus_t ATT_ReadReq(uint16 connHandle, attReadReq_t *pReq);
// Send Read Blob Request.
bStatus_t ATT_ReadBlobReq(uint16 connHandle, attReadBlobReq_t *pReq);
// Send Read Multiple Request.
bStatus_t ATT_ReadMultiReq(uint16 connHandle, attReadMultiReq_t *pReq);
```

```
// Send Read By Group Type Request.
bStatus_t ATT_ReadByGrpTypeReq(uint16 connHandle, attReadByGrpTypeReq_t *pReq);
// Send Write Request.
bStatus_t ATT_WriteReq(uint16 connHandle, attWriteReq_t *pReq);
// Send Prepare Write Request.
bStatus_t ATT_PrepareWriteReq(uint16 connHandle, attPrepareWriteReq_t *pReq);
// Send Execute Write Request.
bStatus_t ATT_ExecuteWriteReq(uint16 connHandle, attExecuteWriteReq_t *pReq);
// Send Handle Value Confirmation.
bStatus_t ATT_HandleValueCfm(uint16 connHandle);
// Send Error Response.
bStatus_t ATT_ErrorRsp(uint16 connHandle, attErrorRsp_t *pRsp);
//Send Exchange MTU Response.
bStatus_t ATT_ExchangeMTURsp(uint16 connHandle, attExchangeMTURsp_t *pRsp);
// Send Find Information Response.
bStatus_t ATT_FindInfoRsp(uint16 connHandle, attFindInfoRsp_t *pRsp);
//Send Find By Tyep Value Response.
bStatus_t ATT_FindByTypeValueRsp(uint16 connHandle, attFindByTypeValueRsp_t *pRsp);
//Send Read By Type Respond.
bStatus_t ATT_ReadByTypeRsp(uint16 connHandle, attReadByTypeRsp_t *pRsp);
//Send Read Response.
bStatus_t ATT_ReadRsp(uint16 connHandle, attReadRsp_t *pRsp);
// Send Read Blob Response.
bStatus_t ATT_ReadBlobRsp(uint16 connHandle, attReadBlobRsp_t *pRsp);
//Send Read Multiple Response.
bStatus_t ATT_ReadMultiRsp(uint16 connHandle, attReadMultiRsp_t *pRsp);
// Send Read By Group Type Respond.
bStatus_t ATT_ReadByGrpTypeRsp(uint16 connHandle, attReadByGrpTypeRsp_t *pRsp);
//Send Write Response.
bStatus_t ATT_WriteRsp(uint16 connHandle);
// Send Prepare Write Response.
bStatus_t ATT_PrepareWriteRsp(uint16 connHandle, attPrepareWriteRsp_t *pRsp);
// Send Execute Write Response.
bStatus_t ATT_ExecuteWriteRsp(uint16 connHandle);
// Send Handle Value Notification.
bStatus_t ATT_HandleValueNoti(uint16 connHandle, attHandleValueNoti_t *pNoti);
// Send Handle Value Indication.
bStatus_t ATT_HandleValueInd(uint16 connHandle, attHandleValueInd_t *pInd);
//设置 ATT 参数 void ATT_SetParamValue(uint16 value);
//取得 ATT 参数
uint16 ATT_GetParamValue(void);
```