



Telink

Telink B85m BLE Single Connection

SDK Development Handbook

AN-21112300-E2

Ver1.0.1

2022.04.21

Keyword

BLE 5.0

Brief

This document is Telink B85 BLE Single Connection SDK development guide, suitable for B85m series chips.

Published by
Telink Semiconductor

Bldg 3, 1500 Zuchongzhi Rd,
Zhangjiang Hi-Tech Park, Shanghai, China

© Telink Semiconductor
All Rights Reserved

Legal Disclaimer

This document is provided as-is. Telink Semiconductor reserves the right to make improvements without further notice to this document or any products herein. This document may contain technical inaccuracies or typographical errors. Telink Semiconductor disclaims any and all liability for any errors, inaccuracies or incompleteness contained herein.

Copyright © 2022 Telink Semiconductor (Shanghai) Co., Ltd.

Information

For further information on the technology, product and business term, please contact Telink Semiconductor Company www.telink-semi.com

For sales or technical support, please send email to the address of:

telinksales@telink-semi.com

telinksupport@telink-semi.com

Revision History

Version	Change Description
---------	--------------------

V1.0.0	Initial release
--------	-----------------

V1.0.1	Corrected the SRAM size of 8273 in section 1.2; Corrected the OTA mark offset and the modification action when receiving OTA_END in section 7.1.2
--------	---

Telink Semiconductor

Contents

Revision History	3
1 SDK Overview	16
1.1 Software architecture	16
1.1.1 main.c	17
1.1.2 app_config.h	18
1.1.3 application file	18
1.1.4 BLE stack entry	18
1.2 Applicable IC	19
1.3 Software Bootloader	19
1.4 Demo codes	22
1.4.1 BLE Slave Demo	23
1.4.2 BLE master demo	24
1.4.3 Feature Demo and driver demo	24
2 MCU Basic Modules	26
2.1 MCU Address Space	26
2.1.1 MCU Address Space Allocation	26
2.1.2 SRAM Space Allocation	27
2.1.2.1 SRAM and Firmware Space	27
2.1.2.2 list file analysis demo	33
2.1.3 MCU Address Space Access	36
2.1.3.1 Peripheral Space R/W Operation	36
2.1.3.2 Flash operation	36
2.1.4 SDK Flash space allocation	37
2.2 Clock Module	37
2.2.1 System clock & System Timer	37
2.2.2 System Timer Usage	39
2.3 GPIO Module	40
2.3.1 GPIO definition	40
2.3.2 GPIO state control	41
2.3.3 GPIO initialization	43
2.3.4 GPIO digital states fail in deepsleep retention mode	45
2.3.5 Configure SWS pull-ups to prevent crashes	45
2.4 System interrupt	46
3 BLE Module	48
3.1 BLE SDK Software Architecture	48
3.1.1 Standard BLE SDK Architecture	48
3.1.2 Telink BLE SDK Architecture	49
3.1.2.1 Telink BLE controller	49
3.1.2.2 Telink BLE Slave	50
3.1.2.3 Telink BLE master	52
3.2 BLE Controller	53
3.2.1 BLE Controller Introduction	53
3.2.2 Link Layer State Machine	53

3.2.3	Link Layer State Machine Combined Application	56
3.2.3.1	Link Layer State Machine Initialization	56
3.2.3.2	Idle + Advertising	57
3.2.3.3	Idle + Scanning	57
3.2.3.4	Idle + Advertising + ConnSlaveRole	59
3.2.3.5	Idle + Scanning + Initiating + ConnMasterRole	61
3.2.4	Link Layer Timing Sequence	62
3.2.4.1	Timing Sequence in Idle State	63
3.2.4.2	Timing Sequence in Advertising State	63
3.2.4.3	Timing Sequence in Scanning State	64
3.2.4.4	Timing Sequence in Initiating State	64
3.2.4.5	Timing Sequence in Conn State Slave Role	65
3.2.4.6	Timing Sequence in Conn State Master Role	66
3.2.4.7	Timing Protect for Conn State Slave role	67
3.2.5	Link Layer State Machine Extension	68
3.2.5.1	Scanning in Advertising state	69
3.2.5.2	Scanning in ConnSlaveRole	69
3.2.5.3	Advertising in ConnSlaveRole	70
3.2.5.4	Advertising and Scanning in ConnSlaveRole	71
3.2.6	Link Layer TX fifo & RX fifo	72
3.2.7	Controller Event	75
3.2.7.1	Controller HCI Event	76
3.2.7.2	HCI event	78
3.2.7.3	HCI LE event	79
3.2.7.4	Telink Defined Event	81
3.2.8	Data Length Extension	90
3.2.9	Controller API	92
3.2.9.1	Controller API Introduction	92
3.2.9.2	API Return Type ble_sts_t	92
3.2.9.3	BLE MAC address initialization	92
3.2.9.4	Link Layer state machine initialization	93
3.2.9.5	bls_ll_setAdvData	93
3.2.9.6	bls_ll_setScanRspData	94
3.2.9.7	bls_ll_setAdvParam	95
3.2.9.8	bls_ll_setAdvEnable	99
3.2.9.9	bls_ll_setAdvDuration	99
3.2.9.10	blc_ll_setAdvCustomedChannel	100
3.2.9.11	rf_set_power_level_index	100
3.2.9.12	blc_ll_setScanParameter	101
3.2.9.13	blc_ll_setScanEnable	102
3.2.9.14	blc_ll_createConnection	103
3.2.9.15	blc_ll_setCreateConnectionTimeout	105
3.2.9.16	blm_ll_updateConnection	105
3.2.9.17	bls_ll_terminateConnection	106
3.2.9.18	blm_ll_disconnect	106
3.2.9.19	Get Connection Parameters	107
3.2.9.20	blc_ll_getCurrentState	107

3.2.9.21	blc_ll_getLatestAvgRSSI	108
3.2.9.22	Whitelist & Resolvinglist	108
3.2.9.23	blc_att_setServerDataPendingTime_upon_ClientCmd	109
3.2.10	Coded PHY/2M PHY	110
3.2.10.1	Coded PHY/2M PHY Introduction	110
3.2.10.2	Coded PHY/2M PHY Demo Introduction	110
3.2.10.3	Coded PHY/2M PHY API Introduction	110
3.2.11	Channel Selection Algorithm #2	111
3.2.12	Extended Advertising	112
3.2.12.1	Extended Advertising Introdcuton	112
3.2.12.2	Extended Advertising Demo Setup	112
3.2.12.3	Extended Advertising Related API	113
3.3	BLE Host	117
3.3.1	BLE Host Introduction	117
3.3.2	L2CAP	117
3.3.2.1	Register L2CAP Data Processing Function	118
3.3.2.2	Update connection parameters	119
3.3.3	ATT & GATT	123
3.3.3.1	GATT basic unit "Attribute"	123
3.3.3.2	Attribute and ATT Table	125
3.3.3.3	Attribute PDU and GATT API	132
3.3.3.4	GATT Service Security	144
3.3.3.5	B85m master GATT	146
3.3.4	SMP	148
3.3.4.1	SMP Security Level	148
3.3.4.2	SMP Parameter Configuration	149
3.3.4.3	Security Request Configuration	155
3.3.4.4	SMP Bonding info	158
3.3.4.5	master SMP	161
3.3.4.6	SMP Failure Management	167
3.3.5	GAP	167
3.3.5.1	GAP initialization	167
3.3.5.2	GAP Event	168
4	Low Power Management	175
4.1	Low Power Driver	175
4.1.1	Low Power Mode	175
4.1.2	Low Power Wake-up Source	177
4.1.3	Sleep and Wake-up from Low Power Mode	179
4.1.4	Low Power Wake-up Procedure	181
4.1.5	API pm_is_MCU_deepRetentionWakeup	184
4.2	BLE Low Power Management	184
4.2.1	BLE PM Initialization	184
4.2.2	BLE PM for Link Layer	184
4.2.3	BLE PM Variables	186
4.2.4	API bls_pm_setSuspendMask	187
4.2.5	API bls_pm_setWakeupSource	188
4.2.6	API blc_pm_setDeepsleepRetentionType	189

4.2.7	PM software processing flow	190
4.2.7.1	blt_sdk_main_loop	190
4.2.7.2	blt_brx_sleep	191
4.2.8	Analysis of deepsleep retention	193
4.2.8.1	API blc_pm_setDeepsleepRetentionThreshold	193
4.2.8.2	blc_pm_setDeepsleepRetentionEarlyWakeupTiming	197
4.2.8.3	Optimization and measurement of T_init	197
4.2.9	Connection Latency	202
4.2.9.1	Sleep timing with non-zero connection latency	202
4.2.9.2	latency_use calculation	203
4.2.10	API bls_pm_getSystemWakeupTick	204
4.3	Issues in GPIO Wake-up	205
4.3.1	Fail to enter sleep mode when wake-up level is valid	205
4.4	BLE System Low Power Management	206
4.5	Timer Wake-up by Application Layer	207
5	Low Battery Detect	209
5.1	The importance of low battery detect	209
5.2	The implementation of low battery detect	209
5.2.1	Notes on low battery detect	210
5.2.1.1	GPIO input channel recommended	210
5.2.1.2	Differential mode only	211
5.2.1.3	Must use Dfifo mode to obtain ADC sampling value	212
5.2.1.4	Need to switch different ADC tasks	212
5.2.2	Stand-alone use of low battery detect	212
5.2.2.1	Low battery detect initialization	212
5.2.2.2	Low battery detect processing	214
5.2.2.3	Low voltage alarm	216
5.2.2.4	Low power detect debug mode	217
5.2.3	Low battery detect and Amic Audio	217
6	Audio	219
6.1	Initialization	219
6.1.1	AMIC and Low Power Detect	219
6.1.2	AMIC Initialization	219
6.1.3	DMIC Initialization	220
6.2	Audio Data Processing	220
6.2.1	Audio Data Volume and RF Transfer	220
6.2.2	Audio Data Compression	222
6.3	Compression and Decompression Algorithm	224
6.4	Audio data processing flow	226
6.4.1	TL_AUDIO_RCU_ADPCM_GATT_GOOGLE	228
6.4.1.1	Initialization	229
6.4.1.2	Voice data transmission	230
6.4.1.3	TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB	231
6.4.2	TL_AUDIO_RCU_SBC_HID_DONGLE_TO_STB	233
7	OTA	236
7.1	Flash Architecture and OTA Procedure	236
7.1.1	FLASH Storage Architecture	236

7.1.2	OTA Update Procedure	237
7.1.3	Modify FW Size and Booting Address	239
7.2	RF Data Processing for OTA Mode	239
7.2.1	OTA Processing in Attribute Table	239
7.2.2	OTA Protocol	240
7.2.3	RF Transfer Processing on Master Side	247
7.3	OTA Security	257
7.3.1	OTA Service data security	257
7.3.2	OTA RF transmission data integrity	258
7.3.2.1	LinkLayer data transfer mechanism	258
7.3.2.2	OTA PDU CRC16 check	258
7.3.2.3	OTA PDU serial number check	258
7.3.3	Firmware CRC32 check	259
7.3.4	OTA abnormal power failure protection	259
8	Flash	260
8.1	Flash address allocation	260
8.2	Flash operation	263
8.3	Flash operation protection	266
8.3.1	Low voltage detection protection	266
8.3.2	Flash lock protection	268
8.3.2.1	Initialize write protection	268
8.3.2.2	Protection operations in the OTA process	269
8.4	Internal Flash introduction	270
8.4.1	Impact of Flash access timing on BLE timing	270
8.4.1.1	Flash access timing	270
8.4.1.2	Impact of Flash API on BLE timing	273
8.4.2	Use of internal Flash API	275
8.4.2.1	GD Flash	275
8.4.2.2	Zbit Flash	276
8.4.2.3	PUYA Flash	276
9	Key Scan	279
9.1	Key Matrix	279
9.2	Keyscan and Keymap	281
9.2.1	Keyscan	281
9.2.2	Keymap & kb_event	282
9.3	Keyscan Flow	283
9.4	Deepsleep wake_up fast keyscan	285
9.5	Repeat Key Processing	286
9.6	Stuck Key Processing	287
10	LED Management	290
10.1	LED task related functions	290
10.2	LED Task Configuration and Management	290
10.2.1	LED Event Definition	290
10.2.2	LED Event Priority	291
11	Software Timer	293
11.1	Timer Initialization	293
11.2	Timer Inquiry Processing	293

11.3	Add Timer Task	295
11.4	Delete Timer Task	296
11.5	Demo	297
12	IR	299
12.1	PWM Driver	299
12.1.1	PWM ID and Pin	299
12.1.2	PWM Clock	300
12.1.3	PWM Cycle and Duty	301
12.1.4	PWM Revert	302
12.1.5	PWM Start and Stop	302
12.1.6	PWM Mode	302
12.1.7	PWM Pulse Number	303
12.1.8	PWM Interrupt	303
12.1.9	PWM phase	305
12.1.10	IR DMA FIFO mode	305
12.1.10.1	Configuration for DMA FIFO	305
12.1.10.2	Set DMA FIFO Buffer	306
12.1.10.3	Start and Stop for IR DMA FIFO Mode	306
12.2	IR Demo	307
12.2.1	PWM mode selection	307
12.2.2	Demo IR Protocol	307
12.2.3	IR Timing Design	308
12.2.4	IR Initialization	311
12.2.4.1	rc_ir_init	311
12.2.4.2	IR Hardware Configuration	311
12.2.4.3	IR Variable Initialization	311
12.2.5	FifoTask Configuration	312
12.2.5.1	FifoTask_data	312
12.2.5.2	FifoTask_idle	313
12.2.5.3	FifoTask_repeat	313
12.2.5.4	FifoTask_repeat*n and FifoTask_idle_repeat*n	314
12.2.6	Check IR Busy Status in APP Layer	314
12.3	IR Learn	315
12.3.1	IR Learn introduction	315
12.3.2	IR Learn hardware principle	315
12.3.3	IR Learn software principle	316
12.3.3.1	IR_Learn initialization	317
12.3.3.2	IR_Learn interrupt handling	318
12.3.3.3	IR_Learn result processing function	318
12.3.3.4	IR_Learn macro definition	318
12.3.3.5	IR_Learn start function	319
12.3.3.6	IR_Learn state query	319
12.3.3.7	IR_Learn_Send initialization	319
12.3.3.8	IR_Learn result copy function	320
12.3.3.9	IR_Learn send function	320
12.3.4	IR Learn algorithm details	320
12.3.5	IR Learn learning parameter adjustment	322

12.3.6 IR Learn common issues	324
12.4 Demo description	325
13 Feature Demo Introduction	326
13.1 Broadcast Power Consumption Test	326
13.1.1 Connectable Broadcast Power Consumption Test	327
13.1.2 Un-connectable Broadcast Power Consumption Test	327
13.2 SMP Test	328
13.2.1 LE_Security_Mode_1_Level_1	328
13.2.2 LE_Security_Mode_1_Level_2	328
13.2.2.1 SMP_TEST_LEGACY_PAIRING_JUST_WORKS	328
13.2.2.2 SMP_TEST_SC_PAIRING_JUST_WORKS	329
13.2.3 LE_Security_Mode_1_Level_3	330
13.2.3.1 SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI	330
13.2.3.2 SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI	332
13.2.4 LE_Security_Mode_1_Level_4	333
13.2.4.1 SMP_TEST_SC_NUMERIC_COMPARISON	334
13.2.4.2 SMP_TEST_SC_PASSKEY_ENTRY_SDMI	335
13.3 GATT Security Test	337
13.4 DLE Test	339
13.5 Soft Timer Test	340
13.6 WhiteList Test	341
13.7 1M Extended Advertising Test	342
13.8 2M/Coded PHY Used on Extended Advertising Test	342
13.9 2M/Coded PHY used on Legacy advertising and Connection Test	344
13.10CSA #2 Test	345
13.11EMI Test	346
13.11.1 Protocol	346
13.11.2 Demo introduction	346
14 Other Modules	347
14.1 24MHz Crystal External Capacitor	347
14.2 32KHz Clock Source Selection	348
14.3 Firmware Digital Signature	348
14.4 Firmware Integrity Self-check	350
15 Debug	351
15.1 Introduction to GPIO simulation UART_TX printing method	351
16 Q&A	352
17 Appendix	356
17.1 crc16 Algorithm	356

List of Figures

1.1	"SDK File Structure"	16
1.2	"Bootloader and boot.link path for different IC"	20
1.3	"software bootloader setting"	21
1.4	"BLE SDK demo code"	23
2.1	"MCU Address Space Allocation"	26
2.2	"SRAM space allocation for each IC at 16k and 32k retention"	27
2.3	"SRAM space allocation & Firmware space allocation"	28
2.4	"list file section analysis"	33
2.5	"list file section address"	34
2.6	"System clock & System Timer"	37
2.7	"IRQ delay"	47
3.1	"BLE SDK software architecture"	48
3.2	"HCI Data Transfer between Host and Controller"	49
3.3	"Telink HCI architecture"	50
3.4	"Telink BLE Slave architecture"	51
3.5	"Telink BLE master architecture"	52
3.6	"Link Layer State Machine in BLE Spec"	54
3.7	"Telink Link Layer State Machine"	55
3.8	"Idle + Advertising"	57
3.9	"Idle + Scanning"	58
3.10	"BLE Slave LL State"	59
3.11	"BLE Master LL State"	61
3.12	"Timing Sequence in Advertising State"	63
3.13	"Timing Sequence in Scanning State"	64
3.14	"Timing Sequence in Initiating State"	64
3.15	"Timing Sequence in Conn State Slave Role"	65
3.16	"Timing Sequence in Conn Master Role"	66
3.17	"Timing of Scanning in Advertising state"	68
3.18	"Timing of Scanning in Advertising state"	69
3.19	"Timing of Scanning in ConnSlaveRole"	70
3.20	"Timing of Advertising in ConnSlaveRole"	71
3.21	"Timing of Advertising and Scanning in ConnSlaveRole"	71
3.22	"RX overflow case 1"	73
3.23	"RX overflow case 2"	74
3.24	"BLE SDK Event Architecture"	76
3.25	"HCI Event"	77
3.26	"Disconnection Complete Event"	78
3.27	"Read Remote Version Information Complete Event"	78
3.28	"LE Connection Complete Event"	79
3.29	"LE Advertising Report Event"	80
3.30	"LE Connection Update Complete Event"	80
3.31	"Connect Request PDU"	85
3.32	"LL_CONNECTION_UPDATE REQ Format in BLE Stack"	89
3.33	"Adv Packet Format in BLE Stack"	93

3.34	"Advertising Event in BLE Stack"	95
3.35	"Four Adv Events in BLE Stack"	96
3.36	"Error in compiling a demo"	113
3.37	"Extended Advertising Initialize Memory Allocation"	114
3.38	"BLE L2CAP Structure and ATT Packet Assembly Model"	117
3.39	"Connection Para Update Req Format in BLE Stack"	119
3.40	"BLE Sniffer Packet Sample Conn Para Update Request and Response"	119
3.41	"Conn Para Update RSP Format in BLE Stack"	121
3.42	"Demo code of b85m master kma dongle"	122
3.43	"Demo code of b85m master kma dongle"	123
3.44	"BLE Sniffer Packet Sample II Conn Update Req"	123
3.45	"GATT Service Containing Attributes"	124
3.46	"BLE Sniffer Packet Sample when Master Reads hidInformation"	128
3.47	"Write Request in BLE Stack"	129
3.48	"Write Command in BLE Stack"	130
3.49	"Execute Write Request in BLE Stack"	130
3.50	"Service Attribute Layout"	132
3.51	"Read by Group Type Request Read by Group Type Response"	133
3.52	"Find by Type Value Request Find by Type Value Response"	134
3.53	"Read by Type Value Request Find by Type Value Response"	135
3.54	"Find Information Request Find Information Response"	136
3.55	"Read Request Read Response"	136
3.56	"Read Blob Request Read Blob Response"	137
3.57	"Exchange MTU Request Exchange MTU Response"	137
3.58	"Write Request Write Response"	139
3.59	"Example for Write Long Characteristic Values"	140
3.60	"Handle Value Notification in BLE Spec"	140
3.61	"Handle Value Indication in BLE Spec"	142
3.62	"Handle Value Confirmation in BLE Spec"	143
3.63	"Mapping Diagram for Service Request and Response"	144
3.64	"ATT Permission Definition"	145
3.65	"Local Device Pairing Status"	148
3.66	"Packet Example for Pairing Disable"	149
3.67	"Usage Rule for MITM OOB Flag in Legacy Pairing Mode"	152
3.68	"Mapping Relationship for KEY Generation Method and IO Capability"	152
3.69	"Packet Example for Pairing Peer Trigger"	157
3.70	"Packet Example for Pairing Conn Trigger"	157
3.71	"master initiates Pairing_Req"	169
4.1	"B85 MCU HW Wakeup Source"	178
4.2	"Sleep Mode Wakeup Work Flow"	182
4.3	"Sleep Timing for Advertising State and Conn State Slave Role"	185
4.4	"Suspend Deep sleep Retention Timing Power"	195
4.5	"T_init Timing"	198
4.6	"Sleep Timing for Valid Conn_latency"	203
4.7	"Low Power Code"	206
4.8	"EarlyWake_upatapp_wakup_tick"	208
6.1	"Audio Data Sample"	221

6.2	"MIC Service in Attribute Table"	222
6.3	"Data Compression Processing"	224
6.4	"Data Corresponding to Compression Algorithm"	225
6.5	"Corresponding library files"	227
6.6	"SBC mode setting method"	228
6.7	"Google Service UUID setting"	229
6.8	"Google Voice initialization flow"	229
6.9	"Packet Interaction Information"	229
6.10	"Audio Data Transmission"	230
6.11	"Search_KEY packet"	230
6.12	"Search packet"	230
6.13	"MIC_Open packet"	231
6.14	"Start packet"	231
6.15	"134-byte Audio frame"	231
6.16	"Audio data interaction in ADPCM_HID_DONGLE_TO_STB mode"	232
6.17	"Start_request packet"	232
6.18	"Ack packet"	232
6.19	"Audio voice data"	233
6.20	"End request packet"	233
6.21	"Ack packet"	233
6.22	"Audio data interaction in SBC_HID_DONGLE_TO_STB mode"	234
6.23	"Start_request packet"	234
6.24	"Ack packet"	234
6.25	"Audio voice data of sbc decode"	234
6.26	"End request packet"	235
6.27	"Ack packet"	235
7.1	"Flash Storage Structure"	236
7.2	"OTA packet in L2CAP PDU"	245
7.3	"PDU length 32"	247
7.4	"PDU length 48"	247
7.5	"PDU length 80"	247
7.6	"OTA Legacy protocol process"	248
7.7	"OTA Extend protocol process"	249
7.8	"OTA Version Compare Process"	250
7.9	"Master Obtains OTA Attribute Handle via Read by Type Request"	251
7.10	"Firmware Sample Starting Part"	252
7.11	"Firmware Sample Ending Part"	252
7.12	"OTA Start Sent From Master"	253
7.13	"Master OTA Data1"	254
7.14	"Master OTA Data2"	254
7.15	"Slave Sends OTA Succuss Result to Master"	256
8.1	"512K/1M FLASH address allocation"	261
8.2	"Write Protection by Flash Type"	269
8.3	"Flash Operation Basic Timing"	270
8.4	"Flash Timing Conflicts Caused by Interrupts"	271
8.5	"Proper interrupt handling and flash operation"	272
8.6	"Flash Operation on Link Layer Risk"	273

8.7	"Sonos Process Flash Action Before and After The Reception Window"	277
8.8	"Flash write action delay due to more data"	278
9.1	"Row Column Key Matrix"	279
9.2	"Repeat Key Application Example"	287
12.1	"PWM cycle & duty"	301
12.2	"PWM interrupt"	304
12.3	"Demo IR Protocol"	308
12.4	"IR Timing 1"	308
12.5	"IR Timing 2"	309
12.6	"IR Learn hardware circuit"	315
12.7	"IR_IN waveform of NEC protocol"	316
12.8	"IR_IN waveform of NEC carrier"	316
12.9	"Carrier and non-carrier"	317
12.10	"A frame of IR code"	321
12.11	"Carrier and no carrier in IR Learn"	321
12.12	"IR learn algorithm"	322
12.13	"IR learn error"	324
13.1	"Feature Test Demo"	326
13.2	"Legacy Just Work Process"	329
13.3	"SC Just Work Process"	330
13.4	"Legacy Just Work SDMI Process"	332
13.5	"Legacy Just Work SIMD Process"	333
13.6	"Numeric Comparison Paring"	335
13.7	"SC SDMI Paring Processing"	337
13.8	"Gatt Security"	339
13.9	"DLE Test Process"	340
13.10	"Whitelist Test Process"	341
13.11	"PHY change flowchart"	344
14.1	"24M Crystal Schematic"	347
16.1	"Error in compiling a SDK project"	352
16.2	"Enter a new name for a project"	352
16.3	"Create new configuration for a project"	353
16.4	"New project in the project list"	353
16.5	"Exclude Test_Demo from build"	354
16.6	"Exclude source project from build"	354
16.7	"Modify compiler symbol"	355
16.8	"Add user config for new code"	355

List of Tables

1.1	IC and memory supported by BLE B85m SDK	19
1.2	Boot file selection for BLE B85m SDK	22
1.3	Demo overview of BLE slave	23
3.13	Input parameter combination of blc_smp_configSecurityRequestSending	156
4.1	Sleep mode description	175
7.1	Firmware size and boot address	239
7.2	OTA protocol	240
7.3	PDU of OTA's CMD	241
7.4	Opcode of CMD	241
7.5	End command of OTA	242
7.6	Packet structure of OTA_START_EXT	242
7.7	Packet structure of OTA_FW_VERSION	242
7.8	Response structure of OTA_FW_VERSION	243
7.9	OTA result return command structure	243
7.10	OTA return results	243
7.11	OTA data	245
7.12	OTA PDU format	245
7.13	Mapping of Adr_Index to firmware address when n=1	246
7.14	Mapping of Adr_Index to firmware address when n=2	246
7.15	Mapping of Adr_Index to firmware address when n=15	246
12.1	PWM pin allocation	299

1 SDK Overview

This BLE SDK supplies demo code for BLE slave/master single connection development, based on which user can develop his own application program.

1.1 Software architecture

Software architecture for this BLE SDK includes application (APP) layer and BLE protocol stack.

Figure below shows the file structure after the SDK project is imported in IDE, which mainly contains 8 top-layer folders below: "algorithm", "application", "boot", "common", "drivers", "proj_lib", "stack" and "vendor".

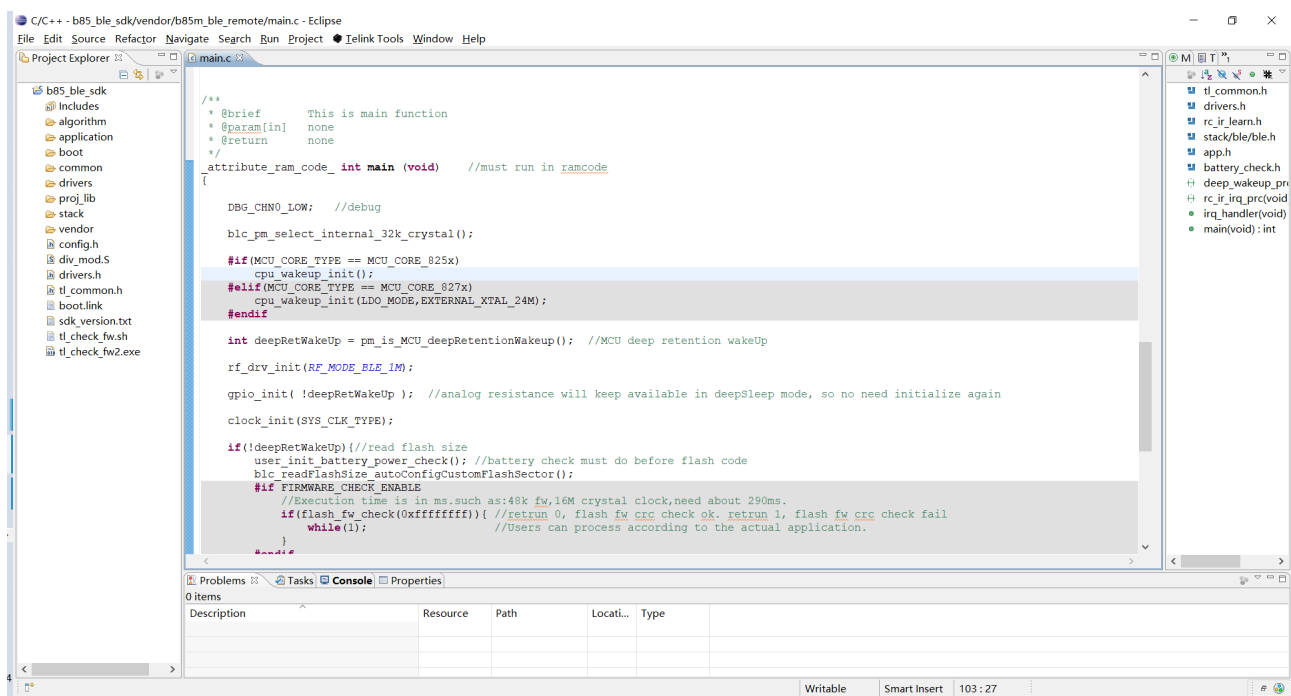


Figure 1.1: "SDK File Structure"

- **Algorithm:** This folder contains functions related to encryption algorithms.
- **Application:** This folder contains general application program, e.g. print, keyboard, audio, and etc.
- **boot:** This folder contains software bootloader for chip, i.e., assembly code after MCU power on or deepsleep wakeup, so as to establish environment for C program running.
- **common:** This folder contains generic handling functions across platforms, e.g. SRAM handling function, string handling function, and etc.
- **drivers:** This folder contains hardware configuration and peripheral drivers closely related to MCU, e.g. clock, flash, i2c, usb, gpio, uart.
- **proj_lib:** This folder contains library files necessary for SDK running, e.g. BLE stack, RF driver, PM driver. Since this folder is supplied in the form of library files (e.g. libtl_825x.a), the source files are not open to users.

- stack: This folder contains header files for BLE stack. Source files supplied in the form of library files are not open to users.
- vendor: This folder contains user application-layer code.

1.1.1 main.c

The “main.c” file includes main function entry, system initialization functions and endless loop “while(1)”. It’s not recommended to make any modification to this file.

```
_attribute_ram_code_ int main (void)    //must run in ramcode
{

    DBG_CHN0_LOW;    //debug
    blc_pm_select_internal_32k_crystal();

    #if(MCU_CORE_TYPE == MCU_CORE_825x)
        cpu_wakeup_init();
    #elif(MCU_CORE_TYPE == MCU_CORE_827x)
        cpu_wakeup_init(LDO_MODE,EXTERNAL_XTAL_24M);
    #endif

    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeUp(); //MCU deep retention wakeUp

    rf_drv_init(RF_MODE_BLE_1M);

    gpio_init( !deepRetWakeUp ); //analog resistance will keep available in deepSleep mode, so
    ↪ no need initialize again

    clock_init(SYS_CLK_TYPE);

    if(!deepRetWakeUp){//read flash size
        blc_readFlashSize_autoConfigCustomFlashSector();
    }

    blc_app_loadCustomizedParameters(); //load customized freq_offset cap value

    if( deepRetWakeUp ){
        user_init_deepRetn ();
    }
    else{
        user_init_normal ();
    }

    irq_enable();
    while (1) {
#if (MODULE_WATCHDOG_ENABLE)
```

```

        wd_clear(); //clear watch dog
#ifdef
    main_loop ();
}
}

```

1.1.2 app_config.h

The user configuration file "app_config.h" serves to configure parameters of the whole system, including parameters related to BLE, GPIO, PM (low-power management), and etc. Parameter details of each module will be illustrated in following sections.

1.1.3 application file

- "app.c": User file for BLE protocol stack initialization, data processing and low power management.
- "app_att.c" of BLE slave project: configuration files for services and profiles. Based on Telink Attribute structure, as well as Attributes such as GATT, standard HID, proprietary OTA and MIC, user can add his own services and profiles as needed.
- UI task files: IR (Infrared Radiation), battery detect, and other user tasks.

1.1.4 BLE stack entry

There are two entry functions in BLE stack code of Telink BLE SDK.

- (1) BLE related interrupt handling entry "irq_blt_sdk_handler" in "irq_handler" function of the main.c file.

```

_attribute_ram_code_ void rf_irq_handler (void)
{
    .....
    irq_blt_sdk_handler ();
    .....
}

```

- (2) BLE logic and data handling entry "irq_blt_sdk_handler" in "main_loop" of the application file.

```

void main_loop (void)
{
    ////////// BLE entry //////////
    blt_sdk_main_loop();
    ////////// UI entry //////////
    .....
    ////////// PM process //////////
    .....
}

```

1.2 Applicable IC

The following IC models are applicable, they belong to B85m series with the same core, among which 8251/8253/8258, 8271/8273/8278 hardware modules are basically the same, only slightly different in SRAM size and flash. The details are shown in the table below.

Table 1.1: IC and memory supported by BLE B85m SDK

IC	Flash size	SRAM size
8251	512 kB	32 kB
8253	512 kB	48 kB
8258	512 kB/1 MB	64 kB
8271	512 kB	32 kB
8273	512 kB	64 kB
8278	1 MB	64 kB

Because the difference between the above 6 ICs is mainly the SRAM size, the other parts are the same, and the SDK file structure is completely shared except for the differences between the SDK/boot/boot script (i.e. software bootloader file) and the boot.link file.

1.3 Software Bootloader

The software bootloader file is stored in the SDK/boot/ directory, as shown below:

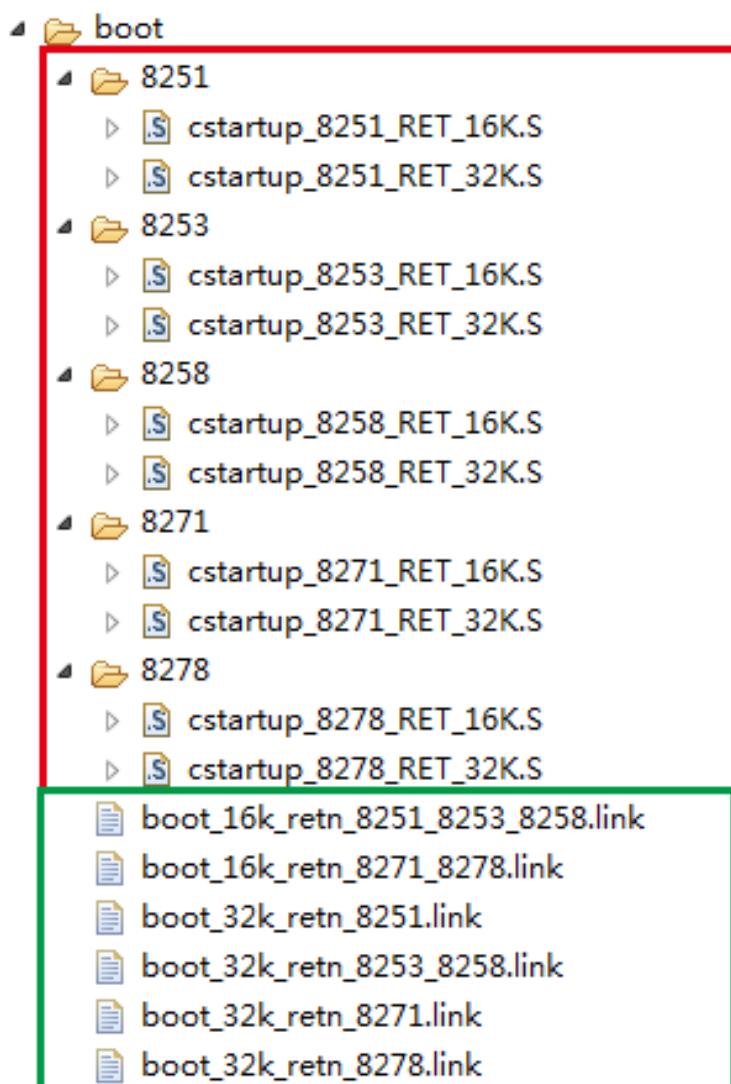


Figure 1.2: "Bootloader and boot.link path for different IC"

Each IC corresponds to two software bootloader files, which enable 16k deep retention and 32k deep retention respectively (for the introduction of deep retention, please refer to the chapter of "Low Power Management"). Since the 8273 and 8278 have the same boot file and link file, the 8278 configuration is used uniformly.

Take `cstartup_8258_RET_16K.S` as an example, the first sentence `#ifdef MCU_STARTUP_8258_RET_16K` illustrates that the bootloader will take effect only when `MCU_STARTUP_8258_RET_16K` is defined by user.

Users can choose different software bootloader according to the actual IC used and whether to use the deep retention (16K or 32K) function.

The default configuration of the project in B85 BLE SDK is 8258 with Sram size 64K , deepsleep retention 16K sram, i.e. the corresponding software bootloader and link files are `cstartup_8258_RET_16K.S` and `boot_16k_retn_8251_8253_8258.link` respectively. Users need to manually modify their configuration according to the type of chip they use and the evaluation of Retention size (for detailed analysis, please refer to the subsection "Sram Space").

Take `8258_ble_remote` as an example to illustrate how to change the software bootloader of 8258 to deep-

sleep retention 32K sram.

Step 1 Define -DMCU_STARTUP_8258_RET_32K in the 8258_ble_remote project settings as shown in the following figure.

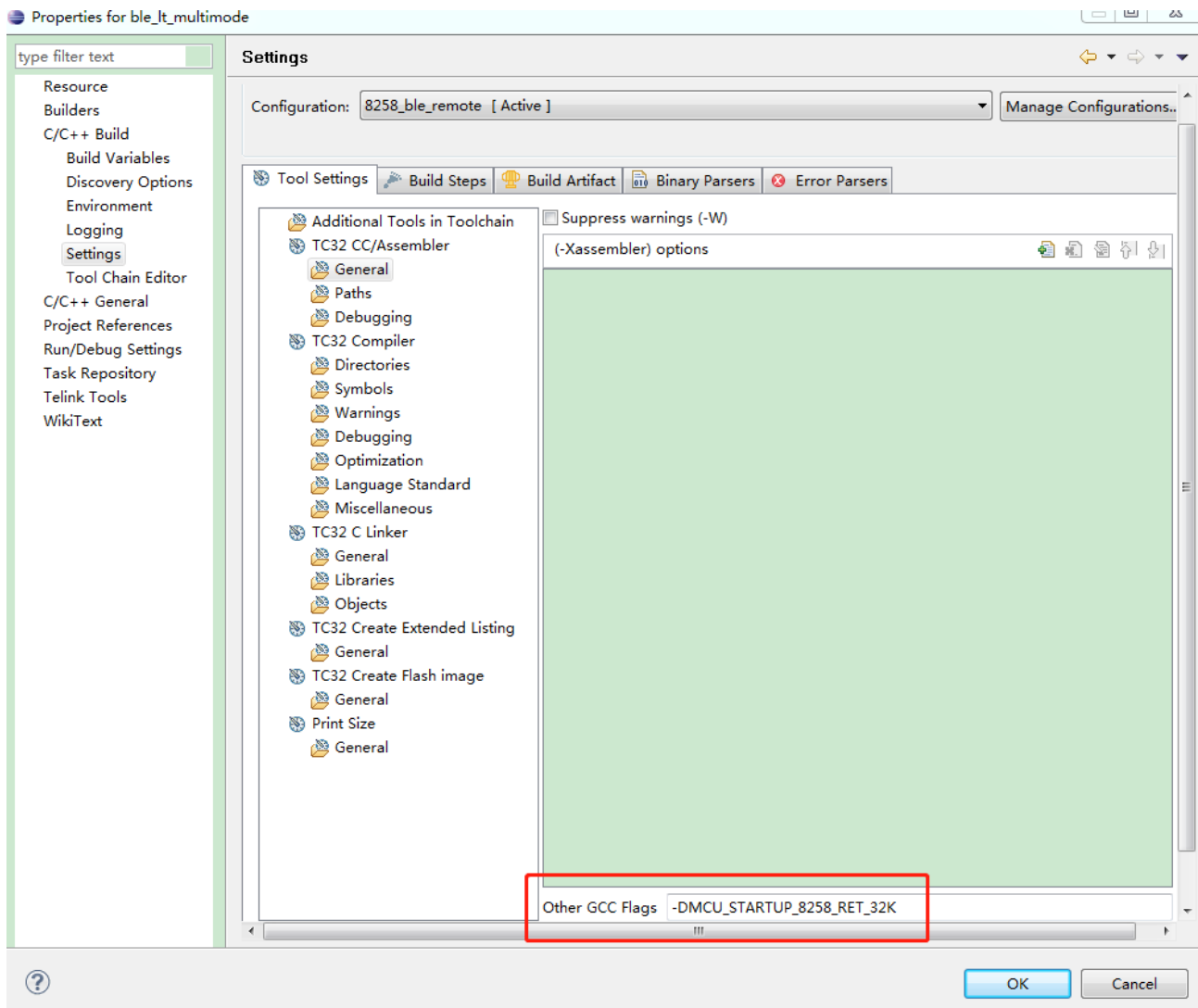


Figure 1.3: “software bootloader setting”

Note:

- According to the previous introduction, the hardware of 8251, 8253 and 8258 in B85m series is the same, and the hardware of 8271 and 8278 is the same, but the Sram size is different, so users need to modify the boot.link file in the root directory of SDK after choosing different software bootloader files (according to the correspondence in the following table, replace the contents of the link file in the SDK root directory according to the following table), the software bootloader and boot.link of different ICs are shown in the following table.

Table 1.2: Boot file selection for BLE B85m SDK

IC	16kB retention	32kB retention
8251	boot_16k_retn_8251_8253_8258.link cstartup_8251_RET_16K.S	boot_32k_retn_8251.link cstartup_8251_RET_32K.S
8253	boot_16k_retn_8251_8253_8258.link cstartup_8253_RET_16K.S	boot_32k_retn_8253_8258.link cstartup_8253_RET_32K.S
8258	boot_16k_retn_8251_8253_8258.link cstartup_8258_RET_16K.S	boot_32k_retn_8253_8258.link cstartup_8258_RET_32K.S
8271	boot_16k_retn_8271_8278.link cstartup_8271_RET_16K.S	boot_32k_retn_8271.link cstartup_8271_RET_32K.S
8273	boot_16k_retn_8271_8278.link cstartup_8278_RET_32K.S	boot_32k_retn_8278.link cstartup_8278_RET_32K.S
8278	boot_16k_retn_8271_8278.link cstartup_8278_RET_32K.S	boot_32k_retn_8278.link cstartup_8278_RET_32K.S

Step 2 According to the above example and the mapping table, the software bootloader file is cstartup_8258_RET_32K.S. You need to use the SDK/boot/boot_32k_retn_8253_8258.link file to replace the boot.link in the root directory of the SDK.

The following API is called after blc_ll_initPowerManagement_module() in API use_init() to set the Retention area of the hardware: blc_pm_setDeepsleepRetentionType(DEEPSLEEP_MODE_RET_ SRAM_LOW32K).

1.4 Demo codes

Telink BLE SDK provides users with multiple BLE demos.

Users can observe intuitive effects by running the software and hardware demo. Users can also modify the demo code to complete their own application development. Demo codes path is shown as below.

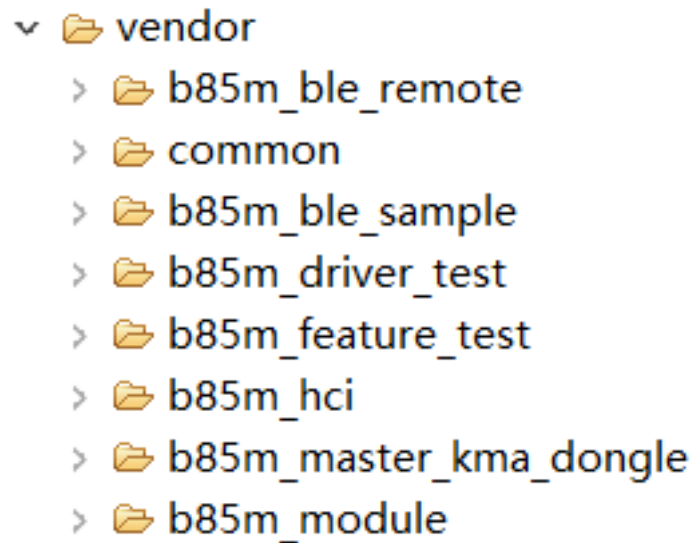


Figure 1.4: "BLE SDK demo code"

1.4.1 BLE Slave Demo

BLE slave demos and their differences are shown in the table below.

Table 1.3: Demo overview of BLE slave

Demo	Stack	Application	MCU Function
B85m hci	BLE controller	No	BLE controller, only Advertising and one Slave connection
B85m module	BLE controller + host	Application is on the host MCU	BLE transmissive module
B85m ble remote	BLE controller + host	Remote application	Host MCU
B85m ble sample	BLE controller + host	The simplest slave demo for broadcast and connection	Host MCU
B85m feature	BLE controller + host	Collection of various features	Host MCU

B85m hci is a BLE slave controller that provides USB/UART based HCI, and communicates with other MCU host to form a complete BLE slave system.

B85m module is only used as a BLE transmissive module to communicate with the host MCU through UART interface, and the general application code is written in the other host MCU.

B85m module realizes the function of controlling the related state change through the transmissive mod-

ule.

Note:

- Due to the complexity of the function implementation, the B85m module must be replaced with a 32k retention-related configuration and compiled for use.

B85m remote is a remote control demo based on the full slave role, including low voltage detection, key scan, NEC format IR transmitting, OTA over-the-air upgrade, application layer power management, Bluetooth control, voice transmission, IR learning and other functions. Users can learn what the structure of a basic use case is based on this project, and how most of the functions are implemented in the application layer.

Note:

- As voice, IR and IR learning consume more ram resources, when opening these functions, B85m remote must be replaced with 32k retention related configuration and compiled for use.

B85m ble sample is a simplified version of B85m_ble_remote and can be paired and connected with standard IOS/android devices.

1.4.2 BLE master demo

B85m master kma dongle is a demo of BLE master single connection, which can connect and communicate with B85m ble sample/B85m ble remote/B85m module.

The corresponding library of B85m ble remote/B85m ble sample provides a standard BLE stack (master and slave share a library), including BLE controller + BLE host, users only need to add their own application code in the app layer and do not have to deal with the BLE host stuff, and completely rely on the controller and host APIs.

The new SDK's library combines the slave and master libraries into one, and the B85m master kma dongle compiled code will only call the standard BLE controller function part of the library, the library does not provide the standard host function of master. The B85m Host master kma dongle demo code gives the reference BLE Host implementation methods on the app layer, including ATT, the simple SDP (service discovery protocol) and the most common SMP (security management protocol).

The most complicated function of BLE master is service discovery of slave server and identification of all services, which is usually realized in android/linux system. Telink B85m IC cannot provide complete service discovery due to the limitation of flash size and sram size. However, the SDK provides all ATT interfaces needed for service discovery. Users can refer to B85m master kma dongle's service discovery process for B85m ble remote to achieve their own specific service traversal.

1.4.3 Feature Demo and driver demo

B85m_feature_test gives demo code for some common BLE-related features, users can refer to these demos to complete their own functional implementation, see code for details. The BLE section of the document will introduce all the features.

The macro "FEATURE_TEST_MODE" is optionally defined in feature_config.h in the B85m_feature_test project to switch to different feature demos.

B85m driver test gives a sample code for the basic drivers for users to refer and implement their own driver functions. The driver section of this document will introduce each driver in detail.

The macro "DRIVER_TEST_MODE" is optionally defined in app_config.h in the B85m driver test project to switch to the demo of different driver test.

Telink Semiconductor

2 MCU Basic Modules

2.1 MCU Address Space

2.1.1 MCU Address Space Allocation

A typical 64K SRAM is used as an example to introduce MCU address space allocation. It is shown in the figure below.

The Telink B85m MCU has a maximum addressable space of 16M bytes.

- The 8M space from 0 to 0x7FFFFFFF is the program space, i.e. the maximum program capacity is 8M bytes.
- 0x800000 to 0xFFFFFFFF is the external device space: 0x800000~0x80FFFF is the register space; 0x840000~0x84FFFF is the 64K SRAM space.

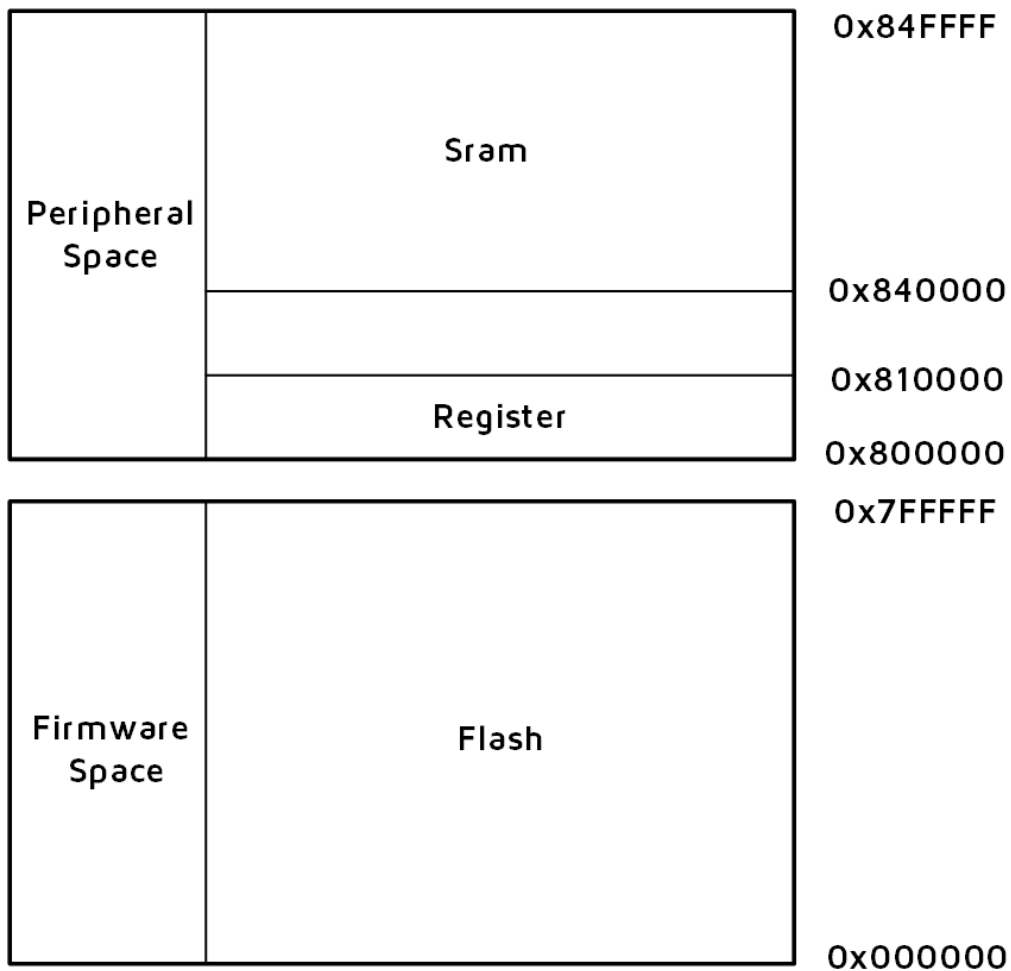


Figure 2.1: "MCU Address Space Allocation"

When the B85m MCU is physically addressed, address line BIT (23) is used to distinguish between program space/peripheral space.

- When this address is 0, access the program space.
- When this address is 1, access the peripheral space.

When the addressing space is peripheral space (BIT(23) is 1), address line BIT(18) is used to distinguish between Register and SRAM.

- When this address is 0, access the register.
- When this address is 1, access the SRAM.

2.1.2 SRAM Space Allocation

The space allocation of B85m SRAM is closely related to the deepsleep retention function in the low-power management section, so please master the knowledge about deepsleep retention first.

If you do not use the deepsleep retention function, and only use the suspend and normal deepsleep functions, the B85m SRAM space allocation is the same as Telink's previous generation BLE IC 826x series. Users who have used 826x BLE SDK can refer to the introduction of SRAM space allocation in "826x BLE SDK handbook" and then compare it with B85m SRAM space allocation to be introduced in this section to deepen their understanding of this part.

2.1.2.1 SRAM and Firmware Space

The allocation of SRAM space in the MCU address space is further explained.

The 32kB SRAM address space range is 0x840000 ~ 0x848000, the 48kB SRAM address space range is 0x840000 ~ 0x84C000, and the 64kB SRAM address space range is 0x840000 ~ 0x850000.

The following figure shows the SRAM space allocation for the 8258, 8253 and 8251 in 16k retention and 32k retention modes. Note that when the IC is 8251 and deepsleep retention 32K SRAM mode is used, the segments of the SRAM space allocation are dynamically adjusted, which can be found in the corresponding software bootloader and link files.

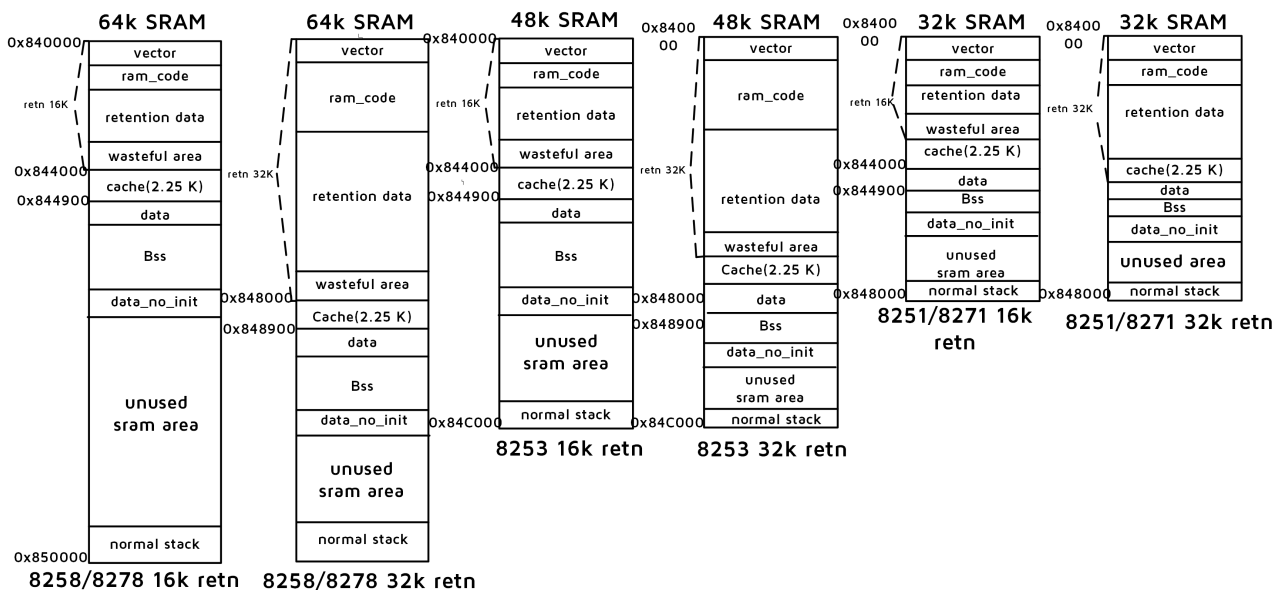


Figure 2.2: "SRAM space allocation for each IC at 16k and 32k retention"

The following is a detailed description of each part of the SRAM area, using the IC 8258 with a SRAM size of 64K and the default deepsleep retention 16K SRAM mode in the SDK as an example. If the SRAM size is other values or deepsleep retention 32k SRAM mode, the user can analogize.

The 64k SRAM corresponds to the SRAM and Firmware space allocation as shown below.

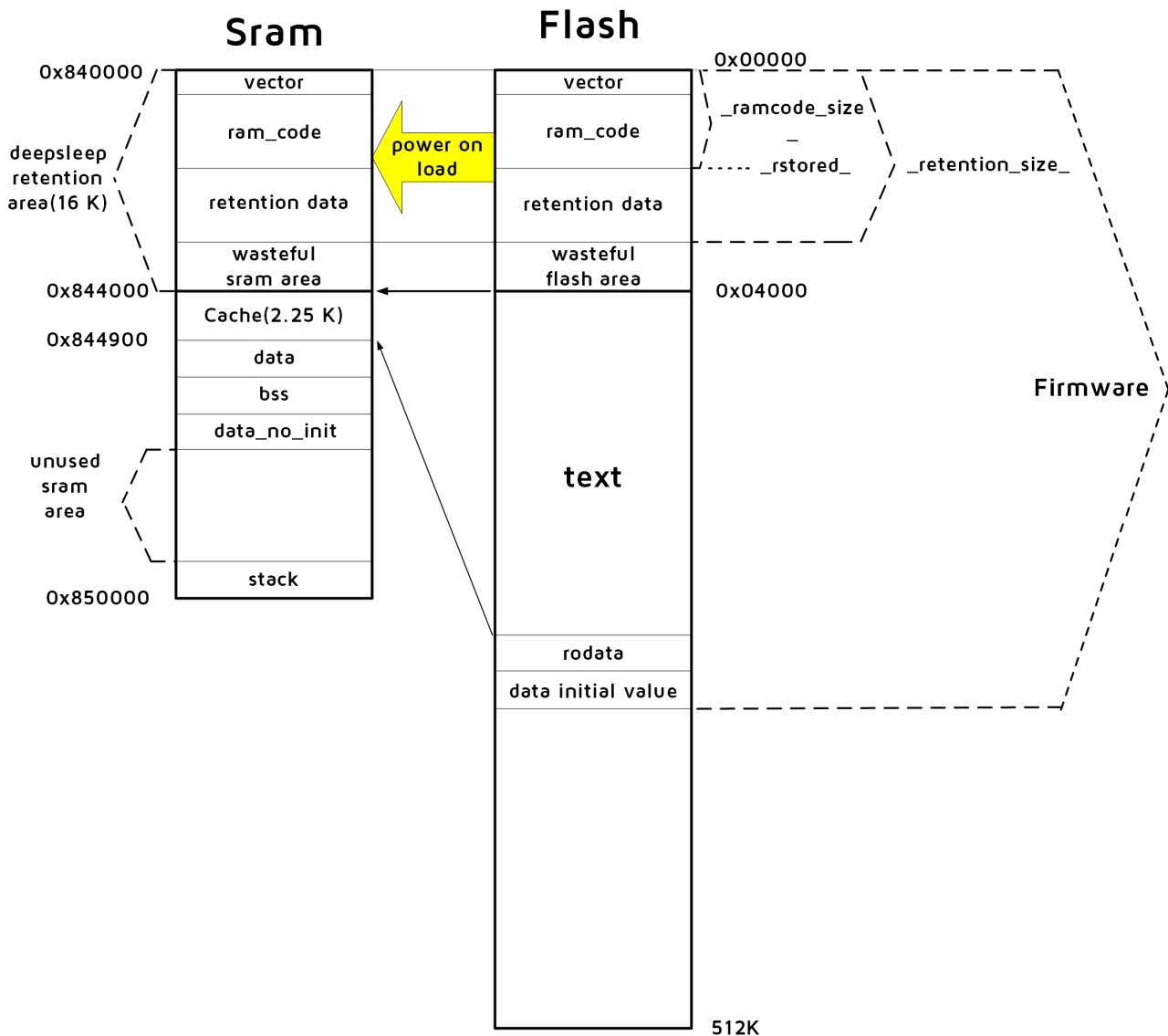


Figure 2.3: "SRAM space allocation & Firmware space allocation"

The files related to SRAM space allocation in SDK are boot.link (as we can see from the section "Introduction to software bootloader", the content of boot.link here is the same as boot_16k_retn_8251_8253_8258.link) and cstartup_8258_RET_16K.S. (If we use deepsleep retention 32K SRAM, the bootloader corresponds to cstartup_8258_RET_32K.S and the link file corresponds to boot_32k_retn_8253_8258.link.)

The firmware in flash includes vector, ramcode, retention_data, text, Rodata and Data initial value.

SRAM includes vector, ramcode, retention_data, Cache, data, bss, stack and unused SRAM area.

The vector/ramcode/ retention_data in SRAM is a copy of vector/ramcode/ retention_data in flash.

(1) vectors and ram_code

The “vectors” segment is the program corresponding to the assembly file `cstartup_8258_RET_16K.S`, which is the software bootloader.

The “ramcode” segment is the code in the Flash Firmware that needs to be resident in memory, corresponding to all functions in the SDK with the keyword “attribute_ram_code”, such as `flash_erase_sector` function.

```
_attribute_ram_code_ void flash_erase_sector(u32 addr);
```

There are two reasons for functions to be resident in memory.

One is that some functions must be resident in memory because they involve timing multiplexing with the four pins of the Flash MSPI, and if they are put into flash there will be timing conflicts that will cause crashes, such as all functions related to flash operations.

The second is that the function in the ram is executed every time it is called without re-reading from flash, which can save time. So for some functions with execution time requirements can be put into the resident memory to improve execution efficiency. SDK will be BLE timing-related functions often to be executed resident in memory, greatly reducing the execution time, and finally achieve power saving.

If you need to make a function resident in memory, you can follow `flash_erase_sector` above and add the keyword “attribute_ram_code” to your function, then you can see the function in the ramcode segment in the list file after compilation.

Both the vector and ramcode in Firmware need to be moved to ram when the MCU is powered up. After compilation, the size of these two parts are added together as `ramcode_size`. The `ramcode_size` is a variable value that the compiler can recognize, and its calculation is implemented in `boot.link`, as shown below. The compiled result `ramcode_size` is equal to the size of all codes of vector and ramcode.

```
. = 0x0;
.vectors :
{
*(.vectors)
*(.vectors.*)
}
.ram_code :
{
*(.ram_code)
*(.ram_code.*)
}
PROVIDE(_ramcode_size_ = . );//Calculate actual ramcode size(vector + ramcode)
```

(2) retention_data

B85m’s deepsleep retention mode supports the first 16K/32K of SRAM to keep the data on SRAM without losing power after the MCU enters retention.

The global variables in the program, if compiled directly, will be allocated in the “data” or “bss” segments, the contents of which are not in the first 16K of the retention area and will be lost after entering deepsleep retention.

If you want some specific variables to be saved without powering down during deepsleep (deepsleep retention mode), you can simply assign them to the "retention_data" segment by adding the keyword "_attribute_data_retention_" to the variable definition. Here are some examples.

```
_attribute_data_retention_ int AA;
_attribute_data_retention_ unsigned int BB = 0x05;
_attribute_data_retention_ int CC[4];
_attribute_data_retention_ unsigned int DD[4] = {0,1,2,3};
```

Referring to the "data/bss" section to be introduced below, we can see that the initial value of the global variable in the data section needs to be stored on the flash in advance; the initial value of the variable in the bss section is 0, so there is no need to prepare it in advance, and the bootloader can set it to 0 directly on the sram when running.

However, the global variables in the "retention_data" segment are unconditionally prepared with their initial values stored in the flash's retention_data area, regardless of whether the initial value is 0 or not. After power on (or normal deepsleep wake-up), they will be copied to the retention_data area of sram as a whole.

The "retention_data" segment follows the "ram_code" segment, i.e. "vector + ramcode + retention_data" which are arranged in order in front of the flash, and their total size is "retention_size". After the MCU is powered on (deepsleep wake-up), "vector+ramcode+ retention_data" is copied to the front of the sram as a whole, and thereafter, as long as the program is not in deepsleep (only suspend/deepsleep retention), the content of this whole block remains on the sram, and the MCU does not need to read it from the flash.

The configuration related to the retention_data segment in the boot.link file is as follows.

```
. = (0x840000 + (_rstored_));
.retention_data :
    AT ( _rstored_ )
    {
        . = (((. + 3) / 4)*4);
        PROVIDE(_retention_data_start_ = . );
        *(.retention_data)
        *(.retention_data.*)
        PROVIDE(_retention_data_end_ = . );
    }
```

The meaning of the above configuration is: when compiling, we see that the variable with the keyword "retention_data" is distributed in the flash firmware with the starting address "_rstored_", and the corresponding address in Sram is 0x840000 + (_rstored_). The value of "_rstored_" is the end of the "ram_code" section.

When using deepsleep retention 16K Sram mode, "retention_size" cannot exceed 16K, if it exceeds the 16K limit, user can choose to switch to deepsleep retention 32K Sram mode. If the user selects a configuration that uses deepsleep retention 16K Sram mode, but the defined "retention_size" exceeds the 16K limit, the compilation will result in the error as shown in the figure below.

The user can correct the error in one of the following ways.

- a. Reduce the data of the defined "attribute_data_retention".
- b. Choose to switch to deepsleep retention 32K Sram, refer to subsection 1.3 for detailed configuration.

When "retention_size" does not exceed 16K (assume 12K), there is a 4K "wasteful flash area" on the flash. In the corresponding firmware binary file we can see that the 12K ~ 16K contents are all invalid "0", and after copying to Sram, there will be 4K "wasteful sram area" (invalid SRAM area) on Sram as well.

If the user does not want to waste too much flash/sram, appropriate ram_code and retention_data can be added, and switch the functions /variables that were not in ram_code/retention_data before by adding the corresponding keywords to ram_code /retention_data. The functions placed in ram_code can save running time to reduce power consumption, and the variables placed in retention_data can save initialization time to reduce power consumption (please refer to the introduction of low-power management section for details).

(3) Cache

The Cache is the MCU's instruction cache and must be configured as a section of Sram to function properly. The Cache size is fixed and consists of 256 bytes of tag and 2048 bytes of instructions cache, totaling 0x900 = 2.25K.

The code of resident in memory can be read and executed directly from SRAM, but the code in firmware that can be resident in SRAM is only partial, and most of the rest is still in flash. According to the principle of program locality, a part of flash code can be stored in Cache, and if the current code to be executed is in Cache, it can be directly read from Cache and executed; if it is not in Cache, the code can be read from flash and moved to Cache, and then read from Cache and executed.

The "text" segment of the Firmware is not placed in the SRAM. This part of the code conforms to the principle of program locality and needs to be loaded into the Cache to be executed.

The Cache size is fixed at 2.25K, its starting address in Sram is configurable, here it is configured to the back of Sram 16K retention area, i.e. starting address is 0x844000 and ending address is 0x844900.

(4) data / bss

The "data" segment is a global variable in Sram that the program is initialized, i.e., global variables with non-zero initial value. The "bss" segment is the global variable in Sram that the program is not initialized, i.e., the global variable with initial value 0. These two parts are linked together, and the data segment is immediately followed by the bss segment, so they are presented here as a whole.

The "data" + "bss" segment follows the Cache and starts at the end of the Cache at 0x844900. The following code from boot.link defines the address of the start of the data segment on Sram.

```
. = 0x844900;
.data :
```

The "data" segment is a global variable that is initialized and its initial value needs to be stored in flash in advance, i.e. the "data initial value" in the Firmware shown in the figure of SRAM space allocation and Firmware space allocation.

(5) data_no_init

We added this segment to save the ram in the retention section. The feature of this segment is that it is in the ram, but not in the retention segment, and the initial value of the variables in this segment is random. This segment is used by the SDK for optimization purposes and is not recommended for users. If you really want to use this segment in your application, you need to make sure that the variable must be assigned a value before it is used, and that it cannot go through deep retention/deep/reboot/restart etc. between the assignment and use.

(6) stack / unused area

For the default 64K Sram, the "stack" starts from the highest address 0x850000 (0x84C000 for 48K Sram and 0x848000 for 32K Sram), and its direction is from bottom to top. The stack pointer SP decreases when the data is put on the stack and increases when the data is popped out of the stack.

By default, the SDK library uses a stack size of no more than 256 bytes, but since the stack size depends on the address of the deepest position of the stack, the final stack usage is related to the user's upper-level program design. If the user uses a troublesome recursive function call, or uses a relatively large local array variable in the function, or other situations that may cause the stack to be deeper, the final stack size will increase.

When the user uses more sram, he needs to know exactly how much stack his program uses. This cannot be analyzed by the list file, but only by letting the application run and making sure it runs all the codes in the program that may use deeper stack, then reset the MCU and read the sram space to determine the amount of stack used.

The "unused area" is the space left between the end of the bss segment and the deepest address of the stack. Only when this space exists, it means that stack is not in conflict with bss and there is no problem with Sram usage. If the deepest part of the stack overlaps with the bss segment, then there is not enough Sram.

Through the list file, we can find out the address of the end of the bss segment, which also determines the maximum space left for stack, and the user needs to analyze whether this space is enough, and in combination with the deepest address of stack mentioned above, we can know whether the use of Sram is exceeded. The analysis method will be given in the following demo.

(7) text

The "text" segment is a collection of all non-ram_code functions in the Flash Firmware. If "_attribute_ram_code_" is added to the function in the program, it will be compiled into the ram_code segment, while all other functions without this keyword will be compiled into the "text" segment. In general, the "text" segment is the largest space in the firmware, much larger than the size of Sram, so it is necessary to load the code to be executed into the Cache first through the cache function of the Cache before it can be executed.

(8) rodata /data init value

Except for vector, ram_code and text, the remaining data in Flash Firmware are "rodata" segment and "data initial value".

The "rodata" segment is the readable and unwritable data defined in the program, and is a variable defined by the keyword "const". For example, the ATT table in Slave.


```
static const attribute_t my_Attributes[] = .....
```

The user can see in the corresponding list file that “my_Attributes” is in the rodata segment.

The “data” segment introduced earlier is a global variable that has been initialized in the program, for example, the global variable is defined as follows.

```
int testValue = 0x1234;
```

Then the compiler will store the initial value 0x1234 in the “data initial value”, and when running the boot-loader, it will copy the initial value to the memory address corresponding to the testValue.

2.1.2.2 list file analysis demo

Here we take the simplest demo 825x ble sample of BLE slave as an example and analyze it with “Sram space allocation & Firmware space allocation”.

The bin file and list file of 825x ble sample can be found in the directory “SDK”->“Demo”->“list file analyze”.

In the following analysis, there will be several screenshots, all from boot.link, cstartup_8258_RET_16K.S, 825x ble sample.bin and 825x ble sample.list, please find the file to find the corresponding location of the screenshots by yourself.

The distribution of each section in the list file is shown in the following figure (note the Algn byte alignment):

Sections:						
Idx	Name	Size	VMA	LMA	File off	Algn
0	.vectors	00000170	00000000	00000000	00008000	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.ram_code	000023f0	00000170	00000170	00008170	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.text	0000614c	00004000	00004000	0000c000	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
3	.rodata	000008ec	0000a14c	0000a14c	0001214c	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.retention_data	00000ce8	00842560	00002560	0000a560	2**2
	CONTENTS, ALLOC, LOAD, DATA					
5	.data	0000002c	00844900	0000aa38	00014900	2**2
	CONTENTS, ALLOC, LOAD, DATA					
6	.bss	00000259	00844930	0000aa68	0001492c	2**4
	ALLOC					
7	.TC32.attributes	00000010	00000000	00000000	0001492c	2**0
	CONTENTS, READONLY					
8	.comment	0000001a	00000000	00000000	0001493c	2**0
	CONTENTS, READONLY					

Figure 2.4: “list file section analysis”

According to the section analysis, below lists the information you need to know, detailed introduction will be introduced later.

- (1) vectors: start from Flash 0, Size is 0x170, the end address is calculated as 0x170;
- (2) ram_code: start from Flash 0x170, Size is 0x23f0, the end address is calculated as 0x2560;
- (3) retention_data: start from Flash 0x2560, Size is 0xce8, the end address is calculated as 0x3248;
- (4) text: start from Flash 0x4000, Size is 0x614c, the end address is calculated as 0xa14c;
- (5) rodata: start from Flash 0xa14c, Size is 0x8ec, the end address is calculated as 0xaa38;
- (6) data: start from Sram 0x844900, Size is 0x2c, the end address is calculated as 0x84492c;
- (7) bss: start from Sram 0x844930, Size is 0x259, the end address is calculated as 0x844b89.

Combined with the previous introduction, the remaining Sram space is $0x850000 - 0x844b89 = 0xb477 = 46199$ byte, minus the 256 byte needed for stack, leaving 45943 byte.

```

Disassembly of section .vectors:

00000000 <__start>:
-----
Disassembly of section .ram_code:

00000170 <blt_packet_crc24_opt>:
-----
Disassembly of section .text:

00004000 <__modsi3>:
-----
Disassembly of section .rodata:

0000a14c <C.1.4443-0x8>:
-----
Disassembly of section .retention_data:

00842560 <_retention_data_start>:
-----
Disassembly of section .data:

00844900 <_start_data>:
-----
Disassembly of section .bss:

00844930 <_start_bss>:
-----
00844b88 <blt_dma_tx_rptr>:
...
-----
00002560 g      *ABS*  00000000 _rstored_

```

Figure 2.5: "list file section address"

The above figure shows the starting address of some of the sections in the list file after the search for "section", combined with this figure and the above "list file section statistics", the analysis is as follows.

- (1) vector:

The "vectors" segment starts at 0, ends at 0x170 (last data address is 0x16e-0x16f), and size is 0x170. After power-on move to Sram, the address on Sram is 0x840000 ~ 0x840170.

(2) ram_code:

The "ram_code" segment starts at 0x170 and ends at 0x2560 (the last data address is 0x255c-0x255f). After the power-on move to Sram, the address on Sram is 0x840170 ~ 0x842560.

(3) retention_data:

The starting address of "retention_data" in flash "_rstored_" is 0x2560, which is also the end of "ram_code"

The starting address of "retention_data" in Sram is 0x842560 and the ending address is 0x843248 (the last data address is 0x843244 ~ 0x843247).

The total size value of "vector+ram_code+retention_data" "retention_size" is 0x3248, so the first 16K of the flash firmware is only 0x3248 byte of valid data. The space from 0x3248 to 0x4000 is about 3.43K which belongs to "wasteful flash area"(invalid flash area) (user can open 825x_ble_sample.bin to see that this space is full of invalid zeros). The space from 0x843248 to 0x844000 about 3.43K belongs to "wasteful sram area" (invalid SRAM area).

(4) Cache:

The Cache address range in Sram is 0x844000~0x844900. The information about Cache will not be reflected in the list file.

(5) text:

The "text" segment in the flash firmware starts at 0x4000, ends at 0xa14c (the last data address is 0xa148~0xa14b), and the Size is 0xa14c- 0x4000 = 0x614c, and the data in the previous Section statistics are consistent.

(6) rodata:

The starting address of the "rodata" segment is the end address of text 0xa14c, and the end address is 0xaa38 (the last data address is 0xaa34~0xaa37).

(7) data:

The starting address of the "data" segment on the Sram is the end address of the Cache 0x844900, and the size given in the Section statistics section of the list file is 0x2c.

The end address of the "data" section on the Sram is 0x84492c (the last data address is 0x844928~0x84492b).

(8) bss:

The starting address of the "bss" segment on the Sram is the end address of the "data" segment 0x844930 (16-byte alignment), and the size given in the Section statistics section of the list file is 0x259.

The end address of the "bss" section on Sram is 0x844b89 (the last data address is 0x844b84~0x844b88).

The remaining Sram space is 0x850000 - 0x844b89 = 0xb477 = 46199 byte, minus the 256 byte needed for stack, leaving 45943 byte.

2.1.3 MCU Address Space Access

Access to the 0x000000 - 0xFFFFFFFF address space in the program is divided into the following two situations.

2.1.3.1 Peripheral Space R/W Operation

Read and write operations in the peripheral space (register and sram) are implemented directly with pointer access.

```
u8  x = *(volatile u8*)0x800066;  //read value of register 0x66
    *(volatile u8*)0x800066 = 0x26;  //assign value to register 0x66
u32 y = *(volatile u32*)0x840000;    //read value of sram 0x40000-0x40003
    *(volatile u32*)0x840000 = 0x12345678; //assign value to sram 0x40000-0x40003
```

The program uses the functions write_reg8, write_reg16, write_reg32, read_reg8, read_reg16, read_reg32 to read and write to the peripheral space, which are essentially pointer operations. For more information, please refer to drivers/8258/bsp.h.

Note the operation similar to write_reg8(0x40000)/ read_reg16(0x40000) in the program, which is defined as shown below, from which the 0x800000 offset is automatically added (address line BIT(23) is 1), so the MCU can ensure that it is accessing the Register/Sram space and not going to flash space.

```
#define REG_BASE_ADDR      0x800000
#define write_reg8(addr,v)  U8_SET((addr + REG_BASE_ADDR),v)
#define write_reg16(addr,v) U16_SET((addr + REG_BASE_ADDR),v)
#define write_reg32(addr,v) U32_SET((addr + REG_BASE_ADDR),v)
#define read_reg8(addr)     U8_GET((addr + REG_BASE_ADDR))
#define read_reg16(addr)    U16_GET((addr + REG_BASE_ADDR))
#define read_reg32(addr)    U32_GET((addr + REG_BASE_ADDR))
```

Note here a memory alignment problem: If you use a pointer to 2 bytes/4 bytes to read or write peripheral space, make sure the address is 2 bytes/4 bytes aligned, if not aligned, data read/write errors will occur. The following two are errors.

```
u16 x = *(volatile u16*)0x840001;  //0x840001 is not 2-byte aligned
*(volatile u32*)0x840005 = 0x12345678; //0x840005 is not 4-byte aligned
```

Modify to the correct read/write operation.

```
u16 x = *(volatile u16*)0x840000;  //0x840000 is 2-byte aligned
*(volatile u32*)0x840004 = 0x12345678; //0x840004 is 4-byte aligned
```

2.1.3.2 Flash operation

This section is detailed in Chapter 8, Flash.

2.1.4 SDK Flash space allocation

This section is detailed in Chapter 8, Flash.

2.2 Clock Module

2.2.1 System clock & System Timer

The system clock is the clock used by the MCU to execute the program.

The system timer is a read-only timer that provides a time reference for timing control of the BLE and is also available to the user.

On Telink's previous generation ICs (826x series), the System Timer clock is from the system clock, while on the B85m series ICs, the System Timer and system clock are independently separated. As shown in the figure below, the System Timer is 16MHz obtained from the external 24MHz Crystal Oscillator by 3/2 division.

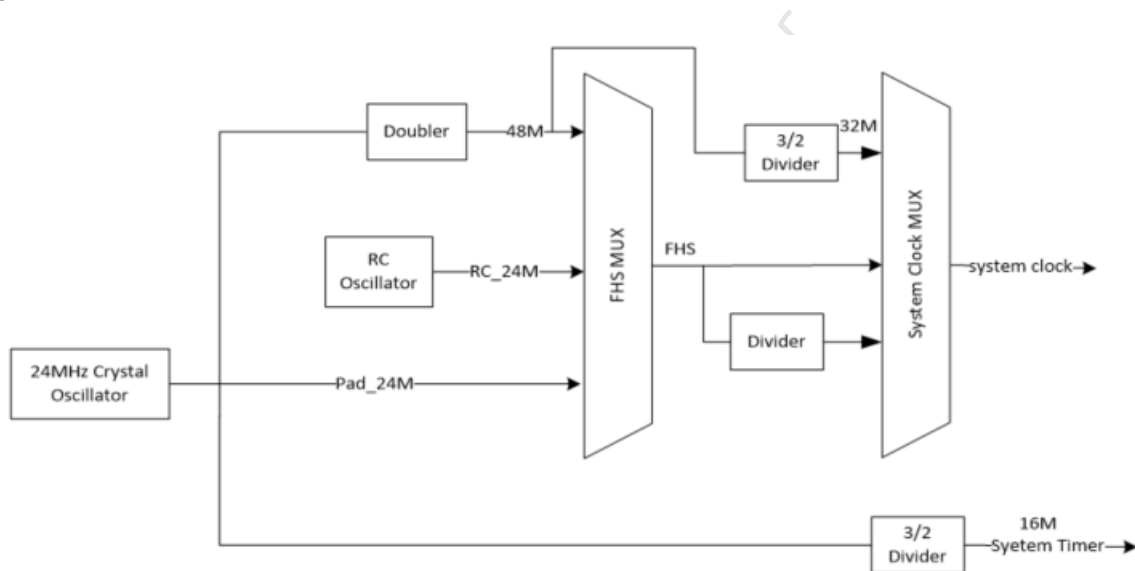


Figure 2.6: "System clock & System Timer"

As you can see, the system clock can be multiplied to 48M by the external 24M crystal oscillator through the "doubler" circuit and then divided to get 16M, 24M, 32M, 48M, etc. This type of clock is called crystal clock (such as 16M crystal system clock, 24M crystal system clock). It can also be processed by the IC internal 24M RC Oscillator to get 24M RC clock, 32M RC clock, 48M RC clock and so on. This category is called RC clock (BLE SDK does not support RC clock).

In the BLE SDK we recommend to use crystal clock.

To configure the system clock, call the following API during initialization and select the clock corresponding to the clock in the enumeration variable `SYS_CLK_TYPEDEF` definition.

```
void clock_init(SYS_CLK_TYPEDEF SYS_CLK)
```

Since the System Timer of B85m series chip is different from the system clock, users need to know whether the clock of each hardware module on MCU is from system clock or System Timer. Let's take the case where the system clock is 24M crystal to illustrate, at this time the system clock is 24M and the System Timer is 16M.

In the file app_config.h, the system clock and the corresponding s, ms and us are defined as follows.

```
#define CLOCK_SYS_CLOCK_HZ      24000000
enum{
    CLOCK_SYS_CLOCK_1S = CLOCK_SYS_CLOCK_HZ,
    CLOCK_SYS_CLOCK_1MS = (CLOCK_SYS_CLOCK_1S / 1000),
    CLOCK_SYS_CLOCK_1US = (CLOCK_SYS_CLOCK_1S / 1000000),
};
```

All hardware modules whose clock source is system clock can only use the above CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_1S, etc. when setting the clock of the module; in other words, if the user sees that the clock setting in the module uses the above definitions, it means that the clock source of the module is system clock.

If the PWM driver PWM period and duty cycle are set as follows, it means that the PWM clock source is system clock.

```
pwm_set_cycle_and_duty(PWM0_ID, (u16) (1000 * CLOCK_SYS_CLOCK_1US), (u16) (500 *
↪ CLOCK_SYS_CLOCK_1US) );
```

The System Timer is a fixed 16M, so for this timer, the SDK code uses the following values for s, ms and us.

```
//system timer clock source is constant 16M, never change
enum{
    CLOCK_16M_SYS_TIMER_CLK_1S = 16000000,
    CLOCK_16M_SYS_TIMER_CLK_1MS = 16000,
    CLOCK_16M_SYS_TIMER_CLK_1US = 16,
};
```

The following APIs in SDK are some operations related to System Timer, so when it comes to these API operations, they all use the above similar to "CLOCK_16M_SYS_TIMER_CLK_xxx" to represent the time.

```
void sleep_us (unsigned long microsec);
unsigned int clock_time(void);
int clock_time_exceed(unsigned int ref, unsigned int span_us);
#define ClockTime      clock_time
#define WaitUs          sleep_us
#define WaitMs(t)       sleep_us((t)*1000)
```

Since the System Timer is the reference for BLE timing, all BLE time-related parameters and variables in the SDK are expressed as "CLOCK_16M_SYS_TIMER_CLK_xxx" when it comes to the time.

2.2.2 System Timer Usage

After the initialization of sys_init in the main function is completed, the System Timer starts to work, and the user can read the value of the System Timer counter (referred to as System Timer tick).

The System Timer tick is incremented by one every clock cycle, and its length is 32bit, that is, every 1/16 us plus 1, the minimum value is 0x00000000, and the maximum value is 0xffffffff. When the System Timer starts, the tick value is 0, and the time required to reach the maximum value of 0xffffffff is: (1/16) us * (2³²) approximately equal to 268 seconds, and the System Timer tick makes one cycle every 268 seconds.

The system tick will not stop when the MCU is running the program.

The reading of System Timer tick can be obtained through the clock_time() function:

```
u32 current_tick = clock_time();
```

The entire BLE timing of the BLE SDK is designed based on the System Timer tick. This System Timer tick is also used extensively in the program to complete various timing and timeout judgments. It is strongly recommended that users use this System Timer tick to implement some simple timing and timeout judgments.

For example, to implement a simple software timing. The realization of the software timer is based on the query mechanism. Because it is implemented through query, it cannot guarantee real-time performance and readiness. It is generally used for applications that are not particularly demanding on error. Implementation:

- (1) Start timer: set a u32 variable, read and record the current System Timer tick.

```
u32 start_tick = clock_time(); // clock_time() returns System Timer tick value
```

- (2) Constantly inquire whether the difference between the current System Timer tick and start_tick exceeds the time value required for timing in the program. If it exceeds, consider that the timer is triggered, perform corresponding operations, and clear the timer or start a new round of timing according to actual needs.

Assuming that the time to be timed is 100 ms, the way to query whether the time is reached is:

```
if( (u32) ( clock_time() - start_tick) > 100 * CLOCK_16M_SYS_TIMER_CLK_1MS)
```

Since the difference is converted to the u32 type, the limit of the system clock tick from 0xffffffff to 0 is solved.

In fact, in order to solve the problem of conversion to u32 caused by different system clocks, the SDK provides a unified calling function. Regardless of the system clock, the following functions can be used to query and judge:

```
if( clock_time_exceed(start_tick, 100 * 1000)) //unit of the second parameter is us
```

Please be noted: since the 16MHz clock takes 268 seconds for one cycle, this query function is only applicable to the timing within 268 seconds. If it exceeds 268 seconds, you need to add a counter to accumulate in the software (not introduced here).

Application example: after 2 seconds when A condition is triggered (only once), the program performs B() operation.

```
u32  a_trig_tick;
int  a_trig_flg = 0;
while(1)
{
    if(A){
        a_trig_tick = clock_time();
        a_trig_flg = 1;
    }
    if(a_trig_flg && clock_time_exceed(a_trig_tick, 2 * 1000 * 1000)){
        a_trig_flg = 0;
        B();
    }
}
```

2.3 GPIO Module

The description of GPIO module please refer to drivers/8258/gpio_8258.h, gpio_default_8258.h, gpio_8258.c to understand, all code is provided in source code form.

The code involves the operation of registers, please refer to the document "gpio_lookuptable" to understand it.

2.3.1 GPIO definition

B85m series chips have 5 groups of GPIOs, totally 36 GPIOs: GPIO_PA0 ~ GPIO_PA7, GPIO_PBO ~ GPIO_PB7, GPIO_PCO ~ GPIO_PC7, GPIO_PDO ~ GPIO_PD7, GPIO_PEO ~ GPIO_PE3

Note:

- There are 36 GPIOs in the core part of the IC, however some GPIOs may not be pinned out in the different packages of each IC, therefore please refer to the actual GPIO pins in the package of the IC when using GPIOs.

When you need to use GPIO in your program, you must define it as written above, see drivers/8258/gpio_8258.h for details.

Note:

The 7 GPIOs are special and need attention.

- 1) 4 GPIOs of MSPI, these 4 GPIOs are the main SPI bus in MCU system bus for read/write flash operation, power on default for spi state, user can never operate them, program can not use them. These 4 GPIOs are PEO, PE1, PE2 and PE3.
- 2) SWS (Single Wire Slave), used for debug and burning firmware, power on default for SWS state, it is generally not used in the program. The SWS pin in b85m chip is PA7.
- 3) DM and DP, power on default GPIO state. DM and DP need to be used when USB function is needed; when USB is not needed, it can be used as GPIO. The DM and DP pin of B85m are PA5 and PA6.

2.3.2 GPIO state control

Only the most basic GPIO states that users need to know are listed here.

- 1) func (function configuration: special function/general GPIO), if you need to use the input and output function, you need to configure it as general GPIO.

```
void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func);
```

The pin is defined for GPIO, the same as below. For func you can choose AS_GPIO or other special functions.

- 2) ie (input enable)

```
void gpio_set_input_en(GPIO_PinTypeDef pin, unsigned int value);
```

value: 1 and 0 means enable and disable respectively.

- 3) datai (data input): When the input enable is on, this value is the current level of this GPIO pin, which is used to read the external voltage.

```
unsigned int gpio_read(GPIO_PinTypeDef pin);
```

Reading a low voltage returns a value of 0; reading a high voltage returns a non-zero value. Be very careful here, when reading high, the return value is not necessarily 1, it is a non-0 value.

So in the program, you can not use code similar if(gpio_read(GPIO_PA0) == 1), the recommended use method is to do the inverse processing for the read value, after the inverse only 1 and 0 two cases:

```
if( !gpio_read(GPIO_PA0) ) //determine high or low level
```

- 4) oe (output enable)

```
void gpio_set_output_en(GPIO_PinTypeDef pin, unsigned int value);
```

value: 1 and 0 means enable and disable respectively.

- 5) dataO (data output): When the output enable is on, the value is 1 to output high, 0 to output low.

```
void gpio_write(GPIO_PinTypeDef pin, unsigned int value)
```

6) For the internal analog pull-up and pull-down resistor configuration, there are 3 analog resistors: 1Mohm pull-up, 10Kohm pull-up and 100Kohm pull-down, and 4 configurable states: 1Mohm pull-up, 10Kohm pull-up, 100Kohm pull-down and float state.

```
void gpio_setup_up_down_resistor( GPIO_PinTypeDef gpio, GPIO_PullTypeDef up_down);
```

The four configurations of up_down:

```
typedef enum {  
    PM_PIN_UP_DOWN_FLOAT    = 0,  
    PM_PIN_PULLUP_1M        = 1,  
    PM_PIN_PULLDOWN_100K    = 2,  
    PM_PIN_PULLUP_10K        = 3,  
}GPIO_PullTypeDef;
```

In the deepsleep and deepsleep retention states, the GPIO input and output states are all disabled, but the analog pull-up and pull-down resistors are still valid.

Note:

The following sequence is required when making GPIO function change configurations.

- (A) The beginning function is GPIO, then you need to configure the required function MUX first, and then disable the GPIO function.
- (B) The beginning function is IO, you need to change to GPIO output, first set the corresponding IO output value and OEN, and then finally enable GPIO function.
- (C) The beginning function is IO, you need to change to GPIO input and IO pullup, first set output to 1, OEN to 1 (corresponding to PA and PD), second set pullup to 1 (corresponding to PB and PC), and finally enable GPIO function.
- (D) Set pullup to 1 (corresponding to PB and PC) and IO not pull up, first set output to 0, OEN to 1 (corresponding to PA and PD), then set pullup to 0 (corresponding to PB and PC), and finally enable GPIO function.

GPIO configuration application examples:

- 1) Configure GPIO_PA4 as output state and output high level.

```
gpio_set_func(GPIO_PA4, AS_GPIO) ; // PA4 is GPIO by default, you can leave it  
gpio_set_input_en(GPIO_PA4, 0);  
gpio_set_output_en(GPIO_PA4, 1);  
gpio_write(GPIO_PA4,1)
```

- 2) Configure GPIO_PC6 as input state to determine whether it reads low and needs to turn on pull-up to prevent the effect of float level.

```
gpio_set_func(GPIO_PC6, AS_GPIO) ; // PC6 is GPIO by default, you can leave it
gpio_setup_up_down_resistor(GPIO_PC6, PM_PIN_PULLUP_10K);
    gpio_set_input_en(GPIO_PC6, 1)
    gpio_set_output_en(GPIO_PC6, 0);
    if(!gpio_read(GPIO_PC6)){ //whether low level
        .....
    }
```

3) Configure PA5 and PA6 pins for USB function.

```
gpio_set_func(GPIO_PA5, AS_USB ) ;
gpio_set_func(GPIO_PA6, AS_USB) ;
gpio_set_input_en(GPIO_PA5, 1);
gpio_set_input_en(GPIO_PA6, 1);
```

2.3.3 GPIO initialization

Calling the gpio_init function in main.c will initialize the state of all 32 GPIOs except for the 4 GPIOs of the MSPI.

This function initializes each IO to its default state when no GPIO parameters are configured in the user's app_config.h. The default states of the 32 GPIOs are:

1) func

Except SWS, all other states are general GPIOs.

2) ie

Except the default ie for SWS is 1, the default ie for all other general GPIOs is 0.

3) oe

All is 0.

4) dataO

All is 0.

5) Internal pull up/down resistors

All is float.

For more details, please refer to drivers/8258/ gpio_8258.h, drivers/8258/ gpio_default_8258.h and drivers/8278/ gpio_8278.h, drivers/8278/ gpio_default_8278.h.

If there is a state configured in app_config.h to one or more GPIOs, then gpio_init no longer uses the default state, but the state configured by the user in app_config.h. The reason for this is that the default state of gpio is represented using macros that are written (using PA0's ie as an example) as follows:

```
#ifndef PA0_INPUT_ENABLE
#define PA0_INPUT_ENABLE      1
#endif
```

When these macros can be defined in advance in app_config, these macros no longer use such default values as above.

The method to configure the GPIO state in app_config.h is (using PA0 as an example):

1) Configure func:

```
#define PA0_FUNC          AS_GPIO
```

2) Configure ie:

```
#define PA0_INPUT_ENABLE  1
```

3) Configure oe:

```
#define PA0_OUTPUT_ENABLE 0
```

4) Configure dataO:

```
#define PA0_DATA_OUT      0
```

5) Configure internal pull up/down resistors:

```
#define PULL_WAKEUP_SRC_PA0  PM_PIN_UP_DOWN_FLOAT
```

Summary of GPIO initialization:

- 1) The initial state of GPIO can be defined in app_config.h in advance and can be set in gpio_init.
- 2) It can be set in user_init function by GPIO state control function (gpio_set_input_en, etc.).
- 3) You can also use a mix of the above two ways: define some in app_config.h in advance, implement them in gpio_init, and set some others in user_init.

Note:

- If a state of the same GPIO is set to a different value in app_config.h and user_init, the setting in user_init will prevail according to the order of program execution.

The gpio_init function is implemented as follows. The value of anaRes_init_en determines whether the analog pull-up and pull-down resistors are set.

```
void gpio_init(int anaRes_init_en)
{
    // gpio digital status setting
    if(anaRes_init_en){
        gpio_analog_resistance_init();
    }
}
```

Referring to the introduction of low-power management in the later chapters of the document, we can see that the registers controlling the GPIO analog pull-up and pull-down resistors can be maintained without power loss during deepsleep retention, so the state of the GPIO analog pull-up and pull-down resistors can be maintained in deepsleep retention mode.

In order to ensure that the state of the GPIO analog pull-up and pull-down resistors is not changed after the deepsleep retention wakeup, it is necessary to determine whether the current deepsleep retention wake_up before gpio_init, and set the value of anaRes_init_en according to this state, as shown in the following code.

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();  
gpio_init( !deepRetWakeUp );
```

2.3.4 GPIO digital states fail in deepsleep retention mode

In the GPIO state control described above, all the states (func, ie, oe, dataO, etc.) are controlled by the digital register, except for the analog pull-down resistor which is controlled by the analog register.

Referring to the introduction of low-power management later in the document, it is clear that all digital register states are lost during deepsleep retention.

On Telink's previous generation 826x series ICs, GPIO output can be used to control some peripheral devices during suspend. But on B85m if suspend is switched to deepsleep retention mode, GPIO output state is disabled and cannot accurately control peripheral devices during sleep. At this point, you can use GPIO to simulate the state of pull-up and pull-down resistors instead: pull-up 10K instead of GPIO output high, pull-down 100K instead of GPIO output low.

Note:

- Do not use pull-up 1M for GPIO state control during deepsleep retention (the pull-up voltage may be lower than the supply voltage VCC). In addition, do not use the pull-up 10K of PC0-PC7 in the pull-up 10K control (there will be a short time jitter in the deepsleep retention wake_up, generating glitches), pull-up 10K for other GPIO is OK.

2.3.5 Configure SWS pull-ups to prevent crashes

All of Telink's MCUs use SWS (single wire slave) to debug and burn in programs. On the final application code, the state of the pin SWS is:

- 1) function set to SWS, not GPIO.
- 2) ie =1, only when input enable, it can receive various commands sent by EVK, which is used to operate MCU.
- 3) Other configurations: oe, dataO are 0.

After setting to the above state, it can receive operation commands from EVK at any time, but it also brings a risk: when the power supply of the whole system is very jittered (such as when sending IR, the instantaneous current may rush to nearly 100mA), as SWS is in float state, it may read a wrong data and mistake it for a command from EVK, and this wrong command may cause the program stuck.

The solution to the above problem is to modify the float state of the SWS to input pull-up. This is solved by an analog pull-up 1M resistor.

B85m's SWS and GPIO_PA7 are multiplexed, just enable the 1M pull-up of PA7 in drivers/8258/gpio_default_8258.h.

