RSL10 Firmware Reference

M-20818-015
August 2019

© SCILLC, 2019
Previous Edition © 2018
"All Rights Reserved"

ON Semiconductor®

Table of Contents

Page

1. Introduction . . . 13
    1.1 Purpose . . . 13
    1.2 Intended Audience . . . 13
    1.3 Conventions . . . 13
    1.4 Further Reading . . . 13
2. Firmware Overview . . . 15
    2.1 Introduction . . . 15
    2.2 Firmware Components . . . 15
        2.2.1 Firmware Files . . . 17
        2.2.2 Compliance Exceptions . . . 20
    2.3 Firmware Naming Conventions . . . 20
    2.4 Firmware Resource Usage . . . 21
    2.5 Versions . . . 21
        2.5.1 Hardware Variants and Firmware Compatibility . . . 21
        2.5.2 Firmware Versions . . . 21
3. Hardware Definitions . . . 23
    3.1 Register and Register Bit-field Definition . . . 23
    3.2 Memory Map Definition . . . 24
    3.3 Non-Volatile Record Memory Map . . . 24
        3.3.1 Application Specific Record . . . 25
        3.3.2 Bond Information Record . . . 25
        3.3.3 Device Configuration Record . . . 26
        3.3.4 Manufacturing Records . . . 27
    3.4 Interrupt Vector Definition . . . 30
4. Event Kernel . . . 32
    4.1 Overview . . . 32
        4.1.1 Feature List . . . 32
        4.1.2 Top-Level Objects . . . 32
        4.1.3 Include Files . . . 32
        4.1.4 API Functions . . . 32
            4.1.4.1 Kernel_Init . . . 33
            4.1.4.2 Kernel_Schedule . . . 33
        4.1.5 Kernel Environment . . . 33
    4.2 Messages . . . 34
        4.2.1 Overview . . . 34
        4.2.2 Message Format . . . 34
        4.2.3 Message Identifier . . . 34
        4.2.4 Parameter Management . . . 35
        4.2.5 Message Queue Object . . . 35
        4.2.6 Message Queue Primitives . . . 35
            4.2.6.1 Message Allocation . . . 35
            4.2.6.2 Message Send . . . 36

4.2.6.3 Message Send Basic . . . 36
4.2.6.4 Message Forward . . . 37
4.2.6.5 Message Free . . . 37
4.3 Scheduler . . . 37
4.3.1 Overview . . . 37
4.3.2 Requirements . . . 38
4.3.2.1 Scheduling Algorithm . . . 38
4.3.2.2 Save Service . . . 38
4.4 Tasks . . . 38
4.4.1 Definition . . . 38
4.5 Kernel Timer . . . 39
4.5.1 Overview . . . 39
4.5.2 Time Definition . . . 39
4.5.3 Timer Object . . . 39
4.5.4 Timer Setting . . . 40
4.5.5 Time Primitives . . . 40
4.5.5.1 Timer Set . . . 40
4.5.5.2 Timer Clear . . . 41
4.5.5.3 Timer Activity . . . 41
4.5.5.4 Timer Expiry . . . 42
4.6 Useful Macros . . . 42
5. Program ROM . . . 43
5.1 Overview . . . 43
5.2 Vector Table . . . 43
5.3 Initialization Support . . . 43
5.3.1 Base System Initialization . . . 44
5.3.2 User-Defined System Initialization . . . 44
5.3.3 Boot and Wakeup Initialization . . . 45
5.4 Application Validation and Boot . . . 47
5.5 Function Table . . . 48
6. Bluetooth Stack and Profiles . . . 50
6.1 Introduction . . . 50
6.1.1 Include and Object Files . . . 50
6.1.2 Bluetooth Stack . . . 54
6.1.3 Stack Support Functions . . . 54
6.1.3.1 BLE_Init . . . 55
6.1.3.2 BLE_InitNoTL . . . 55
6.1.3.3 BLE_Reset . . . 56
6.1.3.4 BLE_Power_Mode_Enter . . . 56
6.2 HCI . . . 57
6.2.1 HCI Software Architecture . . . 58
6.2.1.1 HCI Control Messages Descriptors . . . 60
6.2.1.2 Event Descriptors . . . 63
6.2.1.3 Internal Messages Definition . . . 64
6.2.1.4 Events . . . 65
6.2.1.4.1 Legacy Events . . . 65

6.2.1.4.2 LE Event . . . 66
6.2.1.4.3 Command Complete Event . . . 66
6.2.1.4.4 Command Status Event . . . 66
6.2.1.4.5 LE ACL RX Data . . . 67
6.2.1.4.6 LE ACL TX Data . . . 67
6.2.1.5 Internal Messages Routing . . . 68
6.2.1.5.1 For External Host to Internal Controller . . . 69
6.2.2 Between Internal Host and Controller . . . 71
6.2.3 Proprietary Rules for Connection Handle Allocation . . . 71
6.2.4 Communication with External Host . . . 71
6.2.5 HCI Events . . . 73
6.2.5.1 Legacy Events . . . 73
6.2.5.2 Command Complete Events . . . 74
6.2.5.3 Command Status Events . . . 74
6.2.5.4 LE Events . . . 75
6.2.5.5 HCI ACL TX Data . . . 75
6.2.5.6 HCI ACL RX Data . . . 78
6.2.6 Generic Parameter Packing - Unpacking . . . 78
6.2.6.1 Parameters Format Definition . . . 79
6.2.6.2 Generic Packer . . . 79
6.2.6.3 Generic Unpacker . . . 80
6.2.6.4 Alignment and Data Copy Primitives . . . 81
6.3 GATT . . . 81
6.3.1 GATT Fundamentals . . . 81
6.3.1.1 Roles . . . 81
6.3.1.2 Security Features . . . 82
6.3.1.3 Attribute Grouping . . . 82
6.3.1.3.1 Service . . . 82
6.3.1.3.2 Included Service . . . 83
6.3.1.3.3 Characteristics . . . 83
6.3.1.4 L2CAP . . . 86
6.3.2 Attribute Protocol Toolbox . . . 87
6.3.2.1 Basic Attribute Concepts . . . 87
6.3.2.1.1 Attribute . . . 87
6.3.2.1.2 Protocol Methods . . . 88
6.3.2.2 Attribute Protocol Packet Data Unit Format . . . 89
6.3.2.3 Attribute Protocol Operations . . . 89
6.3.2.3.1 Atomic Operations . . . 89
6.3.2.3.2 Flow Control . . . 89
6.3.2.3.3 Transaction . . . 89
6.3.2.4 Attribute Protocol Module Interfaces . . . 90
6.3.2.4.1 Interface with Upper Layers . . . 90
6.3.2.4.2 Interface with L2CAP . . . 90
6.3.2.5 Attribute Manager (Database Owner) . . . 90