```
#ifndef PULL_WAKEUP_SRC_PA7
#define PULL_WAKEUP_SRC_PA7    PM_PIN_PULLUP_1M //sws pullup
#endif
```

2.4 System interrupt

This document applies to hardware interrupts of ICs with the following two characteristics.

- (1) All interrupts have the same priority, and the MCU does not have the ability to nest interrupts;
- (2) All interrupts share the same interrupt hardware entry, which will eventually trigger the software irq_handler function, in which the function reads the status bits of the relevant interrupt to determine whether the corresponding interrupt is triggered.

The feature 1 above determines that the MCU responds to interrupts on a first-come, first-served basis. When the first interrupt is not processed, a new interrupt is generated and cannot be responded to immediately and enters the waiting queue until the previous interrupts are processed. Therefore, when there are 2 or more interrupts, all interrupts cannot be responded in real time. The response delay of a particular interrupt depends on whether the MCU is processing other interrupts when this interrupt is triggered and how long it takes to process the other interrupts. As shown in the figure below, since IRQ1 is processing when IRQ2 is triggered, it must wait until IRQ1 is finished processing before responding. The worst case of IRQ 2 delay time is the maximum time of IRQ1 process.

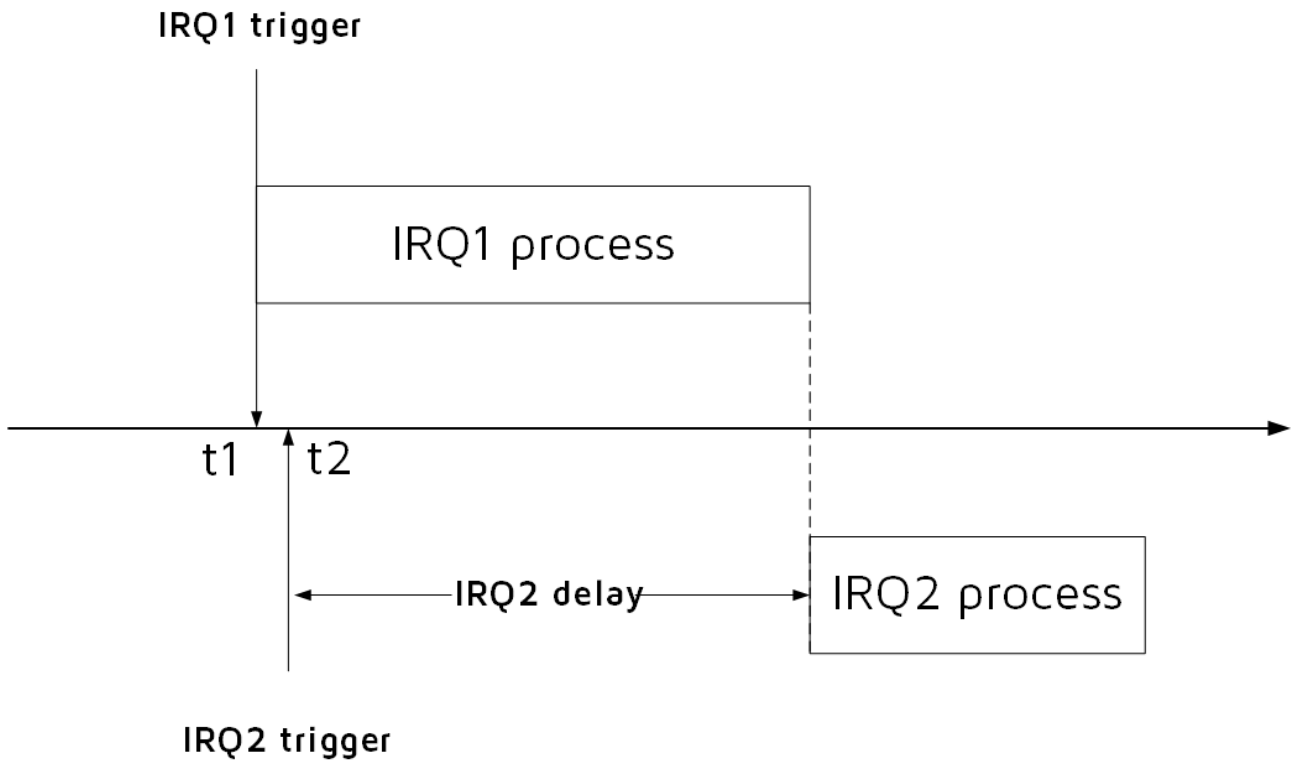


Figure 2.7: "IRQ delay"

In the BLE SDK, two system interrupts, system timer and RF, are used. If the user does not add new interrupts, there is no need to consider the timing of the two system interrupts; if the customer needs to add other interrupts (e.g. UART, PWM, etc.), the details to be considered are as follows.

- (1) For the two system interrupts system timer and RF in the SDK, the maximum possible execution time is 200us. This means that the customer added interrupts may not be able to respond in real time, and the theoretical maximum possible delay time is 200us.
- (2) The two system interrupts system timer and RF are for processing BLE tasks, due to the BLE timing is more strict, can not be delayed too long. Therefore, the processing time of the interrupts added by the customer should not be too long, and it is recommended to be within 50us. If the time is too long, there may be BLE timing synchronization errors, resulting in low efficiency of sending and receiving packets, high power consumption, BLE disconnection and other problems.

3 BLE Module

3.1 BLE SDK Software Architecture

3.1.1 Standard BLE SDK Architecture

Figure below shows standard BLE SDK software architecture compliant with BLE spec.

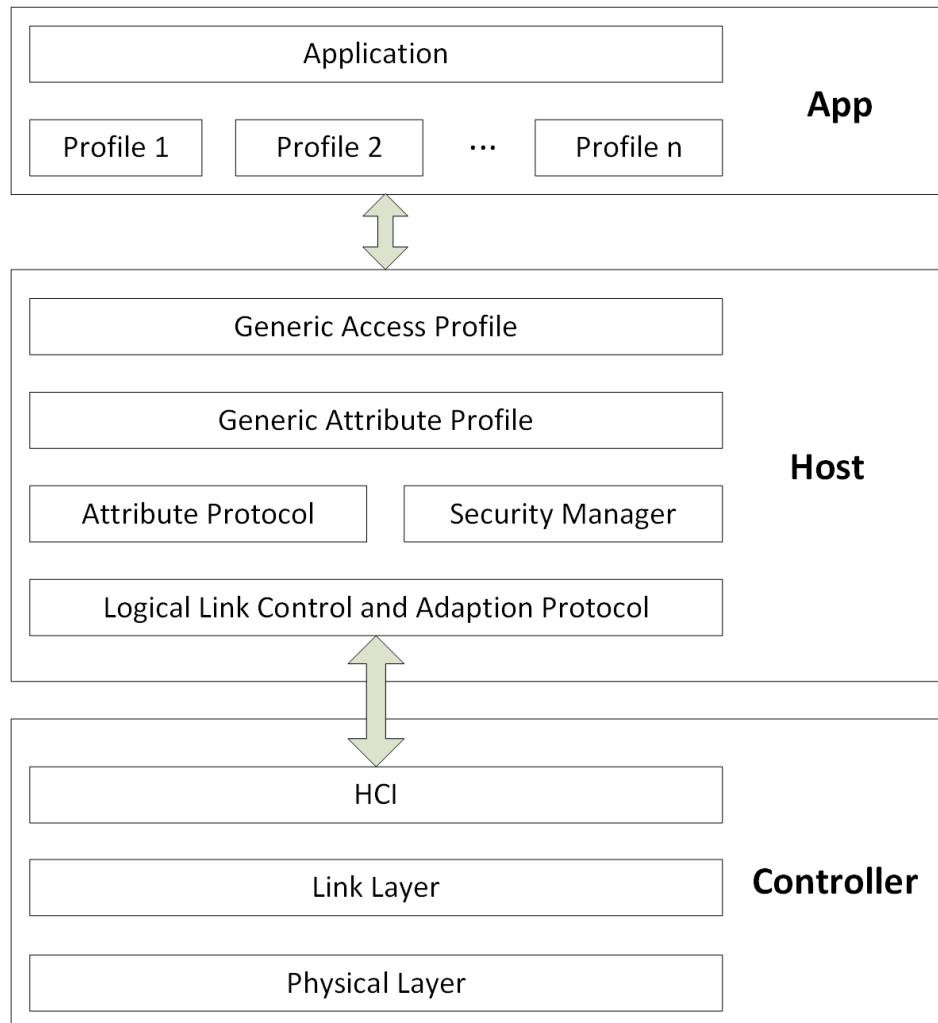


Figure 3.1: "BLE SDK software architecture"

As shown above, BLE protocol stack includes Host and Controller.

- As BLE bottom-layer protocol, the "Controller" contains Physical Layer (PHY) and Link Layer (LL). Host Controller Interface (HCI) is the sole communication interface for all data transfer between Controller and Host.
- As BLE upper-layer protocol, the "Host" contains protocols including Logic Link Control and Adaption Protocol (L2CAP), Attribute Protocol (ATT), Security Manager Protocol (SMP), as well as Profiles including Generic Access Profile (GAP) and Generic Attribute Profile (GATT).

- The “Application” (APP) layer contains user application codes and Profiles corresponding to various Services. User controls and accesses Host via “GAP”, while Host transfers data with Controller via “HCI”, as shown below.

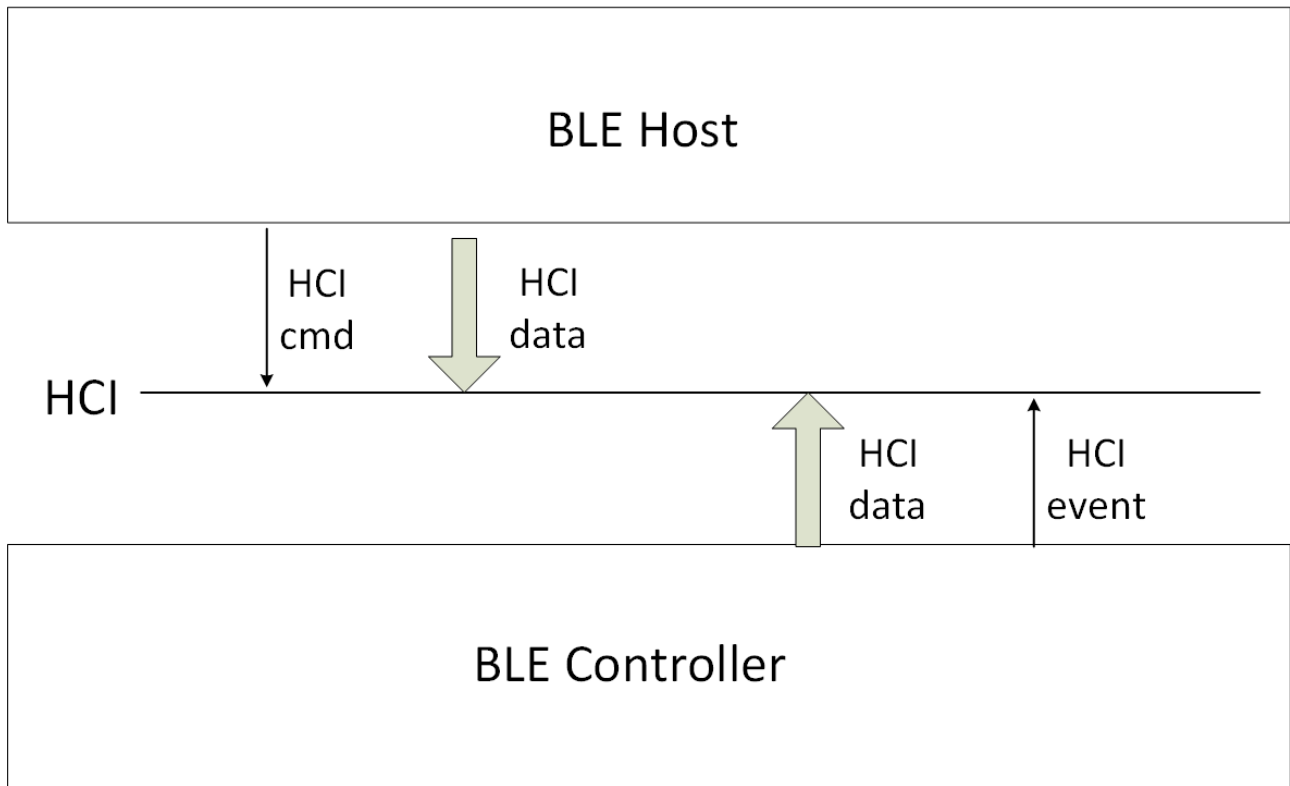


Figure 3.2: “HCI Data Transfer between Host and Controller”

- (1) BLE Host will use HCI cmd to operate and set Controller. Controller API corresponding to each HCI cmd will be introduced in this chapter.
- (2) Controller will report various HCI events to Host via HCI.
- (3) Host will send target data to Controller via HCI, while Controller will directly load data to Physical Layer for transfer.
- (4) When Controller receives RF data in Physical Layer, it will first check whether the data belong to Link Layer or Host, and then process correspondingly: If the data belong to LL, the data will be processed directly; if the data belong to Host, the data will be sent to Host via HCI.

3.1.2 Telink BLE SDK Architecture

3.1.2.1 Telink BLE controller

Telink BLE SDK supports standard BLE controllers, including HCI, PHY (Physical Layer) and LL (Link layer).

Telink BLE SDK includes five standard states of Link Layer (standby, advertising, scanning, initiating, connection), and both Slave role and Master role are supported in the connection state. In B85m BLE Single Connection SDK, Slave role and Master role are only single connection, that is, Link Layer can only maintain one connection, not multiple Slave/Master or Slave and Master at the same time.

The B85m hci in the SDK is a BLE slave controller, which needs to coordinate with another MCU running BLE Host to form a standard BLE Slave system, the architecture diagram is as follows.

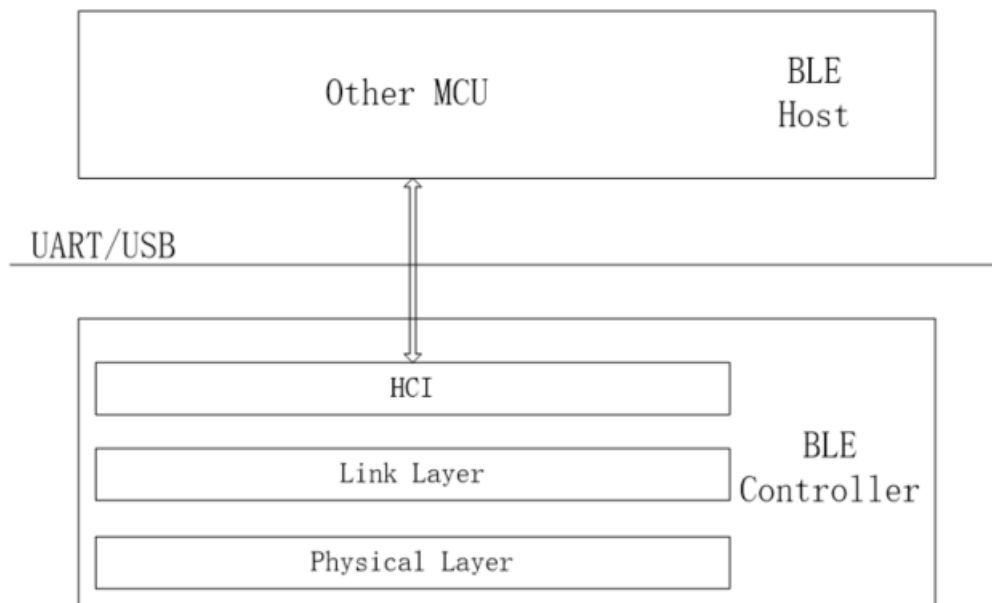


Figure 3.3: "Telink HCI architecture"

Link Layer connection status supports both Slave and Master of single connection, then B85m hci can actually be used as BLE master controller. However, actually for a BLE host running on more complex systems (such as Linux/Android), a single connection master controller can only connect to one device, which is almost meaningless, so the SDK does not put the initialization of the master role in the B85m hci.

3.1.2.2 Telink BLE Slave

Telink BLE SDK in BLE host fully supports stack of Slave; for the Master, it can not fully support, because SDP (service discovery) is too complex.

When user only needs to use standard BLE Slave, and Telink BLE SDK runs Host (Slave part) + standard Controller, the actual stack architecture will be simplified based on the standard architecture, so as to minimize system resource consumption of the whole SDK (including SRAM, running time, power consumption, and etc.). Following shows Telink BLE Slave architecture. In the SDK, B85m ble sample, B85m remote and B85m module are based on this architecture.

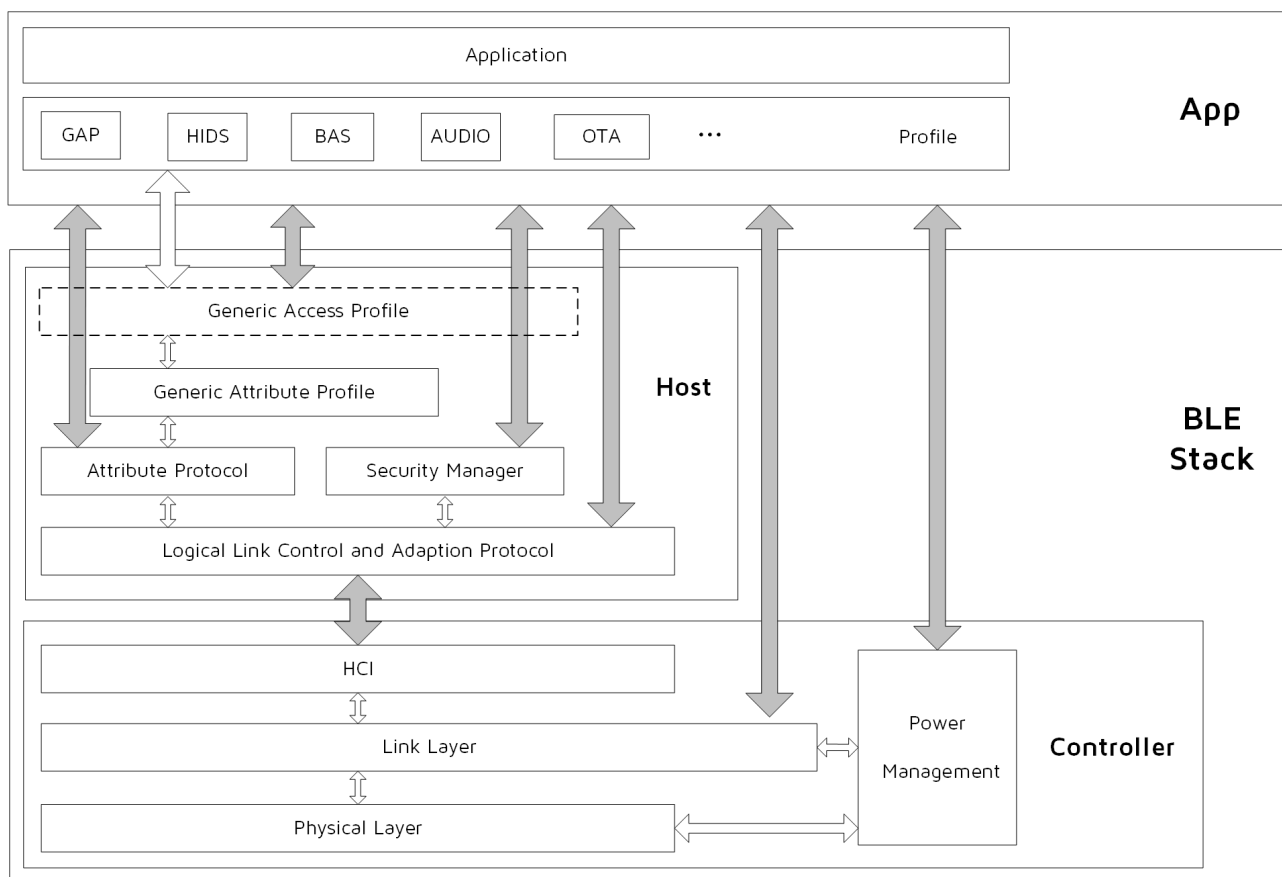


Figure 3.4: “Telink BLE Slave architecture”

In figure above, solid arrows indicate data transfer controllable via user APIs, while hollow arrows indicate data transfer within the protocol stack not involved in user.

Controller can still communicate with Host (L2CAP layer) via HCI; however, the HCI is no longer the sole interface, and the APP layer can directly exchange data with Link Layer of the Controller. Power Management (PM) Module is embedded in the Link Layer, and the application layer can invoke related PM interfaces to set power management.

Considering efficiency, data transfer between the APP layer and the Host is not controlled via GAP; the ATT, SMP and L2CAP can directly communicate with the APP layer via corresponding interface. However, the event of the Host should be communicated with the APP layer via the GAP layer.

Generic Attribute Profile (GATT) is implemented in the Host layer based on Attribute Protocol. Various Profiles and Services can be defined in the APP layer based on GATT. Basic Profiles including HIDS, BAS, AUDIO and OTA are provided in demo code of this BLE SDK.

Following sections explain each layer of the BLE stack according to the structure above, as well as user APIs for each layer.

Physical Layer is totally controlled by Link Layer, since it does not involve the application layer, it will not be covered in this document.

Though HCI still implements part of data transfer between Host and Controller, it is basically implemented by the protocol stack of Host and Controller with little involvement of the APP layer. User only needs to register HCI data callback handling function in the L2CAP layer.

3.1.2.3 Telink BLE master

The implementation of Telink BLE master is different from that of Slave. The SDK provides standard controller packed inside the library, but the app layer implements host and user's own application, as shown in the figure below.

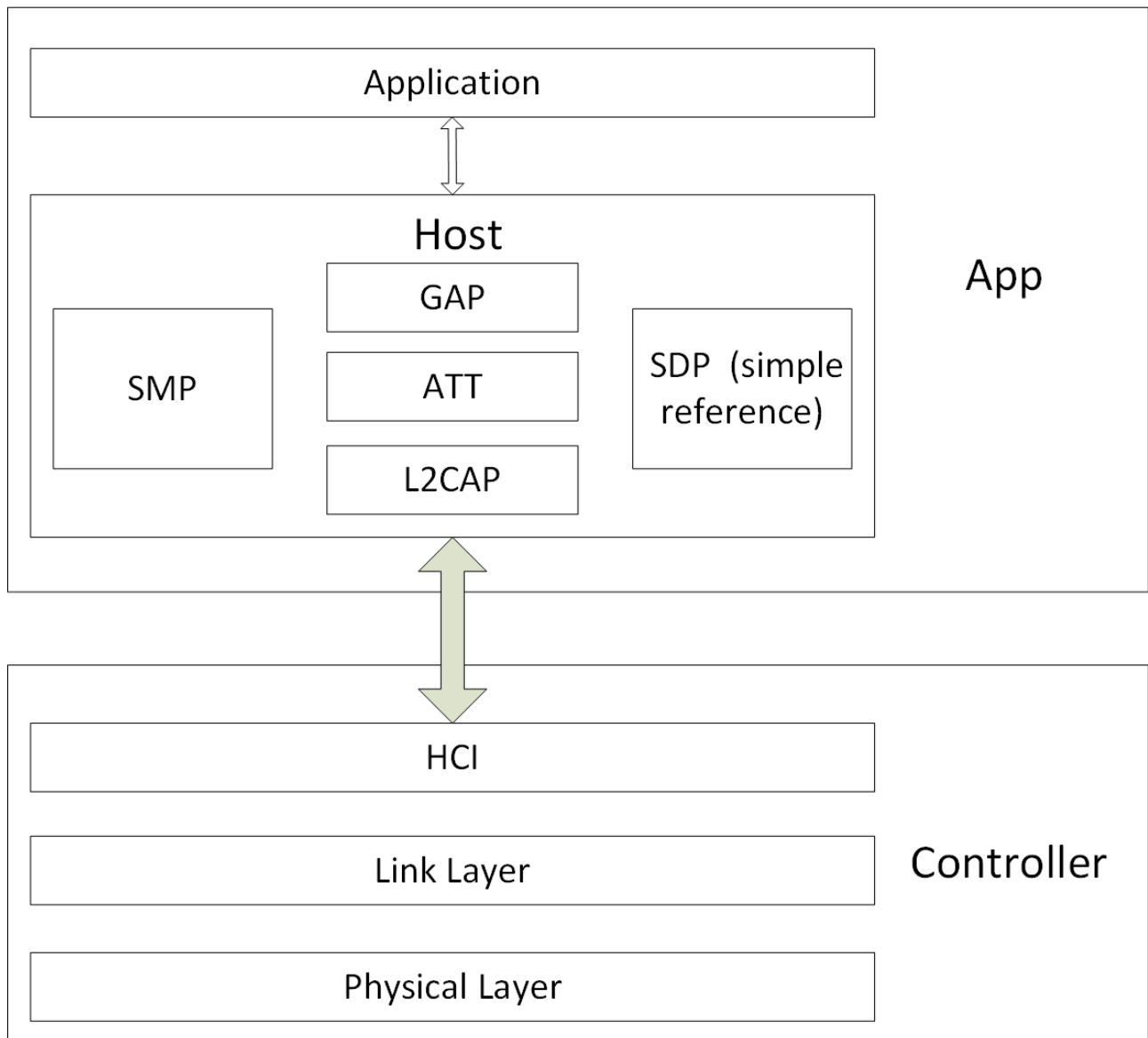


Figure 3.5: "Telink BLE master architecture"

In the B85m master kma dongle project of the SDK, the demo code is implemented based on this architecture, the host layer code is almost all implemented in the app layer. The SDK provides a variety of standard interfaces for users to complete these functions.

The App layer implements the standard l2cap, att and other processing, in the SMP part it only provides the most basic just work way. The B85m master kma dongle default SMP is disable, the user needs to open this macro to enable SMP. Due to the complexity of SMP implementation, the specific code implementation is still packed in the library, the app layer only needs to call the relevant interface, user searching

BLE_HOST_SMP_ENABLE can find all the code processing.

```
#define BLE_HOST_SMP_ENABLE          0
//1 for standard security management,
// 0 for telink referenced pairing&bonding(no security)
```

The SDP is the most complex part, Telink BLE master does not provide a standard SDP, only a simple reference is given to the B85m remote service discovery. The b85m master kma dongle default this simple reference SDP is on.

```
#define BLE_HOST_SIMPLE_SDP_ENABLE  1 //simple service discovery
```

The SDK provides standard interfaces for all service discovery related ATT operations, users can refer to B85m remote's service discovery to implement their own service discovery, or disable BLE_HOST_SIMPLE_SDP_ENABLE and use the agreed service ATT handle with the slave to achieve data access.

Telink BLE master does not support Power Management because the scanning and connection master role of Link Layer does not do suspend processing.

3.2 BLE Controller

3.2.1 BLE Controller Introduction

BLE Controller contains Physical Layer, Link Layer, HCI and Power Management.

Telink BLE SDK fully packs Physical Layer in the library (corresponding to c file of rf.h in driver file), and user does not need to learn about it. Power Management will be introduced in detail in section 4 Low Power Management (PM).

This section will focus on Link Layer, and also introduce HCI related interfaces to operate Link Layer and obtain data of Link Layer.

3.2.2 Link Layer State Machine

Figure below shows Link Layer state machine in BLE spec. Please refer to "Core_v5.0" (Vol 6/Part B/1.1 "LINK LAYER STATES") for more information.

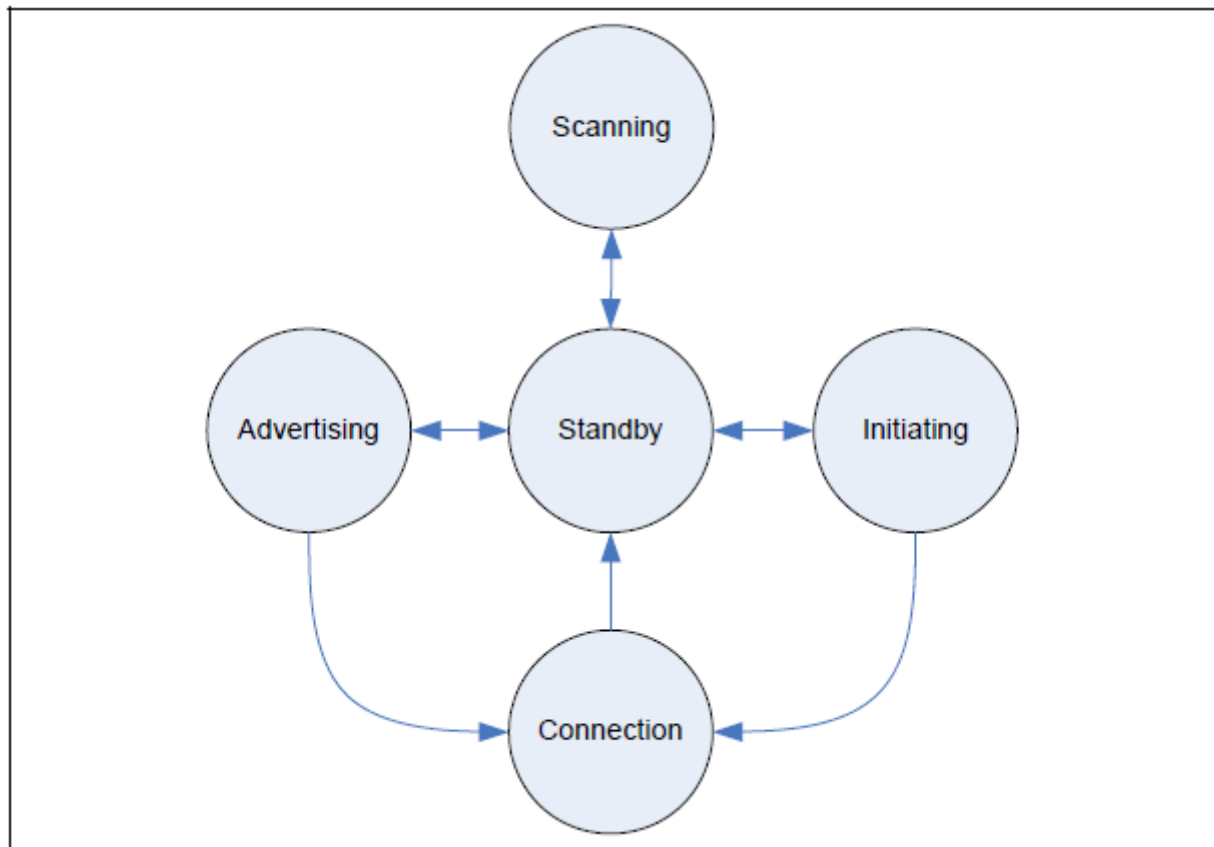


Figure 3.6: "Link Layer State Machine in BLE Spec"

Telink BLE SDK Link Layer state machine is shown as below.

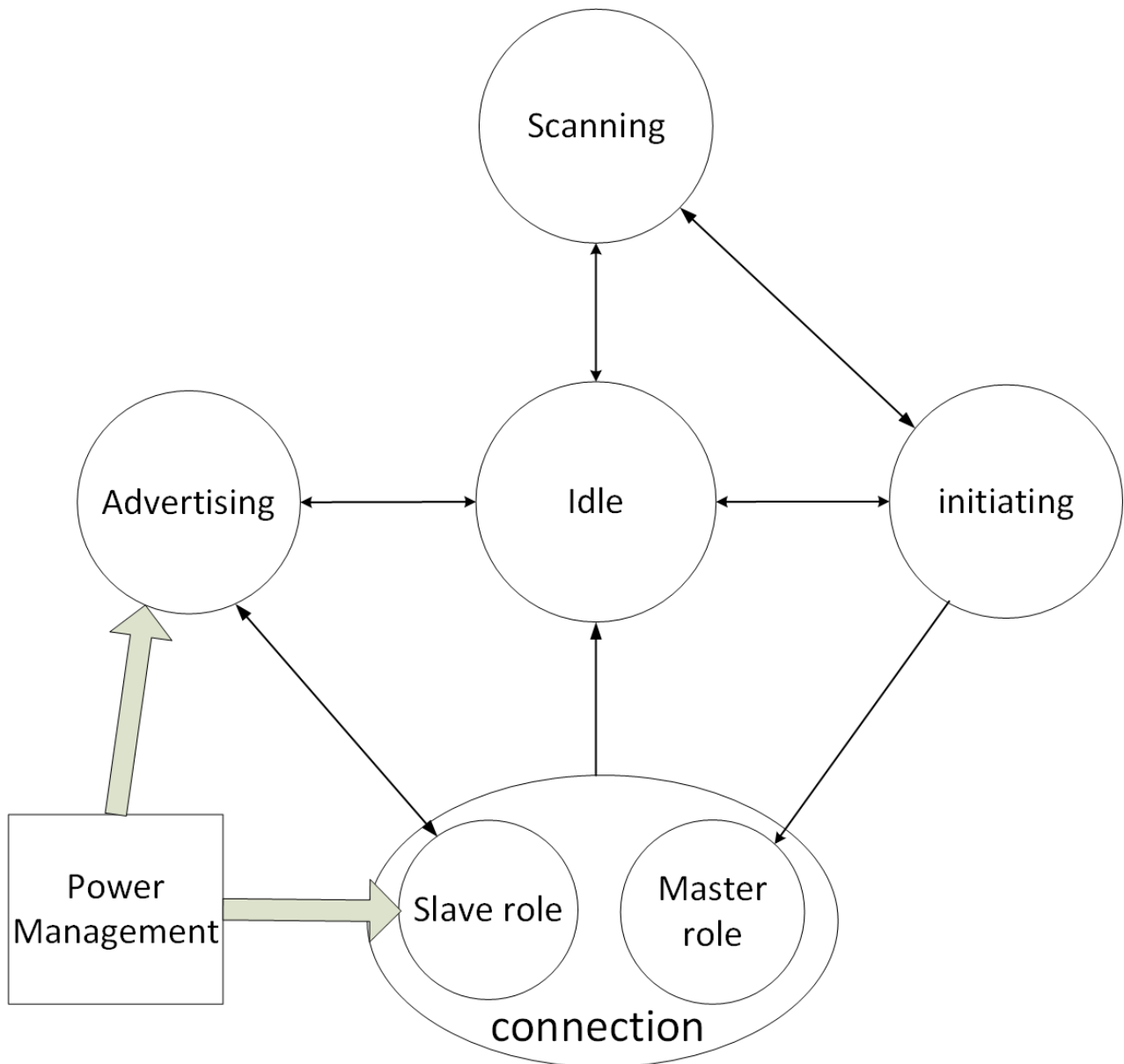


Figure 3.7: “Telink Link Layer State Machine”

Telink BLE SDK Link Layer state machine is consistent with BLE spec, and it contains five basic states: Idle (Standby), Scanning, Advertising, Initiating, and Connection. Connection state contains Slave Role and Master Role.

From the introduction of library earlier in the document, the current Slave Role and Master Role are both designed based on single connection, and the Slave Role is single connection by default; the Master Role is currently provided with single connection, but since it will provide a multi connection in the future, it is named Master role single connection here to distinguish it from the future Master Role multi connection.

In this document, Slave Role will be marked as “Conn state Slave role” or “ConnSlaveRole/Connection Slave Role”, or “ConnSlaveRole” in brief; while Master Role will be marked as “Conn state Master role” or “ConnMasterRole/Connection Master Role”, or “ConnMasterRole” in brief.

The “Power Management” in figure above is not a state of LL, but a functional module which indicates the

SDK only implements low power processing for Advertising and Connection Slave Role. If Idle state needs low power, user can invoke related APIs in the APP layer. For the other states, the SDK does not contain low power management, and user cannot implement low power in the APP layer.

Based on the five states above, corresponding state machine names are defined in the "stack/ble/ll/ll.h". "ConnSlaveRole" and "ConnMasterRole" correspond to state name "BLS_LINK_STATE_CONN".

```
#define BLS_LINK_STATE_IDLE      0
#define BLS_LINK_STATE_ADV      BIT(0)
#define BLS_LINK_STATE_SCAN     BIT(1)
#define BLS_LINK_STATE_INIT     BIT(2)
#define BLS_LINK_STATE_CONN     BIT(3)
```

Switch of Link Layer state machine is automatically implemented in BLE stack bottom layer. Therefore, user cannot modify state in APP layer, but can obtain current state by invoking the API below. The return value will be one of the five states.

```
u8      blc_ll_getCurrentState(void);
```

3.2.3 Link Layer State Machine Combined Application

3.2.3.1 Link Layer State Machine Initialization

Telink BLE SDK Link Layer fully supports all states; however, it's flexible in design. Each state can be assembled as a module; by default there's only the basic Idle module, and user needs to add modules and establish state machine combination for his application. For example, for BLE Slave application, user needs to add Advertising module and ConnSlaveRole, while the remaining Scanning/Initiating modules are not included so as to save code size and ram_code. The code of unused states won't be compiled.

The MCU initialization is mandatory and the API is as follows.

```
void      blc_ll_initBasicMCU (void);
```

The API below serves to add the basic Idle module. This API is also necessary for all BLE applications.

```
void      blc_ll_initStandby_module (u8 *public_adr);
```

The initialization APIs of the corresponding modules for several other states (Scanning, Alerting, Initiating, Slave Role, Master Role Single Connection) are as follows.

```
void      blc_ll_initAdvertising_module(u8 *public_adr);
void      blc_ll_initScanning_module(u8 *public_adr);
void      blc_ll_initInitiating_module(void);
void      blc_ll_initConnection_module(void);
void      blc_ll_initSlaveRole_module(void);
void      blc_ll_initMasterRoleSingleConn_module(void);
```


The real parameter `public_adr` in the above API is a pointer to the BLE public mac address.

The following API is used to initialize the module shared by master and slave.

```
void      blc_ll_initConnection_module(void);
```

User can use the above APIs to combine the Link Layer state machine. The following are some common combinations and corresponding application scenarios, but they are not limited to the following combinations.

3.2.3.2 Idle + Advertising

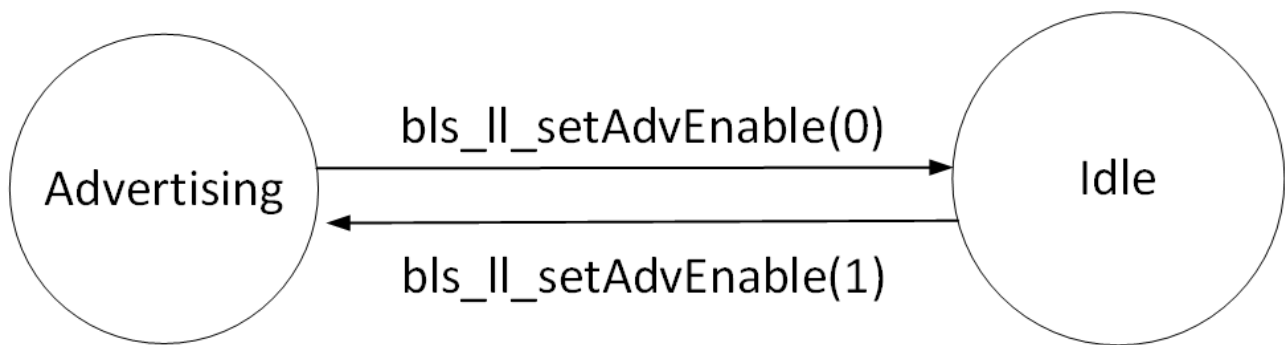


Figure 3.8: "Idle + Advertising"

As shown above, only Idle and Advertising module are initialized, and it applies to applications which use basic advertising function to advertise product information in single direction, e.g. beacon.

Following is module initialization code of Link Layer state machine.

```
u8  mac_public[6] = {.....};
blc_ll_initBasicMCU();
blc_ll_initStandby_module(mac_public);
blc_ll_initAdvertising_module(mac_public);
```

State switch of Idle and Advertising is implemented via the "bls_ll_setAdvEnable".

3.2.3.3 Idle + Scanning

As shown in the figure below, only the Idle and Scanning modules are initialized, and the most basic Scanning function is used to achieve the scanning and discovery of beacons and other product broadcast information.

The Link Layer state machine module initialization code is:

```
u8 mac_public[6] = {……};
blc_ll_initBasicMCU();
blc_ll_initStandby_module(tbl_mac);
blc_ll_initScanning_module( tbl_mac);
```

The switching of Idle and Scanning states is implemented by “blc_ll_setScanEnable”.

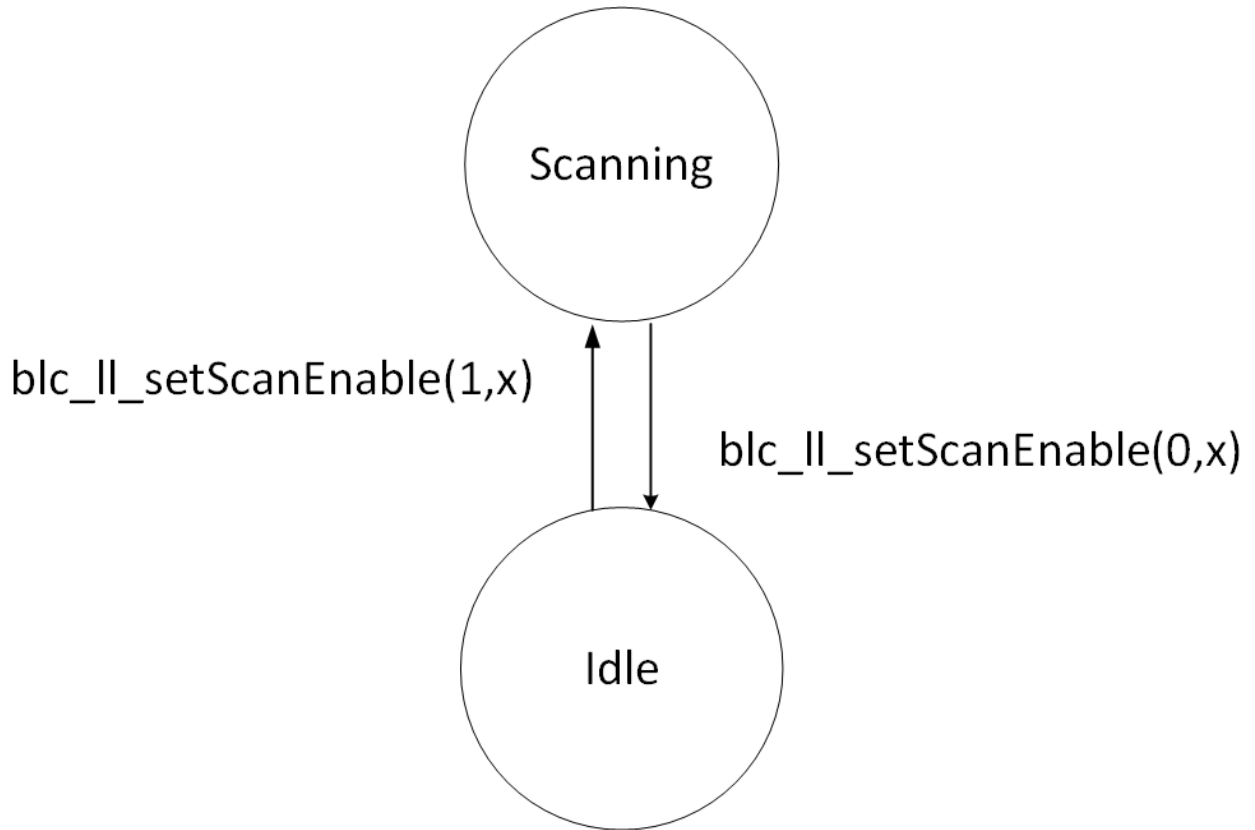


Figure 3.9: “Idle + Scanning”

3.2.3.4 Idle + Advertising + ConnSlaveRole

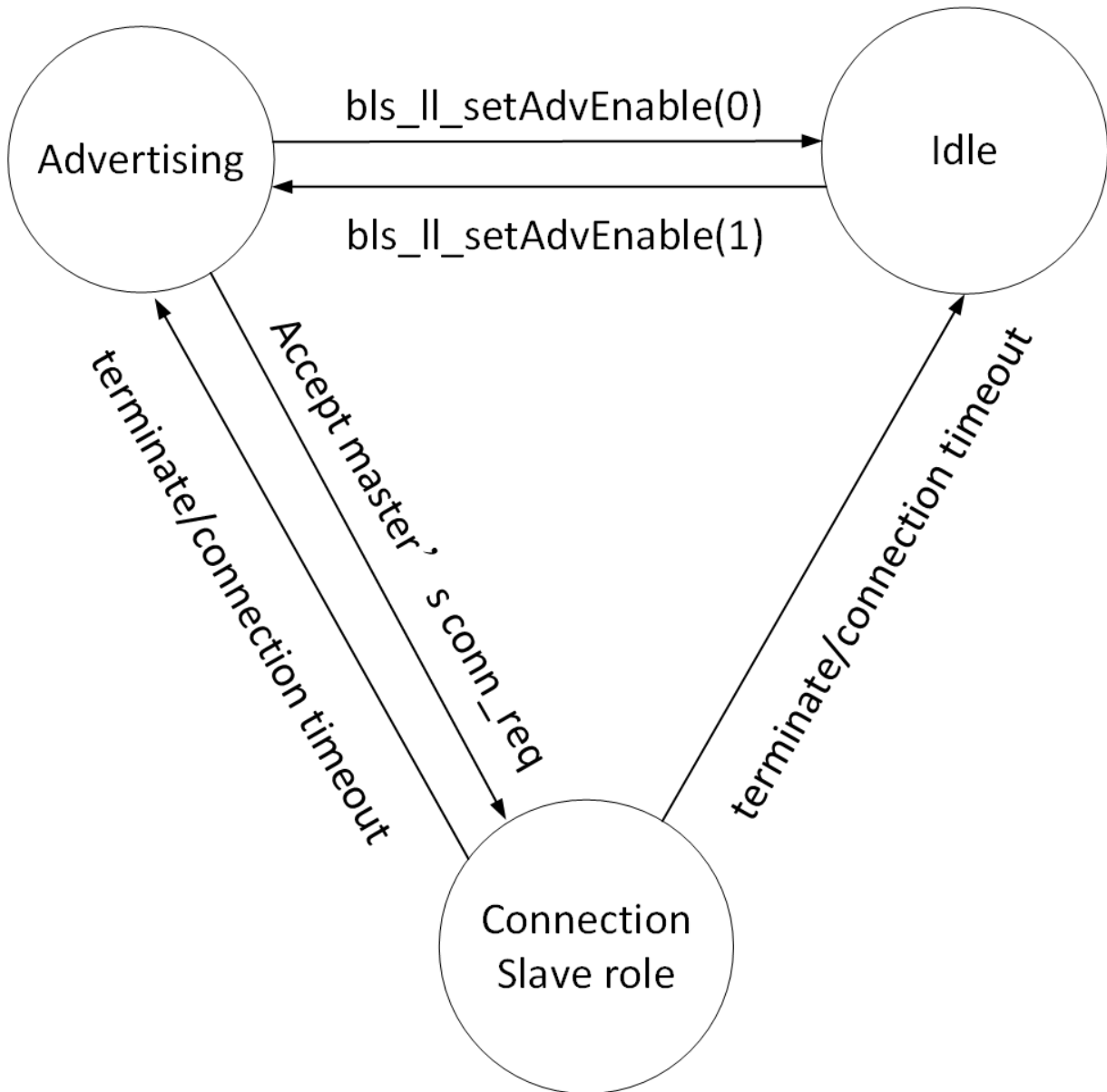


Figure 3.10: "BLE Slave LL State"

The figure above shows the Link Layer state machine combination for a basic BLE slave application. The b85m hci/b85m ble sample/b85m remote/b85m module in the SDK are all based on this state machine combination.

The Link Layer state machine module is initialized with the following code.

```

u8 mac_public[6] = {……};
blc_ll_initBasicMCU();
blc_ll_initStandby_module(mac_public);
blc_ll_initAdvertising_module(mac_public);
  
```

```
blc_ll_initConnection_module();  
blc_ll_initSlaveRole_module();
```

State switch in this combination is shown as below:

- (1) After power on, the MCU enters Idle state. In Idle state, when adv is enabled, Link Layer switches to Advertising state; when adv is disabled, it will return to Idle state.

The API "bls_ll_setAdvEnable" serves to enable/disable Adv.

After power on, Link Layer is in Idle state by default. Typically it's needed to enable Adv in the "user_init" so as to enter Advertising state.

- (2) When Link Layer is in Idle state, Physical Layer won't take any RF operation including packet transmission and reception.
- (3) When Link Layer is in Advertising state, advertising packets are transmitted in adv channels. Master will send connection request if it receives adv packet. After Link Layer receives this connection request, it will respond, establish connection and enter ConnSlaveRole.
- (4) When Link Layer is in ConnSlaveRole, it will return to Idle State or Advertising state in any of the following cases:
 - Master sends "terminate" command to Slave and requests disconnection. Slave will exit ConnSlaveRole after it receives this command.
 - By sending "terminate" command to Master, Slave actively terminates the connection and exits ConnSlaveRole.
 - If Slave fails to receive any packet due to Slave RF Rx abnormality or Master Tx abnormality until BLE connection supervision timeout is triggered, Slave will exit ConnSlaveRole.

When ConnSlaveRole of Link layer exits this state, it switches to a different state depending on whether Adv is enabled or not: if Adv is enabled, Link Layer goes back to Advertising state; if Adv is Disable, Link Layer goes back to Idle state. Whether Adv is Enable or Disable depends on the value set by the user when bls_ll_setAdvEnable was last called by the application layer.

3.2.3.5 Idle + Scanning + Initiating + ConnMasterRole

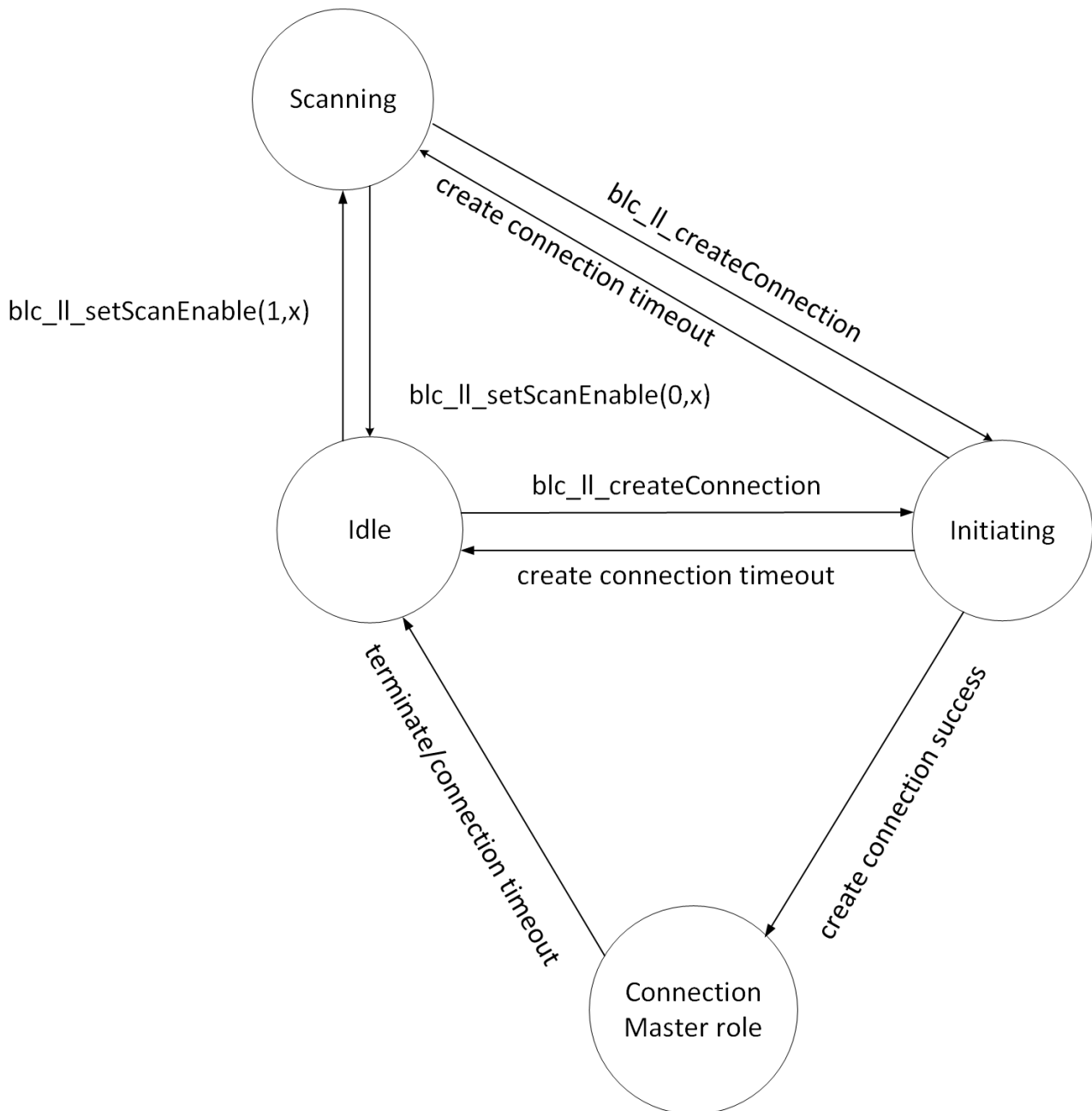


Figure 3.11: "BLE Master LL State"

The above figure shows the Link Layer state machine combination for a basic BLE master application, the B85m master kma dongle in the SDK is based on this state machine combination. The Link Layer state machine module initialization code is.

```

u8 mac_public [6] = {.....};
blc_ll_initBasicMCU();
blc_ll_initStandby_module(mac_public);
blc_ll_initScanning_module(mac_public);

```

```
blc_ll_initInitiating_module();  
blc_ll_initConnection_module();  
blc_ll_initMasterRoleSingleConn_module();
```

The state change under this state machine combination is described as follows.

1) After the MCU is powered on, it enters the Idle state; Scan is enabled in the Idle state, and the Link Layer is switched to the Scanning state; when Scan is disabled in the Scanning state, it returns to the Idle state.

The Scan Enable and Disable are controlled by API `blc_LL_setScanEnable`.

After power on, the Link layer is in Idle state by default. It is generally necessary to set Scan Enable inside `user_init` to enter Scanning state.

When Link Layer is in Scanning state, it will report the adv packet scanned to BLE host through event "HCI_SUB_EVT_LE_ADVERTISING_REPORT".

2) In Idle state and Scanning state, Link Layer can trigger API `blc_LL_createConnection` to enter Initiating state.

The `blc_LL_createConnection` specifies the mac address of one or more BLE devices that need to be connected. After the Link Layer enters the Initiating state, it continuously scans the specified BLE device, sends a connection request and enters the ConnMasterRole after receiving a correct adv packet that can be connected. If the initiating state does not scan the specified BLE device within a certain period of time and cannot initiate a connection, it will trigger a create connection timeout and revert to Idle State or Scanning state.

Note:

- Initiating state can enter from Idle state and Scanning state (B85m master kma dongle enters directly from Scanning state), create connection timeout and then return to the Idle State or Scanning state before create connection.

3) When the Link Layer is in ConnMasterRole, it returns to Idle State in one of three ways:

- a) The slave sends a terminate command to the master to disconnect. The master receives the terminate command and exits ConnMasterRole.
- b) The master sends a terminate command to the slave, actively disconnects, and exits ConnMasterRole.
- c) The master's RF packet receiving exception or the slave's packet sending exception causes the master to receive no packet for a long time, triggering the BLE's connection supervision timeout and exiting the ConnMasterRole.

If the Link layer's ConnMasterRole exits this state, it can only return to the Idle state. if it needs to continue scanning, it must use the API `blc_LL_setScanEnable` to set the Link Layer to enter the Scanning state again.

3.2.4 Link Layer Timing Sequence

In this section, Link Layer timing sequence in various states will be illustrated combining with `irq_handler` and `main_loop` of this BLE SDK.

```

_attribute_ram_code_ void irq_handler(void)
{
    .....
    irq_blt_sdk_handler ();
    .....
}

void main_loop (void)
{
    /////////////////////////////////// BLE entry ///////////////////////////////////
    blt_sdk_main_loop();
    /////////////////////////////////// UI entry ///////////////////////////////////
    .....
}

```

The “blt_sdk_main_loop” function at BLE entry serves to process data and events related to BLE protocol stack. UI entry is for user application code.

3.2.4.1 Timing Sequence in Idle State

When Link Layer is in Idle state, no task is processed in Link Layer and Physical Layer; the “blt_sdk_main_loop” function doesn’t act and won’t generate any interrupt, i.e. the whole timing sequence of main_loop is occupied by UI entry.

3.2.4.2 Timing Sequence in Advertising State

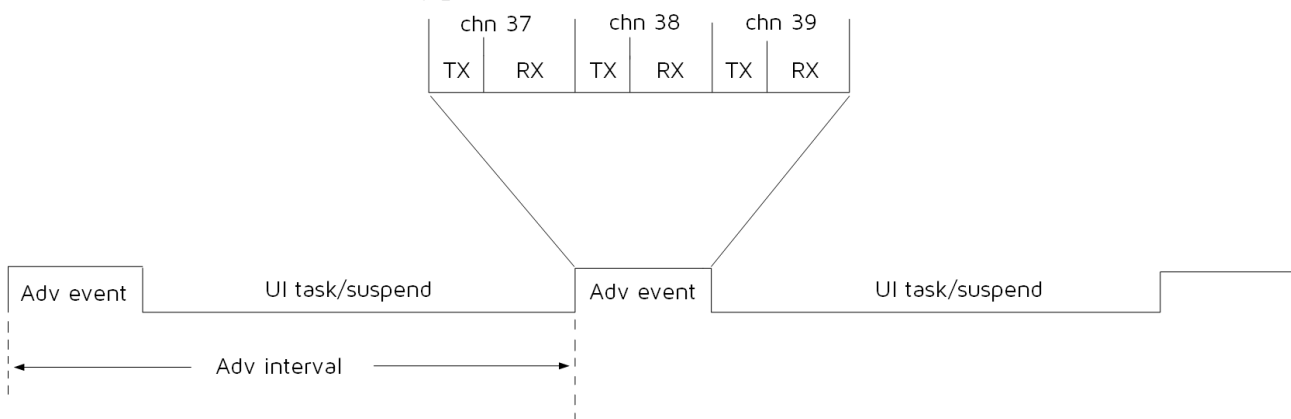


Figure 3.12: “Timing Sequence in Advertising State”

As shown in figure above, an Adv event is triggered by Link Layer during each adv interval. A typical Adv event with three active adv channels will send an advertising packet in channel 37, 38 and 39, respectively. After an adv packet is sent, Slave enters Rx state, and waits for response from Master:

- If Slave receives a scan request from Master, it will send a scan response to Master.

- If Slave receives a connect request from Master, it will establish BLE connection with Master and enter Connection state Slave Role.

Code of UI entry in `main_loop` is executed during UI task/suspend part in figure above. This duration can be used for UI task only, or MCU can enter sleep (suspend or deep sleep retention) for the redundant time to reduce power consumption.

In Advertising state, the `blt_sdk_main_loop` function does not need to process many tasks, and only some callback events related to Adv will be triggered, including `BLT_EV_FLAG_ADV_DURATION_TIMEOUT`, `BLT_EV_FLAG_SCAN_RSP`, `BLT_EV_FLAG_CONNECT`, etc.

3.2.4.3 Timing Sequence in Scanning State

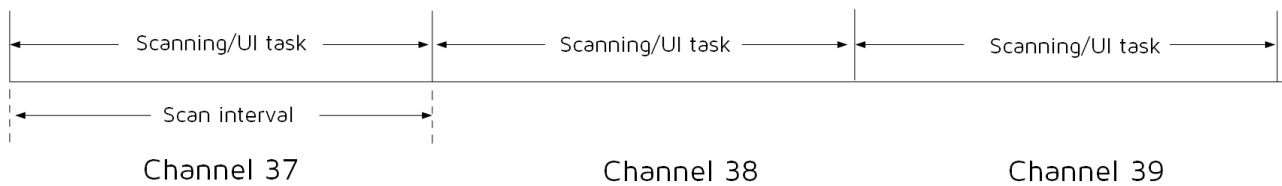


Figure 3.13: "Timing Sequence in Scanning State"

Scan interval is configured by the API `blc_ll_setScanParameter`. During a whole Scan interval, packet reception is implemented in one channel, and Scan window is not designed in the SDK. Therefore, the SDK won't process the setting of Scan window in the `blc_ll_setScanParameter`.

After the end of each Scan interval, it will switch to the next receiving channel, and enters next Scan interval. Channel switch action is triggered by interrupt, and it's executed in irq which takes very short time.

In Scanning interval, PHY Layer of Scan state is always in RX state, and it depends on MCU hardware to implement packet reception. Therefore, all timing in software are for UI task.

After correct BLE packet is received in Scan interval, the data are first buffered in software RX fifo (corresponding to `my_fifo_t blt_rxfifo` in code), and the `blt_sdk_main_loop` function will check whether there are data in software RX fifo. If correct adv data are discovered, the data will be reported to BLE Host via the event `HCI_SUB_EVT_LE_ADVERTISING_REPORT`.

3.2.4.4 Timing Sequence in Initiating State

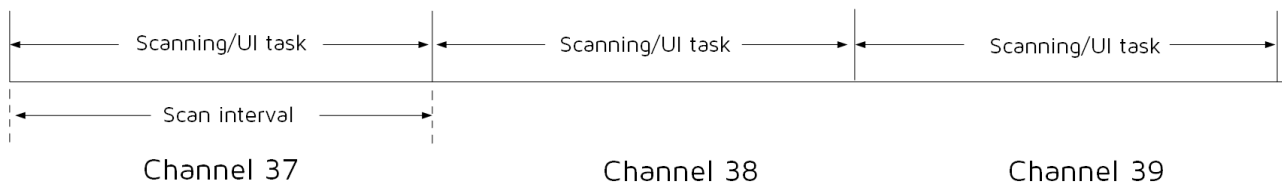


Figure 3.14: "Timing Sequence in Initiating State"

Timing sequence of Initiating state is similar to that of Scanning state, except that Scan interval is configured by the API `blc_ll_createConnection`. During a whole Scan interval, packet reception is implemented in one

channel, and Scan window is not designed in the SDK. Therefore, the SDK won't process the setting of Scan window in the "blc_ll_createConnection".

After the end of each Scan interval, it will switch to the next listening channel, and start a new Scan interval. Channel switch action is triggered by interrupt, and it's executed in irq which takes very short time.

In Scanning state, BLE Controller will report the received adv packet to BLE Host; however, in Initiating state, adv won't be reported to BLE Host, and it only scans for the device specified by the "blc_ll_createConnection". If the specific device is scanned, it will send connection_request and establish connection, then Link Layer enters ConnMasterRole.

3.2.4.5 Timing Sequence in Conn State Slave Role

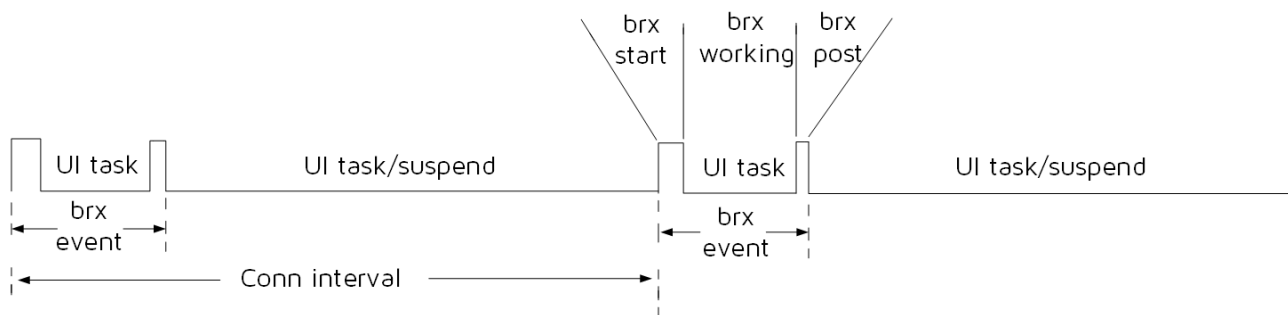


Figure 3.15: "Timing Sequence in Conn State Slave Role"

As shown in the above figure, each conn interval starts with a brx event, i.e. transfer process of BLE RF packets by Link Layer: PHY enters Rx state, and an ack packet will be sent to respond to each received data packet from Master. If there is more data, then continue to receive master packets and reply, this process is called brx event for short.

In this BLE SDK, each brx process consists of three phases according to the assignment of hardware and software.

(1) brx start phase

When Master is about to send packet, an interrupt is triggered by system tick irq to enter brx start phase. During this interrupt, MCU sets BLE state machine of PHY to enter brx state, hardware in bottom layer prepares for packet transfer, and then MCU exits from the interrupt irq.

(2) brx working phase

After brx start phase ends and MCU exits from irq, hardware in bottom layer enters Rx state first and waits for packet from Master. During the brx working phase, all packet reception and transmission are implemented automatically without involvement of software.

(3) brx post phase

After packet transfer is finished, the brx working phase is completed. System tick irq triggeres an interrupt to switch to the brx post phase. During this phase, protocol stack will process BLE data and timing sequence according to packet transfer in the brx working phase.

During the three phases, brx start and brx post are implemented in interrupt, while brx working phase does not need the involvement of software, and UI task can be executed normally (Note that during brx working phase, UI task can be executed in the time slots except RX, TX, and System Timer interrupt handler). During the brx working phase, MCU can't enter sleep (suspend or deep sleep retention) since hardware needs to transfer packets.

Within each conn interval, the duration except for brx event can be used for UI task only, or MCU can enter sleep (suspend or deep sleep retention) for the redundant time to reduce power consumption.

In the ConnSlaveRole, the "blt_sdk_main_loop" needs to process the data received during the brx process. During the brx working phase, the data packet received from Master will be copied out during RX interrupt irq handler; these data won't be processed immediately, but buffered in software RX fifo (corresponding to my_fifo_t blt_rxfifo in code). The "blt_sdk_main_loop" function will check whether there are data in software RX fifo, and process the detected data packet correspondingly:

The processing of packets by blt_sdk_main_loop includes:

- (1) Decryption of data packet
- (2) Parsing of data packet

If the parsed data belongs to the control command sent by Master to Link Layer, this command will be executed immediately; if it's the data sent by Master to Host layer, the data will be transferred to L2CAP layer via HCI interface.

3.2.4.6 Timing Sequence in Conn State Master Role

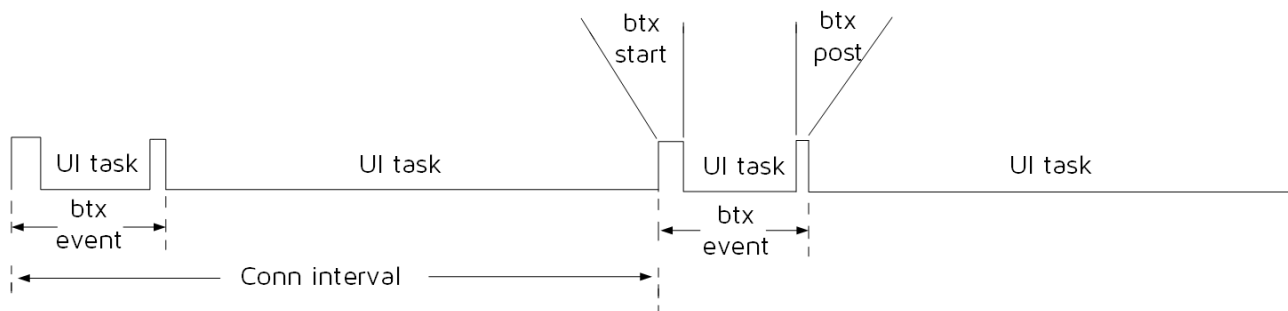


Figure 3.16: "Timing Sequence in Conn Master Role"

The ConnMasterRole timing sequence is shown above. At the beginning of each conn interval, the Link Layer performs a BLE RF packet sending and receiving process: first the PHY enters the packet sending state, sends a packet to the slave and then waits for the other party's ack packet, if there is more data, it continues to send packets to the slave, this process is referred to as brx event.

In this BLE SDK, each brx process consists of three phases according to the assignment of hardware and software.

1) brx start phase

When the time for master to send packets is approaching, it will be triggered by system tick irq to enter the brx start phase, in which the MCU sets the BLE state machine of the PHY to enter the brx state, and the bottom layer hardware prepares to send and receive packets, and then exits the interrupt irq.

2) btx working phase

After btx start, the MCU exits irq and the bottom layer hardware enters the transmitting state and does all the work of sending and receiving packets automatically, without any software involvement, this process is called the btx working phase.

3) btx post phase

After the packet is sent and received, btx working is finished and the system tick irq triggers the btx post phase. This phase is mainly for the protocol stack to process some data and timing of the BLE according to the btx working phase.

The btx start and btx post in the above three phases are both interrupted, while the btx working phase requires no software involvement and the UI task can be executed normally at this point.

At ConnMasterRole, blt_sdk_main_loop needs to process the data received by the btx process. The btx working process actually copies the slave packets received by the hardware during the RX receive interrupt irq processing, this data is not immediately processed in real time, but cached in the software RX fifo. The blt_sdk_main_loop function will check if there is data in the software RX fifo and process it as soon as it is available.

The processing of packets by blt_sdk_main_loop includes:

- 1) Decryption of data packet
- 2) Parsing of data packet

If the parsed data is found to belong to a control command sent by the slave to the Link Layer, the command will be executed immediately. If it is data sent by the master to the Host Layer, the data will be dropped to the L2CAP layer for processing through the HCI interface.

3.2.4.7 Timing Protect for Conn State Slave role

ConnSlaveRole, each interval requires a send/receive packet event, which is the Brx Event above. In the B85m SDK, the Brx Event is triggered entirely by interrupts, so the MCU main system interrupt needs to be turned on all the time. If the user is in the Conn state for a long time and has to turn off the main system interrupt (e.g. to erase the Flash), the Brx Event will be stopped and the BLE timing will soon be messed up and eventually the connection will be disconnected.

In this situation, the SDK provides a protection mechanism that allows the user to disable the Brx Event without breaking the BLE timing, and the user needs to strictly follow this mechanism. The relevant API is as follows.

```
int    bls_ll_requestConnBrxEventDisable(void);
void    bls_ll_disableConnBrxEvent(void);
void    bls_ll_restoreConnBrxEvent(void);
```

Call bls_ll_requestConnBrxEventDisable to request that the Brx Event be switched off.

1) If the return value of this API is 0, it means that the user's application is not currently accepted, i.e. the Brx Event cannot be stopped at this time. During the Brx working phase at Conn state, the application cannot be accepted and the return value is 0. It must wait until the end of a full Brx Event to accept the application for the remaining UI task/suspend time.

2) This API returns a non-zero value to indicate that the request can be accepted and the value returned is the time in ms allowed to stop the Brx Event. There are three cases for this event value:

- If the current Link Layer is Alerting state or Idle state, the return value is 0xffff, that is, there is no Brx Event, and the user is allowed to turn off the system interrupt for any length of time.
- If the current Conn state receives an update map or update connection parameter from the master and has not yet reached the update time point, the return time is the update time point minus the current time. In other words, the time to stop the Brx Event cannot exceed the update time, otherwise all the packets will not be received and the connection will be disconnected.
- If the current state is Conn state and there is no update request from master, the return value is half of the current connection supervision timeout value. For example, if the current timeout is 1s, the return value is 500ms.

The user calls the above API to request to disable the Brx Event, and if the return value corresponds to enough time (ms) for the task to run itself, the task can be performed. Before the task is executed, API `bls_ll_disableConnBrxEvent` is called to disable the Brx Event. After the task is finished, API `bls_ll_restoreConnBrxEvent` is called to re-enable the Brx Event and repair the BLE timing.

The reference usage is as follows. Where the specific time is judged by the actual time of tested task.

```

7         if(bls_ll_requestConnBrxEventDisable() > 300)
8         {
9
10            bls_ll_disableConnBrxEvent();
11
12        #if 0 //test 1
13            irq_disable();
14            DBG_CHN3_HIGH;
15            sleep_us(287*1000);
16            DBG_CHN3_LOW;
17            irq_enable();
18        #else //test 2
19            DBG_CHN3_HIGH;
20            flash_erase_sector(0x40000);
21            DBG_CHN3_LOW;
22        #endif
23
24            bls_ll_restoreConnBrxEvent();
25
26        }
27    }

```

Figure 3.17: "Timing of Scanning in Advertising state"

3.2.5 Link Layer State Machine Extension

The above BLE Link Layer state machine and working timings introduce the most basic states, which can satisfy the basic applications such as BLE slave/master. However, in view of the special applications that

users may have (e.g. advertising during the Conn state Slave role), the Telink BLE SDK adds some special extensions to the Link Layer state machine, which are described in detail below.

3.2.5.1 Scanning in Advertising state

When Link Layer is in Advertising state, the Scanning feature can be added.

The API to add Scanning feature:

```
ble_sts_t    blc_ll_addScanningInAdvState(void);
```

The API to remove Scanning feature:

```
ble_sts_t    blc_ll_removeScanningFromAdvState(void);
```

For the above two API, the return value of ble_sts_t type are both BLE_SUCCESS.

Combining the timing sequence of the Advertising state and Scanning state, the timing sequence is as follows when the Scanning feature is added to the Advertising state.

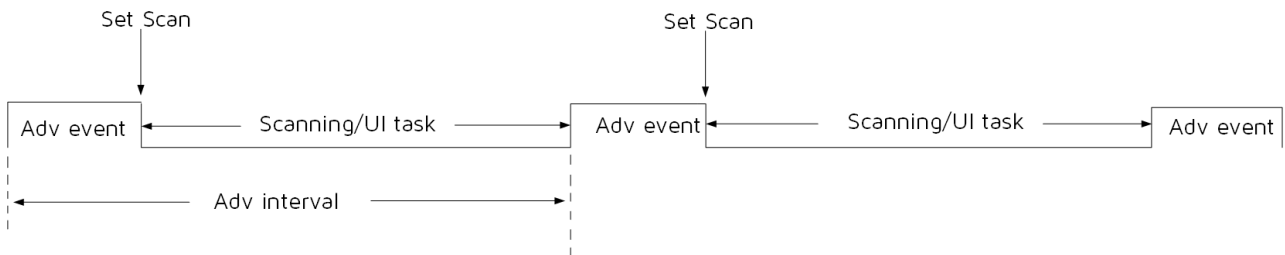


Figure 3.18: "Timing of Scanning in Advertising state"

The current Link Layer is still in the Advertising state (BLS_LINK_STATE_ADV) and the remaining time in each Adv interval, excluding the Adv event, is used for Scanning.

At each Set Scan, it is determined whether the current time exceeds a Scan interval (set by blc_ll_setScanParameter) since the last Set Scan, and if so, the Scan channel is switched (channel 37/38/39).

For the usage of Scanning in Advertising state, please refer to the TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE in the B85m_feature_test.

3.2.5.2 Scanning in ConnSlaveRole

When the Link Layer is in ConnSlaveRole, the Scanning feature can be added.

The API to add Scanning feature:

```
ble_sts_t    blc_ll_addScanningInConnSlaveRole(void);
```

The API to remove Scanning feature:

```
ble_sts_t    blc_ll_removeScanningFromConnSlaveRole(void);
```

For the above two API, the return value of ble_sts_t type are both BLE_SUCCESS.

Combining the timing sequence of Scanning state and ConnSlaveRole, when the Scanning feature is added to ConnSlaveRole, the timing sequence is as follows.

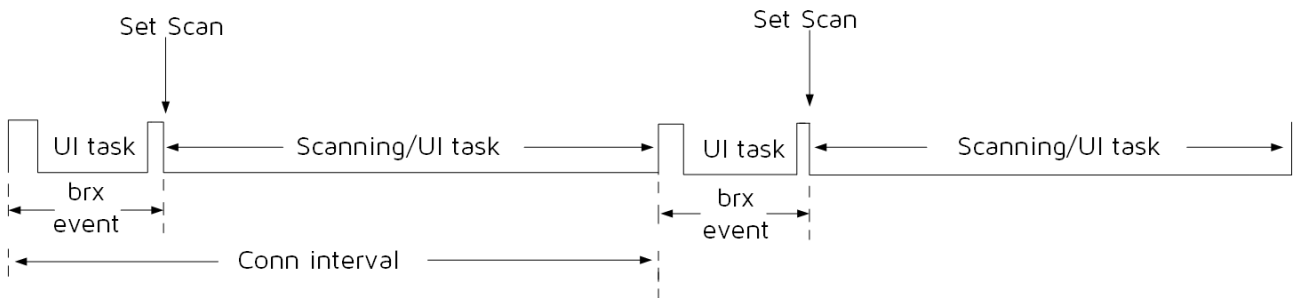


Figure 3.19: "Timing of Scanning in ConnSlaveRole"

The current Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN) and the remaining time in each Conn interval, excluding the brx event, is used for Scanning.

At each Set Scan, it is determined whether the current time exceeds a Scan interval (set by blc_ll_setScanParameter) since the last Set Scan, and if so, the Scan channel is switched (channel 37/38/39).

For the usage of Scanning in ConnSlaveRole, please refer to the TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE in B85m_feature_test.

3.2.5.3 Advertising in ConnSlaveRole

When the Link Layer is in ConnSlaveRole, the Advertising feature can be added.

The API to add Advertising feature:

```
ble_sts_t    blc_ll_addAdvertisingInConnSlaveRole(void);
```

The API to remove Advertising feature:

```
ble_sts_t    blc_ll_removeAdvertisingFromConnSlaveRole(void);
```

For the above two API, the return value of ble_sts_t type are both BLE_SUCCESS.

Combining the timing sequence of Advertising and ConnSlaveRole, when the Advertising feature is added to ConnSlaveRole, the timing sequence is as follows.

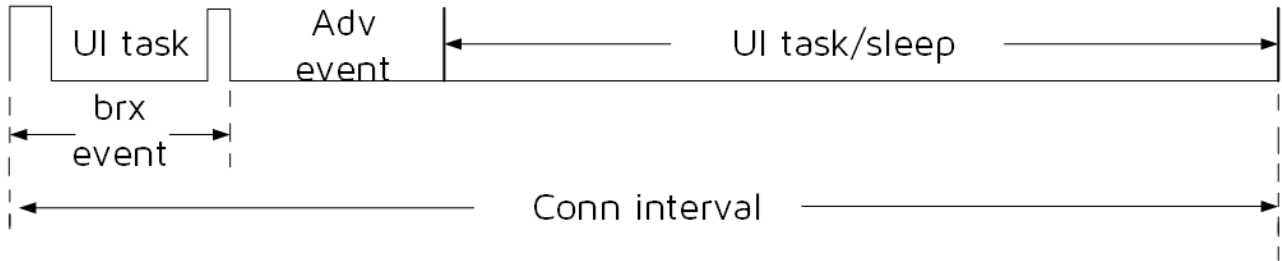


Figure 3.20: “Timing of Advertising in ConnSlaveRole”

The current Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN) and executes an adv event immediately after the brx event in each Conn interval, and then leaves the rest of the time for the UI task or goes into sleep (suspend/ deepsleep retention) to save power.

For the usage of Advertising in ConnSlaveRole, please refer to the TEST_ADVERTISING_IN_CONN_SLAVE_ROLE in B85m_feature_test.

3.2.5.4 Advertising and Scanning in ConnSlaveRole

Combined with the use of Scanning in ConnSlaveRole and Advertising in ConnSlaveRole above, it is possible to add both Scanning and Advertising to ConnSlaveRole. The timing sequence is as follows.

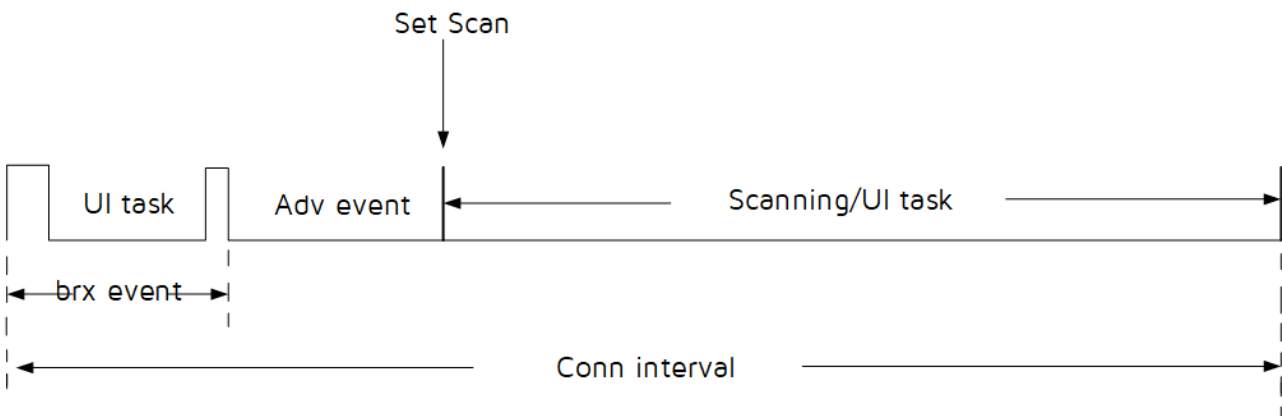


Figure 3.21: “Timing of Advertising and Scanning in ConnSlaveRole”

The current Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN) and an adv event is executed immediately after the brx event in each Conn interval, and then the rest of the time is used for Scanning.

At each Set Scan, it is determined whether the current time exceeds a Scan interval (set by blc_ll_setScanParameter) since the last Set Scan, and if so, the Scan channel is switched (channel 37/38/39).

For the usage of Advertising and Scanning in ConnSlaveRole, please refer to the TEST_ADVERTISING_SCANNING_IN_CONN_SLAVE_ROLE in B85m_feature_test.

3.2.6 Link Layer TX fifo & RX fifo

All data from the application layer and the BLE Host eventually needs to be sent through the Link Layer of the Controller to complete the RF data. A BLE TX fifo is designed in the Link Layer, which can be used to cache the incoming data and to send the data after the brx/btx has started.

All data received from the peer device during Link Layer brx/btx is first stored in a BLE RX fifo before being uploaded to the BLE Host or application layer for processing.

The BLE TX fifo and BLE RX fifo for the Slave role and Master role are handled in the same way. Both the BLE TX fifo and BLE RX fifo are defined at the application layer.

```
MYFIFO_INIT(blt_rxfifo, 64, 8);  
MYFIFO_INIT(blt_txfifo, 40, 16);
```

The RX fifo size is 64 by default and the TX fifo size is 40 by default, and these two sizes are not allowed to be modified unless a data length extension is required.

Both TX fifo number and RX fifo number must be set to the power of 2, i.e. 2, 4, 8, 16, etc. Users can modify them slightly to suit their needs.

RX fifo number is 8 by default, which is a reasonable value and can ensure that the bottom layer of the Link Layer can buffer up to 8 packets. If the setting is too large, it will take up too much SRAM. If the setting is too small, there may be a risk of data overwriting: in the brx event, the Link Layer is likely to work under more data (MD) mode on an interval, and continue to receive multiple packets, if you set 4, it is likely that there will be five or six packets in an interval (such as OTA, playing master voice data, etc.), and the upper layer's response to these data is too long to process due to the long decryption time, then it is possible some data is overflowed.

Here is an example of RX overflow, we have the following assumptions:

- a) The number of RX fifo is 8;
- b) Before brx_event(n) is turned on, the read and write pointers of RX fifo are 0 and 2 respectively;
- c) In the brx_event(n) and brx_event(n+1) stages, the main_loop has task blockage, and the RX fifo is not taken in time;
- d) Both brx_event stages are multi-packet situations.

From the description in the "Conn state Slave role timing" section above, we know that the BLE data packets received in the brx_working stage will only be copied to the RX fifo (RX fifo write pointer++), and the RX fifo data is actually taken out for processing. In the main_loop stage (RX fifo read pointer++), we can see that the sixth data will cover the read pointer 0 area. It should be noted here that the UI task time slot in the brx working stage is the time except for interrupt processing such as RX, TX, and system timer.

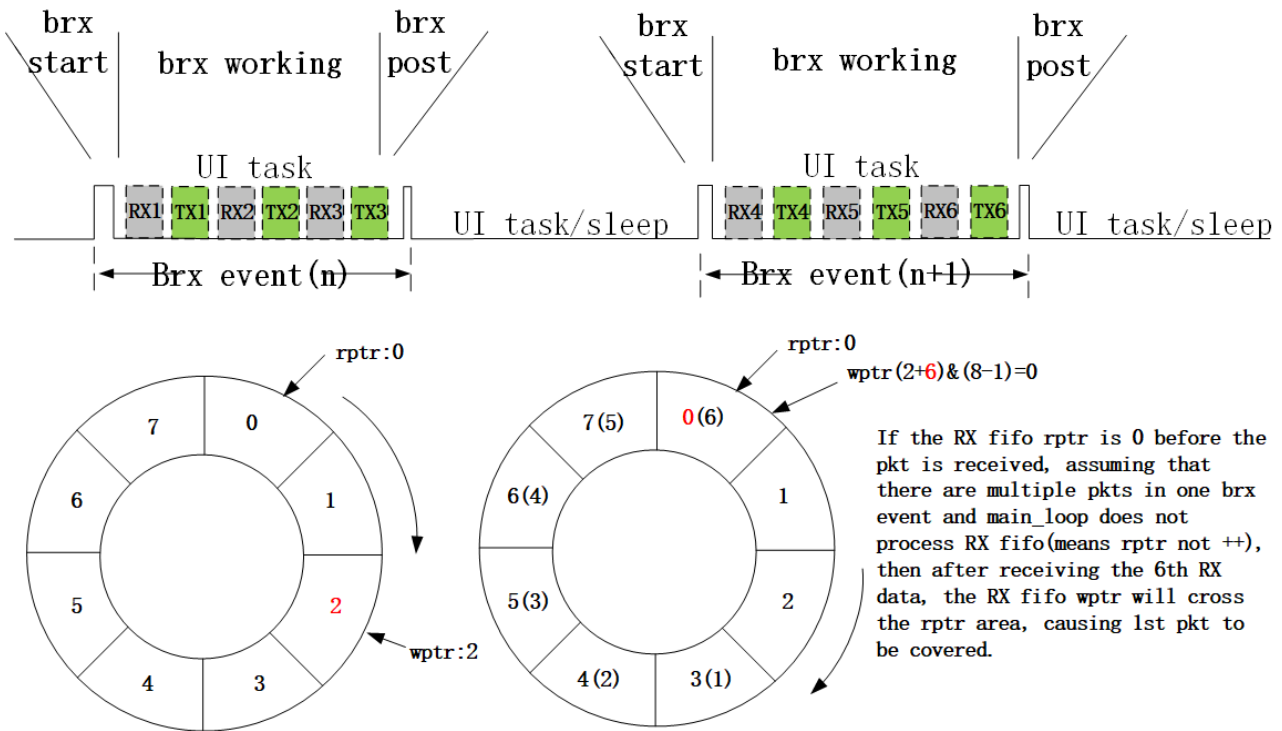


Figure 3.22: "RX overflow case 1"

Relative to the extreme case above with long task blockade duration due to one connection interval, the case below is more likely to occur: During one brx_event, since Master writes multiple packets (e.g. 7/8 packets) into Slave, Slave fails to process the received data in time. As shown below, the rptr (read pointer) is increased by two, but the wptr (write pointer) is also increased by eight, which thus causes data overflow.

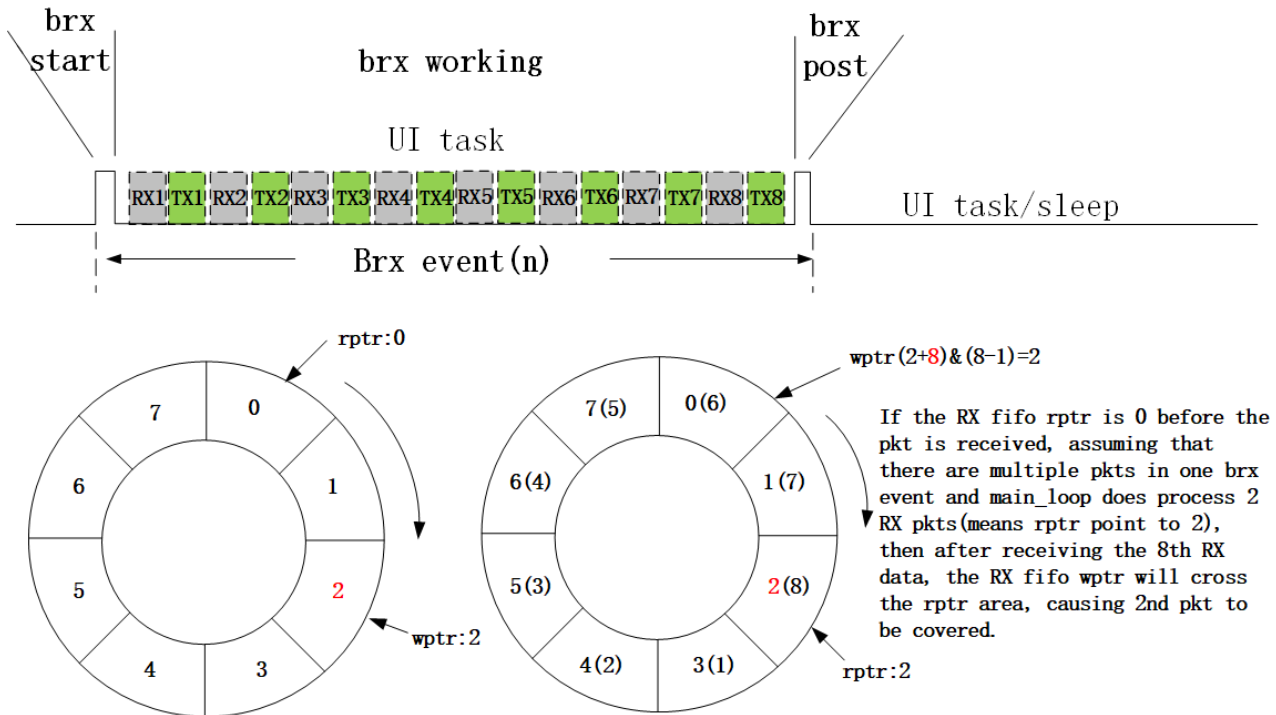


Figure 3.23: "RX overflow case 2"

Once there is a data loss problem caused by overflow, for the encryption system, there will be a MIC failure disconnection problem. (For old SDK, as brx event Rx IRQ will fill data to Rx fifo but not do data overflow check, if main_loop is too slow to process RX fifo it will lead to overflow problem, so when using old SDK, user need to pay more attention to this risk, avoid master to send too much data on one connection interval, pay attention to user UI that the task processing time is as short as possible to avoid blocking problems.)

Rx overflow checks have now been added to the SDK. Check whether the current RX fifo write pointer and read pointer difference is greater than the number of Rx fifo in brx/btx event Rx IRQ. Once the Rx fifo is found to be full, the RF will not ACK the other party. BLE protocol Data retransmission will be ensured. In addition, the SDK also provides the Rx overflow callback function to notify users. This callback will be introduced in the chapter "Telink defined event" later in the document.

Similarly, if there may be more than 8 valid packets in an interval, the default 8 is not enough.

The TX fifo number is 16 by default, which is able to handle the larger data volume of the voice remote control function. Users can modify it to 8 if they do not use such a large fifo.

If set too large (e.g. 32) it will take up too much sram.

In the TX fifo, the SDK bottom layer stack needs to use 2, and the rest is used exclusively by the application layer; for a TX fifo of 16, the application layer can only use 14; for 8, the application layer can only use 6.

When sending data from the application layer (e.g. calling `blc_gatt_pushHandleValueNotify`), the user should first check how many TX fifo's are currently available in the Link Layer.

The following API is used to determine how many TX fifo's are currently occupied, not how many are left.

```
u8          blc_ll_getTxFifoNumber (void);
```

For example, if the TX fifo number defaults to 16, there are 14 users available, so the value returned by the API is available as long as it is less than 14: a return of 13 means there is 1 available, and a return of 0 means there are 14 available.

When using TX fifo, if the customer first looks at how many are left before deciding whether to push the data directly, a fifo should be left in place to prevent various boundary issues from occurring.

In the voice processing of the B85m remote, as each voice data is known to be split into 5 packets, 5 TX fifo's are required and no more than 9 occupied fifo's can be used. In order to avoid exceptions caused by some boundary conditions when the TX fifo is used (e.g. just in time for the BLE stack to reply to the master's command and insert a data into the TX fifo), the final code is written as follows: the voice data is only pushed to the TX fifo when there are no more than 8 occupied TX fifo's.

```
if (blc_ll_getTxFifoNumber() < 9)
{
    .....
}
```

As discussed above, the SDK provides the following API for limiting the amount of more data received on an interval (if the customer wants to limit the data even if the RX fifo is sufficient), in addition to the automatic data overflow handling mechanism.

```
void      blc_ll_init_max_md_nums(u8 num);
```

In which, the maximum number of more data set by parameter num should not to exceed the RX fifo number.

Note:

- Note that the ability to qualify more data on a connection event is only enabled if the API is called at the application level (parameter num is greater than 0).

3.2.7 Controller Event

Considering user may need to record and process some key actions of BLE stack bottom layer in the APP layer, Telink BLE SDK provides three types of event: Standard HCI event defined by BLE Controller; Telink defined event; event-notification type GAP event (Host event) defined by BLE Host for stack flow interaction (see section GAP event).

As shown in the BLE SDK event architecture below: HCI event and Telink defined event are Controller event, while GAP event is BLE Host event.

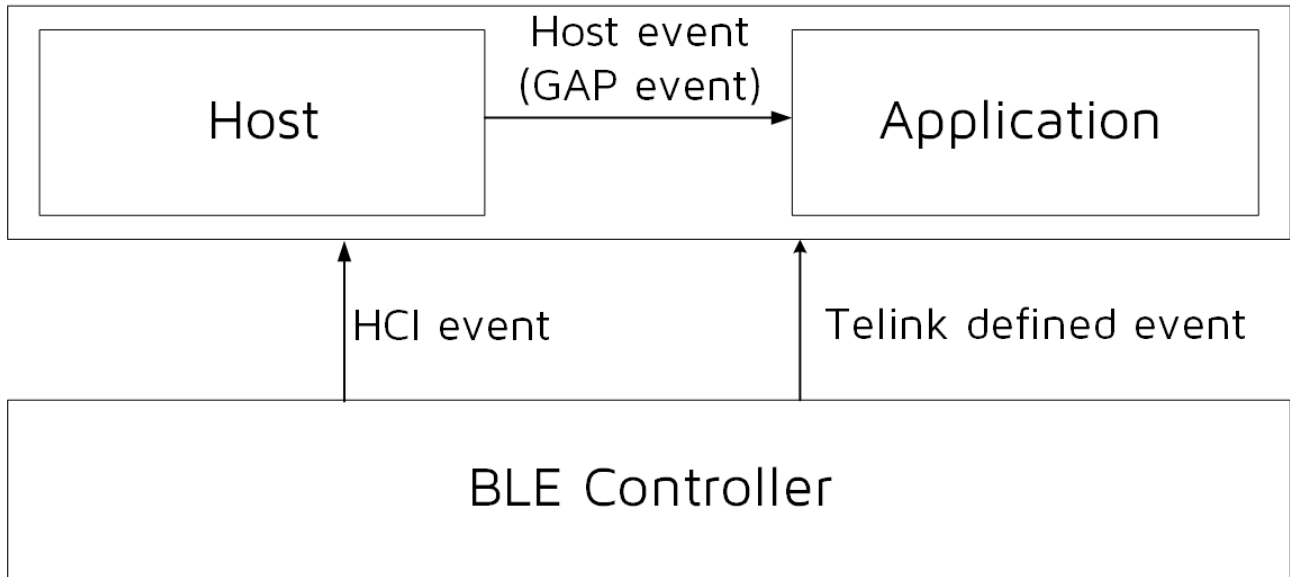


Figure 3.24: "BLE SDK Event Architecture"

3.2.7.1 Controller HCI Event

HCI event is designed according to BLE Spec; Telink defined event only applies to BLE Slave (b85m remote/ b85m module etc).

- BLE Master only supports HCI event.
- BLE Slave supports both HCI event and Telink defined event.

For BLE Slave, basically the two sets of event are independent of each other, except for the connect and disconnect event of Link Layer.

User can select one set or use both as needed. In Telink BLE SDK, b85m hci/b85m master kma dongle module use Telink defined event.

As shown in the "Host + Controller" architecture below, Controller HCI event indicates all events of Controller are reported to Host via HCI.

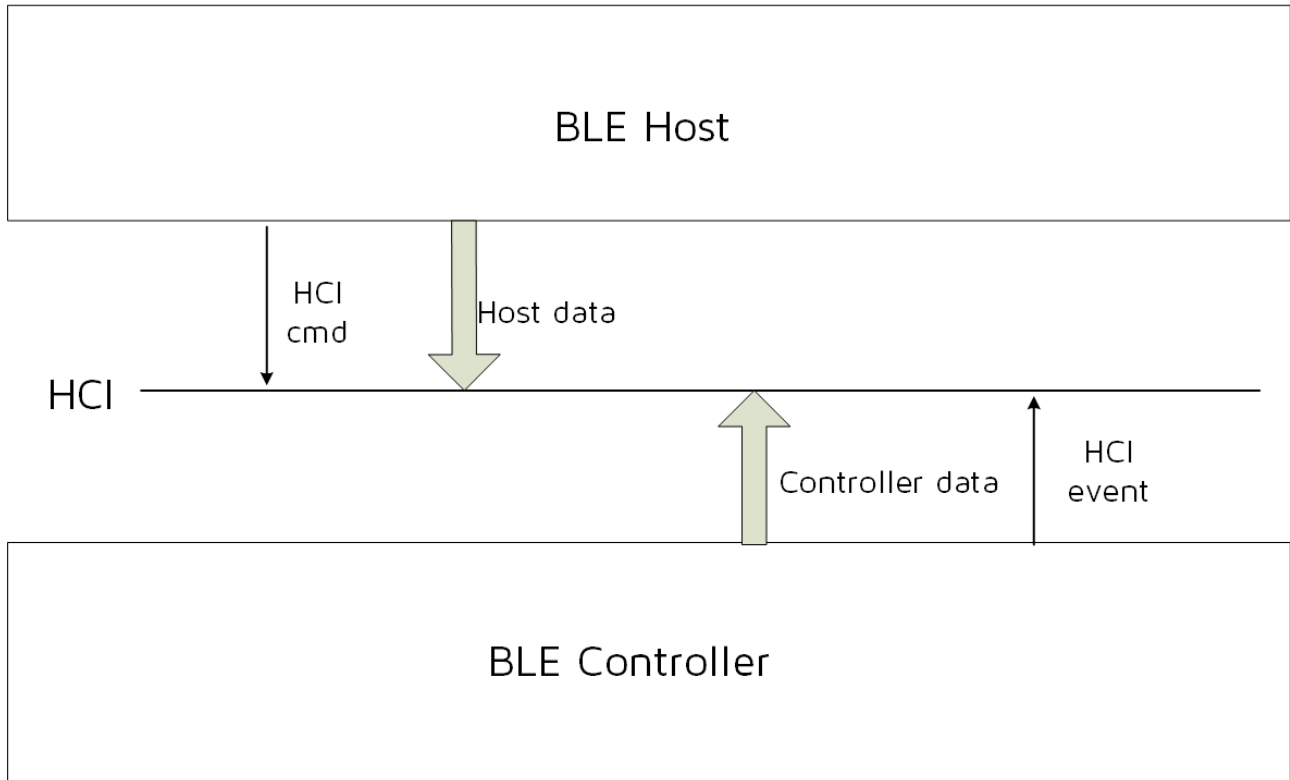


Figure 3.25: “HCI Event”

For definition of Controller HCI event, please refer to “Core_v5.0” (Vol 2/Part E/7.7 “Events”). “LE Meta Event” in 7.7.65 indicates HCI LE (low energy) Event, while the others are common HCI events. As defined in Spec, Telink BLE SDK also divides Controller HCI event into two types: HCI Event and HCI LE event. Since Telink BLE SDK focuses on BLE, it supports most HCI LE events and only a few basic HCI events.

For the definition of macros and interfaces related to Controller HCI event, please refer to head files under "stack/ble/hci".

To receive Controller HCI event in Host or APP layer, user should register callback function of Controller HCI event, and then enable mask of corresponding event.

Following are callback function prototype and register interface of Controller HCI event:

```
typedef int (*hci_event_handler_t) (u32 h, u8 *para, int n);
void blc_hci_registerControllerEventHandler(hci_event_handler_t handler);
```

In the callback function prototype, "u32 h" is a mark which will be frequently used in bottom-layer stack, and user only needs to know the following:

```
#define HCI_FLAG_EVENT_TLK_MODULE (1<<24)
#define HCI_FLAG_EVENT_BT_STD (1<<25)
```

The "HCI_FLAG_EVENT_TLK_MODULE" will be introduced in "Telink defined event", while "HCI_FLAG_EVENT_BT_STD" indicates current event is Controller HCI event.

In the callback function prototype, “para” and “n” indicate data and data length of event. The data is consistent with the definition in BLE spec. User can refer to the following usage in the b85m_master kma dongle and the specific implementation of the controller_event_callback function.

```
blc_hci_registerControllerEventHandler(controller_event_callback);
```

3.2.7.2 HCI event

Telink BLE SDK supports a few HCI events. Following lists some events for user.

```
#define HCI_EVT_DISCONNECTION_COMPLETE      0x05
#define HCI_EVT_ENCRYPTION_CHANGE          0x08
#define HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE 0x0C
#define HCI_EVT_ENCRYPTION_KEY_REFRESH     0x30
#define HCI_EVT_LE_META                    0x3E
```

a) HCI_EVT_DISCONNECTION_COMPLETE

Please refer to “Core_v5.0” (Vol 2/Part E/7.7.5 “Disconnection Complete Event”). Total data length of this event is 7, and 1-byte “param len” is 4, as shown below. Please refer to BLE spec for data definition.

hci event	event code	param len	status	connection handle	reason
0x04	0x05	4	0x00		

Figure 3.26: “Disconnection Complete Event”

b) HCI_EVT_ENCRYPTION_CHANGE and HCI_EVT_ENCRYPTION_KEY_REFRESH

Please refer to “Core_v5.0” (Vol 2/Part E/7.7.8 & 7.7.39). The two events are related to Controller encryption, and the processing is assembled in library.

c) HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE

Please refer to “Core_v5.0” (Vol 2/Part E/7.7.12). When Host uses the “HCI_CMD_READ_REMOTE_VER_INFO” command to exchange version information between Controller and BLE peer device, and version of peer device is received, this event will be reported to Host. Total data length of this event is 11, and 1-byte “param len” is 8, as shown below. Please refer to BLE spec for data definition.

hci event	event code	param len	status	connection handle	version	manufacture name	subversion
0x04	0x0c	8	0x00				

Figure 3.27: “Read Remote Version Information Complete Event”

d) HCI_EVT_LE_META

It indicates current event is HCI LE event, and event type can be judged according to sub event code. Except for HCI_EVT_LE_META, other HCI events should use the API below to enable corresponding event mask.

```
ble_sts_t    blc_hci_setEventMask_cmd(u32 evtMask);
```

Definition of event mask::

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE    0x0000000010
#define HCI_EVT_MASK_ENCRYPTION_CHANGE        0x0000000080
#define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE 0x00000000800
```

If the user does not set the HCI event mask via this API, the SDK will only turn on the mask corresponding to HCI_CMD_DISCONNECTION_COMPLETE by default, i.e. to ensure that the Controller disconnect event is reported.

3.2.7.3 HCI LE event

When event code in HCI event is "HCI_EVT_LE_META" to indicate HCI LE event, common sub-event code are shown as below:

```
#define HCI_SUB_EVT_LE_CONNECTION_COMPLETE    0x01
#define HCI_SUB_EVT_LE_ADVERTISING_REPORT    0x02
#define HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE 0x03
#define HCI_SUB_EVT_LE_CONNECTION_ESTABLISH  0x20
```

a) HCI_SUB_EVT_LE_CONNECTION_COMPLETE

Please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.1 "LE Connection Complete Event"). When connection is established between Controller Link Layer and peer device, this event will be reported. Total data length of this event is 22, and 1-byte "param len" is 19, as shown below. Please refer to BLE spec for data definition.

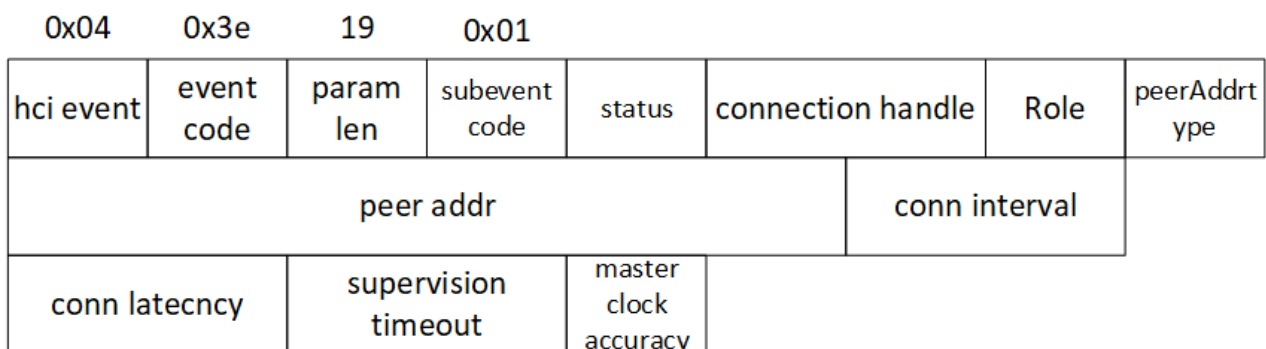


Figure 3.28: "LE Connection Complete Event"

b) HCI_SUB_EVT_LE_ADVERTISING_REPORT

Please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.2 "LE Advertising Report Event"). When Link Layer of Controller scans right adv packet, it will be reported to Host via the "HCI_SUB_EVT_LE_ADVERTISING_REPORT". Data length of this event is not fixed and it depends on payload of adv packet, as shown below. Please refer to BLE spec for data definition.

0x04	0x3e	0x02				
hci event	event code	param len	subevent code	num report	event type	address type[1...i]
address[1...i]						length[1..i]
data[1...i]						rssi[1..i]

Figure 3.29: "LE Advertising Report Event"

Note: In Telink BLE SDK, each "LE Advertising Report Event" only reports an adv packet, i.e. "i" in figure above is 1.

c) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

Please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.3 "LE Connection Update Complete Event"). When "connection update" in Controller takes effect, the "HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE" will be reported to Host. Total data length of this event is 13, and 1-byte "param len" is 10, as shown below. Please refer to BLE spec for data definition.

0x04	0x3e	10	0x03		
hci event	event code	param len	subevent code	status	connection handle
conn interval		conn latency		supervision timeout	

Figure 3.30: "LE Connection Update Complete Event"

d) HCI_SUB_EVT_LE_CONNECTION_ESTABLISH

The "HCI_SUB_EVT_LE_CONNECTION_ESTABLISH" is a supplement to the "HCI_SUB_EVT_LE_CONNECTION_COMPLETE", so all the parameters except for subevent is the same. This event is used by the b85m_master kma dongle in the SDK. It is the only non-BLE spec standard event, is privately defined by Telink and is only used in the b85m_master kma dongle.

Following illustrates the reason for Telink to define this event.

When BLE Controller in Initiating state scans adv packet from specific device to be connected, it will send connection request packet to peer device; no matter whether this connection request is received, it will be

considered as "Connection complete", "LE Connection Complete Event" will be reported to Host, and Link Layer immediately enters Master role. Since this packet does not support ack/retry mechanism, Slave may miss the connection request, thus it cannot enter Slave role, and won't enter brx mode to transfer packets. In this case, Master Controller will process according to the mechanism below: After it enters Master role, it will check whether there's any packet received from Slave during the beginning 6 ~ 10 conn intervals (CRC check is negligible).

- If no packet is received, it's considered that Slave does not receive connection request; suppose "LE Connection Complete Event" has already been reported, it must report a "Disconnection Complete Event" quickly, and indicate disconnect reason is "0x3E (HCI_ERR_CONN_FAILED_TO_ESTABLISH)".
- If there's packet received from Slave, it can confirm that Connection is established, thus Master can continue rest of the flow.

According to the description above, the processing method of BLE Host should be: After it receives "LE Connection Complete Event" of Controller, it cannot confirm that connection has already been established, but instead, starts a timer based on conn interval (timing value should be configured as 10 intervals or above to cover the longest time). After the timer is started, it will check whether there is "Disconnection Complete Event" with disconnect reason of 0x3E; if there is no such event, it will be considered as "Connection Established".

Considering this processing of BLE Host is very complex and error prone, the SDK defines the "HCI_SUB_EVT_LE_CONNECTION_ESTABLISH" in the bottom layer. When Host receives this event, it indicates that Controller has confirmed connection is OK on Slave side and can continue rest of the flow.

"HCI LE event" needs the API below to enable mask.

```
ble_sts_t   blc_hci_le_setEventMask_cmd(u32 evtMask);
```

Following lists some evtMask definitions. User can view the other events in the "hci_const.h".

```
#define HCI_LE_EVT_MASK_CONNECTION_COMPLETE      0x00000001
#define HCI_LE_EVT_MASK_ADVERTISING_REPORT      0x00000002
#define HCI_LE_EVT_MASK_CONNECTION_UPDATE_COMPLETE 0x00000004
#define HCI_LE_EVT_MASK_CONNECTION_ESTABLISH    0x80000000
```

If HCI LE event mask is not set via this API, mask of all HCI LE events in the SDK are disabled by default.

3.2.7.4 Telink Defined Event

Besides standard Controller HCI event, the SDK also provides Telink defined event. Up to 20 Telink defined events are supported, which are defined by using macros in the "stack/ble/ll/ll.h".

Current SDK version supports the following callback events. The "BLT_EV_FLAG_CONNECT / BLT_EV_FLAG_TERMINATE" has the same function as the "HCI_SUB_EVT_LE_CONNECTION_COMPLETE" / "HCI_EVT_DISCONNECTION_COMPLETE" in HCI event, but data definition of these events are different.

```
#define BLT_EV_FLAG_ADV 0
#define BLT_EV_FLAG_ADV_DURATION_TIMEOUT 1
#define BLT_EV_FLAG_SCAN_RSP 2
#define BLT_EV_FLAG_CONNECT 3
#define BLT_EV_FLAG_TERMINATE 4
#define BLT_EV_FLAG_LL_REJECT_IND 5
#define BLT_EV_FLAG_RX_DATA_ABANDON 6
#define BLT_EV_FLAG_PHY_UPDATE 7
#define BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 8
#define BLT_EV_FLAG_GPIO_EARLY_WAKEUP 9
#define BLT_EV_FLAG_CHN_MAP_REQ 10
#define BLT_EV_FLAG_CONN_PARA_REQ 11
#define BLT_EV_FLAG_CHN_MAP_UPDATE 12
#define BLT_EV_FLAG_CONN_PARA_UPDATE 13
#define BLT_EV_FLAG_SUSPEND_ENTER 14
#define BLT_EV_FLAG_SUSPEND_EXIT 15
```

Telink defined event is only triggered in BLE slave applications. There are two ways to implement callback of Telink defined event in BLE slave application.

- (1) The first method, which is called "independent registration", is to independently register callback function for each event.

Prototype of callback function is shown as below:

```
typedef void (*blt_event_callback_t)(u8 e, u8 *p, int n);
```

Where "e": event number. "p": It's the pointer to the data transmitted from the bottom layer when callback function is executed, and it varies with the callback function. "n": length of valid data pointed by pointer.

API to register callback function:

```
void bls_app_registerEventCallback (u8 e, blt_event_callback_t p);
```

Whether each event will respond depends on whether corresponding callback function is registered in APP layer.

- (2) The second method, which is called "shared event entry", is that all event callback functions share the same entry. Whether each event will respond depends on whether its event mask is enabled. This method uses the same API as HCI event to register event callback:

```
typedef int (*hci_event_handler_t) (u32 h, u8 *para, int n);
void blc_hci_registerControllerEventHandler(hci_event_handler_t handler);
```

Although registered callback function of HCI event is shared, they are different in implementation. In HCI event callback function:

```
h = HCI_FLAG_EVENT_BT_STD | hci_event_code;
```

While in Telink defined event "shared event entry":

```
h = HCI_FLAG_EVENT_TLK_MODULE | e;
```

Where "e" is event number of Telink defined event.

Telink defined event "shared event entry" is similar to mask of HCI event; the API below serves to set the mask to determine whether each event will be responded.

```
ble_sts_t bls_hci_mod_setEventMask_cmd(u32 evtMask);
```

Relationship between evtMask and event number is:

```
evtMask = BIT(e);
```

The two methods for Telink defined event are exclusive to each other. The first method is recommended and is adopted by most demo code of the SDK; only "b85m_module" uses the "shared event entry" method.

For the use of Telink defined event, please refer to the demo code of project "b85m_module" for 2 "shared event entry".

The following takes the connect and terminate event callbacks as examples to describe the code implementation methods of these two methods.

(1) The first method: "independent registration"

```
void task_connect (u8 e, u8 *p, int n)
{
    // add connect callback code here
}

void task_terminate (u8 e, u8 *p, int n)
{
    // add terminate callback code here
}

bls_app_registerEventCallback (BLT_EV_FLAG_CONNECT, &task_connect);
bls_app_registerEventCallback (BLT_EV_FLAG_TERMINATE, &task_terminate);
```

(2) The second method: "shared event entry"

```
int event_handler(u32 h, u8 *para, int n)
{
    if( (h&HCI_FLAG_EVENT_TLK_MODULE)!= 0 ) //module event
    {
```

```

switch(event)
{
    case BLT_EV_FLAG_CONNECT:
    {
        // add connect callback code here
    }
    break;
    case BLT_EV_FLAG_TERMINATE:
    {
        // add terminate callback code here
    }
    break;
    default:
    break;
}
}

blc_hci_registerControllerEventHandler(event_handler);
bls_hci_mod_setEventMask_cmd( BIT(BLT_EV_FLAG_CONNECT) | BIT(BLT_EV_FLAG_TERMINATE) );

```

Following will introduce details about all events, event trigger condition and parameters of corresponding callback function for Controller.

(1) BLT_EV_FLAG_ADV

This event is not used in current SDK.

(2) BLT_EV_FLAG_ADV_DURATION_TIMEOUT

Event trigger condition: If the API "bls_ll_setAdvDuration" is invoked to set advertising duration, a timer will be started in BLE stack bottom layer. When the timer reaches the specified duration, advertising is stopped, and this event is triggered. In the callback function of this event, user can modify adv event type, re-enable advertising, re-configure advertising duration, and etc.

Pointer "p": null pointer.

Data length "n": 0.

Note: This event won't be triggered in "advertising in ConnSlaveRole" which is an extended state of Link Layer.

(3) BLT_EV_FLAG_SCAN_RSP

Event trigger condition: When Slave is in advertising state, this event will be triggered if Slave responds with scan response to the scan request from Master.

Pointer "p": null pointer.

Data length "n": 0.

(4) BLT_EV_FLAG_CONNECT

Event trigger condition: When Link Layer is in advertising state, this event will be triggered if it responds to connect request from Master and enters Conn state Slave role.