6.3.2.5.1 Attribute Definition . . . 91
6.3.2.5.2 Service Definition . . . 91
6.3.2.5.3 Service Permission Field . . . 92
6.3.2.5.4 Attribute Permission Field . . . 92
6.3.2.5.5 Data Caching . . . 94
6.3.2.5.6 Attribute Database Example . . . 94
6.3.2.6 Attribute Server . . . 94
6.3.2.6.1 Attribute Discovery / Read . . . 94
6.3.2.6.2 Attribute Write . . . 95
6.3.2.6.3 Server Initiated Events . . . 98
6.3.2.6.4 Data Caching . . . 99
6.3.2.7 Attribute Client . . . 101
6.3.2.7.1 Discovery Command . . . 101
6.3.2.7.2 Read Command . . . 105
6.3.2.7.3 Write Command . . . 107
6.3.2.7.4 Reception of Notification or Indications . . . 110
6.3.3 Features and Functions . . . 111
6.3.3.1 Attribute Packet Size Negotiation . . . 111
6.3.3.2 Primary Service Discovery . . . 111
6.3.3.3 Relationship Discovery . . . 112
6.3.3.4 Characteristic Discovery . . . 112
6.3.3.5 Characteristic Descriptor Discovery . . . 113
6.3.3.6 Characteristic Value Read . . . 113
6.3.3.7 Characteristic Value Write . . . 114
6.3.3.8 Characteristic Value Notification . . . 115
6.3.3.9 Characteristic Value Indication . . . 115
6.3.3.10 Characteristic Descriptor Value Read . . . 116
6.3.3.11 Characteristic Descriptor Value Write . . . 116
6.3.4 Service Discovery Procedure . . . 116
6.3.5 GATT Profile Service . . . 118
6.3.6 GATT Environment Variables . . . 118
6.3.6.1 GATT Manager Environment . . . 118
6.3.6.2 GATT Controller Environment . . . 119
6.4 GAP Functionality . . . 119
6.4.1 Modes and Profile Roles . . . 119
6.4.2 General LE Procedures . . . 121
6.4.2.1 Broadcasting and Observing . . . 121
6.4.2.1.1 Conditions . . . 121
6.4.2.2 Advertising Modes . . . 122
6.4.2.2.1 Broadcast Mode . . . 122
6.4.2.2.2 Non-Discoverable Mode . . . 123
6.4.2.2.3 General Discoverable . . . 123
6.4.2.2.4 Limited Discoverable . . . 123
6.4.2.2.5 Direct Mode . . . 123
6.4.2.3 Scan Modes . . . 123

6.4.2.3.1 Device Discovery . . . 123
6.4.2.3.2 Observer Mode . . . 124
6.4.2.3.3 General Discovery . . . 124
6.4.2.3.4 Limited Discovery . . . 125
6.4.2.3.5 Name Discovery . . . 125
6.4.2.4 Connection . . . 125
6.4.2.4.1 Direct Connection Establishment . . . 127
6.4.2.4.2 General Connection Establishment . . . 128
6.4.2.4.3 Automatic Connection Establishment . . . 128
6.4.2.4.4 Selective Connection Establishment . . . 129
6.4.2.4.5 Update Connection Parameters . . . 130
6.4.2.5 Bonding . . . 133
6.4.3 Low Energy Security . . . 134
6.4.3.1 Security Modes . . . 134
6.4.3.2 Authentication Procedure . . . 135
6.4.3.3 Authorization Procedure . . . 135
6.4.3.4 Data Signing . . . 135
6.4.3.5 Privacy . . . 136
6.4.3.5.1 Host Managed Privacy (1.1) . . . 136
6.4.3.5.2 Controller Managed Privacy (1.2) . . . 136
6.4.3.5.3 LE Address . . . 136
6.4.4 Security Manager Toolbox . . . 138
6.4.4.1 Keys Definition . . . 139
6.4.4.2 AES-CMAC Algorithm . . . 140
6.4.4.3 Identity Root Generation . . . 140
6.4.4.3.1 Identity Resolving Key Generation . . . 141
6.4.4.3.2 Diversifier Hiding Key Generation . . . 141
6.4.4.3.3 Connection Signature Resolving Key Generation . . . 141
6.4.4.3.4 Long Term Key and Diversifier Generation . . . 141
6.4.4.3.5 Encrypted Session Setup . . . 141
6.4.4.3.6 Link Layer Encryption . . . 141
6.4.4.3.7 Signing Algorithm . . . 142
6.4.4.3.8 Slave Initiated Security . . . 142
6.4.4.4 Procedure Details . . . 142
6.4.4.4.1 Random Address Generation . . . 142
6.4.4.4.2 Address Resolution . . . 144
6.4.4.4.3 Encryption Toolbox Access . . . 144
6.4.4.4.4 Pairing . . . 145
6.4.4.4.5 Encryption . . . 154
6.4.4.4.6 Data Signing . . . 156
6.4.4.4.7 Pairing Repeated Attempts . . . 159
6.4.4.5 Security Manager Protocol Data Unit Format . . . 159
6.4.4.5.1 SMP PDU Codes . . . 160

6.4.5 LE Credit Based Channel . . . 160
6.4.5.1 Channel Registration . . . 162
6.4.5.2 Connection Creation . . . 163
6.4.5.3 Disconnection . . . 166
6.4.5.4 Data Exchange . . . 166
6.4.5.5 Credit Management . . . 169
6.4.5.6 LE Ping . . . 169
6.4.5.7 LE Data Packet Length Extension . . . 170
6.4.5.8 Profile Management . . . 170
6.4.5.9 GAP service database . . . 172
6.4.5.10 GAP Environment Variables . . . 173
6.4.5.10.1 GAP Manager Environment . . . 173
6.4.5.10.2 GAP Controller Environment . . . 173
6.4.5.10.3 GAP Profiles Environment . . . 174
6.4.5.11 Device initialization . . . 174
6.4.5.11.1 Software Reset . . . 174
6.4.5.11.2 Device Configuration . . . 174
6.4.6 Profile Functionalities . . . 175
6.4.7 Message API naming requirements . . . 176
6.4.8 Memory Optimization . . . 177
6.4.8.1 Connection Oriented Task . . . 177
6.4.8.2 Operation Model . . . 177
7. Custom Protocols . . . 179
7.1 Overview . . . 179
7.2 Audio Stream Broadcast Custom Protocol . . . 179
7.2.1 Audio Stream Broadcast Packet Structure . . . 180
7.2.2 Audio Stream Broadcast Transmission Structure . . . 181
7.2.2.1 Packet Sets . . . 181
7.2.2.2 RF Physical Layer Configuration . . . 181
7.2.2.3 RF Transmission Structure . . . 182
7.2.3 Audio Stream Broadcast API . . . 184
7.2.3.1 RM_Configure . . . 184
7.2.3.2 RM_Disable . . . 185
7.2.3.3 RM_Enable . . . 185
7.2.3.4 RM_EventHandler . . . 185
7.2.3.5 RM_StatusHandler . . . 186
7.3 Low-Latency Custom Protocol . . . 186
7.3.1 Low-Latency Protocol Physical Layer . . . 187
7.3.2 Low-Latency Protocol Packet Structure . . . 187
7.3.3 Low-Latency Protocol Link Layer Structure . . . 188
7.3.4 Low-Latency Protocol Application Program Interface . . . 189
7.3.5 Low-Latency Protocol Modules/Peripheral Usage . . . 190
7.3.6 Low-Latency Custom Protocol API . . . 190
7.3.6.1 CP_Configure . . . 190
7.3.6.2 CP_Disable . . . 191
7.3.6.3 CP_Enable . . . 191

7.3.6.4 CP_EventHandler . . . 191
8. CMSIS Implementation Library Reference . . . 193
8.1 SystemCoreClockUpdate . . . 193
8.2 SystemInit . . . 194
9. System Library Reference . . . 195
9.1 BLE_DeviceParam_Set_ADV_IFS . . . 195
9.2 BLE_DeviceParam_Set_AdvDelay . . . 196
9.3 BLE_DeviceParam_Set_ClockAccuracy . . . 197
9.4 BLE_DeviceParam_Set_ForcedClockAccuracy . . . 198
9.5 BLE_DeviceParam_Set_MaxNumRAL . . . 199
9.6 BLE_DeviceParam_Set_MaxRxOctet . . . 200
9.7 Device_Param_Prepare . . . 201
9.8 Device_Param_Read . . . 202
9.9 Sys_ADC_Clear_BATMONStatus . . . 203
9.10 Sys_ADC_Get_BATMONStatus . . . 204
9.11 Sys_ADC_Get_Config . . . 205
9.12 Sys_ADC_InputSelectConfig . . . 206
9.13 Sys_ADC_Set_BATMONConfig . . . 207
9.14 Sys_ADC_Set_BATMONIntConfig . . . 208
9.15 Sys_ADC_Set_Config . . . 209
9.16 Sys_AES_Cipher . . . 210
9.17 Sys_AES_Config . . . 212
9.18 Sys_ASRC_CalcPhaseCnt . . . 213
9.19 Sys_ASRC_CheckInputConfig . . . 214
9.20 Sys_ASRC_Config . . . 215
9.21 Sys_ASRC_ConfigRunTime . . . 216
9.22 Sys_ASRC_InputData . . . 217
9.23 Sys_ASRC_IntEnableConfig . . . 218
9.24 Sys_ASRC_OutputCount . . . 219
9.25 Sys_ASRC_OutputData . . . 220
9.26 Sys_ASRC_PhaseIncConfig . . . 221
9.27 Sys_ASRC_Reset . . . 222
9.28 Sys_ASRC_ResetOutputCount . . . 223
9.29 Sys_ASRC_Status . . . 224
9.30 Sys_ASRC_StatusConfig . . . 225
9.31 Sys_Audio_DMICDIOConfig . . . 226
9.32 Sys_Audio_ODDIOConfig . . . 227
9.33 Sys_Audio_ODDIOConfigMult . . . 228
9.34 Sys_Audio_Set_Config . . . 229
9.35 Sys_Audio_Set_DMICConfig . . . 230
9.36 Sys_Audio_Set_ODConfig . . . 231
9.37 Sys_Audiosink_Config . . . 232
9.38 Sys_Audiosink_Counter . . . 233
9.39 Sys_Audiosink_InputClock . . . 234
9.40 Sys_Audiosink_PeriodCounter . . . 235
9.41 Sys_Audiosink_PhaseCounter . . . 236
9.42 Sys_Audiosink_ResetCounters . . . 237
9.43 Sys_Audiosink_Set_Ctrl . . . 238
9.44 Sys_BBIF_ConnectRFFE . . . 239