Data length “n”: 34.

Pointer “p”: p points to one 34-byte RAM area, corresponding to the “connect request PDU” below.

Payload		
InitA (6 octets)	AdvA (6 octets)	LLData (22 octets)

Figure 2.10: CONNECT_REQ PDU payload

The format of the LLData field is shown in Figure 2.11.

LLData									
AA (4 octets)	CRCInit (3 octets)	WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	ChM (5 octets)	Hop (5 bits)	SCA (3 bits)

Figure 2.11: LLData field structure in CONNECT_REQ PDU's payload

Figure 3.31: “Connect Request PDU”

Please refer to the “rf_packet_connect_t” defined in the “ble_formats.h”. In the structure below, the connect request PDU is from scanA[6] (corresponding to InitA in figure above) to hop.

```
typedef struct{
    u32 dma_len;
    u8  type   :4;
    u8  rfu1   :1;
    u8  chan_sel:1;
    u8  txAddr :1;
    u8  rxAddr :1;
    u8  rf_len;
    u8  initA[6];
    u8  advA[6];
    u8  accessCode[4];
    u8  crcinit[3];
    u8  winSize;
    u16 winOffset;
    u16 interval;
    u16 latency;
    u16 timeout;
    u8  chm[5];
    u8  hop;
}rf_packet_connect_t;
```

(5) BLT_EV_FLAG_TERMINATE

Event trigger condition: This event will be triggered when Link Layer state machine exits from Conn state Slave role in any of the three specific cases.

Pointer "p": p points to an u8-type variable "terminate_reason". This variable indicates the reason for disconnection of Link Layer.

Data length "n": 1.

Three cases to exit Conn state Slave role and corresponding reasons are listed as below:

A. If Slave fails to receive packet from Master for a duration due to RF communication problem (e.g. bad RF or Master is powered off), and "connection supervision timeout" expires, this event will be triggered to terminate connection and return to None Conn state. The terminate reason is HCI_ERR_CONN_TIMEOUT (0x08).

B. If Master sends "terminate" command to actively terminate connection, after Slave responds to the command with an ack, this event will be triggered to terminate connection and return to None Conn state. The terminate reason is the Error Code in the "LL_TERMINATE_IND" control packet received in Slave Link Layer. The Error Code is determined by Master. Common Error Codes include HCI_ERR_REMOTE_USER_TERM_CONN (0x13), HCI_ERR_CONN_TERM_MIC_FAILURE (0x3D), and etc.

C. If Slave invokes the API "bls_ll_terminateConnection(u8 reason)" to actively terminate connection, this event will be triggered. The terminate reason is the actual parameter "reason" of this API.

(6) BLT_EV_FLAG_LL_REJECT_IND

Event trigger condition: When Master sends a "LL_ENC_REQ" (encryption request) in the Link Layer and it's declared to use the pre-allocated LTK, if Slave fails to find corresponding LTK and responds with a "LL_REJECT_IND" (or "LL_REJECT_EXT_IND"), this event will be triggered.

Pointer "p": p points to the response command (LL_REJECT_IND or LL_REJECT_EXT_IND).

Data length "n": 1.

For more information, please refer to "Core_v5.0" Vol 6/Part B/2.4.2.

(7) BLT_EV_FLAG_RX_DATA_ABANDON

Event trigger condition: This event will be triggered when BLE RX fifo overflows (see section Link Layer TX fifo & RX fifo), or the number of More Data received in an interval exceeds the preset threshold (Note: User needs to invoke the API "blc_ll_init_max_md_nums" with non-zero parameter, so that SDK bottom layer will check the number of More Data.)

Pointer "p": null pointer.

Data length "n": 0.

(8) BLT_EV_FLAG_PHY_UPDATE

Event trigger condition: This event will be triggered after the update succeeds or fails when the slave or master proactively initiates LL_PHY_REQ; Or when the slave or master passively receives LL_PHY_REQ and meanwhile PHY is updated successfully, this event will be triggered.

Data length "n": 1.

Pointer "p": p points to an u8-type variable indicating the current connection of PHY mode.

```
typedef enum {
    BLE_PHY_1M           = 0x01,
    BLE_PHY_2M           = 0x02,
    BLE_PHY_CODED        = 0x03,
} le_phy_type_t;
```

(9) BLT_EV_FLAG_DATA_LENGTH_EXCHANGE

Event trigger condition: This event will be triggered when Slave and Master exchange max data length of Link Layer, i.e. one side sends "ll_length_req", while the peer responds with "ll_length_rsp". If Slave actively sends "ll_length_req", this event won't be triggered until "ll_length_rsp" is received. If Master initiates "ll_length_req", this event will be triggered immediately after Slave responds with "ll_length_rsp".

Data length "n": 12.

Pointer "p": p points to data of a memory area, corresponding to the beginning six u16-type variables in the structure below.

```
typedef struct {
    u16    connEffectiveMaxRxOctets;
    u16    connEffectiveMaxTxOctets;
    u16    connMaxRxOctets;
    u16    connMaxTxOctets;
    u16    connRemoteMaxRxOctets;
    u16    connRemoteMaxTxOctets;
    .....
}ll_data_extension_t;
```

The "connEffectiveMaxRxOctets" and "connEffectiveMaxTxOctets" are max RX and TX data length finally allowed in current connection; "connMaxRxOctets" and "connMaxTxOctets" are max RX and TX data length of the device; "connRemoteMaxRxOctets" and "connRemoteMaxTxOctets" are max RX and TX data length of peer device.

```
connEffectiveMaxRxOctets = min(supportedMaxRxOctets,connRemoteMaxTxOctets);
```

```
connEffectiveMaxTxOctets = min(supportedMaxTxOctets, connRemoteMaxRxOctets);
```

(10) BLT_EV_FLAG_GPIO_EARLY_WAKEUP

Event trigger condition: Slave will calculate wakeup time before it enters sleep (suspend or deepsleep retention), so that it can wake up when the wakeup time is due (It's realized via timer in sleep). Since user tasks won't be processed until wakeup from sleep, long sleep time may bring problem for real-time demanding applications. Take keyboard scanning as an example: If user presses keys fast, to avoid key press loss and process debouncing, it's recommended to set the scan interval as 10~20ms; longer sleep time (e.g. 400ms or 1s, which may be reached when latency is enabled) will lead to key press loss. So it's needed to judge current sleep time before MCU enters sleep; if it's too long, the wakeup method of user key press should be enabled, so that MCU can wake up from sleep (suspend or deepsleep retention) in advance (i.e. before timer timeout) if any key press is detected. This will be introduced in details in following PM module section.

The event "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" will be triggered if MCU is woke up from sleep (suspend or deepsleep) by GPIO in advance before wakeup timer expires.

Data length "n": 1.

Pointer "p": p points to an u8-type variable "wakeup_status". This variable indicates valid wakeup source status for current suspend. Following types of wakeup status are defined in the "drivers/8258(8278)/pm.h":

```
enum {
    WAKEUP_STATUS_COMPARATOR    = BIT(0),
    WAKEUP_STATUS_TIMER         = BIT(1),
    WAKEUP_STATUS_CORE          = BIT(2),
    WAKEUP_STATUS_PAD           = BIT(3),
    WAKEUP_STATUS_MDEC          = BIT(4),
    STATUS_GPIO_ERR_NO_ENTER_PM = BIT(7),
    STATUS_ENTER_SUSPEND        = BIT(30),
};
```

For parameter definition above, please refer to "Power Management" section.

(11) BLT_EV_FLAG_CHN_MAP_REQ

Event trigger condition: When Slave is in Conn state, if Master needs to update current connection channel list, it will send a "LL_CHANNEL_MAP_REQ" command to Slave; this event will be triggered after Slave receives this request from Master and has not processed the request yet.

Data length "n": 5.

Pointer "p": p points to the starting address of the following channel list array.

unsigned char type bltc.conn_chn_map[5], Note: When the callback function is executed, p points to the old channel map before update.

Five bytes are used in the "conn_chn_map" to indicate current channel list by mapping. Each bit indicates a channel:

conn_chn_map[0] bit0-bit7 indicate channel0~channel7, respectively.

conn_chn_map[1] bit0-bit7 indicate channel8~channel15, respectively.

conn_chn_map[2] bit0-bit7 indicate channel16~channel23, respectively.

conn_chn_map[3] bit0-bit7 indicate channel24~channel31, respectively.

conn_chn_map[4] bit0-bit4 indicate channel32~channel36, respectively.

(12) BLT_EV_FLAG_CHN_MAP_UPDATE

Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated channel map after it receives the "LL_CHANNEL_MAP_REQ" command from Master.

Pointer "p": p points to the starting address of the new channel map conn_chn_map[5] after update.

Data length "n": 5.

(13) BLT_EV_FLAG_CONN_PARA_REQ

Event trigger condition: When Slave is in connection state (Conn state Slave role), if Master needs to update current connection parameters, it will send a "LL_CONNECTION_UPDATE_REQ" command to Slave; this event will be triggered after Slave receives this request from Master and has not processed the request yet.

Data length "n": 11.

Pointer "p": p points to the 11-byte PDU of the LL_CONNECTION_UPDATE_REQ.

CtrData					
WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	Instant (2 octets)

Figure 2.15: CtrData field of the LL_CONNECTION_UPDATE_REQ PDU

Figure 3.32: "LL_CONNECTION_UPDATE REQ Format in BLE Stack"

(14) BLT_EV_FLAG_CONN_PARA_UPDATE

Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated connection parameters after it receives the "LL_CONNECTION_UPDATE_REQ" from Master.

Data length "n": 6.

Pointer "p": p points to the new connection parameters after update, as shown below.

p[0] | p[1]<<8: new connection interval in unit of 1.25ms.

p[2] | p[3]<<8: new connection latency.

p[4] | p[5]<<8: new connection timeout in unit of 10ms.

(15) BLT_EV_FLAG_SUSPEND_ENETR

Event trigger condition: When Slave executes the function "blt_sdk_main_loop", this event will be triggered before Slave enters suspend.

Pointer "p": Null pointer.

Data length "n": 0.

(16) BLT_EV_FLAG_SUSPEND_EXIT

Event trigger condition: When Slave executes the function "blt_sdk_main_loop", this event will be triggered after Slave is woke up from suspend.

Pointer "p": Null pointer.

Data length "n": 0.

Note:

- This callback is executed after SDK bottom layer executes "cpu_sleep_wakeup" and Slave is woke up, and this event will be triggered no matter whether the actual wakeup source is gpio or timer. If the event "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" occurs at the same time, for the sequence to execute the two events, please refer to pseudo code in "Power Management – PM Working Mechanism".

3.2.8 Data Length Extension

BLE Spec core_4.2 and above supports Data Length Extension (DLE).

Link Layer in this BLE SDK supports data length extension to max rf_len of 251 bytes per BLE Spec.

Please refer to "Core_v5.0" (Vol 6/Part B/2.4.2.21 "LL_LENGTH_REQ and LL_LENGTH_RSP").

Following steps explains how to use data length extension.

Step 1 Configure suitable TX & RX fifo size

To receive and transmit long packet, bigger Tx & Rx fifo size is required and thus occupies large SRAM space. So be cautious when setting fifo size to avoid waste of SRAM space.

Tx fifo size should be increased to transmit long packet. Tx fifo size should be larger than Tx rf_len by 12, and must be aligned by 4 bytes.

TX rf_len = 56 bytes: MYFIFO_INIT(bl_txfifo, 68, 8);

TX rf_len = 141 bytes: MYFIFO_INIT(bl_txfifo, 156, 8);

TX rf_len = 191 bytes: MYFIFO_INIT(bl_txfifo, 204, 8);

Rx fifo size should be increased to receive long packet. Rx fifo size should be larger than Rx rf_len by 24, and must be aligned by 16 bytes.

RX rf_len = 56 bytes: MYFIFO_INIT(bl_rxfifo, 80, 8);

RX rf_len = 141 bytes: MYFIFO_INIT(bl_rxfifo, 176, 8);

RX rf_len = 191 bytes: MYFIFO_INIT(bl_rxfifo, 224, 8);

Step 2 Set proper MTU size

MTU, the maximum transmission unit, is used to set the size of the payload in a single packet of the L2CAP layer in BLE. The att.h provides the interface function ble_sts_t blc_att_setRxMtuSize(u16 mtu_size); during the initialization of the BLE stack, users can directly use this function to pass the parameter to set the MTU. However, the MTU size effect is negotiated in the MTU exchange process, MTU size effect = min(MTU_A, MTU_B), MTU_A is the MTU size supported by device A, MTU_B is the MTU size supported by device B; in addition, only MTU size greater than the DLE length can make full use of the DLE to increase the link layer data throughput rate.

For the implementation of MTU size exchange, please refer to the detailed description in the "ATT & GATT" section of this document, or refer to the DLE demo in b85m_feature_test.

```
#define MTU_SIZE_SETTING 196
blc_att_setRxMtuSize(MTU_SIZE_SETTING);
```

For MTU greater than 247, the user can register buffer with the following API.

```
void blc_l2cap_initMtuBuffer(u8 pMTU_rx_buff, u16 mtu_rx_size, u8 pMTU_tx_buff, u16 mtu_tx_size)
```

Step 3 data length exchange

Before transfer of long packets, please make sure the "data length exchange" flow has already been completed in BLE connection. Data length exchange is an interactive process in Link Layer by LL_LENGTH_REQ

and LL_LENGTH_RSP. Either master or slave can initiate the process by sending LL_LENGTH_REQ, while the peer responds with LL_LENGTH_RSP. Through this interaction, master and slave obtain the max Tx and Rx packet size from each other, and adopt the minimum of the two as the max Tx and Rx packet size in current connection.

No matter which side initiates LL_LENGTH_REQ, at the end of data length exchange process, the SDK will generate "BLT_EV_FLAG_DATA_LENGTH_EXCHANGE" event callback assuming this callback has been registered. User can refer to "Telink defined event" section to understand the parameters of this event callback function.

The final max Tx and Rx packet size can be obtained from the "BLT_EV_FLAG_DATA_LENGTH_EXCHANGE" event callback function.

When 8x5x acts as slave device in actual applications, master may or may not initiate LL_LENGTH_REQ. If master does not initiate it, slave should initiate LL_LENGTH_REQ by the following API in the SDK:

```
ble_sts_t      blc_ll_exchangeDataLength (u8 opcode, u16 maxTxOct);
```

In this API, "opcode" is "LL_LENGTH_REQ", and "maxTxOct" is the max Tx packet size supported by current device. For example, if max Tx packet size is 200bytes, the setting below applies:

```
blc_ll_exchangeDataLength(LL_LENGTH_REQ , 200);
```

Since the slave device does not know whether the master will initiate LL_LENGTH_REQ, we recommend a method for your reference: register the BLT_EV_FLAG_DATA_LENGTH_EXCHANGE event callback, turn on a software timer to start timing when the connection is established (e.g. 2S), if this callback has not been triggered after 2s, it means that master has not initiated LL_LENGTH_REQ, at this time slave calls API blc_ll_exchangeDataLength to initiate LL_LENGTH_REQ.

Step 4 MTU size exchange

In addition to data length exchange, MTU size exchange flow should also be executed to ensure large MTU size takes effect, so that the peer can process long packet in BLE L2CAP layer. MTU size should be equal or larger than max packet size of Tx & Rx. Please refer to "ATT & GATT" section or the demo of the B85m_feature for the implementation of MTU size exchange.

Step 5 Transmission/Reception of long packet

Please refer to "ATT & GATT" section for illustration of Handle Value Notification, Handle Value Indication, Write request and Write Command.

Transmission and reception of long packet can start after correct completion of the three steps above.

The APIs corresponding to "Handle Value Notification" and "Handle Value Indication" can be invoked in ATT layer to transmit long packet. As shown below, fill in the address and length of data to be sent to the parameters "*p" and "len", respectively:

```
ble_sts_t      blc_gatt_pushHandleValueNotify(u16 connHandle, u16 attHandle, u8 *p, int len);  
ble_sts_t      blc_gatt_pushHandleValueIndicate(u16 connHandle, u16 attHandle, u8 *p, int len);
```

To receive long packet, it's only needed to use callback function "w" corresponding to "Write Request" and "Write Command". In the callback function, reference the data pointed to by the form reference pointer.

3.2.9 Controller API

3.2.9.1 Controller API Introduction

In standard BLE stack architecture of section 3.1.1, APP layer cannot directly communicate with Link Layer of Controller, i.e. data of APP layer must be first transferred to Host, and then Host can transfer control command to Link Layer via HCI. All control commands from Host to LL via HCI follow the definition in BLE spec "Core_v5.0", please refer to "Core_v5.0" (Vol 2/Part E/ Host Controller Interface Functional Specification) for more information.

Telink BLE SDK based on standard BLE architecture can serve as a Controller and work together with Host system. Therefore, all APIs to operate Link Layer strictly follow the data format of Host commands in the spec.

Although the architecture in the figure above is used in Telink BLE SDK, during which APP layer can directly operate Link Layer, it still use the standard APIs of HCI part.

In BLE spec, all HCI commands to operate Controller have corresponding "HCI command complete event" or "HCI command status event" in response to Host layer. However, in Telink BLE SDK, it is handled case by case.

1) For b85m_hci class applications, Telink IC only acts as a BLE controller and needs to work with other MCU's running BLE hosts, a corresponding HCI command complete event or HCI command status event will be generated for each HCI command.

2) For b85m_master kma dongle application, both BLE Host and Controller are running on Telink IC, when Host calls interface to send HCI command to Controller, all of them are received correctly by Controller and there is no loss, so the Controller no longer replies to the HCI command complete event or HCI command status event when processing the HCI command.

The Controller API is declared in the header files in the stack/ble/ll and stack/ble/hci directories, where the ll directory is divided into ll.h, ll_adv.h, ll_scan.h, ll_ext_adv.h, ll_pm.h and ll_whitelist.h, the user can look for Link Layer functions, for example, the APIs for functions related to advising should be declared in ll_adv.h.

3.2.9.2 API Return Type `ble_sts_t`

An enum type "ble_sts_t" defined in the "stack/ble/ble_common.h" is used as return value type for most APIs in the SDK. When API invoking with right parameter setting is accepted by the protocol stack, it will return "0" to indicate BLE_SUCCESS; if any non-zero value is returned, it indicates a unique error type. All possible return values and corresponding error reason will be listed in the subsections below for each API.

The "ble_sts_t" applies to both APIs in Link Layer and some APIs in Host layer.

3.2.9.3 BLE MAC address initialization

In this document, "BLE MAC address" includes both "public address" and "random static address".

In this BLE SDK, the API below serves to obtain public address and random static address:

```
void blc_initMacAddress(int flash_addr, u8 *mac_public, u8 *mac_random_static);
```

The “flash_addr” is the flash address to store MAC address. As explained earlier, this address in 8x5x 512K flash is 0x76000. If random static address is not needed, set “mac_random_static” as “NULL”.

After the BLE public MAC address has been successfully obtained, the API for Link Layer initialization is called and the MAC address is passed into the BLE protocol stack.

```
blc_ll_initStandby_module(mac_public);
```

If you use the Advertising state or Scanning state in the Link Layer’s state machine, you also need to pass in the MAC address.

```
blc_ll_initAdvertising_module (mac_public) ;
blc_ll_initScanning_module (mac_public) ;
```

3.2.9.4 Link Layer state machine initialization

In conjunction with the previous detailed description of the Link Layer state machine, the following APIs are used to configure the initialisation of the individual modules when building the BLE state machine.

```
void blc_ll_initBasicMCU (void)
void blc_ll_initStandby_module (u8 *public_adr);
void blc_ll_initAdvertising_module(u8 *public_adr);
void blc_ll_initScanning_module(u8 *public_adr);
void blc_ll_initInitiating_module(void);
void blc_ll_initSlaveRole_module(void);
void blc_ll_initMasterRoleSingleConn_module(void);
```

3.2.9.5 bls_ll_setAdvData

Please refer to “Core_v5.0” (Vol 2/Part E/ 7.8.7 “LE Set Advertising Data Command”).

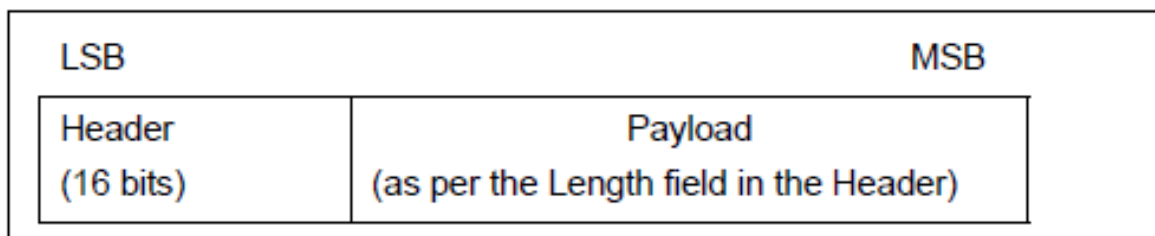


Figure 3.33: “Adv Packet Format in BLE Stack”

As shown above, an Adv packet in BLE stack contains 2-byte header, and Payload (PDU). The maximum length of Payload is 31 bytes.

The API below serves to set PDU data of adv packet:

```
ble_sts_t bls_ll_setAdvData(u8 *data, u8 len);
```

The “data” pointer points to the starting address of the PDU, while the “len” indicates data length.

The table below lists possible results for the return type “ble_sts_t”.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Len exceeds the maximum length 31

This API can be invoked during initialization to set adv data, or invoked in main_loop to modify adv data when firmware is running.

In the “feature_backup” project of this BLE SDK, Adv PDU definition is shown as below. Please refer to “Data Type Specification” in BLE Spec “CSS v6” (Core Specification Supplement v6.0) for introduction to various fields.

```
const u8 tbl_advData[] = {
    0x08, 0x09, 'f', 'e', 'a', 't', 'u', 'r', 'e',
    0x02, 0x01, 0x05,
    0x03, 0x19, 0x80, 0x01,
    0x05, 0x02, 0x12, 0x18, 0x0F, 0x18,
};
```

As shown in the adv data above, the adv device name is set as “feature”.

3.2.9.6 bls_ll_setScanRspData

Please refer to “Core_v5.0” (Vol 2/Part E/ 7.8.8 “LE Set Scan response Data Command”).

The API below serves to set PDU data of scan response packet.

```
ble_sts_t bls_ll_setScanRspData(u8 *data, u8 len);
```

The “data” pointer points to the starting address of the PDU, while the “len” indicates data length.

The table below lists possible results for the return type “ble_sts_t”.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Len exceeds the maximum length 31

The user can call this API to set the Scan response data during initialization, or call this API in the main_loop at any time while the program is running to modify the Scan response data. The scan response data defined in the B91 ble remote project in the BLE SDK is as follows, and the scan device name is "Eaglerc". For the meaning of each field, please refer to the specific description of Data Type Specification in the document BLE Spec "CSS v6" (Core Specification Supplement v6.0).

```
const u8    tbl_scanRsp [] = { 0x08, 0x09, 'V', 'R', 'e', 'm', 'o', 't', 'e',};
```

The device name is set in the advertising data and scan response data above and is not the same. Then when scanning a Bluetooth device on a mobile phone or IOS system, the device name may be different:

- Some devices only watch broadcast packets, then the displayed device name is "feature";
- After seeing the broadcast, some devices send scan request and read back the scan response, then the displayed device name may be "VRemote".

The user can also write the same device name in these two packages, and two different names will not be displayed when scanned.

In fact, after the device is connected by the master, when the master reads the Attribute Table of the device, it will obtain the gap device name of the device. After connecting to the device, it may also display the device name according to the settings there.

3.2.9.7 bls_ll_setAdvParam

Please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.5 "LE Set Advertising Parameters Command").

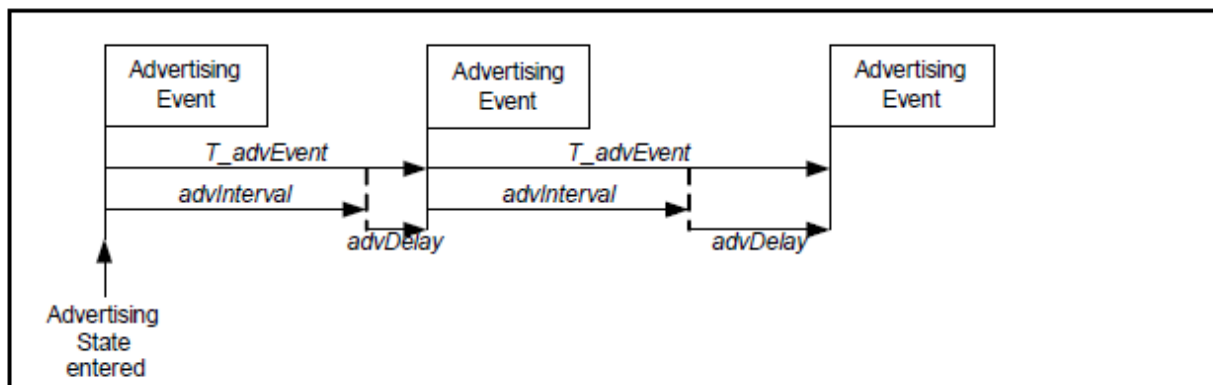


Figure 3.34: "Advertising Event in BLE Stack"

The figure above shows Advertising Event (Adv Event in brief) in BLE stack. It indicates during each T_advEvent, Slave implements one advertising process, and sends one packet in three advertising channels (channel 37, 38 and 39) respectively.

The API below serves to set parameters related to Adv Event.

```
ble_sts_t  ble_ll_setAdvParam( u16 intervalMin, u16 intervalMax, adv_type_t advType,
↪  own_addr_type_t ownAddrType, u8 peerAddrType, u8 *peerAddr,  adv_chn_map_t
↪  adv_channelMap,  adv_fp_type_t  advFilterPolicy);
```

(1) intervalMin & intervalMax

The two parameters serve to set the range of advertising interval in integer multiples of 0.625ms. The valid range is from 20ms to 10.24s, and intervalMin should not exceed intervalMax.

As required by BLE spec, it's not recommended to set adv interval as fixed value; in Telink BLE SDK, the eventual adv interval is random variable within the range of intervalMin ~ intervalMax. If intervalMin and intervalMax are set as same value, adv interval will be fixed as the intervalMin.

Adv packet type has limits to the setting of intervalMin and intervalMax. Please refer to "Core 5.0" (Vol 6/ Part B/ 4.4.2.2 "Advertising Events") for details.

(2) advType

AS per BLE spec, the following four basic advertising event types are supported.

Advertising Event Type	PDU used in this advertising event type	Allowable response PDUs for advertising event	
		SCAN_REQ	CONNECT_REQ
Connectable Undirected Event	ADV_IND	YES	YES
Connectable Directed Event	ADV_DIRECT_IND	NO	YES*
Non-connectable Undirected Event	ADV_NONCONN_IND	NO	NO
Scannable Undirected Event	ADV_SCAN_IND	YES	NO

Table 4.1: Advertising event types, PDUs used and allowable response PDUs

Figure 3.35: "Four Adv Events in BLE Stack"

In the "Allowable response PDUs for advertising event" column, "YES" and "NO" indicate whether corresponding adv event type can respond to "Scan request" and "Connect Request" from other device. For example, "Connectable Undirected Event" can respond to both "Scan request" and "Connect Request", while "Non-connectable Undirected Event" will respond to neither "Scan request" nor "Connect Request".

For "Connectable Directed Event", "YES" marked with an asterisk indicates the matched "Connect Request" received won't be filtered by whitelist and this event will surely respond to it. Other "YES" not marked with asterisk indicate corresponding request can be responded depending on the setting of whitelist filter.

The "Connectable Directed Event" supports two sub-types including "Low Duty Cycle Directed Advertising" and "High Duty Cycle Directed Advertising". Therefore, five types of adv events are supported in all, as defined below.


```

/* Advertisement Type */
typedef enum{
    ADV_TYPE_CONNECTABLE_UNDIRECTED          = 0x00,  // ADV_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY  = 0x01,  // ADV_INDIRECT_IND (high duty cycle)
    ADV_TYPE_SCANNABLE_UNDIRECTED            = 0x02,  // ADV_SCAN_IND
    ADV_TYPE_NONCONNECTABLE_UNDIRECTED       = 0x03,  // ADV_NONCONN_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY   = 0x04,  // ADV_INDIRECT_IND (low duty cycle)
}adv_type_t;

```

By default, the most common adv event type is “ADV_TYPE_CONNECTABLE_UNDIRECTED”.

(3) ownAddrType

There are four optional values for “ownAddrType” to specify adv address type.

```

/* Own Address Type */
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;

```

First two parameters are explained herein.

The “OWN_ADDRESS_PUBLIC” indicates that public MAC address is used during advertising. Actual address is the setting from the API “blc_ll_initAdvertising_module(u8 *public_addr)” during MAC address initialization.

The “OWN_ADDRESS_RANDOM” indicates random static MAC address is used during advertising, and the address comes from the setting of the API below:

```
ble_sts_t  blc_ll_setRandomAddr(u8 *randomAddr);
```

(4) peerAddrType & *peerAddr

When advType is set as directed adv type (ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY or ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY), the “peerAddrType” and “*peerAddr” serve to specify the type and address of peer device MAC Address.

When advType is set as type other than directed adv, the two parameters are invalid, and they can be set as “0” and “NULL”.

(5) adv_channelMap

The “adv_channelMap” serves to set advertising channel. It can be selectable from channel 37, 38, 39 or combination.

```
typedef enum{
    BLT_ENABLE_ADV_37    =    BIT(0),
    BLT_ENABLE_ADV_38    =    BIT(1),
    BLT_ENABLE_ADV_39    =    BIT(2),
    BLT_ENABLE_ADV_ALL   =    (BLT_ENABLE_ADV_37 | BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39),
}adv_chn_map_t;
```

(6) advFilterPolicy

The “advFilterPolicy” serves to set filtering policy for scan request/connect request from other device when adv packet is transmitted. Address to be filtered needs to be pre-loaded in whitelist.

Filtering type options are shown as below. The “ADV_FP_NONE” can be selected if whitelist filter is not needed.

```
typedef enum {
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY    =    0x00,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_ANY     =    0x01,
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_WL     =    0x02,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL      =    0x03,
    ADV_FP_NONE =    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY,
} adv_fp_type_t; //adv_filterPolicy_type_t
```

The table below lists possible values and reasons for the return value “ble_sts_t”.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	The intervalMin or intervalMax value does not meet the requirement of BLE spec.

According to Host command design in HCI part of BLE spec, eight parameters are configured simultaneously by the “bls_ll_setAdvParam” API. This setting also takes some coupling parameters into consideration. For example, the “advType” has limits to the setting of intervalMin and intervalMax, and range check depends on the advType; if advType and advInterval are set in two APIs, the range check is uncontrollable.

However, considering that the user may modify some common parameters frequently and does not want to call bls_ll_setAdvParam every time to set 8 parameters at the same time, the SDK wraps 4 of the parameters that will not be coupled with other parameters separately to facilitate the use of the user. The three separately wrapped APIs are as follows.

```
ble_sts_t    bls_ll_setAdvInterval(u16 intervalMin, u16 intervalMax);
ble_sts_t    bls_ll_setAdvChannelMap(u8 adv_channelMap);
ble_sts_t    bls_ll_setAdvFilterPolicy(u8 advFilterPolicy);
```

These 3 API parameters are the same as in bls_ll_setAdvParam.

Return value ble_sts_t:

- 1) bls_ll_setAdvChannelMap and bls_ll_setAdvFilterPolicy will return BLE_SUCCESS unconditionally.
- 2) bls_ll_setAdvInterval will return BLE_SUCCESS or HCI_ERR_INVALID_HCI_CMD_PARAMS.

3.2.9.8 bls_ll_setAdvEnable

Please refer to “Core_v5.0” (Vol 2/Part E/ 7.8.9 “LE Set Advertising Enable Command”).

```
ble_sts_t bls_ll_setAdvEnable(int en);
```

en”: 1 - Enable Advertising; 0 - Disable Advertising.

- a) In Idle state, by enabling Advertising, Link Layer will enter Advertising state.
- b) In Advertising state, by disabling Advertising, Link Layer will enter Idle state.
- c) In other states, Link Layer state won’t be influenced by enabling or disabling Advertising.

Note:

- Note that at any time this function is called, ble_sts_t unconditionally returns BLE_SUCCESS, which means that the adv-related parameters will be turned on or off internally, but only if they are in idle or adv state.

3.2.9.9 bls_ll_setAdvDuration

```
ble_sts_t bls_ll_setAdvDuration (u32 duration_us, u8 duration_en);
```

After the “bls_ll_setAdvParam” is invoked to set all adv parameters successfully, and the “bls_ll_setAdvEnable (1)” is invoked to start advertising, the API “bls_ll_setAdvDuration” can be invoked to set duration of adv event, so that advertising will be automatically disabled after this duration.

“duration_en”: 1-enable timing function; 0-disable timing function. “duration_us”: The “duration_us” is valid only when the “duration_en” is set as 1, and it indicates the advertising duration in unit of us.

When this duration expires, “AdvEnable” becomes invalid, and advertising is stopped. None Conn state will switch to Idle State. The Link Layer event “BLT_EV_FLAG_ADV_DURATION_TIMEOUT” will be triggered.

As specified in BLE spec, for the adv type “ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY”, the duration time is fixed as 1.28s, i.e. advertising will be stopped after the 1.28s duration. Therefore, for this adv type, the setting of “bls_ll_setAdvDuration” won’t take effect.

The return value “ble_sts_t” is shown as below.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Duration Time can’t be configured for “ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY”.

When Adv Duration Time expires, advertising is stopped, if user needs to re-configure adv parameters (such as AdvType, AdvInterval, AdvChannelMap), first the parameters should be set in the callback function of the event "BLT_EV_FLAG_ADV_DURATION_TIMEOUT", then the "bls_ll_setAdvEnable (1)" should be invoked to start new advertising.

To trigger the "BLT_EV_FLAG_ADV_DURATION_TIMEOUT", a special case should be noted:

Suppose the "duration_us" is set as "2000000" (i.e. 2s).

If Slave stays in advertising state, when adv time reaches the preset 2s timeout, the "BLT_EV_FLAG_ADV_DURATION_TIMEOUT" will be triggered to execute corresponding callback function.

If Slave is connected with Master when adv time is less than the 2s timeout (suppose adv time is 0.5s), the timeout timing is not cleared but cached in bottom layer. When Slave stays in connection state for 1.5s (i.e. the preset 2s timeout moment is reached), since Slave won't check adv event timeout in connection state, the callback of "BLT_EV_FLAG_ADV_DURATION_TIMEOUT" won't be triggered.

Note:

- When Slave stays in connection state for certain duration (e.g. 10s), then terminates connection and returns to adv state, before it sends out the first adv packet, the Stack will regard current time exceeds the preset 2s timeout and trigger the callback of "BLT_EV_FLAG_ADV_DURATION_TIMEOUT". In this case, the callback triggering time largely exceeds the preset timeout moment.

3.2.9.10 blc_ll_setAdvCustomedChannel

The API below serves to customize special advertising channel/scanning channel, and it only applies some special applications such as BLE mesh. It's not recommended to use this API for other conventional application cases.

```
void blc_ll_setAdvCustomedChannel (u8 chn0, u8 chn1, u8 chn2);
```

chn0/chn1/chn2: customized channel. Default standard channel is 37/38/39. For example, to set three advertising channels as 2420MHz, 2430MHz and 2450MHz, the API below should be invoked:

```
blc_ll_setAdvCustomedChannel (8, 12, 22);
```

3.2.9.11 rf_set_power_level_index

This BLE SDK supplies the API to set output power for BLE RF packet, as shown below.

```
void rf_set_power_level_index (rf_power_level_index_e level)
```

The "level" is selectable from the corresponding enum variable rf_power_level_index_e in the "drivers/8258(8278)/rf_drv.h".

The Tx power configured by this API will take effect for both adv packet and conn packet, and it can be set freely in firmware. The actual Tx power will be determined by the latest setting. Please note that

the "rf_set_power_level_index" configures registers related to MCU RF. Once MCU enters sleep (suspend/deepsleep retention), these registers' values will be lost, so they should be reconfigured after each wakeup. For example, SDK demo employs the event callback "BLT_EV_FLAG_SUSPEND_EXIT" to guarantee RF power is recovered after wakeup from sleep.

```
_attribute_ram_code_ void    user_set_rf_power (u8 e, u8 *p, int n)
{
    rf_set_power_level_index (MY_RF_POWER_INDEX);
}
user_set_rf_power(0, 0, 0);
bls_app_registerEventCallback (BLT_EV_FLAG_SUSPEND_EXIT, &user_set_rf_power);
```

3.2.9.12 blc_ll_setScanParameter

Please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.10 "LE Set Scan Parameters Command").

```
ble_sts_t  blc_ll_setScanParameter (u8 scan_type,
u16 scan_interval, u16 scan_window,
own_addr_type_t ownAddrType,
scan_fp_type_t scanFilter_policy);
```

Parameter analysis:

1) scan_type

You can choose between passive scan and active scan, the difference is that active scan will send scan_req on top of the adv packet to get more information about the device scan_rsp, and the scan_rsp packet will also be passed to the BLE Host via adv report event; passive scan does not send a scan_req.

```
typedef enum {
    SCAN_TYPE_PASSIVE = 0x00,
    SCAN_TYPE_ACTIVE,
} scan_type_t;
```

2) scan_interval/scan window

The scan_interval sets the Scanning state frequency switching time in 0.625ms. The scan_window is not handled in the Telink BLE SDK at the moment, the actual scan window is set to scan_interval.

3) ownAddrType

When specifying the scan_req packet address type, the 4 optional values for ownAddrType are as follows.

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

OWN_ADDRESS_PUBLIC means that the public MAC address is used for Scan, the actual address comes from the settings during the MAC address initialisation API `blc_initMacAddress(flash_sector_mac_address, mac_public, mac_random_static)`.

OWN_ADDRESS_RANDOM indicates that a random static MAC address is used for Scan, which is derived from the value set by the following API.

```
ble_sts_t    blc_ll_setRandomAddr(u8 *randomAddr);
```

4) scan filter policy

The current supported scan filter policy are as follows:

```
typedef enum {  
    SCAN_FP_ALLOW_ADV_ANY           =0x00,  
    SCAN_FP_ALLOW_ADV_WL           =0x01,  
    SCAN_FP_ALLOW_UNDIRECT_ADV      =0x02,  
    SCAN_FP_ALLOW_ADV_WL_DIRECT_ADV_MACTH =0x03,  
} scan_fp_type_t;
```

SCAN_FP_ALLOW_ADV_ANY means that the Link Layer does not filter the adv packets from scan and reports them directly to the BLE Host.

SCAN_FP_ALLOW_ADV_WL requires that the adv packets scanned must be in the whitelist before they are reported to the BLE Host.

The return value `ble_sts_t` is only `BLE_SUCCESS`, the API does not check the reasonableness of the parameters, the user needs to pay attention to the reasonableness of the parameters set.

3.2.9.13 blc_ll_setScanEnable

Please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.11 "LE Set Scan Enable Command").

```
ble_sts_t    blc_ll_setScanEnable (scan_en_t scan_enable, dupFilter_en_t filter_duplicate);
```

The `scan_enable` parameter type has the following 2 optional values.

```
typedef enum {  
    BLC_SCAN_DISABLE = 0x00,  
    BLC_SCAN_ENABLE  = 0x01,  
} scan_en_t;
```

When `scan_enable` is 1, Enable Scanning; when `scan_enable` is 0, Disable Scanning.

1) In Idle state, Enable Scanning, Link Layer enters Scanning state.

2) In Scanning state, Disable Scanning, Link layer enters Idle state.

The `filter_duplicate` parameter type has 2 optional values as follows.

```
typedef enum {
    DUP_FILTER_DISABLE    = 0x00,
    DUP_FILTER_ENABLE     = 0x01,
} dupFilter_en_t;
```

When filter_duplicate is 1, duplicate packet filtering is enabled, and the Controller will only report the adv report event to the Host once for each different adv packet; when filter_duplicate is 0, duplicate packet filtering is not enabled, and the adv packet scanned to the Host will always be reported to the Host.

The return value ble_sts_t is as below.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
LL_ERR_CURRENT_STATE_NOT_SUPPORTED_THIS_CMD	See definition in SDK	Link Layer is in BLS_LINK_STATE_ADV / BLS_LINK_STATE_CONN state

When scan_type is set to active scan and Enable Scanning, for each device, scan_rsp is only read once and reported to the Host. Because after each Enable Scanning, the Controller will record the scan_rsp of different devices and store them in the scan_rsp list to ensure that the device's scan_req is not read again later.

If the user needs to report the scan_rsp of the same device multiple times, this can be achieved by setting Enable Scanning repeatedly via blc_ll_setScanEnable, as the device's scan_rsp list is cleared to 0 each time Enable/Disable Scanning is performed.

3.2.9.14 blc_ll_createConnection

Please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.12 "LE Create Connection Command").

```
ble_sts_t blc_ll_createConnection (u16 scan_interval, u16 scan_window,    init_fp_type_t
    ↪ initiator_filter_policy,
u8 adr_type, u8 *mac, u8 own_adr_type,
u16 conn_min, u16 conn_max, u16 conn_latency, u16 timeout, u16 ce_min, u16 ce_max)
```

1) scan_interval/scan window

scan_interval sets the Scan frequency switching time in the Initiating state, in 0.625ms.

scan_window is not handled in the Telink BLE SDK at the moment, the actual scan window is set to scan_interval.

2) initiator_filter_policy

Specify the policy for the currently connected device, either of the following two options.

```
typedef enum {
    INITIATE_FP_ADV_SPECIFY = 0x00, //connect ADV specified by host
    INITIATE_FP_ADV_WL = 0x01, //connect ADV in whitelist
} init_fp_type_t;
```

INITIATE_FP_ADV_SPECIFY means that the connected device address is the adr_type/mac that follows.

INITIATE_FP_ADV_WL means that the connection is based on the device inside the whitelist, at which point adr_type/mac is meaningless.

3) adr_type/ mac

When initiator_filter_policy is INITIATE_FP_ADV_SPECIFY, a device with address type adr_type (BLE_ADDR_PUBLIC or BLE_ADDR_RANDOM) and address mac[5...0] is connected.

4) own_adr_type

Specifies the type of MAC address used by the Master role that establishes the connection. For ownAddrType, the four optional values are as follows.

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

OWN_ADDRESS_PUBLIC means that a public MAC address is used when connecting, the actual address is from the API blc_ll_initStandby_module (u8 *public_adr) set during MAC address initialization.

OWN_ADDRESS_RANDOM indicates that a random static MAC address is used when connecting, which is derived from the value set by the following API.

```
ble_sts_t blc_ll_setRandomAddr(u8 *randomAddr);
```

5) conn_min/ conn_max/ conn_latency/ timeout

These 4 parameters specify the connection parameters for the Master role once the connection is established, and these parameters are also sent to the Slave via the connection request, which will also have the same connection parameters.

conn_min/conn_max specifies the range of conn interval. The Telink BLE SDK for Master role Single Connection uses the value of conn_min directly. The unit is 0.625ms.

conn_latency specifies the connection latency, usually set to 0. timeout specifies the connection supervision timeout, in 10ms.

6) ce_min/ ce_max

The SDK does not handle ce_min/ ce_max yet.

The return value table is as below.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	Link Layer is in Initiating state, no receiving new create connection
HCI_ERR_CONTROLLER_BUSY	0x3A	Link Layer is in Advertising state or Connection state

3.2.9.15 blc_ll_setCreateConnectionTimeout

```
ble_sts_t blc_ll_setCreateConnectionTimeout (u32 timeout_ms);
```

The return value is BLE_SUCCESS and the timeout_ms unit is ms.

According to the Link Layer state machine, when blc_ll_createConnection triggers the Idle state/Scanning state to enter the Initiating state, if the connection cannot be established for a long time, it will trigger Initiate timeout and exit the Initiating state.

Each time blc_ll_createConnection is called, the SDK defaults to the current Initiate timeout time of connection supervision timeout *2. If the User does not want to use the SDK default timeout, they can call blc_ll_createConnection immediately after the blc_ll_setCreateConnectionTimeout immediately after createConnection to set the desired Initiate timeout.

3.2.9.16 blm_ll_updateConnection

Please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.18 "LE Connection Update Command").

```
ble_sts_t blm_ll_updateConnection (u16 connHandle,
                                   u16 conn_min, u16 conn_max, u16 conn_latency, u16 timeout,
                                   u16 ce_min, u16 ce_max);
```

1) connection handle

Specify the connection whose parameters need to be updated.

2) conn_min/ conn_max/ conn_latency/ timeout

Specify the connection parameter to be updated. Master role single connection currently uses conn_min directly as the new interval.

3) ce_min/ce_max

Not currently processed.

The return value ble_sts_t is only BLE_SUCCESS, the API does not check the reasonableness of the parameters, the user needs to pay attention to the reasonableness of the set parameters.

3.2.9.17 bls_ll_terminateConnection

```
ble_sts_t bls_ll_terminateConnection (u8 reason);
```

This API is used for BLE Slave device, and it only applies to Connection state Slave role.

In order to actively terminate connection, this API can be invoked by APP Layer to send a "Terminate" to Master in Link Layer. "reason" indicates reason for disconnection. Please refer to "Core_v5.0" (Vol 2/Part D/ 2 "Error Code Descriptions").

If connection is not terminated due to system operation abnormality, generally APP layer specifies the "reason" as:

```
HCI_ERR_REMOTE_USER_TERM_CONN = 0x13
bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN);
```

In bottom-layer stack of Telink BLE SDK, this API is invoked only in one case to actively terminate connection: When data packets from peer device are decrypted, if an authentication data MIC error is detected, the "bls_ll_terminateConnection(HCI_ERR_CONN_TERM_MIC_FAILURE)" will be invoked to inform the peer device of the decryption error, and connection is terminated.

After Slave invokes this API to actively initiate disconnection, the event "BLT_EV_FLAG_TERMINATE" will be triggered. The terminate reason in the callback function of this event will be consistent with the reason manually configured in this API.

In Connection state Slave role, generally connection will be terminated successfully by invoking this API; however, in some special cases, the API may fail to terminate connection, and the error reason will be indicated by the return value "ble_sts_t". It's recommended to check whether the return value is "BLE_SUCCESS" when this API is invoked by APP layer.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer is not in Connection state Slave role
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy (mass data are being transferred), this command cannot be accepted for the moment.

3.2.9.18 blm_ll_disconnect

```
ble_sts_t blm_ll_disconnect (u16 handle, u8 reason);
```

This API is used for BLE Master devices and is only available for the Connection Master role.

It is the same as the API `bls_ll_terminateConnection` for the Slave role, but with one additional conn handle parameter. This is because the Telink BLE SDK is designed to maintain at most a single connection for the Slave role, while the Master role is designed to have a multi connection, so the connection handle to be disconnected must be specified.

The API returns the following values.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_UNKNOWN_CONN_ID	0x02	Handle error, cannot find corresponding connection
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy (mass data are being transferred), this command cannot be accepted for the moment.

3.2.9.19 Get Connection Parameters

The following APIs serves to obtain current connection paramters including Connection Interval, Connection Latency and Connection Timeout (only apply to Slave role).

```
u16      bls_ll_getConnectionInterval(void);
u16      bls_ll_getConnectionLatency(void);
u16      bls_ll_getConnectionTimeout(void);
```

- If return value is 0, it indicates current Link Layer state is None Conn state without connection parameters available.
- The returned non-zero value indicates the corresponding parameter value.
 - Actual conn interval divided by 1.25ms will be returned by the API "bls_ll_getConnectionInterval". Suppose current conn interval is 10ms, the return value should be 10ms/1.25ms=8.
 - Actual Latency value will be returned by the API "bls_ll_getConnectionLatency".
 - Actual conn timeout divided by 10ms will be returned by the API "bls_ll_getConnectionTimeout". Suppose current conn timeout is 1000ms, the return value would be 1000ms/10ms=100.

3.2.9.20 blc_ll_getCurrentState

The API below serves to obtain current Link Layer state.

```
u8  blc_ll_getCurrentState(void);
```

The user determines the current state at the application level, e.g.

```
if(blc_ll_getCurrentState() == BLS_LINK_STATE_ADV)
if(blc_ll_getCurrentState() == BLS_LINK_STATE_CONN)
```

3.2.9.21 blc_ll_getLatestAvgRSSI

The API serves to obtain latest average RSSI of connected peer device after Link Layer enters Slave role or Master role.

```
u8      blc_ll_getLatestAvgRSSI(void)
```

The return value is u8-type rssi_raw, and the real RSSI should be: rssi_real = rssi_raw- 110. Suppose the return value is 50, rssi = -60 db.

3.2.9.22 Whitelist & Resolvinglist

As introduced above, "filter_policy" of Advertising/Scanning/Initiating state involves Whitelist, and actual operation may depend on devices in Whitelist. Actually Whitelist contains two parts: Whitelist and Resolvinglist.

User can check whether address type of peer device is RPA (Resolvable Private Address) via "peer_addr_type" and "peer_addr". The API below can be invoked directly.

```
#define IS_NON_RESOLVABLE_PRIVATE_ADDR(type, addr)
( (type)==BLE_ADDR_RANDOM && (addr[5] & 0xC0) == 0x00 )
```

Only non-RPA address can be stored in whitelist. In current SDK, whitelist can store up to four devices.

```
#define      MAX_WHITE_LIST_SIZE      4
```

Related interface:

```
ble_sts_t ll_whitelist_reset(void);
```

The return value of reset whitelist is "BLE_SUCCESS".

```
ble_sts_t ll_whitelist_add(u8 type, u8 *addr);
```

Add a device into whitelist, the return value is shown as below.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Add success
HCI_ERR_MEM_CAP_EXCEEDED	0x07	Whitelist is already full, add failure

```
ble_sts_t ll_whitelist_delete(u8 type, u8 *addr);
```

Delete a device from whitelist, the return value is "BLE_SUCCESS".

RPA (Resolvable Private Address) device needs to use Resolvinglist. To save RAM space, "Resolvinglist" can store up to two devices in current SDK.

```
#define MAX_WHITE_IRK_LIST_SIZE 2
```

Corresponding API:

```
ble_sts_t ll_resolvingList_reset(void);
```

Reset Resolvinglist, the return value is "BLE_SUCCESS".

```
ble_sts_t ll_resolvingList_setAddrResolutionEnable (u8 resolutionEn);
```

This API serves to enable/disable device address resolving for Resolvinglist. It is used for device address resolution. If you want to use Resolvinglist to resolve addresses, you must enable it. You can disable it when you do not need to parse it.

```
ble_sts_t ll_resolvingList_add(u8 peerIdAddrType, u8 *peerIdAddr,  
u8 *peer_irk, u8 *local_irk);
```

This API serves to add device using RPA address into Resolvinglist, peerIdAddrType/ peerIdAddr and peer_irk indicate identity address and irk declared by peer device. These information will be stored into flash during pairing encryption process, and corresponding interfaces to obtain the info are available in SMP part. "local_irk" is not processed in current SDK, and it can be set as "NULL".

```
ble_sts_t ll_resolvingList_delete(u8 peerIdAddrType, u8 *peerIdAddr);
```

This API serves to delete a RPA device from Resolvinglist.

For usage of address filter based on Whitelist/Resolvinglist, please refer to "TEST_WHITELIST" in feature test demo of the SDK.

3.2.9.23 blc_att_setServerDataPendingTime_upon_ClientCmd

In the device after the Client has just started connecting to do SDP, at this time the discovered Server needs to reply according to its own service table in time when it receives the relevant query function, and the TX buffer is in a very tight state. Therefore if the user goes to send data at this time, it is easy to fail because the RF tx_buffer is full.

We therefore recommend using a controlled pending time to avoid this problem, where the relevant data sending action will take place after the SDP is completed and the data is pending until then, via the api blc_att_setServerDataPendingTime_upon_ClientCmd(u8 num_10ms) to modify the time with a parameter step of 10ms.

3.2.10 Coded PHY/2M PHY

3.2.10.1 Coded PHY/2M PHY Introduction

Coded PHY and 2M PHY are new features to "Core_5.0", this expands the BLE application scenario, Coded PHY includes S2 (500kbps) and S8 (125kbps) in order to support long range application. 2M PHY (2Mbps) improved the BLE bandwidth. Coded PHY and 2M PHY could be used under both the advertising channel and data channel when in connected state. Connected state application will be introduced in the following section, advertising channel application will be introduced in "Extended Advertising".

3.2.10.2 Coded PHY/2M PHY Demo Introduction

In the B85 BLE SDK, in order to save the sram space, Code PHY and 2M PHY is disabled by default. If user wants to enable this feature, you can enable it manually. You can refer to the BLE SDK demo:

- Slave end reference Demo "b85m_feature_test"

Define macro in vendor/b85m_feature_test/feature_config.h

```
#define FEATURE_TEST_MODE    TEST_2M_CODED_PHY_CONNECTION
```

- Master end reference Demo "b85m_master_kma_dongle"

Users can also choose to use other manufacturers' devices, as long as they support Coded PHY/2M PHY, they can interconnect with Telink's Slave devices.

If using Telink's SDK, Coded PHY and 2M PHY are also disabled by default on the Master end and need to be enabled by the following method.

Add API to the function void user_init(void) in vendor/b85m_master_kma_dongle/app.c (disabled by default in SDK).

```
blc_ll_init2MPhyCodedPhy_feature();
```

3.2.10.3 Coded PHY/2M PHY API Introduction

- (1) API

```
void blc_ll_init2MPhyCodedPhy_feature(void)
```

is used to enable Code PHY and 2M PHY.

- (2) A new event - BLT_EV_FLAG_PHY_UPDATE is introduced to Telink Defined Event in order to support Coded and 2M PHY, the detail implementation could refer to section "Controller Event".
- (3) API:

```
ble_sts_t blc_ll_setPhy (u16 connHandle, le_phy_prefer_mask_t all_phys, le_phy_prefer_type_t
↳ tx_phys, le_phy_prefer_type_t rx_phys, le_ci_prefer_t phy_options);
```

This is a BLE Spec standard interface, please refer to <Core_5.0>, Vol 2/Part7/7.8.49, "LE Set PHY Command".

connHandle: slave mode: it should set to BLS_CONN_HANDLE; master mode: it should set to BLM_CONN_HANDLE.

For other parameters, please refer to Spec's definition along with SDK's enumeration definition.

(4) API blc_ll_setDefaultConnCodingIndication()

```
ble_sts_t blc_ll_setDefaultConnCodingIndication(le_ci_prefer_t prefer_CI);
```

Non-BLE Spec standard interface, when a Peer Device initiates a PHY_Req request via API blc_ll_setPhy (), the requested party can set the local device's preferred Encode Mode (S2/S8) via this API.

3.2.11 Channel Selection Algorithm #2

Channel Selection Algorithm #2 is a new feature added to Core_5.0, with a better interference avoidance capability. You can refer to Core_5.0 (Vol 6/Part B/4.5.8.3 "Channel Selection Algorithm #2") for further information.

The corresponding demo reference in BLE SDK.

- Slave end refer to Demo "b85m_feature_test"

Define macro in vendor/b85m_feature_test/feature_config.h as below:

```
#define FEATURE_TEST_MODE TEST_CSA2
```

- If using a broadcast defined by the Core_4.2 API, the user can choose to use or not to use the frequency hopping algorithm #2, which is not used by default in the SDK. If you want to use the frequency hopping algorithm #2, you need to enable it via the following API.

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void)
```

- If using <Core_5.0> extended advertising and initiate connect through Extend ADV, user will have to use above API to choose Algorithm #2 according to the spec <Core_5.0>. Because if the connection is initiated through Extended Adv, it'll choose Algorithm#2 by default, and on the othe hand, if only uses advertising, in order to save sram space, Algorithm #2 is not recommended.

- Master end refer to Demo "b85m_master_kma_dongle"

By default the master end frequency hopping algorithm #2 is also disabled, if needed the same API has to be enabled manually in the user_init() call.

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void);
```

3.2.12 Extended Advertising

3.2.12.1 Extended Advertising Introduction

Extended Advertising is a new feature to <Core_5.0>

Due to the new feature to Advertising in <Core_5.0>, SDK has new APIs in order to support the legacy Advertising function in <Core_4.2> and the new Advertising function in <Core_5.0>. These APIs will be covered in later sections, named as <Core_5.0> API. (following section will use this name as reference), and <Core_4.2> APIs referred in section, like `bls_ll_setAdvData()`, `bls_ll_setScanRspData()`, `bls_ll_setAdvParam()`, will only support for <Core_4.2>'s Advertising function, but not <Core_5.0> Advertising new function.

Extended Advertising primary feature as following:

- (1) Increase the Advertising PDUs – In <Core_4.2>, the Advertising PDU length is ranging from 6 to 37 bytes, and in <Core_5.0>, the extended Advertising PDU is ranging from 0 to 255 bytes (single PDU). If the Advertising Data length > Adv PDU, it'll be fragmented into N Advertising PDU and send it out.
- (2) It could chose different PHYs (1Mbps, 2Mbps, 125kbps, 500kbps) based on different application.

3.2.12.2 Extended Advertising Demo Setup

Extended Advertising Demo "b85m_feature_test" usage:

Demo1: use to illustrate all the basic advertising functions in <Core_5.0>

- a) Define macro in vendor/b85m_feature_test/feature_config.h

```
#define FEATURE_TEST_MODE    TEST_EXTENDED_ADVERTISING
```

- b) Based on the type of Advertising, select the corresponding macro. The demo could also test all the supported Advertising type in <Core_5.0>, below are all the type that B91 SDK currently supported.

```
/* Advertising Event Properties type*/
typedef enum{
    ADV_EVT_PROP_LEGACY_CONNECTABLE_SCANNABLE_UNDIRECTED        = 0x0013,
    ADV_EVT_PROP_LEGACY_CONNECTABLE_DIRECTED_LOW_DUTY           = 0x0015,
    ADV_EVT_PROP_LEGACY_CONNECTABLE_DIRECTED_HIGH_DUTY          = 0x001D,
    ADV_EVT_PROP_LEGACY_SCANNABLE_UNDIRECTED                     = 0x0012,
    ADV_EVT_PROP_LEGACY_NON_CONNECTABLE_NON_SCANNABLE_UNDIRECTED = 0x0010,
    ADV_EVT_PROP_EXTENDED_NON_CONNECTABLE_NON_SCANNABLE_UNDIRECTED = 0x0000,
    ADV_EVT_PROP_EXTENDED_CONNECTABLE_UNDIRECTED                 = 0x0001,
    ADV_EVT_PROP_EXTENDED_SCANNABLE_UNDIRECTED                   = 0x0002,
    ADV_EVT_PROP_EXTENDED_NON_CONNECTABLE_NON_SCANNABLE_DIRECTED = 0x0004,
    ADV_EVT_PROP_EXTENDED_CONNECTABLE_DIRECTED                   = 0x0005,
    ADV_EVT_PROP_EXTENDED_SCANNABLE_DIRECTED                     = 0x0006,
```



```
ADV_EVT_PROP_EXTENDED_MASK_ANONYMOUS_ADV          = 0x0020,
ADV_EVT_PROP_EXTENDED_MASK_TX_POWER_INCLUDE       = 0x0040,
}advEvtProp_type_t;
```

Demo2: Based on Demo1, enable the Coded PHY/2M PHY option

a) Define macro in vendor/b85m_feature_test/feature_config.h

```
#define FEATURE_TEST_MODE    TEST_2M_CODED_PHY_EXT_ADV
```

b) Based on the type of required packet and PHY mode, select the corresponding macro to enable the functions.

Note:

- When compiling a demo, if the error shown below occurs, it may be because the data size of the defined "attribute_data_retention" attribute exceeds 16K, whereas the SDK default is deepsleep retention 16K sram.

```
C:\TelinkSDK1.3\opt\tc32\bin\tc32-elf-ld.exe: section .text loaded at [00004000,0000ab8b] overlaps section .retention_data loaded at [000037a0,00004fd7]
make: *** [ble_1c_multimode.elf] Error 1
```

Figure 3.36: "Error in compiling a demo"

This can be modified in one of the following ways (for a detailed analysis please refer to the section "Sram and Firmware Space")

Reduce the data in the defined "attribute_data_retention" attribute.

Choose to switch to deepsleep retention 32K Sram, see the chapter "Software bootloader introduction" for details on how to configure it.

3.2.12.3 Extended Advertising Related API

Extended Advertising is using module design. Due to the variable length of adv data length/scan response data where the maximum length will be up to more than 1000 bytes, instead of statically defining the maximum value in BLE stack that might waste the SRAM space, we leave the definition of SRAM space to developer, so that it would have the flexibility for user to review their needs to best use of the SRAM space.

Current SDK only support one Advertising set, but with the design that has flexibility to support multiple adv set for future as well. So you could see the APIs' parameters are all designed in the way to support multiple adv sets for future.

With that design, following are the APIs.

- (1) Initialization stage, you would need to call the following APIs to allocate the SRAM.

```
blc_ll_initExtendedAdvertising_module(app_adv_set_param, app_primary_adv_pkt,
    APP_ADV_SETS_NUMBER);
blc_ll_initExtSecondaryAdvPacketBuffer(app_secondary_adv_pkt, MAX_LENGTH_SECOND_ADV_PKT);
blc_ll_initExtAdvDataBuffer(app_advData, APP_MAX_LENGTH_ADV_DATA);
blc_ll_initExtScanRspDataBuffer(app_scanRspData, APP_MAX_LENGTH_SCAN_RESPONSE_DATA);
```

According to above API, the memory allocation is shown as below:

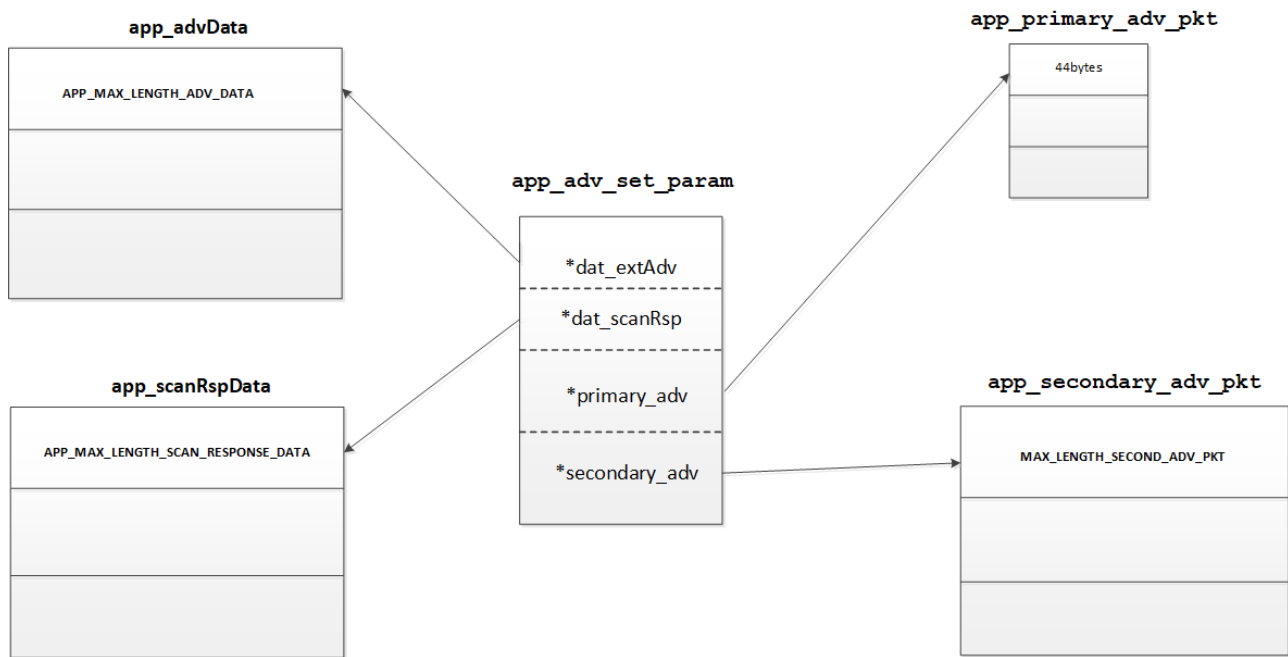


Figure 3.37: “Extended Advertising Initialize Memory Allocation”

- APP_MAX_LENGTH_ADV_DATA: Advertising Set length, developer could adjust the macro to define the size based on the needs in order to save the DeepRetention space.
- APP_MAX_LENGTH_SCAN_RESPONSE_DATA: Scan response data length, developer could adjust the macro to define the size based on the needs in order to save the DeepRetention space.
- app_primary_adv_pkt: Primary Advertising PDU data length, the size is allocated as 44 bytes, user layer can’t change it.
- app_secondary_adv_pkt: Secondary Advertising PDU data length, the size is allocated as 264 bytes, user layer can’t change it.

In the demo of “b85m_feature_test”, (vendor/b85m_feature_test/feature_extend_adv/app.c), users can use the following macro to allocate the sram based on your requirement in order to best use the sram.

```
#define APP_ADV_SETS_NUMBER          1
#define APP_MAX_LENGTH_ADV_DATA      1024
#define APP_MAX_LENGTH_SCAN_RESPONSE_DATA  31
```

(2) API blc_ll_setExtAdvParam:

```
ble_sts_t blc_ll_setExtAdvParam(……);
```

This is a BLE Spec standard interface, used to configure Advertising parameter, please refer to <Core_5.0> (Vol 2/Part E/7.8.53 “LE Set Extended Advertising Parameters Command”) for further information.

Note:

- The parameter `adv_tx_pow` does not support the selection of the send power value at the moment, you need to call the API `void rf_set_power_level_index (rf_power_level_index_e level)` separately to configure the send power.

(3) API `blc_ll_setExtScanRspData`:

```
ble_sts_t blc_ll_setExtScanRspData(u8 advHandle, data_oper_t operation, data_fragm_t  
↪ fragment_prefer, u8 scanRsp_dataLen, u8 *scanRspData);
```

This is a BLE Spec standard interface, used to configure the Scan Response Data, please refer to <Core_5.0> (Vol 2/Part E/7.8.53 “LE Set Extended Scan Response Command”).

(4) API `blc_ll_setExtAdvEnable_n`:

```
ble_sts_t blc_ll_setExtAdvEnable_n(u32 extAdv_en, u8 sets_num, u8 *pData);
```

This is a BLE Spec standard interface, used to enable/disable Extended Advertising, please refer to <Core_5.0> (Vol 2/Part E/7.8.56 “LE Set Extended Advertising Enable Command”), and understand it in the context of the SDK’s enumeration type definitions and demo usage.

However, currently the SDK only supports 1 Adv Sets, so this API is not supported for the time being and is only reserved for multiple Adv sets in the future. However, the Telink SDK has written a simplified API based on this API function to operate on/off 1 Adv Sets for more efficient execution. The simplified API is shown below, with the same input parameters and return values as the standard API, but is only used to set 1 Adv Set.

```
ble_sts_t blc_ll_setExtAdvEnable_1(u32 extAdv_en, u8 sets_num, u8 advHandle, u16 duration,  
↪ u8 max_extAdvEvt);
```

(5) API `blc_ll_setAdvRandomAddr()`

```
ble_sts_t blc_ll_setAdvRandomAddr(u8 advHandle, u8* rand_addr);
```

This is a BLE Spec standard interface for setting the device’s Random address, please refer to Core_5.0 (Vol 2/Part E/7.8.4 “LE Set Random Address Command”) for more details and combine it with the enum type definition on the SDK and demo usage to understand.

(6) API `blc_ll_setDefaultExtAdvCodingIndication`:

```
void blc_ll_setDefaultExtAdvCodingIndication(u8 advHandle, le_ci_prefer_t prefer_CI);
```

This is a Non-BLE Spec standard interface, when setting advertising parameters with BLE standard API `blc_ll_setExtAdvParam()`, if set to Coded PHY (contains S2 and S8) but does not specify which Encode mode, SDK defaults to S2, to facilitate user selection, this API is defined to select preferenced Encode mode S2/S8.

The user can pass a reference via `prefer_CI` for S2/S8 mode selection, as enumerated below.

```
typedef enum {
    CODED_PHY_PREFER_NONE    = 0,
    CODED_PHY_PREFER_S2     = 1,
    CODED_PHY_PREFER_S8     = 2,
} le_ci_prefer_t;  //LE coding indication prefer
```

(7) API `blc_ll_setAuxAdvChnIdxByCustomers`:

```
void blc_ll_setAuxAdvChnIdxByCustomers(u8 aux_chn);
```

This is a Non-BLE Spec standard interface, the user can set the channel value of the Auxiliary Advertising channel through this function, commonly used for debug, if the user does not call this function to define, the Auxiliary Advertising channel value will be generated randomly (random number range 0 - 31).

(8) API `blc_ll_setMaxAdvDelay_for_AdvEvent`:

```
void blc_ll_setMaxAdvDelay_for_AdvEvent(u8 max_delay_ms);
```

This is a non BLE Spec standard interface, used to configure the AdvDelay timing based on the Adv Interval, the input range is from 0, 1, 2, 4, 8 in the unit of ms.

```
advDelay(unit: us) = Random() % (max_delay_ms*1000);
T_advEvent = advInterval + advDelay
```

If `max_delay_ms` = 0, `T_advEvent` is accurate on the `advInterval` timing;

If `max_delay_ms` = 8, `T_advEvent` is based on the `advInterval` with a random offset in between 0-8ms.

(9) The following API, reserved for the Multiple Advertising Sets API, is not supported by this version of the SDK and can be ignored by users for the time being.

```
ble_sts_t blc_ll_removeAdvSet(u8 advHandle);
ble_sts_t blc_ll_clearAdvSets(void);
```

3.3 BLE Host

3.3.1 BLE Host Introduction

BLE Host consists of L2CAP, ATT, SMP, GATT and GAP layer, and user-layer applications are implemented on the basis of the Host layer.

3.3.2 L2CAP

The L2CAP, Logical Link Control and Adaptation Protocol, connects to the upper APP layer and the lower Controller layer. By acting as an adaptor between the Host and the Controller, the L2CAP makes data processing details of the Controller become negligible to the upper-layer application operations.

The L2CAP layer of BLE is a simplified version of classical Bluetooth. In basic mode, it does not implement segmentation and re-assembly, has no involvement of flow control and re-transmission, and only uses fixed channels for communication. The figure below shows simple L2CAP structure: Data of the APP layer are sent in packets to the BLE Controller. The BLE Controller assembles the received data into different CID data and report them to the Host layer.

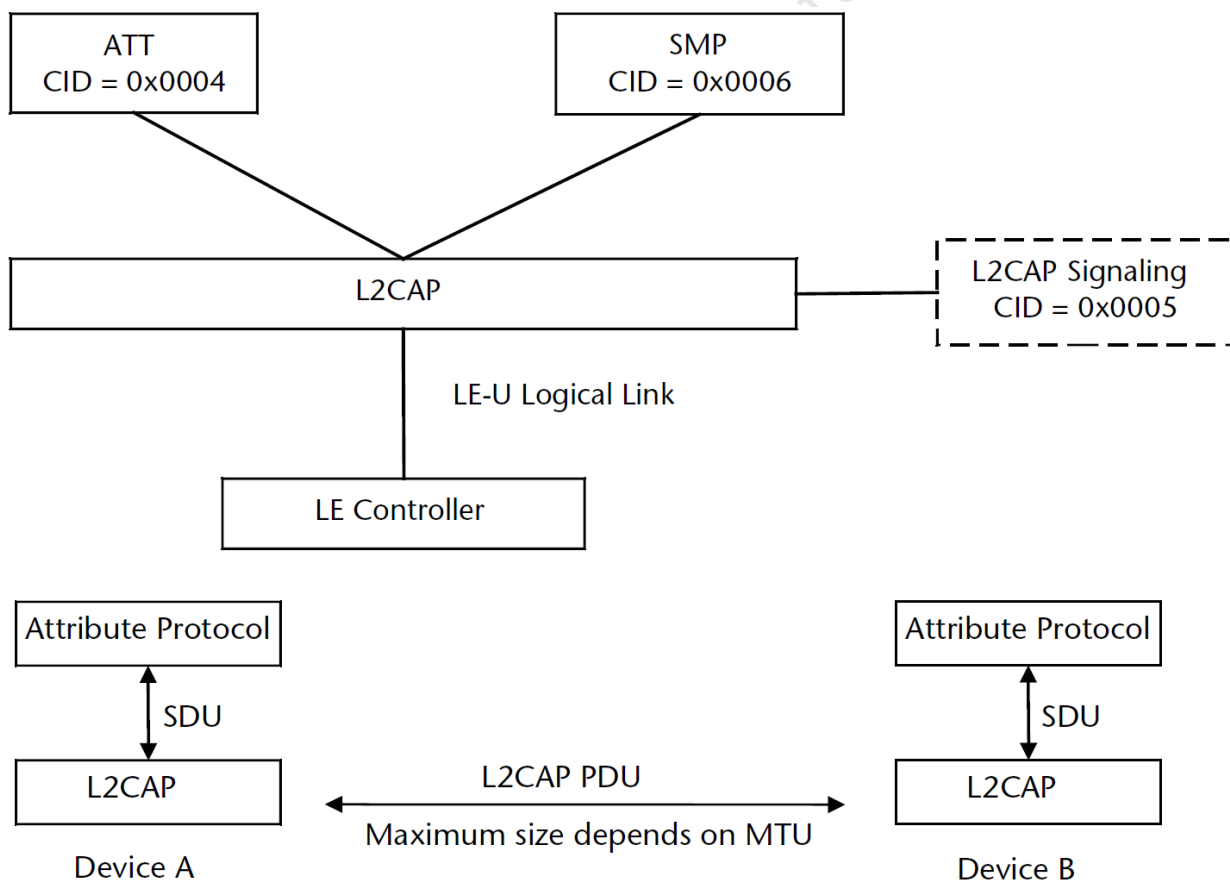


Figure 3.38: “BLE L2CAP Structure and ATT Packet Assembly Model”

As specified in BLE Spec, L2CAP is mainly used for data transfer between Controller and Host. Most work are finished in stack bottom layer with little involvement of user. User only needs to invoke the following

APIs to set correspondingly.

3.3.2.1 Register L2CAP Data Processing Function

In the BLE SDK architecture, the Controller's data is interfaced with the Host via the HCI, and the data from the HCI to the Host is first processed at the L2CAP layer, using the following API to register this processing function.

```
void    blc_l2cap_register_handler (void *p);
```

In BLE Slave applications such as b85m_ble_remote/b85m_module, the functions in the SDK L2CAP layer that process Controller data are:

```
int     blc_l2cap_packet_receive (u16 connHandle, u8 * p);
```

This function has been implemented in the protocol stack and it will parse the received data and transmit it upwards to ATT, SIG or SMP.

Initialization:

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

In the b85m_master kma dongle, the application layer contains the BLE Host function with the following processing functions, the source code of which is provided for user reference.

```
int app_l2cap_handler (u16 conn_handle, u8 *raw_pkt);
```

Initialization:

```
blc_l2cap_register_handler (app_l2cap_handler);
```

In the b85m hci, only the slave controller is implemented. The blc_hci_sendACLData2Host function transmits the controller data to the BLE Host device via a hardware interface such as UART/USB.

```
int blc_hci_sendACLData2Host (u16 handle, u8 *p)
```

Initialization:

```
blc_l2cap_register_handler (blc_hci_sendACLData2Host);
```

3.3.2.2 Update connection parameters

(1) Slave requests for connection parameter update

In BLE stack, Slave can actively apply for a new set of connection parameters by sending a "CONNECTION PARAMETER UPDATE REQUEST" command to Master in L2CAP layer. The figure below shows the command format. Please refer to "Core_v5.0" (Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE REQUEST").

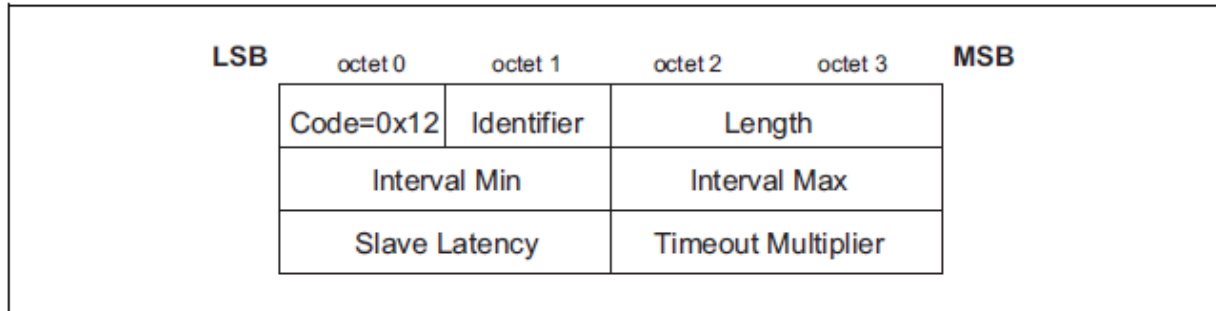


Figure 4.22: Connection Parameters Update Request Packet

Figure 3.39: "Connection Para Update Req Format in BLE Stack"

The BLE SDK provides an API for slaves to actively apply to update connection parameters on the L2CAP layer to send the above CONNECTION PARAMETER UPDATE REQUEST command to the master.

```
void bls_l2cap_requestConnParamUpdate (u16 min_interval, u16 max_interval, u16 latency, u16
↪ timeout);
```

The four parameters of this API correspond to the parameters in the "data" field of the "CONNECTION PARAMETER UPDATE REQUEST". The "min_interval" and "max_interval" are the actual interval time divided by 1.25ms (e.g. for 7.5ms connection interval, the value should be 6); the "timeout" is actual supervision timeout divided by 10ms (e.g. for 1s timeout, the value should be 100).

Application example: Slave requests for new connection parameters when connection is established.

```
void task_connect (u8 e, u8 *p, int n)
{
    bls_l2cap_requestConnParamUpdate (6, 6, 99, 400);
    bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(1000);
}
```

Data Type	Data Header					L2CAP Header		SIG Pkt Header			SIG_Connection_Param_Update_Req				CRC
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Code	Id	Data-Length	IntervalMin	IntervalMax	SlaveLatency	TimeoutMultiplier	0x28D8
	2	1	0	0	16	0x000C	0x0005	0x12	0x01	0x0008	0x0006	0x0006	0x0063	0x0190	

Data Type	Data Header					L2CAP Header		SIG Pkt Header			SIG_Connection_Param_Update_Rsp				CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Code	Id	Data-Length	Result				0x2DE483	-38	OK
	2	1	1	0	10	0x0006	0x0005	0x13	0x01	0x0002	0x0000						

Figure 3.40: "BLE Sniffer Packet Sample Conn Para Update Request and Response"

The API:

```
void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(int time_ms)
```

serves to make the Slave wait for time_ms milliseconds after connection is established, and then invoke the API "bls_l2cap_requestConnParamUpdate" to update connection parameters. After connection is established, if user only invokes the "bls_l2cap_requestConnParamUpdate", the Slave will wait for 1s to execute this request command.

For Slave applications, the SDK provides register callback function interface of obtaining Conn_UpdateRsp result, so as to inform user whether connection parameter update request from Slave is rejected or accepted by Master. As shown in the figure above, Master accepts Connection_Param_Update_Req from Slave.

```
void blc_l2cap_registerConnUpdateRspCb(l2cap_conn_update_rsp_callback_t cb);
```

Please refer to the use case of Slave initialization:

```
blc_l2cap_registerConnUpdateRspCb(app_conn_param_update_response)
```

Following shows the reference for the callback function "app_conn_param_update_response".

```
int app_conn_param_update_response(u8 id, u16 result)
{
    if(result == CONN_PARAM_UPDATE_ACCEPT){
        //the LE master Host has accepted the connection parameters
    }
    else if(result == CONN_PARAM_UPDATE_REJECT){
        //the LE master Host has rejected the connection parameter
    }
    return 0;
}
```

(2) Master responds to connection parameter update request

After Master receives the "CONNECTION PARAMETER UPDATE REQUEST" command from Slave, it will respond with a "CONNECTION PARAMETER UPDATE RESPONSE" command. Please refer to "Core_v5.0" (Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE RESPONSE").

The figure below shows the command format: if "result" is "0x0000", it indicates the request command is accepted; if "result" is "0x0001", it indicates the request command is rejected.

Whether actual Android/iOS device will accept or reject the connection parameter update request is determined by corresponding BLE Master. User can refer to Master compatibility test.

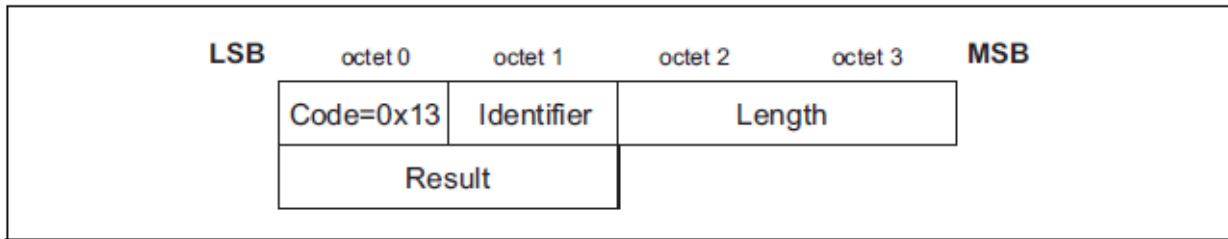


Figure 4.23: Connection Parameters Update Response Packet

The data field is:

- *Result (2 octets)*

The result field indicates the response to the Connection Parameter Update Request. The result value of 0x0000 indicates that the LE master Host has accepted the connection parameters while 0x0001 indicates that the LE master Host has rejected the connection parameters.

Result	Description
0x0000	Connection Parameters accepted
0x0001	Connection Parameters rejected
Other	Reserved

Figure 3.41: "Conn Para Update RSP Format in BLE Stack"

Telink's b85m_master kma dongle handles the connection parameter update demo code for slave as follows.

```

else if(ptrL2cap->chanId == L2CAP_CID_SIG_CHANNEL) //signal
{
    if(ptrL2cap->opcode == L2CAP_CMD_CONN_UPD_PARAM_REQ) //slave send conn param update req on l2cap
    {
        rf_packet_l2cap_connParaUpReq_t * req = (rf_packet_l2cap_connParaUpReq_t *)ptrL2cap;

        u32 interval_us = req->min_interval*1250; //1.25ms unit
        u32 timeout_us = req->timeout*10000; //10ms unit
        u32 long_suspend_us = interval_us * (req->latency+1);

        //interval < 200ms
        //long suspend < 11s
        // interval * (latency +1)*2 <= timeout
        if( interval_us < 200000 && long_suspend_us < 20000000 && (long_suspend_us*2<=timeout_us) )
        {
            //when master host accept slave's conn param update req, should send a conn param update response on l2cap
            //with CONN_PARAM_UPDATE_ACCEPT; if not accpet,should send CONN_PARAM_UPDATE_REJECT
            blc_l2cap_SendConnParamUpdateResponse(conn_handle, req->id, CONN_PARAM_UPDATE_ACCEPT); //send SIG Connection Param Update Response

            //if accept, master host should mark this, add will send update conn param req on link layer later
            //set a flag here, then send update conn param req in mainloop
            host_update_conn_param_req = clock_time() | 1; //in case zero value
            host_update_conn_min = req->min_interval; //backup update param
            host_update_conn_latency = req->latency;
            host_update_conn_timeout = req->timeout;
        }
        else
        {
            blc_l2cap_SendConnParamUpdateResponse(conn_handle, req->id, CONN_PARAM_UPDATE_REJECT); //send SIG Connection Param Update Response
        }
    }
}

```

Figure 3.42: “Demo code of b85m master kma dongle”

After “L2CAP_CMD_CONN_UPD_PARAM_REQ” is received in “L2CAP_CID_SIG_CHANNEL”, it will read interval_min (used as eventual interval), supervision timeout and long suspend time (interval * (latency +1)), and check the rationality of these data. If interval < 200ms, long suspend time<20s and supervision timeout >= 2* long suspend time, this request will be accepted; otherwise this request will be rejected. User can modify as needed based on this simple demo design.

No matter whether parameter request of Slave is accepted, the API below can be invoked to respond to this request.

```

void blc_l2cap_SendConnParamUpdateResponse( u16 connHandle, u8 req_id, conn_para_up_rsp
↪ result);

```

“connHandle” indicates current connection ID. “result” has two options to indicate “accept” and “reject”, respectively.

```

typedef enum{
    CONN_PARAM_UPDATE_ACCEPT = 0x0000,
    CONN_PARAM_UPDATE_REJECT = 0x0001,
}conn_para_up_rsp;

```

If the b85m_master kma dongle accepts the Slave’s request, it must send an update cmd to the Controller via API blm_ll_updateConnection within a certain amount of time, using host_update_conn_param_req on the demo code as a flag and initiates this update after a 50ms delay in the main_loop.

```
//at least 50ms later and make sure smp/sdp is finished
if( host_update_conn_param_req && clock_time_exceed(host_update_conn_param_req, 50000) && !app_host_smp_sdp_pending)
{
    host_update_conn_param_req = 0;

    if(blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){ //still in connection state
        blm_ll_updateConnection (cur_conn_device.conn_handle,
                                host_update_conn_min, host_update_conn_min, host_update_conn_latency, host_update_conn_timeout,
                                0, 0 );
    }
}
```

Figure 3.43: “Demo code of b85m master kma dongle”

(3) Master updates connection parameters in Link Layer

After Master responds with “conn para update rsp” to accept the “conn para update req” from Slave, Master will send a “LL_CONNECTION_UPDATE_REQ” command in Link Layer.

is	Data Header						LL_Opcode		LL_Connect_Update_Req					
	Data Type	LLID	NESN	SN	MD	PDU-Length			WinSize	WinOffset	Interval	Latency	Timeout	Instant
	Control	3	1	1	0	12	Connection_Update_Req(0x00)		0x02	0x001F	0x0006	0x0063	0x0190	0x006C

is	Data Header						CRC	RSSI (dBm)	FCS
	Data Type	LLID	NESN	SN	MD	PDU-Length			
	Empty PDU	1	0	1	0	0	0x8FE90F	0	OK

Figure 3.44: “BLE Sniffer Packet Sample II Conn Update Req”

Slave will mark the final parameter as the instant value of Master after it receives the update request. When the instant value of Slave reaches this value, connection parameters are updated, and the callback of the event “BLT_EV_FLAG_CONN_PARA_UPDATE” is triggered.

The “instant” indicates connection event count value maintained by Master and Slave, and it ranges from 0x0000 to 0xffff. During a connection, Master and Slave should always have consistent “instant” value. When Master sends “conn_req” and establishes connection with Slave, Master switches state from scanning to connection, and clears the “instant” of Master to “0”. When Slave receives the “conn_req”, it switches state from advertising to connection, and clears the instant of Slave to “0”. Each connection packet of Master and Slave is a connection event. For the first connection event after the “conn_req”, the instant value is “1”; for the second connection event, the instant value is 2, and so on.

When Master sends a “LL_CONNECTION_UPDATE_REQ”, the final parameter “instant” indicates during the connection event marked with “instant”, Master will use the values corresponding to the former connection parameters of the “LL_CONNECTION_UPDATE_REQ” packet. After the “LL_CONNECTION_UPDATE_REQ” is received, the new connection parameters will be used during the connection event when the instant of Slave equals the declared instant of Master, thus Slave and Master can finish switch of connection parameters synchronously.

3.3.3 ATT & GATT

3.3.3.1 GATT basic unit “Attribute”

GATT defines two roles: Server and Client. In this BLE SDK, Slave is Server, and corresponding Android/iOS device is Client. Server needs to supply multiple Services for Client to access.

Each Service of GATT consists of multiple Attributes, and each Attribute contains certain information. When multiple Attributes of different kinds are combined together, they can reflect a basic service.

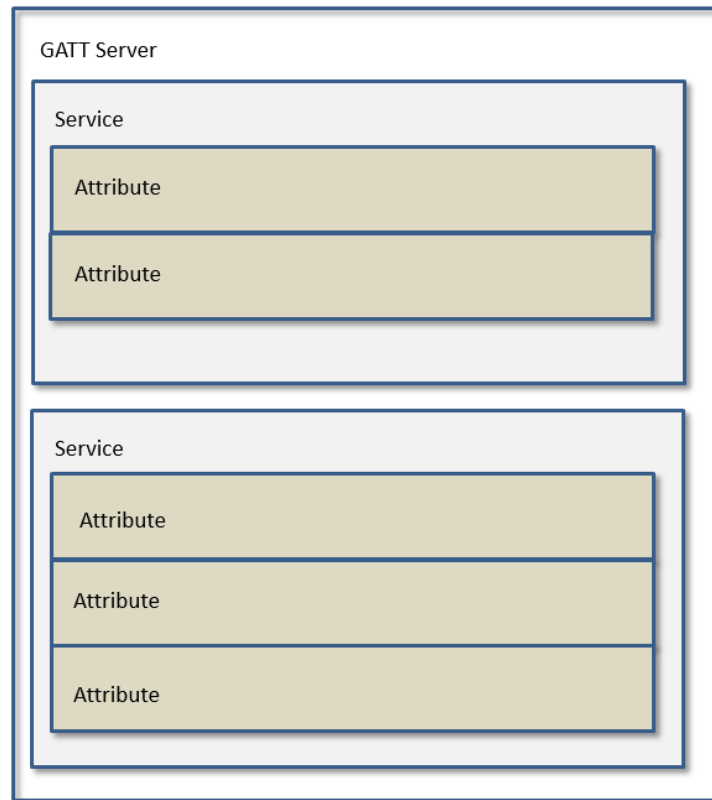


Figure 3.45: "GATT Service Containing Attributes"

The basic contents of Attribute are shown as below:

(1) Attribute Type: UUID

The UUID is used to identify Attribute type, and its total length is 16 bytes. In BLE standard protocol, the UUID length is defined as two bytes, since Master devices follow the same method to transform 2-byte UUID into 16 bytes.

When standard 2-byte UUID is directly used, Master should know device types indicated by various UUIDs. 8x5x BLE stack defines some standard UUIDs in "stack/service/hids.h" and "stack/ble/uuid.h".

Telink proprietary profiles (OTA, MIC, SPEAKER, and etc.) are not supported in standard Bluetooth. The 16-byte proprietary device UUIDs are defined in "stack/ble/uuid.h".

(2) Attribute Handle

Slave supports multiple Attributes which compose an Attribute Table. In Attribute Table, each Attribute is identified by an Attribute Handle value. After connection is established, Master will analyze and obtain the Attribute Table of Slave via "Service Discovery" process, then it can identify Attribute data via the Attribute Handle during data transfer.

(3) Attribute Value

Attribute Value corresponding to each Attribute is used as request, response, notification and indication data. In 8x5x BLE stack, Attribute Value is indicated by one pointer and the length of the area pointed by the pointer.

3.3.3.2 Attribute and ATT Table

To implement GATT Service on Slave, an Attribute Table is defined in this BLE SDK and it consists of multiple basic Attributes. Attribute definition is shown as below.

```
typedef struct attribute
{
    u16 attNum;
    u8  perm;
    u8  uuidLen;
    u32 attrLen;    //4 bytes aligned
    u8* uuid;
    u8* pAttrValue;
    att_readwrite_callback_t w;
    att_readwrite_callback_t r;
} attribute_t;
```

Below is Attribute Table given by the BLE SDK to illustrate the meaning of the above items. See app_att.c for the Attribute Table code, as shown below:

```
static const attribute_t my_Attributes[] = {
    {ATT_END_H - 1, 0,0,0,0,0}, // total num of attribute
    // 0001 - 0007 gap
    {7,ATT_PERMISSIONS_READ,2,2,(u8*)&my_primaryServiceUUID, (u8*)&my_gapServiceUUID, 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devNameCharVal),(u8*)&my_characterUUID, (u8*)
↪ (my_devNameCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)&my_devNameUUID, (u8*)(my_devName), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_appearanceCharVal),(u8*)&my_characterUUID, (u8*)
↪ (my_appearanceCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_appearance), (u8*)&my_appearanceUUID, (u8*)
↪ (&my_appearance), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_periConnParamCharVal),(u8*)&my_characterUUID, (u8*)
↪ (my_periConnParamCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_periConnParameters),(u8*)&my_periConnParamUUID,
↪ (u8*)&my_periConnParameters), 0},
    // 0008 - 000b gatt
    {4,ATT_PERMISSIONS_READ,2,2,(u8*)&my_primaryServiceUUID, (u8*)&my_gattServiceUUID, 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeCharVal),(u8*)&my_characterUUID,
↪ (u8*)(my_serviceChangeCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (serviceChangeVal), (u8*)&serviceChangeUUID, (u8*)
↪ (&serviceChangeVal), 0},
    {0,ATT_PERMISSIONS_RDWR,2,sizeof (serviceChangeCCC),(u8*)&clientCharacterCfgUUID, (u8*)
↪ (serviceChangeCCC), 0},
};
```

Note: The key word "const" is added before Attribute Table definition:

```
const attribute_t my_Attributes[] = { ... };
```

By adding the “const”, the compiler will store the array data to flash rather than RAM, while all contents of the Attribute Table defined in flash are read only and not modifiable.

(1) attNum

The “attNum” supports two functions.

The “attNum” can be used to indicate the number of valid Attributes in current Attribute Table, i.e. the maximum Attribute Handle value. This number is only used in the invalid Attribute item 0 of Attribute Table array.

```
{57,0,0,0,0,0}, // ATT_END_H - 1 = 57
```

“attNum = 57” indicates there are 57 valid Attributes in current Attribute Table.

In BLE, Attribute Handle value starts from 0x0001 with increment step of 1, while the array index starts from 0. When this virtual Attribute is added to the Attribute Table, each Attribute index equals its Attribute Handle value. After the Attribute Table is defined, Attribute Handle value of an Attribute can be obtained by counting its index in current Attribute Table array.

The final index is the number of valid Attributes (i.e. attNum) in current Attribute Table. In current SDK, the attNum is set as 57; if user adds or deletes any Attribute, the attNum needs to be modified correspondingly.

The “attNum” can also be used to specify Attributes constituting current Service.

The UUID of the first Attribute for each Service must be “GATT_UUID_PRIMARY_SERVICE(0x2800)”; the first Attribute of a Service sets “attNum” and it indicates following “attNum” Attributes constitute current Service.

As shown in code above, for the gap service, the Attribute with UUID of “GATT_UUID_PRIMARY_SERVICE” sets the “attNum” as 7, it indicates the seven Attributes from Attribute Handle 1 to Attribute Handle 7 constitute the gap service.

Except for Attribute item 0 and the first Attribute of each Service, attNum values of all Attributes must be set as 0.

(2) perm

The “perm” is the simplified form of “permission” and it serves to specify access permission of current Attribute by Client.

The “perm” of each Attribute should be configured as one or combination of following 10 values.

```
#define ATT_PERMISSIONS_READ          0x01
#define ATT_PERMISSIONS_WRITE         0x02
#define ATT_PERMISSIONS_AUTHEN_READ   0x61
#define ATT_PERMISSIONS_AUTHEN_WRITE  0x62
#define ATT_PERMISSIONS_SECURE_CONN_READ 0xE1
#define ATT_PERMISSIONS_SECURE_CONN_WRITE 0xE2
```

```
#define ATT_PERMISSIONS_AUTHOR_READ      0x11
#define ATT_PERMISSIONS_AUTHOR_WRITE    0x12
#define ATT_PERMISSIONS_ENCRYPT_READ     0x21
#define ATT_PERMISSIONS_ENCRYPT_WRITE    0x22
```

Note: Current SDK version does not support PERMISSION READ and PERMISSION WRITE yet.

(3) uuid and uuidLen

As introduced above, UUID supports two types: BLE standard 2-byte UUID, and Telink proprietary 16-byte UUID. The "uuid" and "uuidLen" can be used to describe the two UUID types simultaneously.

The "uuid" is an u8-type pointer, and "uuidLen" specifies current UUID length, i.e. the uuidLen bytes starting from the pointer are current UUID. Since Attribute Table and all UUIDs are stored in flash, the "uuid" is a pointer pointing to flash.

a) BLE standard 2-byte UUID:

For example, the Attribute "devNameCharacter" with Attribute Handle of 2, related code is shown as below:

```
#define GATT_UUID_CHARACTER      0x2803
static const u16 my_characterUUID = GATT_UUID_CHARACTER;
static const u8 my_devNameCharVal[5] = {0x12, 0x03, 0x00, 0x00, 0x2A};
{0,1,2,5,(u8*)&my_characterUUID, (u8*)&my_devNameCharVal, 0},
```

"UUID=0x2803" indicates "character" in BLE and the "uuid" points to the address of "my_devNameCharVal" in flash. The "uuidLen" is 2. When Master reads this Attribute, the UUID would be "0x2803".

b) Telink proprietary 16-byte UUID:

For example, the Attribute MIC of audio, related code is shown as below:

```
#define TELINK_MIC_DATA {0x18,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,
↳ 0x01,0x0}
const u8 my_MicUUID[16] = TELINK_MIC_DATA;
static u8 my_MicData = 0x80;
{0,1,16,1,(u8*)&my_MicUUID, (u8*)&my_MicData, 0},
```

The "uuid" points to the address of "my_MicData" in flash, and the "uuidLen" is 16. When Master reads this Attribute, the UUID would be "0x000102030405060708090a0b0c0d2b18".

(4) pAttrValue and attrLen

Each Attribute corresponds to an Attribute Value. The "pAttrValue" is an u8-type pointer which points to starting address of Attribute Value in RAM/Flash, while the "attrLen" specifies the data length. When Master reads the Attribute Value of certain Attribute from Slave, the "attrLen" bytes of data starting from the area (RAM/Flash) pointed by the "pAttrValue" will be read by this BLE SDK to Master.

Since UUID is read only, the "uuid" is a pointer pointing to flash; while Attribute Value may involve write operation into RAM, so the "pAttrValue" may points to RAM or flash.

For example, the Attribute hid Information with Attribute Handle of 35, related code is as shown below:

```
const u8 hidInformation[] =
{
    U16_L0(0x0111), U16_HI(0x0111),    // bcdHID (USB HID version), 0x11,0x01
    0x00,                               // bCountryCode
    0x01                               // Flags
};
{0,1,2, sizeof(hidInformation),(u8*)&hidInformationUUID, (u8*)(hidInformation), 0},
```

In actual application, the key word “const” can be used to store the read-only 4-byte hid information “0x01 0x00 0x01 0x11” into flash. The “pAttrValue” points to the starting address of hidInformation in flash, while the “attrLen” is the actual length of hidInformation. When Master reads this Attribute, “0x01000111” will be returned to Master correspondingly.

Figure below shows a packet example captured by BLE sniffer when Master reads this Attribute. Master uses the “ATT_Read_Req” command to set the target AttHandle as 0x23 (35), corresponding to the hid information in Attribute Table of SDK.

us	Data Type	Data Header	Security Enabled	L2CAP Header	ATT_Read_Req	CRC	RSSI (dBm)	FCS
	L2CAP-S	LLID NESN SN MD PDU-Length	Yes	L2CAP-Length ChanId	Opcode AttHandle	0x65CCC5	0	OK
		2 1 0 0 11		0x0003 0x0004	0x0A 0x0023			
us	Data Type	Data Header	Security Enabled	CRC	RSSI (dBm)	FCS		
	Empty PDU	LLID NESN SN MD PDU-Length	Yes	0x2A576A	0	OK		
		1 1 1 0 0						
us	Data Type	Data Header	Security Enabled	CRC	RSSI (dBm)	FCS		
	Empty PDU	LLID NESN SN MD PDU-Length	Yes	0x2A51B9	0	OK		
		1 0 1 0 0						
us	Data Type	Data Header	Security Enabled	L2CAP Header	ATT_Read_Rsp	CRC	RSSI (dBm)	FCS
	L2CAP-S	LLID NESN SN MD PDU-Length	Yes	L2CAP-Length ChanId	Opcode AttValue	0x9BF6A0	0	OK
		2 0 0 0 13		0x0005 0x0004	0x0B 11 01 00 01			

Figure 3.46: “BLE Sniffer Packet Sample when Master Reads hidInformation”

For the Attribute “battery value” with Attribute Handle of 40, related code is as shown below:

```
u8 my_batVal[1] = {99};
{0,1,2,1,(u8*)&my_batCharUUID, (u8*)(my_batVal), 0},
```

In actual application, the “my_batVal” indicates current battery level and it will be updated according to ADC sampling result; then Slave will actively notify or Master will actively read to transfer the “my_batVal” to Master. The starting address of the “my_batVal” stored in RAM will be pointed by the “pAttrValue”.

(5) Callback function w

The callback function w is write function with prototype as below:

```
typedef int (*att_readwrite_callback_t)(u16 connHandle, void* p);
```

User must follow the format above to define callback write function. The callback function w is optional, i.e. for an Attribute, user can select whether to set the callback write function as needed (null pointer 0 indicates not setting callback write function).

The trigger condition for callback function *w* is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function *w* is set.

- a) opcode = 0x12, Write Request.
- b) opcode = 0x52, Write Command.
- c) opcode = 0x18, Execute Write Request.

After Slave receives a write command above, if the callback function *w* is not set, Slave will automatically write the area pointed by the "pAttrValue" with the value sent from Master, and the data length equals the "l2capLen" in Master packet format minus 3; if the callback function *w* is set, Slave will execute user-defined callback function *w* after it receives the write command, rather than writing data into the area pointed by the "pAttrValue". Note: Only one of the two write operations is allowed to take effect.

By setting the callback function *w*, user can process Write Request, Write Command, and Execute Write Request in ATT layer of Master. If the callback function *w* is not set, user needs to evaluate whether the area pointed by the "pAttrValue" can process the command (e.g. If the "pAttrValue" points to flash, write operation is not allowed; or if the "attrLen" is not long enough for Master write operation, some data will be modified unexpectedly.)

3.4.5.1 Write Request

The *Write Request* is used to request the server to write the value of an attribute and acknowledge that this has been achieved in a *Write Response*.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x12 = Write Request
Attribute Handle	2	The handle of the attribute to be written
Attribute Value	0 to (ATT_MTU-3)	The value to be written to the attribute

Figure 3.47: "Write Request in BLE Stack"

3.4.5.3 Write Command

The *Write Command* is used to request the server to write the value of an attribute, typically into a control-point attribute.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x52 = Write Command
Attribute Handle	2	The handle of the attribute to be set
Attribute Value	0 to (ATT_MTU-3)	The value of be written to the attribute

Figure 3.48: "Write Command in BLE Stack"

3.4.6.3 Execute Write Request

The *Execute Write Request* is used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client. This request shall be handled by the server as an atomic operation.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x18 = Execute Write Request

Figure 3.49: "Execute Write Request in BLE Stack"

The void-type pointer "p" of the callback function w points to the value of Master write command. Actually "p" points to a memory area, the value of which is shown as the following structure.

```
typedef struct{
    u8  type;
    u8  rf_len;
    u16 l2cap;
    u16 chanid;
    u8  att;
    u8  hl;
    u8  hh;
    u8  dat[20];
}rf_packet_att_data_t;
```

"p" points to "type", valid length of data is l2cap minus 3, and the first valid data is pw->dat[0].

```
int my_WriteCallback (void *p)
{
    rf_packet_att_data_t *pw = (rf_packet_att_data_t *)p;
    int len = pw->l2cap - 3;
    //add your code
    //valid data is pw->dat[0] ~ pw->dat[len-1]
    return 1;
}
```

The structure "rf_packet_att_data_t" above is available in the "stack/ble/ble_format.h".

(6) Callback function r

The callback function r is read function with prototype as below:

```
typedef int (*att_readwrite_callback_t)(u16 connHandle, void* p);
```

User must follow the format above to define callback read function. The callback function r is also optional, i.e. for an Attribute, user can select whether to set the callback read function as needed (null pointer 0 indicates not setting callback read function), connHandle is connecting sentence between master and slave, type BLS_CONN_HANDLE for slave application, and type BLM_CONN_HANDLE for master application.

The trigger condition for callback function r is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function r is set.

- a) opcode = 0x0A, Read Request.
- b) opcode = 0x0C, Read Blob Request.

After Slave receives a read command above,

- a) If the callback read function is set, Slave will execute this function, and determine whether to respond with "Read Response/Read Blob Response" according to the return value of this function.
 - If the return value is 1, Slave won't respond with "Read Response/Read Blob Response" to Master.
 - If the return value is not 1, Slave will automatically read "attrLen" bytes of data from the area pointed by the "pAttrValue", and the data will be responded to Master via "Read Response/Read Blob Response".
- b) If the callback read function is not set, Slave will automatically read "attrLen" bytes of data from the area pointed by the "pAttrValue", and the data will be responded to Master via "Read Response/Read Blob Response".

Therefore, after a Read Request/Read Blob Request is received from Master, if it's needed to modify the content of Read Response/Read Blob Response, user can register corresponding callback function r, modify contents in RAM pointed by the "pAttrValue" in this callback function, and the return value must be 0.

(7) Attribute Table layout

Figure below shows Service/Attribute layout based on Attribute Table. The "attnum" of the first Attribute indicates the number of valid Attributes in current ATT Table; the remaining Attributes are assigned to different Services, the first Attribute of each Service is the "declaration", and the following "attnum" Attributes constitute current Service. Actually the first item of each Service is a Primary Service.

```
#define GATT_UUID_PRIMARY_SERVICE          0x2800
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```

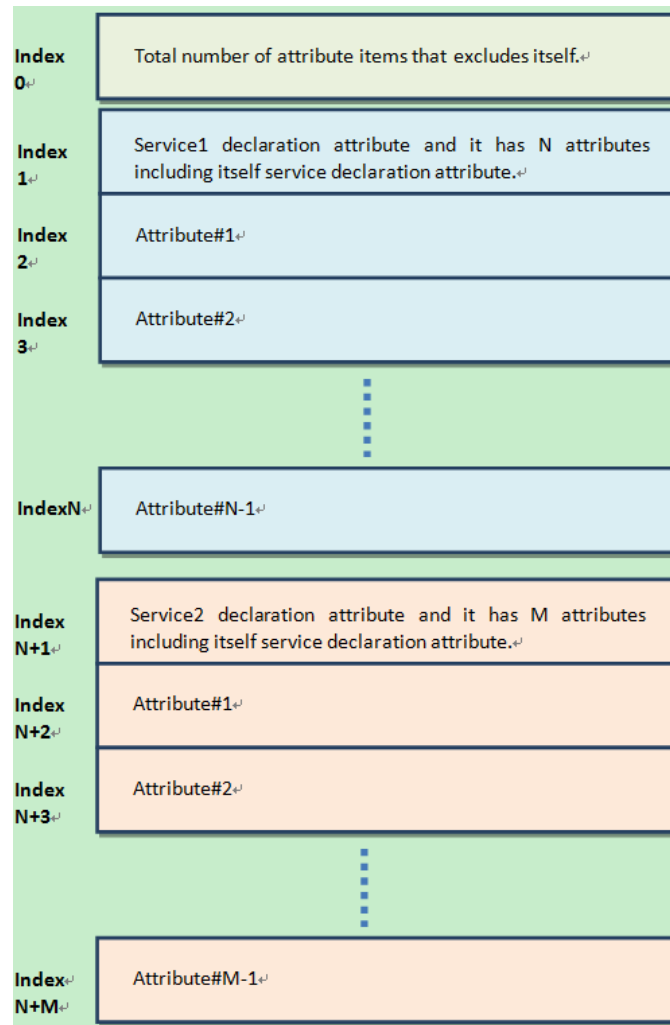


Figure 3.50: “Service Attribute Layout”

(8) ATT table Initialization

GATT & ATT initialization only needs to transfer the pointer of Attribute Table in APP layer to protocol stack, and the API below is supplied:

```
void      bls_att_setAttributeTable (u8 *p);
```

“p” is the pointer of Attribute Table.

3.3.3.3 Attribute PDU and GATT API

As required by BLE spec, the following Attribute PDU types are supported in current SDK.

- Requests: Data request sent from Client to Server.
- Responses: Data response sent by Server after it receives request from Client.
- Commands: Command sent from Client to Server.
- Notifications: Data sent from Server to Client.
- Indications: Data sent from Server to Client.
- Confirmations: Confirmation sent from Client after it receives data from Server.

The following is an analysis of all ATT PDUs at the ATT layer in conjunction with the Attribute structure and Attribute Table structure described previously.

(1) Read by Group Type Request, Read by Group Type Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.9 and 3.4.4.10) for details about the "Read by Group Type Request" and "Read by Group Type Response" commands.

The "Read by Group Type Request" command sent by Master specifies starting and ending attHandle, as well as attGroupType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute Group that matches the specified attGroupType. Then Slave will respond to Master with Attribute Group information via the "Read by Group Type Response" command.

Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	0x89867B	-38	OK	
L2CAP-S	2	0	1	0	11	0x0007	0x0004	0x10	0x0001	0xFFFF	00 28				
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK							
Empty PDU	1	0	0	0	0										
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Rsp				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		0x58FC67	-38	OK	
L2CAP-S	2	0	0	0	24	0x0014	0x0004	0x11	0x06	01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18					
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	0x5A6275	-38	OK	
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x10	0x0026	0xFFFF	00 28				
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0BA0	-38	OK							
Empty PDU	1	1	1	0	0										
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0D73	-38	OK							
Empty PDU	1	0	1	0	0										
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Rsp				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		0x158866	-38	OK	
L2CAP-S	2	0	0	0	12	0x0008	0x0004	0x11	0x06	26 00 28 00 0F 18					
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	0x055C4D	-38	OK	
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x10	0x0029	0xFFFF	00 28				
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0BA0	-38	OK							
Empty PDU	1	1	1	0	0										
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0D73	-38	OK							
Empty PDU	1	0	1	0	0										
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Rsp				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		0x898D99	-38	OK	
L2CAP-S	2	0	0	0	26	0x0016	0x0004	0x11	0x14	29 00 32 00 11 19 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00					
Data Type	Data Header					L2CAP Header		ATT_Read_By_Group_Type_Req				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	0x3C57D1	-38	OK	
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x10	0x0033	0xFFFF	00 28				
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0BA0	-38	OK							
Empty PDU	1	1	1	0	0										
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0D73	-38	OK							
Empty PDU	1	0	1	0	0										
Data Type	Data Header					L2CAP Header		ATT_Error_Response				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ReqOpCode	AttHandle	ErrorCode	0x600FAA	-38	OK	
L2CAP-S	2	0	0	0	9	0x0005	0x0004	0x01	0x10	0x0033	ATT_NOT_FOUND(0x02)				

Figure 3.51: "Read by Group Type Request Read by Group Type Response"

As shown above, Master requests from Slave for Attribute Group information of the “primaryServiceUUID” with UUID of 0x2800.

```
#define GATT_UUID_PRIMARY_SERVICE      0x2800
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```

The following groups in Slave Attribute Table meet the requirement according to current demo code.

a) Attribute Group with attHandle from 0x0001 to 0x0007,

Attribute Value is SERVICE_UUID_GENERIC_ACCESS (0x1800).

b) Attribute Group with attHandle from 0x0008 to 0x000a,

Attribute Value is SERVICE_UUID_DEVICE_INFORMATION (0x180A).

c) Attribute Group with attHandle from 0x000B to 0x0025,

Attribute Value is SERVICE_UUID_HUMAN_INTERFACE_DEVICE (0x1812).

d) Attribute Group with attHandle from 0x0026 to 0x0028,

Attribute Value is SERVICE_UUID_BATTERY (0x180F).

e) Attribute Group with attHandle from 0x0029 to 0x0032,

Attribute Value is TELINK_AUDIO_UUID_SERVICE(0x11,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00).

Slave responds to Master with the attHandle and attValue information of the five Groups above via the “Read by Group Type Response” command. The final ATT_Error_Response indicates end of response. When Master receives this packet, it will stop sending “Read by Group Type Request”.

(2) Find by Type Value Request, Find by Type Value Response

Please refer to “Core_v5.0” (Vol 3/Part F/3.4.3.3 and 3.4.3.4) for details about the “Find by Type Value Request” and “Find by Type Value Response” commands.

The “Find by Type Value Request” command sent by Master specifies starting and ending attHandle, as well as AttributeType and Attribute Value. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType and Attribute Value. Then Slave will respond to Master with the Attribute via the “Find by Type Value Response” command.

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Req				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	AttValue	0x4CEA12	-54	OK
	2	1	1	0	13	0x0009	0x0004	0x06	0x0001	0xFFFF	0x2800	0A 18			

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xC4C0E8	-54	OK
	1	0	0	0	0			

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Rsp				CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	HandleInfo		0xF92ED9	-54	OK	
	2	1	0	0	9	0x0005	0x0004	0x07	0C 00 0E 00					

Figure 3.52: “Find by Type Value Request Find by Type Value Response”

(3) Read by Type Request, Read by Type Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.1 and 3.4.4.2) for details about the "Read by Type Request" and "Read by Type Response" commands.

The "Read by Type Request" command sent by Master specifies starting and ending attHandle, as well as AttributeType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType. Then Slave will respond to Master with the Attribute via the "Read by Type Response".

Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Req				
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	
L2CAP-S	2	0	0	1	11	0x0007	0x0004	0x08	0x0001	0xFFFF	00 2A	0x
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
	LLID	NESN	SN	MD	PDU-Length							
Empty PDU	1	1	0	0	0	0x898717	0	OK				
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
	LLID	NESN	SN	MD	PDU-Length							
Empty PDU	1	1	1	0	0	0x898AB1	0	OK				
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
	LLID	NESN	SN	MD	PDU-Length							
Empty PDU	1	0	1	0	0	0x898C62	0	OK				
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
	LLID	NESN	SN	MD	PDU-Length							
Empty PDU	1	0	0	0	0	0x8981C4	0	OK				
Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Rsp				CRC
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		
L2CAP-S	2	1	0	0	14	0x000A	0x0004	0x09	0x08	03 00 74 53 65 6C 66 69		0xDB602

Figure 3.53: "Read by Type Value Request Find by Type Value Response"

As shown above, Master reads the Attribute with attType of 0x2A00, i.e. the Attribute with Attribute Handle of 00 03 in Slave.

```
const u8    my_devName [] = {'t', 'S', 'e', 'l', 'l', 'f', 'i'};
#define GATT_UUID_DEVICE_NAME          0x2a00
const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;
{0,1,2, sizeof (my_devName),(u8*)&my_devNameUUID,(u8*)(my_devName), 0},
```

In the "Read by Type response", attData length is 8, the first two bytes are current attHandle "0003", followed by 6-byte Attribute Value.

(4) Find information Request, Find information Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.3.1 and 3.4.3.2) for details about the "Find information request" and "Find information response" commands.

The master sends a "Find information request", specifying the starting and ending attHandle. After receiving the command, the slave replies to the master through "Find information response" the UUIDs of all the starting and ending attHandle corresponding Attributes. As shown in the figure below, the master requires information of three Attributes with attHandle of 0x0016~0x0018, and Slave responds with corresponding UUIDs.

Data Type	Data Header					L2CAP Header		ATT_Find_Info_Req			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	0x362A2F	-38	OK
	2	0	1	0	9	0x0005	0x0004	0x04	0x0016	0x0018			
Data Type	Data Header					CRC	RSSI (dBm)	FCS					
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK					
	1	0	0	0	0								
Data Type	Data Header					CRC	RSSI (dBm)	FCS					
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK					
	1	1	0	0	0								
Data Type	Data Header					L2CAP Header		ATT_Find_Info_Rsp			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Format	InfoData			
	2	1	1	0	18	0x000E	0x0004	0x05	0x01	16 00 02 29 17 00 08 29 18 00 03 28			

Figure 3.54: "Find Information Request Find Information Response"

(5) Read Request, Read Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.3 and 3.4.4.4) for details about the "Read Request" and "Read Response" commands.

The "Read Request" command sent by Master specifies certain attHandle. After the request is received, Slave will respond to Master with the Attribute Value of the specified Attribute via the "Read Response" command (If the callback function r is set, this function will be executed), as shown below.

Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0x99C5FD	-38	OK
	2	0	1	0	7	0x0003	0x0004	0x0A	0x0017			

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK
	1	0	0	0	0			

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK
	1	1	0	0	0			

Data Type	Data Header					L2CAP Header		ATT_Read_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue	0x9082A7	-38	OK
	2	1	1	0	7	0x0003	0x0004	0x0B	02 01			

Figure 3.55: "Read Request Read Response"

(6) Read Blob Request, Read Blob Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.5 and 3.4.4.6) for details about the "Read Blob Request" and "Read Blob Response" commands.

If some Slave Attribute corresponds to Attribute Value with length exceeding MTU_SIZE (It's set as 23 in current SDK), Master needs to read the Attribute Value via the "Read Blob Request" command, so that the Attribute Value can be sent in packets. This command specifies the attHandle and ValueOffset. After the request is received, Slave will find corresponding Attribute, and respond to Master with the Attribute Value via the "Read Blob Response" command according to the specified ValueOffset. (If the callback function r is set, this function will be executed.)

As shown below, when Master needs the HID report map of Slave (report map length largely exceeds 23), first Master sends "Read Request", then Slave responds to Master with part of the report map data via "Read response"; Master sends "Read Blob Request", and then Slave responds to Master with data via "Read Blob Response".

Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xF4DC27	-38	OK
	2	0	1	0	7	0x0003	0x0004	0x0A	0x0020			
Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xAE00D5	-38	OK
	1	0	0	0	0	0x0005	0x0004	0x0A	0x0020			
Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xAE0606	-38	OK
	1	1	0	0	0	0x0005	0x0004	0x0A	0x0020			
Data Type	Data Header					L2CAP Header		ATT_Read_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue	0xEE69DD	-38	OK
	2	1	1	0	27	0x0017	0x0004	0x0B	05 01 09 02 A1 01 85 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0x8F3E95	-38	OK
	2	0	1	0	9	0x0005	0x0004	0x0C	0x0020 0x0016			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xAE00D5	-38	OK
	1	0	0	0	0	0x0005	0x0004	0x0A	0x0020			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xAE0606	-38	OK
	1	1	0	0	0	0x0005	0x0004	0x0A	0x0020			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	PartAttValue	0x2DE6F2	-38	OK
	2	1	1	0	27	0x0017	0x0004	0x0D	75 01 95 03 81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0x557D8E	-38	OK
	2	0	1	0	9	0x0005	0x0004	0x0C	0x0020 0x002C			

Figure 3.56: "Read Blob Request Read Blob Response"

(7) Exchange MTU Request, Exchange MTU Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.2.1 and 3.4.2.2) for details about the "Exchange MTU Request" and "Exchange MTU Response" commands.

As shown below, Master and Slave obtain MTU size of each other via the "Exchange MTU Request" and "Exchange MTU Response" commands.

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ClientRxMTU	0xC70102	-38	OK
	2	0	1	0	7	0x0003	0x0004	0x02	0x009E			
Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ServerRxMTU	0x1D88E1	-38	OK
	2	0	0	0	7	0x0003	0x0004	0x03	0x0017			

Figure 3.57: "Exchange MTU Request Exchange MTU Response"

During data access process of Telink BLE Slave GATT layer, if there's data exceeding a RF packet length, which involves packet assembly and disassembly in GATT layer, Slave and Master need to exchange RX MTU size of each other in advance. Transfer of long packet data in GATT layer can be implemented via MTU size exchange.

- User can register callback of GAP event and enable the eventMask "GAP_EVT_MASK_ATT_EXCHANGE_MTU" to obtain EffectiveRxMTU.

EffectiveRxMTU=min(ClientRxMTU, ServerRxMTU)。

The "GAP event" section of this document will introduce GAP event in detail.

- Processing of long Rx packet data in B85 Slave GATT layer

B85 Slave ServerRxMTU is set as 23 by default. Actually maximum ServerRxMTU can reach 250, i.e. 250-byte packet data on Master can be correctly re-assembled on Slave. When it's needed to use packet re-assembly of Master in an application, the API below should be invoked to modify RX size of Slave first.

```
ble_sts_t blc_att_setRxMtuSize(u16 mtu_size);
```

The return value is shown as below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Add success
GATT_ERR_INVALID_PARAMETER	See the definition in the SDK	mtu_size exceeds the max value 250.

When Master GATT layer needs to send long packet data to Slave, Master will actively initiate "ATT_Exchange_MTU_req", and Slave will respond with "ATT_Exchange_MTU_rsp". "ServerRxMTU" is the configured value of the API "blc_att_setRxMtuSize". If user has registered GAP event and enabled the eventMask "GAP_EVT_MASK_ATT_EXCHANGE_MTU", "EffectiveRxMTU" and "ClientRxMTU" of Master can be obtained in the callback function of GAP event.

c) Processing of long Tx packet data in B85 Slave GATT layer

When B85 Slave needs to send long packet data in GATT layer, it should obtain Client RxMTU of Master first, and the eventual data length should not exceed ClientRxMTU.

First Slave should invoke the API "blc_att_setRxMtuSize" to set its ServerRxMTU. Suppose it's set as 158.

```
blc_att_setRxMtuSize (158) ;
```

Then the API below should be invoked to actively initiate an "ATT_Exchange_MTU_req".

```
ble_sts_t blc_att_requestMtuSizeExchange (u16 connHandle, u16 mtu_size);
```

"connHandle" is ID of Slave connection, i.e. "BLS_CONN_HANDLE", while "mtu_size" is ServerRxMTU.

```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 158);
```

After the "ATT_Exchange_MTU_req" is received, Master will respond with "ATT_Exchange_MTU_rsp". After receiving the response, the SDK will calculate EffectiveRxMTU. If user has registered GAP event and enabled the eventMask "GAP_EVT_MASK_ATT_EXCHANGE_MTU", "EffectiveRxMTU" and "ClientRxMTU" will be reported to user.

(8) Write Request, Write Response

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.5.1 and 3.4.5.2) for details about the "Write Request" and "Write Response" commands.

The “Write Request” command sent by Master specifies certain attHandle and attaches related data. After the request is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Finally Slave will respond to Master via “Write Response”.

As shown in below, by sending “Write Request”, Master writes Attribute Value of 0x0001 to the Slave Attribute with the attHandle of 0x0016. Then Slave will execute the write operation and respond to Master via “Write Response”.

Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0xDC8476	-38	OK
	2	0	1	0	9	0x0005	0x0004	0x12	0x0016	01 00			
Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0xAE00D5	-38	OK
	1	0	0	0	0								
Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0xAE0606	-38	OK
	1	1	0	0	0								
Data Type	Data Header					L2CAP Header		ATT_Write_Rsp			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode			0xFBDB12	-38	OK
	2	1	1	0	5	0x0001	0x0004	0x13					

Figure 3.58: “Write Request Write Response”

(9) Write Command

Please refer to “Core_v5.0” (Vol 3/Part F/3.4.5.3) for details about the “Write Command”.

The “Write Command” sent by Master specifies certain attHandle and attaches related data. After the command is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Slave won’t respond to Master with any information.

(10) Queued Writes

“Queued Writes” refers to ATT protocol including “Prepare Write Request/Response” and “Execute Write Request/Response”. Please refer to “Core_v5.0” (Vol 3/Part F/3.4.6/Queued Writes).

“Prepare Write Request” and “Execute Write Request” can implement the two functions below.

- Provide write function for long attribute value.
- Allow to write multiple values in an atomic operation that is executed separately.

Similar to “Read_Blob_Req/Rsp”, “Prepare Write Request” contains AttHandle, ValueOffset and ParAttValue. That means Client can prepare multiple attribute values or various parts of a long attribute value in the queue. Thus, before executing the prepared queue indeed, Client can confirm that all parts of some attribute can be written into Server.

Note: Current SDK version only supports the write function of long attribute value with the maximum length not exceeding 244 bytes. If the length is greater than 244 bytes, the following API interface needs to be called to make changes to the prepare write buffer and its length.

```
void blc_att_setPrepareWriteBuffer(u8 *p, u16 len)
```

The figure below shows the case when Master writes a long character string "I am not sure what a new song" (byte number is far more than 23, and use the default MTU) into certain characteristic of Slave. First Master sends a "Prepare Write Request" with offset of 0x0000, to write the data "I am not sure what" into Slave, and Slave responds to Master with a "Prepare Write Response". Then Master sends a "Prepare Write Request" with offset of 0x12, to write the data " a new song" into Slave, and Slave responds to Master with a "Prepare Write Response". After the write operation of the long attribute value is finished, Master sends an "Execute Write Request" to Slave. "Flags=1" indicates write result takes effect immediately. Then Slave responds with an "Execute Write Response" to complete the whole Prepare Write process.

As we can see, "Prepare Write Response" also contains AttHandle, ValueOffset and PartAttValue in the request, so as to ensure reliable data transfer. Client can compare field value of Response with that of Request, to ensure correct reception of the prepared data.

Data Type	Data Header					L2CAP Header		ATT_Prepare_Write_Rsp																	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue														
	2	0	1	0	27	0x0017	0x0004	0x17	0x0015	0x0000	49 20 61 6D 20 6E 6F 74 20 73 75 72 65 20 77 68 61 74														
Data Type	Data Header					L2CAP Header		ATT_Prepare_Write_Req										CRC	RSSI (dBm)	FCS					
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue											0x98D4A6	-54	OK	
	2	0	0	0	20	0x0010	0x0004	0x16	0x0015	0x0012	20 61 20 6E 65 77 20 73 6F 6E 67														
Data Type	Data Header					CRC	RSSI (dBm)	FCS																	
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x071388	-54	OK																	
	1	1	0	0	0																				
Data Type	Data Header					CRC	RSSI (dBm)	FCS																	
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x071E2E	-54	OK																	
	1	1	1	0	0																				
Data Type	Data Header					L2CAP Header		ATT_Prepare_Write_Rsp										CRC	RSSI (dBm)	FCS					
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue											0xFF79B4	-54	OK	
	2	0	1	0	20	0x0010	0x0004	0x17	0x0015	0x0012	20 61 20 6E 65 77 20 73 6F 6E 67														
Data Type	Data Header					L2CAP Header		ATT_Execute_Write_Req		CRC	RSSI (dBm)	FCS													
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Flags	0x24D166	-54	OK													
	2	0	0	0	6	0x0002	0x0004	0x18	0x01																
Data Type	Data Header					CRC	RSSI (dBm)	FCS																	
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x071388	-54	OK																	
	1	1	0	0	0																				
Data Type	Data Header					CRC	RSSI (dBm)	FCS																	
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x071E2E	-54	OK																	
	1	1	1	0	0																				
Data Type	Data Header					CRC	RSSI (dBm)	FCS																	
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x07155B	-54	OK																	
	1	0	0	0	0																				
Data Type	Data Header					L2CAP Header		ATT_Execute_Write_Rsp		CRC	RSSI (dBm)	FCS													
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode											0x430D57	-54	OK				
	2	0	1	0	5	0x0001	0x0004	0x19																	

Figure 3.59: "Example for Write Long Characteristic Values"

(11) Handle Value Notification

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.7.1).

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1B = Handle Value Notification
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.34: Format of Handle Value Notification

Figure 3.60: "Handle Value Notification in BLE Spec"

The figure above shows the format of "Handle Value Notification" in BLE Spec.

This BLE SDK supplies an API for Handle Value Notification of an Attribute. By invoking this API, user can push the notify data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

```
ble_sts_t blc_gatt_pushHandleValueNotify (u16 handle, u8 *p, int len);
```

The handle is the attHandle of the corresponding Attribute, p is the header pointer to the contiguous memory data to be sent, and len specifies the number of bytes of data to be sent. Since this API supports auto packet disassembly based on EffectiveMaxTxOctets, long indicate data to be sent can be disassembled into multiple BLE RF packets, large "len" is supported. (EffectiveMaxTxOctets indicates the maximum RF TX octets to be sent in the Link Layer. Its default value is 27, and DLE may modify it. Another API as a replacement will be introduced later.)

When Link Layer is in Conn state, generally data will be successfully pushed into bottom-layer software FIFO by invoking this API; however, some special cases may result in invoking failure, and the return value "ble_sts_t" will indicate the corresponding error reason.

When this API is invoked in APP layer, it's recommended to check whether the return value is "BLE_SUCCESS". If the return value is not "BLE_SUCCESS", a delay is needed to re-push the data.

The return value is shown as below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Add success
LL_ERR_CONNECTION_NOT_ESTABLISH	See the definition in the SDK	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	See the definition in the SDK	Data cannot be sent during pairing or encryption phase.
LL_ERR_TX_FIFO_NOT_ENOUGH	See the definition in the SDK	Since task with mass data is being executed, software Tx fifo is not enough.
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	See the definition in the SDK	Data cannot be sent during service discovery phase.

Note: Another alternative API has been added to the SDK (using min(EffectiveMaxTxOctets, EffectiveRxMTU) as the minimum unit for subpackaging and can be called by both master and slave, users are advised to use the new API).

```
ble_sts_t blc_gatt_pushHandleValueNotify (u16 connHandle, u16 attHandle, u8 *p, int len);
```

When calling this API, it is recommended that users check whether the return value is BLE_SUCCESS, and whether it differs from the blc_gatt_pushHandleValueNotify return value:

1) When in the pairing phase, the new API returns the value: SMP_ERR_PAIRING_BUSY;

- 2) When in the encryption phase, the new API returns the value: LL_ERR_ENCRYPTION_BUSY;
- 3) When len is greater than ATT_MTU-3 (3 is the ATT layer packet format length opcode and handle), it means that the data length PDU to be sent exceeds the maximum PDU length ATT_MTU supported by the ATT layer, and the return value is GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE.

(12) Handle Value Indication

Please refer to "Core_v5.0" (Vol 3/Part F/3.4.7.2).

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1D = Handle Value Indication
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.35: Format of Handle Value Indication

Figure 3.61: "Handle Value Indication in BLE Spec"

The figure above shows the format of "Handle Value Indication" in BLE Spec.

This BLE SDK supplies an API for Handle Value Indication of an Attribute. By invoking this API, user can push the indicate data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

```
ble_sts_t bls_att_pushIndicateData (u16 handle, u8 *p, int len);
```

The handle is the attHandle of the corresponding Attribute, p is the header pointer to the contiguous memory data to be sent, and len specifies the number of bytes of data to be sent. Since this API supports auto packet disassembly based on EffectiveMaxTxOctets, long indicate data to be sent can be disassembled into multiple BLE RF packets, large "len" is supported. (EffectiveMaxTxOctets indicates the maximum RF TX octets to be sent in the Link Layer. Its default value is 27, and DLE may modify it. Another API as a replacement will be introduced later.)

The BLE Spec states that each indicate cannot be considered successful until the Master confirms it, and the next indicate cannot be sent without success.

When Link Layer is in Conn state, generally data will be successfully pushed into bottom-layer software FIFO by invoking this API; however, some special cases may result in invoking failure, and the return value "ble_sts_t" will indicate the corresponding error reason. When this API is invoked in APP layer, it's recommended to check whether the return value is "BLE_SUCCESS". If the return value is not "BLE_SUCCESS", a delay is needed to re-push the data.

The return value is shown as below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Add success
LL_ERR_CONNECTION_NOT_ESTABLISH	See the definition in the SDK	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	See the definition in the SDK	Data cannot be sent during pairing or encryption phase.
LL_ERR_TX_FIFO_NOT_ENOUGH	See the definition in the SDK	Since task with mass data is being executed, software Tx fifo is not enough.
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	See the definition in the SDK	Data cannot be sent during service discovery phase.
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	See the definition in the SDK	The previous indicate data has not been confirmed by Master.

Note: Another alternative API has been added to the SDK (using `min(EffectiveMaxTxOctets, EffectiveRxMTU)` as the minimum unit for subpackaging and can be called by both master and slave, users are advised to use the new API).

```
ble_sts_t blc_gatt_pushHandleValueIndicate (u16 connHandle, u16 attHandle, u8 *p, int len);
```

When calling this API, it is recommended that users check whether the return value is BLE_SUCCESS, and whether it differs from the `bls_att_pushIndicateData` return value:

- 1) When in the pairing phase, the new API returns the value: SMP_ERR_PAIRING_BUSY;
- 2) When in the encryption phase, the new API returns the value: LL_ERR_ENCRYPTION_BUSY;
- 3) When len is greater than ATT_MTU-3 (3 is the ATT layer packet format length opcode and handle), it means that the data length PDU to be sent exceeds the maximum PDU length ATT_MTU supported by the ATT layer, and the return value is GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE.

(13) Handle Value Confirmation

The details of Handle Value Confirmation refers to "Core_v5.0" (Vol 3/Part F/3.4.7.3).

Whenever the API "`bls_att_pushIndicateData`" (or "`blc_gatt_pushHandleValueIndicate`") is invoked by APP layer to send an indicate data to Master, Master will respond with "Confirmation" to confirm the data, then Slave can continue to send the next indicate data.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1E = Handle Value Confirmation

Table 3.36: Format of Handle Value Confirmation

Figure 3.62: "Handle Value Confirmation in BLE Spec"

As shown above, "Confirmation" is not specific to indicate data of certain handle, and the same "Confirmation" will be responded irrespective of handle.

To enable the APP layer to know whether the indicate data has already been confirmed by Master, user can register the callback of GAP event (see section 3.3.5.2 GAP event), and enable corresponding eventMask "GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM" to obtain Confirm event.

3.3.3.4 GATT Service Security

Before reading "GATT Service Security", user can refer to section 3.3.4 SMP to learn basic knowledge related to SMP including LE pairing method, security level, and etc.

The figure below shows the mapping relationship of service request for GATT Service Security level given by BLE spec. Please refer to "core5.0" (Vol3/Part C/10.3 AUTHENTICATION PROCEDURE).

Link Encryption State	Local Device's Access Requirement for Service	Local Device Pairing Status			
		No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections
Unencrypted	None	Request succeeds	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection, Secure Connections	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
Encrypted	None	N/A (Not possible to be encrypted without LTK)	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection		Request succeeds	Request succeeds	Request succeeds
	Encryption, MITM Protection		Error Resp.: Insufficient Authentication	Request succeeds	Request succeeds
	Encryption, MITM Protection, Secure Connections		Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Authentication	Request succeeds

Table 10.2: Local device responds to a service request

Figure 3.63: "Mapping Diagram for Service Request and Response"

As shown in the figure above:

- The first column marks whether currently connected Slave device is in encryption state;
- The second column (local Device's Access Requirement for service) is related to Permission Access setting for attributes in ATT table;
- The third column includes four sub-columns corresponding to four levels of LE security mode1 for current device pairing state:

- a) No authentication and no encryption
- b) Unauthenticated pairing with encryption
- c) Authenticated pairing with encryption
- d) Authenticated LE Secure Connections

```

/** @defgroup ATT_PERMISSIONS_BITMAPS GAP ATT Attribute Access Permissions Bit Fields
 * @{
 * (See the Core_v5.0(Vol 3/Part C/10.3.1/Table 10.2) for more information)
 */
#define ATT_PERMISSIONS_AUTHOR      0x10 //Attribute access(Read & Write) requires Authorization
#define ATT_PERMISSIONS_ENCRYPT      0x20 //Attribute access(Read & Write) requires Encryption
#define ATT_PERMISSIONS_AUTHEN      0x40 //Attribute access(Read & Write) requires Authentication(MITM protection)
#define ATT_PERMISSIONS_SECURE_CONN 0x80 //Attribute access(Read & Write) requires Secure Connection
#define ATT_PERMISSIONS_SECURITY    (ATT_PERMISSIONS_AUTHOR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN | ATT_PERMISSIONS_SECURE_CONN)

//user can choose permission below
#define ATT_PERMISSIONS_READ        0x01 //!< Attribute is Readable
#define ATT_PERMISSIONS_WRITE       0x02 //!< Attribute is Writable
#define ATT_PERMISSIONS_RDWR       (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_WRITE) //!< Attribute is Readable & Writable

#define ATT_PERMISSIONS_ENCRYPT_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_ENCRYPT) //!< Read requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT) //!< Write requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT) //!< Read & Write requires Encryption

#define ATT_PERMISSIONS_AUTHEN_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Read requires Authentication
#define ATT_PERMISSIONS_AUTHEN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Write requires Authentication
#define ATT_PERMISSIONS_AUTHEN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Read & Write requires Authentication

#define ATT_PERMISSIONS_SECURE_CONN_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)

#define ATT_PERMISSIONS_AUTHOR_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_AUTHOR) //!< Read requires Authorization
#define ATT_PERMISSIONS_AUTHOR_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_AUTHOR) //!< Write requires Authorization
#define ATT_PERMISSIONS_AUTHOR_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_AUTHOR) //!< Read & Write requires Authorization

```

Figure 3.64: "ATT Permission Definition"

The final implementation of GATT Service Security is related to parameter settings during SMP initialization, including the highest security level, permission access of attributes in ATT table. It is also related to Master, for example, suppose Slave sets the highest security level supported by SMP as "Authenticated pairing with encryption", but the highest level supported by Master is "Unauthenticated pairing with encryption"; if the permission for some write attribute in ATT table is "ATT_PERMISSIONS_AUTHEN_WRITE", when Master writes this attribute, an error will be responded to indicate "encryption level is not enough".

User can set permission of attributes in ATT table to implement the application below:

Suppose the highest security level supported by Slave is "Unauthenticated pairing with encryption", but it's not hoped to trigger Master pairing by sending "Security Request" after connection, user can set the permission for CCC (Client Characteristic Configuration) attribute with notify attribute as "ATT_PERMISSIONS_ENCRYPT_WRITE". Only when Master writes the CCC, will Slave respond that security level is not enough and trigger Master to start pairing encryption.

Note:

- Security level set by user only indicates the highest security level supported by device, and GATT Service Security can be used to realize control as long as ATT Permission does not exceed the highest level that takes effect indeed. For LE security mode1 level 4, if use only sets the level "Authenticated LE Secure Connections", the setting supports LE Secure Connections only.

For the example of GATT security level, please refer to "b85m_feature_test/ feature_gatt_security/app.c".

3.3.3.5 B85m master GATT

In the b85m_master kma dongle, the following GATT API is provided for doing simple service discovery or other data access functions.

```
void att_req_find_info(u8 *dat, u16 start_attHandle, u16 end_attHandle);
```

The real length of dat (byte): 11

```
void att_req_find_by_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid, u8*  
↪ attr_value, int len);
```

The real length of dat (byte): 13 + attr_value length

```
void att_req_read_by_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid, int  
↪ uuid_len);
```

The real length of dat (byte): 11 + uuid length

```
void att_req_read (u8 *dat, u16 attHandle);
```

The real length of dat (byte): 9

```
void att_req_read_blob (u8 *dat, u16 attHandle, u16 offset);
```

The real length of dat (byte): 11

```
void att_req_read_by_group_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid,  
↪ int uuid_len);
```

The real length of dat (byte): 11 + uuid length

```
void att_req_write (u8 *dat, u16 attHandle, u8 *buf, int len);
```

The real length of dat (byte): 9 + buf data length

```
void att_req_write_cmd (u8 *dat, u16 attHandle, u8 *buf, int len);
```

The real length of dat (byte): 9 + buf data length

For the above API, you need to pre-define the memory space *dat, then call the API to assemble the data, and finally call blm_push_fifo to send the dat to the Controller, and note that you need to determine whether the return value is TRUE. Take att_req_find_info as an example, other interfaces can use similar methods.

```
u8 cmd[12];
att_req_find_info(cmd, 0x0001, 0x0003);
if( blm_push_fifo (BLM_CONN_HANDLE, cmd) ){
    //cmd send OK
}
```

After sending the corresponding find info req, read req, etc. cmd to the Slave using the method referenced above, you will soon receive the corresponding response message from the Slave in reply to find info rsp, read rsp, etc. in int app_l2cap_handler (u16 conn_handle, u8 *raw_pkt) will be processed according to the following framework.

```
if(ptrL2cap->chanId == L2CAP_CID_ATTR_PROTOCOL) //att data
{
    if(pAtt->opcode == ATT_OP_EXCHANGE_MTU_RSP){
        //add your code
    }
    if(pAtt->opcode == ATT_OP_FIND_INFO_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_FIND_BY_TYPE_VALUE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BLOB_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_GROUP_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_WRITE_RSP){
        //add your code
    }
}
```

3.3.4 SMP

Security Manager (SM) in BLE is mainly used to provide various encryption keys for LE device to ensure data security. Encrypted link can protect the original contents of data in the air from being intercepted, decoded or read by any attacker. For details about the SMP, please refer to "Core_v5.0" (Vol 3/Part H/ Security Manager Specification).

3.3.4.1 SMP Security Level

BLE 4.2 Spec adds a new pairing method "LE Secure Connections" which further strengthens security. The pairing method in earlier version is called "LE legacy pairing".

Recalling the section "GATT service Security", the following types of pairing status are available for local devices.

Local Device Pairing Status			
No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections

Figure 3.65: "Local Device Pairing Status"

The four states correspond to the four levels of LE security mode1, details refer to "Core_v5.0" (Vol 3//Part C/10.2 LE SECURITY MODES)

- No authentication and no encryption (LE security mode1 level1)
- Unauthenticated pairing with encryption (LE security mode1 level2)
- Authenticated pairing with encryption (LE security mode1 level3)
- Authenticated LE Secure Connections (LE security mode1 level4)

Note:

Security level set by local device only indicates the highest security level that local device may reach. However, to reach the preset level indeed, the two factors below are important:

- The supported highest security level set by peer Master device \geq the supported highest security level set by local Slave device.
- Both local device and peer device complete the whole pairing process (if pairing exists) correctly as per the preset SMP parameters.

For example, even if the highest security level supported by Slave is set as "mode1 level3" (Authenticated pairing with encryption), when the highest security level supported by peer Master is set as "mode1 level1"

(No authentication and no encryption), after connection Slave and Master won't execute pairing, and indeed Slave uses security mode1 level 1.

User can use the API below to set the highest security level supported by SM:

```
void blc_smp_setSecurityLevel(le_security_mode_level_t mode_level);
```

Following shows the definition for the enum type le_security_mode_level_t:

```
typedef enum {
    LE_Security_Mode_1_Level_1 = BIT(0),          No_Authentication_No_Encryption = BIT(0),
    ↪ No_Security = BIT(0),
    LE_Security_Mode_1_Level_2 = BIT(1),          Unauthenticated_Paring_with_Encryption = BIT(1),
    LE_Security_Mode_1_Level_3 = BIT(2),          Authenticated_Paring_with_Encryption = BIT(2),
    LE_Security_Mode_1_Level_4 = BIT(3),
    ↪ Authenticated_LE_Secure_Connection_Paring_with_Encryption =BIT(3),
    .....
}le_security_mode_level_t;
```

3.3.4.2 SMP Parameter Configuration

SMP parameter configuration In Telink BLE SDK is introduced according to the configuration of four SMP security levels. For Slave, SMP function currently can support the highest security level "LE security mode1 level4"; for master, currently the SMP function in the traditional pairing method can support the highest security level "LE security mode1 level2" (traditional pairing Just Works method).

(1) LE security mode1 level1

Level 1 indicates device does not support encryption pairing. If it's needed to disable SMP function, user only needs to invoke the function below during initialization:

```
blc_smp_setSecurityLevel(No_Security);
```

It means the device won't implement pairing encryption for current connection. Even if the peer requests for pairing encryption, the device will reject it. It generally applies to the device that does not support encryption pairing process. As shown in the figure below, Master sends a pairing request, and Slave responds with "SM_Pairing_Failed".

0x2AC799C5	S->M	OK	Empty PDU	1	1	0	0	0	0x000011	-54	OK									
Access Address	Direction	ACK Status	Data Type	Data Header					L2CAP Header		SM_Pairing_Req							CRC		
0x2AC799C5	?	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist	0x000014		
0x2AC799C5	?	OK	L2CAP-S	2	1	1	0	11	0x0007	0x0006	0x01	0x04	0x00	0x05	0x10	0x07	0x07	0x000014		
Access Address	Direction	ACK Status	Data Type	Data Header					CRC	RSSI	FCS									
0x2AC799C5	?	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x000014	-54	OK									
0x2AC799C5	?	OK	Empty PDU	1	0	1	0	0	0x000014	-54	OK									
Access Address	Direction	ACK Status	Data Type	Data Header					CRC	RSSI	FCS									
0x2AC799C5	?	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x000015	-62	OK									
0x2AC799C5	?	OK	Empty PDU	1	0	0	0	0	0x000015	-62	OK									
Access Address	Direction	ACK Status	Data Type	Data Header					L2CAP Header		SM_Pairing_Failed		CRC	RSSI	FCS					
0x2AC799C5	?	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Reason	0x00000E	-54	OK					
0x2AC799C5	?	OK	L2CAP-S	2	1	0	0	6	0x0002	0x0006	0x05	0x05								

Figure 3.66: "Packet Example for Pairing Disable"

(2) LE security mode1 level2

Level 2 indicates device supports the highest security level “Unauthenticated_Paring_with_Encryption”, e.g. “Just Works” pairing mode in legacy pairing and secure connection pairing method.

A. As introduced earlier, SMP supports legacy encryption and secure connection pairing. The SDK provides the API below to set whether the new encryption feature in BLE4.2 is supported.

```
void blc_smp_setParingMethods (paring_methods_t method);
```

Following shows the definition for the enum type paring_methods_t:

```
typedef enum {  
    LE_Legacy_Paring      = 0,    // BLE 4.0/4.2  
    LE_Secure_Connection = 1,    // BLE 4.2/5.0/5.1  
}paring_methods_t;
```

B. When using security level other than LE security mode1 level1, the API below must be invoked to initialize SMP parameter configuration, including flash initialization setting of bonded area.

```
int blc_smp_peripheral_init (void);
```

If only this API is invoked during initialization, the SDK will use default parameters to configure SMP:

- The highest security level supported by default: Unauthenticated_Paring_with_Encryption.
- Default bonding mode: Bondable_Mode (store KEY that is distributed after pairing encryption into flash).
- Default IO capability: IO_CAPABILITY_NO_INPUT_NO_OUTPUT.

The default parameters above follow the configuration of legacy pairing “Just Works” mode. Therefore invoking this API only is equivalent to configure LE security mode1 level2. LE security mode1 level2 has two types of setting:

A. Device supports initialization setting of “Just Works” in legacy pairing.

```
blc_smp_peripheral_init();
```

B. Device supports initialization setting of “Just Works” in secure connections.

```
blc_smp_setParingMethods(LE_Secure_Connection);  
blc_smp_peripheral_init();
```

(3) LE security mode1 level3

Level 3 indicates device supports the highest security level “Authenticated pairing with encryption”, e.g. “Passkey Entry” / “Out of Band” in legacy pairing mode.

As required by this level, device should support Authentication, i.e. legal identity of two pairing sides should be ensured. The three Authentication methods below are supported in BLE:

- Method 1 with involvement of user, e.g. device has button or display capability, so that one side can display TK, while the other side can input the same TK (e.g. Passkey Entry).
- Method 2: The two pairing sides can exchange information using the method of non-BLE RF transfer to implement pairing (e.g. Out of Band which transfers TK via NFC generally).
- Method 3: Use the TK negotiated and agreed by two device sides (e.g. Just Works with TK 0 used by two sides). Since this method is Unauthenticated, the security level of "Just Works" corresponds to LE security mode1 level2.

Authentication can ensure the legality of two pairing sides, and this protection method is called MITM (Man-in-the-Middle) protection.

A. Device with Authentication should set its MITM flag or OOB flag. The SDK provides the two APIs below to set MITM flag and OOB flag.

```
void blc_smp_enableAuthMITM (int MITM_en);
void blc_smp_enableOobAuthentication (int OOB_en);
```

"MITM_en"/"OOB_en": 1 - enable; 0 - disable.

B. As introduced earlier, SM provides three Authentication methods selectable depending on IO capability of two sides. The SDK provides the API below to set IO capability for current device.

```
void blc_smp_setIoCapability (io_capability_t ioCapability);
```

Following shows the definition for the enum type io_capability_t:

```
typedef enum {
    IO_CAPABILITY_UNKNOWN = 0xff,
    IO_CAPABILITY_DISPLAY_ONLY = 0,
    IO_CAPABILITY_DISPLAY_YESNO = 1,
    IO_CAPABILITY_KEYBOARD_ONLY = 2,
    IO_CAPABILITY_NO_IN_NO_OUT = 3,
    IO_CAPABILITY_KEYBOARD_DISPLAY = 4,
} io_capability_t;
```

C. The figure below shows the rule to use MITM flag and OOB flag in legacy pairing mode.

		Initiator			
		OOB Set	OOB Not Set	MITM Set	MITM Not Set
Responder	OOB Set	Use OOB	Check MITM		
	OOB Not Set	Check MITM	Check MITM		
	MITM Set			Use IO Capabilities	Use IO Capabilities
	MITM Not Set			Use IO Capabilities	Use Just Works

Table 2.6: Rules for using Out-of-Band and MITM flags for LE legacy pairing

Figure 3.67: “Usage Rule for MITM OOB Flag in Legacy Pairing Mode”

The OOB and MITM flag of local device and peer device will be checked to determine whether to use OOB method or select certain KEY generation method as per IO capability. As shown in the figure below, the SDK will select different KEY generation methods according to IO capability (Row/Column parameter type `io_capability_t`):

```
// H: Initiator Capabilities
// V: Responder Capabilities
// See the Core v5.0(Vol 3/Part H/2.3.5.1) for more information.
static const stk_generationMethod_t gen_method_legacy[5 /*Responder*/][5 /*Initiator*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, JustWorks, PK_Init_Dsply_Resp_Input },
};

#if SECURE_CONNECTION_ENABLE
static const stk_generationMethod_t gen_method_sc[5 /*Responder*/][5 /*Initiator*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks, Numric_Comparison, PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsply_Resp_Input, Numric_Comparison, PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
};
#endif
```

Figure 3.68: “Mapping Relationship for KEY Generation Method and IO Capability”

For details about the mapping relationship, please refer to “core5.0” (Vol3/Part H/2.3.5.1 Selecting Key Generation Method).

LE security mode1 level 3 supports the methods below to configure initial values:

A. Initialization setting of OOB for device with legacy pairing:

```
blc_smp_enableOobAuthentication(1);
blc_smp_peripheral_init(); //SMP parameter configuration must be placed before this API
```

Considering TK value transfer by OOB, the SDK provides related GAP event in the APP layer (see section 3.3.5.2 GAP event). The API below serves to set TK value of OOB.


```
void blc_smp_setTK_by_00B (u8 *oobData);
```

The parameter “oobData” indicates the head pointer for the array of 16-digit TK value to be set.

B. Initialization setting of Passkey Entry (PK_Resp_Dsply_Init_Input) for device with legacy pairing:

```
blc_smp_enableAuthMITM(1);  
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);  
blc_smp_peripheral_init();
```

C. Initialization setting of Passkey Entry (PK_Init_Dsply_Resp_Input or PK_BOTH_INPUT) for device with legacy pairing:

```
blc_smp_enableAuthMITM(1);  
blc_smp_setIoCapability(IO_CAPABILITY_KEYBOARD_ONLY);  
blc_smp_peripheral_init();
```

Considering TK value input by user, the SDK provides related GAP event in the APP layer (see section 3.3.5.2 GAP event). The API below serves to set TK value of Passkey Entry:

```
void blc_smp_setTK_by_PasskeyEntry (u32 pinCodeInput);
```

The parameter “pinCodeInput” indicates the pincode value to be set and its range is 0-9999999. It applies to the case of Passkey Entry method in which Master displays TK and Slave needs to input TK.

The KEY generation method finally adopted is related to SMP security level supported by two pairing sides. If Master only supports LE security mode1 level1, since Master does not support pairing encryption, Slave won't enable SMP function.

(4) LE security mode1 level4

Level 4 indicates device supports the highest security level “Authenticated LE Secure Connections”, e.g. Numeric Comparison/Passkey Entry/Out of Band in secure connection pairing mode.

LE security mode1 level4 supports the methods below to configure initial values:

A. Initialization setting of Numeric Comparison for device with secure connection pairing:

```
blc_smp_setParingMethods(LE_Secure_Connection);  
blc_smp_enableAuthMITM(1);  
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_YESNO);
```

Considering display of numerical comparison result to user, the SDK provides related GAP event in the APP layer (see section 3.3.5.2 GAP event).The API below serves to set numerical comparison result as “YES” or “NO”.

```
void blc_smp_setNumericComparisonResult(bool YES_or_NO);
```

The parameter "YES_or_NO" serves to confirm whether six-digit values on two sides are consistent. If yes, input 1 to indicate "YES"; otherwise input 0 to indicate "NO".

B. Initialization setting of Passkey Entry for device with secure connection pairing:

User initialization code of this part is almost the same with that of the configuration mode B/C (Passkey Entry in legacy pairing) in LE security mode1 level3, except that pairing method herein should be set as "secure connection pairing" at the start of initialization.

```
blc_smp_setPairingMethods(LE_Secure_Connection);  
.....//Refer to configuration method B/C in LE security mode1 level3
```

C. Initialization setting of Out of Band for device with secure connection pairing:

This part is not implemented in current SDK yet.

(5) Several APIs related to SMP parameter configuration:

A. The API below serves to set whether to enable bonding function::

```
void blc_smp_setBondingMode(bonding_mode_t mode);
```

Following shows the enum type bonding_mode_t:

```
typedef enum {  
    Non_Bondable_Mode = 0,  
    Bondable_Mode     = 1,  
}bonding_mode_t;
```

For device with security level other than mode1 level1, bonding function must be enabled. Since the SDK has enabled bonding function by default, generally user does not need to invoke this API.

B. The API below serves to set whether to enable Key Press function:

```
void blc_smp_enableKeypress (int keyPress_en);
```

It indicates whether it's supported to provide some necessary input status information for KeyboardOnly device during Passkey Entry. Since the current SDK does not support this function yet, the parameter must be set as 0.

C. The API below serves to set whether to enable key pairs for ECDH (Elliptic Curve Diffie-Hellman) debug mode:

```
void blc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode);
```

Following shows the definition for the enum type ecdh_keys_mode_t:

```
typedef enum {
    non_debug_mode = 0, //ECDH distribute private/public key pairs
    debug_mode = 1, //ECDH use debug mode private/public key pairs
} ecdh_keys_mode_t;
```

This API only applies to the case with secure connection pairing. The ellipse encryption algorithm can prevent eavesdropping effectively, but at the same time, it's not very friendly to debugging and development, since user cannot capture BLE packet in the air by sniffer and analyze the data. Thus, as defined in BLE spec, ellipse encryption mode with private and public key pairs is provided for debugging. As long as this mode is enabled, BLE sniffer tool can use the known key to decrypt the link.

D. Following is a unified API to set whether to enable bonding, whether to enable MITM flag, whether to support OOB, whether to support Keypress notification, as well as to set supported IO capability(The previous documents are all separate configuration APIs. For the convenience of user settings, the SDK also provides a unified configuration API).

```
void bhc_smp_setSecurityParameters (bonding_mode_t mode, int MITM_en, int OOB_en, int keyPress_en,
io_capability_t ioCapability);
```

Definition for each parameter herein is consistent with the same parameter in the corresponding independent API.

3.3.4.3 Security Request Configuration

Only Slave can send SMP Security Request, so this part only applies to Slave device.

During phase 1 of pairing process, there's an optional Security Request packet which serves to enable Slave to actively trigger pairing process to start. The SDK provides the API below to flexibly set whether Slave sends Security Request to Master immediately after connection/re-connection, or delay for pending_ms milliseconds before sending Security Request, or does not send Security Request, so as to implement different pairing trigger combination.

```
bhc_smp_configSecurityRequestSending( secReq_cfg newConn_cfg, secReq_cfg reConn_cfg, u16
↪ pending_ms);
```

Following shows the definition for the enum type secReq_cfg:

```
typedef enum {
    SecReq_NOT_SEND = 0,
    SecReq_IMM_SEND = BIT(0),
    SecReq_PEND_SEND = BIT(1),
}secReq_cfg;
```

The meaning of each parameter is introduced as below:

- SecReq_NOT_SEND: After connection is established, Slave won't send Security Request actively.

- SecReq_IMM_SEND: After connection is established, Slave will send Security Request immediately.
 - SecReq_PEND_SEND: After connection is established, Slave will wait for pending_ms milliseconds and then determine whether to send Security Request.
- (1) For the first connection, Slave receives Pairing_request from Master before pending_ms milliseconds, and it won't send Security Request;
 - (2) For re-connection, if Master has already sent LL_ENC_REQ before pending_ms milliseconds to encrypt reconnection link, Slave won't send Security Request.

The parameter "newConn_cfg" serves to configure new device, while the parameter "reConn_cfg" serves to configure device to be reconnected. During reconnection, the SDK also supports the configuration whether to send purpose of pairing request: During reconnection for a bonded device, Master may not actively initiate LL_ENC_REQ to encrypt link, and Security Request sent by Slave will trigger Master to actively encrypt the link. Therefore, the SDK provides reConn_cfg configuration, and user can configure it as needed.

Note:

- This API must be invoked before connection. It's recommended to invoke it during initialization.

The input parameters for the API "blc_smp_configSecurityRequestSending" supports the nine combinations below:

Table 3.13: Input parameter combination of blc_smp_configSecurityRequestSending

Parameter	SecReq_NOT_SEND	SecReq_IMM_SEND	SecReq_PEND_SEND
SecReq_NOT_SEND	Not send SecReq after the first connection or reconnection (the para pending_ms is invalid).	Not send ecReq after the first connection, and immediately send SecReq after reconnection (the para pending_ms is invalid).	Not send ecReq after the first connection, and wait for pending_ms milliseconds to send SecReq after reconnection.
SecReq_IMM_SEND	Immediately send SecReq after the first connection, and not send SecReq after reconnection (the para pending_ms is invalid).	Immediately send SecReq after the first connection or reconnection (the para pending_ms is invalid).	Immediately send SecReq after the first connection, and wait for pending_ms milliseconds to send SecReq after reconnection.
SecReq_PEND_SEND	Wait for pending_ms milliseconds to send SecReq after the first connection, and not send SecReq after reconnection.	Wait for pending_ms milliseconds to send SecReq after the first connection, and immediately send SecReq after reconnection.	Wait for pending_ms milliseconds to send SecReq after the first connection or reconnection.

Following shows two examples:

(1) newConn_cfg: SecReq_NOT_SEND

reConn_cfg: SecReq_NOT_SEND

pending_ms: This parameter does not take effect.

When newConn_cfg is set as SecReq_NOT_SEND, it means new Slave device won't actively initiate Security Request, and it will only respond to the pairing request from the peer device. If the peer device does not send pairing request, encryption pairing won't be executed. As shown in the figure below, when Master sends a pairing request packet "SM_Pairing_Req", Slave will respond to it, but won't actively trigger Master to initiate pairing request.

Access Address	Direction	ACK Status	Data Type	Data Header	CRC	RSI (dBm)	FCS		
xA84714E5	S→M	OK	Empty PDU	LLID NESN SN MD PDU-Length 1 1 0 0 0	0x00000D	-54	OK		
Access Address	Direction	ACK Status	Data Type	Data Header	L2CAP Header	SM_Pairing_Req		CRC	RSI (dBm)
xA84714E5	?	OK	L2CAP-S	LLID NESN SN MD PDU-Length 2 1 1 0 11	L2CAP-Length ChanId 0x0007 0x0006	Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist 0x01 0x04 0x00 0x05 0x10 0x07 0x07	0x000008	-78	
Access Address	Direction	ACK Status	Data Type	Data Header	CRC	RSI (dBm)	FCS		
xA84714E5	?	OK	Empty PDU	LLID NESN SN MD PDU-Length 1 0 1 0 0	0x00001C	-54	OK		
Access Address	Direction	ACK Status	Data Type	Data Header	CRC	RSI (dBm)	FCS		
xA84714E5	?	OK	Empty PDU	LLID NESN SN MD PDU-Length 1 0 0 0 0	0x00000C	-78	OK		
Access Address	Direction	ACK Status	Data Type	Data Header	L2CAP Header	SM_Pairing_Rsp		CRC	RSI (dBm)
xA84714E5	?	OK	L2CAP-S	LLID NESN SN MD PDU-Length 2 1 0 0 11	L2CAP-Length ChanId 0x0007 0x0006	Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist 0x02 0x03 0x00 0x01 0x10 0x03 0x03	0x000012	-54	

Figure 3.69: "Packet Example for Pairing Peer Trigger"

When reConn_cfg is set as SecReq_NOT_SEND, it means device pairing has already been completed, and Slave won't send Security Request after reconnection.

(2) newConn_cfg: SecReq_IMM_SEND

reConn_cfg: SecReq_NOT_SEND

pending_ms: This parameter does not take effect.

When newConn_cfg is set as SecReq_IMM_SEND, it means new Slave device will immediately send Security Request to Master after connection, to trigger Master to start pairing process. As shown in the figure below, Slave actively sends a SM_Security_Req to trigger Master to send pairing request.

592	=8321694	0x09	0x4CD612E9	M->S	OK	Control	3 0 0 0 9	Feature_Req(0x08)	00 00 00 00 00 00 01	0x000021	-54	OK
Pnbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header	L2CAP-Header	SM_Security_Req	CRC	RSI (dBm)	FCS
593	=8321995	0x09	0x4CD612E9	S->M	OK	L2CAP-S	LLID NESN SN MD PDU-Length 2 1 0 0 6	L2CAP-Length ChanId 0x0002 0x0006	Opcode AuthReq 0x0B 01	0x000041	-54	OK
Pnbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header	L2CAP-Header	SM_Pairing_Req	CRC	RSI (dBm)	FCS
594	=8361694	0x12	0x4CD612E9	M->S	OK	L2CAP-S	LLID NESN SN MD PDU-Length 2 1 1 0 11	L2CAP-Length ChanId 0x0007 0x0006	Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist 0x01 0x04 0x00 0x0D 0x10 0x0F	0x000008	-54	OK
Pnbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header	LL_Opcode	LL_Feature_Rsp	CRC	RSI (dBm)	FCS

Figure 3.70: "Packet Example for Pairing Conn Trigger"

When reConn_cfg is set as SecReq_NOT_SEND, it means Slave won't send Security Request after reconnection.

The SDK also provides an API to send Security Request packet only for special use case. The APP layer can invoke this API to send Security Request at any time.

```
int blc_smp_sendSecurityRequest (void);
```

Note: If user invokes the "blc_smp_configSecurityRequestSending" to control secure pairing request packet, the "blc_smp_sendSecurityRequest" should not be invoked.

3.3.4.4 SMP Bonding info

SMP bonding information herein is discussed relative to Slave device. User can refer to the code of “direct advertising” setting during initialization in the SDK demo “b85m_ble_remote”.

Slave can store pairing information of up to four Master devices at the same time, so that all of the four devices can be reconnected successfully. The API below serves to set the max number of bonding devices with the upper limit of 4 which is also the default value.

```
#define SMP_BONDING_DEVICE_MAX_NUM 4
ble_sts_t blc_smp_param_setBondingDeviceMaxNumber( int device_num);
```

If using `blc_smp_param_setBondingDeviceMaxNumber (4)` to set the max number as 4, after four devices have been paired, excuting pairing for the fifth device will automatically delete the pairing info of the earliest connected (first) device, so as to store the pairing info of the fifth device.

If using `blc_smp_param_setBondingDeviceMaxNumber (2)` to set the max number as 2, after two devices have been paired, excuting pairing for the third device will automatically delete the pairing info of the earliest connected (first) device, so as to store the pairing info of the third device.

The API below serves to obtain the number of currently bonded Master devices (successfully paired with Slave) stored in the flash.

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(void);
```

Assuming a return value of 3, this means that there are currently 3 successfully paired devices stored on the flash, and that all 3 devices can be connected back successfully.

(1) Storage sequence for bonding info

Index is a concept related to `BondingDeviceNumber`. If current `BondingDeviceNumber` is 1, there’s only one bonding device whose index is 0; if `BondingDeviceNumber` is 2, there’re two bonding devices with index 0 and 1.

The SDK provides two methods to update device index, `Index_Update_by_Connect_Order` and `Index_Update_by_Pairing_Order`, i.e. update index as per the time sequence of latest connection or pairing for devices. The API below serves to select index update method.

```
void bls_smp_setIndexUpdateMethod(index_updateMethod_t method);
```

Following shows the enum type `index_updateMethod_t`:

```
typedef enum {
    Index_Update_by_Pairing_Order = 0,    //default value
    Index_Update_by_Connect_Order = 1,
} index_updateMethod_t;
```

Two index update methods are introduced below:

A. Index_Update_by_Connect_Order

If BondingDeviceNumber is 2, device index stored in Slave flash includes 0 and 1. Index sequence is updated by the order of the latest successful connection rather than the latest pairing. Suppose Slave is paired with MasterA and MasterB in sequence, since MasterB is the latest connected device, the index for MasterA is 0, while the index for MasterB is 1. Then reconnect Slave with MasterA. Now MasterA becomes the latest connected device, so the index for MasterB is 0, and the index for MasterA is 1.

If BondingDeviceNumber is 3, device index includes 0, 1 and 2. The index for the latest connected device is 2, and index for the earliest connected device is 0.

If BondingDeviceNumber is 4, device index includes 0, 1, 2 and 3. The index for the latest connected device is 3, and index for the earliest connected device is 0. Suppose Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, the index for the latest connected MasterD is 3. If Slave is reconnected with MasterB, the index for the latest connected MasterB is 3.

Since the upper limit for bonding devices is 4, please note the case when more than four Master devices are paired: When Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, pairing Slave with MasterE will make Slave delete the pairing info for MasterA; however, if Slave is reconnected with MasterA before pairing Slave with MasterE, since the sequence changes to B-C-D-A, the latest pairing operation between Slave and MasterE will delete the pairing info for MasterB.

B. Index_Update_by_Pairing_Order

If BondingDeviceNumber is 2, device index stored in Slave flash includes 0 and 1. Index sequence is updated by the order of the latest pairing. Suppose Slave is paired with MasterA and MasterB in sequence, since MasterB is the latest paired device, the index for MasterA is 0, while the index for MasterB is 1. Then reconnect Slave with MasterA. Now the index sequence for MasterA and MasterB is not changed.

If BondingDeviceNumber is 4, device index includes 0, 1, 2 and 3. The index for the latest paired device is 3, and the index for the earliest paired device is 0. Suppose Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, the index for the latest paired MasterD is 3. No matter how Slave is reconnected with MasterA/B/C/D, the index sequence won't be changed.

Note:

- When Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, pairing Slave with MasterE will make Slave delete the pairing info for MasterA; if Slave is reconnected with MasterA before pairing Slave with MasterE, since the sequence is still A-B-C-D, the latest pairing operation between Slave and MasterE will delete the pairing info for MasterA.

(2) Format for bonding info and related APIs

Bonding info of Master device is stored in flash with the format below:

```
typedef struct {
    u8      flag;
    u8      peer_addr_type; //address used in link layer connection
    u8      peer_addr[6];
    u8      peer_key_size;
    u8      peer_id_addrType; //peer identity address information in key distribution, used to
    ↪ identify
    u8      peer_id_addr[6];
}
```

```

u8    own_ltk[16];    //own_ltk[16]
u8    peer_irk[16];
u8    peer_csrk[16];
}smp_param_save_t;

```

Bonding info includes 64 bytes.

- peer_addr_type and peer_addr indicate Master connection address in the Link Layer which is used during device direct advertising.
- peer_id_addrType/peer_id_addr and peer_irk are identity address and irk declared in the key distribution phase.

Only when the peer_addr_type and peer_addr are Resolvable Private Address (RPA), and address filtering is needed, should related info be added into resolving list for Slave to analyze it (refer to TEST_WHITELIST in the B91_feature_test).

Other parameters are negligible to user.

The API below serves to obtain device information from flash by using index.

```

u32  bls_smp_param_loadByIndex(u8 index, smp_param_save_t* smp_param_load);

```

If the return value is 0, it indicates failure to get info; non-zero return value indicates starting flash address to store the info. For example, suppose there're three bonded devices, get the information about the nearest connected device.

```

bls_smp_param_loadByIndex(2, ...)

```

The API below serves to obtain bonding device info from flash by using Master address (connection address in the Link Layer).

```

u32  bls_smp_param_loadByAddr(u8 addr_type, u8* addr, smp_param_save_t* smp_param_load);

```

If the return value is 0, it indicates failure to get info; non-zero return value indicates starting flash address to store the info.

The API below is used for Slave device to erase all pairing info stored in local flash.

```

void  bls_smp_eraseAllParingInformation(void);

```

Note: Before invoking this API, please ensure the device is in non-connection state.

The API below is used for Slave device to configure address to store pairing info in flash.

```

void  bls_smp_configParingSecurityInfoStorageAddr(int addr);

```

User can set the parameter "addr" as needed, and please refer to the section 2.1.4 SDK flash space partition so as to determine a suitable flash area for bonding info storage.

3.3.4.5 master SMP

The highest level of the master SMP function currently supported in the traditional pairing method is LE security mode1 level2 (traditional pairing Just Works method). The user can refer to the "b85m_master kma dongle" and simply modify the macros in the "b85m_master kma dongle/app_config.h" file as follows

```
#define BLE_HOST_SMP_ENABLE 0
```

If this macro is configured to 1, standard SMP is used: the highest security level supported by the configured master is LE security mode1 level2, which supports the traditional pairing Just Works method; if this macro is configured to 0, it means that the non-standard custom pairing management function is enabled.

(1) master enable SMP (set macro BLE_HOST_SMP_ENABLE as 1)

To use this security level configuration, the following API calls must be made to initialize the SMP parameters, including the initial configuration of the bound area FLASH:

```
int blc_smp_central_init (void);
```

If only this API is called during the initialization phase, the SDK will use the default parameters to configure SMP:

- The highest security level supported by default: Unauthenticated_Paring_with_Encryption.
- The default bonding mode: Bondable_Mode (stores the KEY distributed after pairing encryption to FLASH).
- Default IO capability is IO_CAPABILITY_NO_INPUT_NO_OUTPUT.

When the paired device supports LE security mode1 level2, the user is also required to configure the following three APIs.

```
void blm_smp_configParingSecurityInfoStorageAddr (int addr);  
void blm_smp_registerSmpFinishCb (smp_finish_callback_t cb);  
void blm_host_smp_setSecurityTrigger(u8 trigger);
```

Three APIs are described below:

A. void blm_smp_configParingSecurityInfoStorageAddr (int addr);

This API can be used to master the location of the device configuration binding information stored in FLASH, where the parameter addr can be modified according to actual needs.

B. void blm_smp_registerSmpFinishCb (smp_finish_callback_t cb);

This callback function is triggered after the key distribution in the third phase of the pairing is completed and the user can register at the application layer to get the pairing completion event.

C. void blm_host_smp_setSecurityTrigger(u8 trigger);

This API is mainly used to configure whether master initiates encryption and encrypts the link actively when connecting back. The specific parameters can be selected as follows.

```
#define SLAVE_TRIGGER_SMP_FIRST_PAIRING      0
#define MASTER_TRIGGER_SMP_FIRST_PAIRING    BIT(0)
#define SLAVE_TRIGGER_SMP_AUTO_CONNECT      0
#define MASTER_TRIGGER_SMP_AUTO_CONNECT    BIT(1)
```

Specifically: 1) When pairing for the first time, whether the master choose to initiate the pairing request or start pairing after receiving the Security Request from the slave; 2) When connecting back to a device that has already been paired, whether the master initiate the LL_ENC_REQ encrypted link or wait until it receives the Security Request from the slave. Generally, we will configure the master to initiate the pairing request for the first time and send LL_ENC_REQ when reconnecting.

The final user initialisation code reference is as follows and the user can refer to the "b85m_master kma dongle".

```
blm_smp_configParingSecurityInfoStorageAddr(0x78000);
blm_smp_registerSmpFinishCb(app_host_smp_finish);
blc_smp_central_init();
//SMP trigger by master
blm_host_smp_setSecurityTrigger(MASTER_TRIGGER_SMP_FIRST_PAIRING |
↪ MASTER_TRIGGER_SMP_AUTO_CONNECT);
```

As for the following APIs related to binding information on the master end, they are for use by the bottom layer master SMP protocol.

```
int    tbl_bond_slave_search(u8 adr_type, u8 * addr);
int    tbl_bond_slave_delete_by_adr(u8 adr_type, u8 *addr);
void   tbl_bond_slave_unpair_proc(u8 adr_type, u8 *addr);
```

(2) Non-standard self-defined pairing management (set the macro "BLE_HOST_SMP_ENABLE" as 0)

When using self-defined pairing management, initialization related APIs are shown as below:

```
blc_smp_setSecurityLevel(No_Security);//disable SMP function
user_master_host_pairing_flash_init();//custom method
```

A. Design flash storage method

The default flash sector used for pairing is 0x78000 ~ 0x78FFF, and it's modifiable in the "app_config.h".

```
#define FLASH_ADR_PAIRING    0x78000
```

Starting from flash address 0x78000, every eight bytes form an area (named 8 bytes area). Each area can store MAC address of one Slave, and includes 1-byte bonding mark, 1-byte address type and 6-byte MAC address.

```
typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;
```

All valid Slave MAC addresses are stored in 8 bytes areas successively: The first valid Slave MAC address is stored in 0x78000~0x78007, and the mark in 0x78000 is set as "0x5A" to indicate current address is valid. The second valid Slave MAC address is stored in the next 8 bytes area 0x78008~ 0x7800f and the mark in 0x78008 is set as "0x5A".The third valid Slave MAC address is stored in the next 8 bytes area 0x78010~0x78017 and the mark in 0x78010 is set as "0x5A".

To un-pair certain Slave device, it's needed to erase its MAC address in the Dongle side by setting the mark of the corresponding 8 bytes area as "0x00". For example, to erase the MAC address of the first Slave device as shown above, user should set 0x78000 as "0x00".

The reason to adopt this design is: During execution of program, the SDK cannot invoke the function "flash_erase_sector" to erase flash, since this operation takes 20~200ms to erase a 4kB sector of flash and thus will result in BLE timing error.

Mark of "0x5A" and "0x00" are used to indicate pairing storage and un-pairing erasing of all Slave MAC addresses. Considering 8 bytes areas may occupy the whole 4kB sector of flash and thus result in error, a special processing is added during initialization: Read info of 8 bytes areas starting from address 0x78000, and store all valid MAC addresses into Slave MAC table of RAM. During this process, it will check whether there're too many 8 bytes areas. If yes, erase the whole sector and then write the contents of Slave MAC table in RAM back to 8 bytes areas starting from 0x78000.

B. Slave mac table

```
#define USER_PAIR_SLAVE_MAX_NUM    4 //telink demo use max 4, you can change this value
typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;
typedef struct {
    u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM]; //mark paired slave mac address in flash
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM]; //macAddr_t already defined in ble stack
    u8 curNum;
} user_slaveMac_t;
user_slaveMac_t user_tbl_slaveMac;
```

The structure above serves to use Slave MAC table in RAM to maintain all paired devices. The macro "USER_PAIR_SLAVE_MAX_NUM" serves to set the max allowed number of maintainable paired devices, and the default value is 4 which indicates four paired device is maintainable. User can modify this value as needed.

Suppose the "USER_PAIR_SLAVE_MAX_NUM" is set as 3 to indicate up to three paired devices can be maintained. In the "user_tbl_slaveMac", the "curNum" indicates the number of current valid Slave devices in

flash, the array "bond_flash_idx" records offset relative to 0x78000 for starting address of each valid 8 bytes area in flash (When un-pairing certain device, based on corresponding offset, user can locate the mark of the 8 bytes area, and then write the mark as 0x00), while the array "bond_device" records MAC address.

C. Related APIs

Based on the design of flash storage and Slave MAC table above, user can invoke the APIs below.

a) user_master_host_pairing_flash_init

```
void user_master_host_pairing_flash_init(void);
```

This API should be invoked to implement flash initialization when enabling user-defined pairing management.

b) user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

The API above should be invoked when a new device is paired, and it serves to add one Slave MAC address. The return value should be either 1 (success) or 0 (failure). The API will check whether current number of devices in flash and Slave MAC table has reached the maximum. If not, directly add the MAC address of the new device into Slave MAC table, and store it in an 8 bytes area of flash. If yes, the viable processing policy may be: "pairing is not allowed", or "directly delete the earliest MAC address". Telink demos adopts the latter. Since Telink supported max number of paired device is 1, this method will preempt current paired device, i.e. delete current device by using the "user_tbl_slave_mac_delete_by_index(0)" and then add MAC address of new device into Slave MAC table. User can modify the implementation of this API as per his own policy.

c) user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

This API serves to check whether the device is already available in Slave MAC table according to device address reported by adv, i.e. whether the device sending adv packet currently has already been paired with Master. The device that has already been paired can be directly reconnected.

d) user_tbl_slave_mac_delete_by_adr

```
int user_tbl_slave_mac_delete_by_adr(u8 adr_type, u8 *adr)
```

This API serves to delete MAC addr of certain paired device from Slave MAC table by specified address.

e) user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

This API serves to delete MAC addr of certain paired device from Slave MAC table by specified index. The parameter "index" indicates device pairing sequence. If the max pairing number is 1, the index for the paired device is always 0; if the max pairing number is 2, the index for the first paired device is 0, and the index for the second paired device is 1.....

f) user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

This API serves to delete MAC addr of all the paired devices from Slave MAC table.

g) user_tbl_slave_mac_unpair_proc

```
void user_tbl_slave_mac_unpair_proc(void)
```

This API serves to process un-pairing. The demo code adopts the processing method using the default max pairing number (1) to delete all paired devices. User can modify the implementation of the API.

D. Connection and pairing

When Master receives adv packet reported by Controller, it will establish connection with Slave in the two cases below:

Invoke the function "user_tbl_slave_mac_search" to check whether current Slave device has already been paired with Master and un-pairing has not been executed. If yes, Master can automatically establish connection with the device.

```
master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);
if(master_auto_connect) { create connection }
```

If current adv device is not available in Slave MAC table, auto connection won't be initiated, and it's needed to check whether manual pairing condition is met. The SDK provides two manual pairing solutions by default. Premise: Current adv device is close enough. Solution 1: The pairing button on Master Dongle is pressed. Solution 2: Current adv data is pairing adv packet data defined by Telink.

```
//manual paring methods 1: button triggers
user_manual_paring = dongle_pairing_enable && (rssi > -56); //button trigger pairing(rssi
↳ threshold, short distance)
//manual paring methods 2: special paring adv data
if(!user_manual_paring){ //special adv pair data can also trigger pairing
user_manual_paring =
↳ (memcmp(pa->data,telink_adv_trigger_paring,sizeof(telink_adv_trigger_paring)) == 0)
&& (rssi > -56);
}
if(user_manual_paring) { create connection }
```

After connection triggered by manual pairing is established successfully, the current device is added into Slave MAC table when reporting "HCI LE CONECTION ESTABLISHED EVENT".

```
//manual paring, device match, add this device to slave mac table
if(blm_manPair.manual_pair && blm_manPair.mac_type == pCon->peer_adr_type &&
!memcmp(blm_manPair.mac,pCon->mac, 6))
{
    blm_manPair.manual_pair = 0;
    user_tbl_slave_mac_add(pCon->peer_adr_type, pCon->mac);
}
```

E. Un-pairing

```
_attribute_ram_code_void host_pair_unpair_proc (void)
{
    //terminate and unpair proc
    static int master_disconnect_flag;
    if(dongle_unpair_enable){
        if(!master_disconnect_flag && blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){
            if( blm_ll_disconnect(cur_conn_device.conn_handle, HCI_ERR_REMOTE_USER_TERM_CONN) ==
BLE_SUCCESS){
                master_disconnect_flag = 1;
                dongle_unpair_enable = 0;

                #if (BLE_HOST_SMP_ENABLE)
                    tbl_bond_slave_unpair_proc(cur_conn_device.mac_adrType,
↪ cur_conn_device.mac_addr);
                #else
                    user_tbl_salve_mac_unpair_proc();
                #endif
            }
        }
        if(master_disconnect_flag && blc_ll_getCurrentState() != BLS_LINK_STATE_CONN){
            master_disconnect_flag = 0;
        }
    }
}
```

As shown in the code above, when un-pairing condition is triggered, Master first invokes the "blm_ll_disconnect" to terminate connection, and then invokes the "user_tbl_salve_mac_unpair_proc" to process un-pairing. The demo code will directly delete all paired devices. In the default case, the max pairing number is 1, so only one device will be deleted. If user sets the max number larger than 1, the "user_tbl_slave_mac_delete_by_adr" or "user_tbl_slave_mac_delete_by_index" should be invoked to delete specified device.

The demo code provides two conditions to trigger un-pairing:

- The un-pairing button on Master Dongle is pressed.

- The un-pairing key value "0xFF" is received in "HID keyboard report service".

User can modify un-pairing trigger method as needed.

3.3.4.6 SMP Failure Management

When SMP fails, a callback function can be used to control whether the connection is maintained or disconnected. This is implemented as follows.

- a. Register the handler function for the gap layer and open the event mask with the event GAP_EVT_SMP_PAIRING_FAIL , as follows.

```
bhc_gap_registerHostEventHandler( app_host_event_callback );  
bhc_gap_setEventMask( GAP_EVT_MASK_SMP_PAIRING_FAIL );
```

- b. Modify the corresponding processing under this mask in the processing function.

```
int app_host_event_callback (u32 h, u8 *para, int n)  
{  
    u8 event = h & 0xFF;  
  
    switch(event)  
    {  
        case GAP_EVT_SMP_PAIRING_FAIL:  
        {  
            gap_smp_paringFailEvt_t* p = (gap_smp_paringFailEvt_t*)para;  
            //the operation wanted  
        }  
        break;  
  
        default:  
        break;  
    }  
  
    return 0;  
}
```

3.3.5 GAP

3.3.5.1 GAP initialization

GAP initialization for Master and Slave is different. Slave uses the API below to initialize GAP.

```
void bhc_gap_peripheral_init(void);
```

The Master initialises the GAP using the following API.

```
void blc_gap_central_init(void);
```

As introduced earlier, data transfer between the APP layer and the Host is not controlled via GAP; the ATT, SMP and L2CAP can directly communicate with the APP layer via corresponding interface. In current SDK version, the GAP layer mainly serves to process events in the Host layer, and GAP initialization mainly registers processing function entry for events in the Host layer.

3.3.5.2 GAP Event

GAP event is generated during the communication process of Host protocol layers such as ATT, GATT, SMP and GAP. As introduced earlier, current SDK supports two types of event: Controller event, and GAP (Host) event. Controller event also includes two sub types: HCI event, and Telink defined event.

GAP event processing is added in current BLE SDK, which enables the protocol stack to layer events more clearly and to process event communication in the user layer more conveniently. SMP related processing, such as Passkey input and notification of pairing result to user, is also included.

If user wants to receive GAP event in the APP layer, it's needed to register the corresponding callback function, and then enable the corresponding mask.

Following shows the prototype and register interface for callback function of GAP event.

```
typedef int (*gap_event_handler_t) (u32 h, u8 *para, int n);
void blc_gap_registerHostEventHandler (gap_event_handler_t handler);
```

The "u32 h" in the callback function prototype is the mark of GAP event which will be frequently used in the bottom layer protocol stack.

Following lists some events which user may use.

```
#define GAP_EVT_SMP_PAIRING_BEAGIN      0
#define GAP_EVT_SMP_PAIRING_SUCCESS    1
#define GAP_EVT_SMP_PAIRING_FAIL       2
#define GAP_EVT_SMP_CONN_ENCRYPTION_DONE 3
#define GAP_EVT_SMP_SECURITY_PROCESS_DONE 4
#define GAP_EVT_SMP_TK_DISPALY        8
#define GAP_EVT_SMP_TK_REQUEST_PASSKEY 9
#define GAP_EVT_SMP_TK_REQUEST_OOB     10
#define GAP_EVT_SMP_TK_NUMERIC_COMPARE 11
#define GAP_EVT_ATT_EXCHANGE_MTU       16
#define GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM 17
```

In the callback function prototype, "para" and "n" indicate data and data length of event. User can refer to the usage below in the "b85m_feature_test/feature_smp_security/app.c" and the implementation of the function "app_host_event_callback".


```
blc_gap_registerHostEventHandler( app_host_event_callback );
```

The API below serves to enable the mask for GAP event.

```
void blc_gap_setEventMask(u32 evtMask);
```

Following lists the definition for some common eventMasks. For other event masks, user can refer to the "ble/gap/gap_event.h".

```
#define GAP_EVT_MASK_SMP_PAIRING_BEAGIN      (1<<GAP_EVT_SMP_PAIRING_BEAGIN)
#define GAP_EVT_MASK_SMP_PAIRING_SUCCESS    (1<<GAP_EVT_SMP_PAIRING_SUCCESS)
#define GAP_EVT_MASK_SMP_PAIRING_FAIL      (1<<GAP_EVT_SMP_PAIRING_FAIL)
#define GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE (1<<GAP_EVT_SMP_CONN_ENCRYPTION_DONE)
#define GAP_EVT_MASK_SMP_SECURITY_PROCESS_DONE (1<<GAP_EVT_SMP_SECURITY_PROCESS_DONE)
#define GAP_EVT_MASK_SMP_TK_DISPALY        (1<<GAP_EVT_SMP_TK_DISPALY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_PASSKEY (1<<GAP_EVT_SMP_TK_REQUEST_PASSKEY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_OOB    (1<<GAP_EVT_SMP_TK_REQUEST_OOB)
#define GAP_EVT_MASK_SMP_TK_NUMERIC_COMPARE (1<<GAP_EVT_SMP_TK_NUMERIC_COMPARE)
#define GAP_EVT_MASK_ATT_EXCHANGE_MTU      (1<<GAP_EVT_ATT_EXCHANGE_MTU)
#define GAP_EVT_MASK_GATT_HANDLE_VLAUE_CONFIRM (1<<GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM)
```

If user does not set GAP event mask via this API, the APP layer won't receive notification when corresponding GAP event is generated.

Note:

- For the description about GAP event below, it's supposed that GAP event callback has been registered, and corresponding eventMask has been enabled.

(1) GAP_EVT_SMP_PAIRING_BEAGIN

Event trigger condition: When entering connection state, Slave sends a SM_Security_Req command, and Master sends a SM_Pairing_Req to request for pairing. When Slave receives the pairing request, this event will be triggered to indicate that pairing starts.

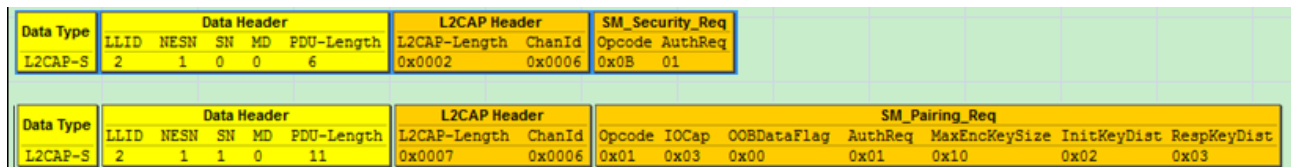


Figure 3.71: "master initiates Pairing_Req"

Data length "n": 4.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u8  secure_conn;
    u8  tk_method;
} gap_smp_paringBeginEvt_t;
```

"connHandle": current connection handle.

"secure_conn": If it's 1, secure encryption feature (LE Secure Connections) will be used; otherwise LE legacy pairing will be used.

"tk_method": It indicates the method of TK value to be used in the subsequent pairing, e.g. JustWorks, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, Numric_Comparison.

(2) GAP_EVT_SMP_PAIRING_SUCCESS

Event trigger condition: This event will be generated when the whole pairing process is completed correctly. This phase is called "Key Distribution, Phase 3" of LE pairing phase. If there's key to be distributed, the pairing success event will be triggered after the two sides have completed key distribution; otherwise the pairing success event will be triggered directly.

Data length "n": 4.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u8  bonding;
    u8  bonding_result;
} gap_smp_paringSuccessEvt_t;
```

"connHandle": current connection handle.

"bonding": If it's 1, bonding function is enabled; otherwise bonding function is disabled.

"bonding_result": It indicates bonding result. If bonding function is disabled, the result value should be 0. If bonding function is enabled, it's also needed to check whether encryption Key is correctly stored in flash; if yes, the result value is 1; otherwise the result value is 0.

(3) GAP_EVT_SMP_PAIRING_FAIL

Event trigger condition: If Slave or Master does not conform to standard pairing flow, or pairing process is terminated due to abnormality such as error report during communication, this event will be triggered.

Data length "n": 2.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u8  reason;
} gap_smp_paringFailEvt_t;
```

"connHandle": current connection handle.

"reason": It indicates the reason for pairing failure. Following lists some common reason values, and for other values, please refer to the file "stack/ble/smp/smp_const.h".

For the definition of pairing failure values, please refer to "Core_v5.0" (Vol 3/Part H/3.5.5 "Pairing Failed").

```
#define PAIRING_FAIL_REASON_CONFIRM_FAILED      0x04
#define PAIRING_FAIL_REASON_PAIRING_NOT_SUPPORTED 0x05
#define PAIRING_FAIL_REASON_DHKEY_CHECK_FAIL   0x0B
#define PAIRING_FAIL_REASON_NUMERIC_FAILED     0x0C
#define PAIRING_FAIL_REASON_PAIRING_TIMEOUT    0x80
#define PAIRING_FAIL_REASON_CONN_DISCONNECT    0x81
```

(4) GAP_EVT_SMP_CONN_ENCRYPTION_DONE

Event trigger condition: When Link Layer encryption is completed (the LL receives "start encryption response" from Master), this event will be triggered.

Data length "n": 3.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u8 re_connect;    //1: re_connect, encrypt with previous distributed LTK; 0: pairing ,
    ↪ encrypt with STK
} gap_smp_connEncDoneEvt_t;
```

"connHandle": current connection handle.

"re_connect": If it's 1, it indicates fast reconnection (The LTK distributed previously will be used to encrypt the link); if it's 0, it indicates current encryption is the first encryption.

(5) GAP_EVT_MASK_SMP_SECURITY_PROCESS_DONE

Event trigger condition: when pairing for the first time, it is triggered after the GAP_EVT_SMP_PAIRING_SUCCESS event, and in the fast reconnection, triggered after GAP_EVT_SMP_CONN_ENCRYPTION_DONE event.

Data length "n": 3.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u8 re_connect;    //1: re_connect, encrypt with previous distributed LTK; 0: pairing ,
    ↪ encrypt with STK
} gap_smp_securityProcessDoneEvt_t;
```

"re_connect": If it's 1, it indicates fast reconnection (The LTK distributed previously will be used to encrypt the link); if it's 0, it indicates current encryption is the first encryption.

(6) GAP_EVT_SMP_TK_DISPALY

Event trigger condition: After Slave receives a Pairing_Req from Master, as per the pairing parameter configuration of the peer device and the local device, the method of TK (pincode) value to be used for pairing will be known. If the method "PK_Resp_Dsply_Init_Input" is enabled, which means Slave displays 6-digit pincode and Master inputs 6-digit pincode, this event will be triggered.

Data length "n": 4.

Pointer "p": p points to an u32-type variable "tk_set". The value is 6-digit pincode that Slave needs to inform the APP layer, and the APP layer needs to display the pincode.

The user can also manually set a user-specified pincode code such as "123456" without using the 6-digit pincode code generated by underlying layer randomly.

```
case GAP_EVT_SMP_TK_DISPALY:
{
    char pc[7];
    #if 1 //Set pincode manually
        u32 pinCode = 123456;
        memset(smp_param_own.paring_tk, 0, 16);
        memcpy(smp_param_own.paring_tk, &pinCode, 4);
    #else//Using the pincode generated by bottom layer randomly
        u32 pinCode = *(u32*)para;
    #endif
}
break;
```

User should get the 6-digit pincode from Slave and input the pincode on Master side (e.g. Mobile phone), to finish TK input and continue pairing process. If user has input wrong pincode, or has clicked "cancel", the pairing process fails.

The demo "vendor/b85m_feature/feature_smp_security/app.c" provides an example for Passkey Entry application.

(7) GAP_EVT_SMP_TK_REQUEST_PASSKEY

Event trigger condition: When the slave device enables the Passkey Entry mode and the PK_Init_Dsply_Resp_Input or PK_BOTH_INPUT pairing mode is used, this event will be triggered to notify the user that the TK value needs to be input. After receiving the event, the user needs to input the TK value through the IO input port (if the timeout is 30s, the pairing fails). The API for inputting the TK value: `btc_smp_setTK_by_PasskeyEntry` is explained in the "SMP parameter configuration" chapter.

Data length "n": 0.

Pointer "p": NULL.

(8) GAP_EVT_SMP_TK_REQUEST_OOB

Event trigger condition: When Slave device enables the OOB method of legacy pairing, this event will be triggered to inform user that 16-digit TK value should be input by the OOB method. After this event is received, user needs to input 16-digit TK value within 30s via IO input capability, otherwise pairing will fail

due to timeout. For the API "blc_smp_setTK_by_OOB" to input TK value, please refer to section 3.3.4.2 SMP parameter configuration.

Data length "n": 0.

Pointer "p": NULL.

(9) GAP_EVT_SMP_TK_NUMERIC_COMPARE

Event trigger condition: After Slave receives a Pairing_Req from Master, as per the pairing parameter configuration of the peer device and the local device, the method of TK (pincode) value to be used for pairing will be known. If the method "Numeric_Comparison" is enabled, this event will be triggered immediately. For "Numeric_Comparison", a method of SMP4.2 secure encryption, dialog window will pop up on both Master and Slave to show 6-digit pincode, "YES" and "NO"; user needs to check whether pincodes on the two sides are consistent, and decide whether to click "YES" to confirm TK check result is OK.

Data length "n": 4.

Pointer "p": p points to an u32-type variable "pinCode". The value is 6-digit pincode that Slave needs to inform the APP layer. The APP layer needs to display the pincode, and supplies "YES" or "NO" confirmation mechanism.

The demo "vendor/b85m_feature/feature_smp_security/app.c" provides an example for Numeric_Comparison application.

(10) GAP_EVT_ATT_EXCHANGE_MTU

Event trigger condition: This event will be triggered in either of the two cases below.

- Master sends "Exchange MTU Request", and Slave responds with "Exchange MTU Response".
- Slave sends "Exchange MTU Request", and Master responds with "Exchange MTU Response".

Data length "n": 6.

Pointer "p": p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u16 peer_MTU;
    u16 effective_MTU;
} gap_gatt_mtuSizeExchangeEvt_t;
```

connHandle: current connection handle.

peer_MTU: RX MTU value of the peer device.

effective_MTU = min(CleintRxMTU, ServerRxMTU). "CleintRxMTU" and "ServerRxMTU" indicate RX MTU size value of Client and Server respectively. After Master and Slave exchanges MTU size of each other, the minimum of the two values is used as the maximum MTU size value for mutual communication between them.

(11) GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM

Event trigger condition: Whenever the APP layer invokes the "blc_gatt_pushHandleValueIndicate" to send indicate data to Master, Master will respond with a confirmation for the data. This event will be triggered when Slave receives the confirmation.

Data length "n": 0.

Pointer "p": Null pointer.

Telink Semiconductor

4 Low Power Management

Low Power Management is also called Power Management, or PM as referred by this document.

4.1 Low Power Driver

4.1.1 Low Power Mode

For the 8x5x family, when MCU works in normal mode, or working mode, current is about 3~7mA. To save power consumption, MCU should enter low power mode.

There are three low power modes, or sleep modes: suspend mode, deepsleep mode, and deepsleep retention mode.

Table 4.1: Sleep mode description

Module	suspend	deepsleep retention	deepsleep
Sram	100% keep	first 32K(or 64K) keep, others lost	100% lost
digital register	99% keep	100% lost	100% lost
analog register	100% keep	99% lost	99% lost

The table above illustrates statistically data retention and loss for SRAM, digital registers and analog registers during each sleep mode.

(1) Suspend mode (sleep mode 1)

In this mode, program execution pauses, most hardware modules of MCU are powered off, and the PM module still works normally. In this mode, IC current is about 60~70uA. Program execution continues after wakeup from suspend mode.

In suspend mode, data of the SRAM and all analog registers are maintained. In order to reduce power consumption, the SDK has set the power-down mode for some modules when entering the suspend low-power processing, at which time the digital register of the module will also be powered down, and must be re-initialized and configured after waking up. Involving:

- A small number of digital registers in the baseband circuit. User should pay close attention to the registers configured by the API "rf_set_power_level_index". This API needs to be invoked after each wakeup from suspend mode.
- The digital register that controls the state of the Dfifo. Corresponding to the related APIs in drivers/8258(8278)/dfifo.h. When using these APIs, the user must ensure that they are reset after each suspend wake_up.

(2) Deepsleep mode (sleep mode 2)

In this mode, program execution pauses, vast majority of hardware modules are powered off, and the PM module still works. In this mode, IC current is less than 1uA, but if flash standby current comes up at 1uA or

so, total current may reach 1~2uA. After wakeup from deepsleep mode, similar to power on reset, MCU will restart, and program will reboot and re-initialize.

In deepsleep mode, except a few retention analog registers, data of all registers (analog & digital) and SRAM are lost.

(3) Deepsleep retention mode (sleep mode 3)

In deepsleep mode, the current is very low, but cannot store SRAM data; while in suspend mode, though SRAM and most registers are non-volatile, but the current is high.

The deepsleep with SRAM retention (deepsleep retention or deep retention) mode is designed in the 8x5x family, so as to achieve application scenes with low sleep current and quick wakeup to restore state, e.g. maintain BLE connection during long sleep. Corresponding to 16K or 32K SRAM retention area, deepsleep retention 16K Sram and deepsleep retention 32K Sram are introduced.

The deepsleep retention mode is also a kind of deepsleep. Most of the hardware modules of the MCU are powered off, and the PM hardware modules remain working. Power consumption is the power consumed by retention Sram plus that of deepsleep mode, and the current is between 2~3uA. When deepsleep mode wake up, the MCU will restart and the program will restart to initialize.

The deepsleep retention mode and deepsleep mode are consistent in register state, almost all of them are powered off. Compare with in deepsleep mode, in deepsleep retention mode, the first 32K (or the first 64K) of Sram can be kept without power-off, and the remaining Sram is powered off.

In deepsleep mode and deepsleep retention mode, there are very few analog registers that can be kept without power-down. These non-power-down analog registers include:

a) Analog registers to control GPIO pull-up/down resistance

When configured via the API "gpio_setup_up_down_resistor" or the following method in the app_config.h, GPIO pull-up/down resistance are non-volatile:

```
#define PULL_WAKEUP_SRC_PD5      PM_PIN_PULLDOWN_100K
```

Please refer to the introduction of the GPIO module. Using GPIO output belongs to the state controlled by the digital register. 8x5x can use GPIO output to control some peripherals during suspend, but after being switched to deepsleep retention mode, the GPIO output status becomes invalid and it cannot accurately control peripherals during sleep. At this point, you can use GPIO to simulate the state of the pull-up and pull-down resistors instead: pull-up 10K ohm instead of GPIO output high, and pull-down 100K ohm instead of GPIO output low.

b) Special retention analog registers of the PM module:

The code below shows the "DEEP_ANA_REG" in the "drivers/8258(8278)/pm.h".

```
#define PM_ANA_REG_POWER_ON_CLR_BUF1    0x3a // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF2    0x3b // initial value 0x00
#define PM_ANA_REG_POWER_ON_CLR_BUF3    0x3c // system used, user can not use
```

Please note that customers are not allowed to use ana_3c. This analog register is reserved for the underlying stack. If the application layer code uses this register, it needs to be modified to ana_3a, ana_3b. Because the

number of non-power-off analog registers is relatively small, it is recommended that customers use each of its bits to indicate different status bits. For details, please refer to “b85m_ble_remote” in the vendor directory of the SDK.

The following groups of non-drop analog registers may lose information due to wrong GPIO wakeup. For example, GPIO_PAD wakes up deepsleep at high level, but gpio is already at high level before calling `cpu_sleep_wakeup` function. It will cause wrong GPIO wakeup, then these analog register values will be lost.

```
#define DEEP_ANA_REG6    0x35
#define DEEP_ANA_REG7    0x36
#define DEEP_ANA_REG8    0x37
#define DEEP_ANA_REG9    0x38
#define DEEP_ANA_REG10   0x39
```

The user can save some important information in these analog register and read the previously stored values after deepsleep/deepsleep retention wake_up.

4.1.2 Low Power Wake-up Source

The low-power wake-up source diagram of B85m MCU is shown below, suspend/deepsleep/deepsleep retention can all be awakened by GPIO PAD and timer. In the BLE SDK, only two types of wake-up sources are concerned, as shown below (note that the two definitions of `PM_TIM_RECOVER_START` and `PM_TIM_RECOVER_END` in the code are not wake-up sources):

```
typedef enum {
    PM_WAKEUP_PAD    = BIT(3),
    PM_WAKEUP_TIMER = BIT(5),
}SleepWakeupSrc_TypeDef;
```

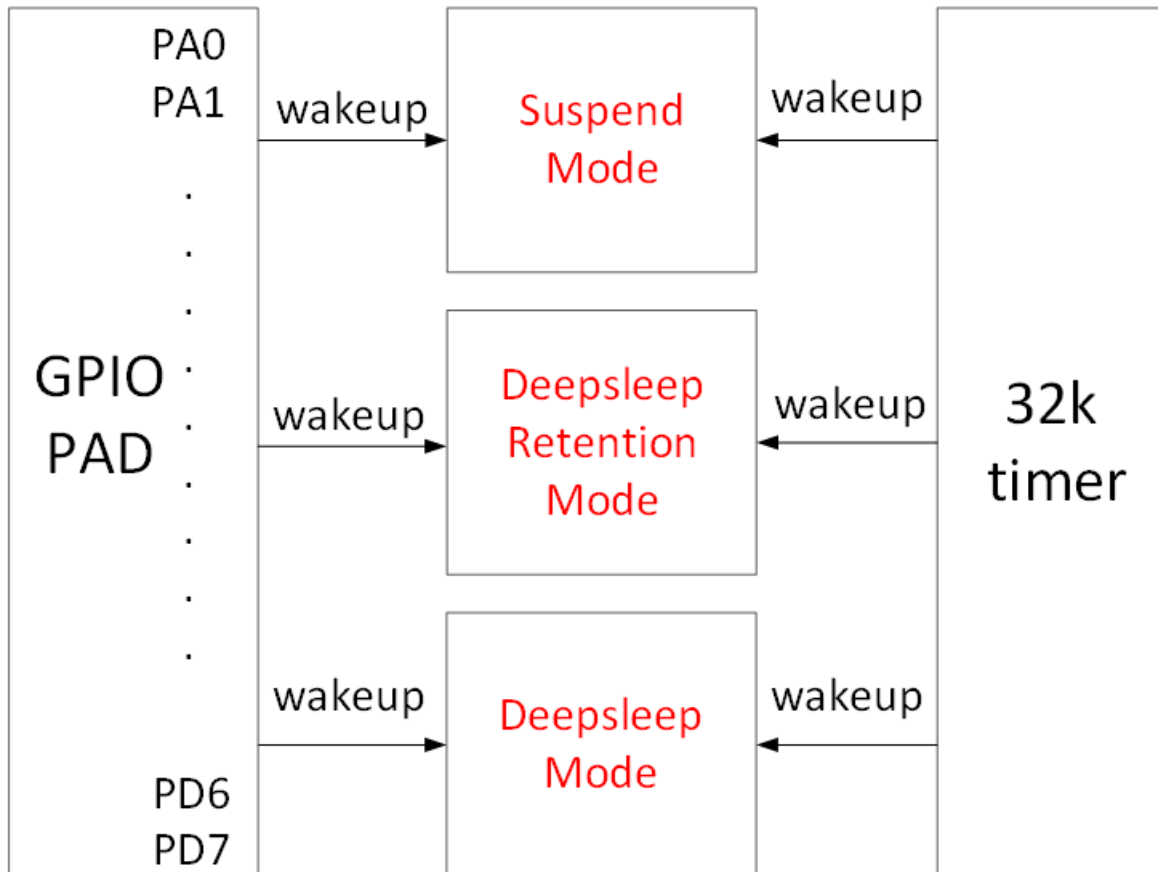


Figure 4.1: “B85 MCU HW Wakeup Source”

As shown above, there are two hardware wakeup sources: TIMER and GPIO PAD.

- The “PM_WAKEUP_TIMER” comes from 32k HW timer (32k RC timer or 32k Crystal timer). Since 32k timer is correctly initialized in the SDK, no configuration is needed except setting wakeup source in the “cpu_sleep_wakeup ()”.
- The “PM_WAKEUP_PAD” comes from GPIO module. Except 4 MSPI pins, all GPIOs (PAX/PBx/PCx/PDx) support high or low level wakeup .

The API below serves to configure GPIO PAD as wakeup source for sleep mode.

```
typedef enum{
    Level_Low=0,
    Level_High =1,
} GPIO_LevelTypeDef;
void cpu_set_gpio_wakeup (GPIO_PinTypeDef pin, GPIO_LevelTypeDef pol, int en);
```

- “pin”: GPIO pin
- “pol”: wakeup polarity, Level_High: high level wakeup, Level_Low: low level wakeup
- “en”: 1 indicates enable, 0 indicates disable.

Examples:

```
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 1); //Enable GPIO_PC2 PAD high level wakeup
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 0); //Disable GPIO_PC2 PAD wakeup
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 1); //Enable GPIO_PB5 PAD low level wakeup
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 0); //Disable GPIO_PB5 PAD wakeup
```

4.1.3 Sleep and Wake-up from Low Power Mode

The API below serves to configure MCU sleep and wakeup.

```
int cpu_sleep_wakeup (SleepMode_TypeDef sleep_mode, SleepWakeupSrc_TypeDef wakeup_src,
unsigned int wakeup_tick);
```

- sleep_mode: This parameter serves to set sleep mode as suspend mode, deepsleep mode, deepsleep retention 16K Sram or deepsleep retention 32K Sram.

```
typedef enum {
    SUSPEND_MODE                = 0,
    DEEPSLEEP_MODE              = 0x80,
    DEEPSLEEP_MODE_RET_SRAM_LOW16K = 0x43,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K = 0x07,
}SleepMode_TypeDef;
```

- wakeup_src: This parameter serves to set wakeup source for suspend/deep retention/deepsleep as one or combination of PM_WAKEUP_PAD and PM_WAKEUP_TIMER. If set as 0, MCU wakeup is disabled for sleep mode.
- "wakeup_tick": if PM_WAKEUP_TIMER is assigned as wakeup source, the "wakeup_tick" serves to set MCU wakeup time. If PM_WAKEUP_TIMER is not assigned, this parameter is negligible.

The "wakeup_tick" is an absolute value, which equals current value of System Timer tick plus intended sleep duration. When System Timer tick reaches the time defined by the wakeup_tick, MCU wakes up from sleep mode. The value of wakeup_tick needs to be based on the current System Timer tick value, plus an absolute time converted from the time to be slept, in order to effectively control the sleep time. If the wakeup_tick is set directly without taking into account the current System Timer tick, the wake-up time point cannot be controlled.

Since the wakeup_tick is an absolute time, it follows the max range limit of 32bit System Timer tick. In current SDK, 32bit max sleep time corresponds to 7/8 of max System Timer tick. Since max System Timer tick is 268s or so, max sleep time is $268 \times 7/8 = 234s$, which means the "delta_Tick" below should not exceed 234s. If a longer sleep time is needed, user can call the long sleep function, as described in section 4.2.7.

```
cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER, clock_time() + delta_tick);
```

The return value is an ensemble of current wakeup sources. Following shows wakeup source for each bit of the return value.

```
enum {
    WAKEUP_STATUS_TIMER    = BIT(1),
    WAKEUP_STATUS_PAD      = BIT(3),

    STATUS_GPIO_ERR_NO_ENTER_PM = BIT(7),
};
```

- a) If WAKEUP_STATUS_TIMER bit = 1, wakeup source is Timer.
- b) If WAKEUP_STATUS_PAD bit = 1, wakeup source is GPIO PAD.
- c) If both WAKEUP_STATUS_TIMER and WAKEUP_STATUS_PAD equal 1, wakeup source is Timer and GPIO PAD.
- d) STATUS_GPIO_ERR_NO_ENTER_PM is a special state indicating GPIO wakeup error. E.g. Suppose a GPIO is set as high level PAD wakeup (PM_WAKEUP_PAD). When MCU attempts to invoke the "cpu_sleep_wakeup" to enter suspend, and PM_WAKEUP_PAD wake-up source is set, MCU will fail to enter suspend and immediately exit the "cpu_sleep_wakeup" with return value STATUS_GPIO_ERR_NO_ENTER_PM.

Sleep time is typically set in the following way:

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER, clock_time() + delta_Tick);
```

The "delta_Tick", a relative time (e.g. 100* CLOCK_16M_SYS_TIMER_CLK_1MS), plus "clock_time()" becomes an absolute time.

Some examples on cpu_sleep_wakeup:

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_PAD, 0);
```

When it's invoked, MCU enters suspend, and wakeup source is GPIO PAD.

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER, clock_time() + 10*
↳ CLOCK_16M_SYS_TIMER_CLK_1MS;
```

When it's invoked, MCU enters suspend, wakeup source is timer, and wakeup time is current time plus 10ms, so the suspend duration is 10ms.

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_PAD | PM_WAKEUP_TIMER,
clock_time() + 50* CLOCK_16M_SYS_TIMER_CLK_1MS);
```

When it's invoked, MCU enters suspend, wakeup source includes timer and GPIO PAD, and timer wakeup time is current time plus 50ms.

If GPIO wakeup is triggered before 50ms expires, MCU will be woke up by GPIO PAD in advance; otherwise, MCU will be woke up by timer.

```
cpu_sleep_wakeup (DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);
```

When it's invoked, MCU enters deepsleep, and wakeup source is GPIO PAD.

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K , PM_WAKEUP_TIMER, clock_time() + 8*  
↪ CLOCK_16M_SYS_TIMER_CLK_1S);
```

When it's invoked, MCU enters deepsleep retention 32K Sram mode, wakeup source is timer, and wakeup time is current time plus 8s.