9.45 Sys_BBIF_DIOConfig 240
9.46 Sys_BBIF_RFFE 241
9.47 Sys_BBIF_RFFEDrivenExternal 242
9.48 Sys_BBIF_SPIConfig 243
9.49 Sys_BBIF_SyncConfig 244
9.50 Sys_BootROM_Reset 245
9.51 Sys_BootROM_StartApp 246
9.52 SYS_BOOTROM_STARTAPP_RETURN 247
9.53 Sys_BootROM_StrictStartApp 248
9.54 Sys_BootROM_ValidateApp 249
9.55 Sys_Clocks_ClkDetEnable 250
9.56 Sys_Clocks_Osc 251
9.57 Sys_Clocks_Osc32kCalibratedConfig 252
9.58 Sys_Clocks_Osc32kHz 253
9.59 Sys_Clocks_OscRCCalibratedConfig 254
9.60 Sys_Clocks_Set_ClkDetConfig 255
9.61 Sys_Clocks_SystemClkConfig 256
9.62 Sys_Clocks_SystemClkPrescale0 257
9.63 Sys_Clocks_SystemClkPrescale1 258
9.64 Sys_Clocks_SystemClkPrescale2 259
9.65 Sys_CRC_Calc 260
9.66 Sys_CRC_Check 261
9.67 Sys_CRC_Get_Config 262
9.68 Sys_CRC_Set_Config 263
9.69 Sys_Delay_ProgramROM 264
9.70 Sys_DIO_Config 265
9.71 Sys_DIO_Get_Mode 266
9.72 Sys_DIO_IntConfig 267
9.73 Sys_DIO_NMIConfig 268
9.74 Sys_DIO_Set_Direction 269
9.75 Sys_DMA_ChannelConfig 270
9.76 Sys_DMA_ChannelDisable 272
9.77 Sys_DMA_ChannelEnable 273
9.78 Sys_DMA_ClearAllChannelStatus 274
9.79 Sys_DMA_ClearChannelStatus 275
9.80 Sys_DMA_Get_ChannelStatus 276
9.81 Sys_DMA_Set_ChannelDestAddress 277
9.82 Sys_DMA_Set_ChannelSourceAddress 278
9.83 Sys_Flash_Compare 279
9.84 Sys_Flash_Copy 280
9.85 Sys_Flash_ECC_Config 281
9.86 Sys_GPIO_Set_High 282
9.87 Sys_GPIO_Set_Low 283
9.88 Sys_GPIO_Toggle 284
9.89 Sys_I2C_ACK 285
9.90 Sys_I2C_Config 286
9.91 Sys_I2C_DIOConfig 287
9.92 Sys_I2C_Get_Status 288
9.93 Sys_I2C_LastData 289
9.94 Sys_I2C_NACK 290
9.95 Sys_I2C_NACKAndStop 291