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K , PM_WAKEUP_PAD | PM_WAKEUP_TIMER, clock_time()  
↪ + 10* CLOCK_16M_SYS_TIMER_CLK_1S);
```

When it's invoked, MCU enters deepsleep retention 32K Sram mode, wakeup source includes GPIO PAD and Timer, and timer wakeup time is current time plus 10s. If GPIO wakeup is triggered before 10s expires, MCU will be woke up by GPIO PAD in advance; otherwise, MCU will be woke up by timer.

4.1.4 Low Power Wake-up Procedure

When user calls the API `cpu_sleep_wakeup`, the MCU enters the sleep mode; when the wake-up source triggers the MCU to wake up, the MCU software operation flow is inconsistent for different sleep modes.

The following is a detailed description of the MCU operating process after the suspend, deepsleep, and deepsleep retention three sleep modes are awakened. Please refer to the figure below.

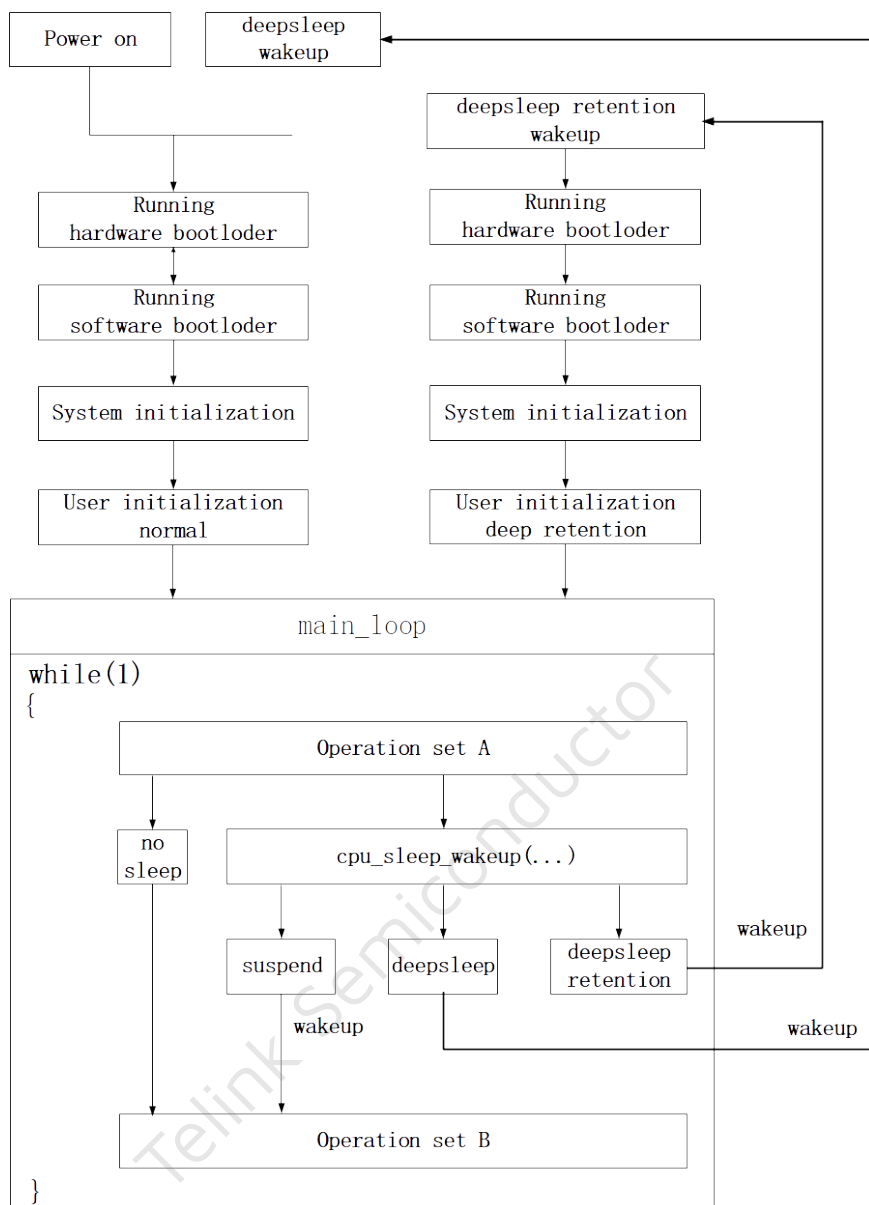


Figure 4.2: “Sleep Mode Wakeup Work Flow”

Detailed process after the MCU is powered on is introduced as following:

- (1) Run hardware bootloader

It is pure MCU hardware operation without involvement of software.

Couple of examples:

Read the boot flag of flash to determine whether the firmware that should be run currently is stored on flash address 0 or on flash address 0x20000 (related to OTA); read the value of the corresponding location of flash to determine how much data currently needs to be copied from flash to Sram as resident memory data (refer to Chapter (refer to the introduction of Sram allocation in Chapter 2). The part of running the hardware bootloader involves copying data from flash to sram, which generally takes a long time to execute. For example, it takes about 5ms to copy 10K data.

(2) Run software bootloader

After hardware bootloader, MCU starts "Running software bootloader". Software bootloader is vector side corresponding to the .s assembly program in the boot directory of the b85m sdk.

Software bootloader serves to set up memory environment for C program execution, so it can be regarded as memory initialization.

(3) System initialization

System initialization corresponds to the initialization of each hardware module (including `cpu_wakeup_init`, `rf_drv_init`, `gpio_init`, `clock_init`) from `cpu_wakeup_init` to `user_init` in the main function, and sets the digital/analog register status of each hardware module.

(4) User initialization

User initialization is divided into User initialization normal and User initialization deep retention, which correspond to the functions `user_init_normal` and `user_init_deepRetn` in the SDK respectively. For projects that do not use deep retention, such as kma master dongle, there is only `user_init`, which is equivalent to `user_init_normal`.

The `user_init` and `user_init_normal` require all the initialisation to be done, which takes longer time.

The `user_init_deepRetn` only needs to do some hardware related initialization, no software initialization is needed because the variables involved in software initialization are put into the MCU retention area during design time, and these variables remain unchanged during deep retention and do not need to be reset after waking up. So `user_init_deepRetn` is an accelerated mode, saving time and therefore power.

(5) main_loop

After User initialization, program enters `main_loop` inside `while(1)`. The operation is called "Operation Set A" before `main_loop` enters sleep mode, and called "Operation Set B" after wakeup from sleep.

As shown in figure above, sleep mode process is detailed as following:

(1) no sleep

Without sleep mode, MCU keeps looping inside `while(1)` between "Operation Set A" -> "Operation Set B".

(2) suspend

MCU enters suspend mode by invoking `cpu_sleep_wakeup`, wakes up from suspend after exiting from `cpu_sleep_wakeup`, and then executes "Operation Set B".

Suspend can be regarded as the most "clean" sleep mode, in which data of SRAM, digital and analog registers are retained (a few special exceptions). After wakeup from suspend, program continues from the breakpoint, with almost no need to recover SRAM or registers. However, in suspend current is relatively high.

(3) deepsleep

MCU can also enter deepsleep by invoking `cpu_sleep_wakeup`. After wakeup from deepsleep, MCU restarts from "Running hardware bootloader". Almost the same as power on reset, all hardware and software initialization are required after deepsleep wakeup. Since SRAM and registers - except a few retention analog registers - will lose their data in deepsleep, MCU current is decreased to less than 1uA.

(4) deepsleep retention

MCU can also enter deepsleep retention mode by invoking `cpu_sleep_wakeup`. After wakeup from deepsleep retention, MCU restarts from "Running software bootloader".

The deepsleep retention is an intermediate sleep mode between suspend and deepsleep. In suspend mode, both SRAM and most registers need to retain data, which thus ends up with higher current. In deepsleep retention, it's only needed to maintain states of a few retention analog registers, as well as data of first 16K or 32K SRAM, so current is largely decreased to 2uA or so.

After deepsleep wake_up, MCU needs to restart the whole flow. Since first 16K or 32K SRAM are non-volatile in deepsleep retention, there's no need to re-load from flash to SRAM after wake_up, and thus "Running hardware bootloader" is skipped. Due to limited SRAM retention size, "Running software bootloader" cannot be skipped. Since deepsleep retention does not keep register state, system initialization must also be executed to re-initialize registers.

The user initialization after deepsleep retention wake_up can actually be optimized to differentiate from MCU power on and deepsleep wake_up.

4.1.5 API `pm_is_MCU_deepRetentionWakeup`

According to the figure "sleep mode wakeup work flow" above, MCU power on, deepsleep wake_up and deepsleep retention wake_up all need to go through "Running software bootloader", "system initialization", and "user initialization".

While running system initialization and user initialization, user needs to know whether MCU is woke up from deepsleep retention, so as to differentiate from power on and deepsleep wake_up. The following API in the PM driver serves to make this judgement.

```
int pm_is_MCU_deepRetentionWakeup(void);
```

Return value: 1 indicates deepsleep retention wake_up; 0 indicates power on or deepsleep wake_up.

4.2 BLE Low Power Management

4.2.1 BLE PM Initialization

For applications with low power mode, BLE PM module needs to be initialized by following API.

```
void blc_ll_initPowerManagement_module(void);
```

If low power is not required, DO NOT use this API, so as to skip compiling of PM related code and variables into program and thus save firmware and SRAM space.

4.2.2 BLE PM for Link Layer

In this BLE SDK, PM module manages power consumption in BLE Link Layer. It would be helpful referring to introduction to Link Layer in earlier chapter.

Current SDK only applies low power management to Advertising state and Connection state Slave role with a set of APIs for user. It's not applicable yet to Scanning state, Initiating state and Connection state Master role.

The SDK does not apply low power management to Idle state either. In Idle state, since there is no RF activity, i.e. the "blt_sdk_main_loop" function is not valid, user can use PM driver for certain low power management. E.g. In the demo code below, when Link Layer is in Idle state, every main_loop would suspend for 10ms.

```
void main_loop (void)
{
    ////////////////////////////////////////////////// BLE entry ///////////////////////////////////
    blt_sdk_main_loop();

    ////////////////////////////////////////////////// UI entry ///////////////////////////////////
    // add user task

    ////////////////////////////////////////////////// PM configuration ///////////////////////////////////
    if(blc_ll_getCurrentState() == BLS_LINK_STATE_IDLE ){ //Idle state
        cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER,
clock_time() + 10*CLOCK_16M_SYS_TIMER_CLK_1MS);
    }
    else{
        blt_pm_proc(); //BLE Adv & Conn state
    }
}
```

The figure below shows timing of sleep mode when Link Layer is in Advertising state or Conn state Slave role with connection latency = 0.

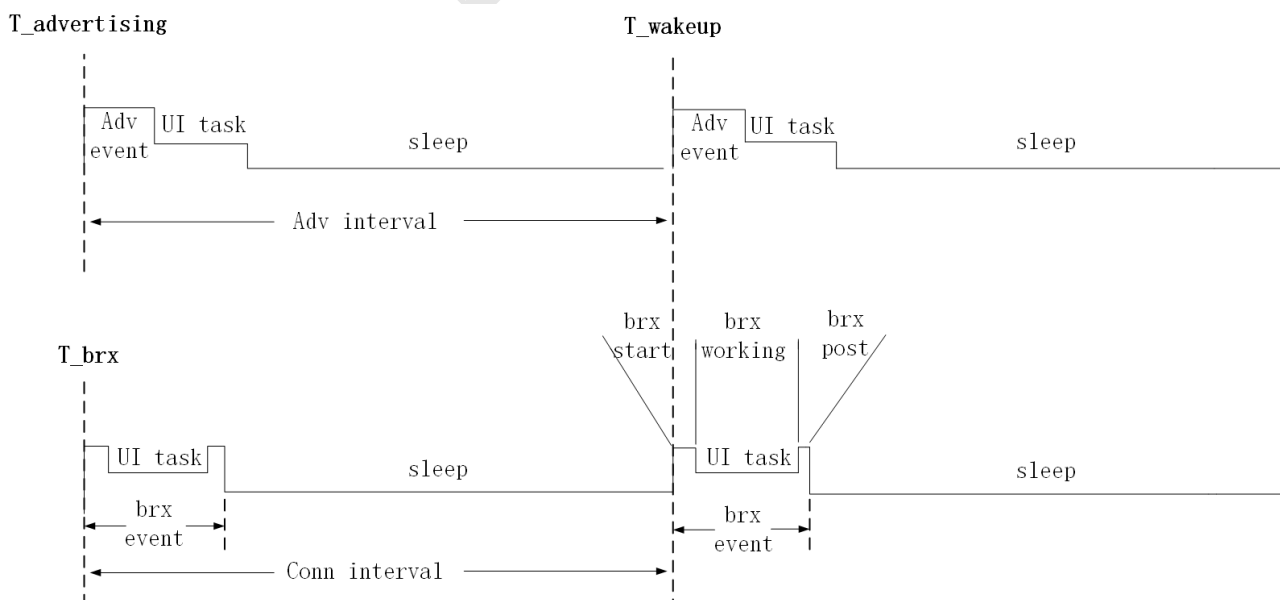


Figure 4.3: "Sleep Timing for Advertising State and Conn State Slave Role"

(1) In Advertising state, during each Adv Internal, Adv Event is mandatory; MCU can enter sleep mode

(suspend/deepsleep retention) during the rest time other than UI task.

In figure above, the starting time of Adv event at first Adv interval is defined as $T_{\text{advertising}}$, and the time for MCU to wake up from sleep is defined as T_{wakeUp} . T_{wakeUp} is also the start of Adv event at next Adv interval. Both these two parameters will be elaborated in later section.

- (2) During each Conn-interval at Conn state Slave role, the time for brx Event (brx start+brx working+brx post) is mandatory. MCU can enter sleep mode (suspend/ deepsleep retention) during the rest time other than UI task.

The starting time of of Brx event at first Connection interval is defined as T_{brx} , and the time for MCU to wake up from sleep is T_{wakeUp} . T_{wakeUp} is also the start of BRx event at next Connection interval. Both these two parameters will be elaborated in later section.

BLE PM is basically the sleep mode management in Advertising state or Conn state Slave role. User can select sleep mode and set related time parameters: enter sleep, enter suspend mode, or enter deepsleep retention mode.

As explained earlier, the 8x5x family has 3 sleep modes: suspend, deepsleep, and deepsleep retention.

For suspend and deepsleep retention, since the `blt_sdk_main_loop` of the SDK includes low PM in BLE stack according to Link Layer state, to configure low power management, user only needs to invoke corresponding APIs instead of the "`cpu_sleep_wakeup`".

Deepsleep is not included in BLE low PM, so user needs to manually invoke the API "`cpu_sleep_wakeup`" in APP layer to enter deepsleep. Please refer to the "`blt_pm_proc`" function in the project "`b85m_ble_remote`" of the SDK.

Following sections illustrate details of low power management in Advertising state and Connection state Slave role.

4.2.3 BLE PM Variables

The variables in this section are helpful to understand BLE PM software flow.

The struct "`st_ll_pm_t`" is defined in BLE SDK. Following lists some variables of the struct which will be used by PM APIs.

```
typedef struct {
    u8      suspend_mask;
    u8      wakeup_src;
    u16     sys_latency;
    u16     user_latency;
    u32     deepRet_advThresTick;
    u32     deepRet_connThresTick;
    u32     deepRet_earlyWakeupTick;
}st_ll_pm_t;
```

Following struct is defined in the file "`ll_pm.c`" for understanding purpose.

```
st_ll_pm_t bltPm;
```

Please note that this file is assembled in library, and user is not allowed to make any operation on this struct variable.

There will be a lot of variables like the "bltPm.suspend_mask" in later sections.

4.2.4 API bls_pm_setSuspendMask

The APIs below serve to configure low power management in Link Layer at "Advertising state" and "Conn state Slave role".

```
void    bls_pm_setSuspendMask (u8 mask);
u8      bls_pm_getSuspendMask (void);
```

The "bltPm.suspend_mask" is set by the "bls_pm_setSuspendMask" and its default value is SUSPEND_DISABLE.

Following shows source code of the 2 APIs.

```
void bls_pm_setSuspendMask (u8 mask)
{
    bltPm.suspend_mask = mask;
}
u8 bls_pm_getSuspendMask (void)
{
    return bltPm.suspend_mask;
}
```

The "bltPm.suspend_mask" can be set as any one or the "or-operation" of following values:

```
#define    SUSPEND_DISABLE    0
#define    SUSPEND_ADV        BIT(0)
#define    SUSPEND_CONN       BIT(1)
#define    DEEPSLEEP_RETENTION_ADV    BIT(2)
#define    DEEPSLEEP_RETENTION_CONN    BIT(3)
```

The "SUSPEND_DISABLE" means sleep is disabled which allows MCU to enter neither suspend nor deepsleep retention.

The "SUSPEND_ADV" and "DEEPSLEEP_RETENTION_ADV" decide whether MCU at Advertising state can enter suspend and deepsleep retention.

The "SUSPEND_CONN" and "DEEPSLEEP_RETENTION_CONN" decide whether MCU at Conn state Slave role can enter suspend and deepsleep retention.

In low power sleep mode design of the SDK, deepsleep retention is a substitute of suspend mode to reduce sleep power consumption.

Take Conn state slave role as an example:

The SDK will first check whether `SUSPEND_CONN` is enabled in the `"bltPm.suspend_mask"`, and MCU can enter suspend only when `SUSPEND_CONN` is enabled. Further on, based on the value of the `DEEPSLEEP_RETENTION_CONN`, MCU can decide whether it will enter suspend mode or deepsleep retention mode.

Therefore, to enable MCU to enter suspend, user only needs to enable `SUSPEND_ADV/SUSPEND_CONN`. To enable MCU to enter deepsleep retention mode, both `SUSPEND_CONN` and `DEEPSLEEP_RETENTION_CONN` should be enabled.

Following shows 3 typical use cases:

```
bls_pm_setSuspendMask(SUSPEND_DISABLE);
```

MCU will not enter sleep mode (suspend/deepsleep retention).

```
bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);
```

At Advertising state and Conn state Slave role, MCU can only enter suspend mode, and it's not allowed to enter deepsleep retention.

```
bls_pm_setSuspendMask(SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV  
                      | SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);
```

At Advertising state and Conn state Slave role, MCU can enter both suspend and deepsleep retention, but the sleep mode to enter depends on sleeping time which will be explained later.

There may be some special applications, for example:

```
bls_pm_setSuspendMask(SUSPEND_ADV)
```

Only at Advertising state can MCU enter suspend, and at Conn state Slave role it's not allowed to enter sleep mode.

```
bls_pm_setSuspendMask(SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN)
```

Only at Conn state Slave role, can MCU enter suspend or deepsleep retention, and at Advertising state it's not allowed to enter sleep mode.

4.2.5 API `bls_pm_setWakeupSource`

User can set the `bls_pm_setSuspendMask` to enable MCU to enter sleep mode (suspend or deepsleep retention), and use the following API to set wakeup source.

```
void      bls_pm_setWakeupSource(u8 source);
```

“source”: Wakeup source, can be set as PM_WAKEUP_PAD.

This API sets the bottom-layer variable “bltPm.wakeup_src”. Following shows source code in the SDK.

```
void      bls_pm_setWakeupSource (u8 src)
{
    bltPm.wakeup_src = src;
}
```

When MCU enters sleep mode at Advertising state or Conn state Slave role, its actual wakeup source is:

```
bltPm.wakeup_src | PM_WAKEUP_TIMER
```

So PM_WAKEUP_TIMER is mandatory, not depending on user setup. This guarantees that MCU will wake up at specified time to handle Adv Event or Brx Event.

Everytime wakeup source is set by the “bls_pm_setWakeupSource”, after MCU wakes up from sleep mode, the bltPm.wakeup_src is set to 0.

4.2.6 API blc_pm_setDeepsleepRetentionType

Deepsleep retention further separates into 16K SRAM retention or 32K SRAM retention. When entering deepsleep retention mode, the following API can be set to decide which sub-mode to enter:

```
void blc_pm_setDeepsleepRetentionType(SleepMode_TypeDef sleep_type);
```

Only two options are available:

```
typedef enum {
    DEEPSLEEP_MODE_RET_SRAM_LOW16K    = 0x43,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K    = 0x07,
}SleepMode_TypeDef;
```

In the SDK, default deepsleep retention mode is set as DEEPSLEEP_MODE_RET_SRAM_LOW16K, and to use 32K retention mode, user needs to invoke the API below during initialization.

Please note that this API must be invoked after the “blc_ll_initPowerManagement_module” to take effect.

```
blc_pm_setDeepsleepRetentionType(DEEPSLEEP_MODE_RET_SRAM_LOW32K);
```

Refer to Chapter 2, Sram memory allocation is designed according to deepsleep retention 16K Sram by default. According to the description in Chapter 1, we know that we need to select different software

bootloader boot files and boot. link for different ICs and deep retention size values. Please refer to the section “Software bootloader” for the specific mapping relationship and modification setting method.

If the current IC is 8258, using deepsleep retention 32K Sram requires the following two steps to modify:

Step 1 Select the software bootloader file as cstartup_8258_RET_32K.S;

Step 2 Modify the boot.link file: replace the content of the SDK/boot/boot_32k_retn_8253_8258.link file with the boot.link file in the SDK root directory.

The settings of other ICs are similar to the above, and users can modify them according to the actual situation.

4.2.7 PM software processing flow

Both actual code and pseudo-code are used herein to explain the flow details.

4.2.7.1 blt_sdk_main_loop

As shown below, the “blt_sdk_main_loop” is repetitively executed in while (1) loop of the SDK.

```
while(1)
{
    ////////////////////////////////////////////////// BLE entry ///////////////////////////////////
    blt_sdk_main_loop();
    ////////////////////////////////////////////////// UI entry ///////////////////////////////////
    //UI task
    ////////////////////////////////////////////////// user PM config ///////////////////////////////////
    //blt_pm_proc();
}
```

The blt_sdk_main_loop function is executed continuously in while(1), and the code for BLE low-power management is in the blt_sdk_main_loop function, so the code for low-power management is also executed all the time.

Following shows the implementation of BLE PM logic inside the “blt_sdk_main_loop”.

```
int blt_sdk_main_loop (void)
{
    .....
    if(bltPm. suspend_mask == SUSPEND_DISABLE) // SUSPEND_DISABLE, can not
    {
        // enter sleep mode
        return 0;
    }

    if( (Link Layer State == Advertising state) || (Link Layer State == Conn state Slave role)
    {
        ↵ )
    }
```

```

        if(Link Layer is in Adv Event or Brx Event) //RF is working, can not enter
        {
            //sleep mode
            return 0;
        }
        else
        {
            blt_brx_sleep (); //process sleep & wakeup
        }
    }
    return 0;
}

```

- (1) When the "bltPm.suspend_mask" is SUSPEND_DISABLE, the SW directly exits without executing the "blt_brx_sleep" function. So when using the "bls_pm_setSuspendMask(SUSPEND_DISABLE)", PM logic is completely ineffective; MCU will never enter sleep and the SW always execute while(1) loop.
- (2) When the SW is executing Adv Event at Advertising State or Brx Event at Conn state Slave role, the "blt_brx_sleep" won't be executed either due to RF task ongoing. The SDK needs to guarantee completion of Adv Event/Brx Event before MCU enters sleep mode.

Only when both cases above are invalid, the blt_brx_sleep will be executed.

4.2.7.2 blt_brx_sleep

Following shows logic implementation of the "blt_brx_sleep" function in the case of default deepsleep retention 16K Sram.

```

void    blt_brx_sleep (void)
{
    if( (Link Layer state == Adv state)&& (bltPm. suspend_mask &SUSPEND_ADV) )
    { //current state is adv state, suspend is allowed
        T_wakeup = T_advertising + advInterval;
        " BLT_EV_FLAG_SUSPEND_ENTER" event callback execution
        T_sleep = T_wakeup - clock_time();
        if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_ADV &&
            T_sleep > bltPm.deepRet_advThresTick )
        { //suspend is automatically switched to deepsleep retention
            cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,
                PM_WAKEUP_TIMER | bltPm.wakeup_src,T_wakeup); //suspend
            //MCU reset to 0 after wakeup, restart on "software bootloader"
        }
        else
        {
            cpu_sleep_wakeup ( SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
        }
        " BLT_EV_FLAG_SUSPEND_EXIT " event callback execution
    }
}

```

```

        if(suspend is woke up by GPIO PAD)
        {
            " BLT_EV_FLAG_GPIO_EARLY_WAKEUP" event callback execution
        }
    }
    else if((Link Layer state == Conn state Slave role)&& (SuspendMask&SUSPEND_CONN) )
{
    //current Conn state, enter suspend
    if(conn_latency != 0)
    {
        latency_use = bls_calculateLatency();
        T_wakeup = T_brx + (latency_use +1) * conn_interval;
    }
    else
    {
        T_wakeup = T_brx + conn_interval;
    }
    " BLT_EV_FLAG_SUSPEND_ENTER" event callback execution
    T_sleep = T_wakeup - clock_time();
    if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_CONN &&
    T_sleep > bltPm.deepRet_connThresTick )
    { //suspend is automatically switched to deepsleep retention
        cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,
        PM_WAKEUP_TIMER | bltPm.wakeup_src,T_wakeup); //suspend
        //MCU reset to 0 after wakeup, restart on "software bootloader"
    }
    else
    {
        cpu_sleep_wakeup ( SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
    }

    " BLT_EV_FLAG_SUSPEND_EXIT" event callback execution
    if(suspend is waken up by GPIO PAD)
    {
        " BLT_EV_FLAG_GPIO_EARLY_WAKEUP" event callback execution
        Adjust BLE timing
    }
}

    bltPm.wakeup_src = 0;
    bltPm.user_latency = 0xFFFF;
}

```

To simplify the discussion, let's begin with an easy case: conn_latency =0, only suspend mode, no deepsleep retention. This is the case when setting suspend mask in APP layer via the "bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN)".

Referring to controller event introduced earlier, please pay close attention to the timing of these suspend related events and callback functions: BLT_EV_FLAG_SUSPEND_ENTER, BLT_EV_FLAG_SUSPEND_EXIT,

BLT_EV_FLAG_GPIO_EARLY_WAKEUP.

When Link Layer is in Advertising state with "bltPm. suspend_maskis" set to SUSPEND_ADV, or at Conn state slave role with "bltPm. suspend_mask" set to SUSPEND_CONN, MCU can enter suspend mode.

In suspend mode, the API "cpu_sleep_wakeup" in the driver is finally invoked.

```
cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER | bltPm.wakeup_src, T_wakeup);
```

This API sets wakeup source as PM_WAKEUP_TIMER | bltPm.wakeup_src, so Timer wakeup is mandatory to guarantee MCU wakeup before next Adv Event or Brx Event. For wakeup time "T_wakeup", please refer to earlier "sleep timing for Advertising state & Conn state Slave role" diagram.

When exiting the "blt_brx_sleep" function, both the "bltPm.wakeup_src" and the "bltPm.user_latency" are reset. So the API "bls_pm_setWakeupSource" and "bls_pm_setManualLatency" are only effective for current sleep mode.

4.2.8 Analysis of deepsleep retention

Introduce deepsleep retention, and continue to analyze the above software processing flow. When the application layer is set as follows, deepsleep retention mode is enabled.

```
bls_pm_setSuspendMask( SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV | SUSPEND_CONN |
↪ DEEPSLEEP_RETENTION_CONN);
```

4.2.8.1 API blc_pm_setDeepsleepRetentionThreshold

At Advertising state and Conn state slave role, suspend can switch to deep retention only when following conditions are met, respectively:

```
if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_ADV &&T_sleep > bltPm.deepRet_advThresTick )
if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_CONN &&T_sleep > bltPm.deepRet_connThresTick )
```

Firstly, the "bltPm. suspend_mask" should be set to DEEPSLEEP_RETENTION_ADV or DEEPSLEEP_RETENTION_CONN, as explained before.

Secondly, for T_sleep > bltPm.deepRet_advThresTick or T_sleep > bltPm.deepRet_connThresTick, T_sleep, sleep duration time, equals Wakeup time "T_wakeup" minus current time "clock_time()". It means that sleep duration should exceed certain threshold so that MCU can switch sleep mode from suspend to deepsleep retention.

Here is the API to set the two threshold in unit of ms for Advertising state and Conn state slave role.

```
void blc_pm_setDeepsleepRetentionThreshold( u32 adv_thres_ms,
u32 conn_thres_ms)
{
    bltPm.deepRet_advThresTick = adv_thres_ms * CLOCK_16M_SYS_TIMER_CLK_1MS;
```

```
bltPm.deepRet_connThresTick = conn_thres_ms * CLOCK_16M_SYS_TIMER_CLK_1MS;  
}
```

API `blc_pm_setDeepsleepRetentionThreshold` is used to set the time threshold when suspend is switched to deepsleep retention trigger condition. This design is to pursue lower power consumption.

Refer to the description of the "Run Process After Sleep Wake_up" section above. After suspend mode wake_up, you can immediately return to the environment before suspend to continue running. In the above software flow, after `T_wakeup` wakes up, it can immediately start executing the Adv Event/Brx Event task.

After deepsleep retention wake_up, you need to return to the place where "Run software bootloader" started. Compared with suspend wake_up, you need to run 3 more steps (Run software bootloader + System initialization + User initialization) before you can return to `main_loop` to execute Adv Event again. / Brx Event task.

Taking Conn state slave role as an example, the following figure shows the timing (sequence) & power (power consumption) comparison when sleep mode is suspend and deepsleepretention respectively.

The time difference `T_cycle` between two adjacent Brx events is the current time period. Average the power consumption of Brx Event, the equivalent current is `I_brx`, and the duration is `t_brx` (the name `t_brx` here is to distinguish it from the previous concept `T_brx`). The bottom current of Suspend is `I_suspend`, and the bottom current of deep retention is `I_deepRet`.

The average current in the process of "Run software bootloader + System initialization + User initialization" is equivalent to `I_init`, and the total duration is `T_init`. In actual applications, the value of `T_init` needs to be controlled and measured by the user, and how to implement it will be introduced later.

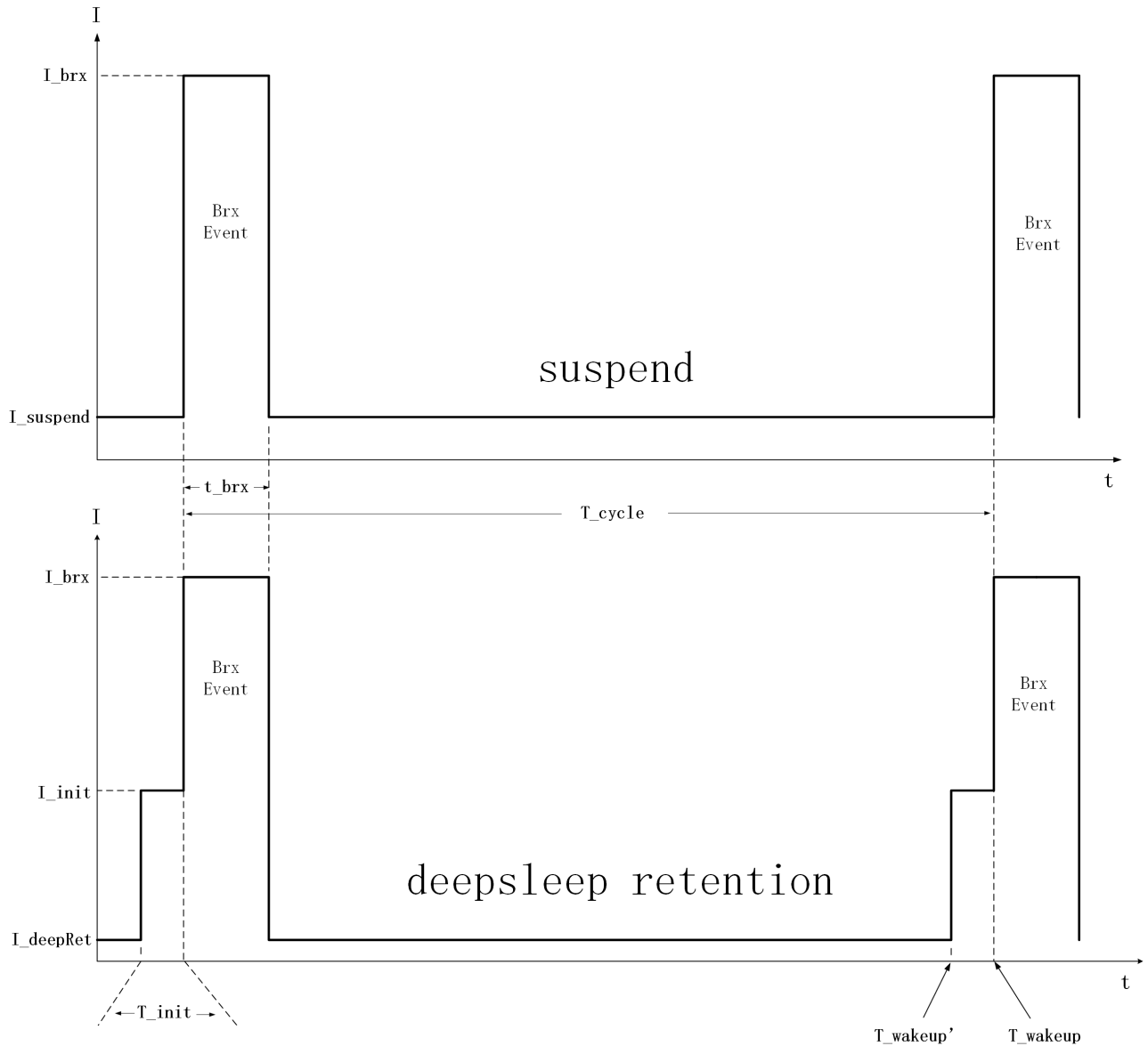


Figure 4.4: "Suspend Deep sleep Retention Timing Power"

The following is the description of terms in the figure.

- T_{cycle} : the time difference between two adjacent Brx events
- I_{brx} : average the power consumption of Brx Event, the equivalent current is I_{brx}
- t_{brx} : I_{brx} duration
- $I_{suspend}$: suspend bottom current
- $I_{deepRet}$: bottom current of deep retention
- I_{init} : Software bootloader + System initialization + User initialization process equivalent average current
- T_{init} : the total duration of I_{init}

Average Brx current with suspend mode is:

$$I_{avgSuspend} = I_{brx} \cdot t_{brx} + I_{suspend} \cdot (T_{cycle} - t_{brx})$$

Simplified by $T_{cycle} \gg t_{brx}$, $(T_{cycle} - t_{brx})$ can be regarded as T_{cycle} .

$$I_{avgSuspend} = I_{brx} \cdot t_{brx} + I_{suspend} \cdot T_{cycle}$$

Average Brx current with deepsleep retention mode is:

$$\begin{aligned} I_{avgDeepRet} &= I_{brx} \cdot t_{brx} + I_{init} \cdot T_{init} + I_{deepRet} \cdot (T_{cycle} - t_{brx}) \\ &= I_{brx} \cdot t_{brx} + I_{init} \cdot T_{init} + I_{deepRet} \cdot T_{cycle} \end{aligned}$$

Comparing $I_{avgSuspend}$ and $I_{avgDeepRet}$, removing the same " $I_{brx} \cdot t_{brx}$ ", the final part of the comparison is

$$\begin{aligned} I_{avgSuspend} - I_{avgDeepRet} &= I_{suspend} \cdot T_{cycle} - I_{init} \cdot T_{init} - I_{deepRet} \cdot T_{cycle} \\ &= T_{cycle} \cdot (I_{suspend} - I_{deepRet}) - (I_{init} \cdot T_{init}) \end{aligned}$$

For application program with correct power debugging on both HW circuit and SW, the " $(I_{suspend} - I_{deepRet})$ " and " $(I_{init} \cdot T_{init})$ " can be regarded as fixed value.

Suppose $I_{suspend}=30\mu A$, $I_{deepRet}=2\mu A$, $(I_{suspend} - I_{deepRet}) = 28\mu A$; $I_{init}=3mA$, $T_{init}=400\mu s$, $(I_{init} \cdot T_{init})=1200\mu A \cdot \mu s$:

$$I_{avgSuspend} - I_{avgDeepRet} = T_{cycle} \cdot (28 - 1200/T_{cycle})$$

$$I_{avgSuspend} - I_{avgDeepRet}$$

>0 when $T_{cycle} > (1200/28) = 43ms$, DeepRet consumes less power;

<0 when $T_{cycle} < 43ms$, Suspend mode consumes less power.

Mathematically, when $T_{cycle} < 43ms$, suspend mode is more power efficient; when $T_{cycle} > 43ms$, deepsleep retention mode is a better choice.

Note:

- As you can see in the PM software processing flow section, the suspend is automatically switched to deepsleep retention only when $T_{sleep} > 43ms$. We generally consider the MCU working time (Brx Event + UI task) to be relatively short, and when T_{cycle} is large, we can consider T_{sleep} to be approximately equal to T_{cycle} .

By using the threshold setting API below, MCU will automatically switch suspend to deepsleep retention for T_{sleep} more than 43mS, and maintain suspend for T_{sleep} less than 43mS.

```
blc_pm_setDeepsleepRetentionThreshold(43, 43);
```

Take a long connection of 10ms connection interval * (99 + 1) = 1s as an example:

During the Conn state slave role, due to the tasks of the application layer, manual latency settings, etc., it may lead to time values such as 10ms, 20ms, 50ms, 100ms, 1s, etc. when the MCU suspend. According to the 43ms threshold setting, the MCU will automatically switch the 50ms, 100ms, 1s etc. suspend to

deepsleep retention, while the 10ms, 20ms etc. suspend will still maintain suspend, such processing can ensure an optimal power consumption.

Since the power consumption of deepsleep retention is lower than that of suspend, and the presence of the 3 steps "Run software bootloader + System initialization + User initialization" results in some additional power consumption. Based on the above analysis, it must be the case that deepsleep retention will be more power efficient when T_{cycle} is greater than a certain threshold. The values in the above example are just a simple demo, the user needs to measure the corresponding values in the above equation according to certain methods when implementing power optimisation, and only then can the threshold value be determined.

In practice, following demos in the SDK, as long as user initialization does not incorrectly run across extended time, for T_{cycle} larger than 100ms, deepsleep retention mode should end up with lower power in most application scenarios.

4.2.8.2 blc_pm_setDeepsleepRetentionEarlyWakeupTiming

According to the "suspend & deepsleep retention timing & power", suspend wake_up time " T_{wakeup} " is exactly the starting point of next Brx Event, or the time point when BLE master starts sending packet.

For deepsleep retention, wake_up time needs to start earlier than T_{wakeup} to allow T_{init} : running software bootloader + system initialization + user initialization, or it will miss Brx Event, i.e., the time when BLE master starts sending packet. So MCU wake_up time should be pulled in to T_{wakeup} :

$$T_{wakeup}' = T_{wakeup} - T_{init}$$

When applying to:

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K, PM_WAKEUP_TIMER | bltPm.wakeup_src,
T_wakeup - bltPm.deepRet_earlyWakeupTick);
```

The T_{wakeup} is automatically calculated by the BLE stack, while the " $bltPm.deepRet_earlyWakeupTick$ " can be assigned to the measured T_{init} (or slightly larger) by following API:

```
void blc_pm_setDeepsleepRetentionEarlyWakeupTiming(u32 earlyWakeup_us)
{
    bltPm.deepRet_earlyWakeupTick = earlyWakeup_us *    CLOCK_16M_SYS_TIMER_CLK_1US;
}
```

User can directly set the measured value of T_{init} to the above API, or set a value slightly larger than T_{init} , but not less than this value.

4.2.8.3 Optimization and measurement of T_{init}

For SRAM concept to be discussed in this section such as ram_code, retention_data, deepsleep retention area, please refer to section 2.1.2 SRAM space partition.

(1) T_{init} timing

From the figure “suspend & deepsleep retention timing & power”, combined with the previous analysis, we can see that for the larger T_{cycle} , the sleep mode uses deepsleep retention with lower power consumption, but in this mode the T_{init} time is mandatory. In order to minimize the power consumption of long sleep, the time of T_{init} needs to be optimized to the minimum. The value of I_{init} is basically stable and does not need to be optimized.

The T_{init} is the sum of the time consumed by the 3 steps of Run software bootloader + System initialization + User initialization. The 3 steps are disassembled and analyzed, and the time of each step is defined first.

- T_{cstartup} is the time of running software bootloader, i.e. executing assembly file `cstartup_xxx.S`.
- T_{sysInit} is system initialization time.
- T_{userInit} is user initialization time.

$$T_{\text{init}} = T_{\text{cstartup}} + T_{\text{sysInit}} + T_{\text{userInit}}$$

Following is a complete timing diagram of T_{init} :

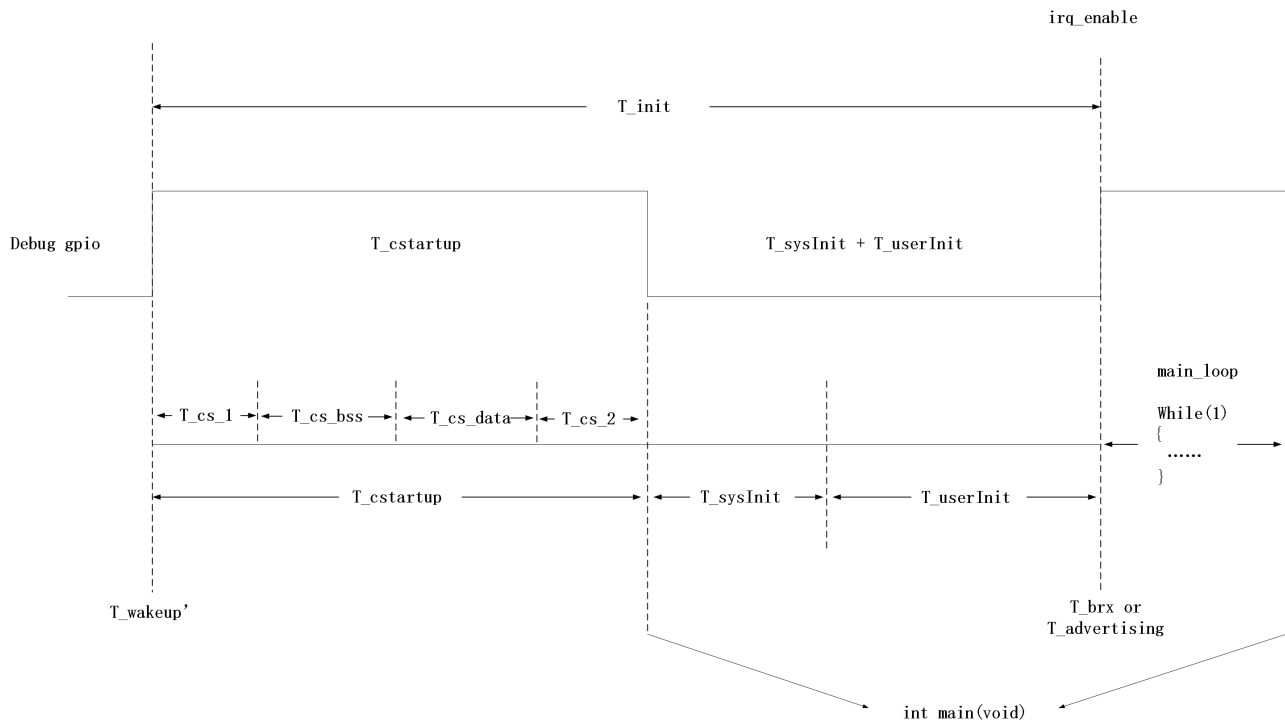


Figure 4.5: “ T_{init} Timing”

Based on earlier definition, T_{wakeup} is the starting point of next Adv Event/Brx Event, and $T_{\text{wakeup'}}$ is MCU early wakeuptime.

After wake_up, MCU will execute `cstartup_xxx.S`, jump to `main()` to start system initialization followed by user initialization, and then enter `main_loop`. Once getting in `main_loop`, it can start processing of Adv Event/Brx Event. The end of T_{userInit} is the starting point of Adv Event/Brx Event, or $T_{\text{brx}}/T_{\text{advertising}}$ as shown in above diagram. “irq_enable” in the diagram is the separation between T_{userInit} and `main_loop`, matching the code in the SDK.

In the SDK, T_{sysInit} includes execution time of `cpu_wakeup_init`, `rf_drv_init`, `gpio_init` and `clock_init`. These timing parameters have been optimized in the SDK by placing the associated code into the `ram_code`.

The T_cstatus and T_userInit in the SDK are elaborated herein.

(2) T_userInit

User initialization is executed at power on, deepsleep wake_up, and deepsleep retention wake_up.

For applications without deepsleep retention mode, user initialization does not need to differentiate between deepsleep retention wake_up and power on/ deepsleep wake_up. In the BLE SDK, all user initialization can be completed with the following functions. The same applies to the "b85m_master_kma_dongle" project in the BLE SDK.

```
void user_init(void);
```

For applications with deepsleep retention mode, to reduce power, T_userInit needs to be as short as possible as explained earlier, so deepsleep retention wake_up would be different from power on / deepsleep wakeup.

The initialization tasks in the user_init falls into 2 categories: initialization of hardware registers, and initialization of logic variables in SRAM.

Since in deepsleep retention mode first 16K or 32K SRAM is non-volatile, logic variables can be defined as retention_data to save time for initialization. Since registers cannot retain data across deepsleep retention, re-initialization is required for registers.

In summary, for deepsleep retention wake_up, user_init_deepRetn applies; while for power on and deepsleep wake_up, user_init_normal function applies, as shown in following code:

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();
if( deepRetWakeUp ){
    user_init_deepRetn ();
}
else{
    user_init_normal ();
}
```

The user can compare the implementation of these two functions. The following is the implementation of the user_init_deepRetn function in the SDK demo "b85m_ble_remote".

```
_attribute_ram_code_ void user_init_deepRetn(void)
{
    #if (PM_DEEPSLEEP_RETENTION_ENABLE)
        blc_app_loadCustomizedParameters();
        blc_ll_initBasicMCU();    //mandatory
        rf_set_power_level_index (MY_RF_POWER_INDEX);
        blc_ll_recoverDeepRetention();
        app_ui_init_deepRetn();
    #endif
}
```

First 3 lines (from code `blc_app_loadCustomizedParameters` to `rf_set_power_level_index`) are mandatory BLE initialization of hardware registers.

The `blc_ll_recoverDeepRetention()` is to recover software and hardware state at Link Layer by low level stack.

User is not recommended to modify these lines.

Finally, `app_ui_init_deepRetn` is the user's re-initialization of the hardware registers used by the application layer. The GPIO wakeup configuration and LED state setting in the demo "b85m_ble_remote" are hardware initialization. The UART hardware register state in the demo "b85m_module" needs to re-initialize.

On top of SDK demo, if additional items are added to user initialization, following judgement is recommended:

- If it is SRAM variable, put it to the "retention_data" section by adding the keyword "attribute_data_retention", so as to save re-initialization time after deepsleep retention wake_up. Then it can be run at `user_init_normal` function.
- If it is hardware register, it should be placed inside `user_init_deepRetn` function to ensure the correct hardware status.

With above implementation, after deepsleep retention wake_up, `T_userInit` is execution time of `user_init_deepRetn`. The SDK also tries to place these functions inside `ram_code` to save time. If deepsleep retention area allows, user should place added register initialization functions inside `ram_code` as well.

(3) `T_userInit` Optimization for Conn state slave role

TBD

(4) `T_cstartup`

`T_cstartup` is the execution time of `cstartup_xxx.S`, e.g. `cstartup_8258_RET_16K.S` in the SDK. Please refer to the boot file in the SDK.

`T_cstartup` has 4 components, in time sequence:

$$T_{cstartup} = T_{cs_1} + T_{cs_bss} + T_{cs_data} + T_{cs_2}$$

`T_cs_1` and `T_cs_2` are fixed timing which user is not allowed to modify.

The `T_cs_data` is initialization of "data" sector in SRAM. The "data" is already initialized global variables with initial values stored in "data initial value" sector of flash. Therefore, `T_cs_data` is the time transferring "data" from flash "data initial value" sector to SRAM "data" sector. Corresponding assembly code is:

```

tloadr    r1, DATA_I
tloadr    r2, DATA_I+4
tloadr    r3, DATA_I+8
COPY_DATA:
tcmp      r2, r3
tjge      COPY_DATA_END
tloadr    r0, [r1, #0]
tstorer   r0, [r2, #0]
tadd      r1, #4

```



```

tadd      r2, #4
tj        COPY_DATA
COPY_DATA_END:

```

Data transferring from flash is slow. As a reference, 16 bytes would take 7us. So more data are in "data" sector, the longer T_cs_data and T_init would be, or vice versa.

User can use method explained earlier to check size of "data" sector in list file.

If "data" sector is too big and there is enough space in deepsleep retention area, user can add the keyword "attribute_data_retention" to place some of the variables in "data" sector into "retention_data" sector, so as to reduce T_cs_data and T_init.

T_cs_bss is time to initialize SRAM "bss" sector. Initial values of "bss" sector are all 0s. It's only need to reset SRAM "bss" sector to 0, and no flash transferring is needed. Corresponding assembly code is:

```

tmov      r0, #0
tloadr    r1, DAT0 + 16
tloadr    r2, DAT0 + 20
ZERO:
tcmp      r1, r2
tjge      ZERO_END
tstorer   r0, [r1, #0]
tadd      r1, #4
tj        ZERO
ZERO_END:

```

Resetting each word (4 byte) to 0 can be very fast. So when "bss" is small, T_cs_bss is very small. But if "bss" sector is large, for example when a huge global data array is defined (int AAA[2000] = {0}), T_cs_bss can also be very long. So it is worth paying attention to "bss" size in list file.

To optimize T_cs_bss when "bss" sector is large, if retention area allows, some of them can also be defined as "attribute_data_retention" to place in "retention_data" sector.

(5) T_init measurement

After T_cstartup and T_userInit are optimized to minimize T_init, it's also needed to measure T_init, and apply to API: `blc_pm_setDeepSleepRetentionEarlyWakeupTiming`

T_init starts at the timing as T_cstartup, which is the "_reset" point in cstartup_8258_RET_16K.S file as shown below:

```

__reset:

#if 0
@ add debug, PB4 output 1
tloadr    r1, DEBUG_GPIO    @0x80058a  PB oen
tmov      r0, #139          @0b 11101111
tstorerb   r0, [r1, #0]

```

```

    tmov      r0, #16      @0b 00010000
    tstorerb  r0, [r1, #1] @0x800583 PB output
#endif

```

Combined with the Debug gpio indication in the picture “T_init timing”, the Debug GPIO PB4 output high operation is placed in “__reset”. The user only needs to change “#if 0” to “#if 1” to enable the PB4 output high operation.

T_cstartup finishes at “tjl main”.

```

tjl main
END:    tj  END

```

Since main function starts almost at the end of T_cstartup, PB4 can be set to output low at beginning of main function as shown below. Please note that DBG_CHN0_LOW requires enabling “DEBUG_GPIO_ENABLE” in app_config.h.

```

_attribute_ram_code_ int main (void)    //must run in ramcode
{
    DBG_CHN0_LOW;    //debug
    cpu_wakeup_init();
    .....
}

```

By scoping signal of PB4, T_cstartup is obtained.

Adding PB4 output high at end of T_userInit inside user_init_deepRetn will generate same timing diagram as Debug gpio as shown above. T_init and T_cstartup can be measured by oscilloscope or logic analyzer. Following understanding of GPIO operation, user can modify the Debug gpio code as needed, so as to get other timing parameters as well, e.g. T_sysInit, T_userInit etc.

4.2.9 Connection Latency

4.2.9.1 Sleep timing with non-zero connection latency

The previous introduction to the sleep mode of Conn state slave role (refer to the figure “sleep timing for Advertising state & Conn state Slave role”) is based on the premise that connection latency (conn_latency for short) does not take effect.

In the PM software processing flow, $T_{wakeup} = T_{brx} + conn_interval$, the corresponding code is as follows.

```

if(conn_latency != 0)
{
    latency_use = bls_calculateLatency();
}

```

```

    T_wakeup = T_brx + (latency_use + 1) * conn_interval;
}
else
{
    T_wakeup = T_brx + conn_interval;
}

```

When the BLE slave goes through the connection parameters update process and `conn_latency` takes effect, the sleep wake_up time is:

$$T_{wakeup} = T_{brx} + (latency_use + 1) * conn_interval;$$

Following diagram illustrates sleep timing with non-zero `conn_latency` when `latency_use = 2`.

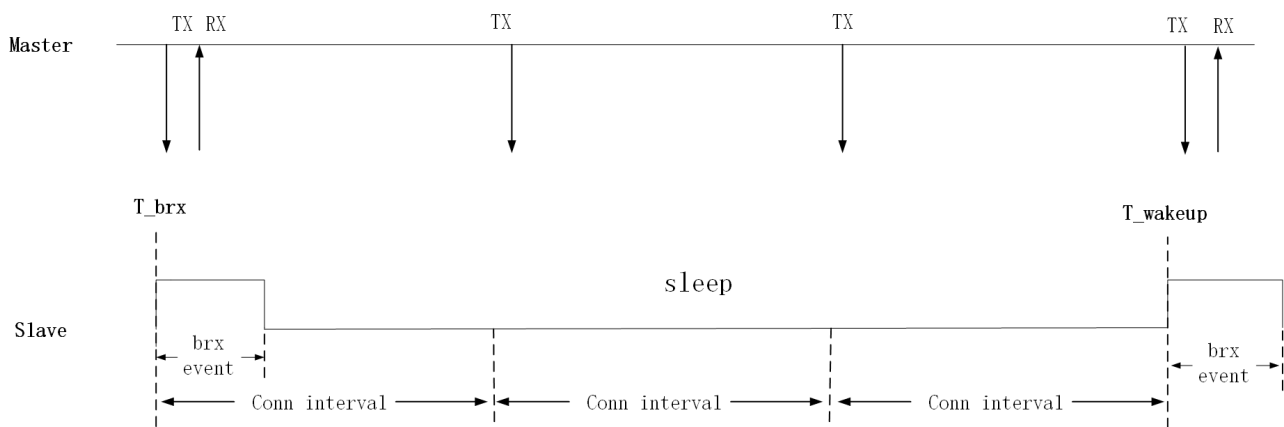


Figure 4.6: "Sleep Timing for Valid Conn_latency"

When `conn_latency` is not effective, the sleep duration is no more than 1 connection interval (generally small). After `conn_latency` becomes effective, the sleep time may have a relatively large value, such as 1S, 2S, etc., and the system power consumption can become very low. It makes sense to use deepsleep retention mode with lower power consumption during long sleep.

4.2.9.2 latency_use calculation

At effective `conn_latency`, T_{wakeup} is determined by `latency_use`, so it is not necessarily equal to `conn_latency`.

```
latency_use = bls_calculateLatency();
```

In the calculation of `latency_use`, `user_latency` is involved. This is the value that the user can set. The API to be called and its source code are:

```

void bls_pm_setManualLatency(u16 latency)
{
    bltPm.user_latency = latency;
}

```

Initial value of `bltPm.user_latency` is `0xFFFF`, and at the end of `blt_brx_sleep` function it will be reset to `0xFFFF`, which means the value set by the API `bls_pm_setManualLatency` is only valid for latest sleep, so it needs to be set on every sleep event.

The calculation process of `latency_use` is as follows.

First calculate the system latency:

- (1) If connection latency is 0, system latency is 0
- (2) If connection latency is not 0:
 - If system task is not done in current connection interval, MCU needs to wake up on next connection interval to continue the task such as transfer packet not completely sent out, or handle data from master not fully processed yet, and under this scenario, system latency is 0.
 - If no task is left over, system latency = connection latency. However, if slave receives master's update map request or update connection parameter request, and its updated timing is before (connection latency+1)*interval, then the actual system latency would force MCU to wake up before the updated timing point to ensure correct BLE timing sequence.

Combining `user_latency` and `system_latency`:

`latency_use = min(system_latency, user_latency)`

Accordingly, if `user_latency` set by the API `bls_pm_setManualLatency` is less than system latency, `user_latency` would be the final `latency_use`; otherwise, system latency is the final `latency_use`.

4.2.10 API `bls_pm_getSystemWakeupTick`

Following API is used to obtain wakeup time out of suspend (System Timer tick), or `T_wakeup`:

```
u32 bls_pm_getSystemWakeupTick(void);
```

According to `blt_brx_sleep` function in PM software process flow, `T_wakeup` is calculated fairly late, almost next to `cpu_sleep_wakeup`. Application layer can only get an accurate `T_wakeup` by `BLT_EV_FLAG_SUSPEND_ENTER` event callback function.

Following keyscan example explains usage of `BLT_EV_FLAG_SUSPEND_ENTER` event callback function and `bls_pm_getSystemWakeupTick`.

```
bls_app_registerEventCallback(BLT_EV_FLAG_SUSPEND_ENTER, &ble_remote_set_sleep_wakeup);

↵
void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN && ((u32)
        ↵ (bls_pm_getSystemWakeupTick() - clock_time())) >
        80 * CLOCK_SYS_CLOCK_1MS){
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);
    }
}
```

Above callback function is meant to prevent loss of key press.

A normal key press lasts for a few hundred ms, or at least 100-200ms for a fast press. When Advertising state and Conn state are configured by `bls_pm_setSuspendMask` to enter sleep mode, without `conn_latency` in effect, as long as Adv interval or `conn_interval` is not very long, typically less than 100ms, sleep time will not exceed Adv Interval or `conn_interval`, in other words, sleep time is less than 100ms or a fast key press time, loss of key press can be prevented and there is no need to enable GPIO wakeup.

With `conn_latency` ON, for example, with `conn_interval` = 10ms, `connec_latency` = 99, sleep time may last 1s, obviously key loss may occur. If current state is Conn state and wakeup time of suspend to be entered is more than 80ms from current time as determined by `BLT_EV_FLAG_SUSPEND_ENTER` callback function, key loss can be prevented by using GPIO level trigger to wake up MCU for keyscan process in case timer wakeup is too late.

4.3 Issues in GPIO Wake-up

4.3.1 Fail to enter sleep mode when wake-up level is valid

In 8x5x, GPIO wakeup is level triggered instead of edge triggered, so when GPIO PAD is configured as wakeup source, for example, suspend wakeup triggered by GPIO high level, MCU needs to make sure when MCU invokes `cpu_wakeup_suspend` to enter suspend, that the wakeup GPIO is not at high level. If the current level is already high, the actual entry into the `cpu_wakeup_sleep` function will be invalid when the suspend is triggered, and it will exit immediately, i.e. it will not enter the suspend at all.

If the above situation occurs, it may cause unexpected problems, for example, it was intended to enter deepsleep and be woken up and the program re-executed, but it turns out that the MCU cannot enter deepsleep, resulting in the code continuing to run, not in the state we expected, and the whole flow of the program may be messed up.

User should pay attention to avoid this problem when using Telink's GPIO PAD to wake up.

If the APP layer does not avoid this problem, and GPIO PAD wakeup source is already effective at invoking of `cpu_wakeup_sleep`, PM driver makes some improvement to avoid flow mess:

(1) Suspend & deepsleep retention mode

For both suspend and deepsleep retention mode, the SW will fast exit `cpu_wakeup_sleep` with two potential return values:

- Return `WAKEUP_STATUS_PAD` if the PM module has detected effective GPIO PAD state.
- Return `STATUS_GPIO_ERR_NO_ENTER_PM` if the PM module has not detected effective GPIO PAD state.

(2) deepsleep mode

For deepsleep mode, PM diver will reset MCU automatically in bottom layer (equivalent to watchdog reset). The SW restarts from "Run hardware bootloader".

To prevent this problem, following is implemented in the SDK demo "b85m_ble remote".

In `BLT_EV_FLAG_SUSPEND_ENTER`, it is configured that only when suspend time is larger than a certain value, can GPIO PAD wakeup be enabled.

```
void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN && ((u32)(bls_pm_getSystemWakeupTick() -
        ↪ clock_time())) >
        80 * CLOCK_SYS_CLOCK_1MS){
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);
    }
}
```

When key is pressed, manually set latency to 0 or a small value (as shown in below code), so as to ensure short sleep time, e.g. shorter than 80ms as set in above code. Therefore, the high level on drive pin due to a pressed key will never become a high-level GPIO PAD wakeup trigger.

```
int user_task_flg = ota_is_working || scan_pin_need || key_not_released || DEVICE_LED_BUSY;

if(user_task_flg){

    bls_pm_setSuspendMask (SUSPEND_ADV | SUSPEND_CONN);

    #if (LONG_PRESS_KEY_POWER_OPTIMIZE)
        extern int key_matrix_same_as_last_cnt;
        if(!ota_is_working && key_matrix_same_as_last_cnt > 5){ //key matrix stable can optimize
            bls_pm_setManualLatency(3);
        }
        else{
            bls_pm_setManualLatency(0); //latency off: 0
        }
    #else
        bls_pm_setManualLatency(0);
    #endif
}
```

Figure 4.7: "Low Power Code"

There are 2 scenarios that will make MCU enter deepsleep.

- First one is if there's no event for 60s, MCU will enter deepsleep. The events here include keys being pressed, so there is no drive pin high at this point to make deepsleep inaccessible.
- The other scenario is if a key is stuck for more than 60s, MCU will enter deepsleep. Under the second scenario, the SDK will invert polarity from high level trigger to low level trigger to solve the problem.

4.4 BLE System Low Power Management

Based upon understanding of PM principle of this BLE SDK, user can configure PM under different application scenarios, referring to the demo "b85m_ble remote" low power management code as explained below.

Function blt_pm_proc is added in PM configuration of main_loop. This function must be placed at the end of main_loop to ensure it is immediate to blt_sdk_main_loop in time, since blt_pm_proc needs to configure low power management according to different UI entry tasks.

Summary of highlights in blt_pm_proc function:

- (1) When UI task requires turning off sleep mode, such as audio (ui_mic_enable) and IR, set `bltm.suspend_mask` to `SUSPEND_DISABLE`.
- (2) After advertising for 60s in Advertising state, MCU enters deepsleep with wakeup source set to GPIO PAD in user initialization. The 60s timeout is determined by software timer using `advertise_begin_tick` variable to capture advertising start time.

The design of 60s into deepsleep is to save power, prevent slave wasting power on advertising even when not connected with master. User can justify 60s setting based on different applications.

- (3) At Conn state slave role, under conditions of no key press, no audio or LED task for over 60s from last task, MCU enters deepsleep with GPIO PAD as wakeup source, and at the same time set `DEEP_ANA_REG0` label in deepsleep register, so that once after wakeup slave will connect quickly with master.

The design of 60s into deepsleep is to save power. Actually if power consumption under connected state is tuned low enough as with deepsleep retention, it is not absolutely necessary to enter deepsleep.

To enter deepsleep at Conn state slave role, slave first issues a `TERMINATE` command to master by calling `bls_ll_terminateConnection`, after receiving ack which triggers `BLT_EV_FLAG_TERMINATE` callback function, slave will enter deepsleep. If slave enters deepsleep without sending any request, since master is still at connected state and would constantly try to synchroinoriz with slave till connection timeout. The connection timeout could be a very large value, e.g. 20s. If slave wakes up before 20s, slave would send advertising packet attempting to connect with master. But since master would assume it is already in connected state, it would not be able to connect to slave, and user experience is therefore very slow reconnection.

- (4) If certain task can not be disrupt by long sleep time, `user_latency` can be set to 0, so `latency_use` is 0.

Under applications such as `key_not_released`, or `DEVICE_LED_BUSY`, call API `bls_pm_setManualLatency` to set `user_latency` to 0. When `conn_interval` is 10ms, sleep time is no more than 10ms.

- (5) For scenario as in item 4, with latency set to 0, slave will wakeup at every conn interval, power might be unnecessarily too high since key scan and LED task does not repeat on every conn interval. Further power optimization can be done as following:

When `LONG_PRESS_KEY_POWER_OPTIMIZE=1`, once key press is stable (`key_matrix_same_as_last_cnt > 5`), manually set latency. With `bls_pm_setManualLatency` (3), sleep time will not exceed $4 * \text{conn_interval}$. If `conn_interval=10 ms`, MCU will wake up every 40ms to process LED task and keyscan.

User can tweak this approach toward different conn intervals and task response time requirements.

4.5 Timer Wake-up by Application Layer

At Advertising state or Conn state Slave role, without GPIO PAD wakeup, once MCU enters sleep mode, it only wakes up at `T_wakeup` pre-determined by BLE SDK. User can not wake up MCU at an earlier time which might be needed at certain scenario. To provide more flexibility, application layer wakeup and associated callback function are added in the SDK:

Application layer timer wakeup API:

```
void bls_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

“wakeup_tick” is wakeup time at System Timer tick value.

“enable”: 1-wakeup is enabled; 0-wakeup is disabled.

Registered call back function `bls_pm_registerAppWakeupLowPowerCb` is executed at application layer wakeup:

```
typedef void (*pm_appWakeupLowPower_callback_t)(int);
void bls_pm_registerAppWakeupLowPowerCb(pm_appWakeupLowPower_callback_t cb);
```

Take Conn state Slave role as an example:

When the user uses `bls_pm_setAppWakeupLowPower` to set the `app_wakeup_tick` for the application layer to wake up regularly, the SDK will check whether `app_wakeup_tick` is before `T_wakeup` before entering sleep.

- If `app_wakeup_tick` is before `T_wakeup`, as shown in the figure below, it will trigger sleep in `app_wakeup_tick` to wake up early;
- If `app_wakeup_tick` is after `T_wakeup`, MCU will still wake up at `T_wakeup`.

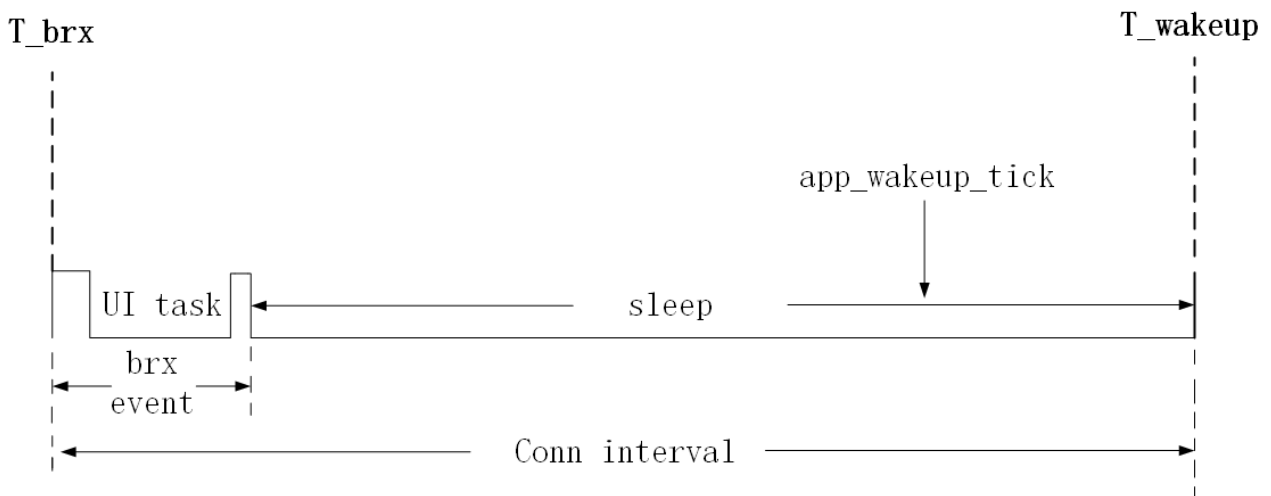


Figure 4.8: “EarlyWake_upatapp_wakeup_tick”

5 Low Battery Detect

Battery power detect/check, which may also appear in the Telink BLE SDK and related documentation under other names, includes: battery power detect/check, low battery detect/check low power detect/check), battery detect/check, etc. For example, the relevant files and functions in the SDK are named `battery_check`, `battery_detect`, `battery_power_check`, etc.

This document is unified under the name of “low battery detect”.

5.1 The importance of low battery detect

For battery-powered products, as the battery power will gradually drop, when the voltage is low to a certain value, it will cause many problems.

- a) The operating voltage range of 8x5x chip is 1.8V~3.6V. When the voltage is lower than 1.8V, 8x5x chip can no longer guarantee stable operation.
- b) When the battery voltage is low, due to the unstable power supply, the “write” and “erase” operations of Flash may have the risk of error, causing the program firmware and user data to be modified abnormally, and eventually causing the product to fail. Based on our previous mass production experience, we set the low voltage threshold for this risk to 2.0V.

According to the above description, for battery-powered products, a secure voltage must be set, and the MCU is allowed to continue working only when the voltage is higher than this secure voltage; once the voltage falls below the secure voltage, the MCU stops running and needs to be shutdown immediately (this is achieved by entering deepsleep mode on the SDK).

The secure voltage is also called alarm voltage, and the value of this voltage is 2.0 V by default in the SDK.

Note:

- The low voltage protection threshold 2.0V is an example and reference value. Customers should evaluate and modify these thresholds according to the actual situation. If users have unreasonable designs in the hardware circuit, which leads to a decrease in the stability of the power supply network, the safety thresholds must be increased as appropriate.

For the product developed and implemented using Telink BLE SDK, as long as the use of battery power, low power detection must be a real-time operation of the task for the product’s entire life cycle to ensure the stability of the product.

The protection of low voltage detection for flash operation will be introduced further in the chapter on flash.

5.2 The implementation of low battery detect

The low battery detect requires the use of ADC to measure the power supply voltage. Users can refer to the 8258/8278 Datasheet and Driver SDK Developer Handbook chapter on ADC to get the necessary understanding of the B85 ADC module first.

The implementation of the low battery detect is described in the SDK demo "B85m_ble_sample", refer to the files battery_check.h and battery_check.c.

Make sure the macro "BATT_CHECK_ENABLE" is enabled in app_config.h. This macro is disabled by default, and users need to pay attention to it when using the low battery detect function.

```
#define BATT_CHECK_ENABLE 1
```

5.2.1 Notes on low battery detect

Low battery detect is a basic ADC sampling task, and there are a number of issues that need attention when implementing an ADC to sample the supply voltage, as described below.

5.2.1.1 GPIO input channel recommended

The 8x5x ADC input channels support ADC sampling of the supply voltage on the "VCC/VBAT" input channel, which corresponds to the last "VBAT" in the variable ADC_InputPchTypeDef below. However, for some special reasons, the 8x5x "VBAT" channel cannot be used. Telink stipulates that the "VBAT" input channel is not allowed, and the GPIO input channel must be used.

The available GPIO input channels are the input channels corresponding to PBO~PB7, PC4, and PC5.

```
/*ADC analog positive input channel selection enum*/
typedef enum {
.....
    B0P,
    B1P,
    B2P,
    B3P,
    B4P,
    B5P,
    B6P,
    B7P,
    C4P,
    C5P,
.....
    VBAT,
}ADC_InputPchTypeDef;
```

ADC sampling of the supply voltage using the GPIO input channel can be implemented in two ways:

- a) Power supply connected to the GPIO input channel

In the hardware circuit design, the power supply is directly connected to the GPIO input channel, and the ADC is initialized by setting the GPIO to high resistance (ie, oe, output all set to 0), at which time the voltage on the GPIO is equal to the power supply voltage, and ADC sampling can be performed directly.

- b) Power supply does not touch the GPIO input channel

There is no need for power supply and GPIO input channel connection on the hardware circuit. It needs to be measured with the output high level of GPIO. The 8x5x internal circuit structure is designed to ensure that the voltage value of GPIO output high level and power supply voltage value are always equal. The high level of the GPIO output can be used as the power supply voltage, and ADC sampling is performed through the GPIO input channel.

The current GPIO input channel selected by "b85m_ble_remote" is PB7, which adopts the second "power supply does not touch the GPIO input channel" method.

Choose PB7 as GPIO input channel, PB7 as ordinary GPIO function, initialize all states (ie, oe, output) using the default state, no special modification.

```
#define GPIO_VBAT_DETECT    GPIO_PB7
#define PB7_FUNC           AS_GPIO
#define PB7_INPUT_ENABLE    0
#define ADC_INPUT_PCHN     B7P
```

When ADC sampling is required, PB7 outputs high level:

```
gpio_set_output_en(GPIO_VBAT_DETECT, 1);
gpio_write(GPIO_VBAT_DETECT, 1);
```

The output state of PB7 can be turned off after the ADC sampling is finished. Since the PB7 pin on the "b85m_ble_remote" hardware circuit is floating (not connected to other circuits), the high output level does not cause any leakage, so the output state of PB7 is not turned off on the SDK.

5.2.1.2 Differential mode only

Although the 8x5x ADC input mode supports both Single Ended Mode and Differential Mode, for some specific reasons, Telink specifies that only Differential Mode can be used, and Single Ended Mode is not allowed.

The differential mode input channel is divided into positive input channel and negative input channel, the measured voltage is the voltage difference obtained by subtracting the negative input channel voltage from the positive input channel voltage.

If the ADC sample has only one input channel, when using differential mode, set the current input channel as the positive input channel and GND as the negative input channel, so that the voltage difference between the two is equal to the positive input channel voltage.

The differential mode is used in SDK low battery detect, the interface function is as follows. The "#if 1" and "#else" branches are the exact same function settings, the "#if 1" is just to make the code run faster to save time. It can be understood by looking at "#else", the adc_set_ain_channel_differential_mode API selects PB7 as the positive input channel and GND as the negative input channel.

```
#if 1 //optimize, for saving time
    //set misc channel use differential_mode,
    //set misc channel resolution 14 bit, misc channel differential mode
    analog_write (anareg_adc_res_m, RES14 | FLD_ADC_EN_DIFF_CHN_M);
#else
```

```

    adc_set_ain_chn_misc(ADC_INPUT_PCHN, GND);
#else
    ////set misc channel use differential mode,
    adc_set_ain_channel_differential_mode(ADC_MISC_CHN, ADC_INPUT_PCHN, GND);
    //set misc channel resolution 14 bit
    adc_set_resolution(ADC_MISC_CHN, RES14);
#endif

```

5.2.1.3 Must use Dfifo mode to obtain ADC sampling value

For 8x5x, Telink stipulates that only Dfifo mode can be used to read ADC sampling values. Refer to the following function.

```

unsigned int adc_sample_and_get_result(void);

```

5.2.1.4 Need to switch different ADC tasks

Refer to the "8258 Datasheet" that the ADC state machine includes several channels such as Left, Right, and Misc. Due to some special reasons, these state channels cannot work at the same time. Telink stipulates that the channels in the ADC state machine must operate independently.

The Misc channel is used for low battery detect as the most basic ADC sampling. Users need to use the Misc channel if they need other ADC tasks besides low battery detect. Amic Audio uses Left channel. The low battery detect cannot run simultaneously with other ADC tasks and must be implemented by switching.

5.2.2 Stand-alone use of low battery detect

Users define the macro "BLE_AUDIO_ENABLE" in the "b85m_ble_remote" app_config.h file to 0 (turn off all functions of Audio) to get a demo of the ADC being used only by low voltage detection. Or refer to the "b85m_module" demo for low voltage detection.

5.2.2.1 Low battery detect initialization

Refer to the implementation of the adc_vbat_detect_init function.

The order of ADC initialization must satisfy the following procedure: first power off sar adc, then configure other parameters, and finally power on sar adc. All initialization of ADC sampling must follow this flow.

```

void adc_vbat_detect_init(void)
{
    ////////power off sar adc////////
    adc_power_on_sar_adc(0);

    //add ADC configuration

```

```

/******power on sar adc******/
//note: this setting must be set after all other settings
adc_power_on_sar_adc(1);
}

```

For the configuration before sar adc power on and power off, the user try not to modify, and use the default settings. If users choose a different GPIO input channel, directly modify the ADC_INPUT_PCHN related macro definition described earlier. If user adopts the hardware circuit design “power supply connected to GPIO input channel”, the operation of “GPIO_VBAT_DETECT” output high level needs to be removed

The adc_vbat_detect_init initialization function is called in app_battery_power_check with the following code:

```

if(!adc_hw_initialized){
    adc_hw_initialized = 1;
    adc_vbat_detect_init();
}

```

Here a variable adc_hw_initialized is used, which is called once only when it is 0 and set to 1; it is not initialized again when it is 1. The adc_hw_initialized is also manipulated in the following API.

```

void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
    if(!en){
        adc_hw_initialized = 0; //need initialized again
    }
}

```

The functions that can be implemented by a design using adc_hw_initialized are:

- a) Switching with other ADC task

The effect of sleep mode (suspend/deepsleep retention) is not considered first, and only the switching between low battery detect and other ADC tasks is analyzed.

Because of the need to consider the switch between low battery detect and other ADC tasks, adc_vbat_detect_init may be executed several times, so it cannot be written to user initialization and must be implemented in main_loop.

The first time the app_battery_power_check function is executed, adc_vbat_detect_init is executed and will not be executed repeatedly.

Once the “ADC other task” needs to be executed, it will take away the ADC usage and make sure that the “ADC other task” must call battery_set_detect_enable(0) when it is initialized, which will clear adc_hw_initialized to 0.

When the "ADC other task" is finished, the right to use the ADC is handed over. The `app_battery_power_check` is executed again, and since the value of `adc_hw_initialized` is 0, `adc_vbat_detect_init` must be executed again. This ensures that the low battery detect is reinitialized each time it is switched back.

b) Adaptive handling of suspend and deepsleep retention

Take sleep mode into account.

The `adc_hw_initialized` variable must be defined as a variable on the "data" or "bss" segment, not on the `retention_data`. Defining it on the "data" segment or "bss" ensures that this variable is used after each deepsleep retention wake_up when the software bootloader is executed (i.e., `cstartup.xxx.S`) will be re-initialized to 0; after sleep wake_up, this variable can be left unchanged.

The common feature of the register configured inside the `adc_vbat_detect_init` function is that it does not power down in suspend mode and can maintain the state; it will power down in deepsleep retention mode.

If the MCU enters into suspend mode, when it wakes up and executes `app_battery_power_check` again, the value of `adc_hw_initialized` is the same as before suspend, so there is no need to re-execute the `adc_vbat_detect_init` function.

If the MCU enters deepsleep retention mode and wakes up with `adc_hw_initialized` to 0, `adc_vbat_detect_init` must be re-executed and the ADC-related register state needs to be reconfigured.

The state of register set in the `adc_vbat_detect_init` function can be kept from powering down during the suspend.

Refer to the "Low Power Management" section that the Dfifo related registers will be powered down in suspend mode, so the following two codes are not put in the `adc_vbat_detect_init` function, but in the `app_battery_power_check` function, to ensure that it is reset before each low power detection.

```
adc_config_misc_channel_buf((u16 *)adc_dat_buf, ADC_SAMPLE_NUM<<2);
dfifo_enable_dfifo2();
```

The keyword "attribute_ram_code" is added to the `adc_vbat_detect_init` function in the SDK to set it to `ram_code`, with the ultimate goal of optimizing power consumption for long sleep connection states. For example, for a typical long sleep connection of 10ms * (99+1) = 1s, waking up every 1s and using deepsleep retention mode during long sleep, `adc_vbat_detect_init` must be executed again after each wake-up, and the execution speed will become faster after adding to `ram_code`.

This "_attribute_ram_code_" is not required. In the product application, the user can decide whether to put this function into `ram_code` based on the usage of the deepsleep retention area and the results of the power test.

5.2.2.2 Low battery detect processing

In `main_loop`, the `app_battery_power_check` function is called to implement the processing of low battery detect, and the related code is as follows.

```
_attribute_data_retention_ u8      lowBattDet_enable = 1;
                           u8      adc_hw_initialized = 0;
void battery_set_detect_enable (int en)
```

```

{
    lowBattDet_enable = en;

    if(!en){
        adc_hw_initialized = 0;    //need initialized again
    }
}
int battery_get_detect_enable (void)
{
    return lowBattDet_enable;
}

if(battery_get_detect_enable() && clock_time_exceed(lowBattDet_tick, 500000) ){
    lowBattDet_tick = clock_time();
app_battery_power_check(bat_deep_thres,bat_suspend_thres);
}

```

The default value of lowBattDet_enable is 1. Low battery detect is allowed by default, and the MCU can start low battery detect immediately after powering up. This variable needs to be set to retention_data to ensure that deepsleep retention cannot modify its state.

The value of lowBattDet_enable can only be changed when other ADC tasks need to seize ADC usage: when other ADC tasks start, battery_set_detect_enable(0) is called, at this time app_battery_power_check is not called again in main_loop; After the other ADC tasks are finished, call battery_set_detect_enable(1) to surrender the right to use ADC, then the app_battery_power_check function can be called again in main_loop.

The frequency of low battery detect is controlled by the variable lowBattDet_tick, which is executed every 500ms in the demo. Users can modify this time according to their needs.

The specific implementation of the app_battery_power_check function seems to be cumbersome, involving the initialization of low-power detection, Dfifo preparation, data acquisition, data processing, low-power alarm processing, etc.

The ADC sampling data is acquired using Dfifo mode. Dfifo samples 8 strokes of data by default and calculates the average value after removing the maximum and minimum values.

The adc_vbat_detect_init function shows that the period of each adc sample is 10.4us, so the data acquisition process is about 83us.

The macro "ADC_SAMPLE_NUM" in the demo can be modified to 4, which shortens the ADC sampling time to 41 us. it is recommended to use the method of 8 data strokes for more accurate calculation results.

```

#define ADC_SAMPLE_NUM      8

#if (ADC_SAMPLE_NUM == 4)    //use middle 2 data (index: 1,2)
    u32 adc_average = (adc_sample[1] + adc_sample[2])/2; #elif(ADC_SAMPLE_NUM == 8)    //use
    ↪ middle 4 data (index: 2,3,4,5)
    u32 adc_average = (adc_sample[2] + adc_sample[3] + adc_sample[4] +
adc_sample[5])/4;
#endif

```

The `app_battery_power_check` function is put on `ram_code`, refer to the above description of “`adc_vbat_detect_init`” `ram_code`, also to save running time and optimize power consumption.

The “`_attribute_ram_code_`” is not necessary. In the product application, the user can decide whether to put this function into `ram_code` based on the usage of the deepsleep retention area and the results of the power test.

```
_attribute_ram_code_ void app_battery_power_check(u16 a1ram_vol_mv);
```

5.2.2.3 Low voltage alarm

The parameter `alarm_vol_mv` of `app_battery_power_check` is to specify the alarm voltage in mV for low battery detect. According to the previous content, the default setting in SDK is 2000 mV. In the low voltage detection of `main_loop`, when the power supply voltage is lower than 2000 mV, it enters low voltage range.

The demo code for handling low voltage alarm is shown below. The MCU must be shutdown after low voltage, and no other work can be done.

The “B85m_ble_remote” uses the way of entering deepsleep to shut down the MCU, and a button is set to wake up the remote control.

In addition to shutdown, user can modify other alarm behaviors for low voltage alarm processing.

In the code below, 3 quick flashes are made using LED lights to inform the product user that the battery needs to be charged or replaced.

```
if(batt_vol_mv < alarm_vol_mv){
    #if (1 && BLT_APP_LED_ENABLE) //led indicate
        gpio_set_output_en(GPIO_LED, 1); //output enable
        for(int k=0;k<3;k++){
            gpio_write(GPIO_LED, LED_ON_LEVAL);
            sleep_us(200000);
            gpio_write(GPIO_LED, !LED_ON_LEVAL);
            sleep_us(200000);
        }
    #endif

    analog_write(DEEP_ANA_REG2, LOW_BATT_FLG); //mark
    cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);
}
```

After “B85m_ble_remote” is shutdown, they enter the deepsleep mode where they can be woken up. If a key wake-up occurs, the SDK will do a quick low battery detect during user initialization instead of waiting until the `main_loop`. The reason for this process is to avoid application errors, as illustrated by the following example.

If the product user has been alerted by the flashing LED during the low power alarm and then wakes up again by entering deepsleep, it takes at least 500ms to do the low battery detect from the processing of

main_loop. Before 500ms, the slave's broadcast packet has been sent for a long time, and it is likely to be connected to the master already. In this case, there is a bug that the device already having low power alarm continues to work again.

For this reason, the SDK must do the low battery detect in advance during user initialization, and must prevent the above situation from happening at this step. So add low battery detect during user initialization, and the function interface in the SDK is:

```
if(analog_read(DEEP_ANA_REG2) == LOW_BATT_FLG){  
    app_battery_power_check(VBAT_ALRAM_THRES_MV + 200); //2.2 V  
}
```

According to the value of DEEP_ANA_REG2 analog register can determine whether the low power alarm shutdown is woken up, at this time, a fast low power detection is performed and the previous 2000mV alarm voltage is increased to 2200mV (called recovery voltage). The reason for the 200mV increase is:

Low voltage detection will have some errors, can not guarantee the accuracy and consistency of the measurement results. For example, if the error is 20mV, the voltage detected for the first time may be 1990mV to enter shutdown mode, and then the voltage value detected again after waking up in user initialization is 2005mV. If the alarm voltage is still 2000mV, it still can't stop the bug described above.

Therefore, it is necessary to adjust the alarm voltage slightly higher than the maximum error of low power detection during the fast low power detection after the shutdown mode wakeup.

Only when a certain low power detection found that the voltage is lower than 2000mV into shutdown mode, the recovery voltage 2200mV will appear, so user does not have to worry about this 2200mV will misreport low voltage to the actual voltage 2V~2.2V products. Product users see the low voltage alarm indication, after charging or replacing the battery to meet the requirements of recovery voltage, the product back to normal use.

5.2.2.4 Low power detect debug mode

The "8258_ble_remote" demo code leaves two debug-related macros available to user for debugging.

```
#define DBG_ADC_ON_RF_PKT      0  
#define DBG_ADC_SAMPLE_DAT    0
```

It is only possible to open the above two "macros" when debugging.

After "DBG_ADC_ON_RF_PKT" is turned on, the ADC sampling result information will be displayed on the data packet of the broadcast packet and the key value of the connection state. Note: At this time, the broadcast packet and key data are modified, so they can only be used for debugging.

When "DBG_ADC_SAMPLE_DAT" is turned on, the intermediate results of ADC sampling can be stored on Sram.

5.2.3 Low battery detect and Amic Audio

Referring to the detailed introduction in Low Battery Detect Stand-alone Use mode, for products that need to implement Amic Audio, just switch between Low Battery Detect and Amic Audio.

According to the low battery detection stand-alone use mode, after the program starts running, the default low battery detection is enabled first. When Amic Audio is triggered, do the following two things.

(1) Disable low battery detection

Call `battery_set_detect_enable(0)` to inform the low battery detect module that the ADC resources have been seized.

(2) Amic Audio ADC initialization

Since the ADC is used in a different way than the low battery detection, the ADC needs to be initialized again. For details, refer to the "Audio" section of this document.

At the end of Amic Audio, `battery_set_detect_enable(1)` is called to inform the low battery detect module that the ADC resources have been released. At this point the low battery detection needs to reinitialize the ADC module and then start the low battery detection.

If it is low battery detection and other non-Amic Audio ADC tasks at the same time, the processing of other ADC tasks can imitate the processing flow of Amic Audio.

If there are three kinds of tasks at the same time: low battery detection, Amic Audio and other ADC tasks, user can refer to the method of switching between low battery detection and Amic Audio to implement them according to the principle of "switching if ADC circuit needs".

6 Audio

The source of Audio can be AMIC or DMIC.

- DMIC is a chip that directly uses peripheral audio processing to read digital signals onto 827x or 825x;
- AMIC needs to use the codec module inside the chip to sample and post-process the original Audio signal, and finally convert it into a digital signal and transmit it to the MCU.

6.1 Initialization

6.1.1 AMIC and Low Power Detect

Refer to the introduction of “low-power detection” in this document, when Amic Audio and low-power detection use the ADC module, ADC must be switched.

Similarly, if the two tasks of Amic Audio and other ADC task, ADC need to be switched. If the three tasks of Amic Audio, low-power detect and other ADC task, ADC need to be switched.

825x/827x Amic needs to be set when the Audio task is on so that low power detection and Amic to ADC module switching can be used.

6.1.2 AMIC Initialization

Refer to the SDK demo “b85m_8258_ble_remote” speech processing related code.

```
void ui_enable_mic (int en)
{
    ui_mic_enable = en;

    gpio_set_output_en (GPIO_AMIC_BIAS, en); //AMIC Bias output
    gpio_write (GPIO_AMIC_BIAS, en);

    if(en){ //audio on
        audio_config_mic_buf ( buffer_mic, TL_MIC_BUFFER_SIZE);
        audio_amic_init(AUDIO_16K);
    }
    else{ //audio off
        adc_power_on_sar_adc(0); //power off sar adc
    }

    #if (BATT_CHECK_ENABLE)
        battery_set_detect_enable(!en);
    #endif
}
```

In the function "ui_enable_mic", the parameter "en" serves to enable (1) or disable (0) Audio task.

At the beginning of Audio, GPIO_AMIC_BIAS needs to output a high level to drive Amic; after Audio ends, GPIO_AMIC_BIAS needs to be turned off to prevent this pin from leaking in sleep mode.

Following shows AMIC initialization setting:

```
audio_config_mic_buf ( buffer_mic, TL_MIC_BUFFER_SIZE);  
audio_amic_init(AUDIO_16K);
```

In the working process of Audio, the data is continuously copied to SRAM through designated Dfifo. audio_config_mic_buf is used to configure the starting address and length of the Dfifo on the Sram.

The configuration of Dfifo is handled in the ui_enable_mic function, which is equivalent to doing it again every time Audio starts. The reason is that the Dfifo control register will be lost during suspend.

After the Audio task is over, the SAR ADC must be closed to prevent leakage during suspend:

```
adc_power_on_sar_adc(0);
```

Since Amic and low power detect need to switch between using the ADC module, add battery_set_detect_enable(!en) to the ui_enable_mic function to turn off and on the low power detect. Please refer to the introduction in the low power detect section of this document.

The execution of the Audio task is placed in the UI entry part of the main_loop.

```
#if (BLE_AUDIO_ENABLE)  
    if(ui_mic_enable){ //audio  
        task_audio();  
    }  
#endif
```

6.1.3 DMIC Initialization

TBD

6.2 Audio Data Processing

6.2.1 Audio Data Volume and RF Transfer

The raw data sampled by AMIC adopt pcm format. The demo currently provides three compression algorithms, sbc, msbc and adpcm, with adpcm using the pcm-to-adpcm algorithm to compress the raw data into adpcm format with compression ratio of 25%, thus BLE RF data volume will be decreased largely. Master will decompress the received adpcm-format data back to pcm format.

AMIC sampling rate is 16K x 16bits, corresponding to 16K samples of raw data per second, i.e. 16 samples per millisecond (16*16bits=32bytes per ms).

For every 15.5ms, 496-byte (15.5*16=248 samples) raw data are generated. Via pcm-to-adpcm conversion with compression ratio of 1/4, the 496-byte data are compressed into 124 bytes.

The 128-byte data, including 4-byte header and 124-byte compression result, will be disassembled into five packets, and sent to Master in L2CAP layer; since the maximum length of each packet is 27 bytes, the first packet must contain 7-byte l2cap information, including: 2-byte l2caplen, 2-byte chanid, 1-byte opcode and 2-byte AttHandle.

Figure below shows the RF data captured by sniffer. The first packet contains 7-byte extra information and 20-byte audio data, followed by four packets with 27-byte audio data each. As a result, total audio data length is 20 + 27*4 = 128 bytes.

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FEFDC	-38	OK
	1	1	1	0	0			

Data Type	Data Header					L2CAP Header													
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode AttHandle AttValue											
	2	0	1	1	27	0x0083	0x0004	0x1B 0x002B 3F 03 07 7C A9 BE 13 65 21 43 51 B1 43 22 14 10 C3 40 22 25											

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE4A9	-38	OK
	1	0	0	0	0			

Data Type	Data Header					Generic L2CAP Payload														CRC	RSSI (dBm)	FCS
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	80 94 38 33 73 08 11 2A 32 61 94 11 99 53 41 92 99 A9 E9 81 8B 1C 9A 09 AA D1 8B														0x132A61	-38	OK
	1	1	0	1	27																	

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FEFDC	-38	OK
	1	1	1	0	0			

Data Type	Data Header					Generic L2CAP Payload														CRC	RSSI (dBm)	FCS
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	AC BB C9 B9 C9 8A 8D CB 4B 9C 09 AB 99 29 0F AB 0B 1A 0F 04 15 21 53 30 C8 17 90														0x36A693	-38	OK
	1	0	1	1	27																	

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE4A9	-38	OK
	1	0	0	0	0			

Data Type	Data Header					Generic L2CAP Payload														CRC	RSSI (dBm)	FCS
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	19 09 89 89 89 A8 08 8A 50 E9 19 8A B8 D0 08 AA F9 88 C1 A0 9A B1 1B 9A 9E CA C9														0x441600	-38	OK
	1	1	0	1	27																	

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FEFDC	-38	OK
	1	1	1	0	0			

Data Type	Data Header					Generic L2CAP Payload														CRC	RSSI (dBm)	FCS
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	E0 81 0B 09 1A DB B3 99 A9 D2 99 0F B9 91 C9 B0 B1 CB B2 E1 1A AA 13 0F 3A 47 32														0xF05ACB	-38	OK
	1	0	1	0	27																	

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE4A9	-38	OK
	1	0	0	0	0			

Data Type	Data Header					CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE27A	-38	OK
	1	1	0	0	0			

Figure 6.1: "Audio Data Sample"

According to "Exchange MTU size" in ATT & GATT (section 3.3.3 ATT & GATT), since 128-byte long audio data packet are disassembled on Slave side, if peer device needs to re-assemble these received packets, we should determine maximum ClientRxMTU of peer device. Only when "ClientRxMTU" is 128 or above, can the 128-byte long packet of Slave be correctly processed by peer device.

So when the audio task is on and needs to send 128 byte long packets, `blc_att_requestMtuSizeExchange` will be called for Exchange MTU size.

```
void voice_press_proc(void)
{
    key_voice_press = 0;
    ui_enable_mic (1);
}
```

```
if(ui_mtu_size_exchange_req && blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){
    ui_mtu_size_exchange_req = 0;
    blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 0x009e);
}
}
```

The recommended practice is:

- (1) Turn on the MUT registration switch through `blc_gap_setEventMask (GAP_EVT_MASK_ATT_EXCHANGE_MTU)`;
- (2) Then register the GAP callback function through `blc_gap_registerHostEventHandler (gap_event_handler_t handler)`;
- (3) Add `if(event==GAP_EVT_ATT_EXCHANGE_MTU)` judgment statement in the callback function to implement the MTU size Exchange callback, and in the callback to determine whether the ClientRxMTU of the peer device is greater than or equal to 128. Since the master device's ClientRxMTU is generally larger than 128, the SDK does not determine the actual ClientRxMTU through the callback.

Following is the audio service in Attribute Table:

```
// 0033 - 0036 MIC
{0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characterUUID), (u8*)(&PROP_READ_NOTIFY), 0}, //prop
{0,ATT_PERMISSIONS_READ,16,sizeof(my_MicData),(u8*)(&my_MicUUID), (u8*)(&my_MicData), 0}, //value
{0,ATT_PERMISSIONS_RDWR,2,sizeof(micDataCCC),(u8*)(&clientCharacterCfgUUID), (u8*)(micDataCCC), 0}, //value
{0,ATT_PERMISSIONS_RDWR,2,sizeof(my_MicName),(u8*)(&userdesc UUID), (u8*)(my_MicName), 0},
```

Figure 6.2: "MIC Service in Attribute Table"

The second Attribute above is used to transfer audio data. This Attribute uses "Handle Value Notification" to send Data to Master. After Master receives Handle Value Notification, the Attribute Value data corresponding to the five successive packets will be assembled into 128 bytes, and then decompressed back to the pcm-format audio data.

6.2.2 Audio Data Compression

Related macros are defined in the "application/audio/audio_config.h", as shown below:

```
#if (TL_AUDIO_MODE == TL_AUDIO_RCU_ADPCM_GATT_TLEINK)
#define ADPCM_PACKET_LEN 128
#define TL_MIC_ADPCM_UNIT_SIZE 248
#define TL_MIC_BUFFER_SIZE 992
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_ADPCM_GATT_GOOGLE)
#define ADPCM_PACKET_LEN 136 //(128+6+2)
#define TL_MIC_ADPCM_UNIT_SIZE 256
#define TL_MIC_BUFFER_SIZE 1024
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB)
#define ADPCM_PACKET_LEN 120
#define TL_MIC_ADPCM_UNIT_SIZE 240
#define TL_MIC_BUFFER_SIZE 960
```

```
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_ADPCM_HID)
    #define ADPCM_PACKET_LEN          120
    #define TL_MIC_ADPCM_UNIT_SIZE     240
    #define TL_MIC_BUFFER_SIZE        960
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_SBC_HID_DONGLE_TO_STB)
    #define ADPCM_PACKET_LEN          20
    #define MIC_SHORT_DEC_SIZE        80
    #define TL_MIC_BUFFER_SIZE        320
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_SBC_HID)
    #define ADPCM_PACKET_LEN          20
    #define MIC_SHORT_DEC_SIZE        80
    #define TL_MIC_BUFFER_SIZE        320
#elif (TL_AUDIO_MODE == TL_AUDIO_RCU_MSBC_HID)
    #define ADPCM_PACKET_LEN          57
    #define MIC_SHORT_DEC_SIZE        120
    #define TL_MIC_BUFFER_SIZE        480
```

Each compression needs to process 248-sample, i.e. 496-byte data. Since AMIC continuously samples audio data and transfers the processed pcm-format data into `buffer_mic`, considering data buffering and preservation, this buffer should be pre-configured so that it can store 496 samples for two compressions. If 16K sampling rate is used, then 496 samples correspond to 992 bytes, i.e. "TL_MIC_BUFFER_SIZE" should be configured as 992.

The "buffer_mic" is defined as below:

```
s16 buffer_mic[TL_MIC_BUFFER_SIZE>>1]; //496 sample,992 bytes
config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);
```

Following shows the mechanism of data filling into `buffer_mic` via HW control.

Data sampled by AMIC are transferred into memory starting from `buffer_mic` address with 16K speed; once the maximum length 992 is reached, data transfer returns to the `buffer_mic` address, the old data will be replaced directly without checking whether it's read. It's needed to maintain a write pointer when transferring data into RAM; the pointer is used to indicate the address in RAM for current newest audio data.

The "buffer_mic_enc" is defined to store the 128-byte compression result data; the buffer number is configured as 4 to indicate result of up to four compressions can be buffered.

```
int buffer_mic_enc[BUFFER_PACKET_SIZE];
```

Since "BUFFER_PACKET_SIZE" is 128, and "int" occupies four bytes, it's equivalent to 128*4 signed char.

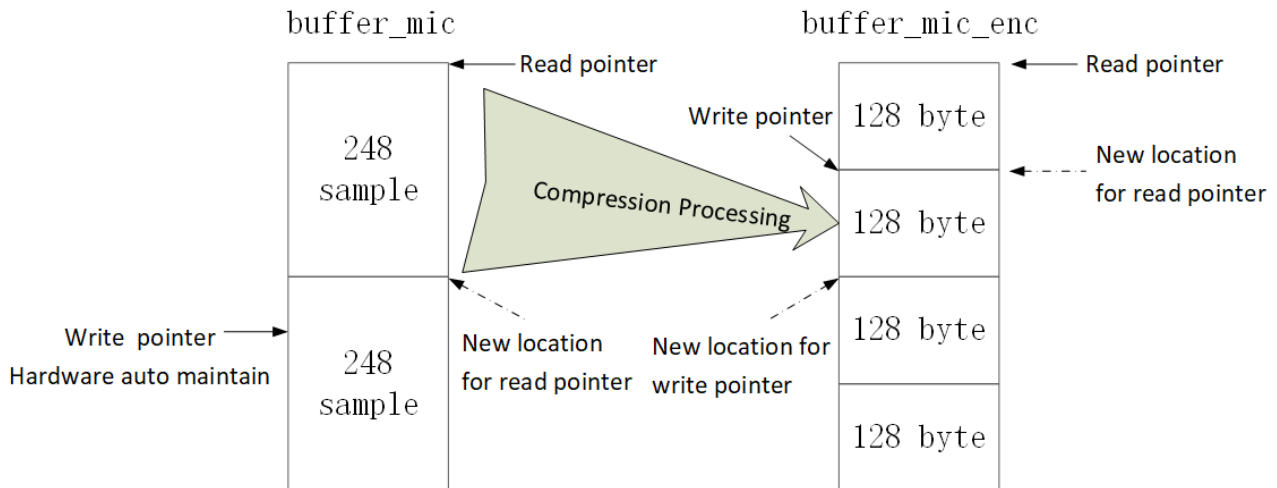


Figure 6.3: "Data Compression Processing"

The figure above shows data compression processing method:

The buffer_mic automatically maintains a write pointer by hardware, and maintains a read pointer by software.

Whenever SW detects there are 248 samples between the two pointers, the compression handler is invoked to read 248-sample data starting from the read pointer and compress them into 128 bytes; the read pointer moves to a new location to indicate following data are new and not read. The buffer_mic is continuously checked whether there are enough 248-sample data; if so, the data are compressed and transferred into the buffer_mic_enc.

Since 248-sample data are generated for every 15.5ms, the firmware must check the buffer_mic with maximum frequency of 1/15.5ms. The FW only executes the task_audio once during each main_loop, so the main_loop duration must be less than 15.5ms to avoid audio data loss. In Conn state, the main_loop duration equals connection interval; so for applications with audio task, connection interval must be less than 15.5ms. It's recommended to configure connection interval as 10ms.

The buffer_mic_enc maintains a write pointer and a read pointer by software: after the 248-sample data are compressed into 128 bytes, the compression result are copied into the buffer address starting from the write pointer, and the buffer_mic_enc is checked whether there's overflow; if so, the oldest 128-byte data are discarded and the read pointer switches to the next 128 bytes.

The compression result data are copied into BLE RF Tx buffer as below:

The buffer_mic_enc is checked if it's non-empty (when writer pointer equals read pointer, it indicates "empty", otherwise it indicates "non-empty"); if the buffer is non-empty, the 128-byte data starting from the read pointer are copied into the BLE RF Tx buffer, then the read pointer moves to the new location.

The function "proc_mic_encoder" is used to process Audio data compression.

6.3 Compression and Decompression Algorithm

The B85m single connection SDK provides sbc, msbc and adpcm compression and decompression algorithms, the following mainly takes adpcm to explain the entire compression and decompression algorithm.

About sbc and msbc, the user can refer to the project implementation to understand.

The function below is used to invoke the adpcm compression algorithm:

```
void mic_to_adpcm_split (signed short *ps, int len, signed short *pds, int start);
```

- “ps” points to the starting storage address for data before compression, which corresponds to the read pointer location of the buffer_mic as shown in figure above;
- “len” is configured as “TL_MIC_ADPCM_UNIT_SIZE (248)”, which indicates 248 samples;
- “pds” points to the starting storage address for compression result data, which corresponds to the write pointer location of the buffer_mic_enc as shown in figure above.

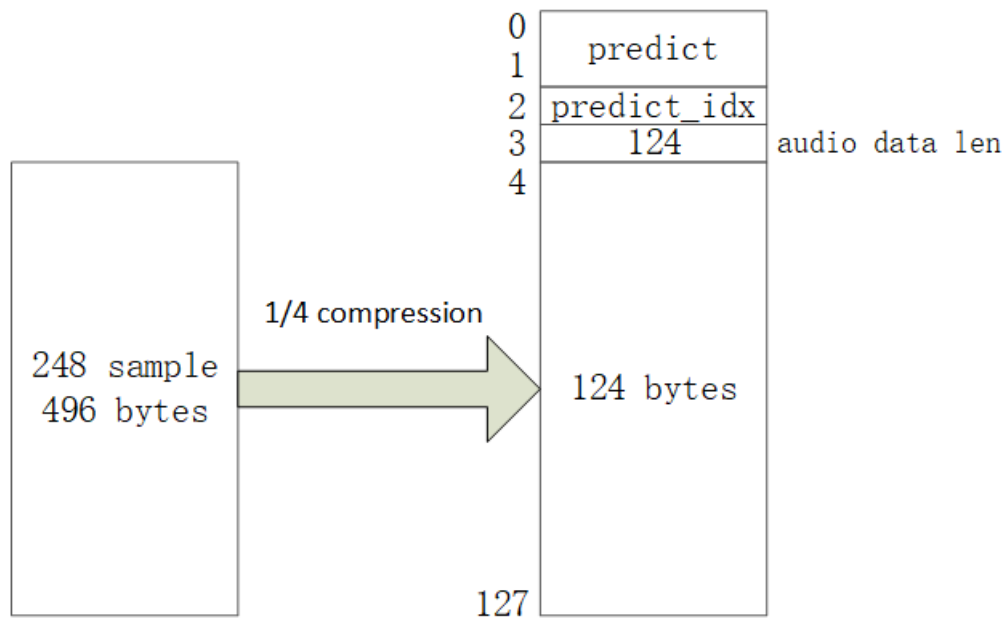


Figure 6.4: “Data Corresponding to Compression Algorithm”

After compression, the data space stores 2-byte predict, 1-byte predict_idx, 1-byte length of current valid adpcm-format audio data (i.e. 124), and 124-byte data compressed from the 496-byte raw data with compression ratio of 1/4.

The function below is used to invoke the decompression algorithm:

```
void adpcm_to_pcm (signed short *ps, signed short *pd, int len);
```

- “ps” points to the starting storage address for data to be decompressed (i.e. 128-byte adpcm-format data). This address needs user to define a buffer to store 128-byte data copied from BLE RF.
- “pd” points to the starting storage address for 496-byte pcm-format audio data after decompression. This address needs user to define a buffer to store data to be transferred when playing audio.
- “len” is 248, same as the “len” during compression.

As shown in figure above, during decompression, the data read from the buffer are two-byte predict, 1-byte predict_idx, 1-byte valid audio data length "124", and the 124-byte adpcm-format data which will be decompressed into 496-byte pcm-format audio data.

6.4 Audio data processing flow

The project in B85m SDK's "b85m_ble_remote" and "b85m_master_kma_dongle" contains a number of mode options, the user can select by changing the macro in app_config.h, the default is TL_AUDIO_RCU_ADPCM_GATT_TLEINK, that is, Telink custom Audio processing, its related settings are as follows.

```
/* Audio MODE:
 * TL_AUDIO_RCU_ADPCM_GATT_TLEINK
 * TL_AUDIO_RCU_ADPCM_GATT_GOOGLE
 * TL_AUDIO_RCU_ADPCM_HID
 * TL_AUDIO_RCU_SBC_HID
 * TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB
 * TL_AUDIO_RCU_SBC_HID_DONGLE_TO_STB
 * TL_AUDIO_RCU_MSBC_HID
 */
#define TL_AUDIO_MODE TL_AUDIO_RCU_ADPCM_GATT_TLEINK
```

Since several of these modes have similar processes and the default Telink customization is just a single compression of voice data for transmission, the whole process is relatively simple.

The TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB and TL_AUDIO_RCU_SBC_HID_DONGLE_TO_STB are two modes with similar implementation functions but different encoding. So this chapter mainly describes on TL_AUDIO_RCU_ADPCM_GATT_GOOGLE, TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB and TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB. To implement audio function in the provided sdk, for slave program user can refer to the b85m_ble_remote project, for master program user can refer to the b85m_master_kma_dongle project.

Note:

- If in setting different modes, compile prompt error that XX function or variable lack of definition, this is due to the voice related lib library is not added, Users in the use of TL_AUDIO_RCU_ADPCM_GATT_GOOGLE, TL_AUDIO_RCU_MSBC_HID, TL_AUDIO_RCU_SBC_HID, respectively, need to add the corresponding library file, The code identified by encode is the encoding library on the slave side (b85m_ble_remote project), and the decoding library identified by decode is the decoding library on the master side (b85m_master_kma_dongle). The library file directory in the project is as shown in the figure below:

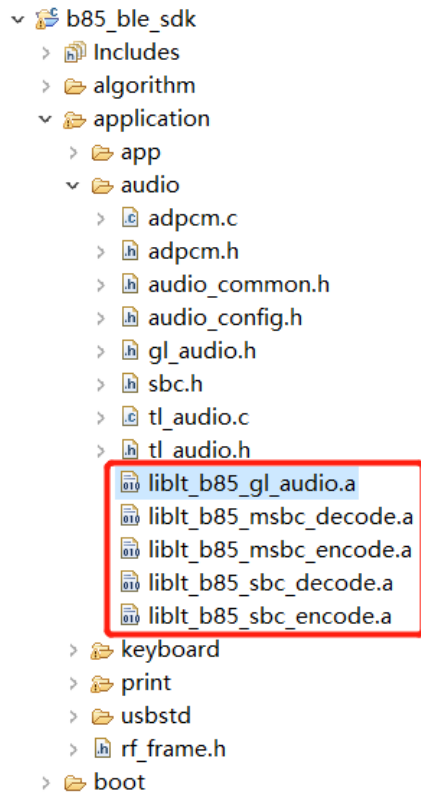


Figure 6.5: “Corresponding library files”

For example, if using SBC mode, the setting method is shown as below.

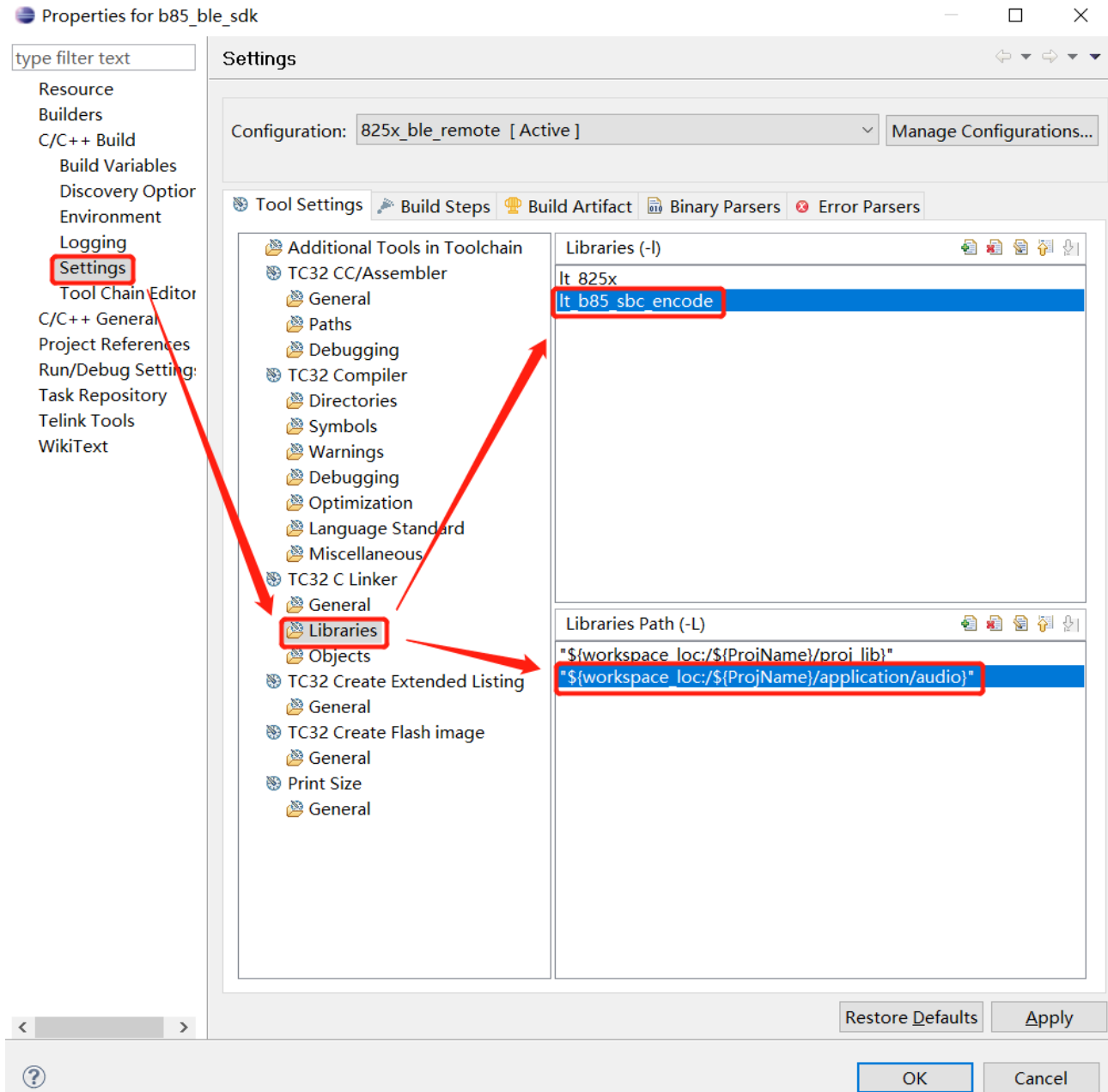


Figure 6.6: “SBC mode setting method”

6.4.1 TL_AUDIO_RCU_ADPCM_GATT_GOOGLE

Audio demo refers Google Voice VO.4 Spec for implementation, the user can use this demo and google TV box for voice-related product development. Google’s Service UUID is also set in accordance with the Spec provisions, as follows.

Type	Short-form	UUID	Properties
ATV Voice Service	ATVV_SERVICE_UUID	AB5E0001-5A21-4F05-BC7D-AF01F617B664	
Write Characteristic	ATVV_CHAR_TX	AB5E0002-5A21-4F05-BC7D-AF01F617B664	Write
Read Characteristic	ATVV_CHAR_RX	AB5E0003-5A21-4F05-BC7D-AF01F617B664	Notify

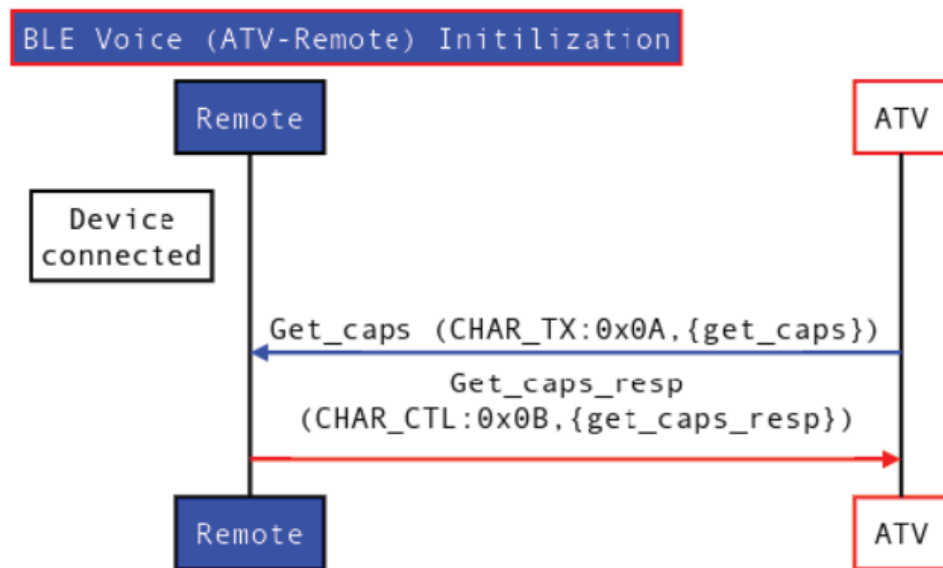
Google Confidential

3

Control Characteristic	ATVV_CHAR_CTL	AB5E0004-5A21-4F05-BC7D-AF01F617B664	Notify
------------------------	---------------	--------------------------------------	--------

Figure 6.7: "Google Service UUID setting"

6.4.1.1 Initialization


Figure 6.8: "Google Voice initialization flow"

Initialization is mainly the slave end to obtain the configuration information of the master end, the entire packet interaction information is as follows.



	ATT Write Command Packet (AB5E0002-5A21-4F05-BC7D-AF01F617B664: 0A 01 00 00 03 01)
	ATT Notification Packet (AB5E0004-5A21-4F05-BC7D-AF01F617B664: 0B 00 04 00 03 00 86 00 14)

Figure 6.9: "Packet Interaction Information"

6.4.1.2 Voice data transmission

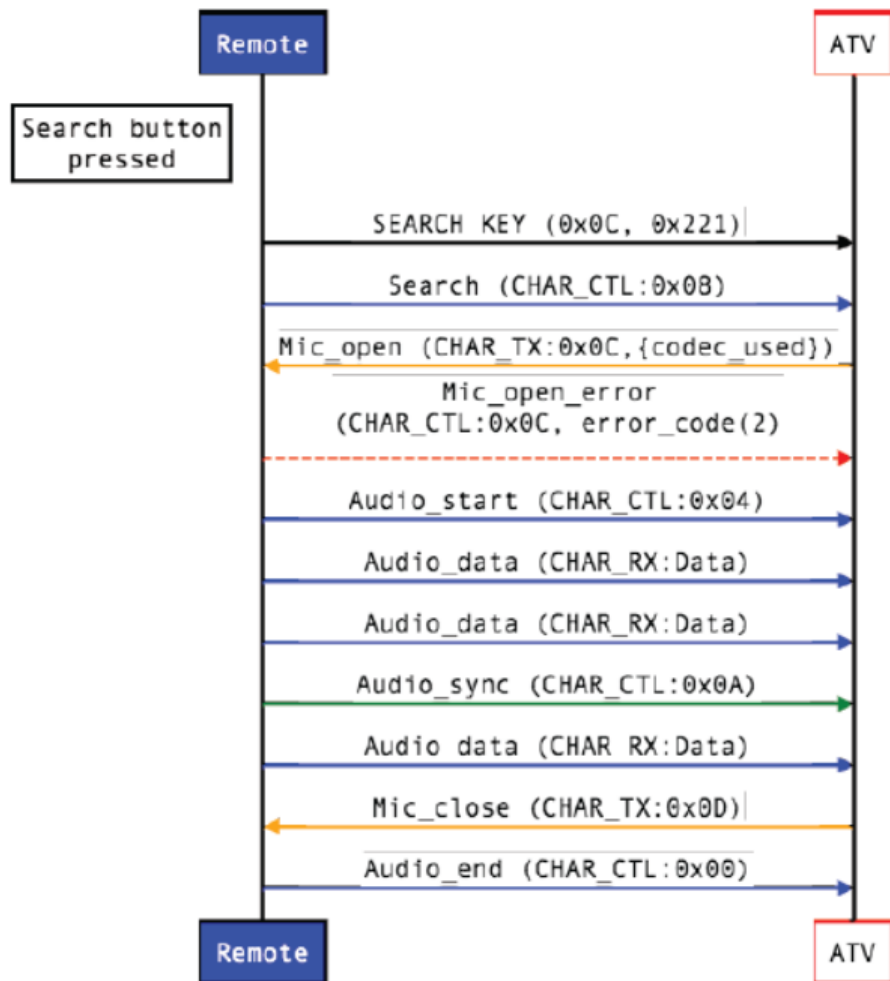


Figure 6.10: "Audio Data Transmission"

After the initialization is completed, the Slave end will send Search_KEY to the Master end, and the packet is as follows.



Figure 6.11: "Search_KEY packet"

Then the Slave end will send Search to the Master end with the following packet.



Figure 6.12: "Search packet"

Then the Master end will send MIC_Open to the Slave end, and the packet is as follows.

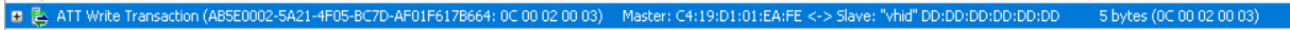


Figure 6.13: "MIC_Open packet"

Then the Slave end sends Start to the Master end with the following packet.



Figure 6.14: "Start packet"

According to Google Voice's Spec, the voice data transmission implemented in the program is 134 bytes per frame, and the entire packet is displayed as follows.



Figure 6.15: "134-byte Audio frame"

Note:

- On the Dongle side, we do not send a close command to end the voice transmission, but use a timeout judgment to end the voice. For details, please refer to the code of Dongle implementation on Master end.

6.4.1.3 TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB

This mode uses Service for the HID service specified in the Bluetooth Spec, through which the service can achieve communication with the Dongle connected devices, provided that the Dongle and the master computer device support the HID service method of interaction.

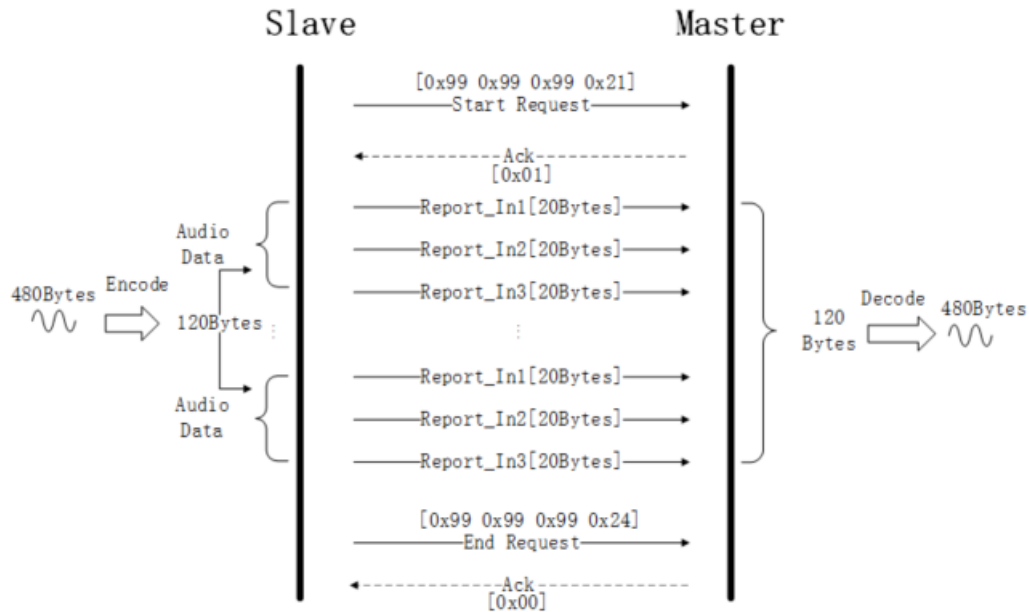


Figure 6.16: “Audio data interaction in ADPCM_HID_DONGLE_TO_STB mode”

At the beginning, the Slave sends `start_request` to the Master with the following packet.

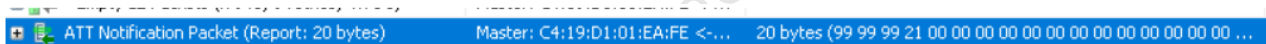


Figure 6.17: “Start_request packet”

After the Master receives the start_request, it sends the Ack_packet as follows.



Figure 6.18: “Ack packet”

Slave starts to send Audio voice data, the decompression and compression of voice data are operated in 480Bytes size, the voice data is first compressed to 120 bytes by ADPCM compression algorithm, then split into 6 groups of packets and sent to Master end in turn, each group packet size is 20 bytes. In order to ensure the sequence of voice packets, use every three groups of packets are changed in turn for a fixed handle value. The receiver side starts to decompress and restore the voice signal after completing 6 groups of packets. The packets are as follows.

ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (77 77 77 40 80 80 88 00 88 08 08 08 08 00 00 80 90 18)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (08 08 09 2A 92 B4 09 23 B1 05 C2 94 18 89 2B 41 1B 21 E8 13)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (B7 91 09 5B 39 90 9B 1A 07 80 89 59 89 A9 3B 44 B7 01 08 0B)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (00 29 03 1D 28 02 BB A9 59 08 01 3A D4 98 A9 43 00 BB 31 B5)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (3C 22 91 97 08 3C 48 18 8B 59 3B 11 A3 04 C3 0D 39 B1 18 0B)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (48 1B 97 19 9A 1B 89 BC 1B 29 F8 81 39 A9 94 0C 7B 11 90 B3)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (98 31 C2 90 92 0B 39 34 A5 A1 92 C8 B0 78 1C 1A 93 14 B1 32)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (80 9B 3A CA 2D 10 B0 58 91 F2 00 11 0D 32 00 E1 A1 21 AD 29)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (30 BA BB A9 CB 07 33 91 1B 14 B3 1B 90 D0 83 BF 3C 12 CA 29)

Figure 6.19: "Audio voice data"

At the end of the voice transmission, the Slave sends an End Request to the Master with the following packet.

ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-...	20 bytes (99 99 99 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00)
--	---------------------------------	--

Figure 6.20: "End request packet"

The Master sends an Ack after receiving the End Request with the following packet.

ATT Write Command Packet (Report: 1 byte)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAu..."	1 byte (00)
---	--	-------------

Figure 6.21: "Ack packet"

6.4.2 TL_AUDIO_RCU_SBC_HID_DONGLE_TO_STB

This mode and TL_AUDIO_RCU_ADPCM_HID_DONGLE_TO_STB, the same use of Service for the HID service specified in the Bluetooth Spec, through the service can achieve the communication among the Dongle connected, the premise is that the Dongle and the master computer device support the HID service interaction.

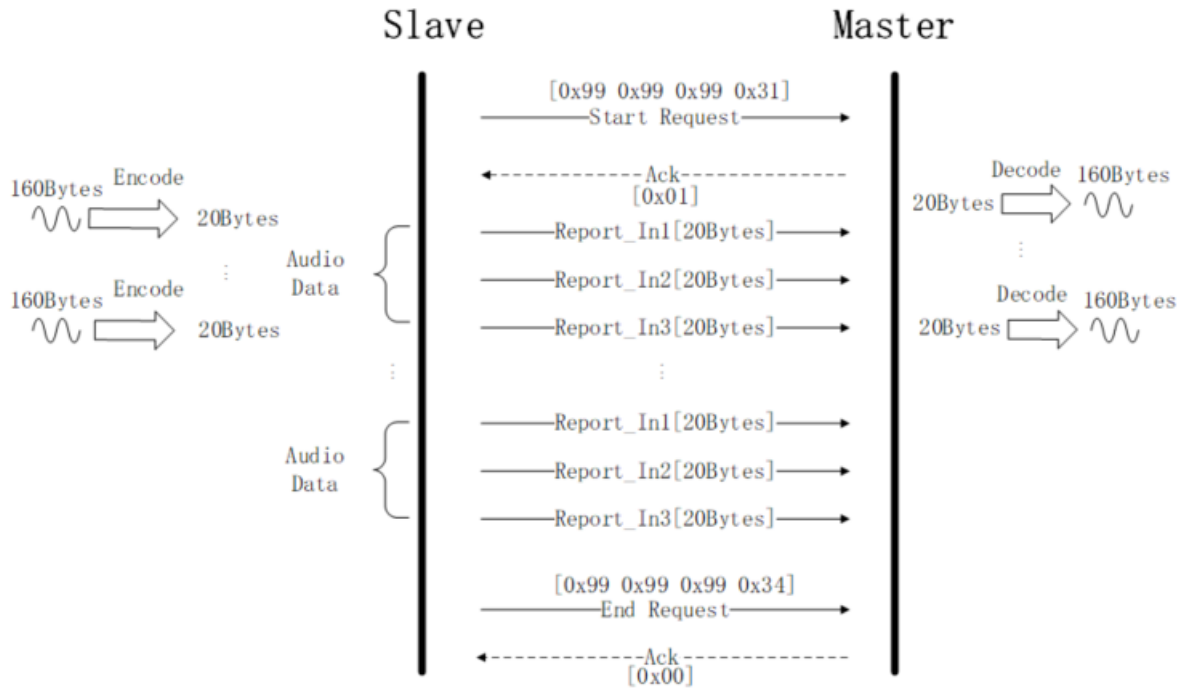


Figure 6.22: "Audio data interaction in SBC_HID_DONGLE_TO_STB mode"

At the beginning, the Slave sends start_request to the Master with the following packet.

ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE... 20 bytes (99 99 99 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00)
--	---

Figure 6.23: "Start_request packet"

After the Master receives the start_request, it sends the Ack packet as follows.

ATT Write Command Packet (Report: 1 byte)	Master: C4:19:D1:01:EA:FE... 1 byte (01)
---	--

Figure 6.24: "Ack packet"

Slave starts to send Audio voice data, voice data decompression and compression are operated in 160 bytes size, voice data is first compressed to 20 bytes by SBC compression algorithm, and then sent to Master end, each group of packet size is 20 bytes. In order to ensure the sequence of voice packets, use every three groups of packets for fixed handle value. The receiver end starts to decompress and restore the voice signal after each group of packets is completed. The packets are as follows.

ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (ED B8 77 67 66 7B 57 B5 83 58 29 BE 0C 31 B8 5B B5 B8 5B B5)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (BF B2 21 11 11 C3 6C 29 C3 1C 49 C4 1C 49 C3 1C 45 C5 1C 45)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (8A B2 11 11 10 C4 2C 43 C4 6C 3C C4 1C 52 C5 6C 49 C4 6C 46)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (BC B3 21 12 21 C3 1C 25 C2 5C 25 C3 5C 45 C5 5C 54 C5 5C 45)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (AE B3 22 11 11 C3 9C 24 C2 5C 21 C2 5C 35 C4 5C 55 C5 9C 55)
ATT Notification Packet (Report: 20 bytes)	Master: C4:19:D1:01:EA:FE <-> Slave: "testAudio" DD:DD:DD:DD:DD:DD	20 bytes (9F B2 22 21 22 C5 1C 45 C3 9C 45 C5 5C 45 C3 5C 31 C4 8C 48)

Figure 6.25: "Audio voice data of sbc decode"

At the end of the voice transmission, the Slave sends an End Request to the Master with the following packet.


 A blue horizontal bar representing a packet capture entry. On the left, there is a small icon of a document with a green checkmark. The text inside the bar reads: "ATT Notification Packet (Report: 20 bytes)" followed by "Master: C4:19:D1:01:EA..." and "20 bytes (99 99 99 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00)".

Figure 6.26: "End request packet"

The Master sends an Ack after receiving the End Request with the following packet.


 A blue horizontal bar representing a packet capture entry. On the left, there is a small icon of a document with a green checkmark. The text inside the bar reads: "ATT Write Command Packet (Report: 1 byte)" followed by "Master: C4:19:D1:01:EA..." and "1 byte (00)".

Figure 6.27: "Ack packet"

7 OTA

In order to realize the OTA function of the 8x5x BLE slave, a device is required as a BLE OTA master.

The OTA master can be a Bluetooth device actually used with the slave (you need to implement OTA in the APP), or you can use Telink's BLE master kma dongle. The following uses Telink's BLE master kma dongle as the ota master to introduce OTA in detail. The related code implementation can also be found in `feature_ota_big_pdu` under the Multi-Connection SDK.

8x5x supports Flash multi-address boot: In addition to the first address of Flash 0x00000, it also supports reading firmware from Flash high addresses 0x20000 (128K), 0x40000 (256K), 0x80000 (512K). This document uses high address 0x20000 as an example to introduce OTA.

7.1 Flash Architecture and OTA Procedure

7.1.1 FLASH Storage Architecture

When booting address is 0x20000, size of firmware compiled by the SDK should not exceed 128kB, i.e. the flash area 0-0x20000 serves to store firmware. If you're using boot address as 0x0 and 0x20000, the firmware size shouldn't be larger than 124K. if your firmware size is larger than 124K, then you would need to use 0x0 and 0x40000 as boot address, the firmware size shouldn't be larger than 252K. If more than 252K must be upgraded alternately using boot address 0 and 0x80000, the maximum firmware size must not exceed 508K.

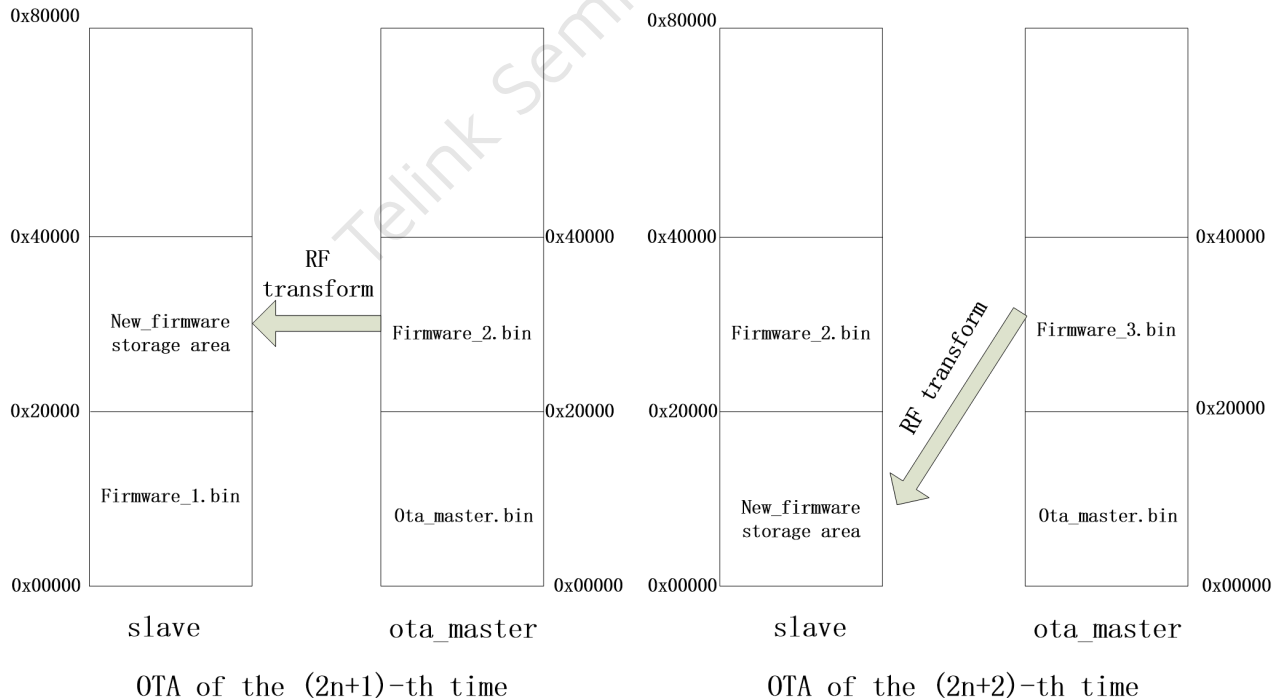


Figure 7.1: "Flash Storage Structure"

(1) OTA Master burns new firmware2 into the Master flash area from 0x20000 to 0x40000.

(2) OTA for the first time:

- When power on, Slave starts booting and executing firmware1 from flash 0~0x20000.
- When firmware1 is running, the area of Slave flash starting from 0x20000 (i.e. flash 0x20000~0x40000) is cleared during initialization and will be used as storage area for new firmware.
- OTA process starts, Master transfers firmware2 into Slave flash area from 0x20000 to 0x40000 via RF. Then slave reboot (Restart, similar to a power outage and power on again).

(3) For subsequent OTA updates, OTA Master first burns new firmware3 into the Master flash area from 0x20000 to 0x40000.

(4) OTA for the second time:

- When power on, Slave starts booting and executing firmware2 from flash 0x20000~0x40000.
- When firmware2 is running, the area of Slave flash starting from 0x0 (i.e. flash 0~0x20000) is cleared during initialization and will be used as storage area for new firmware.
- OTA process starts, Master transfers firmware3 into Slave flash area 0~0x20000 via RF. Then slave reboot (Restart, similar to a power outage and power on again).

(5) Subsequent OTA process repeats steps 1)~4): 1)~2) represents OTA of the $(2n+1)$ -th time, while 3)~4) represents OTA of the $(2n+2)$ -th time.

7.1.2 OTA Update Procedure

Based on the flash storage structure introduced, the OTA update procedure is illustrated as below:

First introduce the multi-address booting mechanism (only the first two booting addresses 0x00000 and 0x20000 will be introduced here): after MCU is powered on, it boots from address 0 by default. First, read the content of flash 0x08. If the value is 0x4b, the code starting from 0 are transferred to RAM, and the following instruction fetch address equals 0 plus PC pointer value; if the value of 0x08 is not 0x4b, the MCU directly reads the value of 0x20020, if the value is 0x4b, the MCU moves the code from 0x20000 to RAM, and all subsequent fetches start from the 0x20000 address, that is, the fetch address = 0x20000+PC pointer value.

So as long as you modify the value of the 0x08 and 0x20008 flag bits, you can specify which part of the FLASH code that the MCU executes.

The power-on and OTA process of a certain SDK ($2n+1$ or $2n+2$) is:

- (1) The MCU is powered on, and the values of 0x08 and 0x20008 are read and compared with 0x4b to determine the booting address, and then boots from the corresponding address and execute the code. This function is automatically completed by the MCU hardware.
- (2) During the program initialization process, read the MCU hardware register to determine which address the MCU boots from:

If boots from 0, set ota_program_offset to 0x20000, and erase all non-0xff content in the 0x20000 area to 0xff, which means that the new firmware obtained by the next OTA will be stored in the area starting at 0x20000;

If boots from 0x20000, set ota_program_offset to 0x0, and erase all the non-0xff content in the 0x0 area to 0xff, which means that the new firmware obtained by the next OTA will be stored in the area starting from 0x0.

- (3) Slave MCU executes the firmware after booting; OTA Master is powered on and establishes BLE connection with Slave.
- (4) Trigger OTA Master to enter OTA mode by UI (e.g. button press, write memory by PC tool, etc.). After entering OTA mode, OTA Master needs to obtain Handle value of Slave OTA Service Data Attribute (The handle value can be pre-appointed by Slave and Master, or obtained via "read_by_type".)
- (5) After the Attribute Handle value is obtained, OTA Master may need to obtain version number of current Slave Flash firmware, and compare it with the version number of local stored new firmware.

Note:

- If legacy protocol is used, user needs to implement the version number; if extend protocol is used, the operation related to version number acquisition has been implemented. For the difference between legacy and extend protocol, user can refer to section 7.2.2.
- (6) To enable OTA upgrade, OTA Master will send an OTA_start command to inform Slave to enter OTA mode.
 - (7) After the OTA_start command is received, Slave enters OTA mode and waits for OTA data to be sent from Master.
 - (8) Master reads the firmware stored in the flash area starting from 0x20000, and continuously sends OTA data to Slave until the entire firmware is sent.
 - (9) Slave receives OTA data and stores it in the area starting with ota_program_offset.
 - (10) After the master sends all the OTA data, check whether the data is received correctly by the slave (call the relevant function of the underlying BLE to determine whether the data of the link layer is correctly acknowledged).
 - (11) After the master confirms that all OTA data has been correctly received by the slave, it sends an OTA_END command.
 - (12) Slave receives the OTA_END command and writes the offset address of the new firmware area 0x08 (that is, ota_program_offset+0x08) as 0x4b, and writes the offset address of the old firmware storage area 0x08 as 0x00, which means it will Move code execution from the new area.
 - (13) Slave reports the results of OTA to master through Handle Value Notification.
 - (14) Reboot the slave, the new firmware takes effect.

During the whole OTA upgrade process, Slave will continuously check whether there's packet error, packet loss or timeout (A timer is started when OTA starts). Once packet error, packet loss or timeout is detected, Slave will determine the OTA process fails. Then Slave reboots, and executes the old firmware.

The OTA related operations on Slave side described above have been realized in the SDK and can be used by user directly. On Master side, extra firmware design is needed and it will be introduced later.

7.1.3 Modify FW Size and Booting Address

API `blc_ota_setNewFirmwareStorageAddress` supports modification of the boot address. Herein booting address means the address except 0 to store New_firmware, so it should be one of 0x20000, 0x40000 or 0x80000.

Table 7.1: Firmware size and boot address

Firmware_Boot_address	Firmware size (max)/K
0x20000	124
0x40000	252
0x80000	508

The default maximum firmware size in the SDK is 252K (due to some special reasons, the firmware size of the startup address 0x40000 must not be greater than 252K), and the corresponding startup addresses are 0x00000 and 0x40000. These two values are consistent with the previous description. User can call API `blc_ota_setNewFirmwareStorageAddress` to set the maximum firmware size.

```
ble_sts_t blc_ota_setNewFirmwareStorageAddress(multi_boot_addr_e new_fw_addr);
```

The parameter `multi_boot_addr_e` indicates the available boot addresses, including three.

```
typedef enum{
    MULTI_BOOT_ADDR_0x20000    = 0x20000,  //128 K
    MULTI_BOOT_ADDR_0x40000    = 0x40000,  //256 K
    MULTI_BOOT_ADDR_0x80000    = 0x80000,  //512 K
};
```

This API can only be called before `cpu_wakeup_init` in main function, otherwise it is invalid. The reason is that the `cpu_wakeup_init` function needs to do some settings according to the values of `firmware_size` and `boot_addr`.

7.2 RF Data Processing for OTA Mode

7.2.1 OTA Processing in Attribute Table

OTA related contents needs to be added in the Attribute Table on slave end. The "att_readwrite_callback_t r" and "att_readwrite_callback_t w" of the OTA data Attribute should be set as `otaRead` and `otaWrite`, respectively; the attribute should be set as `Read` and `Write_without_Rsp` (Telink Master KMA Dongle sends data via "Write Command" by default, with no need of ack from Slave to enable faster speed).

```
// OTA attribute values
static const u8 my_OtaCharVal[19] = {
    CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP | CHAR_PROP_NOTIFY,
    U16_LO(OTA_CMD_OUT_DP_H), U16_HI(OTA_CMD_OUT_DP_H),
    TELINK_SPP_DATA_OTA, };
    {4,ATT_PERMISSIONS_READ, 2,16,(u8*)&my_primaryServiceUUID),    (u8*)&my_OtaServiceUUID),
    ↪ 0},
    {0,ATT_PERMISSIONS_READ, 2, sizeof(my_OtaCharVal),(u8*)&my_characterUUID), (u8*)
    ↪ (my_OtaCharVal), 0},
    {0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)&my_OtaUUID), (&my_OtaData), &otaWrite,
    ↪ NULL},
    {0,ATT_PERMISSIONS_RDWR,2,sizeof(otaDataCCC),(u8*)&clientCharacterCfgUUID),    (u8*)
    ↪ (otaDataCCC), 0},
    {0,ATT_PERMISSIONS_READ, 2,sizeof (my_OtaName),(u8*)&userdesc_UUID), (u8*)(my_OtaName), 0},
```

When Master sends OTA data to Slave, it actually writes data to the second Attribute as shown above, so Master needs to know the Attribute Handle of this Attribute in the Attribute Table. To use the Attribute Handle value pre-appointed by Master and Slave, user can directly define it on Master side.

7.2.2 OTA Protocol

The current OTA architecture extends the functionality and is compatible with previous versions of the protocol. The entire OTA protocol consists of two parts: the Legacy protocol and the Extend protocol.

Table 7.2: OTA protocol

OTA Protocol	-
Legacy protocol	Extend protocol

Note:

- Functions supported by OTA protocol are:
 - OTA Result feedback function: this function is not optional, added by default.
 - Firmware Version Compare function and Big PDU function: This function is optional and can not be added, it should be noted that the version number comparison function is different in Legacy protocol and Extend protocol, please refer to the following OTA_CMD section for details.

The following introductions are all focused on Legacy and Extend protocols.

OTA_CMD composition

The PDUs of OTA's CMD are as follows.

Table 7.3: PDU of OTA's CMD

OTA Command Payload -	
Opcode (2 octet)	Cmd_data (0-18 octet)

Opcode

Table 7.4: Opcode of CMD

Opcode	Name	Use*
0xFF00	CMD_OTA_VERSION	Legacy
0xFF01	CMD_OTA_START	Legacy
0xFF02	CMD_OTA_END	All
0xFF03	CMD_OTA_START_EXT	Extend
0xFF04	CMD_OTA_FW_VERSION_REQ	Extend
0xFF05	CMD_OTA_FW_VERSION_RSP	Extend
0xFF06	CMD_OTA_RESULT	All

Note:

- Use: To identify the command use in Legacy protocol, Extend protocol or both of all
- Legacy: Only use in the Legacy protocol
- Extend: Only use in the Extend protocol
- All: use both in the Legacy protocol and Extend protocol

(1) CMD_OTA_VERSION

It is a command to get the current firmware version number of the slave, and the user can choose to use it if he adopts OTA Legacy protocol for OTA upgrade. It is Optional. This command can be used to pass the firmware version number through the callback function reserved on the slave end.

```
void b1c_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

The server side will trigger this callback function when it receives the CMD_OTA_VERSION command.

(2) CMD_OTA_START

This command is the OTA update start command. The master sends this command to the slave to officially start the OTA update. This command is only for Legacy Protocol, if user uses OTA Legacy protocol, this command must be used.

(3) CMD_OTA_END

This command is the end command, which is used by both legacy and extend protocol in OTA. When Master confirms all OTA data are correctly received by Slave, it will send this command, which can be followed by four valid bytes to re-confirm Slave has received all data from Master.

Table 7.5: End command of OTA

-	CMD_data	-
Adr_index_max (2 octets)	Adr_index_max_xor (2 octets)	Reserved (16 octets)

- Adr_index_max: the maximum adr_index value
- Adr_index_max_xor: the anomaly value of Adr_index_max for verification
- Reserved: Reserved for future function extension

(4) CMD_OTA_START_EXT

This command is the OTA update start command in the extend protocol. master sends this command to slave to officially start the OTA update. User must use this command as the start command if using OTA extend protocol.

Table 7.6: Packet structure of OTA_START_EXT

-	CMD_data	-
Length (1 octets)	Version_compare (1 octets)	Reserved (16 octets)

- Length: PDU length
- Version_compare: 0x01: enable version compare 0x00: disable version compare
- Reserved: Reserved for future extension

(5) CMD_OTA_FW_VERSION_REQ

This command is the version comparison request command in the OTA upgrade process. This command is initiated by client to Server side to request for version number and upgrade permission.

Table 7.7: Packet structure of OTA_FW_VERSION

-	CMD_data	-
version_num (2 octets)	version_compare (1 octets)	Reserved (16 octets)

- Version num: the firmware version number to be upgraded on the client side

- Version compare: 0x01: Enable version compare 0x00: Disable version compare
- Reserved: Reserved for future extensions

(6) CMD_OTA_FW_VERSION_RSP

This command is a version response command, the server side will compare the existing firmware version number with the version number requested by the client side after receiving the version comparison request command (CMD_OTA_FW_VERSION_REQ) from the client side to determine whether to upgrade, and the related information will be sent back to the client via this command.

Table 7.8: Response structure of OTA_FW_VERSION

-	CMD_data	-
version_num (2 octets)	version_accept (1 octets)	Reserved (16 octets)

- Version num: the firmware version number that Server side is currently running
- Version_accept: 0x01: accept client side upgrade request, 0x00: reject client side upgrade request
- Reserved: Reserved for future extensions

(7) CMD_OTA_RESULT

This command is the OTA result return command, the slave will send the result information to the master after the OTA is finished. In the whole OTA process, no matter success or failure, the OTA_result will only be reported once, the user can judge whether the upgrade is successful according to the returned result.

Table 7.9: OTA result return command structure

CMD_data	-
Result (1 octets)	Reserved (16 octets)

Result: OTA result information, all possible return results are shown in the following table.

Table 7.10: OTA return results

Value	Type	info
0x00	OTA_SUCCESS	success
0x01	OTA_DATA_PACKET_SEQ_ERR	OTA data packet sequence number error: repeated OTA PDU or lost some OTA PDU
0x02	OTA_PACKET_INVALID	invalid OTA packet: 1. invalid OTA command; 2. addr_index out of range; 3. not standard OTA PDU length
0x03	OTA_DATA_CRC_ERR	packet PDU CRC err

Value	Type	info
0x04	OTA_WRITE_FLASH_ERR	write OTA data to flash ERR
0x05	OTA_DATA_UNCOMPLETE	lost last one or more OTA PDU
0x06	OTA_FLOW_ERR	peer device send OTA command or OTA data not in correct flow
0x07	OTA_FW_CHECK_ERR	firmware CRC check error
0x08	OTA_VERSION_COMPARE_ERR	the version number to be update is lower than the current version
0x09	OTA_PDU_LEN_ERR	PDU length error: not 16*n, or not equal to the value it declare in "CMD_OTA_START_EXT" packet
0x0a	OTA_FIRMWARE_MARK_ERR	firmware mark error: not generated by telink's BLE SDK
0x0b	OTA_FW_SIZE_ERR	firmware size error: no firmware_size; firmware size too small or too big
0x0c	OTA_DATA_PACKET_TIMEOUT	time interval between two consequent packet exceed a value(user can adjust this value)
0x0d	OTA_TIMEOUT	OTA flow total timeout
0x0e	OTA_FAIL_DUE_TO_CONNECTION_TERMINANTE	OTA fail due to current connection terminate(maybe connection timeout or local/peer device terminate connection)
0x0f-0xff	Reserved for future use	/

Reserved: Reserved for future extensions

OTA Packet structure composition

When the Master sends commands and data to the Slave using WriteCommand or WriteResponse, the value of the Attribute Handle of the ATT layer is the handle_value of the OTA data on the slave side. According to the specification of the Ble Spec L2CAP layer regarding the PDU format, Attribute Value length is defined as the OTA_DataSize part in the following figure.

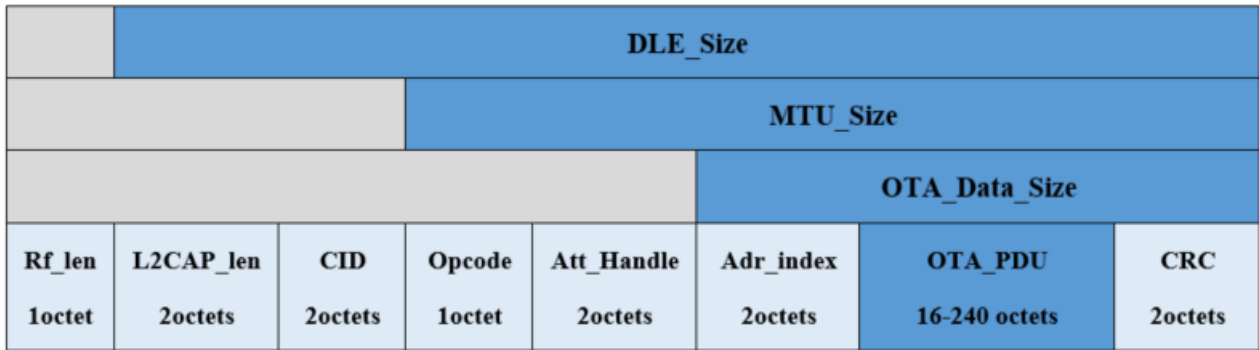


Figure 7.2: "OTA packet in L2CAP PDU"

- DLE Size: CID + Opcode + Att_Handle + Adr_index + OTA_PDU + CRC
- MTU_Size: Opcode + Att_Handle + Adr_index + OTA_PDU +CRC
- OTA_Data_Size: Adr_index + OTA_PDU + CRC

OTA_Data introduction

Table 7.11: OTA data

Type	Length
Default* + BigPDU*	16octets -240octets(n*16,n=1..15)

Note:

- Default: OTA PDU length fixed default size is 16 octets
- BigPDU: OTA PDU length can be changed in the range of 16octets - 240 octets, and is an integer multiple of 16 bytes

OTA_PDU Format

When user adopts Extend protocol in OTA and supports Big PDU, it can support long packet for OTA upgrade operation and reduce the time of OTA upgrade. User can customize the PDU size at the client side according to the need. The last two bytes are a CRC_16 calculation of the previous Adr_Index and Data to get the first CRC value, the slave will do the same CRC calculation after receiving the OTA data, and only when the CRC calculated by both matches, it will be considered a valid data.

Table 7.12: OTA PDU format

-	OTA PDU	-
Adr_Index (2 octets)	Data(n*16 octets) n=1..15	CRC (2 octets)

(1) PDU packet length: n=1

Data : 16 octets

Mapping of Adr_Index to Firmware address.

Table 7.13: Mapping of Adr_Index to firmware address when n=1

Adr_Index	Firmware_address
0x0001	0x0000 - 0x000F
0x0002	0x0010 - 0x001F
.....
XXXX	$(XXXX - 1) * 16 - (XXXX) * 16 + 15$

(2) PDU packet length: n=2

Data : 32 octets

Mapping of Adr_Index to Firmware address.

Table 7.14: Mapping of Adr_Index to firmware address when n=2

Adr_Index	Firmware_address
0x0001	0x0000 - 0x001F
0x0002	0x0020 - 0x003F
.....
XXXX	$(XXXX - 1) * 32 - (XXXX) * 32 + 31$

(3) PDU packet length: n=15

Data : 240 octets

Mapping of Adr_Index to Firmware address.

Table 7.15: Mapping of Adr_Index to firmware address when n=15

Adr_Index	Firmware_address
0x0001	0x0000 - 0x00EF
0x0002	0x0010 - 0x01DF
.....
XXXX	$(XXXX - 1) * 240 - (XXXX) * 240 + 239$

- In the OTA upgrade process, each packet of PDU length sent needs to be aligned with 16 bytes, that is, when the valid OTA data in the last packet is less than 16 bytes, the 0xFF data is added to make up the alignment, as listed below.

Note:

- The default PDU length of 16 octets is used, which does not involve the operation of DLE long packets.
- Firmware compare function is not selected.

The specific operation flow is shown in the following figure.

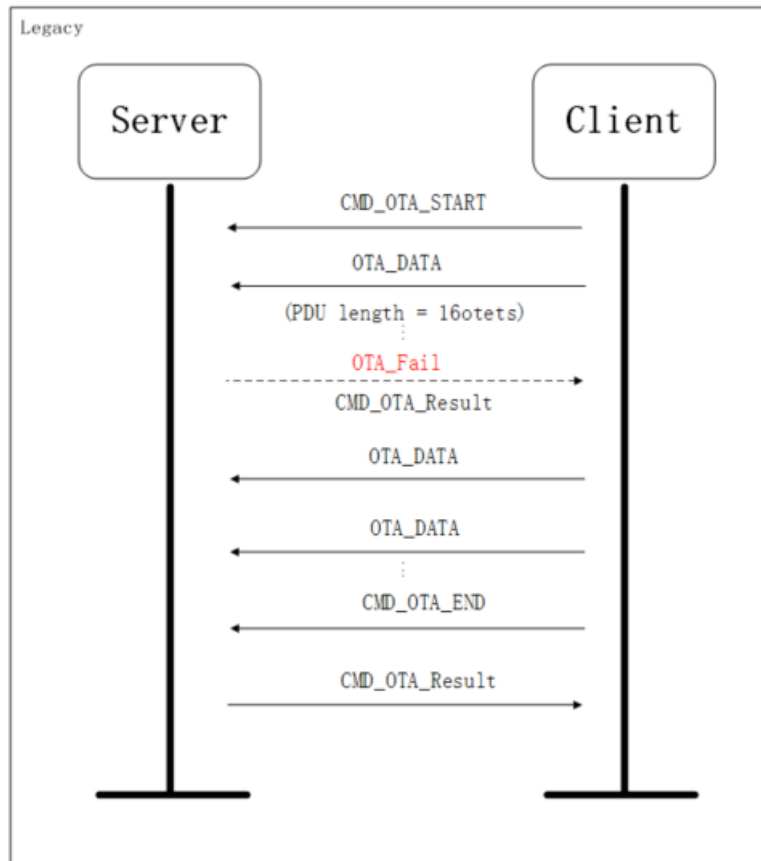


Figure 7.6: “OTA Legacy protocol process”

Client side will first send CMD_OTA_START command to Server side, Server side will start to prepare to receive OTA data after receiving the command, then Client side will start to send OTA_Data. If there is any interaction failure during the process, Server side will send CMD_OTA_Result to Client side, that is, return an error message and re-run the original program but will not enter reboot, the client side will stop the OTA data transfer when receiving this message. If the Client side and Server side successfully complete the OTA_Data transfer, the Client side will send CMD_OTA_END to the Server side, and the Server side will send CMD_OTA_Result to the Client side after receiving the result information, and enter reboot and run the new firmware.

OTA Extend Protocol Process

As mentioned above, there are some differences between the interaction commands of OTA Extend and Legacy introduced above. To better illustrate the whole interaction process between Slave and Master, the following example is used.

Note:

- PDU length adopts 64 octets size, which involves the operation of DLE long packets.
- Firmware compare function is not selected.

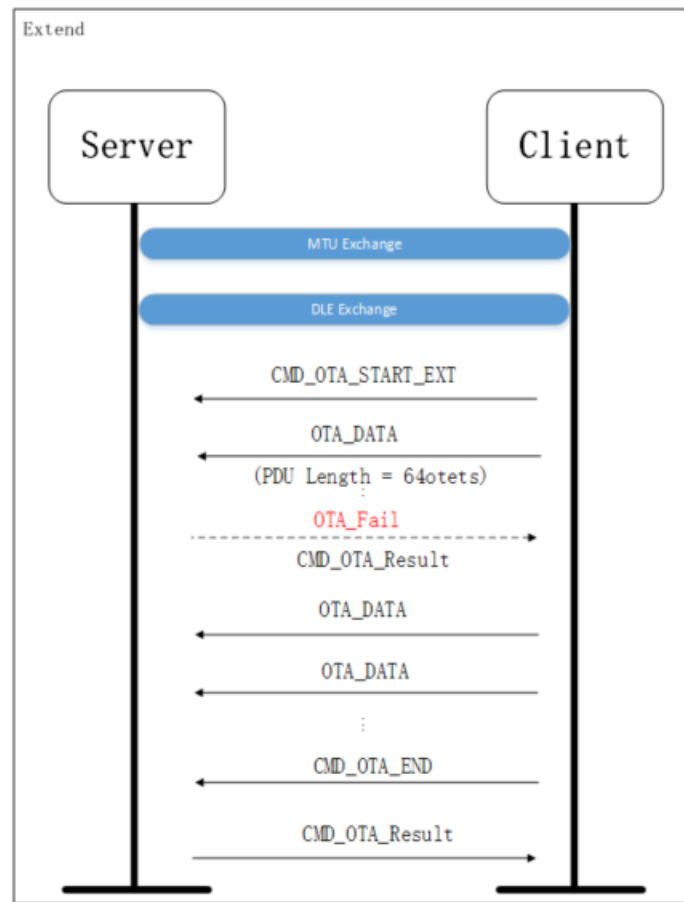


Figure 7.7: “OTA Extend protocol process”

Due to the DLE long packet function, the Client side first needs to interact with the Server side for MTU and DLE, then the next process is similar to the previous Legacy. The Client side sends CMD_OTA_START_EXT command to the Server side, the Server side starts to prepare to receive OTA data after receiving the command, then client side starts sending OTA_Data. If there is any interaction failure during the process, the Server side will send CMD_OTA_Result to the Client side, which returns the error message and re-runs the original program but will not enter reboot. If the Client side and Server side successfully complete the OTA_Data transfer, the Client side will send CMD_OTA_END to the Server side, and the Server side will send CMD_OTA_Result to the Client side after receiving the result information, and enter reboot and run the new firmware.

OTA Version Compare Process

In the Slave side, both Extend and Legacy Protocol have version comparison function, where Legacy reserved the interface, need to be implemented by the user, while Extend has implemented the version comparison function, the user can directly use, as follows, need to enable the following macro.

```
#define OTA_FW_VERSION_EXCHANGE_ENABLE    1    //user can change
#define OTA_FW_VERSION_COMPARE_ENABLE     1    //user can change
```

The following is an example of the interaction flow in Extend with version comparison.

Note:

- PDU length is 16 octets size, no operation of DLE long packet is involved.
- Firmware compare function selection (OTA to be upgraded version number is 0x0001, enable version compare enable)

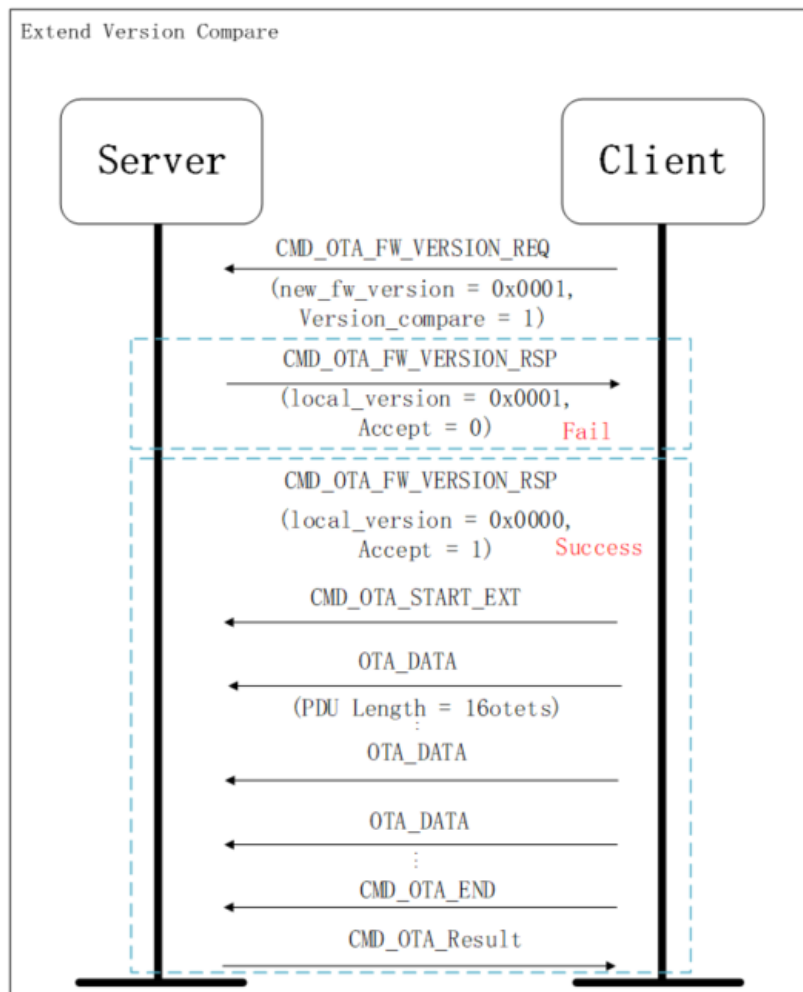


Figure 7.8: "OTA Version Compare Process"

After enabling the version comparison function, the Client side first sends the CMD_OTA_FW_VERSION_REQ version comparison request command to the Server side, where the PDU sent includes the Firmware version number of the Client side (new_fw_version = 0x0001), and the Server side gets the version number information of the Client side and compares it with the local version number (local_version).

If the received version number (new_fw_version = 0x0001) is not greater than the local version number (local version = 0x0001), the Server side will reject the Client side OTA upgrade request and send the Client

side version response command (CMD_OTA_FW_VERSION_RSP). The information sent includes the receiving parameter (accept = 0) and the local version number (local_version = 0x0001), and the Client will stop the OTA related operation after receiving it, that is, the current version upgrade is not successful.

If the received version number (new_fw_version = 0x0001) is larger than the local version number (local_version = 0x0000), the Server side will receive the OTA upgrade request from the Client side and send the version response command (CMD_OTA_FW_VERSION_RSP) to the Client side. The message sent includes the acceptance parameter (accept = 1) and the local version number (local_version = 0x0000), which the Client receives to start preparing the OTA upgrade related operations. The process is similar to the previous content, that is, first send the CMD_OTA_START command to the Server side, and then the Server side starts to prepare to receive the OTA data after receiving the command, client side starts sending OTA_data. If there is any interaction failure during the process, the Server side will send CMD_OTA_Result to the Client side, which will return the error message and re-run the original program but will not enter reboot, and the Client side will stop OTA data transmission immediately after receiving it. If the Client side and Server side successfully complete the OTA_Data transfer, the Client side will send CMD_OTA_END to the Server side, and the Server side will send CMD_OTA_Result to the Client side after receiving the result information, and enter reboot and run the new firmware.

OTA implementation

The above describes the entire OTA interaction process, the following example illustrates the specific data interaction between Master and Slave.

Note:

- OTA Protocol: Legacy Protocol
- The PDU length is 16 octets, which does not involve the operation of long DLE packets.
- The Master side enables Firmware compare function.

- (1) Check if there's any behavior to trigger entering OTA mode. If so, Master enters OTA mode.
- (2) To send OTA commands and data to Slave, Master needs to know the Attribute Handle value of current OTA data Attribute on Slave side. User can decide to directly use the pre-appointed value or obtain the Handle value via "Read By Type Request".

UUID of OTA data in Telink BLE SDK is always 16-byte value as shown below:

```
#define TELINK_SPP_DATA_OTA      {0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,  
↪ 0x03,0x02,0x01,0x00}
```

In "Read By Type Request" from Master, the "Type" is set as the 16-byte UUID. The Attribute Handle for the OTA UUID is available from "Read By Type Rsp" responded by Slave. In the figure below, the Attribute Handle value is shown as "0x0031".

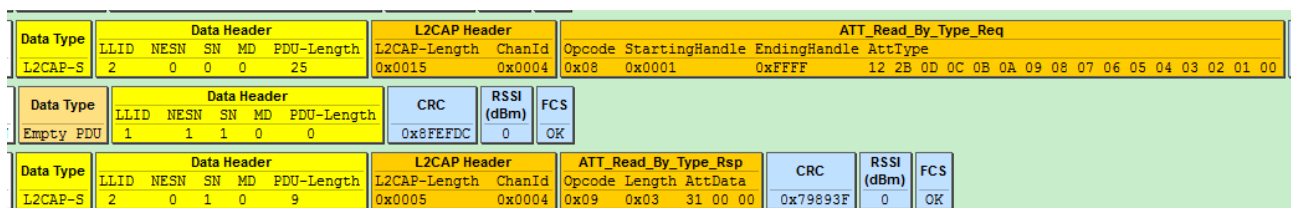


Figure 7.9: "Master Obtains OTA Attribute Handle via Read by Type Request"

- (3) Obtain the current firmware version number of the slave and decide whether to continue the OTA update (if the version is already the latest, no update is required). This step is for the user to choose whether to do it or not. The BLE SDK does not provide a specific version number acquisition method, users can play by themselves. In the current BLE SDK, legacy protocol does not implement version number transmission. The user can use write cmd or write response to send a request to obtain the OTA version to the slave through the OTA version cmd, but the slave side only provides a callback function when receiving the OTA version request, and the user finds a way to set the slave side in the callback function. The version number is sent to the master (such as manually sending a NOTIFY/INDICATE data).
- (4) Start a timing at the beginning of the OTA, and then continue to check whether the timing exceeds 30 seconds (this is only a reference time, and the actual evaluation will be made after the normal OTA required by the user test).

If it takes more than 30 seconds to consider the OTA timeout failure, because the slave side will check the CRC after receiving the OTA data. Once the CRC error or other errors (such as programming flash errors) occur, the OTA will be considered as a failure and the program will be restarted directly. The layer cannot ack the master, and the data on the master side has not been sent out, resulting in a timeout.

- (5) Read the four bytes of Master flash 0x20018-0x2001b to determine the size of the firmware.

This size is implemented by our compiler. Assuming the size of the firmware is 20k = 0x5000, then the value of 0x18-0x1b of the firmware is 0x00005000, so the size of the firmware can be read from 0x20018-0x2001b.

In the bin file shown in the figure below, the content of 0x18 ~ 0x1b is 0x0000cf94, so the size is 0xcf94 = 53140Bytes, from 0x0000 to 0xcf96.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	F3	22	20	34	11	A0	5D	02	63	84	02	00	6F	00	60	15
00000010	21	A8	00	00	00	00	00	00	94	CF	00	00	00	00	00	00
00000020	4B	4E	4C	54	00	00	3B	17	97	01	08	E0	93	81	81	7D
00000030	97	02	0A	E0	93	82	02	FD	16	81	97	02	00	E0	93	82

Figure 7.10: "Firmware Sample Starting Part"

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000CF60	00	00	E8	03	00	28	00	00	20	0A	00	05	2A	01	00	00
0000CF70	0A	48	0A	00	02	01	00	00	01	01	00	00	01	02	08	29
0000CF80	20	A1	0A	08	08	48	0A	09	05	2A	01	29	FF	FF	FF	FF
0000CF90	EC	6E	DD	A9												

Figure 7.11: "Firmware Sample Ending Part"

- (6) Master sends an OTA start command "0xff01" to Slave, so as to inform it to enter OTA mode and wait for OTA data from Master, as shown below.

Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x61875B	0	OK
	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF			

Figure 7.12: "OTA Start Sent From Master"

- (7) Read 16 bytes of firmware each time starting from Master flash 0x20000, assemble them into OTA data packet, set corresponding adr_index, calculate CRC value, and push the packet into TX FIFO, until all data of the firmware are sent to Slave.

The data sending method is described above, using the OTA data format: 20-byte valid data contains 2-byte adr_index, 16-byte firmware data and 2-byte CRC value to the former 18 bytes.

Note: If firmware data for the final transfer are less than 16 bytes, the remaining bytes should be complemented with "0xff" and need to be considered for CRC calculation.

Below illustrates how to assemble OTA data.

Data for first transfer: "adr_index" is "0x00 00", 16-byte data are values of addresses 0x0000 ~ 0x000f. Suppose CRC calculation result for the former 18 bytes is "0xXYZW", the 20-byte data should be:

0x00 0x00 0xf3 0x22 (12 bytes not listed)..... 0x60 0x15 0xZW 0xXY

Data for second transfer:

0x01 0x00 0x21 0xa8(12 bytes not listed)..... 0x00 0x00 0xJK 0xHI

Data for third transfer:

0x02 0x00 0x4b 0x4e(12 bytes not listed)..... 0x81 0x7d 0xNO 0xLM

.....

Data for penultimate transfer:

0xf8 0x0c 0x20 0xa1(12 bytes not listed)..... 0xff 0xff 0xST 0xPQ

Data for final transfer:

0xf9 0x0c 0xec 0x6e 0xdd 0xa9 0xff 0xff 0xff 0xff

0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xWX 0xUV

12 "0xff" are added to complement 16 bytes.

0xec 0x6e 0xdd 0xa9 is the third to sixth, which is the CRC_32 calculation result of the entire firmware bin. The slave will synchronously calculate the CRC_32 check value of the entire bin received during the OTA upgrade process, and compare it with 0xec 0x6e 0xdd 0xa9 at the end.

The CRC calculation result for a total of 18 bytes from 0xf9 to 0xff is 0xUVWX.

The above data is shown in the figure below:

```

+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 01 FF)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 00 00 F3 22 20 34 11 A0 5D 02 63 84 02 00 6F 00 60 15 7B 35)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 01 00 21 A8 00 00 00 00 00 00 94 CF 00 00 00 00 00 67 BD)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 02 00 4B 4E 4C 54 00 00 3B 17 97 01 08 E0 93 81 81 7D 74 D4)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 03 00 97 02 0A E0 93 82 02 FD 16 81 97 02 00 E0 93 82 59 5B)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 04 00 62 FC 73 90 02 80 99 62 F3 A2 02 30 73 10 30 00 63 36)

```

Figure 7.13: "Master OTA Data1"

```

+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: F8 0C 20 A1 0A 08 08 48 0A 09 05 2A 01 29 FF FF FF FF AA 24)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: F9 0C EC 6E DD A9 FF FF FF FF FF FF FF FF FF FF E3 DF)
+ ATT Write Command Packet (00010203-0405-0607-0809-0A0B0C0D2B12: 02 FF F9 0C 06 F3)

```

Figure 7.14: "Master OTA Data2"

- (8) After the firmware data is sent, check whether the data of the BLE link layer has been completely sent (because only when the data of the link layer is acked by the slave, the data is considered to be sent successfully). If it is completely sent, the master sends an ota_end command to notify the slave that all data has been sent.

The packet effective bytes of the OTA end are set to 6, the first two are 0xff02, and the middle two bytes are the maximum adr_index value of the new firmware (this is for the slave to confirm again that the last or several OTA data is not lost) , The last two bytes are the inverse of the largest adr_index value in the middle, which is equivalent to a simple check. OTA end does not require CRC check.

Take the bin shown in the above figure as an example, the largest adr_index is 0x0cf9, and its inverse value is 0xf306, and the final OTA end package is shown in the figure above.

- (9) Check if link-layer TX FIFO on Master side is empty: If it's empty, it indicates all data and commands in above steps are sent successfully, i.e. OTA task on Master succeeds.

Please refer to Appendix for CRC_16 calculation function.

As introduced above, Slave can directly invoke the otaWrite and otaRead in OTA Attribute. After Slave receives write command from Master, it will be parsed and processed automatically in BLE stack by invoking the otaWrite function.

In the otaWrite function, the 20-byte packet data will be parsed: first judge whether it's OTA CMD or OTA data, then process correspondingly (respond to OTA cmd; check CRC to OTA data and burn data into specific addresses of flash).

The OTA related operations on Slave side are shown as below:

- (1) OTA_FIRMWARE_VERSION command is received: Master requests to obtain Slave firmware version number.

In this BLE SDK, after Slave receives this command, it will only check whether related callback function is registered and determine whether to trigger the callback function correspondingly.

The interface in ble_ll_ota.h to register this callback function is shown as below:

```
typedef void (*ota_versionCb_t)(void);  
void blc_ota_registerOtaFirmwareVersionReqCb(ota_versionCb_t cb);
```

(2) OTA start command is received: Slave enters OTA mode.

If the "bls_ota_registerStartCmdCb" function is used to register the callback function of OTA start, then the callback function is executed to modify some parameter states after entering OTA mode (e.g. disable PM to stabilize OTA data transfer).

And the slave also starts and maintains a slave_adr_index to record the adr_index of the latest correct OTA data. The slave_adr_index is used to check whether there's packet loss in the whole OTA process, and its initial value is -1. Once packet loss is detected, OTA fails, Slave MCU exits OTA and reboots; since Master cannot receive any ack from Slave, it will discover OTA failure by software after timeout.

The following interface is used to register the callback function of OTA start:

```
typedef void (*ota_startCb_t)(void);  
void blc_ota_registerOtaStartCmdCb(ota_startCb_t cb);
```

User needs to register this callback function to carry out operations when OTA starts, for example, configure LED blinking to indicate OTA process.

After Slave receives "OTA start", it enters OTA and starts a timer (The timeout duration is set as 30s by default in current SDK). If OTA process is not finished within 30s, it's regarded as OTA failure due to timeout. User can evaluate firmware size (larger size takes more time) and BLE data bandwidth on Master (narrow bandwidth will influence OTA speed), and modify this timeout duration accordingly via the variable as shown below.

```
blotaSvr.process_timeout_us = 30 * 1000000;    //default 30s  
blotaSvr.packet_timeout_us = 5 * 1000000;     //default 5s
```

After initializing the variable, the user can call the following timeout function to perform the timeout process.

```
void blt_ota_procTimeout(void);
```

The other is the timeout period of the receive packet. It will be updated every time an OTA data packet is received. The timeout period is 5s, that is, if the next data is not received within 5s, the OTA_RF_PACKET_TIMEOUT is considered as a failure.

(3) Valid OTA data are received (first two bytes are 0~0x1000):

Whenever Slave receives one 20-byte OTA data packet, it will first check if the adr_index equals slave_adr_index plus 1. If not equal, it indicates packet loss and OTA failure; if equal, the slave_adr_index value is updated.

Then carry out CRC_16 check to the former 18 bytes. If not matched, OTA fails; if matched, the 16-byte valid data are written into corresponding flash area (ota_program_offset+adr_index*16 ~

ota_program_offset+adr_index16 + 15). During flash writing process, if there's any error, OTA also fails.

In order to ensure the integrity of the firmware after the OTA is completed, a CRC_32 check will be performed on the entire firmware at the end, and it will be compared with the check value calculated by the same method sent by the master. If it is not equal, it means there is a data error in the middle, and the OTA is considered a failure.

(4) "OTA end" command is received:

Check whether adr_max in OTA end packet and the inverted check value are correct. If yes, the adr_max can be used to double check whether maximum index value of data received by Slave from Master equals the adr_max in this packet. If equal, OTA succeeds; if not equal, OTA fails due to packet loss.

After successful OTA, Slave will set the booting flag of the old firmware address in flash as 0, set the booting flag of the new firmware address in flash as 0x4b, then reboot MCU.

(5) The slave sends the OTA result back to the master:

Once the OTA is started on the slave side, regardless of whether the OTA succeeds or fails, the slave will finally send the result to the master. The following is an example of the result information sent by the slave after the OTA is successful (the length is only 3 bytes):

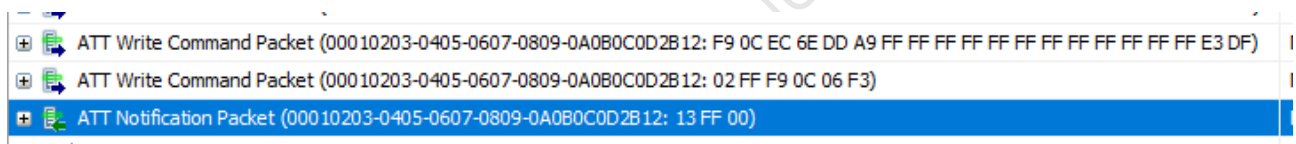


Figure 7.15: "Slave Sends OTA Success Result to Master"

(6) Slave supplies OTA state callback function:

After Slave starts OTA, MCU will finally reboot when OTA is successful.

If OTA succeeds, Slave will set flag before rebooting so that MCU executes the New_firmware.

If OTA fails, the incorrect new firmware will be erased before rebooting, so that MCU still executes the Old_firmware.

Before rebooting, user can judge whether the OTA state callback function is registered and determine whether to trigger it correspondingly.

The corresponding codes are as following:

```
void blc_ota_registerOtaResultIndicationCb (ota_resIndicateCb_t cb);
```

After the callback function is set, the enum of the parameter result of the callback function is the same as the result reported by the OTA. The first 0 is OTA success, and the rest are different reasons for failure.

The OTA upgrade success or failure will trigger the callback function, the actual code can be debugged by the result of the function to return parameters. When the OTA is unsuccessful, you can read the above result and stop the MCU with while(1) to understand what causes the OTA failure.

7.3 OTA Security

7.3.1 OTA Service data security

OTA Service is a kind of GATT service, and the problem of OTA service security protection is the problem of BLE GATT service data security protection, that is, data cannot be accessed illegally. According to the design of BLE Spec, user can use the following methods:

- (1) To enable SMP, it is recommended to use the Security Level as high as possible to achieve the function that only legally paired devices have access to OTA server data. Refer to the introduction of SMP in this document.

For example, using Security Mode 1 Level 3, the pairing of Authentication and MITM can effectively control the product slave device and the specific master to pair encryption success and back connection, the attacker cannot successfully encrypt with the slave device. Add the corresponding security level settings to the read and write of the protected GATT service data, and the attacker will not be able to access these data. If you use Mode 1 Level 4, Secure Connection + Authentication, the security level is even higher.

The codes that may be involved include the following:

```
typedef enum {
    LE_Security_Mode_1_Level_1 = BIT(0), No_Authentication_No_Encryption = BIT(0),
    ↪ No_Security = BIT(0),
    LE_Security_Mode_1_Level_2 = BIT(1), Unauthenticated_Pairing_with_Encryption = BIT(1),
    LE_Security_Mode_1_Level_3 = BIT(2), Authenticated_Pairing_with_Encryption = BIT(2),
    LE_Security_Mode_1_Level_4 = BIT(3),
    ↪ Authenticated_LE_Secure_Connection_Pairing_with_Encryption = BIT(3),
}le_security_mode_level_t;

#define ATT_PERMISSIONS_AUTHOR          0x10 //Attribute access(Read & Write) requires
    ↪ Authorization
#define ATT_PERMISSIONS_ENCRYPT          0x20 //Attribute access(Read & Write) requires
    ↪ Encryption
#define ATT_PERMISSIONS_AUTHEN          0x40 //Attribute access(Read & Write) requires
    ↪ Authentication(MITM protection)
#define ATT_PERMISSIONS_SECURE_CONN     0x80 //Attribute access(Read & Write) requires
    ↪ Secure_Connection
#define ATT_PERMISSIONS_SECURITY         (ATT_PERMISSIONS_AUTHOR | ATT_PERMISSIONS_ENCRYPT |
    ↪ ATT_PERMISSIONS_AUTHEN | ATT_PERMISSIONS_SECURE_CONN)
```

- (2) Use whitelist. Users can use the whitelist to connect only to the master device they want to connect to, or they can effectively intercept the attacker's connection.
- (3) Use address privacy protection, local device and peer device use resolvable private address (RPA), which effectively hides the identity address of the other party or us and makes the connection more secure.

7.3.2 OTA RF transmission data integrity

Since RF is an unstable transmission, a certain protection mechanism is needed to ensure the integrity and correctness of the firmware during the OTA process.

Refer to the previous introduction, the OTA master needs to divide the Firmware into multiple data packets according to a certain size in advance. The first 2 byte of each data packet is the packet sequence number, starting from 0 and increasing by 1.

7.3.2.1 LinkLayer data transfer mechanism

BLE Spec has been made corresponding design in terms of data transmission integrity:

- a) When the LinkLayer sender is transmitting a piece of data, it needs to see the other party's response before switching to the next packet of data transmission to ensure that the data transmission will not be lost;
- b) The LinkLayer receiver needs to check the packet sequence number of each piece of data, and the repeated data will be discarded to ensure that the data will not be received repeatedly;
- c) For each packet of data, the sender adds a CRC24 check value at the end, and the receiver recalculates the check value and compares it to eliminate the data with RF transmission errors.

Telink BLE SDK has passed the official Sig BQB certification and is implemented in full accordance with the above design.

These design mechanisms of LinkLayer can prevent data errors caused by RF transmission.

7.3.2.2 OTA PDU CRC16 check

Refer to the previous introduction, on the basis of LinkLayer data protection, add a CRC16 checksum to the OTA protocol to make data transmission more secure.

7.3.2.3 OTA PDU serial number check

The OTA master splits the Firmware into several OTA PDUs, each PDU has its own package serial number.

For the convenience of explanation, assume the Firmware size is 50K, split according to OTA PDU 16Byte, the number of PDUs is $50 \times 1024 / 16 = 3200$, then the serial number is 0 ~ 3199, i.e. 0x0 ~ 0xC7F.

After the OTA starts, set the expected serial number to 0. For each OTA data received, use the expected serial number and the actual serial number to compare, only when the two are equal, the process is considered correct and the expected serial number is updated +1. If the two are not equal, the process is considered failed and the OTA is ended. This design can ensure the continuity and uniqueness of OTA PDUs.

At the end of OTA, you can read the serial number 0xC7F of the last OTA PDU of Firmware on the OTA_END packet, and use this serial number to compare with the actual maximum serial number received, you can determine whether the OTA PDU loss a number. If the actual received maximum serial number is 0xC7E, it means the master missed the last packet, and the OTA will fail at this time.

The combination of the above designs can ensure that the OTA master splits the firmware correctly, and each OTA PDU is effectively sent out.

7.3.3 Firmware CRC32 check

There is a subjectively incorrect operation of the OTA master, which may cause the BLE product to be down. After generating a correct binary file with Telink compiler tool, the binary file may be accidentally modified due to operation error, for example, the content of a byte is tampered with. When this wrong binary file is taken to the OTA master for OTA upgrade, the master cannot know the error and will be used as the correct firmware to upgrade the slave, causing the program on the slave side to not run correctly.

To solve the above problems, Firmware CRC32 checksum is added in SDK. In the last step of compiling and generating the binary file, CRC32 calculation is performed on the binary file and the result is spliced in. During the OTA upgrade process, the server performs CRC32 calculation while receiving data. The OTA_END link uses the calculated value to compare with the value on the last OTA PDU. Only when the two are equal can the Firmware be considered to have not been tampered with.

7.3.4 OTA abnormal power failure protection

The Telink OTA design can ensure that the device is powered off at any time without the risk of device downtime.

Refer to the previous introduction in this chapter, the MCU uses a multi-address boot mechanism, and uses a byte marker for the MCU to determine which address the firmware starts from when the MCU is powered on. For the convenience of explanation, assume that the current firmware of the device is stored in the Flash 0x0 ~ 0x20000 range, where the value at address 0x0008 marks the Firmware as valid, which is 0x4B at this time; the new Firmware will be stored in the 0x20000 ~ 0x40000 range, where the value at address 0x20008 is used to mark the validity of the new Firmware, and the initial value is 0xFF.

After the OTA upgrade starts, the value on the 0x08 address of the first OTA PDU received is 0x4B, but this PDU will be intentionally written to 0x20008 as 0xFF when it is written to Flash, which is not effective. When all OTA processes are correct and all Flash writes are correct, the value of address 0x20008 will be written as 0x4B. Any power failure occurs at any time before this, it does not affect the Firmware of 0x0 ~ 0x20000, even if the power failure causes some Flash writes on 0x20000 ~ 0x40000 to fail, after re-powering, the Firmware of 0x0 ~ 0x20000 can be written to 0x40000. The Firmware of 0x20000 can be operated after re-powering.

In the last step, after confirming that the firmware on 0x20000 ~ 0x40000 is processed correctly, write 0x4B corresponding to the address 0x0008 in 0x0 ~ 0x20000 to 0x00, which means that the firmware on this area is invalid. We can write address 0x0008 to 0x00 as an atomic operation, no matter at which moment the power is turned off, this operation either succeeds (the value is changed to 0x00 or mistakenly written as a value other than 0x4B) or fails (i.e. 0x4B remains unchanged). If the operation succeeds, the next time the power is turned on, it will run the Firmware on 0x20000 ~ 0x40000; if it fails, it will run the Firmware on 0x0 ~ 0x20000.

8 Flash

8.1 Flash address allocation

The basic unit of FLASH storage information is the size of a sector (4K byte), because the flash erase is based on the sector (the erase function is `flash_erase_sector`). Theoretically the same kind of information needs to be stored in one sector, and different kinds of information need to be stored in different sectors (to prevent other types of information from being erased by mistake when erasing information). Therefore, it is recommended that users follow the principle of “different types of information in different sectors” when using FLASH to store customized information. The default location of system related information (Customized Value, MAC address, Pair&Sec Info) will be adaptively shifted to a later position of the flash according to the actual size of the flash.

The following figure shows the address allocation of various information in 512K/1M Flash. Take the default OTA Firmware maximum size not exceeding 128K as an example to illustrate, if the user modifies the OTA Firmware size.

Telink Semiconductor

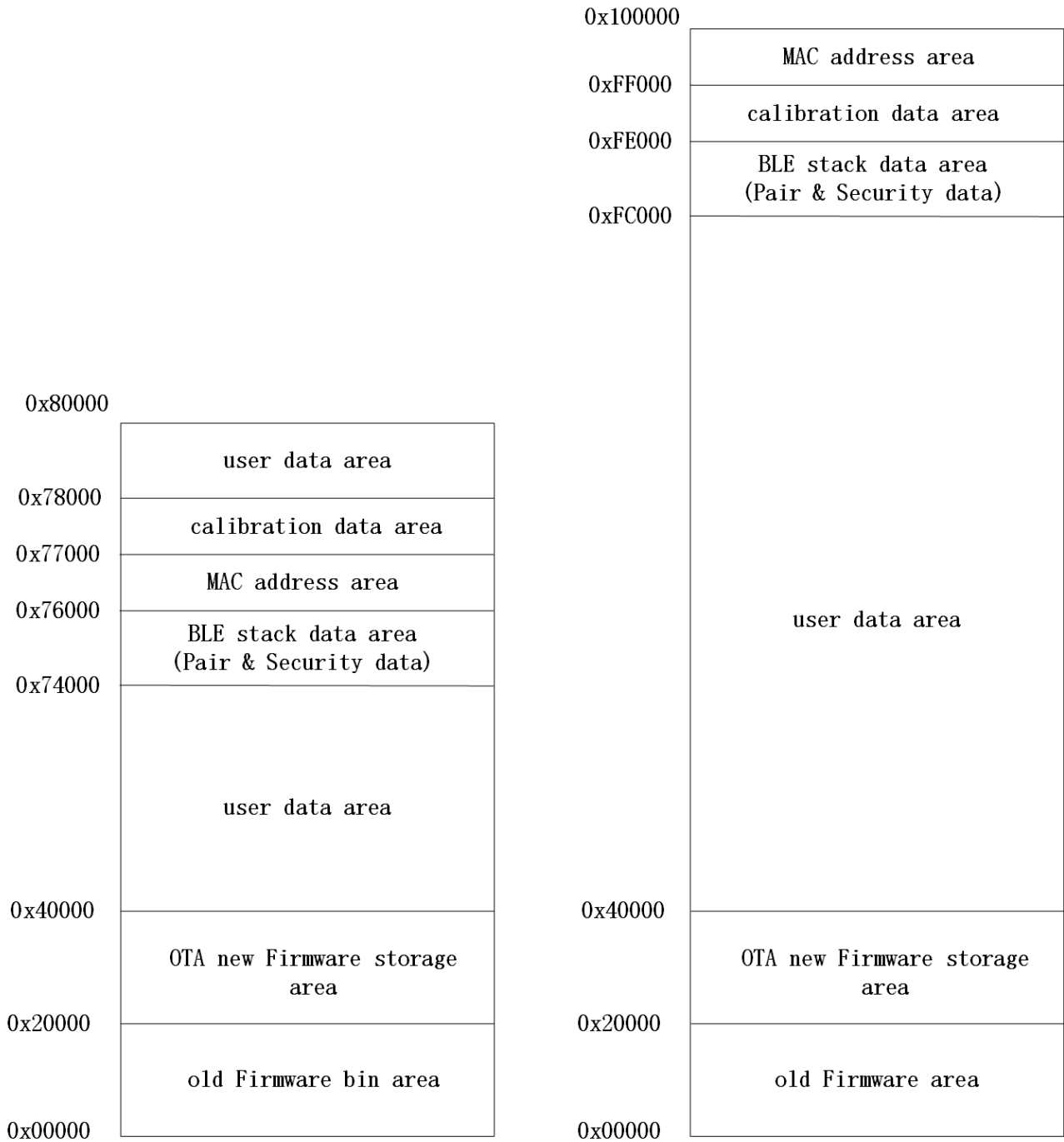


Figure 8.1: "512K/1M FLASH address allocation"

As shown in the figure above, all address assignments provide users with corresponding modification interfaces, and users can plan address assignments according to their needs. The following introduces the default address allocation method and the corresponding interface to modify the address. In actual applications, the corresponding Flash size can be determined by mid, and the corresponding application address space can be allocated according to the corresponding Flash size.

```

u8 flash_cap = temp_buf[2];
unsigned char adc_vref_calib_value_rd[4] = {0};

if(flash_cap == FLASH_SIZE_512K){
    flash_sector_mac_address = CFG_ADR_MAC_512K_FLASH;
    flash_sector_calibration = CFG_ADR_CALIBRATION_512K_FLASH;
}
else if(flash_cap == FLASH_SIZE_1M){
    flash_sector_mac_address = CFG_ADR_MAC_1M_FLASH;
    flash_sector_calibration = CFG_ADR_CALIBRATION_1M_FLASH;
}
else{
    //This SDK do not support flash size other than 512K/1M
    //If code stop here, please check your Flash
    while(1);
}

```

- (1) When using 512K Flash, the sector 0x76000~0x76FFF stores the MAC address. When using 1M Flash, it is 0xFF000~0x100000. In fact, the 6 bytes of MAC address are stored in 0x76000~0x76005 (0xFF000~0xFF005). In fact, the 6 bytes of MAC address are stored in 0x76000~0x76005 (0xFF000~0xFF005). When using 512K Flash, the high byte address is stored in 0x76005, and the low byte address is stored in 0x76000. For example, the contents of FLASH 0x76000 to 0x76005 are 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, then the MAC address is 0x665544332211. Telink's mass production fixture system will burn the actual product's MAC address to 0x76000 (0xFF000) address, which corresponds to the SDK. If the user needs to modify this address, please make sure that the address programmed by the fixture system is also modified accordingly. In the SDK, the MAC address will be read from the CFG_ADR_MAC of FLASH in the user_init function. This macro can be modified in stack/ble/blt_config.h.

```

/***** 512 K Flash *****/
#ifndef CFG_ADR_MAC_512K_FLASH
#define CFG_ADR_MAC_512K_FLASH 0x76000
#endif

/***** 1 M Flash *****/
#ifndef CFG_ADR_MAC_1M_FLASH
#define CFG_ADR_MAC_1M_FLASH 0xFF000
#endif

```

- (2) For the sector storage of 512K Flash 0x77000~0x77FFF, Telink MCU needs to calibrate the customized information, which is 0xFE000~0xFEFFF for 1M Flash. Only this part of the information does not follow the principle of "different types of information are placed in different sectors". Divide the 4096 bytes of this sector into different units for each 64 bytes, and each unit stores one type of calibration information. The calibration information can be placed in the same sector, because the calibration information is burned to the corresponding address during the fixture burning process. The actual firmware can only read the calibration information when it is running, and it is not allowed to write or erase it.

For details of calibration area information, please refer to "Telink_IC_Flash Customize Address_Spec". The calibration information is described with the offset address relative to the start address of the calibration area. For example, the offset address 0x00 refers to 0x77000 or 0xFE000.

- a) Offset address 0x00, 1 byte, stores the BLE RF frequency offset calibration value.
 - b) Offset address 0x40, 4 byte, stores the TP value calibration. B85m IC does not require TP calibration, which is ignored here.
 - c) Offset address 0xC0, stores the ADC Vref calibration value.
 - d) Offset address 0x180, 16 byte, stores the Firmware digital signature, which is used to prevent theft of the client Firmware.
 - e) Offset address 0x1C0, 2 byte, stores the Flash VDDF calibration value.
 - f) Others, reserved.
- (3) 512K Flash 0x74000 ~ 0x75FFF these two sectors are occupied by the BLE protocol stack system. For 1M Flash, they are 0xFC000~0xFDFFF, which are used to store pairing and encryption information. User can also modify the location of these two sectors. The size is fixed at two sectors 8K and cannot be modified. User can call the following function to modify the starting address of the paired encryption information storage:

```
void bls_smp_configPairingSecurityInfoStorageAddr (int addr);
```

- (4) Use 0x00000 ~ 0x3FFFF 256K space as program space by default;

0x00000 ~ 0x1FFFF total 128K is Firmware storage space; 0x20000 ~ 0x3FFFF 128K is the space to store new Firmware when OTA update, i.e. the maximum Firmware space supported is 128K.

If the default 128K program space is too large for the user and the user wants to free up some space in the 0x00000 ~ 0x3FFFF area for data storage, the protocol stack also provides the corresponding API, the modification method is described in the OTA chapter.

- (5) All the remaining flash space is used as data storage space for USER.

8.2 Flash operation

Flash space read/write operations use `flash_read_page` and `flash_write_page` functions, and flash erase uses `flash_erase_sector` function.

- (1) Flash read/write operations

Flash read/write operations use `flash_read_page` and `flash_write_page` functions.

```
void flash_read_page(u32 addr, u32 len, u8 *buf);  
void flash_write_page(u32 addr, u32 len, u8 *buf)
```

The `flash_read_page` function reads the contents of the flash:

```
void flash_read_page(u32 addr, u32 len, u8 *buf);
u8 data[6] = {0 };
flash_read_page(0x11000, 6, data); //Read 6 bytes starting from flash 0x11000 to the data array.
```

The flash_write_page function writes the flash:

```
flash_write_page(u32 addr, u32 len, u8 *buf);
u8 data[6] = {0x11,0x22,0x33,0x44,0x55,0x66 };
flash_write_page(0x12000, 6, data); //Write 0x665544332211 to the 6 byte starting at flash
↪ 0x12000.
```

The flash_write_page function is an operation on the page. A page in the flash is 256 byte. The maximum address size of this function operation is 256 byte, which cannot span two different page ranges.

When the operated address is the first address of a page, the maximum address is 256 byte, flash_write_page(0x12000, 256, data) is operated correctly, but flash_write_page(0x12000, 257, data) is wrong, because the last address does not belong to the page where 0x12000 is located anymore, the write operation will fail.

When the address being operated is not the first address of a page, pay more attention to the problem of cross-page. For example, flash_write_page (0x120f0, 20, data) is wrong. The first 16 addresses are in the page 0x12000, and the last 4 addresses are in the page 0x12100.

The flash_read_page does not have the cross-page problem mentioned above, and can read more than 256 bytes of data at one time.

Note:

- When using the flash_write_page function, user can only write up to 16 bytes at a time, more than that will cause a BLE interrupt exception.
- For the principle of this limitation, please refer to the introduction in the section "The Impact of Flash API on BLE Timing".

(2) flash erase operation

Use the flash_erase_sector function to erase the flash.

```
void flash_erase_sector(u32 addr);
```

A sector is 4096 byte, e.g. 0x13000 ~ 0x13fff is a complete sector.

addr must be the first address of a sector, and this function erases the entire sector each time.

It takes a long time to erase a sector. When the system clock is 16M, it takes about 30 ~ 100ms or even longer.

(3) Impact of flash read/write and erase operations on system interrupts

The three flash operation functions flash_read_page, flash_write_page, and flash_erase_sector introduced above must first turn off the system interrupt irq_disable() when executing, and then restore the interrupt

irq_restore() after the operation is completed, the purpose of which is to ensure the integrity and continuity of the flash MSPI timing operation, and to prevent the reentry of hardware resources caused by another flash operation calling the MSPI bus in the interrupt.

The timing of the BLE SDK RF receiving and sending packets is all controlled by interrupts. The consequence of turning off the system interrupt during the flash operation is that the timing of BLE receiving and sending packets will be destroyed and no timely response.

The execution time of the flash_read_page function is not too long and has little impact on the interruption of BLE. When using flash_write_page, it is recommended to write up to 16 Bytes at a time when BLE is connected, if it is too long, it may affect the BLE timing. Therefore, it is strongly recommended that users do not continuously read and write too long addresses in the main_loop when BLE is connected.

The execution time of the flash_erase_sector function is tens to hundreds of ms, so in the main_loop of the main program, once the BLE connection state is entered, it is not allowed to call the flash_erase_sector function, otherwise it will destroy the time point of BLE receiving and sending packets, causing the connection to be disconnected. If it is unavoidable to erase the flash when BLE is connected, please follow the Conn state Slave role timing protection implementation method described later in this document to operate.

(4) Use pointer access to read flash

The firmware of the BLE SDK is stored on the flash, and when the program runs, only the first part of the flash is placed on the ram for execution as resident memory code, and the vast majority of the remaining code is read from the flash to the ram cache area (cache for short) when needed according to the program's locality principle. MCU reads the content on the flash by automatically controlling the internal MSPI hardware module.

User can use the pointer form to read the content on the flash. The principle of the pointer form to read the flash is that when the MCU system bus accesses the data, when it finds that the data address is not on the resident memory ramcode, the system bus will automatically switch to the MSPI, and the four lines MSCN, MCLK, MSDI and MSDO will operate the timing of the spi to get to read the flash data.

List three examples below:

```
u16 x = *(volatile u16*)0x10000; //Read flash 0x10000 2 byte
u8 data[16];
memcpy(data, 0x20000, 16); //Read flash 0x20000 16 byte copy to data
if(!memcmp(data, 0x30000, 16)){ //Read flash 0x30000 16 bytes and compare with data
//.....
}
```

When reading the calibration value on the flash in user_init and setting it to the corresponding register, it is implemented by using pointers to access the flash. Please refer to the functions in the SDK.

```
static inline void blc_app_loadCustomizedParameters(void);
```

Read flash with pointer, but can't write flash with pointer (write flash can only be achieved by flash_write_page).

It should be noted that there is a problem with pointer read flash: as long as the data is read through the MCU system bus, the MCU will cache the data in the cache, if the data in the cache is not covered by other

content, and there is a new request to access the data, the MCU will directly use the cached content in the cache as the result. If the following situation occurs in the user's code.

```
u8 result;
result = *(volatile u16*)0x40000;    //Pointer read flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
result = *(volatile u16*)0x40000;    //Pointer read flash
if(result != 0x5A){ ..... }
```

The flash address 0x40000 was originally 0xff, the first read result is 0xff, then write 0x5A, theoretically the second read value is 0x5A, but the actual program gives the result is still 0xff, which was the first cache taken from the cache.

Note:

- If this happens when the same address is read multiple times and the value of this address will be rewritten, do not use the pointer form, use API flash_read_page to achieve the safest, this function reads the result without taking the previously cached value from the cache.

It is correct to implement it as follows:

```
u8 result;
flash_read_page(0x40000, 1, &result );    //API read flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
```

The position will be adaptively shifted to a later position of the flash according to the actual size of the flash.

8.3 Flash operation protection

Since the process of writing flash and erasing flash requires to transfer the address and data to flash through the SPI bus, the level stability on the SPI bus is very important. Any error in these critical data will cause irreversible consequences, such as writing the firmware wrong or erasing it by mistake will cause the firmware to no longer work and the OTA function will be disabled.

In the years of mass production experience of Telink chips, there have been errors caused by Flash operation under unstable conditions. Unstable conditions mainly include low power supply voltage, excessive power supply ripple, and intermittent power consumption of other modules on the system causing power supply jitter, and so on. In order to avoid similar operational risks in subsequent products, here we introduce some related Flash operation protection methods. After reading carefully, customers need to consider these issues as much as possible and add more security protection mechanisms to ensure product stability.

8.3.1 Low voltage detection protection

Combine with the introduction of low power protection chapter, it is necessary to consider doing voltage detection before all Flash write and erase operations to avoid the situation of operating Flash at too low

voltage. In addition, in order to ensure that the system is always working at a safe voltage, it is also recommended to do low voltage detection in main_loop at regular intervals to ensure the normal operation of the system.

Note:

- About flash low-voltage protection, the following many places appear 2.0V, 2.2V and other thresholds, emphasize that these values are only examples, reference values. Customers have to assess the actual situation to modify these thresholds, such as single-layer boards, power supply fluctuations and other factors, are to improve the safety threshold as appropriate.

Take the low voltage detection in the SDK demo as an example:

Step 1 First, when powering on or waking up from deepsleep, before calling the Flash function, a low voltage test must be performed to prevent flash problems caused by low voltage:

```
_attribute_ram_code_ int main (void)    //must run in ramcode
{
    ...
    if(!deepRetWakeUp){//read flash size
        user_init_battery_power_check(); //battery check must do before flash code
        blc_readFlashSize_autoConfigCustomFlashSector();
    }
    ...
    Main_loo();
}
```

Step 2 In the main_loop, low-voltage detection is required every 500ms:

```
if(battery_get_detect_enable() && clock_time_exceed(lowBattDet_tick, 500000) ){
    lowBattDet_tick = clock_time(); //Each detection to get the latest time, 500ms apart for a
    ↪ detection

    u8 analog_deep = analog_read(USED_DEEP_ANA_REG);
    u16 vbat_deep_thres = VBAT_DEEP_THRES_MV;
    u16 vbat_suspend_thres = VBAT_SUSPEND_THRES_MV;
    if(analog_deep & LOW_BATT_FLG){
        if(analog_deep & LOW_BATT_SUSPEND_FLG){//When the previous voltage has been lower than
            ↪ 1.8V
            vbat_deep_thres += 200;
            vbat_suspend_thres += 100;
        }
        else{//The previous voltage was between 1.8 - 2.0V
            vbat_deep_thres += 200;
        }
    }
    app_battery_power_check(vbat_deep_thres,vbat_suspend_thres);
}
```

Considering the working voltage of MCU and the working voltage of flash, if the Demo is set below 1.8V, the chip will directly enter suspend, and the chip below 1.8~2.0V will directly enter deepsleep, and once the chip is detected to be lower than 2.0V, it needs to wait until the voltage rises to 2.2V, the chip will resume normal operation. Consider the following points in this design:

- At 1.8V, there is a risk that the voltage is lower than the flash operating voltage. When you wake up after entering deepsleep, it may cause the flash to be abnormal and crash, so enter the suspend below 1.8V to ensure the safety of the chip;
- At 2.0V, when other modules are operated, the voltage may be pulled down and the flash will not work normally. Therefore, it is necessary to enter deepsleep below 2.0V to ensure that the chip no longer runs related modules;
- When there is a low voltage situation, need to restore to 2.2V in order to make other functions normal, this is to ensure that the power supply voltage is confirmed in the charge and has a certain amount of power, then start to restore the function can be safer.

The above is the timing detection voltage and management method in SDK Demo, users can refer to it for design.

Note:

- About flash low-voltage protection, the threshold values that appear above are only reference values. Customers have to assess the actual situation to modify these thresholds, such as single-layer boards, power supply fluctuations and other factors, are to improve the safety threshold as appropriate.

8.3.2 Flash lock protection

In addition to the above-mentioned timing voltage detection and management solutions, it is strongly recommended that customers do Flash erase and write protection. This is because in some cases, even if the low voltage detection result is safe, there is a small risk that the operation of each module in the application layer after the detection will cause the Flash power supply voltage to be pulled down, resulting in the Flash content being tampered when the Flash power supply voltage does not meet the conditions for real operation. Therefore, it is recommended that customers perform Flash erasing protection after the program is started, so that even if there is a misoperation, the content of the Flash will be more secure.

Generally, it is recommended that customers only write-protect the part of the program (the front part of Flash), so that the remaining Flash addresses can still be used for user-level data storage. Here we take the SDK Sample project as an example to describe how to calculate the protection size and the protection method.

8.3.2.1 Initialize write protection

- a) Calculate the protection size: Before initialization, first calculate the size of the flash address to be write-protected. The following figure is the list file compiled by 825x_ble_sample. According to the knowledge in Chapter 8.1, the bin area and the area where the OTA is located in the flash need to be protected, so it is low 256k.
- b) Call flash_read_mid to determine the flash type, call the related function according to the result, and pass in the corresponding parameters according to the size to be protected. Since the mid value read

back is 13325e, the functions to be used can be found in flash_mid13325e.c/ flash_mid13325e.h in the drivers directory. From the previous calculations, it is known that the required protection area is 256k lower, and the chip flash corresponds to the parameter of protecting the lower 256k, so the implementation method is as follows:

```
/**
 * @brief This function serves to lock the flash.
 * @param none.
 * @return none.
 */
__attribute__((ram_code)) void flash_lock(void)
{
    flash_lock_mid = flash_read_mid();
    #if (MCU_CORE_TYPE == MCU_CORE_825x)
    switch (flash_lock_mid)
    {
        case 0x13325e:
            flash_lock_mid13325e(FLASH_LOCK_LOW_256K_MID13325E);
            break;
        case 0x134051:
            flash_lock_mid134051(FLASH_LOCK_LOW_256K_MID134051);
            break;
    }
    #endif
}
```

Figure 8.2: "Write Protection by Flash Type"

8.3.2.2 Protection operations in the OTA process

In OTA, because the flash needs to be erased and written, if there is a write-protected operation when power is on, it needs to be unlocked and protected during the OTA process. Flash unlock protection can be performed in the OTA_START callback, the steps are as follows:

Step 1 First, register the callback function in the initialization function as follows:

```
blc_ota_registerOtaStartCmdCb (&flash_ota_start);
```

Step 2 In the callback function, call the corresponding function to unlock the protection according to the flash type obtained before power-on:

```
void flash_ota_start(void)
{
    switch (flash_lock_mid)
    {
        case 0x13325e:
            flash_unlock_mid13325e();
            break;

            ...
    }
}
```

After the OTA ends, regardless of success or failure, the program will be re-run. Therefore, at the beginning of the program, the program will be write-protected again by the flash_lock method described in the previous section to form a closed loop to ensure the security of the application.

8.4 Internal Flash introduction

8.4.1 Impact of Flash access timing on BLE timing

8.4.1.1 Flash access timing

(1) Flash Operation Basic Timing

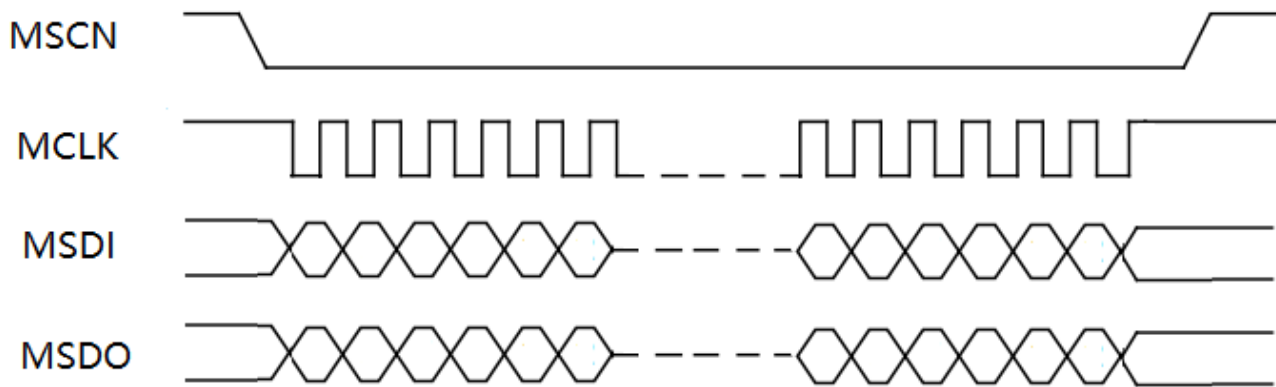


Figure 8.3: "Flash Operation Basic Timing"

The figure above shows a typical MCU accessing Flash timing. During MSCN pull-down, the data interaction with Flash is completed through the level state change of MSDI and MSDO under the control of MCLK.

Flash access timing is the basic timing of Flash operations. The period of MSCN being pulled down is data interaction, and it ends after being pulled up. All Flash functions are based on it, and complex Flash functions can be divided into several basic sequence of Flash operations.

The basic timing of each Flash operation is relatively independent, and the next round of operations can only be performed after one operation timing is completed.

(2) MCU hardware access to Flash

Firmware is stored in Flash, and the MCU execution program needs to read instructions and data from Flash in advance. Combining with the introduction of section 2.1.2.1, we can see that the content that needs to be read is the text segment and the "read only data" segment. The MCU reads the instructions on the Flash in real time during the running process, so it will start the basic sequence of the Flash operation continuously. This process is automatically controlled by the MCU hardware and the software does not participate.

If an interrupt occurs during the main_loop program, it enters irq_handler. Even if the programs in main_loop and irq_handler are both in the text segment, there will be no Flash timing conflict because it is done by the MCU hardware, which will do the relevant arbitration and control work.

(3) Software access to Flash

MCU hardware access to Flash only solves the problem of reading program instructions and "read only data". If you need to manually read, write, and erase the Flash, use the flash_read_page, flash_write_page, flash_erase_sector and other APIs in the flash driver. Looking at the specific implementation of these APIs, it can be seen that the software controls the basic timing of Flash operations, first pulling down MSCN, then reading and writing data, and finally pulling up MSCN.

(4) Flash access timing conflicts and solutions

Since the basic timing of Flash operation is an indivisible and destructive process, when software and MCU hardware access Flash at the same time, there is a possibility of timing conflicts because software and MCU hardware do not have coordination and arbitration mechanisms.

The scenario where this timing conflict occurs is that the software calls `flash_read_page`, `flash_write_page`, `flash_erase_sector` and other APIs in `main_loop`, and when the `MSCN` is pulled low and data is being read or written, an interrupt occurs and some instructions in the `irq_handler` are stored in the text segment, the MCU hardware also starts a new basic timing for Flash operation, and this timing conflicts with the previous timing in `main_loop`, causing errors such as MCU crash.

As shown in the figure below, when Software access to Flash ends, an interrupt occurs and responds, and MCU hardware starts to access Flash. At this time, the result of Flash access will inevitably be wrong.

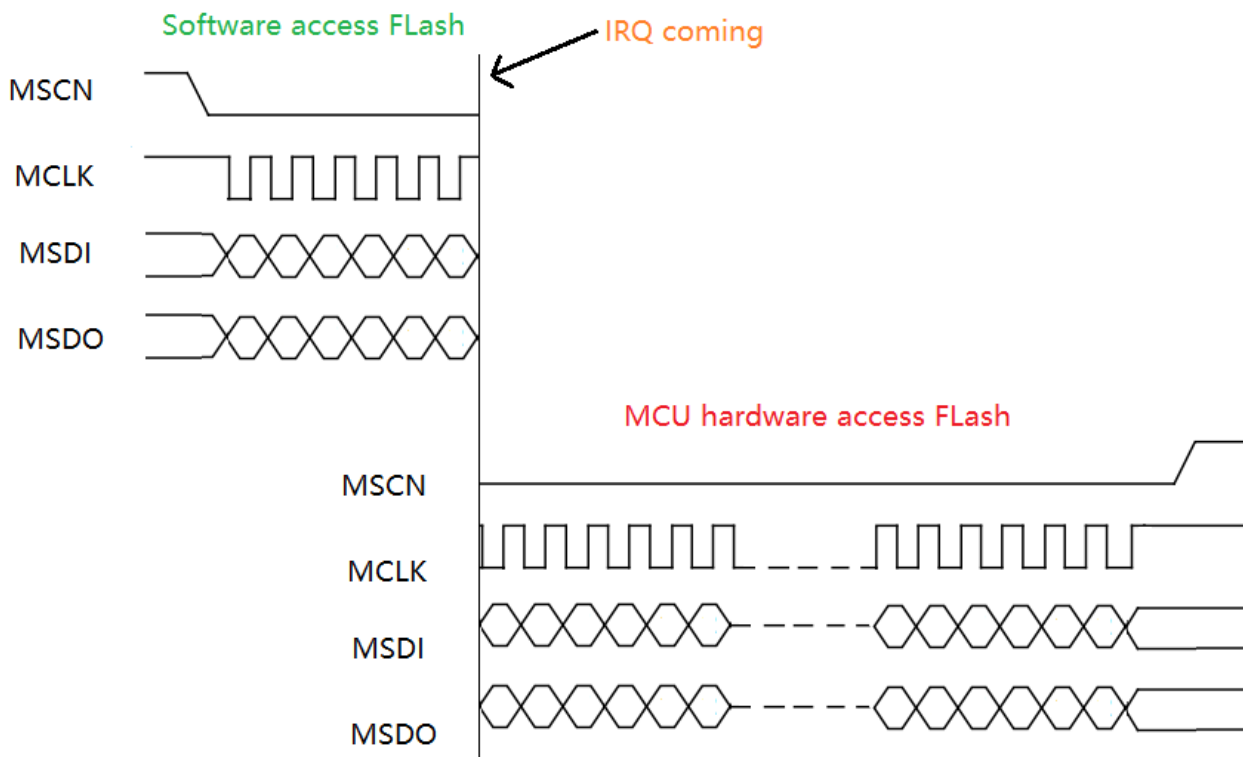


Figure 8.4: "Flash Timing Conflicts Caused by Interrupts"

Analyzing the conditions that must be met at the same time for timing conflicts, it can be concluded that the methods for resolving the conflicts include the following:

- Do not show any APIs for software manipulation of Flash timings in `main_loop`. this approach is not feasible and the use of APIs such as `flash_write_page` will appear on both the SDK and the application.
- All procedures in the `irq_handler` function are stored in the ramcode in advance, without relying on any text segment or "read_only_data" segment. This method is not good either. It is limited by the Sram size of 825x/827x chips, if all the interrupt codes are stored in ramcode, the Sram resources are not enough. In addition, it is not easy to control this restriction for users, and it is not possible to ensure that the user interrupt code is written so tightly.

- c) In the several APIs of the software operating Flash timing, add protection, close the interrupt, and prevent the irq_handler from responding. After the Flash access is over, the interrupt is resumed.

The Telink BLE SDK currently uses method 3, the Flash API to turn off interrupt protection. As shown in the code below (several codes are omitted in between), use irq_disable to turn off interrupts and irq_restore to restore them.