9.96 Sys_I2C_Reset . . . 292
9.97 Sys_I2C_StartRead . . . 293
9.98 Sys_I2C_StartWrite . . . 294
9.99 Sys_Initialize . . . 295
9.100 Sys_Initialize_Base . . . 296
9.101 Sys_IP_Lock . . . 297
9.102 Sys_IP_Unlock . . . 298
9.103 Sys_LPDSP32_Command . . . 299
9.104 Sys_LPDSP32_DIOJTAG . . . 300
9.105 Sys_LPDSP32_Get_ActivityCounter . . . 301
9.106 Sys_LPDSP32_IntClear . . . 302
9.107 Sys_LPDSP32_Pause . . . 303
9.108 Sys_LPDSP32_Reset . . . 304
9.109 Sys_LPDSP32_Run . . . 305
9.110 Sys_LPDSP32_Run_Status . . . 306
9.111 Sys_LPDSP32_RuntimeAddr . . . 307
9.112 Sys_LPDSP32_Set_DebugConfig . . . 308
9.113 Sys_NVIC_ClearAllPendingInt . . . 309
9.114 Sys_NVIC_DisableAllInt . . . 310
9.115 Sys_PCM_ClearStatus . . . 311
9.116 Sys_PCM_Config . . . 312
9.117 Sys_PCM_ConfigClk . . . 313
9.118 Sys_PCM_DIOConfig . . . 314
9.119 Sys_PCM_Disable . . . 315
9.120 Sys_PCM_Enable . . . 316
9.121 Sys_PCM_Get_Status . . . 317
9.122 Sys_Power_BandGapCalibratedConfig . . . 318
9.123 Sys_Power_BandGapConfig . . . 319
9.124 Sys_Power_BandGapStatus . . . 320
9.125 Sys_Power_DCDCCalibratedConfig . . . 321
9.126 Sys_Power_Get_ResetAnalog . . . 322
9.127 Sys_Power_Get_ResetDigital . . . 323
9.128 Sys_Power_ResetAnalogClearFlags . . . 324
9.129 Sys_Power_ResetDigitalClearFlags . . . 325
9.130 Sys_Power_VCCConfig . . . 326
9.131 Sys_Power_VDDAConfig . . . 327
9.132 Sys_Power_VDDCCalibratedConfig . . . 328
9.133 Sys_Power_VDDCConfig . . . 329
9.134 Sys_Power_VDDCStandbyCalibratedConfig . . . 330
9.135 Sys_Power_VDDMCalibratedConfig . . . 331
9.136 Sys_Power_VDDMConfig . . . 332
9.137 Sys_Power_VDDMStandbyCalibratedConfig . . . 333
9.138 Sys_Power_VDDPACalibratedConfig . . . 334
9.139 Sys_Power_VDDPAConfig . . . 335
9.140 Sys_Power_VDDRFCalibratedConfig . . . 336
9.141 Sys_Power_VDDRFConfig . . . 337
9.142 Sys_PowerModes_Sleep . . . 338
9.143 Sys_PowerModes_Sleep_Init . . . 339
9.144 Sys_PowerModes_Sleep_Init_2Mbps . . . 340