```
void flash_mspi_write_ram(unsigned char cmd, unsigned long addr, unsigned char addr_en, unsigned
↪ char *data, unsigned long data_len)
{
    unsigned char r = irq_disable();

    ..... //flash access

    irq_restore(r);
}
```

The following diagram shows the principle of turning off the interrupt to protect the Flash access timing. The interrupt is turned off when the software accesses Flash, and the interrupt occurs in the middle but does not respond immediately (interrupt wait). When the software accesses Flash timing is all finished correctly, the interrupt is turned back on, and the interrupt responds immediately at this time, and then the MCU hardware accesses Flash.

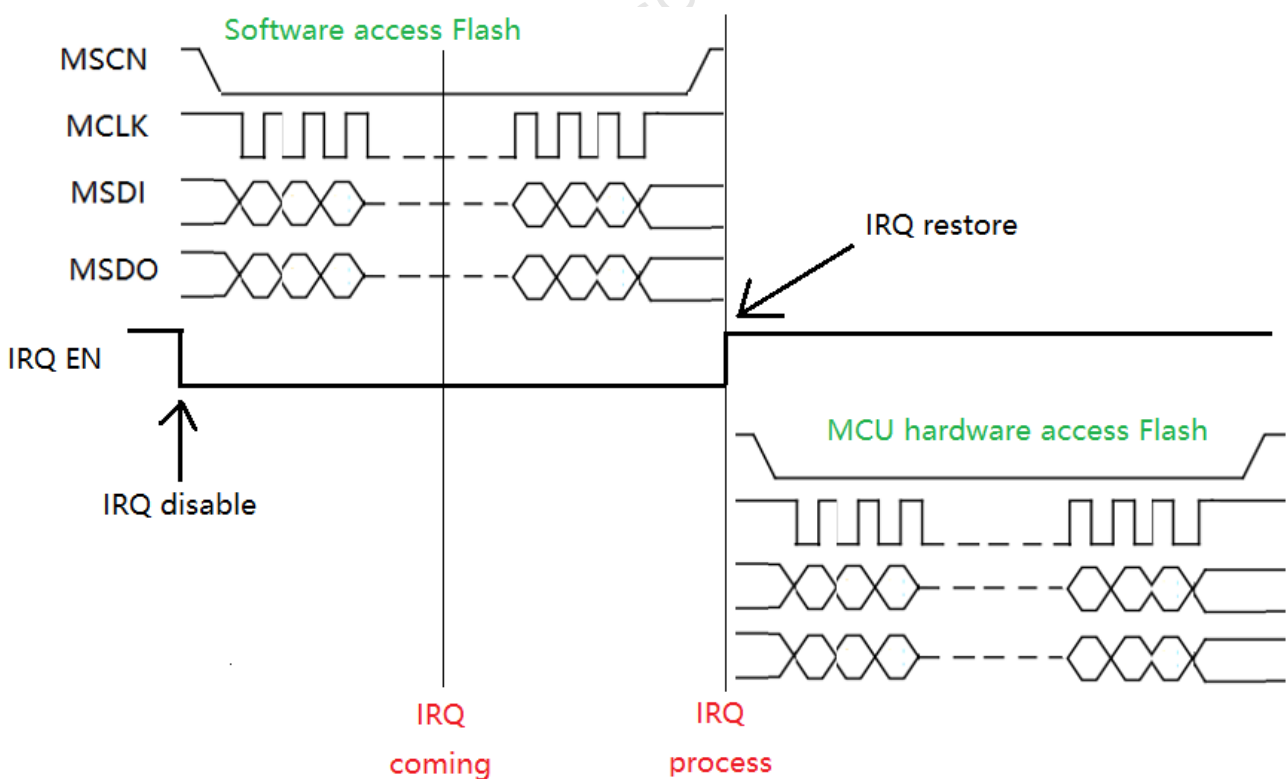


Figure 8.5: "Proper interrupt handling and flash operation"

8.4.1.2 Impact of Flash API on BLE timing

Previously introduced the use of Flash API to turn off the interrupt protection to solve the problem of timing conflicts between software and hardware MCU access to Flash. Since turning off the interrupt will make all interrupts unable to respond in real time, queuing to wait for the interrupt to resume and delay execution, you need to consider the possible side effects of the delayed time.

(1) Impact of off interrupt on BLE timing

Combine the characteristics of BLE timing to introduce. The BTX and BRX state machines in the BLE connection state in this SDK are all completed by interrupt tasks. BTX and BRX are similar implementations. Take BRX of slave role as an example.

The processing of BRX timing is more complicated. Take the processing of RX IRQ when more data appears in the BLE slave BRX as an example, as shown in the figure below. The SDK design requires the software to respond to every RX IRQ, which can be delayed but cannot be discarded. If a certain RX IRQ is lost, the RX packet that triggered this RX IRQ will also be lost, causing Linklayer packet loss errors.

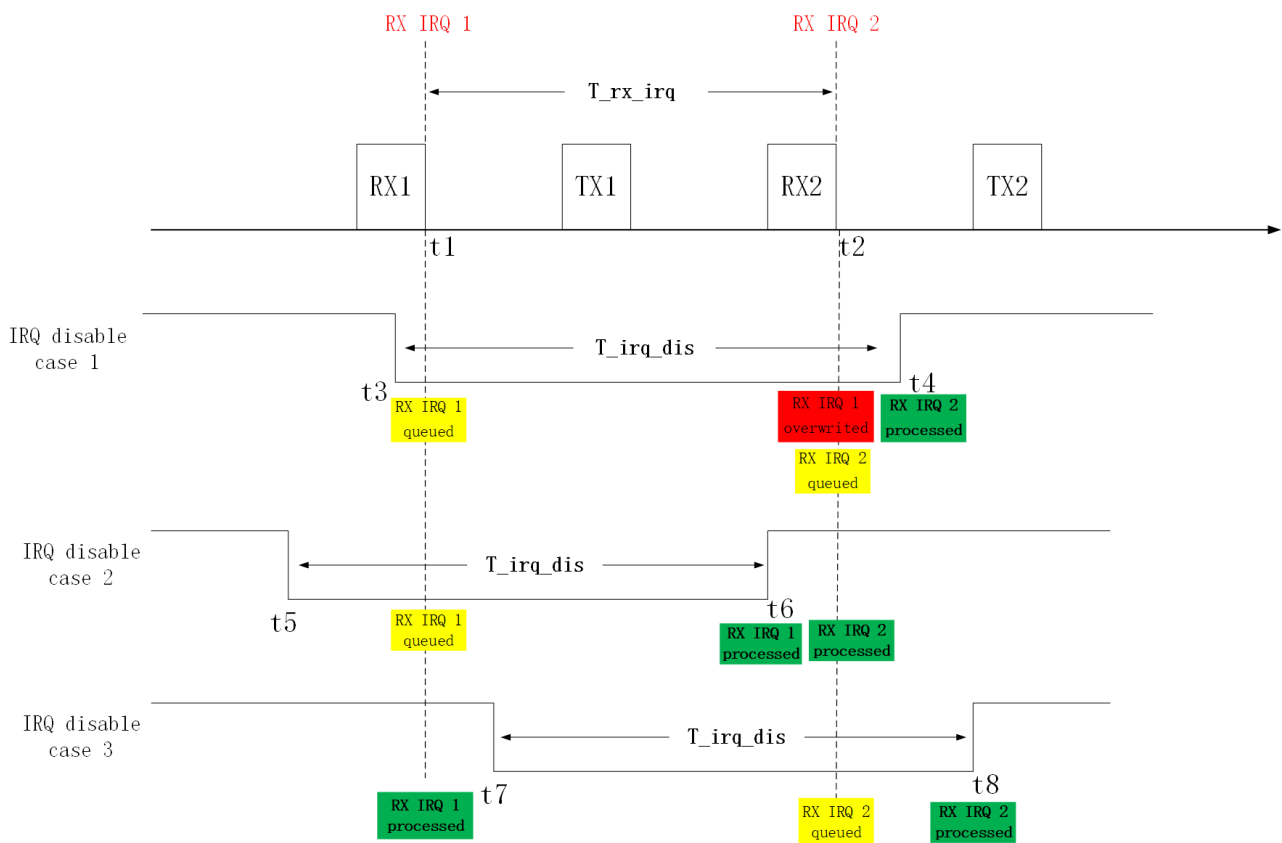


Figure 8.6: "Flash Operation on Link Layer Risk"

In the figure, RX1 triggers RX IRQ 1 at t1, and RX2 triggers RX IRQ 2 at t2. If no close interrupt occurs, the interrupt will respond in real time at t1 and t2, and the software correctly processes the RX packet.

The time difference between t1 and t2 is T_{rx_irq} , the off interrupt duration is T_{irq_dis} , $T_{irq_dis} > T_{rx_irq}$.

The off interrupt duration for all three cases IRQ disable case 1, IRQ disable case 2 and IRQ disable case 3 is $T_{\text{irq_dis}}$, but the starting point of the off interrupt and the relative time of t_1 are not the same.

IRQ disable case 1, t_3 turns off the interrupt, t_4 resumes the interrupt. $t_3 < t_1$; $t_4 > t_2$. RX IRQ 1 fails to respond at t_1 and the interrupt is queued. RX IRQ 2 is triggered at t_2 , overwriting RX IRQ1 (because there can be only one RX IRQ in the interrupt wait queue), and RX IRQ 2 is queued and executed correctly at t_4 . RX IRQ 1 corresponding to RX1 is lost and Linklayer errors out if RX1 is a valid packet.

IRQ disable case 2 and IRQ disable case 3, RX IRQ 1 and RX IRQ 2 are delayed, but they are not lost and no error occurs.

An important conclusion can be drawn from the analysis of the above examples:

When the interrupt closing duration is greater than a certain safety threshold, there may be a risk of Linklayer error.

This safety threshold is related to the timing design of the Linklayer in the SDK and the timing characteristics of the BLE Spec. The $T_{\text{rx_irq}}$ in the scale is much more complicated. The specific details will not be introduced in detail, and the safety threshold is directly given here as 220 μ s.

The same is the off-interrupt duration $T_{\text{irq_dis}}$. In the above example, IRQ disable case 2 and IRQ disable case 3 are different. Because the off interrupt occurs at different time points, RX IRQ 1 or RX IRQ2 will be delayed in response, and RX IRQ 2 will not overwrite packet loss caused by RX IRQ1. Even with IRQ disable case 1, if RX1 and RX2 are irrelevant empty packets, packet loss will not cause any errors.

When the interrupt off duration is greater than 220 μ s, it is not certain that an error will occur. Multiple conditions must be met at the same time to trigger an error. These conditions include: a long time to turn off the interrupt, and the time point of the RX IRQ occurrence match a specific relationship, more data appears in BTX or BRX, the two RX packets that continuously trigger RX IRQ are valid data packets rather than empty packets, and so on. So the final conclusion is:

There is a risk of linklayer errors when the interrupt off duration is greater than 220 μ s, and the probability is very low.

The design of BLE SDK Linklayer aims at zero risk, that is, the interrupt closing duration is always less than 220 μ s, and no chance of error is given.

Here is an additional introduction to the problem of RX packet loss in the above example. In the production of Telink BLE SDK, we often encounter this problem with customer feedback: under the premise that encryption is turned on, we see that the device sends a terminate packet with a reason of 0x3D (MIC_FAILURE), which leads to disconnection.

The above analysis shows that a long interrupt off time will cause the RX IRQ to be delayed for too long and then overwritten, and eventually lost packets. However, the SDK will handle the interrupt shutdown time correctly, which will be described in detail later in the document. The more likely reason is that the user uses other interrupts (such as Uart, USB, etc.), and the software execution time for these interrupts to respond is too long, which will have the same effect as the interrupt shutdown, and will also delay the RX IRQ. Here we limit the maximum safe time for a user interrupt execution to 100 μ s.

(2) Impact of Flash API off interrupt protection on BLE timing

In order to avoid the timing conflict between software access to Flash and MCU hardware access to Flash, the Flash API uses a method of turning off interrupts. When the interrupt closing duration is greater than 220 μ s, there is a risk of error in Linklayer. In order to solve the contradiction between the two, it is necessary to pay attention to the maximum time for the Flash API to close the interrupt.

The affected BLE timing is connection state slave role and master role. System initialization and Advertising state in mainloop are not affected. In the mainloop connection state, the following three Flash APIs are mainly concerned: `flash_read_page`, `flash_write_page`, `flash_erase_sector`. Other Flash APIs are generally not used or used during initialization.

a) `flash_read_page`

It has been tested and verified that when the number of bytes read by `flash_read_page` at a time does not exceed 64, the time is very safe, within 220 μ s. After this value is exceeded, there will be a certain risk.

It is strongly recommended that users read up to 64 bytes when using `flash_read_page` to read Flash. If it exceeds 64 bytes, it needs to be split into multiple calls to `flash_read_page` to achieve.

b) `flash_erase_sector`

The time of `flash_erase_sector` is generally in the order of 10ms ~ 100ms, which is far more than 220 μ s. So this SDK requires users not to call `flash_erase_sector` in the BLE connection state. If you call this API directly, the connection will definitely go wrong.

We recommend that users use other methods to replace the design of `flash_erase_sector`. For example, some applications are designed to repeatedly update some key information stored in Flash. In the design, you can consider selecting a larger area and using `flash_write_page` to continuously extend back.

For BLE slave applications, if the unavoidable `flash_erase_sector` occurs occasionally, you can use the Conn state Slave role timing protection mechanism to avoid it. Please refer to the details of this document.

Note that because the timing protection mechanism is very complicated, it is not recommended to use the high-frequency `flash_erase_sector` as it cannot guarantee the stability of the mechanism of repeated connection calls when the BLE slave is connected. It is recommended that users avoid this situation as much as possible by design.

c) `flash_write_page`

The `flash_write_page` time is affected by many key factors, including: Flash type, Flash technology, write byte number, high and low temperature, etc. The following is a detailed description from several types of internal Flash.

8.4.2 Use of internal Flash API

According to the previous section, `flash_write_page` in Flash API is related to the type of internal Flash. This section describes in detail with several kinds of internal Flash already supported by 825x and 827x.

Note:

- PUYA Flash will not be mass-produced on the 825x and 827x for the time being, but it retains the introduction of related compatibility principles to deepen users' understanding of BLE timing.

8.4.2.1 GD Flash

GD Flash belongs to the ETOX process and is the earliest internal Flash supported by 825x and 827x.

The consumption time of `flash_write_page` is related to the parameter `len` (i.e. the number of bytes written at once), which is close to a positive relationship. After detailed testing and analysis within Telink, it is found

that when the number of bytes is less than or equal to 16, the writing time can be stabilized within 220us; if the number of bytes exceeds 16, there will be risks.

For GD Flash, the maximum number of bytes written by `flash_write_page` is required to be 16. If it exceeds 16, such as 32, it can be split into two and write 16 bytes.

In the SDK design, there are two places involving `flash_write_page`. One is SMP storage configuration information, which uses 16 bytes per write; the other is when OTA writes new firmware, it also uses 16 bytes per write. In the design of OTA long package, for example, each package of 240 bytes of valid data is divided into 15 writes ($16 \times 15 = 240$).

It is strongly recommended that customers use `flash_write_page` to write at most 16 bytes each time, otherwise there will be a risk of conflict with BLE timing.

8.4.2.2 Zbit Flash

Zbit Flash is ETOX process, `flash_write_page` time is similar to GD Flash, and the maximum number of bytes written is limited to 16.

However, due to some characteristics of Zbit Flash itself, when the temperature rises, `flash_write_page` consumes longer time. For this characteristic, the BLE SDK handles it as follows:

- (1) For products with high-temperature application scenarios, the operations department will not send customers chips with internal Zbit Flash. Please note this point as well.
- (2) The SDK increases the power supply voltage of Flash where `flash_write_page` is involved (SMP stores pairing information, OTA writes new firmware), which can prevent Zbit Flash write time from being too long.

The measures in point 2 above are very important. Products that use Zbit Flash must ensure that the above measures have taken effect.

The corresponding B85m single-connection SDK of the Handbook (including 825x and 827x series chips) has added this measure to support Zbit Flash.

For several important historical versions of the historical 825x/827x BLE single connection SDK, the Telink BLE Team released related patches to support Zbit Flash. Customers must update the patch to ensure risk-free production.

8.4.2.3 PUYA Flash

The internal PUYA Flash added on 825x and 827x is Sonos process, `flash_write_page` time law and ETOX process is very different.

Note:

- PUYA Flash will not be mass-produced on the 825x and 827x for the time being, but it retains the introduction of related compatibility principles to deepen users' understanding of BLE timing.

The flash with Sonos process does not support byte programming, only page programming.

For example, call `flash_write_page` to write a byte value of 0x5A to the address 0x1000. the internal practice of Flash is to first read all the contents of the page corresponding to 0x1000 (0x1000 ~ 0x10FF

total 256 byte) out of the cache, then modify the first byte on the cache to 0x5A, and finally write all the values of the entire page cache to the page 0x1000.

The problem caused by this mechanism is that whether the number of bytes written is 1, 2, 4, 8, or 200, 255, etc., the time consumed is similar to that of writing a page 256 byte, which takes about 2ms. But as we said before, if the interrupt time exceeds 220µs, the linklayer is at risk of error.

Since the Sonos process Flash write time of 2ms is much larger than the safety threshold of 220µs, it needs to be solved by doing circumvention solutions in the Linklayer design.

When users need to write Flash, they must call the `flash_write_page` function. `flash_write_page` is a function in the old version of the SDK, and a pointer in the new version of the SDK. This pointer points to the `flash_write_data` function by default, which is aimed at the implementation of `flash_write_page` of GD and Zbit Flash.

BLE SDK detects the Flash type during initialization, and if it is found to be Sonos process Flash, it will modify the `flash_write_page` pointer to another special function that interacts with Linklayer's timing design to actively avoid the BTX, BRX timings, so that the write flash action never coincides with BTX and BRX in time. This design is not visible to the user in the underlying SDK implementation, so the user can use `flash_write_page` without worry.

The following is an example to explain BRX as an example, BTX principle is similar. The simplified model is shown in the figure below.

Assume that the execution time of `flash_write_page` is the standard 2ms (the actual time is not so standard, it is far more complicated than it, and the SDK internally deals with time fluctuations).

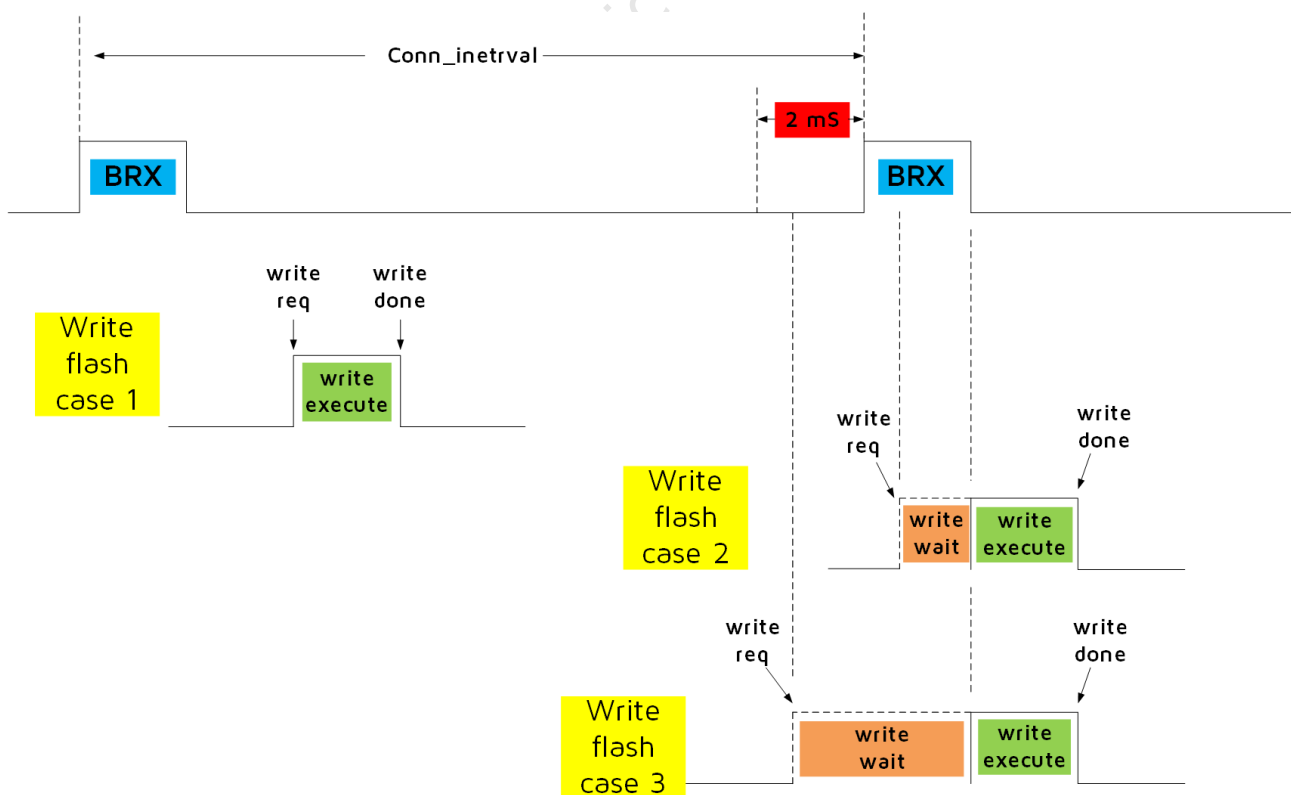


Figure 8.7: "Sonos Process Flash Action Before and After The Reception Window"

In the figure, write_req refers to calling the flash_write_page function in the software, write_wait refers to a safe time point when the BRX state machine is idle (a while loop is used in the software to read), write_execute is to call the basic timing of the Flash operation to complete the flash write. Write_done indicates a successful write. The total time consumed by flash_write_pag as seen by the application layer is the schedule from write_req to write_done.

Write flash case 1, write_req occurs at a very safe time outside of BRX, execute write_execute directly. The flash_write_pag time seen by the application layer is 2ms.

Write flash case 2, write_req is in the event of BRX, if write_execute at this time, BRX timing may be broken and packet loss may occur, so execute write_wait first, and then write_execute after BRX ends. The flash_write_pag time seen by the application layer is the write_wait time plus the write_execute time (2ms), and the write wait time is related to the actual running time of the BRX.

Write flash case 3. Although write_req is not in BRX timing, because there is a BRX that will respond within 2ms in the future, if write_execute directly consumes 2ms, the BRX start time will be postponed, causing the BLE slave to receive the packet time error. Therefore, it is considered that write_req is an unsafe point in time, execute write_wait until the end of BRX and then write_execute. The flash_write_pag time seen by the application layer is the write_wait time plus the write_execute time (2ms), and the write wait time is related to the actual running time of the BRX.

From the above introduction, when using the flash_write_page function on PUYA Flash, the write flash action is not executed immediately, and it may take a short time to wait before the write action is executed. If flash_write_page is called frequently and frequently, the efficiency of the program may decrease, which is a side effect. The wasted waiting time is the time consumed by write_wait. The figure below is an example of flash_write_page time in an extreme case. At this time, there is a large amount of more data in BRX, which leads to a very long BRX duration.

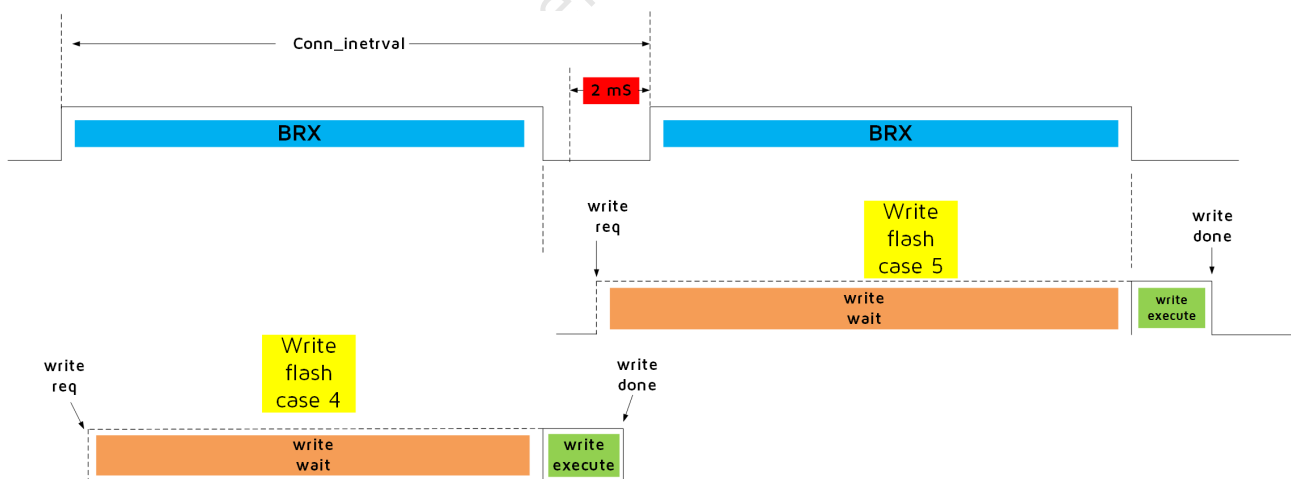


Figure 8.8: "Flash write action delay due to more data"

9 Key Scan

Telink provides a keyscan architecture based on row/column scan to detect and process key state update (press/release). User can directly use the demo code, or realize the function by developing his own code.

9.1 Key Matrix

Figure shows a 5*6 Key matrix which supports up to 30 buttons. Five drive pins (Row0-Row4) serve to output drive level, while six scan pins (CoL0-CoL5) serve to scan for key press in current column.

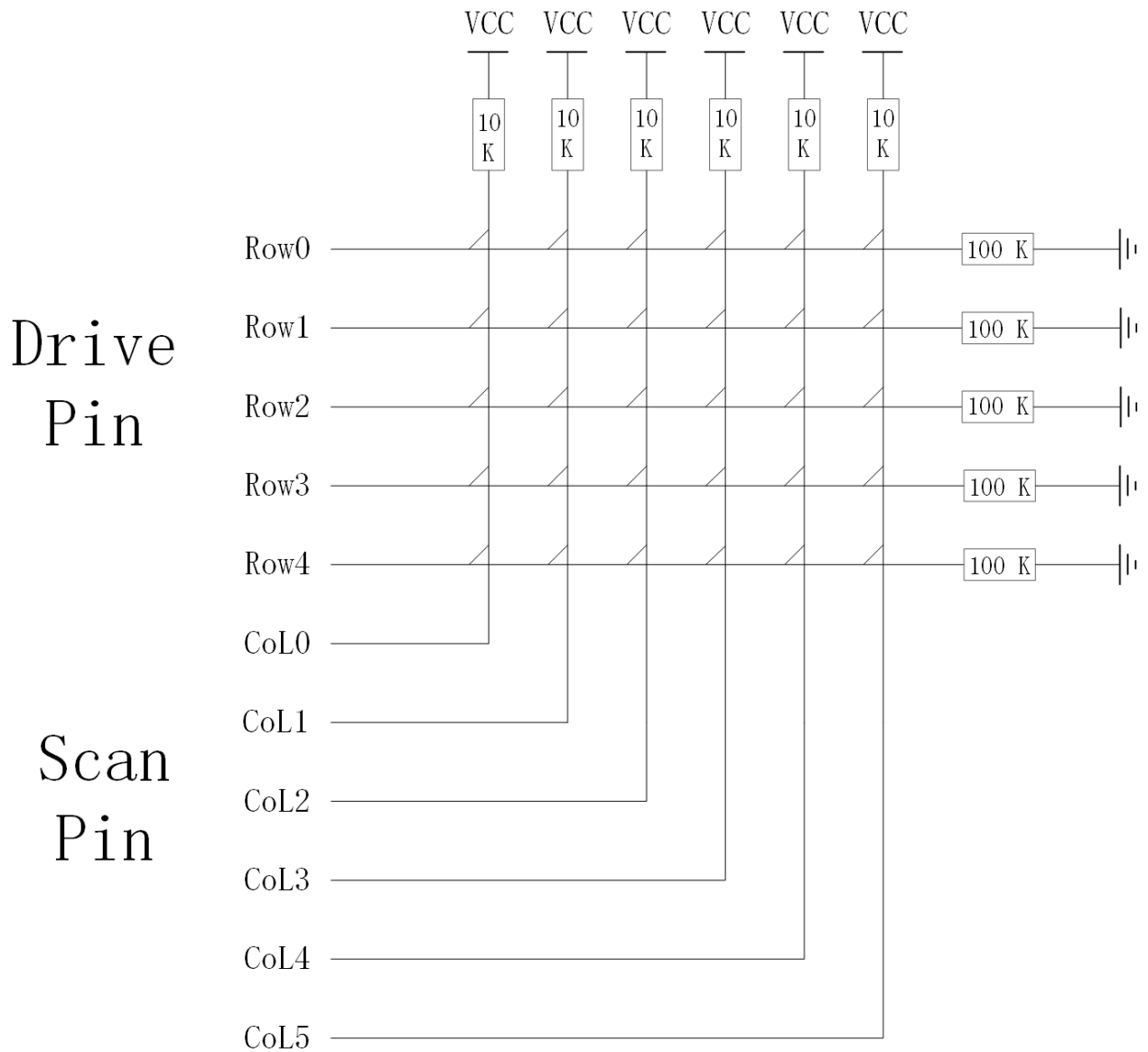


Figure 9.1: "Row Column Key Matrix"

The Telink EVK board is a 2*2 keyboard matrix. In the actual product application, more keys may be needed, such as remote control switches, etc. The following is an example of Telink's demo board providing remote

control. Combined with the above figure, the keyscan related configuration in app_config.h is explained in detail as follows.

According to the real hardware circuit, on Telink demo board, Row0~Row4 pins are PE2, PB4, PB5, PE1 and PE4, CoL0~CoL5 pins are PB1, PB0, PA4, PA0, PE6 and PE5.

Define drive pin array and scan pin array:

```
#define KB_DRIVE_PINS    {GPIO_PE2, GPIO_PB4, GPIO_PB5, GPIO_PE1, GPIO_PE4}
#define KB_SCAN_PINS     {GPIO_PB1, GPIO_PB0, GPIO_PA4, GPIO_PA0, GPIO_PE6, GPIO_PE5}
```

Keyscan adopts analog pull-up/pull-down resistor of GPIO: drive pins use 100K pull-down resistor, and scan pins use 10K pull-up resistor. When no button is pressed, scan pins act as input GPIOs and read high level due to 10K pull-up resistor. When key scan starts, drive pins output low level; if low level is detected on a scan pin, it indicates there's button pressed in current column (Note: Drive pins are not in float state, if output is not enabled, scan pins still detect high level due to voltage division of 100K and 10K resistor.)

Define valid voltage level detected on scan pins when drive pins output low level in Row/Column scan:

```
#define KB_LINE_HIGH_VALID    0
```

Define pull-up resistor for scan pins and pull-down resistor for drive pins:

```
#define MATRIX_ROW_PULL      PM_PIN_PULLDOWN_100K
#define MATRIX_COL_PULL      PM_PIN_PULLUP_10K
#define PULL_WAKEUP_SRC_PE2  MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PB4  MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PB5  MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PE1  MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PE4  MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PB1  MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PB0  MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PA4  MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PA0  MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PE6  MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PE5  MATRIX_COL_PULL
```

Since "ie" of general GPIOs is set as 0 by default in gpio_init, to read level on scan pins, corresponding "ie" should be enabled.

```
#define PB1_INPUT_ENABLE      1
#define PB0_INPUT_ENABLE      1
#define PA4_INPUT_ENABLE      1
#define PA0_INPUT_ENABLE      1
#define PE6_INPUT_ENABLE      1
#define PE5_INPUT_ENABLE      1
```


When MCU enters sleep mode, it's needed to configure PAD GPIO wakeup. Set drive pins as high level wakeup; when there's button pressed, drive pin reads high level, which is 10/11 VCC. To read level state of drive pins, corresponding "ie" should be enabled.

```
#define PE2_INPUT_ENABLE    1
#define PB4_INPUT_ENABLE    1
#define PB5_INPUT_ENABLE    1
#define PE1_INPUT_ENABLE    1
#define PE4_INPUT_ENABLE    1
```

9.2 Keyscan and Keymap

9.2.1 Keyscan

After configuration as shown above, the function below is invoked in main_loop to implement keyscan.

```
u32 kb_scan_key (int numlock_status, int read_key)
```

numlock_status: Generally set as 0 when invoked in main_loop. Set as "KB_NUMLOCK_STATUS_POWERON" only for fast keyscan after wakeup from deepsleep (corresponding to DEEPBACK_FAST_KEYSCAN_ENABLE).

read_key: Buffer processing for key values, generally not used and set as 1 (if it's set as 0, key values will be pushed into buffer and not reported to upper layer).

The return value is used to inform user whether matrix keyboard update is detected by current scan: if yes, return 1; otherwise return 0.

The "kb_scan_key" is invoked in main_loop. As in BLE timing sequence, each main_loop is an adv_interval or conn_interval. In advertising state (suppose adv_interval is 30ms), key scan is processed once for each 30ms; in connection state (suppose conn_interval is 10ms), key scan is processed once for each 10ms.

In theory, when button states in matrix are different during two adjacent key scans, it's considered as an update.

In actual code, a debounce filtering processing is enabled: It will be considered as a valid update, only when button states stay the same during two adjacent key scans, but different with the latest stored matrix keyboard state. "1" will be returned by the function to indicate valid update, matrix keyboard state will be indicated by the structure "kb_event", and current button state will be updated to the newest matrix keyboard state. Corresponding code in keyboard.c is shown as below:

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

The newest button state means press or release state set of all buttons in the matrix. When power on, initial matrix keyboard state shows all buttons are "released" by default, and debounce filtering processing is enabled. As long as valid update occurs to the button state, "1" will be returned, otherwise "0" will be returned.

For example: press a button, a valid update is returned; release a button, a valid update is returned; press another button with a button held, a valid update is returned; press the third button with two buttons held, a valid update is returned; release a button of the two pressed buttons, a valid update is returned.....

9.2.2 Keymap & kb_event

If a valid button state update is detected by invoking the “kb_scan_key”, user can obtain current button state via a global structure variable “kb_event”.

```
#define KB_RETURN_KEY_MAX    6
typedef struct{
    u8 cnt;
    u8 ctrl_key;
    u8 keycode[KB_RETURN_KEY_MAX];
}kb_data_t;
kb_data_t    kb_event;
```

The “kb_event” consists of 8 bytes:

- “cnt” serves to indicate valid count number of pressed buttons currently;
- “ctrl_key” is not used generally except for standard USB HID keyboard (user is not allowed to set keycode in keymap as 0xe0-0xe7).
- keycode[6] indicates keycode of up to six pressed buttons can be stored (if more than six buttons are pressed actually, only the former six can be reflected).

Keycode definition of all buttons in the “app_config.h” is shown as below:

```
#define KB_MAP_NORMAL { \
VK_B,    CR_POWER,    VK_NONE,    VK_C,    CR_HOME,    | \
VOICE,   VK_NONE,    VK_NONE,    CR_VOL_UP, CR_VOL_DN, | \
VK_2,    VK_RIGHT,   CR_VOL_DN,   VK_3,    VK_1,    | \
VK_5,    VK_ENTER,   CR_VOL_UP,   VK_6,    VK_4,    | \
VK_8,    VK_DOWN,    VK_UP,    VK_9,    VK_7,    | \
VK_0,    CR_BACK,    VK_LEFT,   CR_VOL_MUTE, CR_MENU,  } \
```

The keymap follows the format of 5*6 matrix structure. The keycode of pressed button can be configured accordingly, for example, the keycode of the button at the cross of Row0 and CoL0 is “VK_B”.

In the “kb_scan_key” function, the “kb_event.cnt” will be cleared before each scan, while the array “kb_event.keycode[]” won’t be cleared automatically. Whenever “1” is returned to indicate valid update, the “kb_event.cnt” will be used to check current valid count number of pressed buttons.

- If current kb_event.cnt = 0, previous valid matrix state “kb_event.cnt” must be uncertain non-zero value; the update must be button release, but the number of released button is uncertain. Data in kb_event.keycode[] (if available) is invalid.
- If current kb_event.cnt = 1, the previous kb_event.cnt indicates button state update. If previous kb_event.cnt is 0, it indicates the update is one button is pressed; if previous kb_event.cnt is 2, it indicates the update is one of the two pressed buttons is released; if previous kb_event.cnt is 3, it indicates the update is two of the three pressed buttons are released.....kb_event.keycode[0] indicates the key value of currently pressed button. The subsequent keycodes are negligible.

- c) If current `kb_event.cnt = 2`, the previous `kb_event.cnt` indicates button state update. If previous `kb_event.cnt` is 0, it indicates the update is two buttons are pressed at the same time; if previous `kb_event.cnt` is 1, it indicates the update is another button is pressed with one button held; if previous `kb_event.cnt` is 3, it indicates the update is one of the three pressed buttons is released..... `kb_event.keycode[0]` and `kb_event.keycode[1]` indicate key values of the two pressed buttons currently. The subsequent keycodes are negligible.

User can manually clear the “`kb_event.keycode`” before each key scan, so that it can be used to check whether valid update occurs, as shown in the example below.

In the sample code, when `kb_event.keycode[0]` is not zero, it’s considered a button is pressed, but the code won’t check further complex cases, such as whether two buttons are pressed at the same time or one of the two pressed buttons is released.

```
kb_event.keycode[0] = 0; //clear keycode[0]
int det_key = kb_scan_key (0, 1);
if (det_key)
{
    key_not_released = 1;
    u8 key0 = kb_event.keycode[0];
    if (kb_event.cnt == 2) //two key press, do not process
    {
    }
    else if(kb_event.cnt == 1)
    {
key_buf[2] = key0;
        //send key press
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H, key_buf, 8);
    }
    else //key release
    {
        key_not_released = 0;
        key_buf[2] = 0;
        //send key release
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H, key_buf, 8);
    }
}
```

9.3 Keyscan Flow

When “`kb_scan_key`” is invoked, a basic keyscan flow is shown as below:

- (1) Initial full scan through the whole matrix.

All drive pins output drive level (0). Meanwhile read all scan pins, check for valid level, and record the column on which valid level is read. (The `scan_pin_need` is used to mark valid column number.)

If row-by-row scan is directly adopted without initial full scan through the whole matrix, each time all rows should be scanned at least, even if no button is pressed. To save scan time, initial full scan through the whole matrix can be added, thus it will directly exit keyscan if no button press is detected on any column.

The first full scan codes:

```
scan_pin_need = kb_key_pressed (gpio);
```

In the "kb_key_pressed" function, all rows output low level, and stabilized level of scan pins will be read after 20us delay. A release_cnt is set as 6; if a detection shows all pressed buttons in the matrix are released, it won't consider no button is pressed and stop row-by-row scan immediately, but buffers for six frames. If six successive detections show buttons are all released, it will stop row-by-row scan. Thus key debounce processing is realized.

(2) Scan row by row according to full scan result through the whole matrix.

If button press is detected by full scan, row-by-row scan is started: Drive pins (ROW0~ROW4) output valid drive level row by row; read level on columns, and find the pressed button. Following is related code:

```
u32 pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};
kb_scan_row (0, gpio);
for (int i=0; i<=ARRAY_SIZE(drive_pins); i++) {
    u32 r = kb_scan_row (i < ARRAY_SIZE(drive_pins) ? i : 0, gpio);
    if (i) {
        pressed_matrix[i - 1] = r;
    }
}
```

The following methods are used to optimize code execution time for row-by-row scan.

- When a row outputs drive level, it's not needed to read level of all columns (CoL0~CoL5). Since the scan_pin_need marks valid column number, user can read the marked columns only.
- After a row outputs drive level, a 20us or so delay is needed to read stabilized level of scan pins, and a buffer processing is used to utilize the waiting duration.

The array variable "u32 pressed_matrix[5]" (up to 40 columns are supported) is used to store final matrix keyboard state: pressed_matrix[0] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row0,, pressed_matrix[4] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row4.

(3) Debounce filtering for pressed_matrix[.].

Corresponding codes:

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
u32 key_changed = key_debounce_filter( pressed_matrix, (numlock_status &
↳ KB_NUMLOCK_STATUS_POWERON) ? 0 : 1);
```

During fast keyscan after wakeup from deepsleep, "numlock_status" equals "KB_NUMLOCK_STATUS_POWERON"; the "filt_en" is set as 0 to skip filtering and fast obtain key values.

In other cases, the "filt_en" is set as 1 to enable filtering. Only when pressed_matrix[] stays the same during two adjacent key scans, but different from the latest valid pressed_matrix[], will the "key_changed" set as 1 to indicate valid update in matrix keyboard.

(4) Buffer processing for pressed_matrix[].

Push pressed_matrix[] into buffer. When the "read_key" in "kb_scan_key (int numlock_status, int read_key)" is set as 1, the data in the buffer will be read out immediately. When the "read_key" is set as 0, the buffer stores the data without notification to the upper layer; the buffered data won't be read until the read_key is 1.

In current SDK, the "read_key" is fixed as 1, i.e. the buffer does not take effect actually.

(5) According to pressed_matrix[], look up the KB_MAP_NORMAL table and return key values.

Corresponding functions are "kb_remap_key_code" and "kb_remap_key_row".

9.4 Deepsleep wake_up fast keyscan

When the Slave device enters deepsleep while it is connected, it is woken up by a keystroke. After waking up, the program starts from the beginning and has to send broadcast packets in main_loop after user_init and wait until it is connected before sending the value of the key to ble master.

The BLE SDK has already done the relevant processing to make the deep back as fast as possible, but this time may still reach the 100 ms level (e.g. 300 ms). In order to prevent the wake up keystroke from being lost, a fast keyscan and data caching process is done in the SDK.

The fast keyscan is because the MCU will consume some time to re-initialize from the flash load program after the key wakes up, and the time of keyscan in main_loop will also take some more time due to the anti-jitter filter processing, which may lead to the loss of this key.

The data is cached because if valid key data is scanned in the broadcast state, after pushing to the BLE TX fifo, the data in the connected state will be cleared again.

The relevant code is controlled by the macro DEEPBACK_FAST_KEYSCAN_ENABLE in app_config.h.

```
#define DEEPBACK_FAST_KEYSCAN_ENABLE    1
void deep_wakeup_proc(void)
{
    #if(DEEPBACK_FAST_KEYSCAN_ENABLE)
        if(analog_read(DEEP_ANA_REG0) == CONN_DEEP_FLG){
            if(kb_scan_key (KB_NUMLOCK_STATUS_POWERON,1) && kb_event.cnt){
                deepback_key_state = DEEPBACK_KEY_CACHE;
                key_not_released = 1;
                memcpy(&kb_event_cache,&kb_event,sizeof(kb_event));
            }
        }
    #endif
}
```

When initializing, scan the key before user_init, read the deep non-power-off analog register to detect that it is connected state into deep wakeup, call kb_scan_key. Then do not start the anti-jitter filtering process, and directly get the key state of the whole matrix currently read. If the scan finds that a key is pressed (the key change is returned and kb_event.cnt is not 0), the kb_event variable is copied to the cache variable kb_event_cache.

Add deepback_pre_proc and deepback_post_proc processing to the keyscan of main_loop.

```
void proc_keyboard (u8 e, u8 *p)
{
    kb_event.keycode[0] = 0;
    int det_key = kb_scan_key (0, 1);
    #if(DEEPBACK_FAST_KEYSCAN_ENABLE)
        if(deepback_key_state != DEEPBACK_KEY_IDLE){
            deepback_pre_proc(&det_key);
        }
    #endif
    if (det_key){
        key_change_proc();
    }
    #if(DEEPBACK_FAST_KEYSCAN_ENABLE)
        if(deepback_key_state != DEEPBACK_KEY_IDLE){
            deepback_post_proc();
        }
    #endif
}
```

The deepback_pre_proc processing is to wait until the slave and master are connected, and when there is no key state change in a certain kb_key_scan, the value of the previously cached kb_event_cache is used as the current latest key change, which realizes the cache processing of fast sweep key value.

Pay attention to the processing of key release: When manually setting the key value, determine whether the current matrix key is still pressed. If a key is pressed, there is no need to add a manual release, because a release action will be generated when the actual key is released; if the current key has been released, mark a manual release later, otherwise there may be a cached key event is always valid and cannot be released.

The deepback_post_proc processing is to decide whether to put a key release event in the ble TX fifo according to whether there is a manual release event left in the deepback_pre_proc.

9.5 Repeat Key Processing

The most basic keyscan described above only generates a change event when the state of a key is changed and reads the current key value via kb_event, but it is not possible to implement the repeat key function. When a key is pressed all the time, a key value needs to be sent at regular intervals.

The “repeat key” function is masked by default. By configuring related macros in the “app_config.h”, this function can be controlled correspondingly.

```
#define KB_REPEAT_KEY_ENABLE          0
#define KB_REPEAT_KEY_INTERVAL_MS    200
#define KB_REPEAT_KEY_NUM            1
#define KB_MAP_REPEAT                 {VK_1, }
```

(1) KB_REPEAT_KEY_ENABLE

This macro serves to enable or mask the repeat key function. To use this function, first set "KB_REPEAT_KEY_ENABLE" as 1.

(2) KB_REPEAT_KEY_INTERVAL_MS

This macro serves to set the repeat interval time. For example, if it's set as 200ms, it indicates when a button is held, kb_key_scan will return an update with the interval of 200ms. Current button state will be available in kb_event.

(3) KB_REPEAT_KEY_NUM & KB_MAP_REPEAT

The two macros serve to define current repeat key values: KB_REPEAT_KEY_NUM specifies the number of keycodes, while the KB_MAP_REPEAT defines a map to specify all repeat keycodes. Note that the keycodes in the KB_MAP_REPEAT must be the values in the KB_MAP_NORMAL.

Following example shows a 6*6 matrix keyboard: by configuring the four macros, eight buttons including UP, DOWN, LEFT, RIGHT, V+, V-, CHN+ and CHN- are set as repeat keys with repeat interval of 100ms, while other buttons are set as non-repeat keys.

```
#define KB_MAP_NORMAL { \
    {VK_POWER, VK_LOW_BATT, VK_TV_PLUS, VK_TV_MINUS, VK_IN_OUTPUT, VK_VOL_UP, }, \
    {VK_VOICE_SEARCH, VK_PROGRAM, VK_RETURN, VK_HOME, VK_MENU, VK_EXIT, }, \
    {VK_UP, VK_CH_UP, VK_W_MUTE, VK_LEFT, VK_CONFIRM, VK_RIGHT, }, \
    {VK_VOL_DN, VK_DOWN, VK_CH_DN, VK_FAST_BACKWARD, VK_PLAY_PAUSE, VK_1, }, \
    {VK_2, VK_3, VK_4, VK_5, VK_6, VK_7, }, \
    {VK_9, VKPAD_ASTERIX, VK_0, VK_NUMBER, VK_W_SRCH, VK_8, }, \
}

#define KB_REPEAT_KEY_ENABLE          1
#define KB_REPEAT_KEY_INTERVAL_MS    100
#define KB_REPEAT_KEY_NUM            8
#define KB_MAP_REPEAT                 { VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, \
                                       VK_VOL_UP, VK_VOL_DN, VK_CH_UP, VK_CH_DN, }
```

Figure 9.2: "Repeat Key Application Example"

User can search for the four macros in the project to locate the code about repeat key.

9.6 Stuck Key Processing

Stuck key processing is used to save power when one or multiple buttons of a remote control/keyboard is/are pressed and held for a long time unexpectedly, for example a RC is pressed by a cup or ashtray. If keyscan detects some button is pressed and held, without the stuck key processing, MCU won't enter deepsleep or other low power state since it always considers the button is not released.

Following are two related macros in the app_config.h:

```
#define STUCK_KEY_PROCESS_ENABLE      0
#define STUCK_KEY_ENTERDEEP_TIME     60//in s
```

By default the stuck key processing function is masked. User can set the "STUCK_KEY_PROCESS_ENABLE" as 1 to enable this function.

The "STUCK_KEY_ENTERDEEP_TIME" serves to set the stuck key time: if it's set as 60s, it indicates when button state stays fixed for more than 60s with some button held, it's considered as stuck key, and MCU will enter deepsleep.

User can search for the macro "STUCK_KEY_PROCESS_ENABLE" to locate related code in the keyboard.c, as shown below:

```
#if (STUCK_KEY_PROCESS_ENABLE)
    u8 stuckKeyPress[ARRAY_SIZE(drive_pins)];
#endif
```

An u8-type array stuckKeyPress[5] is defined to record row(s) with stuck key in current key matrix. The array value is obtained in the function "key_debounce_filter".

Upper-layer processing is shown as below:

```
kb_event.keycode[0] = 0;
int det_key = kb_scan_key (0, 1);
if (det_key){
    if(kb_event.cnt){ //key press
        stuckKey_keyPressTime = clock_time() | 1;;
    }
    .....
}
```

For each button state update, when button press is detected (i.e. kb_event.cnt is non-zero value), the "stuckKey_keyPressTime" is used to record the time for the latest button press state.

Processing in the "blt_pm_proc" is shown as below:

```
#if (STUCK_KEY_PROCESS_ENABLE)
if(key_not_released && clock_time_exceed(stuckKey_keyPressTime,
↳ STUCK_KEY_ENTERDEEP_TIME*1000000)){
    u32 pin[] = KB_DRIVE_PINS;
    for (u8 i = 0; i < ARRAY_SIZE(pin); i ++){
        extern u8 stuckKeyPress[];
        if(stuckKeyPress[i])
            continue;
        cpu_set_gpio_wakeup (pin[i],0,1);
    }
}
```



```

..... if(sendTerminate_before_enterDeep == 1){ //sending Terminate and wait for ack before enter
↳ deepsleep
    if(user_task_flg){ //detect key Press again, can not enter deep now
        sendTerminate_before_enterDeep = 0;
        bls_ll_setAdvEnable(BLC_ADV_ENABLE); //enable adv again
    }
}
else if(sendTerminate_before_enterDeep == 2){ //Terminate OK
    cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0); //deepSleep
} }#endif

```

Determine whether the time of the most recent key press has exceeded 60s continuously. If it exceeds, it is considered that the stuck key processing has occurred. According to the stuckKeyPress[] of the bottom layer, all the row numbers where the stuck key occurs are obtained, and the original high-level PAD wake-up deepsleep is changed to the low-level PAD wake-up deepsleep.

The reason for the modification is that when the key is pressed, the drive pin on the corresponding line reads a high level of 10/11 VCC. At this time, it is impossible to enter deepsleep because it is already high. As long as you enter deepsleep, it will immediately Wake up by this high level; after modifying it to low level, you can enter deepsleep normally, and when the button is released, the level of the drive pin on the row changes to a low level of 100K pull-down, releasing the button can wake up the entire MCU.

10 LED Management

10.1 LED task related functions

The source code about LED management is available in vendor/common/blt_led.c of this BLE SDK for user reference. User can directly include the "vendor/common/blt_led.h" into his C file.

User needs to invoke the following three functions:

```
void device_led_init(u32 gpio,u8 polarity);
int device_led_setup(led_cfg_t led_cfg);
static inline void device_led_process(void);
```

During initialization, the "device_led_init(u32 gpio,u8 polarity)" is used to set current GPIO and polarity corresponding to LED. If "polarity" is set as 1, it indicates LED will be turned on when GPIO outputs high level; if "polarity" is set as 0, it indicates LED will be turned on when GPIO outputs low level.

The "device_led_process" function is added at UI Entry of main_loop. It's used to check whether LED task is not finished (DEVICE_LED_BUSY). If yes, MCU will carry out corresponding LED task operation.

10.2 LED Task Configuration and Management

10.2.1 LED Event Definition

The following structure serves to define a LED event.

```
typedef struct{
    unsigned short onTime_ms;
    unsigned short offTime_ms;
    unsigned char repeatCount;
    unsigned char priority;
} led_cfg_t;
```

The unsigned short int type "onTime_ms" and "offTime_ms" specify light on and off time (unit: ms) for current LED event, respectively. The two variables can reach the maximum value 65535.

The unsigned char type "repeatCount" specifies blinking times (i.e. repeat times for light on and off action specified by the "onTime_ms" and "offTime_ms"). The variable can reach the maximum value 255.

The "priority" specifies the priority level for current LED event.

To define a LED event when the LED always stays on/off, set the "repeatCount" as 255(0xff), set "onTime_ms"/"offTime_ms" as 0 or non-zero correspondingly.

LED event examples:

- (1) Blink for 3s with 1Hz frequency: keep on for 500ms, stay off for 500ms, and repeat for 3 times.

```
led_cfg_t led_event1 = {500, 500, 3, 0x00};
```

(2) Blink for 50s with 4Hz frequency: keep on for 125ms, stay off for 125ms, and repeat for 200 times.

```
led_cfg_t led_event2 = {125, 125, 200, 0x00};
```

(3) Always on: onTime_ms is non-zero, offTime_ms is zero, and repeatCount is 0xff.

```
led_cfg_t led_event3 = {100, 0, 0xff, 0x00};
```

(4) Always off: onTime_ms is zero, offTime_ms is non-zero, and repeatCount is 0xff.

```
led_cfg_t led_event4 = {0, 100, 0xff, 0x00};
```

(5) Keep on for 3s, and then turn off: onTime_ms is 1000, offTime_ms is 0, and repeatCount is 0x3.

```
led_cfg_t led_event5 = {1000, 0, 3, 0x00};
```

The "device_led_setup" can be invoked to deliver a led_event to LED task management.

```
device_led_setup(led_event1);
```

10.2.2 LED Event Priority

User can define multiple LED events in the SDK, however, only a LED event is allowed to be executed at the same time.

No task list is set for the simple LED management: When LED is idle, LED will accept any LED event delivered by invoking the "device_led_setup". When LED is busy with a LED event (old LED event), if another event (new LED event) comes, MCU will compare priority level of the two LED events; if the new LED event has higher priority level, the old LED event will be discarded and MCU starts to execute the new LED event; if the new LED event has the same or lower priority level, MCU continues executing the old LED event, while the new LED event will be completely discarded, rather than buffered.

By defining LED events with different priority levels, user can realize corresponding LED indicating effect.

Since inquiry scheme is used for LED management, MCU should not enter long suspend (e.g. 10ms * 50 = 500ms) with latency enabled and LED task ongoing (DEVICE_LED_BUSY); otherwise LED event with small onTime_ms value (e.g. 250ms) won't be responded in time, thus LED blinking effect will be influenced.

```
#define DEVICE_LED_BUSY (device_led.repeatCount)
```

The corresponding processing is needed to add in blt_pm_proc, as shown below:

```
user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;  
if(user_task_flg){  
    bls_pm_setManualLatency(0); // manually disable latency  
}
```

Telink Semiconductor

11 Software Timer

Telink BLE SDK supplies source code of blt software timer demo for user reference on timer task. User can directly use this timer or modify as needed.

The source code are available in "vendor/common/blt_soft_timer.c" and "blt_soft_timer.h". To use this timer, the macro below should be set as 1.

```
#define BLT_SOFTWARE_TIMER_ENABLE    0    //enable or disable
```

Since blt software timer is inquiry timer based on system tick, it cannot reach the accuracy of hardware timer, and it should be continuously inquired during main_loop.

The blt soft timer applies to the use scenarios with timing value more than 5ms and without high requirement for time error.

Its key feature is: This timer will be inquired during main_loop, and it ensures MCU can wake up in time from suspend and execute timer task. This design is implemented based on "Timer wakeup by Application layer" (section 4.5 Timer wakeup by Application Layer).

The current design can run up to four timers, and maximum timer number is modifiable via the macro below:

```
#define    MAX_TIMER_NUM    4    //timer max number
```

11.1 Timer Initialization

The API below is used for blt software timer initialization:

```
void blt_soft_timer_init(void);
```

Timer initialization only registers "blt_soft_timer_process" as callback function of APP layer wakeup in advance.

```
void blt_soft_timer_init(void){  
    bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);  
}
```

11.2 Timer Inquiry Processing

The function "blt_soft_timer_process" serves to implement inquiry processing of blt software timer.

```
void blt_soft_timer_process(int type);
```

On one hand, main_loop should always invoke this function in the location as shown in the figure below. On the other hand, this function must be registered as callback function of APP layer wakeup in advance. Whenever MCU is woke up from suspend in advance by timer, this function will be quickly executed to process timer task.

```
_attribute_ram_code_ void main_loop (void)
{
    tick_loop++;
    #if (FEATURE_TEST_MODE == TEST_USER_BLT_SOFT_TIMER)
        blt_soft_timer_process(MAINLOOP_ENTRY);
    #endif
    blt_sdk_main_loop();
}
```

The parameter "type" of the "blt_soft_timer_process" indicates two cases to enter this function: If "type" is 0, it indicates entering this function via inquiry in main_loop; if "type" is 1, it indicates entering this function when MCU is woke up in advance by timer.

```
#define MAIN_LOOP_ENTRY 0
#define CALLBACK_ENTRY 1
```

The implementation of the "blt_soft_timer_process" is rather complex, and its basic principle is shown as below:

- (1) First check whether there is still user-defined timer in current timer table. If not, directly exit the function and disable timer wakeup of APP layer; if there's timer task, continue the flow.

```
if(!blt_timer.currentNum){
    bls_pm_setAppWakeupLowPower(0, 0); //disable
    return;
}
```

- (2) Check whether the nearest timer task is reached: if the task is not reached, exit the function; otherwise continue the flow. Since the design will ensure all timers are time-ordered, herein it's only needed to check the nearest timer.

```
if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ){
    return;
}
```

- (3) Inquire all current timer tasks, and execute corresponding task as long as timer value is reached.

```
for(int i=0; i<blt_timer.currentNum; i++){
    if(blt_is_timer_expired(blt_timer.timer[i].t ,now) ){ //timer trigger
        if(blt_timer.timer[i].cb == NULL){
        }
    }
}
```

```

else{
    result = blt_timer.timer[i].cb();
    if(result < 0){
        blt_soft_timer_delete_by_index(i);
    }
    else if(result == 0){
        change_flg = 1;
        blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
    }
    else{ //set new timer interval
        change_flg = 1;
        blt_timer.timer[i].interval = result * CLOCK_16M_SYS_TIMER_CLK_1US;
        blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
    }
}
}
}

```

The code above shows processing of timer task function: If the return value of this function is less than 0, this timer task will be deleted and won't be responded; if the return value is 0, the previous timing value will be retained; if the return value is more than 0, this return value will be used as the new timing cycle (unit: us).

- (4) In step 3, if tasks in timer task table change, the previous time sequence may be disturbed, and re-ordering is needed.

```

if(change_flg){
    blt_soft_timer_sort();
}

```

- (5) If the nearest timer task will be responded within 3s (it can be changed to a value larger than 3s as needed) from now, the response time will be set as wakeup time by APP layer in advance; otherwise APP layer wakeup in advance will be disabled.

```

if( (u32)(blt_timer.timer[0].t - now) < 3000 * CLOCK_16M_SYS_TIMER_CLK_1MS){
    bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
}
else{
    bls_pm_setAppWakeupLowPower(0, 0); //disable
}

```

11.3 Add Timer Task

The API below serves to add timer task.

```
typedef int (*blt_timer_callback_t)(void);
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us);
```

“func”: timer task function.

“interval_us”: timing value (unit: us).

The int-type return value corresponds to three processing methods:

- a) If the return value is less than 0, this task will be automatically deleted after execution. This feature can be used to control the number of timer execution times.
- b) If the return value is 0, the old interval_us will be used as timing cycle.
- c) If the return value is more than 0, this return value will be used as new timing cycle (unit: us).

```
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us)
{
    int i;
    u32 now = clock_time();
    if(blt_timer.currentNum >= MAX_TIMER_NUM){ //timer full
        return 0;
    }
    else{
        blt_timer.timer[blt_timer.currentNum].cb = func;
        blt_timer.timer[blt_timer.currentNum].interval = interval_us *
↪ CLOCK_16M_SYS_TIMER_CLK_1US;
        blt_timer.timer[blt_timer.currentNum].t = now +
↪ blt_timer.timer[blt_timer.currentNum].interval;
        blt_timer.currentNum ++;
        blt_soft_timer_sort();
        bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
        return 1;
    }
}
```

As shown in the implementation code, if timer number exceeds the maximum value, the adding operation will fail. Whenever a new timer task is added, re-ordering must be implemented to ensure timer tasks are time-ordered, while the index corresponding to the nearest timer task should be 0.

11.4 Delete Timer Task

As introduced above, timer task will be automatically deleted when the return value is less than 0. Except for this case, the API below can be invoked to specify the timer task to be deleted.

```
int blt_soft_timer_delete(blt_timer_callback_t func);
```


11.5 Demo

For Demo code of blt soft timer, please refer to "TEST_USER_BLT_SOFT_TIMER" in feature_test.

```
int gpio_test0(void)
{
    DBG_CHN3_TOGGLE;
    return 0;
}
int gpio_test1(void)
{
    DBG_CHN4_TOGGLE;
    static u8 flg = 0;
    flg = !flg;
    if(flg){
        return 7000;
    }
    else{
        return 17000;
    }
}
int gpio_test2(void)
{
    DBG_CHN5_TOGGLE;
    if(clock_time_exceed(0, 5000000)){
        //return -1;
        blt_soft_timer_delete(&gpio_test2);
    }
    return 0;
}
int gpio_test3(void)
{
    DBG_CHN6_TOGGLE;
    return 0;
}
```

Initialization:

```
blt_soft_timer_init();
blt_soft_timer_add(&gpio_test0, 23000);
blt_soft_timer_add(&gpio_test1, 7000);
blt_soft_timer_add(&gpio_test2, 13000);
blt_soft_timer_add(&gpio_test3, 27000);
```

Four timer tasks are defined with different features:

- (1) gpio_test0: Toggle once for every 23ms.

- (2) gpio_test1: Switch between two timers of 7ms/17ms.
- (3) gpio_test2: Delete itself after 5s, which can be implemented by invoking "blt_soft_timer_delete(&gpio_test2)" or "return -1".
- (4) gpio_test3: Toggle once for every 27ms.

Telink Semiconductor

12 IR

12.1 PWM Driver

By operating registers, hardware configurations for PWM are very simple. To improve execution efficiency and save code size, related APIs, implemented via “static inline function”, are defined in the “pwm.h”.

12.1.1 PWM ID and Pin

B85 supports up to 12-channel PWM: PWM0 ~ PWM5 and PWM0_N ~ PWM5_N.

Six-channel PWM is defined in driver:

```
typedef enum {
    PWM0_ID = 0,
    PWM1_ID,
    PWM2_ID,
    PWM3_ID,
    PWM4_ID,
    PWM5_ID,
}pwm_id;
```

Only six channels PWM0-PWM5 are configured in software, while the other six channels PWM0_N-PWM5_N are inverted output of PWM0-PWM5 waveform. For example: PWM0_N is inverted output of PWM0 waveform. When PWM0 is high level, PWM0_N is low level; When PWM0 is low level, PWM0_N is high level. Therefore, as long as PWM0-PWM5 are configured, PWM0_N-PWM5_N are also configured.

IC pins corresponding to 12-channel PWM are shown as below:

Table 12.1: PWM pin allocation

PWMx	Pin	PWMx_n	Pin
PWM0	PA2/PC1/PC2/PD5	PWM0_N	PA0/PB3/PC4/PD5
PWM1	PA3/PC3	PWM1_N	PC1/PD3
PWM2	PA4/PC4	PWM2_N	PD4
PWM3	PB0/PD2	PWM3_N	PC5
PWM4	PB1/PB4	PWM4_N	PC0/PC6
PWM5	PB2/PB5	PWM5_N	PC7

The “void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func)” serves to set specific pin as PWM function.

“pin”: GPIO pin corresponding to actual PWM channel

“func”: Must set as corresponding PWM function, i.e. AS_PWM0 ~ AS_PWM5_N in table above, as shown below.

```
typedef enum{
.....
    AS_PWM0      = 20,
    AS_PWM1      = 21,
    AS_PWM2      = 22,
    AS_PWM3      = 23,
    AS_PWM4      = 24,
    AS_PWM5      = 25,
    AS_PWM0_N    = 26,
    AS_PWM1_N    = 27,
    AS_PWM2_N    = 28,
    AS_PWM3_N    = 29,
    AS_PWM4_N    = 30,
    AS_PWM5_N    = 31,
}GPIO_FuncTypeDef;
```

For example, to use PA2 as PWM0:

```
gpio_set_func(GPIO_PA2, AS_PWM0);
```

12.1.2 PWM Clock

Use API void pwm_set_clk (int system_clock_hz, int pwm_clk) to set the PWM clock.

The system_clock_hz fills in the current system clock CLOCK_SYS_CLOCK_HZ (this macro is defined in app_config.h);

The pwm_clk is the clock to be set, system_clock_hz must be divisible by pwm_clk to get the correct PWM clock.

The PWM clock needs to be as large as possible to make the PWM waveform accurate, the maximum value cannot exceed the system clock. It is recommended to set pwm_clk to CLOCK_SYS_CLOCK_HZ, that is:

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);
```

For example, the current system clock CLOCK_SYS_CLOCK_HZ is 16000000, and the PWM clock set above is equal to the system clock, which is 16M.

If you want the PWM clock to be 8M, you can set it as follows: no matter how the system clock changes (CLOCK_SYS_CLOCK_HZ is 16000000, 24000000 or 32000000), the PWM clock is 8M.

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, 8000000);
```

12.1.3 PWM Cycle and Duty

The basic unit of the PWM waveform is the PWM Signal Frame.

Configuring a PWM Signal Frame requires setting both the cycle and cmp parameters.

`void pwm_set_cycle (pwm_id id, unsigned short cycle_tick)` is used to set the PWM cycle, the unit is the number of PWM clocks.

`void pwm_set_cmp (pwm_id id, unsigned short cmp_tick)` is used to set the PWM cmp, the unit is the number of PWM clock.

The following API combines the above two APIs into one, which can improve the efficiency of settings.

```
void pwm_set_cycle_and_duty(pwm_id id, unsigned short cycle_tick, unsigned short cmp_tick)
```

Then for a PWM signal frame, calculate the PWM duty:

$\text{PWM duty} = \text{PWM cmp} / \text{PWM cycle}$

The figure below is equivalent to the result obtained by `pwm_set_cycle_and_duty (PWM0_ID, 5, 2)`. The cycle of a Signal Frame is 5 PWM clocks, the high level time is 2 PWM clocks, and the PWM duty is 40%.

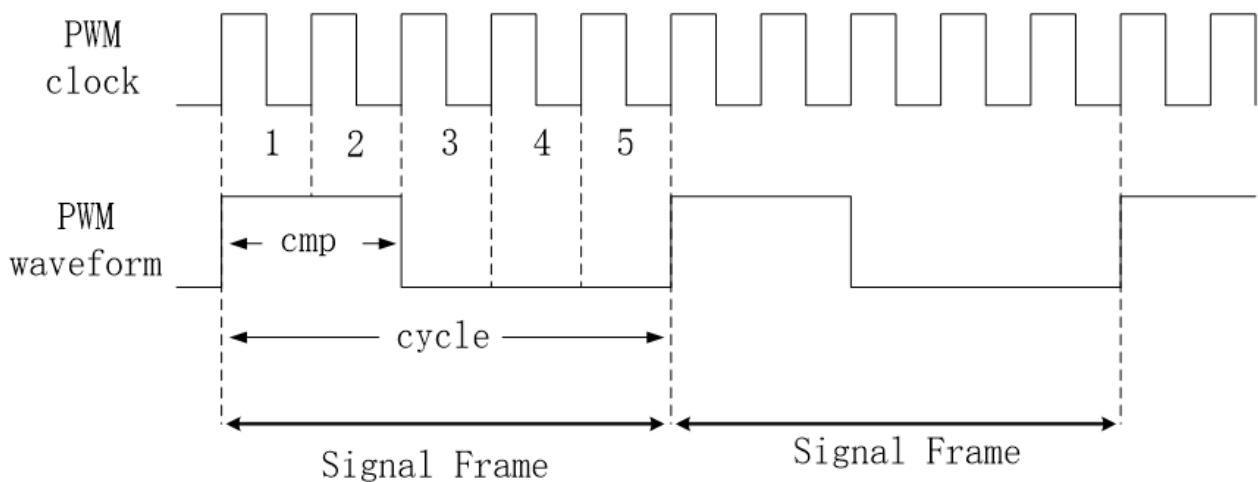


Figure 12.1: "PWM cycle & duty"

For PWM0 ~ PWM5, the hardware will automatically put the high level in the front and the low level in the back. If you want the low level first, there are several ways:

- 1) Use the corresponding PWM0_N ~ PWM5_N, which is the negative of PWM0 ~ PWM5.
- 2) Use the API static inline `void pwm_revert (pwm_id id)` to invert the PWM0 ~ PWM5 waveforms directly.

For example, the current pwm clock is 16MHz, you need to set the PWM period of 1ms, duty cycle of 50% of the PWM0 a frame method as follows:

```
pwm_set_cycle(PWM0_ID , 16000)
pwm_set_cmp (PWM0_ID , 8000)
```

or

```
pwm_set_cycle_and_duty(PWM0_ID, 16000, 8000);
```

12.1.4 PWM Revert

The following API serves to invert PWM0~PWM5 waveform.

```
void pwm_revert (pwm_id id)
```

The following API serves to invert PWM0_N ~ PWM5_N waveform.

```
void pwm_n_revert (pwm_id id)
```

12.1.5 PWM Start and Stop

The following 2 APIs serve to enable/disable specified PWM.

```
void pwm_start(pwm_id id) ;
void pwm_stop(pwm_id id) ;
```

12.1.6 PWM Mode

PWM supports 5 modes: Normal mode (Continuous mode), while only PWM0 supports Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode, defined as following:

```
typedef enum{
typedef enum{
    PWM_NORMAL_MODE      = 0x00,
    PWM_COUNT_MODE       = 0x01,
    PWM_IR_MODE          = 0x03,
    PWM_IR_FIFO_MODE     = 0x07,
    PWM_IR_DMA_FIFO_MODE = 0x0F,
}pwm_mode;
```

PWM0 supports all 5 modes, Normal mode (Continuous mode), while only PWM0 supports Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode, PWM1~PWM5 supports only normal mode.

12.1.7 PWM Pulse Number

The API below serves to set pulse number, i.e. number of Signal Frames, for output waveform of specified PWM channel.

```
void pwm_set_pulse_num(pwm_id id, unsigned short pulse_num)
```

This API is only used for Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode, but not applies to Normal mode with continuous pulses. Normal mode (Continuous mode) is not limited by the number of Signal Frames, so this API is not relevant for Normal mode.

12.1.8 PWM Interrupt

First introduce some basic concepts of Telink MCU interrupt.

The interrupt "status" is a status marker bit generated by a specific action of the hardware (i.e. interrupt action). It does not depend on any software setting, no matter whether the interrupt "mask" is on or not, as soon as the interrupt action occurs, "status" will be set (value is 1). This "status" can be cleared by writing a 1 to "status" (the value returns to 0).

Define the concept of interrupt response: interrupt response means that after the hardware interrupt action is generated, the software program pointer PC jumps to irq_handler for related processing.

If the user wants the interrupt to be responded to, he needs to make sure that all the "masks" corresponding to the current interrupt are turned on. There may be more than one "mask", and the relationship between multiple "masks" is a logical "with" relationship. Only when all "masks" are turned on, the interrupt "status" will trigger the final interrupt response and the MCU will jump to irq_handler to execute; as long as one "mask" is not turned on, the interrupt "status" will not be generated to trigger the interrupt response.

The interrupts that users may need to use in the PWM driver are as follows (code is in the file register_8258.h). Other interrupts are not needed, and users don't need to pay attention.

```
#define reg_pwm_irq_mask      REG_ADDR8(0x7b0)
#define reg_pwm_irq_sta      REG_ADDR8(0x7b1)
enum{
    FLD_IRQ_PWM0_PNUM =      BIT(0),
    FLD_IRQ_PWM0_IR_DMA_FIFO_DONE =  BIT(1),
    FLD_IRQ_PWM0_FRAME =      BIT(2),
    FLD_IRQ_PWM1_FRAME =      BIT(3),
    FLD_IRQ_PWM2_FRAME =      BIT(4),
    FLD_IRQ_PWM3_FRAME =      BIT(5),
    FLD_IRQ_PWM4_FRAME =      BIT(6),
    FLD_IRQ_PWM5_FRAME =      BIT(7),
};
```

The above 8 interrupts correspond to BIT<0:7> of core_7b0/7b1. core_7b0 is the "mask" of these 8 interrupts, and core_7b1 is the "status" of the 8 interrupts.

Divide the 8 interrupt “status” into 3 types. Refer to the figure below, assume PWM0 is operating in IR mode, the duty cycle of PWM Signal Frame is 50%, and the pulse number (or Signal Frame number) of each IR task is 3.

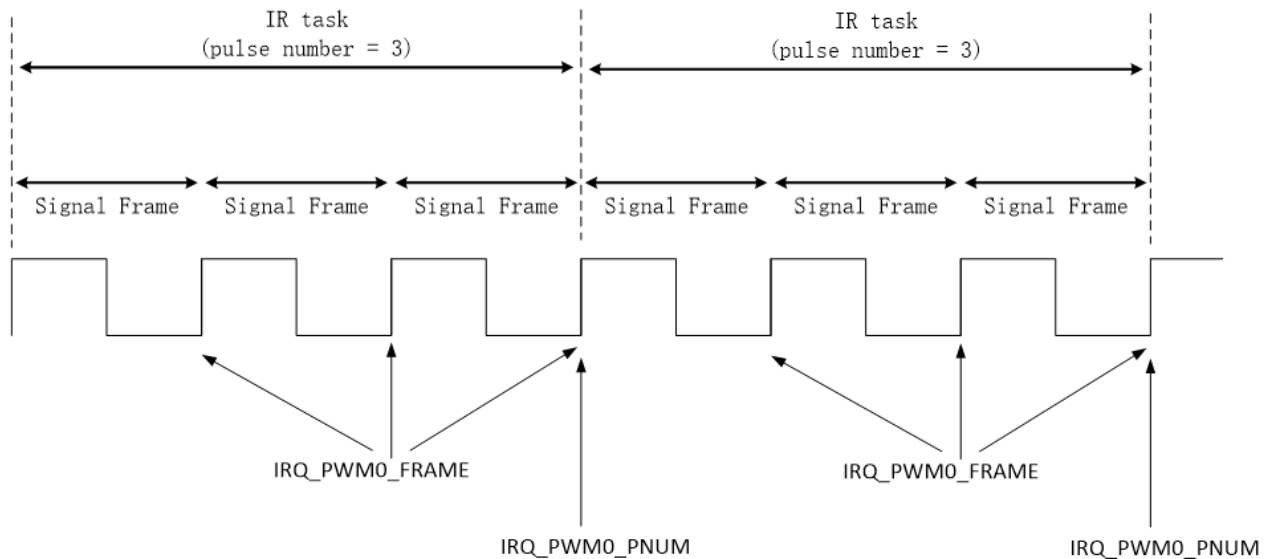


Figure 12.2: “PWM interrupt”

1) First type: IRQ_PWMn_FRAME (n=0,1,2,3,4,5)

The latter 6 interrupt sources are the same interrupt, which are generated on PWM0~PWM5 respectively.

As shown in the figure, IRQ_PWMn_FRAME is an interrupt generated after each PWM Signal Frame ends. In the five modes of PWM, Signal Frame is the basic unit of PWM waveform. So no matter which PWM mode, IRQ_PWMn_FRAME will appear.

2) Second type: IRQ_PWM0_PNUM

IRQ_PWM0_PNUM is an interrupt generated at the end of a Signal Frame (the number is determined by API `pwm_set_pulse_num`). In the figure, one IRQ_PWM0_PNUM is generated after every three Signal Frames.

The Counting mode and IR mode of PWM will use API `pwm_set_pulse_num`. Therefore, only the counting mode and IR mode of PWM0 will generate IRQ_PWM0_PNUM.

3) Third type: IRQ_PWM0_IR_DMA_FIFO_DONE

When PWM0 is operating in IR DMA FIFO mode, IRQ_PWM0_IR_DMA_FIFO_DONE is triggered when all configured PWM waveforms on the DMA have been sent.

As mentioned above, the interrupt response will be triggered when all related interrupt “masks” are turned on at the same time. For PWM interrupts, taking FLD_IRQ_PWM0_PNUM as an example, there are 3 layers of “masks” that need to be turned on:

1) “mask” of FLD_IRQ_PWM0_PNUM

That is the “mask” corresponding to `core_7b0`, can be opened as follows:

```
reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;
```


Generally, clear the previous status before opening the mask to prevent the interrupt response from being triggered by mistake:

```
reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
```

2) PWM “mask” on MCU system interrupt

That is, BIT<14> of core_640.

```
#define reg_irq_mask          REG_ADDR32(0x640)
enum{
    .....
    FLD_IRQ_SW_PWM_EN =      BIT(14),  //irq_software | irq_pwm
    .....
};
```

Open as follows:

```
reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
```

3) MCU total interrupt enable bit, irq_enable().

12.1.9 PWM phase

```
void pwm_set_phase(pwm_id id, unsigned short phase)
```

It is used to set the delay time before the PWM starts.

The phase is the delay time, and the unit is the number of PWM clocks. Generally, there is no need to delay, set to 0.

12.1.10 IR DMA FIFO mode

IR DMA FIFO mode is to write configuration data to FIFO through DMA. Each FIFO uses 2 bytes to represent a PWM waveform. When the DMA data buffer takes effect, the PWM hardware module will sent out PWM waveform 1, waveform 2 Waveform n in chronological order continuously, when fifo finishes executing the cfg_data sent by DMA, it triggers the interrupt IRQ_PWM0_IR_DMA_FIFO_DONE.

12.1.10.1 Configuration for DMA FIFO

On each DMA FIFO, use 2bytes (16 bits) to configure a PWM waveform. Call the following API to return 2 bytes of DMA FIFO data.

```
unsigned short pwm_config_dma_fifo_waveform(int carrier_en,
Pwm0Pulse_SelectDef pulse, unsigned short pulse_num);
```

The API has three parameters: "carrier_en", "pulse" and "pulse_num". The configured PWM waveform is a collection of "pulse_num" PWM signal frames.

BIT(15) determines the format of Signal Frame, the basic unit of the current PWM waveform, corresponding to the "carrier_en" in the API:

- When "carrier_en" is 1, the high pulse in the Signal Frame is effective;
- When "carrier_en" is 0, the signal frame is all 0 data, and the high pulse is invalid.

"Pulse_num" is the number of Signal Frames in the current PWM waveform.

"pulse" can choose the following two definitions.

```
typedef enum{
    PWM0_PULSE_NORMAL =    0,
    PWM0_PULSE_SHADOW =    BIT(14),
}Pwm0Pulse_SelectDef;
```

When "pulse" is PWM0_PULSE_NORMAL, the Signal Frame comes from the configuration of API `pwm_set_cycle_and_duty`; when "pulse" is PWM0_PULSE_SHADOW, the Signal Frame comes from the configuration of PWM shadow mode.

The purpose of PWM shadow mode is to add a set of signal frame configuration, thereby adding more flexibility to the PWM waveform configuration of IR DMA FIFO mode. The configuration API is as follows, and the method is exactly the same as API `pwm_set_cycle_and_duty`.

```
void pwm_set_pwm0_shadow_cycle_and_duty(unsigned short cycle_tick,
unsigned short cmp_tick);
```

12.1.10.2 Set DMA FIFO Buffer

After DMA FIFO buffer is configured, the API below should be invoked to set the starting address of the buffer to DMA module.

```
void pwm_set_dma_address(void * pdat);
```

12.1.10.3 Start and Stop for IR DMA FIFO Mode

After DMA FIFO buffer is prepared, the API below should be invoked to start sending PWM waveforms.

```
void pwm_start_dma_ir_sending(void);
```

After all PWM waveforms in DMA FIFO buffer are sent, the PWM module will be stopped automatically. The API below can be invoked to manually stop the PWM module in advance.

```
void pwm_stop_dma_ir_sending(void);
```

12.2 IR Demo

User can refer to the code of IR in SDK demo "ble_remote" and set the macro REMOTE_IR_ENABLE in app_config.h to 1.

12.2.1 PWM mode selection

As required by IR transmission, PWM output needs to switch at specific time with small error tolerance of switch time accuracy to avoid incorrect IR.

As described in Link Layer timing sequence (section 3.2.4), Link Layers uses system interrupt to process brx event. (In the new SDK, adv event is processed in the main_loop and does not occupy system interrupt time.) When IR is about to switch PWM output soon, if brx event related interrupt comes first and occupies MCU time, the time to switch PWM output may be delayed, thus to result in IR error. Therefore IR cannot use PWM Normal mode.

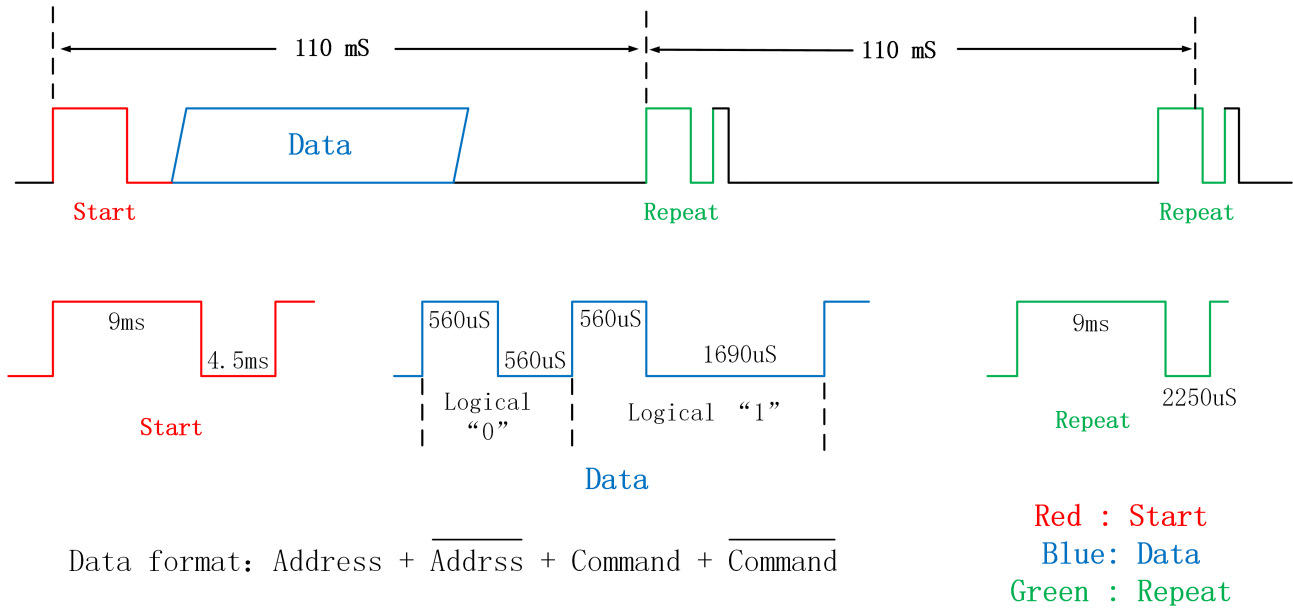
The B85 family introduces an extra IR DMA FIFO mode. In IR DMA FIFO mode, since FIFO can be defined in SRAM, more FIFOs are available, which can effectively solve the shortcoming of PWM IR mode above.

The IR DMA FIFO mode supports pre-storage of multiple PWM waveforms into SRAM. Once DMA is started, no software involvement is needed. This can save frequent SW processing time, and avoid PWM waveform delay caused by simultaneous response to multiple IRQs in interrupt system.

Only PWM0 with IR DMA FIFO mode can be used to implement IR. Therefore, in HW design, IR control GPIO must be PWM0 pin or PWM0_n pin.

12.2.2 Demo IR Protocol

The figure below shows demo IR protocol in the SDK.

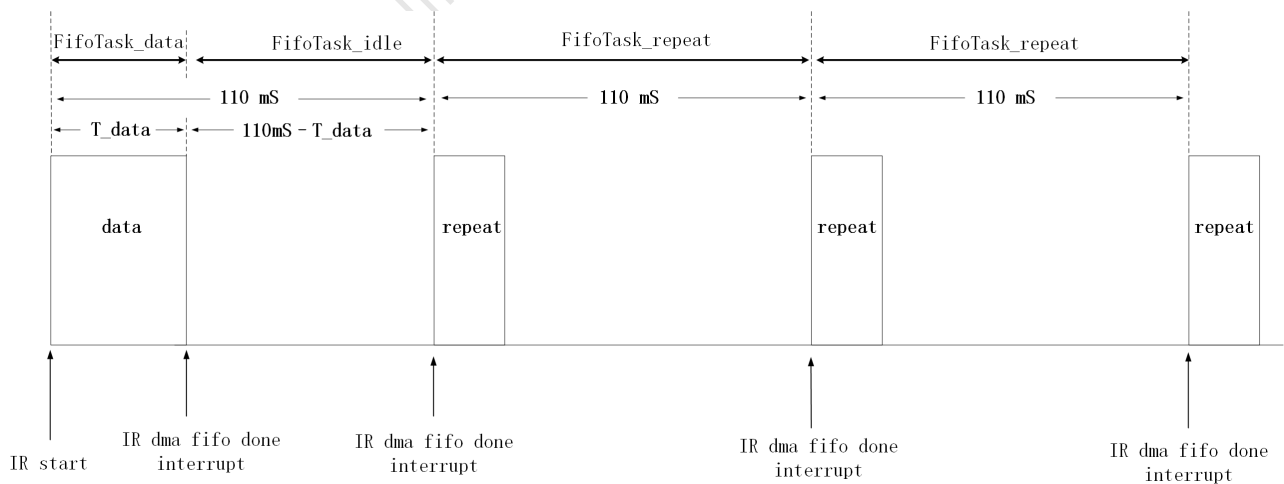

Figure 12.3: "Demo IR Protocol"

12.2.3 IR Timing Design

The figure below shows basic IR timing abased demo IR protocol and feature of IR DMA FIFO mode.

In IR DMA FIFO mode, a complete task is defined as FifoTask. Herein the processing of IR repeat signal adopts the method of "add repeat one by one", i.e. the macro below is defined as 1.

```
#define ADD_REPEAT_ONE_BY_ONE 1
```


Figure 12.4: "IR Timing 1"

When a button is pressed to trigger IR transmission, IR is disassembled to FifoTasks as shown in the figure above.

- (1) After IR is started, run `FifoTask_data` to send valid data. The duration of `FifoTask_data`, marked as T_{data} , is not certain due to the uncertainty of data. After `FifoTask_data` is finished, trigger `IRQ_PWM0_IR_DMA_FIFO_DONE`.
- (2) In interrupt function of `IRQ_PWM0_IR_DMA_FIFO_DONE`, start `FifoTask_idle` phase to send signal without carrier and it lasts for a duration of $(110\text{ms} - T_{data})$. This phase is designed to guarantee the time point the first `FifoTask_repeat` is 110ms later after IR is started. After `FifoTask_idle` is finished, trigger `IRQ_PWM0_IR_DMA_FIFO_DONE`.
- (3) In interrupt function of `IRQ_PWM0_IR_DMA_FIFO_DONE`, start the first `FifoTask_repeat`. Each `FifoTask_repeat` lasts for 110ms. By adding `FifoTask_repeat` in corresponding interrupt function, IR repeat signals can be sent continuously.
- (4) The time point to stop IR is not certain, and it depends on the time to release the button. After the APP layer detects key release, as long as `FifoTask_data` is correctly completed, IR transmission is finished by manually stopping IR DMA FIFO mode.

Following shows some optimization steps for the IR timing design above.

- (1) Since `FifoTask_repeat` timing is fixed, and there are many DMA fifos in IR DMA FIFO mode, multiple `FifoTask_repeat` of 110ms can be assembled into one `FifoTask_repeat*n`, so as to reduce the number of times to process `IRQ_PWM0_IR_DMA_FIFO_DONE` in SW. Corresponding to the processing of "ADD_REPEAT_ONE_BY_ONE" macro defined as 0, the Demo herein assembles five IR repeat signals into one `FifoTask_repeat*5`. User can further optimize it according to the usage of DMA fifos.
- (2) Based on step 1, combine `FifoTask_idle` and the first "`FifoTask_repeat*n`" to form "`FifoTask_idle_repeat*n`".

The figure below shows IR timing after optimization.

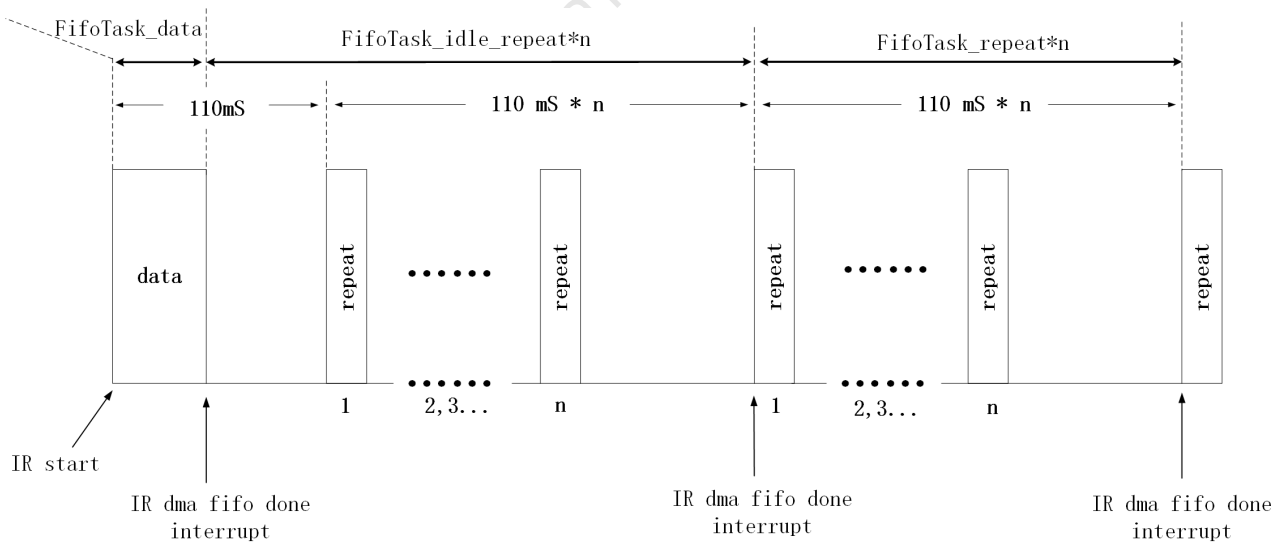


Figure 12.5: "IR Timing 2"

As per the IR timing design above, corresponding code in SW flow is shown as below:

At IR start, invoke the function "ir_nec_send", enable `FifoTask_data`, and use interrupt to control the following flow. In the interrupt when `FifoTask_data` is finished, enable `FifoTask_idle`. In the interrupt when `FifoTask_idle` is finished, enable `FifoTask_repeat`. Before manually stopping IR DMA FIFO mode, `FifoTask_repeat` is executed continually.

```

void ir_nec_send(u8 addr1, u8 addr2, u8 cmd)
{
    if(ir_send_ctrl.last_cmd != cmd)
    {
        if(ir_sending_check())
        {
            return;
        }
        ir_send_ctrl.last_cmd = cmd;

        // set waveform input in sequence
        T_dmaData_buf.data_num = 0;

        //waveform for start bit
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_start_bit_1st;
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_start_bit_2nd;

        //add data
        u32 data = (~cmd)<<24 | cmd<<16 | addr2<<8 | addr1;
        for(int i=0;i<32;i++){
            if(data & BIT(i)){
                //waveform for logic_1
                T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_1_1st;
                T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_1_2nd;
            }
            else{
                //waveform for logic_0
                T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_0_1st;
                T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_0_2nd;
            }
        }

        //waveform for stop bit
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_stop_bit_1st;
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_stop_bit_2nd;
        T_dmaData_buf.dma_len = T_dmaData_buf.data_num * 2;
        ir_send_ctrl.repeat_enable = 1; //need repeat signal
        ir_send_ctrl.is_sending = IR_SENDING_DATA;

        //dma init
        pwm_set_dma_config(PWM_DMA_CHN);
        pwm_set_dma_buf(PWM_DMA_CHN, (u32) &T_dmaData_buf ,T_dmaData_buf.dma_len);
        pwm_ir_dma_mode_start(PWM_DMA_CHN);
        pwm_set_irq_mask(FLD_PWM0_IR_DMA_FIFO_IRQ);
        pwm_clr_irq_status(FLD_PWM0_IR_DMA_FIFO_IRQ );
        core_interrupt_enable();//
        plic_interrupt_enable(IRQ16_PWM);
    }
}

```

```

        ir_send_ctrl.sending_start_time = clock_time();
    }
}

```

12.2.4 IR Initialization

12.2.4.1 rc_ir_init

IR initialization function is shown as below. Please refer to demo code in the SDK.

```
void rc_ir_init(void)
```

12.2.4.2 IR Hardware Configuration

Following shows the demo code.

```

pwm_n_revert(PWM0_ID);
gpio_set_func(GPIO_PB3, AS_PWM0_N);
pwm_set_mode(PWM0_ID, PWM_IR_DMA_FIFO_MODE);
pwm_set_phase(PWM0_ID, 0); //no phase at pwm beginning
pwm_set_cycle_and_duty(PWM0_ID, PWM_CARRIER_CYCLE_TICK,
                      PWM_CARRIER_HIGH_TICK );
pwm_set_dma_address(&T_dmaData_buf);
reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
reg_pwm_irq_sta = FLD_IRQ_PWM0_IR_DMA_FIFO_DONE;

```

Only PWM0 supports ID DMA FIFO mode, so choose PB3 corresponding to PWM0_N.

The Demo IR carrier frequency is 38K, the cycle is 26.3uS, and the duty is 1/3. Use API `pwm_set_tmax` and `pwm_set_tcmp` and configure the cycle and duty. In Demo IR, there are no multiple different carrier frequencies. This 38K carrier is sufficient for all FifoTask configurations. So there is no need to use PWM shadow mode.

DMA FIFO buffer is `T_dmaData_buf`.

Turn on the system interrupt mask "FLD_IRQ_SW_PWM_EN".

Clear interrupt status "FLD_IRQ_PWM0_IR_DMA_FIFO_DONE".

12.2.4.3 IR Variable Initialization

Related variables in the SDK demo includes `waveform_start_bit_1st`, `waveform_start_bit_2nd`, and etc.

As introduced in IR timing design, `FifoTask_data` and `FifoTask_repeat` should be configured.

Start signal = 9ms carrier signal + 4.5ms low level signal (no carrier). The "pwm_config_dma_fifo_waveform" is invoked to configure the two corresponding DMA FIFO data.

```
//start bit, 9000 us carrier, 4500 us low
    waveform_start_bit_1st = pwm_config_dma_fifo_waveform(1, PWM0_PULSE_NORMAL, 9000 *
↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    waveform_start_bit_2nd = pwm_config_dma_fifo_waveform(0, PWM0_PULSE_NORMAL, 4500 *
↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    u16 waveform_stop_bit_2nd;
```

The method also applies to configure stop signal, repeat signal, data logic "1" signal, and data logic "0" signal.

12.2.5 FifoTask Configuration

12.2.5.1 FifoTask_data

As per demo IR protocol, to send a cmd (e.g. 7), first send start signal, i.e. 9ms carrier signal + 4.5ms low level signal (no carrier); then send "address+ ~address+ cmd + ~cmd". In the demo code, address is 0x88.

When sending the final bit of "~cmd", logical "0" or logical "1" always contains some non-carrier signals at the end. If "~cmd" is not followed by any data, there may be a problem on Rx side: Since there's no boundary to differentiate carrier, the FW does not know whether the non-carrier signal duration of the final bit is 560us or 1690us, and fails to recognize whether it's logical "0" or logical "1".

To solve this problem, the Data signal should be followed by a "stop" signal which is defined as 560us carrier signal + 500us non-carrier signal.

Thus, the FifoTask_data mainly contains the three parts below:

- (1) start signal: 9ms carrier signal + 4.5ms low level signal (no carrier)
- (2) data signal: address+ ~address+ cmd + ~cmd
- (3) stop signal: 560us carrier signal + 500us non-carrier signal

According to the above 3 signals, configure the Dma Fifo buffer to start the IR transmission, which is partially implemented in the ir_nec_send function, where part of the relevant code is as follows.

```
// set waveform input in sequence
    T_dmaData_buf.data_num = 0;
    //waveform for start bit
    T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_start_bit_1st;
    T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_start_bit_2nd;

    //add data
    u32 data = (~cmd)<<24 | cmd<<16 | addr2<<8 | addr1;
    for(int i=0;i<32;i++){
        if(data & BIT(i)){
            //waveform for logic_1
            T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_1_1st;
            T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_1_2nd;
```



```

    }
    else{
        //waveform for logic_0
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_0_1st;
        T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_logic_0_2nd;
    }
}

//waveform for stop bit
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_stop_bit_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_stop_bit_2nd;
T_dmaData_buf.dma_len = T_dmaData_buf.data_num * 2;

```

12.2.5.2 FifoTask_idle

As introduced in IR timing design, FifoTask_idle lasts for the duration "110ms - T_data". Record the time when FifoTask_data starts:

```
ir_send_ctrl.sending_start_time = clock_time();
```

Then calculate FifoTask_idle time in the interrupt triggered when FifoTask_data is finished:

```
110ms - (clock_time() - ir_send_ctrl.sending_start_time)
```

Demo code:

```

u32 tick_2_repeat_sysClockTimer16M = 110*CLOCK_16M_SYS_TIMER_CLK_1MS - (clock_time()
↳ ir_send_ctrl.sending_start_time);
u32 tick_2_repeat_sysTimer = (tick_2_repeat_sysClockTimer16M*CLOCK_PWM_CLOCK_1US>>4);

```

Please pay attention to the switch of time unit. As introduced in Clock module, Sytem Timer frequency used in software timer is fixed as 16MHz. Since PWM clock is derived from system clock, user needs to consider the case with system clock rather than 16MHz (e.g. 24MHz, 32MHz).

FifoTask_idle does not send PWM waveform, which can be considered to continually send non-carrier signal. It can be implemented by setting the first parameter "carrier_en" of the API "pwm_config_dma_fifo_waveform" as 0.

```

waveform_wait_to_repeat = pwm_config_dma_fifo_waveform(0, PWM0_PULSE_NORMAL,
↳ tick_2_repeat_sysTimer/PWM_CARRIER_CYCLE_TICK);

```

12.2.5.3 FifoTask_repeat

As per Demo IR protocol, repeat signal is 9ms carrier signal + 2.25ms non-carrier signal.

Similar to the processing of FifoTask_data, the end of repeat signal should be followed by 560us carrier signal as stop signal.

As introduced in IR timing design, repeat signal lasts for 110ms, so the duration of non-carrier signal after the 560us carrier signal should be:

$$110\text{ms} - 9\text{ms} - 2.25\text{ms} - 560\text{us} = 99190\text{us}$$

The code below shows the configuration for a complete repeat signal.

```
//repeat signal first part, 9000 us carrier, 2250 us low
    waveform_repeat_1st = pwm_config_dma_fifo_waveform(1, PWM0_PULSE_NORMAL, 9000 *
    ↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    waveform_repeat_2nd = pwm_config_dma_fifo_waveform(0, PWM0_PULSE_NORMAL, 2250 *
    ↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

//repeat signal second part, 560 us carrier, 99190 us low
    waveform_repeat_3rd = pwm_config_dma_fifo_waveform(1, PWM0_PULSE_NORMAL, 560 *
    ↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    waveform_repeat_4th = pwm_config_dma_fifo_waveform(0, PWM0_PULSE_NORMAL, 99190 *
    ↪ CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_2nd;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_3rd;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_4th;
```

12.2.5.4 FifoTask_repeat*n and FifoTask_idle_repeat*n

By simple superposition in DMA Fifo buffer, "FifoTask_repeat*n" and "FifoTask_idle_repeat*n" can be implemented on the basis of FifoTask_idle and FifoTask_repeat.

12.2.6 Check IR Busy Status in APP Layer

In the Application layer, user can use the variable "ir_send_ctrl.is_sending" to check whether IR is busy sending data or repeat signal.

The following shows the determination of whether IR is busy in power management. When IR is busy, MCU cannot enter suspend.

```
if( ir_send_ctrl.is_sending)
{
    bls_pm_setSuspendMask(SUSPEND_DISABLE);
}
```

12.3 IR Learn

12.3.1 IR Learn introduction

The IR learning is done by using the characteristics of IR tube to transmit and receive IR signals, and using the amplifier circuit to amplify and convert the received weak signals into digital signals, thus completing the learning of IR waveforms. After learning, the relevant data is stored in RAM/Flash, and then the learned waveform is sent out using the transmitting characteristics of the IR tube.

12.3.2 IR Learn hardware principle

The hardware circuit of IR learn is shown as below.

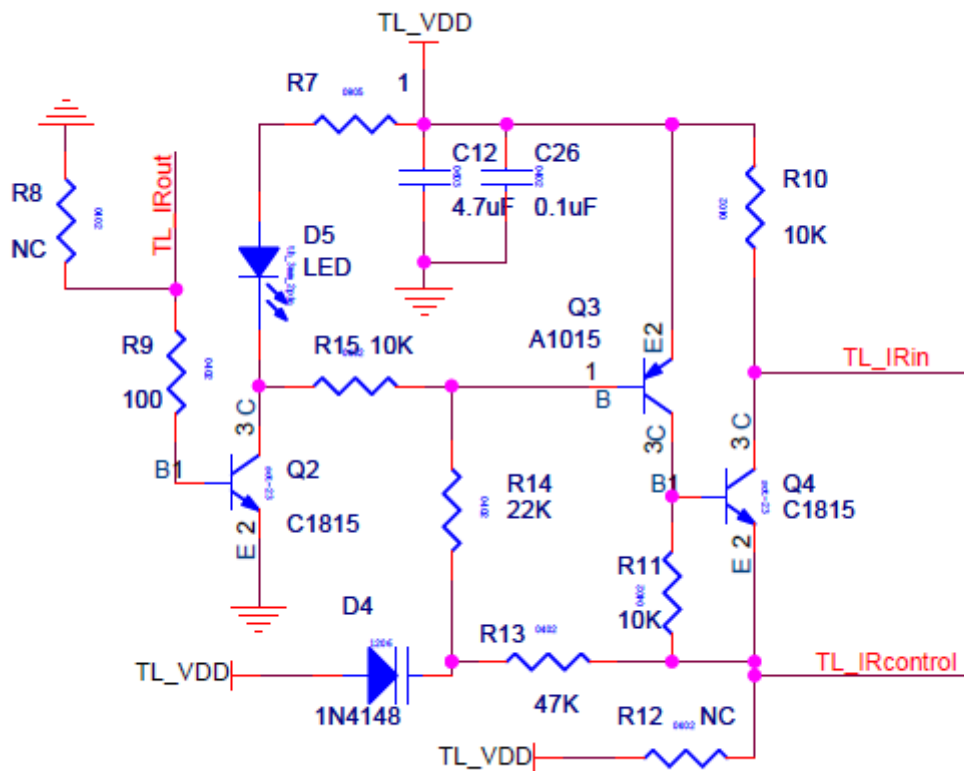


Figure 12.6: "IR Learn hardware circuit"

When in the IR learning state, IR_OUT and IR_CONTROL pin should be set to GPIO function and pulled low at the same time, then Q2 and Q3 will be in the cutoff state, IR_IN level is high when there is no waveform, and then will follow the waveform received by the triode and change: when the input waveform is high, IR_IN is pulled low, on the contrary IR_IN back to high. IR learning also takes advantage of this feature, using GPIO low level trigger to complete the learning algorithm, which will be described in detail later. As shown in the figure below, the transmitter is using NEC format IR, and the waveform of IR_IN is captured as shown in the figure below.

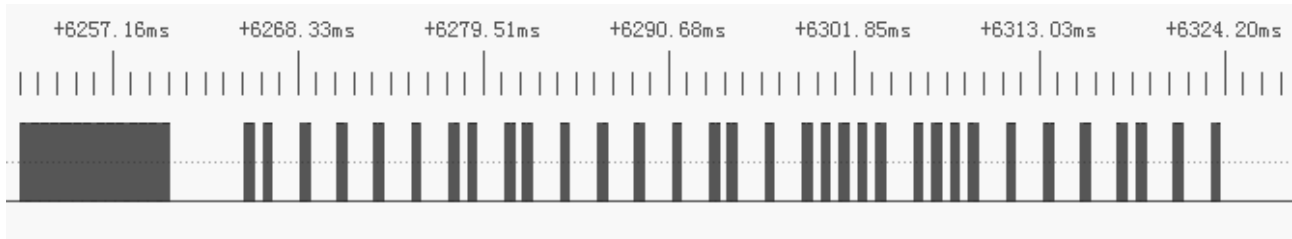


Figure 12.7: "IR_IN waveform of NEC protocol"

The dark part is the carrier waveform and the white part is the non-carrier waveform. The waveform of the amplified carrier part is shown in the figure below, which is high when no IR signal is received in front, and IR_IN is pulled down when a signal is received. The IR_IN low level is 9.35us and the period is 26.4us, which is converted to a carrier frequency of 37.88kHz. This matches the NEC protocol carrier of 38kHz with a duty cycle of 1/3.

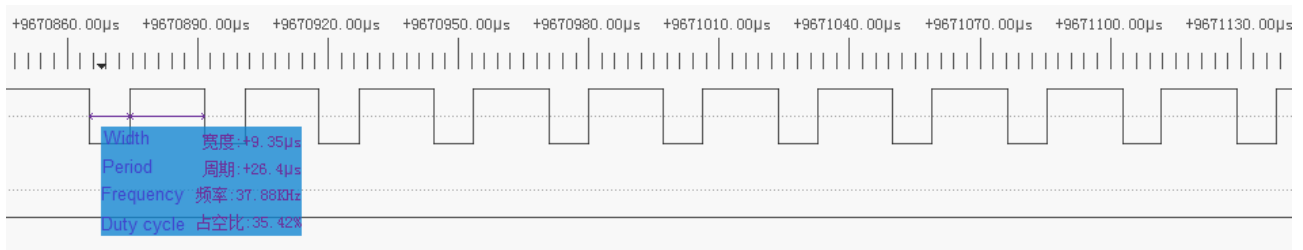


Figure 12.8: "IR_IN waveform of NEC carrier"

12.3.3 IR Learn software principle

During IR learning, the chip will set and enable the IR_IN trigger interrupt at falling edge. Every time it receives an IR carrier waveform from its device, IR_IN will be pulled low and trigger the interrupt. In the interrupt, the timing, number of waveforms, and carrier period of the carrier and non-carrier waveforms will be recorded by the algorithm, and the waveforms will be copied and sent out according to the above information when sending.

The following figure shows the order of recording during interrupt processing, the duration of the carrier/non-carrier part shown in the previous 1, 2... will be recorded in the buff. At the same time, the duration of a certain number of single carriers constituting 1 is recorded, and the carrier frequency f_c of the waveform is obtained by averaging. When the waveform is sent after the recording is completed, the carrier frequency f_c is fixed with a duty cycle of 1/3, and the time corresponding to 1 and 2 is sent out in the carrier/non-carrier order to complete the IR learning process.

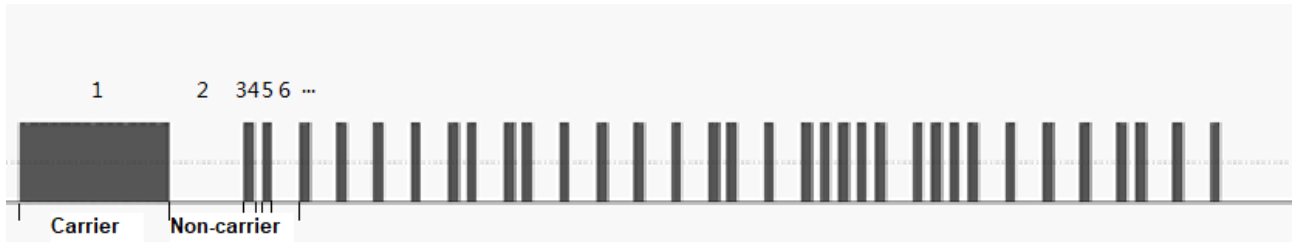


Figure 12.9: "Carrier and non-carrier"

To quickly complete the IR learning and sending function, the following steps are required.

- (1) Initialize with `ir_learn_init(void)`.
- (2) Add the relevant part of the `ir_learn_irq_handler(void)` interrupt handling function to the interrupt function.
- (3) Add the `ir_learn_detect(void)` section to the program to determine the learning results.
- (4) Modify the relevant macro definitions in `rc_ir_learn.h`.
- (5) Add the `ir_learn_start(void)` function to the appropriate location in the UI layer to start learning.
- (6) After judging the result by the judgment function set in step 3, use `get_ir_learn_state(void)` to check the IR learning status and do UI layer operations according to the success or failure of learning: if successful, continue steps 7-9 to finish sending, if failure, you can re-execute step 5 or perform other custom UI actions.
- (7) After successful learning, the learning result can be sent. The first step of sending is to initialize the IR transmission, using `ir_learn_send_init(void)`. Be noted that after calling this function `IR_OUT` will be changed to PWM output pin, if you want to re-enter the IR learning state, you must re-execute step 1 to re-initialize the pin function.
- (8) The second step of sending is to copy the useful parameters of the learning result to a fixed area, RAM/Flash are suitable, use the `ir_learn_copy_result(ir_learn_send_t* send_buffer)` function to copy to the structure defined for sending the IR learning result.
- (9) The final step of sending is to call the `ir_learn_send(ir_learn_send_t* send_buffer)` function to send the learning results.

At this point, the entire functionality of IR learning has been implemented. In the following section, we will specify how to add the functions mentioned in the steps, one by one, in the order of the above steps.

12.3.3.1 IR_Learn initialization

When using the IR Learn function, after copying `rc_ir_learn.c` and `rc_ir_learn.h` into the project, the first step is to call the initialization function.

```
void ir_learn_init(void)
```

This function finds its entity in `rc_ir_learn.c`. It first clears the structure used, then sets `IR_OUT` and `IR_CONTROL` to GPIO and outputs 0. Then it sets the GPIO interrupt enable and clears the interrupt flag bit.

12.3.3.2 IR_Learn interrupt handling

Since the IR Learn function is implemented based on interrupts, the second step requires adding interrupt handling functions to the interrupts. As the protocol stack will be constructed to enter the interrupt several times, in order to distinguish it is a GPIO interrupt, the interrupt flag bit will be read first and then recorded when it is an interrupt generated by GPIO. The implementation code is as follows.

```
void ir_learn_irq_handler(void)
{
    gpio_clr_irq_status(FLD_GPIO_IRQ_CLR);
    if ((g_ir_learn_ctrl -> ir_learn_state != IR_LEARN_WAIT_KEY) && (g_ir_learn_ctrl ->
        ↪ ir_learn_state != IR_LEARN_BEGIN))
    {
        return;
    }
    ir_record(clock_time()); // IR Learning
}
```

Where ir_record() is the specific learning algorithm, the function pre_attribute_ram_code_ is put into the ram in order to speed up the learning and avoid errors caused by long execution time.

12.3.3.3 IR_Learn result processing function

The main role of the result processing function is to change the state of IR learning in time according to the current IR learning situation, and each loop needs to be executed to complete the detection in time. The function can be called in the main_loop().

```
void ir_learn_detect(void)
```

As can be seen from the function entity, when the time after the start of learning exceeds IR_LEARN_OVERTIME_THRESHOLD, the waveform is still not received and it is a timeout failure; after learning has started and has received the signal, the set threshold time passed but no new signal received, it is considered to have completed the learning state, at this time, if the received carrier and non-carrier part exceeds the set number (default is 15), the learning is considered successful, otherwise it is considered failed.

12.3.3.4 IR_Learn macro definition

To increase extensibility, some macro definitions are added to rc_ir_learn.h.

```
#define GPIO_IR_OUT          PWM_PIN    // GPIO_PE3
#define GPIO_IR_CONTROL      GPIO_PE0
#define GPIO_IR_LEARN_IN     GPIO_PE1
```

The first three define the GPIO pins, IN/OUT/CONTROL, which change according to the specific design.

12.3.3.5 IR_Learn start function

The IR Learn start function is called where needed in the UI layer to start the IR learning process. The function is as follows.

```
ir_learn_start();
```

12.3.3.6 IR_Learn state query

Users can call the status query function to query the learning results, the function is as follows.

```
unsigned char get_ir_learn_state(void)
{
    if(g_ir_learn_ctrl -> ir_learn_state == IR_LEARN_SUCCESS)
        return 0;
    else if(g_ir_learn_ctrl -> ir_learn_state < IR_LEARN_SUCCESS)
        return 1;
    else
        return (g_ir_learn_ctrl -> ir_learn_state);
}
```

Return value = 0: IR learning is successful.

Return value = 1: IR learning is in progress or not started.

Return value > 1: IR learning failed, the return value is the reason for failure, which corresponds to the reason for failure known in `ir_learn_states`. The `ir_learn_states` is defined as follows.

```
enum {
    IR_LEARN_DISABLE = 0x00,
    IR_LEARN_WAIT_KEY,
    IR_LEARN_KEY,
    IR_LEARN_BEGIN,
    IR_LEARN_SAMPLE_END,
    IR_LEARN_SUCCESS,
    IR_LEARN_FAIL_FIRST_INTERVAL_TOO_LONG,
    IR_LEARN_FAIL_TWO_LONG_NO_CARRIER,
    IR_LEARN_FAIL_WAIT_OVER_TIME,
    IR_LEARN_FAIL_WAVE_NUM_TOO_FEW,
    IR_LEARN_FAIL_FLASH_FULL,
    IR_LEARN_FAIL,
}ir_learn_states;
```

12.3.3.7 IR_Learn_Send initialization

After the UI layer determines that the learning is successful, the send initialization function needs to be called before sending the learned waveform, and the function is as follows.

```
void ir_learn_send_init(void)
```

The initialization function mainly sets PWM-related parameters, interrupt-related parameters, and sets IR_OUT as the output port of PWM, noting that the IR learning function stops after this function is used, and the initialization function described in 11.3.4.1 needs to be called again if it needs to be enabled again.

12.3.3.8 IR_Learn result copy function

In the design, there are often cases where several keys need to have IR learning functions, so the UI layer wants to be able to copy the learning results to a location in RAM/Flash for later transmission after successful learning, and to start the learning process for other keys. Therefore, a result copy function is provided to copy the necessary parameters for sending. The function is as follows.

```
void ir_learn_copy_result(ir_learn_send_t* send_buffer)
```

The send_buffer is the structure needed for IR learning to send, which contains the clock_tick value for one carrier cycle, the total number of carriers and non-carriers (counting from 0), and the buffer of carriers and non-carriers already to be sent.

```
typedef struct{
    unsigned int    ir_learn_carrier_cycle;
    unsigned short  ir_learn_wave_num;
    unsigned int    ir_learn_send_buf[MAX_SECTION_NUMBER];
}ir_learn_send_t;
```

12.3.3.9 IR_Learn send function

After the learning is successful and the pre-send operation is done, the send function can be called to send the learning result. The function is as follows.

```
void ir_learn_send(ir_learn_send_t* send_buffer);
```

where send_buffer is the structure used in the previous function. The send function does not carry the repeat function, each call to the function will send the learned waveform, if you need to repeat the user can use the timer in the UI layer to design their own repeated calls to the function.

12.3.4 IR Learn algorithm details

To facilitate understanding the code, the principle of the IR learning algorithm is explained in detail here. The following is a simulated waveform, which simulates a complete packet of IR data. The data contains Start carrier, Start No carrier, bit 1 carrier, bit 1 no carrier, bit 0 carrier, bit 0 no carrier, End carrier, End no carrier.

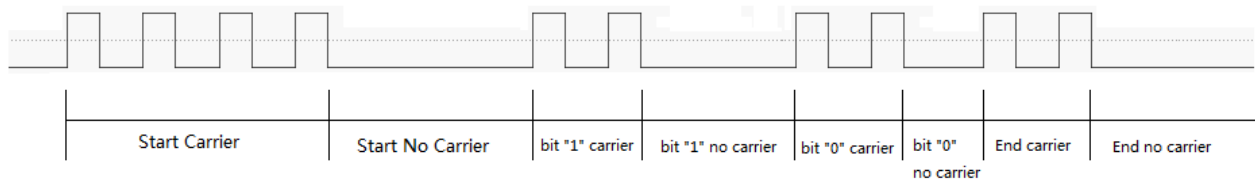


Figure 12.10: "A frame of IR code"

Since IR_IN is set in the IR learning state to wake up on the falling edge of the GPIO, normally every falling edge goes to an interrupt where we do the recording operation. In the IR learning algorithm, instead of identifying the waveform to a specific code type, the waveform is recorded with the concept of carrier/non-carrier. Consecutive carriers are considered as one carrier segment, while two carriers separated by a long time are considered as non-carriers. Thus the above is considered in the IR learning algorithm as follows.

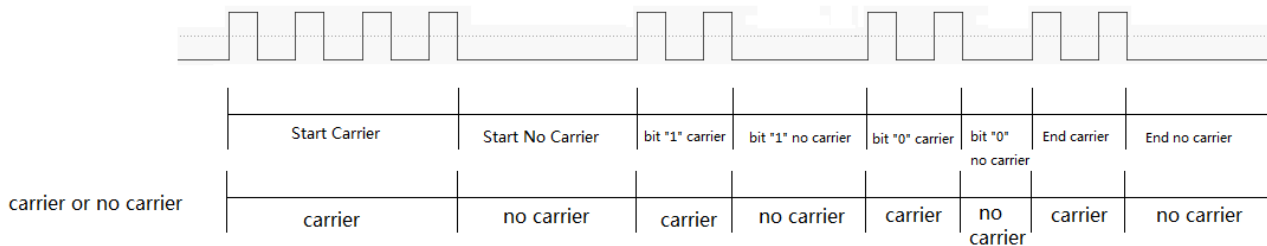


Figure 12.11: "Carrier and no carrier in IR Learn"

Each time the algorithm is executed, the current time `curr_trigger_tm_point` is recorded, and the `last_trigger_tm_point` is subtracted from the last interrupt time to get a `time_interval` of one cycle. If this time is relatively small, it is considered to be still in the carrier; if this time exceeds the set threshold, it is considered to be in the middle of a no carrier segment, and at this time it is in the first waveform of the new carrier segment: at this time, it is necessary to record the last carrier time and put it into the buffer, which is the difference between the first interrupt entry time and the last interrupt time, as shown below.

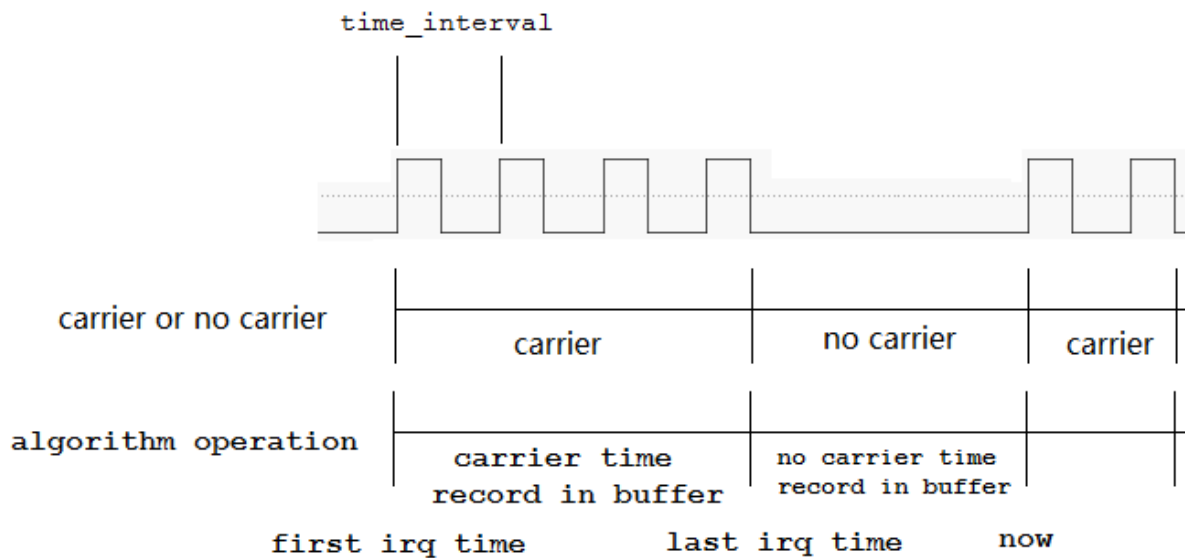


Figure 12.12: "IR learn algorithm"

According to this method, let `wave_series_cnt` increase from 0, corresponding to the first carrier segment, the first non-carrier segment, the second carrier segment, the second non-carrier segment... At the same time, the calculated time of each segment is stored in the corresponding location (`wave_series_buf[wave_series_cnt]`) in `wave_series_buf[0]`, `wave_series_buf[1]`, and `wave_series_buf[2]`. All the way to the end of the waveform, `wave_series_cnt` represents the total number of segments, and `wave_series_buf` is loaded with the length of each segment.

In addition, during the first N (settable) interrupts, N times are recorded, and the smallest one of them is taken as the carrier period, which is used when sending after the learning is finished, and the duty cycle is 1/3 (settable) by default.

After the IR learning process is finished, the learning result can be sent. When sending the learning result, it is also sent according to the concept of carrier and non-carrier. Using PWM DMA_FIFO mode, after putting the learned carrier frequency, duty cycle, and duration of each segment into DMA buffer, enable DMA, the chip will automatically send out the learned waveform until all the sending is finished, and generate `FLD_IRQ_PWM0_IR_DMA_FIFO_DONE` interrupt.

12.3.5 IR Learn learning parameter adjustment

Some parameters related to IR learning are defined in `rc_ir_learn.h`. When setting the parameter mode to `USER_DEFINE` is selected and set by yourself, it will have different effects on the learning effect, which will be described in detail here.

```
#define IR_LEARN_MAX_FREQUENCY 40000
#define IR_LEARN_MIN_FREQUENCY 30000

#define IR_LEARN_CARRIER_MIN_CYCLE 16000000/IR_LEARN_MAX_FREQUENCY
```

```
#define IR_LEARN_CARRIER_MIN_HIGH_TICK IR_LEARN_CARRIER_MIN_CYCLE/3
#define IR_LEARN_CARRIER_MAX_CYCLE 16000000/IR_LEARN_MIN_FREQUENCY
#define IR_LEARN_CARRIER_MAX_HIGH_TICK IR_LEARN_CARRIER_MAX_CYCLE/3
```

The above parameters set the frequencies supported by IR learning. The default value is set to 30k~40k. The following parameters are the maximum and minimum values of the sys_tick value per carrier cycle, default 1/3 duty cycle high level to sys_tick value, calculated from the frequency parameters for later parameter calculation. Other parameters that affect the learning results are described below, and each parameter is defined in rc_ir_learn.h using macros.

```
#define IR_LEARN_INTERVAL_THRESHOLD (IR_LEARN_CARRIER_MAX_CYCLE*3/2)
#define IR_LEARN_END_THRESHOLD (30*SYSTEM_TIMER_TICK_1MS)
#define IR_LEARN_OVERTIME_THRESHOLD 10000000 // 10s
#define IR_CARR_CHECK_CNT 10
#define CARR_AND_NO_CARR_MIN_NUMBER 15
#define MAX_SECTION_NUMBER 100
```

(1) IR_LEARN_INTERVAL_THRESHOLD.

For carrier period threshold, the default value is 1.5 times the IR_LEARN_CARRIER_MAX_CYCLE value, when the time to enter the interrupt twice is more than this threshold is considered at the carrier side.

(2)IR_LEARN_END_THRESHOLD

For IR learn end threshold, when the time to enter interrupt twice exceeds this threshold, or the threshold is exceeded without entering the next interrupt, the IR learning process is considered to be finished.

(3) IR_LEARN_OVERTIME_THRESHOLD

For timeout time, after the start of IR learning process, if the threshold value is exceeded and the received waveform enters interrupt, the learning process is considered to be finished and failed.

(4) IR_CARR_CHECK_CNT

Set the number of packets to be collected to determine the carrier cycle time, the default is set to 10, which means the smallest of the time_interval of the first 10 interrupts will be taken as the carrier time and used to calculate the carrier cycle when sending learning results.

(5) CARR_AND_NO_CARR_MIN_NUMBER

The minimum threshold of carrier and non-carrier segments. When the IR learning process is completed, if the total number of recorded carrier and non-carrier segments is less than this threshold, the entire waveform is considered not learned and the IR learning fails.

(6) MAX_SECTION_NUMBER

The maximum threshold value of carrier and non-carrier section, which will be used when setting the buffer size. If setting to 100, the IR learning process will record at most 100 carrier and non-carrier sections; if it exceeds, the IR learning will be considered failed.

12.3.6 IR Learn common issues

During the learning process, sometimes it encounters that the frequency of the waveform sent after successful learning changes. The possible cause is that the frequency of the learned waveform is too high, resulting in the execution of the algorithm in the interrupt for more than the carrier period. This is shown in the figure below.

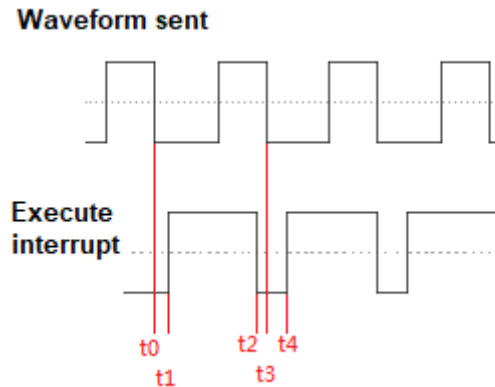


Figure 12.13: "IR learn error"

Take the IR signal with duty cycle 1/3 and transmitting frequency 38K as an example, one carrier cycle is about 26.3us, high level accounts for 1/3 about 8.7us. At the moment of t0, the external waveform carrier end point is pulled low from high, the chip GPIO triggers an interrupt, and the interrupt needs to execute several instructions in the assembly to save the site to enter the interrupt, after testing at t1 after about 4us to enter the interrupt function to start executing the operation. Due to the long execution time in the interrupt, the interrupt execution ends at t2, and it also takes about 4us to restore the site. In the process of restoring the site at t3 moment, as the next falling edge of the transmit waveform arrives, the interrupt flag bit is cleared at this time and the hardware will trigger the interrupt again. The interrupt has been triggered again after restoring the site about 4us after t2, so the chip saves the site again to enter the interrupt at t4 after 4us in entering the interrupt for operation, after which the above process will be repeated. As seen by the waveform executed by the interrupt, its time is completely deformed and the time to enter the interrupt twice is also larger than the time of one carrier cycle of the original waveform. Since the IR learning is done exactly according to the time recorded in the interrupt, the abnormal time of entering the interrupt will lead to abnormal IR learning results.

There are several ways to solve this problem.

One is to put the IR learning algorithm into the ram_code to reduce the execution time, by default this operation is already performed and does not need to be modified.

The second is to make sure to reduce other processing of interrupts. BLE needs to be disabled in IR learning because it takes up a lot of time in interrupts during non-IDLE states, and the UI layer also tries to prohibit other interrupt sources from causing interrupts during IR learning to prevent exceptions.

12.4 Demo description

The feature_IR of the BLE SDK contains the normal IR sending function and IR learning function, and the IR encoding method used is NEC encoding. The switch between the different modes is shown in the following code.

```
void key_change_proc(void)
{
    switch(key0)
    {
        .....
        if(switch_key == IR_mode){.....
        }
        else if(switch_key == IR_Learn_mode){.....
        }
    }
    else{.....
    }
}
}
```

Each mode can be switched to a different mode by pressing a key to perform the corresponding initialization operation, the specific code implementation can be referred to the BLE SDK.

13 Feature Demo Introduction

B85m_feature_test provides demo codes for some commonly used BLE-related features. Users can refer to these demos to complete their own function implementation. See code for details. Select the macro "FEATURE_TEST_MODE" in app_config.h in the B91_feature_test project to switch to the demo of different feature test.

```

//////////////////////////////// TEST FEATURE SELECTION //////////////////////////////////

//power test
#define TEST_POWER_ADV 10
#define TEST_POWER_CONN 11

//smp test
#define TEST_SMP_SECURITY 20 //If testing SECURITY, such as Passkey Entr

//gatt secure test
#define TEST_GATT_SECURITY 21 //If testing SECURITY, such as Passkey Entr

//slave data length exchange test
#define TEST_SDATA_LENGTH_EXTENSION 22

//other test
#define TEST_USER_BLT_SOFT_TIMER 30
#define TEST_WHITELIST 31
//phy test
#define TEST_BLE_PHY 32 // BQB PHY_TEST demo
#define TEST_EMI 33 // EMI Test demo

#define TEST_EXTENDED_ADVERTISING 40 // Extended ADV demo

#define TEST_2M_CODED_PHY_EXT_ADV 50 // 2M/Coded PHY used on Extended ADV
#define TEST_2M_CODED_PHY_CONNECTION 60 // 2M/Coded PHY used on Legacy_ADV/Ex

#define TEST_STUCK_KEY 90
#define TEST_AUDIO 91
#define TEST_IR 92
#define TEST_L2CAP_PREPARE_WRITE_BUFF 93

#define TEST_OTA 95

#define TEST_FEATURE_BACKUP 200

#define FEATURE_TEST_MODE TEST_FEATURE_BACKUP // TEST_FEATURE_BACKUP

```

Figure 13.1: "Feature Test Demo"

Test methods of each demo are described below.

13.1 Broadcast Power Consumption Test

This item mainly tests the power consumption during broadcasting of different broadcasting parameters. Users can measure the power consumption with an external multimeter during the test. Need to modify FEATURE_TEST_MODE to TEST_POWER_ADV in app_config.h.

```
#define FEATURE_TEST_MODE          TEST_POWER_ADV
```

Modify the broadcast type and broadcast parameters in feature_adv_power as required. There are two types of broadcasts provided in Demo: connectable broadcast and non-connectable broadcast.

13.1.1 Connectable Broadcast Power Consumption Test

In feature_adv_power, the default test non-connectable broadcast power consumption needs to be changed from #if 0 to #if 1, as shown in the following code.

```
#if 1    //connectable undirected ADV
```

The default broadcast data length of Demo is 12 bytes, and users can modify it according to their needs.

```
//ADV data length: 12 byte
u8 tbl_advData[12] = {0x08, 0x09, 't', 'e', 's', 't', 'a', 'd', 'v', 0x02, 0x01, 0x05,};
```

The Demo provides 1s1channel, 1s3channel, 500ms3channel broadcast parameters, users can select the corresponding test items according to their needs.

13.1.2 Un-connectable Broadcast Power Consumption Test

In feature_adv_power, the default test is non-connectable broadcast power consumption.

```
#if 0    //un-connectable undirected ADV
```

The Demo provides two broadcast data lengths of 16byte and 31byte, which users can choose according to their needs.

```
#if 1    //ADV data length: 16 byte
    u8 tbl_advData[8] = {
        0x0C, 0x09, 't', 'e', 's', 't', 'a', 'd',
    };
#else    //ADV data length: max 31 byte
    u8 tbl_advData[] = {
        0x1E, 0x09, 't', 'e', 's', 't', 'a', 'd', 'v', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F',
        ↪ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D'
    };
#endif
```

The Demo provides 1s3channel, 1.5s3channel, and 2s3channel broadcast parameters. Users can select the corresponding test items according to their needs.

13.2 SMP Test

SMP test mainly tests the process of pairing encryption, mainly divided into the following ways:

- (1) LE_Security_Mode_1_Level_1, no authentication and no encryption.
- (2) LE_Security_Mode_1_Level_2, unauthenticated paring with encryption.
- (3) LE_Security_Mode_1_Level_3, authenticated paring with encryption-legacy.
- (4) LE_Security_Mode_1_Level_4, authenticated paring with encryption-sc.

Users need to set FEATURE_TEST_MODE to TEST_SMP_SECURITY in app_config.h.

```
#define FEATURE_TEST_MODE          TEST_SMP_SECURITY
```

Below is a brief introduction to each pairing mode.

13.2.1 LE_Security_Mode_1_Level_1

LE_Security_Mode_1_Level_1 is the simplest pairing method, neither authentication nor encryption. The user changes the SMP_TEST_MODE of feature_security.c to SMP_TEST_NO_SECURITY.

```
#define SMP_TEST_MODE          SMP_TEST_NO_SECURITY
```

13.2.2 LE_Security_Mode_1_Level_2

The LE_Security_Mode_1_Level_2 mode is just work, only encryption but not authentication. Just work is divided into legacy just work and sc just work. The user changes the SMP_TEST_MODE of feature_security.c to SMP_TEST_LEGACY_PARING_JUST_WORKS or SMP_TEST_SC_PARING_JUST_WORKS as required. Introduced separately below.

13.2.2.1 SMP_TEST_LEGACY_PARING_JUST_WORKS

The user makes the following modifications:

```
#define SMP_TEST_MODE          SMP_TEST_LEGACY_PARING_JUST_WORKS
```

The process is shown as following:

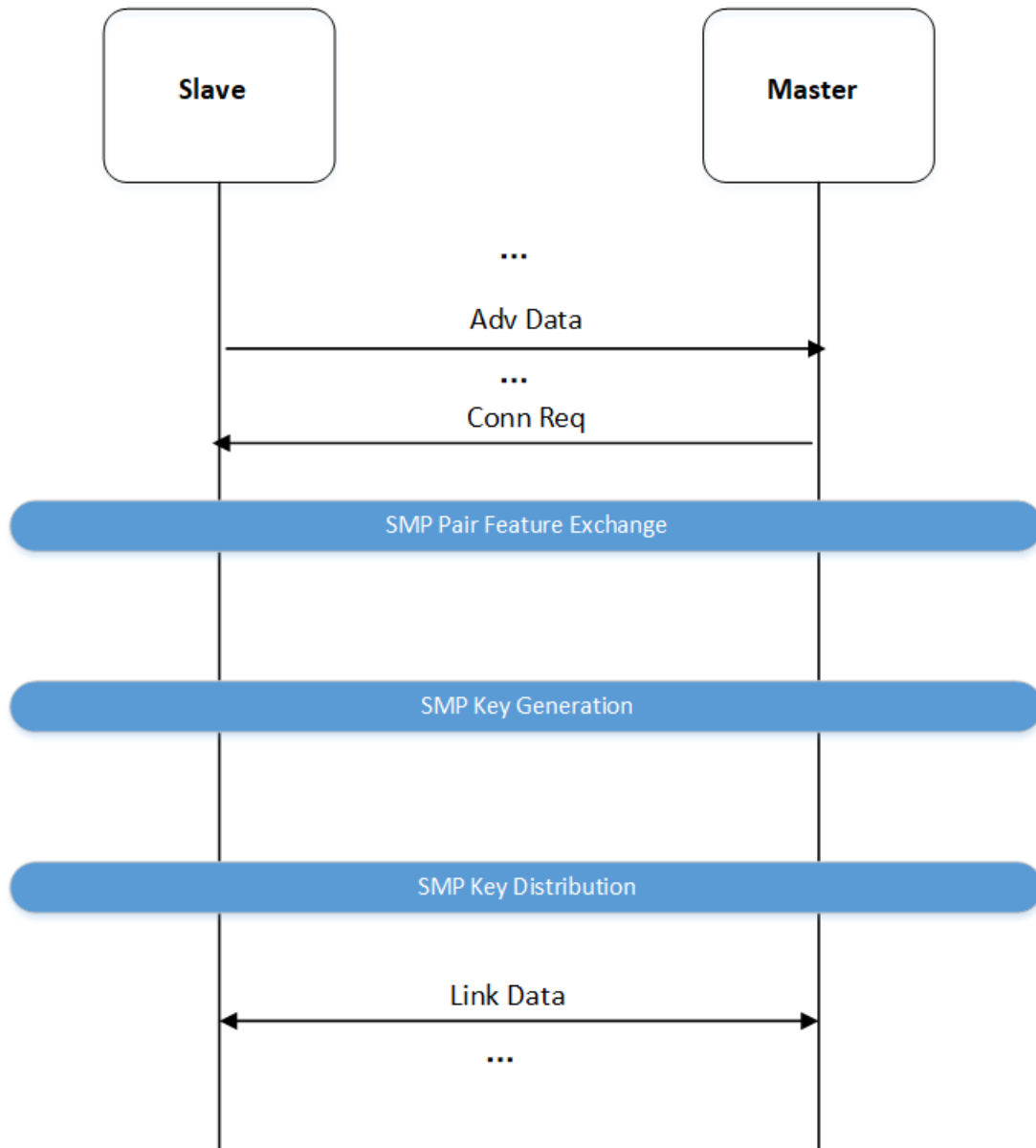


Figure 13.2: "Legacy Just Work Process"

13.2.2.2 SMP_TEST_SC_PAIRING_JUST_WORKS

The user makes the following modifications:

```
#define    SMP_TEST_MODE                SMP_TEST_SC_PAIRING_JUST_WORKS
```

The process is shown as following:

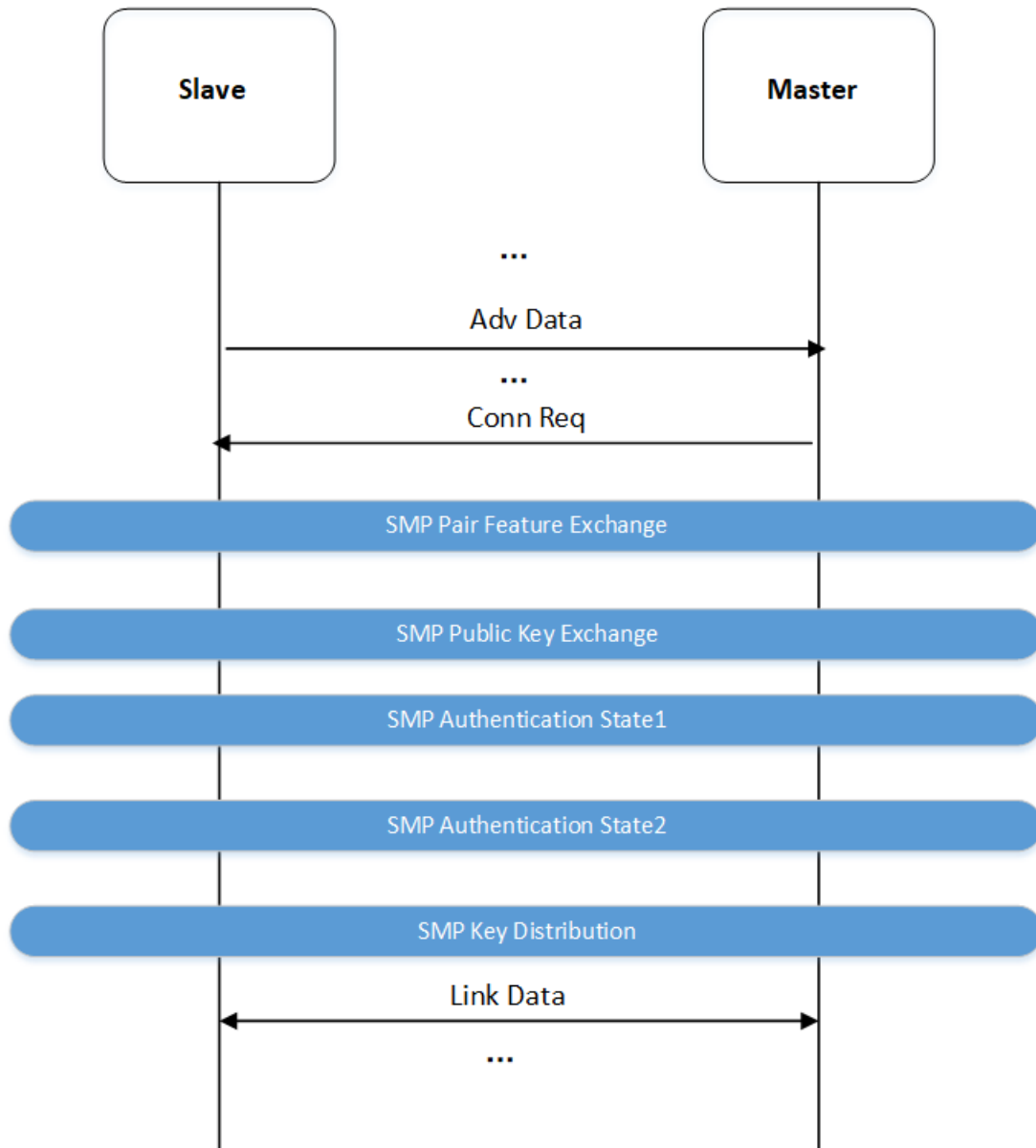


Figure 13.3: "SC Just Work Process"

13.2.3 LE_Security_Mode_1_Level_3

LE_Security_Mode_1_Level_3 is both the authentication and encryption Legacy pairing method. According to the pairing parameter settings, it is divided into OOB, PassKey Entry, and Numeric Comparison. Currently the demo provides two sample codes for PassKey Entry, namely SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI and SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI. Users can choose according to their needs. The two methods are briefly introduced below.

13.2.3.1 SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI

The user needs to modify as follows in feature_security.c:

```
#define SMP_TEST_MODE SMP_TEST_LEGACY_PASSKEY_ENTRY_SDMI
```

During the pairing process, the slave side needs to display the key and the master side enters the key. During initialization, a gap event related to pairing is registered. The pairing information will be notified to the app layer.

```
blc_gap_registerHostEventHandler( app_host_event_callback );
blc_gap_setEventMask( GAP_EVT_MASK_SMP_PAIRING_BEAGIN | \
                      GAP_EVT_MASK_SMP_PAIRING_SUCCESS | \
                      GAP_EVT_MASK_SMP_PAIRING_FAIL | \
                      GAP_EVT_MASK_SMP_TK_DISPALY | \
                      GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE );
```

The user needs to print the current key information when receiving the GAP_EVT_MASK_SMP_TK_DISPLAY message.

```
int app_host_event_callback (u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;
    switch(event)
    {
    ...
        case GAP_EVT_SMP_TK_DISPALY:
        {
            char pc[7];
            u32 pinCode = *(u32*)para;
        }
        break;
    ...
    }
}
```

The process is shown as following:

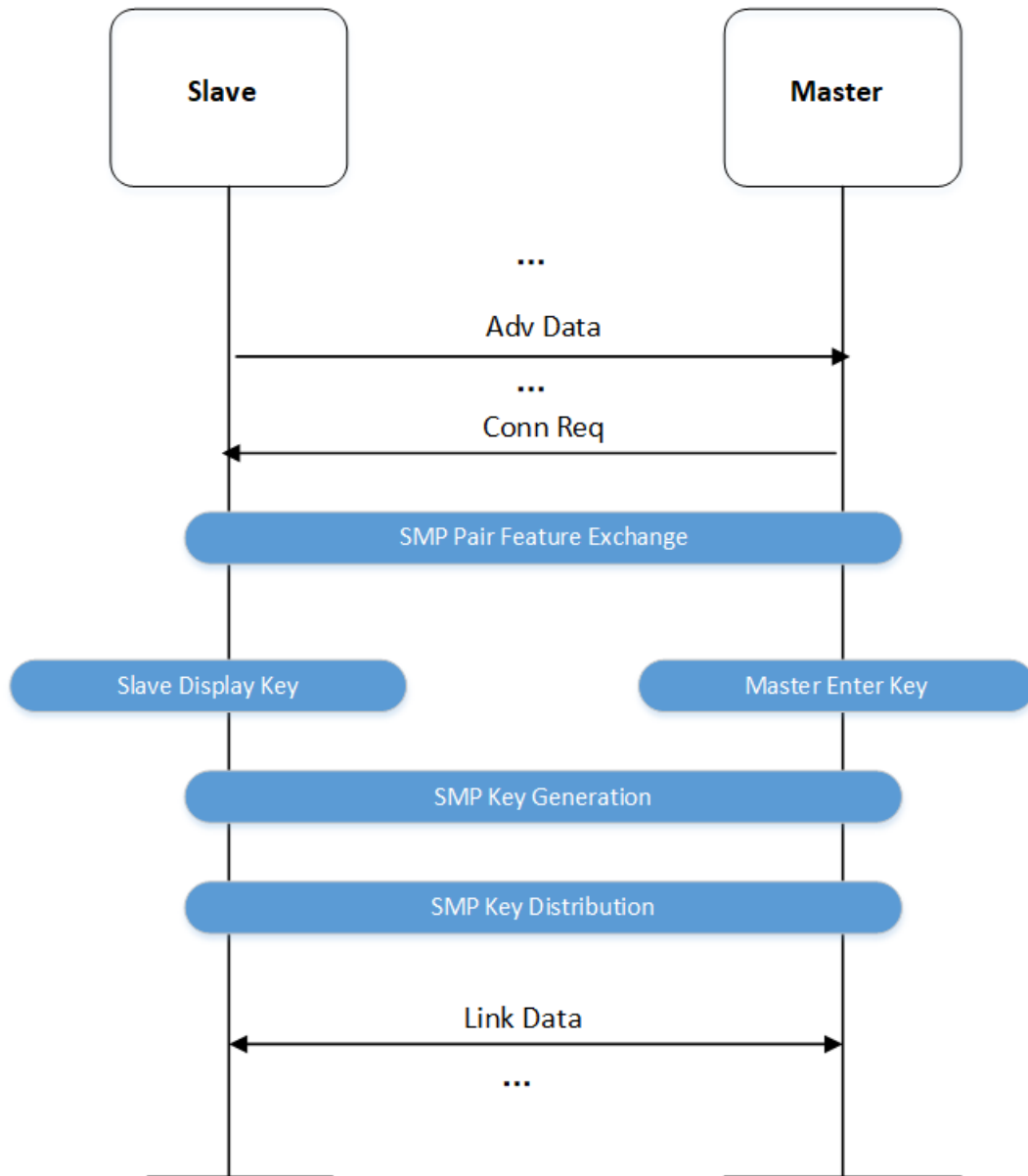


Figure 13.4: "Legacy Just Work SDMI Process"

13.2.3.2 SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI

The difference from the above is that the key is displayed on the master and the key is entered by the slave. The user needs to modify the code:

```
#define SMP_TEST_MODE SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI
```

The process is shown as following:

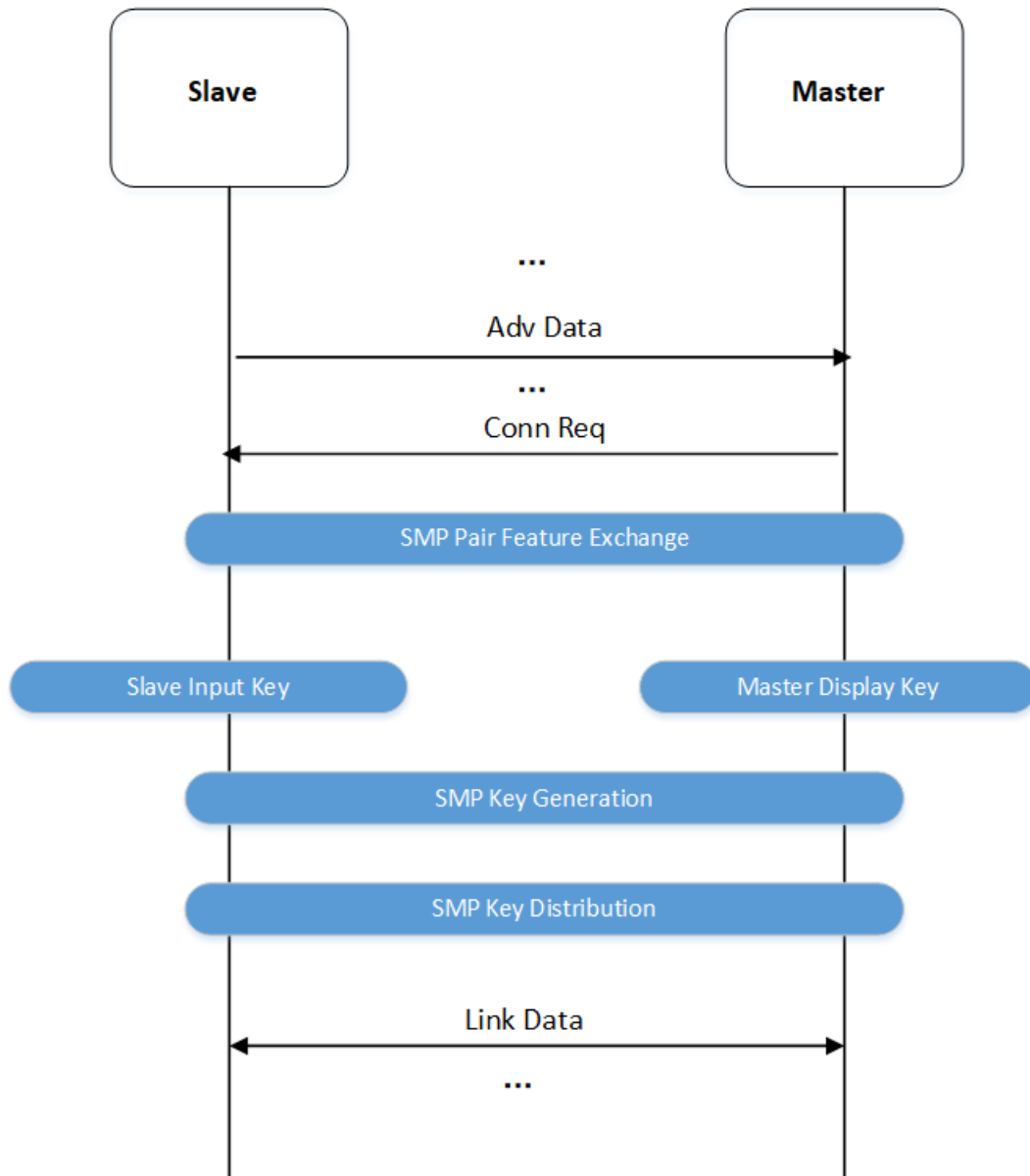


Figure 13.5: "Legacy Just Work SIMD Process"

13.2.4 LE_Security_Mode_1_Level_4

LE_Security_Mode_1_Level_4 is both an authentication and encryption SC pairing method. According to the pairing parameter settings, it is divided into OOB, PassKey Entry, and Numeric Comparison. Currently the demo provides three sample codes of SC PassKey Entry and SC Numeric Comparison, namely SMP_TEST_SC_PASSKEY_ENTRY_SDMI, SMP_TEST_SC_PASSKEY_ENTRY_MDSI and SMP_TEST_SC_NUMERIC_COMPARISON. Users can choose according to their needs. These methods are briefly introduced below.

13.2.4.1 SMP_TEST_SC_NUMERIC_COMPARISON

The user needs to modify as follows in feature_security.c:

```
#define      SMP_TEST_MODE      SMP_TEST_SC_NUMERIC_COMPARISON
```

This pairing method is numeric comparison, that is, during the pairing process, both the master and slave will display a six-digit PIN code. If the user compares the numbers for the same, if they are the same, click to confirm and agree to the pairing. Demo is to send YES or NO in the form of a button. The sample code is as follows:

```
if(consumer_key == MKEY_VOL_DN){  
    blc_smp_setNumericComparisonResult(1); // YES  
    /*confirmed YES*/  
    led_onoff(LED_ON_LEVAL);  
}  
else if(consumer_key == MKEY_VOL_UP){  
    blc_smp_setNumericComparisonResult(0); // NO  
    /*confirmed NO*/  
    led_onoff(LED_ON_LEVAL);  
}
```

The process is shown as following:

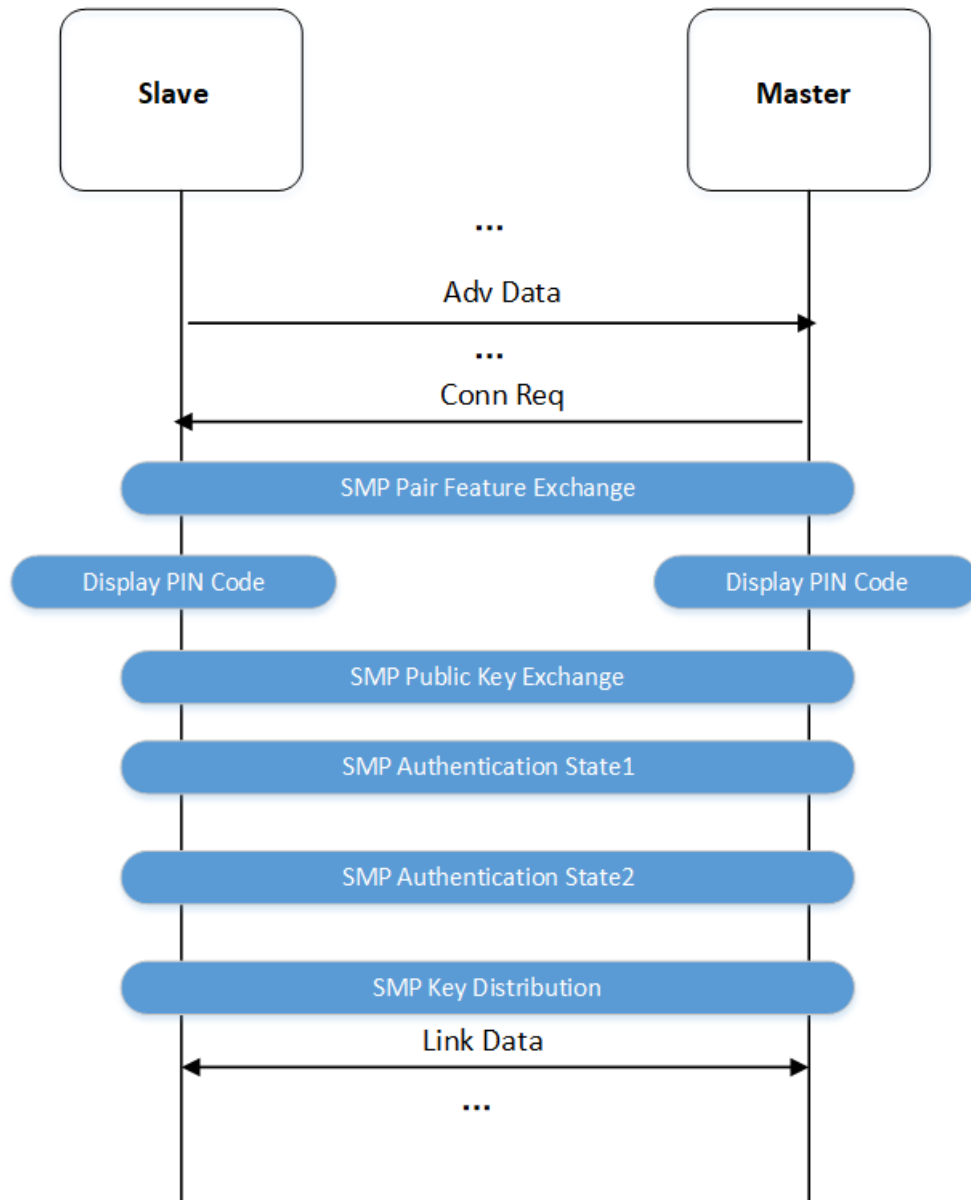


Figure 13.6: "Numeric Comparison Pairing"

13.2.4.2 SMP_TEST_SC_PASSKEY_ENTRY_SDMI

The user needs to modify as follows in feature_security.c:

```
#define SMP_TEST_MODE SMP_TEST_SC_PASSKEY_ENTRY_SDMI
```

During the pairing process, the slave side needs to display the key and the master side enters the key. During initialization, a gap event related to pairing is registered. The pairing information will be notified to the app layer.

```
blc_gap_registerHostEventHandler( app_host_event_callback );
blc_gap_setEventMask(      GAP_EVT_MASK_SMP_PARING_BEAGIN          | \
                           GAP_EVT_MASK_SMP_PARING_SUCCESS        | \
                           GAP_EVT_MASK_SMP_PARING_FAIL           | \
                           GAP_EVT_MASK_SMP_TK_DISPALY            | \
                           GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE );
```

The user needs to print the current key information when receiving the GAP_EVT_MASK_SMP_TK_DISPLAY message.

```
int app_host_event_callback (u32 h, u8 *para, int n)
{
    u8 event = h & 0xFF;
    switch(event)
    {
    ...
        case GAP_EVT_SMP_TK_DISPLAY:
        {
            char pc[7];
            u32 pinCode = *(u32*)para;
        }
        break;
    ...
    }
}
```

The process is shown as following:

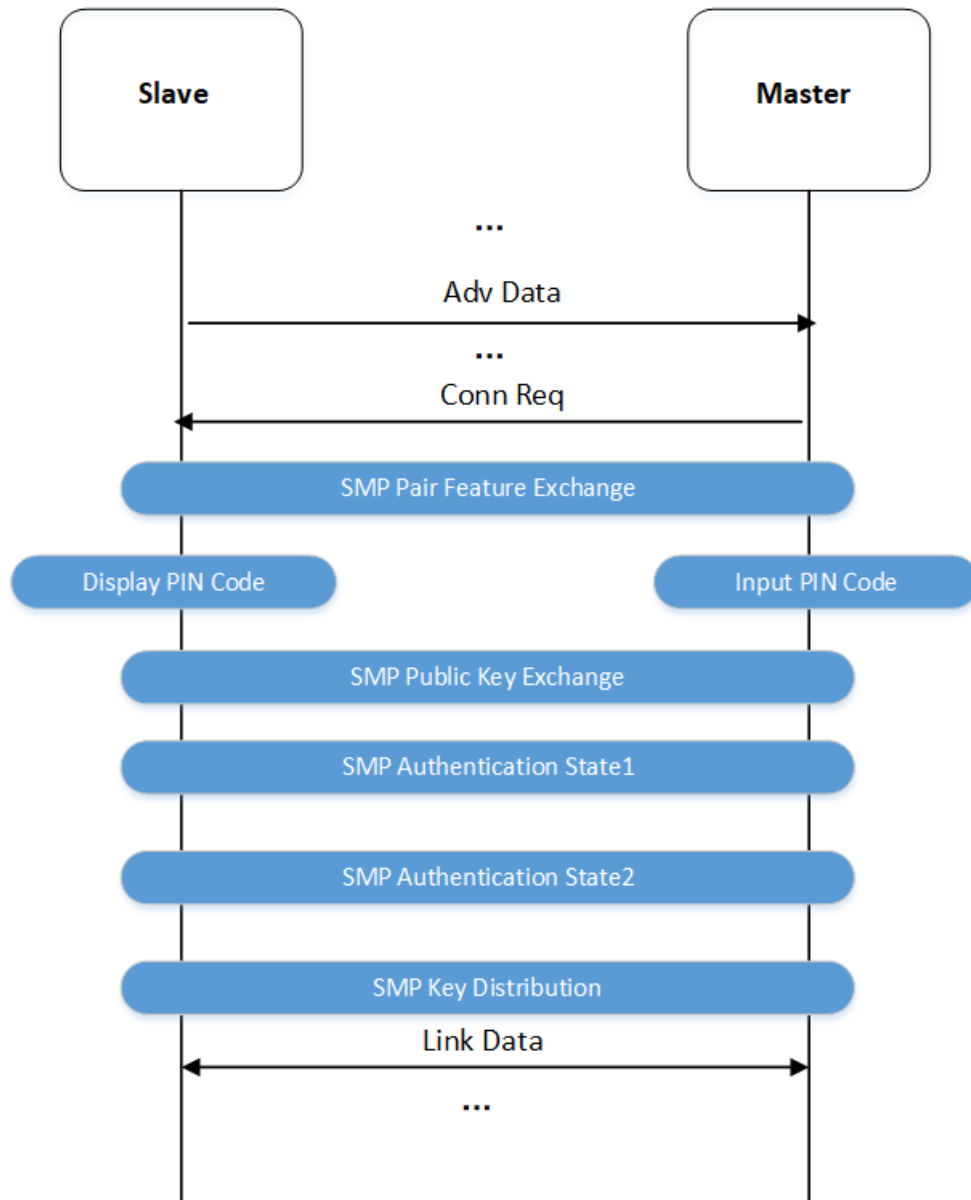


Figure 13.7: "SC SDMI Pairing Processing"

13.3 GATT Security Test

As known from the BLE module 3.3.3 ATT&GATT chapter, each Attribute in the service list defines read and write permissions, that is, the pairing mode must reach the corresponding level to read or write. For example, in the SPP service of Demo:

```

// client to server RX
{0,ATT_PERMISSIONS_READ,2,sizeof(TelinkSppDataClient2ServerCharVal),(u8*)&my_characterUUID),
↪ (u8*)(TelinkSppDataClient2ServerCharVal), 0}, //prop
{0,SPP_C2S_ATT_PERMISSIONS_RDWR,16,sizeof(SppDataClient2ServerData),(u8*)
↪ (&TelinkSppDataClient2ServerUUID), (u8*)(SppDataClient2ServerData), &module_onReceiveData},
↪ //value

```

```
{0,ATT_PERMISSIONS_READ,2,sizeof(TelinkSPPC2SDescriptor),(u8*)&userdesc_UUID,(u8*)
(&TelinkSPPC2SDescriptor)},
```

The read and write permissions of the second Attribute are defined as: SPP_C2S_ATT_PERMISSIONS_RDWR.

This read and write permission is up to the user to choose, you can choose one of the following:

```
#define SPP_C2S_ATT_PERMISSIONS_RDWR ATT_PERMISSIONS_RDWR
#define SPP_C2S_ATT_PERMISSIONS_RDWR ATT_PERMISSIONS_ENCRYPT_RDWR
#define SPP_C2S_ATT_PERMISSIONS_RDWR ATT_PERMISSIONS_AUTHEN_RDWR
#define SPP_C2S_ATT_PERMISSIONS_RDWR ATT_PERMISSIONS_SECURE_CONN_RDWR
```

No matter which one you choose, the current pairing mode must be higher than or equal to this level of read and write permissions to read and write services correctly.

The user needs to modify feature_config.h as follows:

```
#define FEATURE_TEST_MODE TEST_GATT_SECURITY
```

SMP test encryption levels are LE_SECURITY_MODE_1_LEVEL_1, LE_SECURITY_MODE_1_LEVEL_2, LE_SECURITY_MODE_1_LEVEL_3, LE_SECURITY_MODE_1_LEVEL_4. The user needs to select app_config.h according to the needs of the corresponding pairing mode.

```
#define SMP_TEST_MODE LE_SECURITY_MODE_1_LEVEL_3
```

For example, the current pairing mode is LE_SECURITY_MODE_1_LEVEL_3, that is, there are both authentication and encryption Legacy pairing modes. So the current read and write permissions can be selected as follows.

```
#define SPP_C2S_ATT_PERMISSIONS_RDWR ATT_PERMISSIONS_AUTHEN_RDWR
```

The process is shown as following:

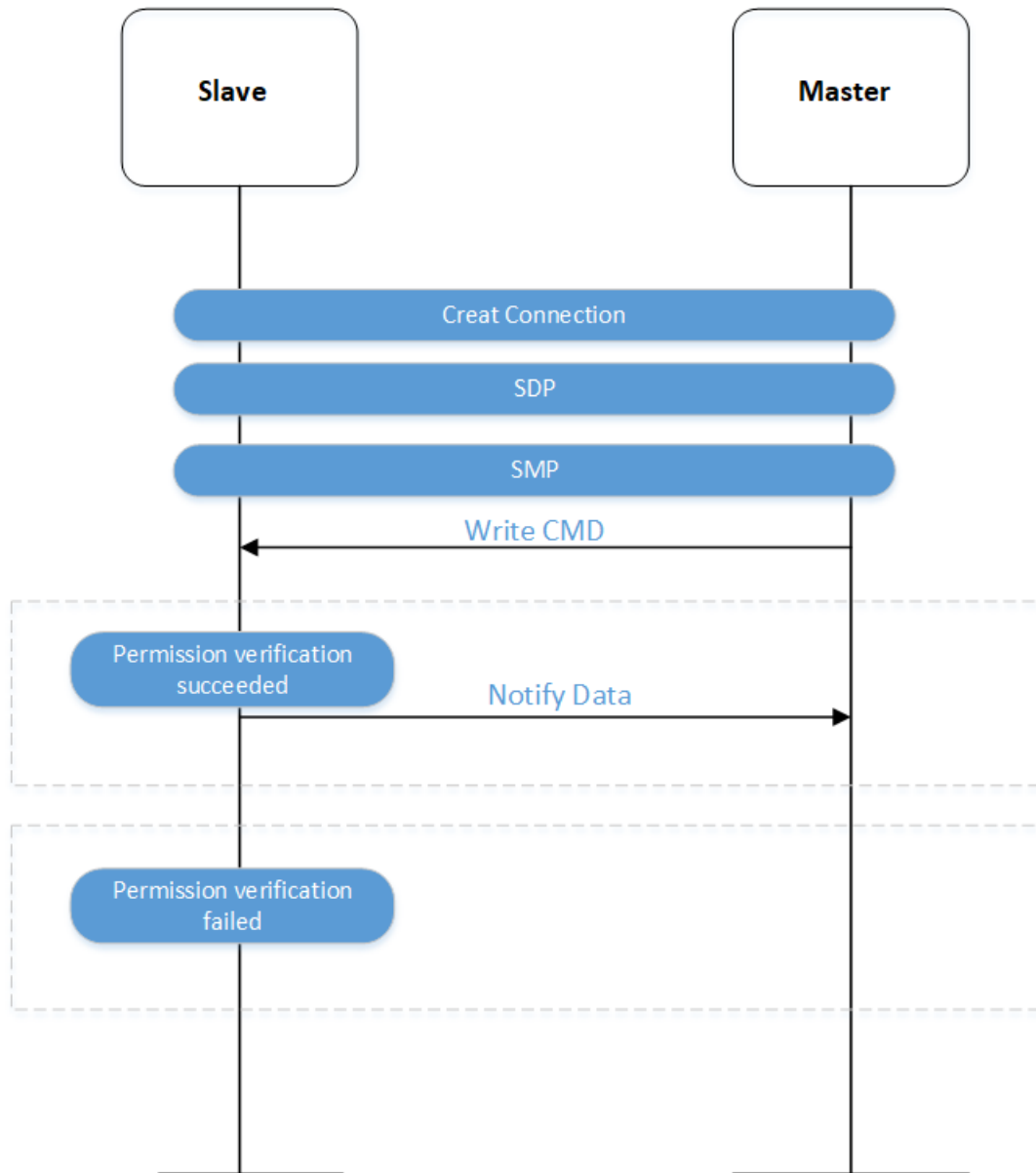


Figure 13.8: "Gatt Security"

13.4 DLE Test

The DLE test mainly tests the long package. Demo is divided into master and slave. Users need to compile and burn to two EVB boards respectively. For the code at master end, users can refer to `feature_master_dle`. For the corresponding `feature_config.h` selection at slave end, the code is as follows:

```
#define FEATURE_TEST_MODE    TEST_SDATA_LENGTH_EXTENSION
```

After programming, they are reset respectively, and the master is triggered to establish a connection. After the connection is successful, the MTU and DataLength are exchanged respectively.

```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, MTU_SIZE_SETTING);
blc_ll_exchangeDataLength(LL_LENGTH_REQ , DLE_TX_SUPPORTED_DATA_LEN);
```

After the exchange is successful, the slave will send a long packet of data to the master every 3.3s, or the master will trigger the pairing key GPIO_PC6 every time at a low level, the mater will write a long packet of data to the slave, and the slave will send the same data to the master after receiving it.

The test process is as follows:

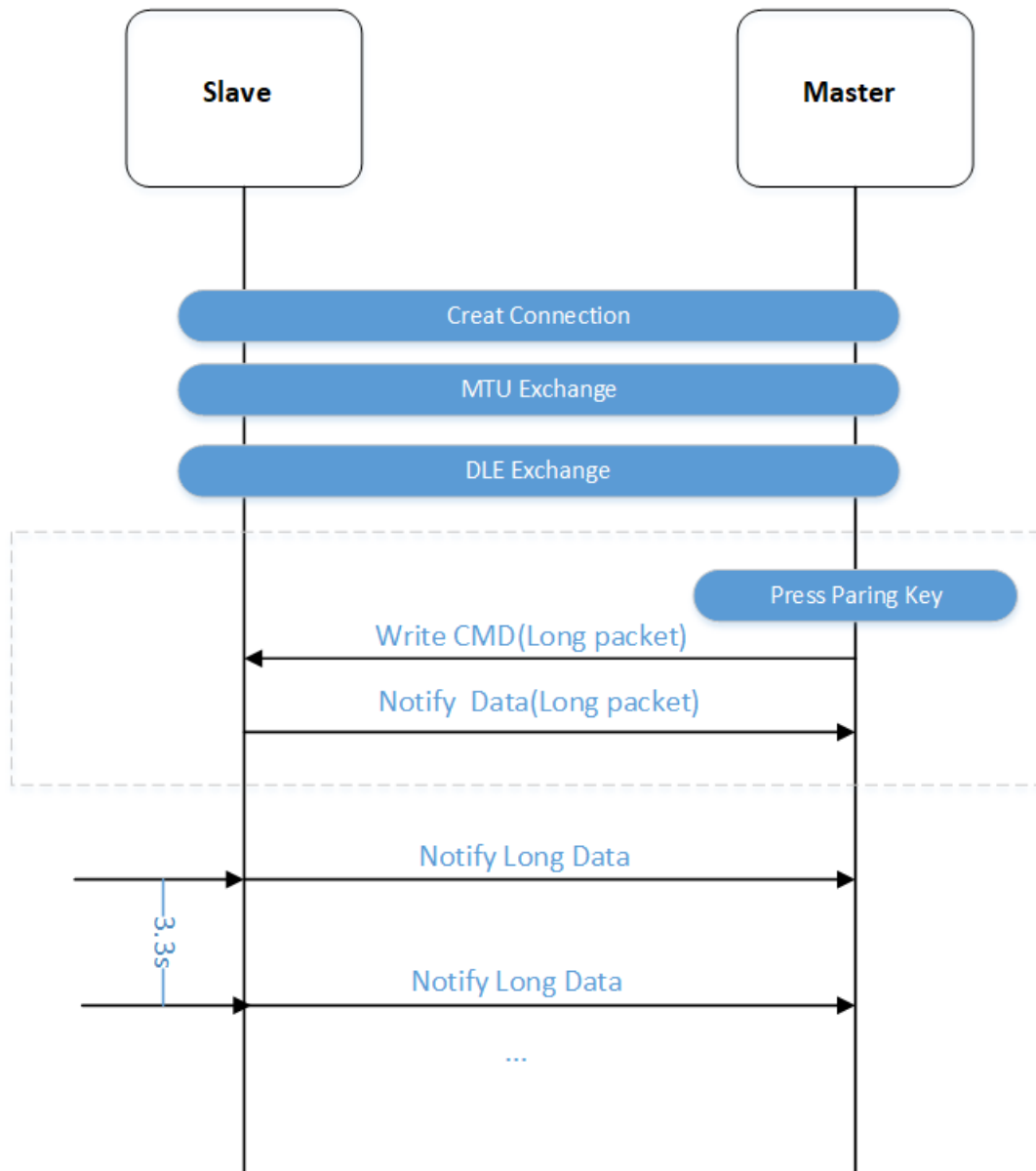


Figure 13.9: "DLE Test Process"

13.5 Soft Timer Test

Please refer to the chapter of Software Timer.

13.6 WhiteList Test

If the whitelist is set, only the devices in the whitelist are allowed to establish connections. The user needs to modify app_config.h as follows:

```
#define FEATURE_TEST_MODE TEST_WHITELIST
```

When the slave has no binding information, any other device is allowed to connect. After the connection is successful, the slave will add the current master's information to the whitelist, and then only the current device can connect with the slave.

The test process is as follows:

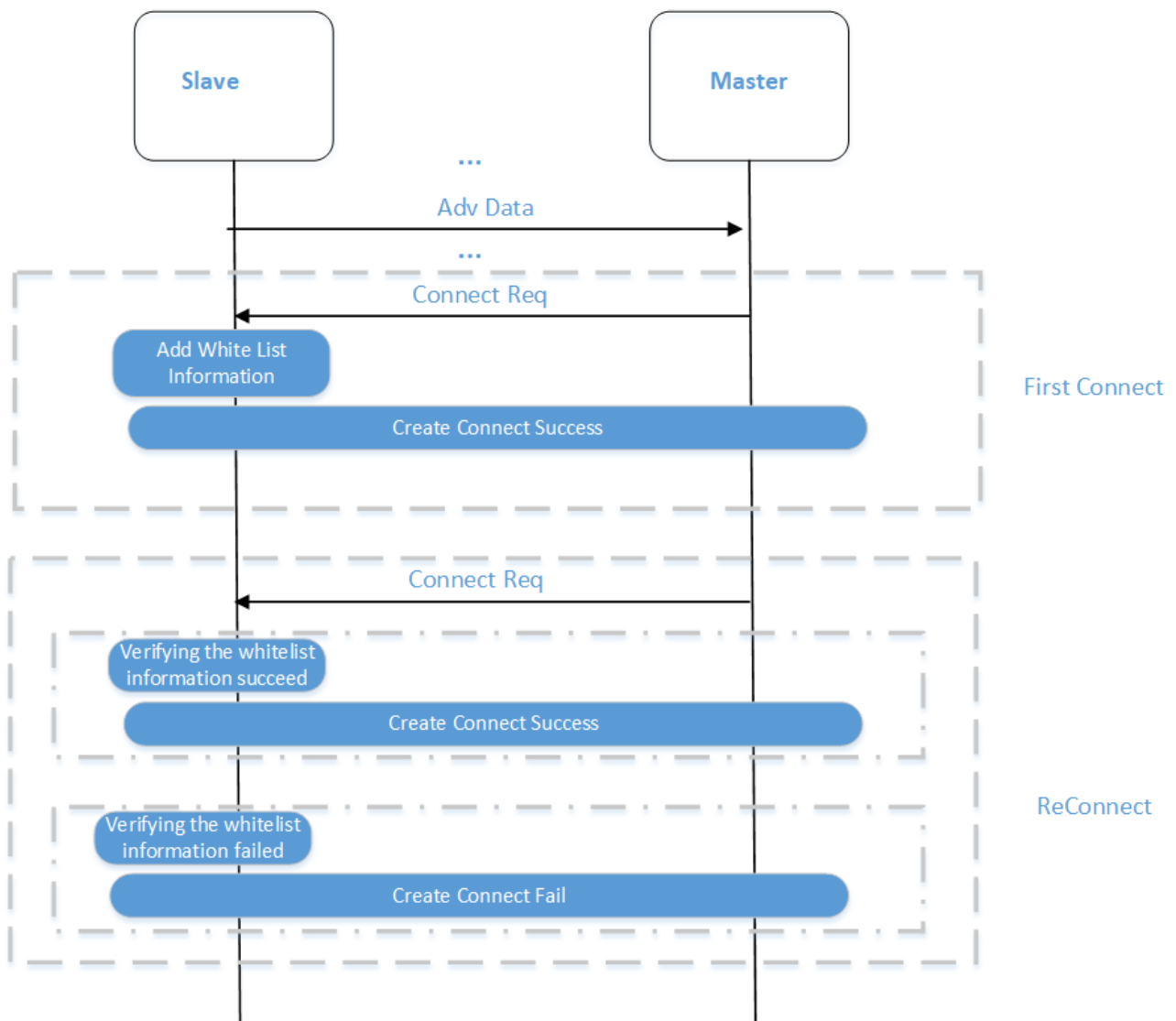


Figure 13.10: "Whitelist Test Process"

13.7 1M Extended Advertising Test

The 1M Extended advertising demo is mainly to test the extended broadcasting of 1M PHY. You need to modify `FEATURE_TEST_MODE` to `TEST_EXTENDED_ADVERTISING` in `app_config.h`.

```
#define FEATURE_TEST_MODE TEST_EXTENDED_ADVERTISING
```

The relevant codes are in `vendor/ b85m_feature_test /feature_extend_adv`, and the slave demo is provided.

Set the maximum length of broadcast data as follows:

```
#define APP_ADV_SETS_NUMBER      1      // Number of Supported Advertising Sets
#define APP_MAX_LENGTH_ADV_DATA  1024   // Maximum Advertising Data Length
#define APP_MAX_LENGTH_SCAN_RESPONSE_DATA 31 // Maximum Scan Response Data Length
```

In `feature_ext_adv_init_normal`, different types of extended broadcast packets based on 1M PHY configuration have been reserved.

```
#if 1 //Legacy, non_connectable_non_scannable
.....
#elif 0 //Legacy, connectable_scannable
.....
#elif 0 // Extended, None_Connectable_None_Scannable undirected, without auxiliary packet
.....
#elif 0 // Extended, None_Connectable_None_Scannable directed, without auxiliary packet
.....
#elif 0 // Extended, None_Connectable_None_Scannable undirected, with auxiliary packet
.....
#elif 0 // Extended, None_Connectable_None_Scannable Directed, with auxiliary packet
.....
#elif 0 // Extended, Scannable, Undirected
.....
#elif 0 // Extended, Connectable, Undirected
.....
#endif
```

Users need to use a mobile phone or protocol analysis device that supports the Bluetooth 5 Low Energy Advertising Extension function to see the extended broadcast data.

13.8 2M/Coded PHY Used on Extended Advertising Test

The 2M/Coded PHY used on Extended advertising demo is mainly to test the extended broadcasting of various combinations of 1M/2M/Coded PHY. You need to modify `FEATURE_TEST_MODE` to `TEST_2M_CODED_PHY_EXT_ADV` in `app_config.h`.

```
#define FEATURE_TEST_MODE                TEST_2M_CODED_PHY_EXT_ADV
```

The relevant codes are in vendor/ b85m_feature_test /feature_phy_extend_adv, and the slave demo is provided.

Set the maximum length of broadcast data as follows:

```
#define APP_ADV_SETS_NUMBER            1    // Number of Supported Advertising Sets
#define APP_MAX_LENGTH_ADV_DATA        1024  // Maximum Advertising Data Length
#define APP_MAX_LENGTH_SCAN_RESPONSE_DATA  31 // Maximum Scan Response Data Length
```

Feature_ext_adv_init_normal has reserved different types of extended broadcast packets based on various combinations of 1M PHY / Coded PHY(S2) / Coded PHY(S8).

```
#if 0 // Extended, None_Connectable_None_Scannable undirected, without auxiliary packet
    #if 0    // ADV_EXT_IND: 1M PHY
    #elif 1   // ADV_EXT_IND: Coded PHY(S2)
    #elif 0    // ADV_EXT_IND: Coded PHY(S8)
    #endif#
#elif 0 // Extended, None_Connectable_None_Scannable undirected, with auxiliary packet
    #if 1      // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: 1M PHY
    #elif 0     // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: 2M PHY
    #elif 0     // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S2)
    #elif 0     // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S8)
    #elif 0     // ADV_EXT_IND: Coded PHY(S2);    AUX_ADV_IND/AUX_CHAIN_IND: 1M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: 1M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S2);    AUX_ADV_IND/AUX_CHAIN_IND: 2M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: 2M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S2);    AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S2)
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S8)
    #endif
#elif 1 // Extended, Scannable, Undirected
    #if 1      // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: 1M PHY
    #elif 0     // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: 2M PHY
    #elif 0     // ADV_EXT_IND: 1M PHY;          AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S8)
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: 1M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: 2M PHY
    #elif 0     // ADV_EXT_IND: Coded PHY(S8);    AUX_ADV_IND/AUX_CHAIN_IND: Coded PHY(S8)
    #endif
#endif
```

Users can refer to the demo to combine the types of extended broadcast packages they need.

Users need to use mobile phones or protocol analysis devices that support Bluetooth 5 Low Energy Advertising Extension, Bluetooth 5 Low Energy 2Mbps and Bluetooth 5 Low Energy Coded (Long Range) functions to see the data broadcast by the above various types of extensions.

Note:

- API `blc_ll_init2MPhyCodedPhy_feature()` is used to enable 2M PHY/Coded PHY.

13.9 2M/Coded PHY used on Legacy advertising and Connection Test

The 2M/Coded PHY used on Legacy advertising and Connection demo is mainly to test that after establishing a connection based on Legacy advertising, switch to 1M/2M/Coded PHY in the connected state, and change `FEATURE_TEST_MODE` to `TEST_2M_CODED_PHY_CONNECTION` in `app_config.h`.

```
#define FEATURE_TEST_MODE          TEST_2M_CODED_PHY_CONNECTION
```

The relevant codes are in `vendor/b85m_feature_test/feature_phy_conn`, and the slave demo is provided. Initially open 2M Phy and Coded Phy:

```
blc_ll_init2MPhyCodedPhy_feature();    // mandatory for 2M/Coded PHY
```

After the connection is successful, the mainloop will use the API `blc_ll_setPhy()` to initiate a PHY change request in a 2-second cycle, 1M → Coded PHY(S2) → 2M → Coded PHY(S8) → 1M. The process is shown in the figure below:

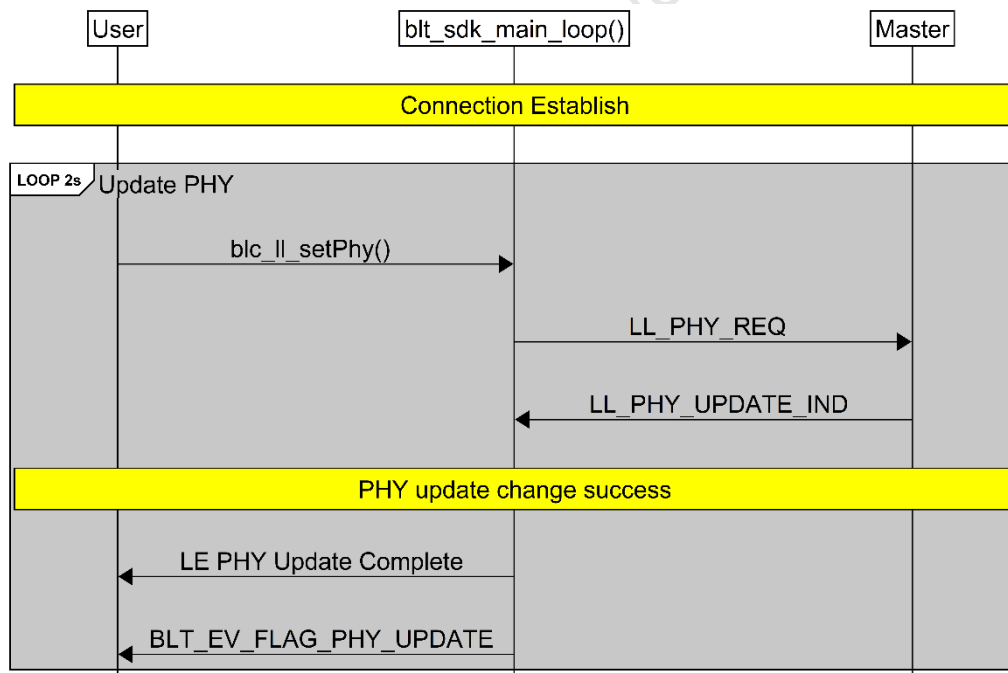


Figure 13.11: "PHY change flowchart"

```
if(phy_update_test_tick && clock_time_exceed(phy_update_test_tick, 2000000)){
    phy_update_test_tick = clock_time() | 1;
    int AAA = phy_update_test_seq%4;
    if(AAA == 0){
        blc_ll_setPhy(BLS_CONN_HANDLE, PHY_TRX_PREFER, PHY_PREFER_CODED, PHY_PREFER_CODED,
        ↪ CODED_PHY_PREFER_S2);
    }
}
```



```

    }
    else if(AAA == 1){
        blc_ll_setPhy(BLS_CONN_HANDLE, PHY_TRX_PREFER, PHY_PREFER_2M, PHY_PREFER_2M,
        ↪ CODED_PHY_PREFER_NONE);
    }
    else if(AAA == 2){
        blc_ll_setPhy(BLS_CONN_HANDLE, PHY_TRX_PREFER, PHY_PREFER_CODED, PHY_PREFER_CODED,
        ↪ CODED_PHY_PREFER_S8);
    }
    else{
        blc_ll_setPhy(BLS_CONN_HANDLE, PHY_TRX_PREFER, PHY_PREFER_1M, PHY_PREFER_1M,
        ↪ CODED_PHY_PREFER_NONE);
    }
    phy_update_test_seq ++;
}

```

Peer Master Device can use Demo “b85m_master_kma_dongle”, but also need to use API `blc_ll_init2MPhyCodedPhy_feature()` to open 2M Phy and Coded Phy.

Users can also choose to use other manufacturers’ Master devices or mobile phones that support Bluetooth 5 Low Energy 2Mbps and Bluetooth 5 Low Energy Coded (Long Range) functions.

13.10 CSA #2 Test

CSA #2 demo mainly uses Channel Selection Algorithm #2 (Channel Selection Algorithm #2) for frequency hopping when testing the connection state. You need to modify `FEATURE_TEST_MODE` to `TEST_CSA2` in `app_config.h`.

```
#define FEATURE_TEST_MODE TEST_CSA2
```

The relevant codes are all in `vendor/b85m_feature_test/feature_misc`, and the slave demo is provided.

Initial CSA #2:

```
blc_ll_initChannelSelectionAlgorithm_2_feature()
```

After enabling CSA #2, the `ChSel` field in the broadcast packet of Slave has been set to 1. If the `CONNECT_IND` PDU of the Peer Master Device has also set the `ChSel` field to 1, the channel selection algorithm #2 is used after the connection is successful. Otherwise, channel selection algorithm #1 should be used.

Peer Master Device can use SDK Demo “b85m_master_kma_dongle”, but also need to use API `blc_ll_initChannelSelectionAlgorithm_2_feature()` to open CSA #2.

Users can also choose to use Master devices or mobile phones from other manufacturers that support Bluetooth 5 Low Energy CSA #2.

13.11 EMI Test

The feature_emi is used to generate the required EMI test signals. This routine needs to be used with the EMI_Tool and "Non_Signaling_Test_Tool" tools.

13.11.1 Protocol

Please refer to "Telink SoC EMI Test User Guide" for the communication protocol.

13.11.2 Demo introduction

EMI test in B85 supports carrieronly mode, continue mode, burst mode, and packet receiving mode.

Supported wireless communication methods include Ble1M, Ble2M, Ble125K, Ble500K, Zigbee250K.

For the introduction of each mode and functional function, users can refer to "Telink Driver SDK Developer Handbook".

Telink Semiconductor

14 Other Modules

14.1 24MHz Crystal External Capacitor

Refer to the position C19/C20 of the 24MHz crystal matching capacitor in the figure below.

The SDK defaults to use B85 internal capacitance (that is, the cap corresponding to `ana_8a<5:0>`) as the matching capacitance of the 24MHz crystal oscillator. At this time, C19/C20 does not need to be soldered. The advantage of using this solution is that the capacitance can be measured and adjusted on the Telink fixture to make the frequency value of the final application product reach the best.

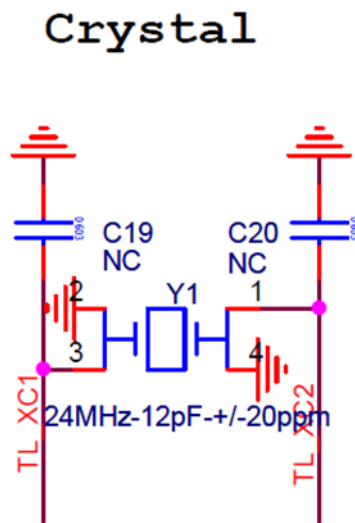


Figure 14.1: "24M Crystal Schematic"

If you need to use an external welding capacitor as the matching capacitor (C19/C20 welding capacitor) of the 24MHz crystal oscillator, just call the following API at the beginning of the main function (must be before the `blc_app_loadCustomizedParameters()` function and after the `cpu_wakeup_init` function):

```
static inline void blc_app_setExternalCrystalCapEnable(u8 en)
{
    blt_miscParam.ext_cap_en = en;
    WriteAnalogReg (0x8a, ReadAnalogReg(0x8a)|0x80);//close internal cap
}
```

As long as the API is called before `cpu_wakeup_init`, the SDK will automatically handle all the settings, including disabling the internal matching capacitor, no longer reading the frequency bias correction value, etc.

14.2 32KHz Clock Source Selection

The SDK supports the use of either the MCU's internal 32k RC oscillator circuit (referred to as 32k RC) or an external 32k RC oscillator circuit (referred to as 32k Pad). The error of 32k RC is relatively large, so for applications with long suspend or deep retention time, the time accuracy will be worse. At present, the maximum long connection supported by 32k RC by default cannot exceed 3s. Once exceeding this time, ble_timing will have errors, resulting in inaccurate packet receiving time points, prone to receiving and sending packets retry, increased power consumption, and even disconnection. The error is much smaller when using the 32k Pad.

The user only needs to call the following API at the beginning of the main function (must be before the cpu_wakeup_init function):

Call 32k RC:

```
void blc_pm_select_internal_32k_crystal(void);
```

Call 32k Pad:

```
void blc_pm_select_external_32k_crystal (void);
```

14.3 Firmware Digital Signature

There is a method of malicious copying of products in the market. For example, if customer A develops a product using Telink's chip and SDK, customer A's competitor customer B, who also uses Telink's chip, gets the product and can copy the same hardware circuit design. If the product's data burn-in bus is not disabled, it is possible for customer B to read the complete firmware on the product's Flash, and customer B can copy the product using the same hardware and software.

To address these security risks, the SDK supports a software digital signature function. The principle is to take advantage of the fact that the chip's internal Flash has a unique UID. The product reads the 16 byte UID from the internal Flash during the fixture burn-in process and then performs a complex encryption operation with the contents of the Firmware to produce a set of checksum values called Signature, which are stored at the corresponding address in the Flash calibration area. Which is:

Signature = Encryption_function (Firmware, Flash_UID)

Signature is related to both Firmware and Flash_UID. The same calculation is done when the program is initialized on the SDK, the result is compared with the Signature burned on the fixture and if it does not match, the program is not legal and is prohibited from running.

It is important to emphasize that this feature involves a number of technical aspects, including the fitment of the jig, the corresponding configuration on the SDK, etc. Customers must confirm the details with Telink FAE in advance if required.

Below are some technical details of the implementation of this feature.

- (1) The jig end must be correctly matched, including file configuration, writing scripts, etc. Please refer to the Telink testbench documentation and instructions for details.

- (2) The Signature memory address is the Flash Calibration area offset address 0x180 continuous 16 byte.
- (3) This feature is disabled by default on the SDK, to use it, enable the following macro in app_config.h.

```
#define FIRMWARES_SIGNATURE_ENABLE 1 //firmware check
```

Note:

- There are only a few projects on the SDK that add Firmware digital signature verification to the initialization of the main function (see FIRMWARES_SIGNATURE_ENABLE by searching for it). If the customer is using a project that does not have this feature, please make sure to merge from another project to your own.

The code in the SDK is shown below and the program needs to be disabled when the digital signature does not match. The SDK uses the simplest while(1) to disable the program when the digital signature does not match, this is just a sample writeup, please evaluate for yourself if this method meets the requirements, if not you can use other methods such as putting the MCU into deepsleep, modifying various data, bss, stack segments etc. stack segments, etc.

```
void blt_firmware_signature_check(void)
{
    unsigned int flash_mid;
    unsigned char flash_uid[16];
    unsigned char signature_enc_key[16];
    int flag = flash_read_mid_uid_with_check(&flash_mid, flash_uid);

    if(flag==0){ //reading flash UID error
        while(1);
    }

    firmware_encrypt_based_on_uid (flash_uid, signature_enc_key);

    //signature not match
    if(memcmp(signature_enc_key, (u8*)(flash_sector_calibration +
    ↪ CALIB_OFFSET_FIRMWARE_SIGNKEY), 16)){
        while(1); //user can change the code here to stop firmware running
    }
}
```

- (4) The calculation method for the digital signature, Encryption_function, is defined by Telink and takes into account both the Firmware content and the Flash_UID, using AES 128 encryption. The details of the calculation are not publicly available and are packaged in a sealed library. The above firmware_encrypt_based_on_uid function is implemented in libfirmware_encrypt.a.

If customers feel that the generic encryption algorithm is not secure enough and need to use their own encryption algorithm, they can contact Telink FAE to discuss the solution.

14.4 Firmware Integrity Self-check

The Firmware Integrity Self-check function is used to check the integrity of the current firmware during initialization to prevent errors in the firmware from causing problems.

This function is disabled by default on the SDK, to use it, enable the following macro in app_config.h.

```
#define FIRMWARES_CHECK_ENABLE 1 //firmware check
```

Note:

- There are only a few projects on the SDK that add the Firmware self-check function to the initialization of the main function (you can see this by searching for FIRMWARES_CHECK_ENABLE). If the customer is using a project that does not have this feature, please make sure to merge from another project to your own.

Referring to the introduction of Firmware CRC32 checksum in OTA chapter, the SDK adds the checksum value at the end when generating the Firmware, the last step of OTA upgrade can confirm whether the Firmware is complete by the checksum value. Here the Firmware integrity self-check is done with the help of CRC32 checksum value, during initialization the Firmware is read to calculate the CRC32 and compared, if it does not meet, the Firmware is considered corrupted.

The code is as follows, the simplest while(1) is used on the SDK to disable the run when the checksum fails, this is just an example write up, users need to modify this write up to suit their own UI design, such as LED indication errors etc.

```
#if FIRMWARE_CHECK_ENABLE
    if(flash_fw_check(0xffffffff)){ //return 0, flash fw crc check ok. return 1, flash
        ↪ fw crc check fail
        while(1);                //Users can process according to the actual application.
    }
#endif
```

It is clear from the above description of the principle and details that the Firmware self-check cannot detect 100% of program abnormalities, for example, if the Firmware is extensively damaged, the program may not be initialized properly and will not run here with the self-check function. Therefore, the self-check function can only be effective when the Firmware is less damaged.

15 Debug

15.1 Introduction to GPIO simulation UART_TX printing method

To facilitate the user to print information when debugging, B85 supports gpio simulation printing `printf(const char *fmt, ...)`, `array_printf(unsigned char*data, unsigned int len)`, the relevant information needs to be defined in `app_config.h` as follows.

```
#ifndef UART_PRINT_DEBUG_ENABLE
#define UART_PRINT_DEBUG_ENABLE      1
#endif

//////////////////// PRINT DEBUG INFO //////////////////////////////////////
#if (UART_PRINT_DEBUG_ENABLE)
    #define PRINT_BAUD_RATE           115200
    #define DEBUG_INFO_TX_PIN        GPIO_PA0
    #define PULL_WAKEUP_SRC_PA0      PM_PIN_PULLUP_10K
    #define PA0_OUTPUT_ENABLE        1
    #define PA0_DATA_OUT             1 //must
#endif
```

The default baud rate here is 115200 and the TX_PIN is GPIO_PA0, the user can change the baud rate and TX_PIN according to the actual needs.

If the user wants to use a higher baud rate (greater than 115200, maximum support 1M), the user needs to increase the cclk, at least to 24MHz or more, change the cclk in `app_config.h`.

```
//////////////////// Clock //////////////////////////////////////
/**
 * @brief MCU system clock
 */
#define CLOCK_SYS_CLOCK_HZ          24000000
```

Note:

- The information printed by `printf` may be interrupted by a garbled message, do not print in an interrupt!

16 Q&A

Q. When compiling a project of the SDK, the following error is generated, how to solve it?

```
./boot/8258/cstartup_8258_RET_16K.o ./boot/8258/cstartup_8258_RET_32K.o ./boot/8253/cstartup_8253_RET_16K.o ./boot/8253/cstartup_8253_RET_32K.o
./boot/8251/cstartup_8251_RET_16K.o ./boot/8251/cstartup_8251_RET_32K.o ./application/usbstd/usb.o ./application/usbstd/usbdesc.o
./application/usbstd/usbhw.o ./application/print/putchar.o ./application/print/u_printf.o ./application/keyboard/keyboard.o
./application/audio/adpcm.o ./application/audio/gl_audio.o ./application/audio/msbc_decode.o ./application/audio/msbc_encode.o
./application/audio/sbc_decode.o ./application/audio/sbc_encode.o ./application/audio/tl_audio.o ./application/app/usbaud.o
./application/app/usbcd.o ./application/app/usbb.o ./application/app/usbmouse.o ./algorithm/ecc/ecc_ll.o ./algorithm/ecc/hw_ecc.o
./algorithm/ecc/sw_ecc.o ./algorithm/aes_ccm/aes_ccm.o
C:\TelinkSDK\opt\tc32\bin\tc32-elf-ld.exe: Warning: size of symbol `blt_rxfifo_b' changed from 1 in ./vendor/common/blt_common.o to 512 in
./vendor/b85m_module/app.o
C:\TelinkSDK\opt\tc32\bin\tc32-elf-ld.exe: Warning: size of symbol `blt_txfifo_b' changed from 1 in ./vendor/common/blt_common.o to 640 in
./vendor/b85m_module/app.o
C:\TelinkSDK\opt\tc32\bin\tc32-elf-ld.exe: section .text loaded at [00004000,0000c90b] overlaps section .retention_data loaded at [0000304c,000040db]
make: *** [ble_b85m_single_connection_src.elf] Error 1
```

Figure 16.1: "Error in compiling a SDK project"

A. This situation is due to the default 16k retention used in the SDK, and some projects need to switch to a 32k retention configuration due to the complexity of the project or the need for a larger buffer and the ram in the retention_code section exceeds 16k. The modification process is as follows.

Step 1 Right-click on the project, select Properties -> C/C++ Build -> Settings -> General, in Other GCC Flags change *****16K to *****32K.

Step 2 Paste the contents of the boot_32k_retn....link file corresponding to the required boot in the boot/ directory into the boot.link in the subfolder, then clean the project.

For the known Module projects, and slave projects with large DLEs, phy_extend_adv, phy_conn, Big_pdu, etc. in feature projects need to be changed to the 32k retention case.

Q. How to create my own project in the SDK?

A. Generally, to ensure that the various settings in the project are in place, we usually build new projects based on a particular demo. For example, we use b85m_ble_sample as a base to complete a new project.

Step 1 Copy and paste the code and rename it. As shown below, we copy the code for b85m_ble_sample and newly name it Test_Demo.

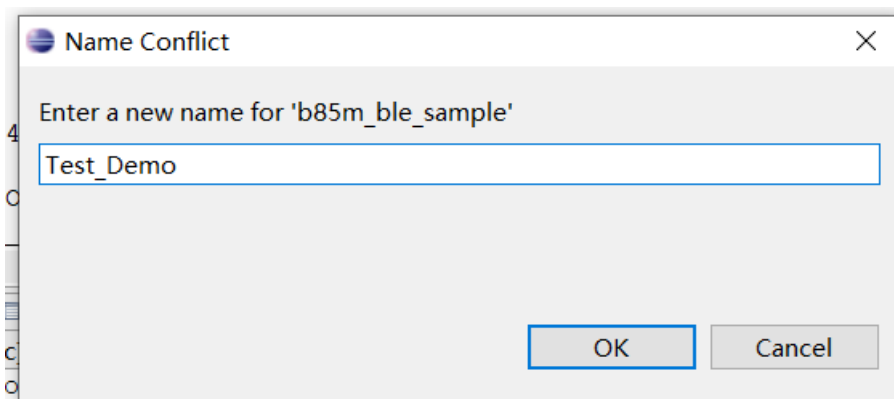


Figure 16.2: "Enter a new name for a project"

Step 2 Right-click on the project, Properties -> Settings -> Manage Configurations, create a new project, for example: Test_Demo.

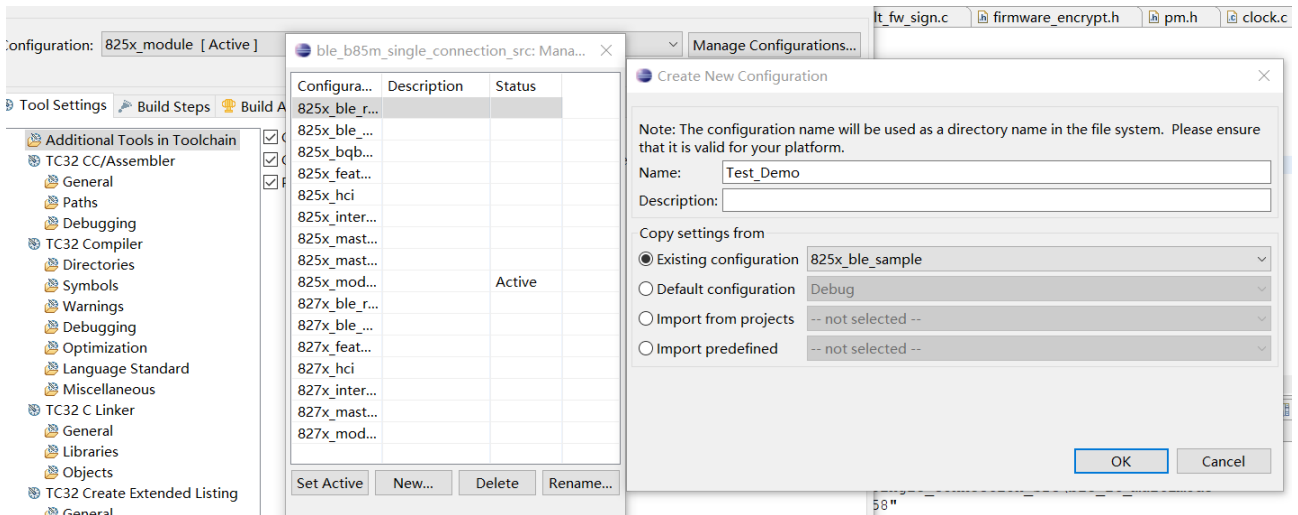


Figure 16.3: “Create new configuration for a project”

After clicking OK, you can see the new project in the project list, as shown below.

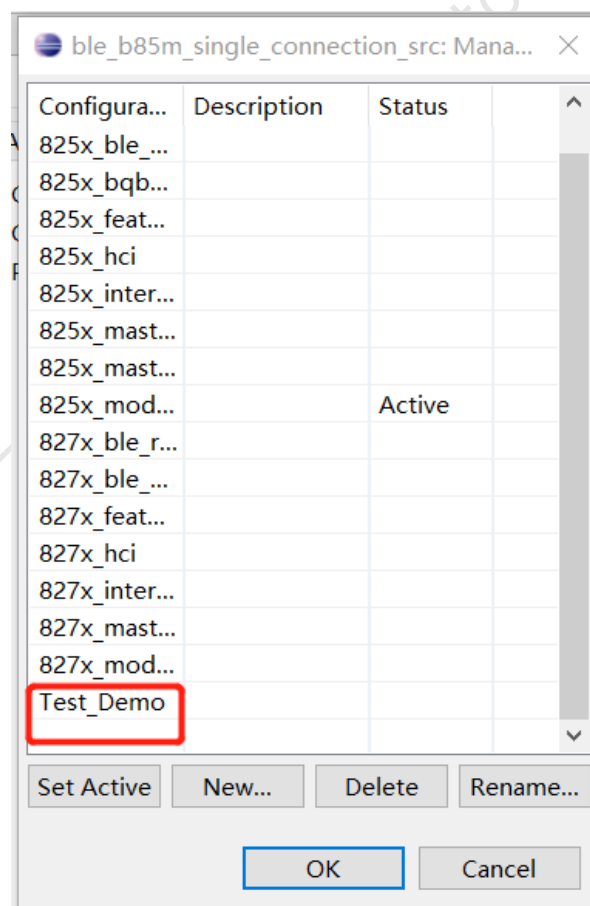


Figure 16.4: “New project in the project list”

Step 3 Right-click on the Test_Demo folder, Resource Configurations -> Exclude from Build, tick all the items in Test_Demo’s settings except for itself. And tick Test_Demo in the same settings for its copy source. This

is shown as below.

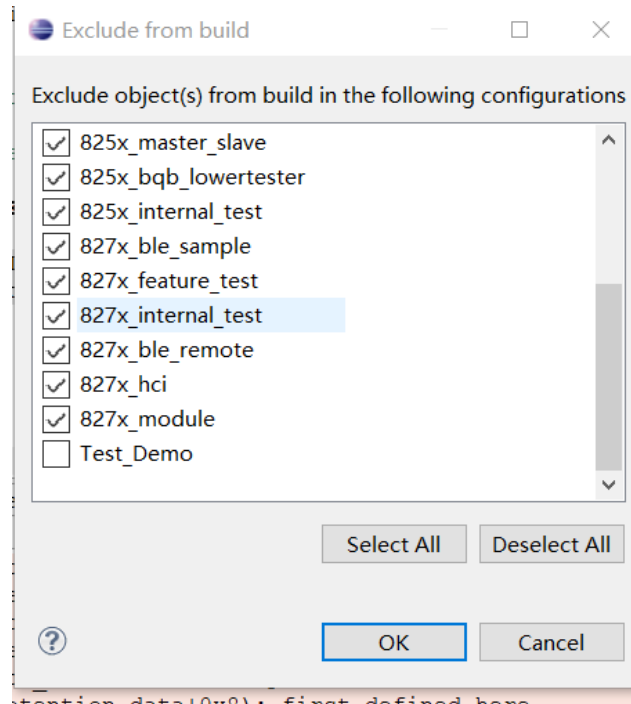


Figure 16.5: "Exclude Test_Demo from build"

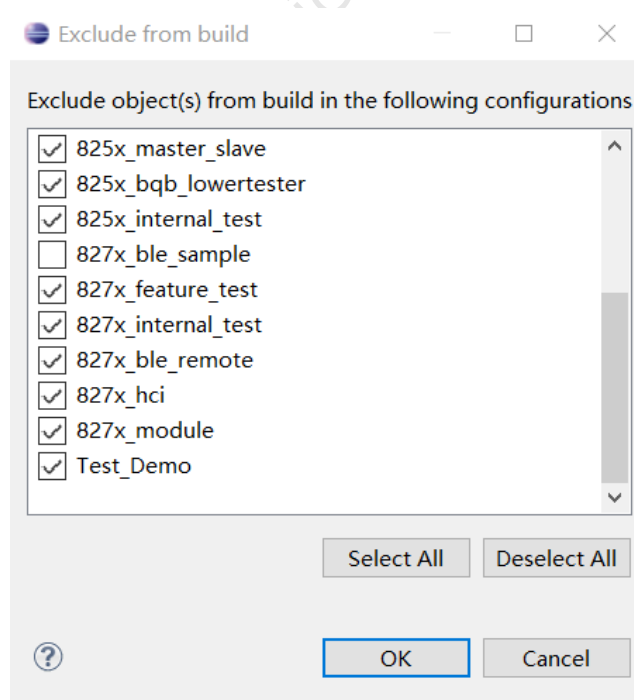


Figure 16.6: "Exclude source project from build"

Step 4 Change the Setting-TC32 Compiler-symbols in the Test_Demo property to the new symbols and click Apply after the change, as shown below:

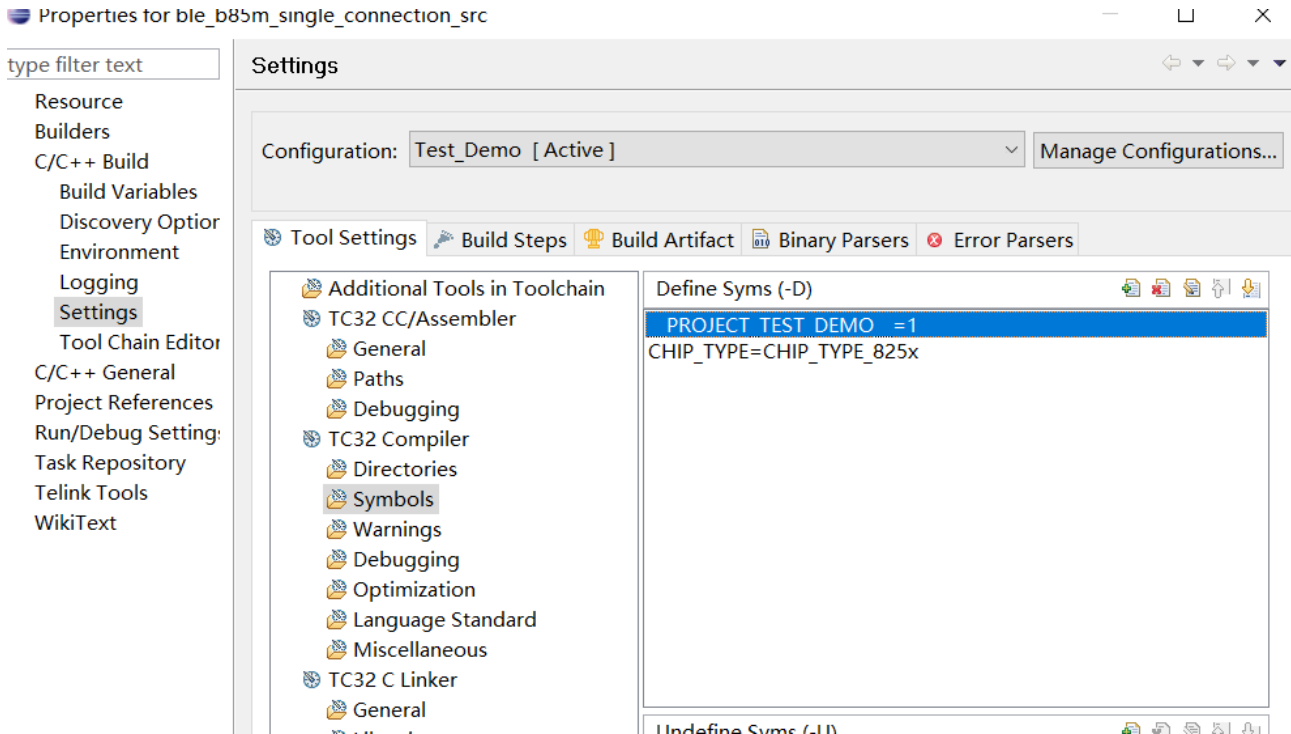


Figure 16.7: "Modify compiler symbol"

Step 5 In vender/common/user_config.h, add corresponding to the settings for the new code. As shown below:



Figure 16.8: "Add user config for new code"

At this point, the new project has been built. You can now select the new project, clean it and build it for use.

17 Appendix

17.1 crc16 Algorithm

```
unsigned short crc16 (unsigned char *pD, int len)
{
    static unsigned short poly[2]={0, 0xa001};
    unsigned short crc = 0xffff;
    unsigned char ds;
    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```