9.145 Sys_PowerModes_Sleep_WakeupFromFlash 341
9.146 Sys_PowerModes_Standby 342
9.147 Sys_PowerModes_Standby_Wakeup 343
9.148 Sys_PowerModes_Wakeup 344
9.149 Sys_PowerModes_Wakeup_2Mbps 345
9.150 Sys_ProgramROM_UnlockDebug 346
9.151 Sys_PWM_Config 347
9.152 Sys_PWM_ConfigAll 348
9.153 Sys_PWM_Control 349
9.154 Sys_PWM_DIOConfig 350
9.155 Sys_PWM_Enable 351
9.156 Sys_ReadNVR4 352
9.157 Sys_RFFE_InputDIOConfig 353
9.158 Sys_RFFE_OutputDIOConfig 354
9.159 Sys_RFFE_SetTXPower 355
9.160 Sys_RFFE_SPIDIOConfig 356
9.161 Sys_RTC_Config 357
9.162 Sys_RTC_Start 358
9.163 Sys_RTC_Value 359
9.164 Sys_SPI_Config 360
9.165 Sys_SPI_DIOConfig 361
9.166 Sys_SPI_MasterInit 362
9.167 Sys_SPI_Read 363
9.168 Sys_SPI_ReadWrite 364
9.169 Sys_SPI_TransferConfig 365
9.170 Sys_SPI_Write 366
9.171 Sys_Timer_BBConfig 367
9.172 Sys_Timer_Get_Status 368
9.173 Sys_Timer_Set_Control 369
9.174 Sys_Timers_Start 370
9.175 Sys_Timers_Stop 371
9.176 Sys_UART_DIOConfig 372
9.177 Sys_UART_Disable 373
9.178 SYS_WAIT_FOR_EVENT 374
9.179 SYS_WAIT_FOR_INTERRUPT 375
9.180 Sys_Watchdog_Refresh 376
9.181 Sys_Watchdog_Set_Timeout 377
10. Math Library Reference 378
10.1 Math_Add_frac32 378
10.2 Math_AttackRelease 379
10.3 Math_AttackRelease_frac32 380
10.4 Math_ExpAvg 381
10.5 Math_ExpAvg_frac32 382
10.6 Math_LinearInterp 383
10.7 Math_LinearInterp_frac32 384
10.8 Math_Mult_frac32 385
10.9 Math_SingleVar_Reg 386
10.10 Math_Sub_frac32 387
11. Flash Library Reference 388
11.1 Flash_EraseAll 388

11.2 Flash_EraseSector . . . 389
11.3 Flash_WriteBuffer . . . 390
11.4 Flash_WriteCommand . . . 391
11.5 Flash_WriteInterfaceControl . . . 392
11.6 Flash_WriteWordPair . . . 393
12. Calibration Library Reference . . . 394
12.1 Calibrate_Clock_32K_RCOSC . . . 394
12.2 Calibrate_Clock_Initialize . . . 395
12.3 Calibrate_Clock_Start_OSC . . . 396
12.4 Calibrate_Power_DCDC . . . 397
12.5 Calibrate_Power_Initialize . . . 398
12.6 Calibrate_Power_VBG . . . 399
12.7 Calibrate_Power_VDDC . . . 400
12.8 Calibrate_Power_VDDM . . . 401
12.9 Calibrate_Power_VDDPA . . . 402
12.10 Calibrate_Power_VDDRF . . . 403
A. Glossary . . . 404

CHAPTER 1

Introduction

1.1 Purpose

This manual describes the firmware for RSL10. The firmware provides developers with a convenient software layer on which to build their applications. It is also responsible for system-level tasks such as coordinating Bluetooth communications, booting the system and implementing portions of the device security layer. It consists of include files, libraries, and ROM code. This manual includes descriptions, function listings, and usage examples to help you to understand the firmware and its parts.

1.2 Intended Audience

This manual is for developers who are designing and implementing applications for RSL10. Both novice and experienced developers can benefit from this information.

This manual assumes the reader has a basic understanding of:

- C and the fundamentals of the Arm® Thumb-2 assembly language
- The integrated development environment and toolchains that form the development tools
- RSL10 architecture

1.3 Conventions

The following conventions are used in this manual to signify particular types of information:

`monospace font`
: Assembly code, macros, functions, defines and addresses.

italics
: File and path names, or any portion of them.

`<angle brackets>`
: Optional parameters and placeholders for specific information. To use an optional parameter or replace a placeholder, specify the information within the brackets; do not include the brackets themselves.

1.4 Further Reading

For more information about the Event Kernel, Bluetooth Stack and Profile library interface specifications, refer to the documents referenced in Chapter 4, “Event Kernel” on page 32 and Chapter 6, “Bluetooth Stack and Profiles” on page 50.

For more information about RSL10, refer to the following documents:

- *RSL10 Hardware Reference*
- *RSL10 Getting Started Guide*
- The datasheet for RSL10
- *Bluetooth Core Specification* v5.0, available from https://www.bluetooth.com/specifications/adopted-specifications

For more information about the Arm® Cortex®-M3 processor, refer to the following documents:

- *ARMv7M Architecture Reference Manual*
- *ARM and Thumb-2 Instruction Set Quick Reference Card*
- *ARM Core Cortex-M3/Cortex-M3 with ETM (AT420/AT425) Errata Notice*
- *ARM Cortex-M3 Processor Toolchain Reference for Ezairo 7100*

CHAPTER 2

Firmware Overview

2.1 Introduction

RSL10 is supported by firmware sets that provide:

- A thin layer of support between the hardware and the developer. This firmware allows the developer to focus on application development and reduces the number of details a developer is required to know about the underlying RSL10 hardware.
- A support layer for common complex operations that will be used by user applications.
- Wireless protocol support functionality for Bluetooth low energy and custom protocol support.

The system firmware provides an interface for common operations that is easier to use and understand than low-level C or assembly code. For instance, you can control and configure the underlying hardware while avoiding the use of absolute addresses, which helps you to produce code quickly, with fewer errors. The support and wireless protocol firmware provide more advanced functionality that can form the basis for any application that has been developed to take advantage of the hardware provided by the RSL10 SoC.

When multiple programmers are involved in development, using the firmware leads to increased consistency, which in turn leads to increased overall robustness and correctness of code.

In some cases, depending on the particulars of the application, the firmware implementation might not be optimal; however, even in these situations, the firmware serves as an example and advanced starting point for custom-developed functions and macros.

The firmware also encapsulates the details of the hardware such that many changes due to hardware revisions are transparent at the application level. The encapsulation also provides a basic level of error checking of the hardware usage. Compatibility information for firmware and hardware revisions is described in Section 2.5, “Versions” on page 21.

2.2 Firmware Components

Firmware supporting the RSL10 SoC can be divided into three groups:

System Firmware

This set of firmware provides a layer of support between the application developer and the underlying hardware. It is a collection of files, macros, functions and libraries designed to make application development simpler, quicker and more reliable. The firmware serves many functions, including performing system-level tasks (e.g., switching applications), implementing part of the security functionality, power mode configuration, and improving application readability.

The system support firmware consists of:

- Hardware definitions such as a register description, memory maps, interrupt vectors, and related components. The firmware applies a layer of names and labels to the underlying hardware for ease of use. This firmware is described in Chapter 3, “Hardware Definitions” on page 23.
- A C-startup implementation for applications paired with a Cortex Microcontroller Software Interface Standard (CMSIS) compliant application template provided by the CMSIS

implementation library. Reference information for this library is provided in Chapter 8, “CMSIS Implementation Library Reference” on page 193.
- A set of macros, in-line functions and other functions that provide basic and complex system functionality to simplify development provided through the System Library. Reference material for this library is provided in Chapter 9, “System Library Reference” on page 195.
- A set of functions providing fixed point mathematics and an extension of standard floating point functions from *math.h*, provided through the Math Library. Reference material for this library is provided in Chapter 10, “Math Library Reference” on page 378.

Support Firmware

This set of firmware consists of more complex firmware components that are used to complete complex tasks which can be used as an extension to an application, or can be used as the common core functionality of an application.

The support firmware consists of:

- An event handler kernel that can be used to exchange and store messages, schedule events, and register call-back functions that respond to other events that have occurred in the system. More information about this kernel is provided in Chapter 4, “Event Kernel” on page 32.
- The Program ROM which contains code for ensuring that the system starts and restarts in known states with known behaviors. The Program ROM loads information for the security implementations. The Program ROM also integrates the Flash Write Support Library and provides implementations to a number of ROM vector-based functions that are provided by the System Library. For more information about the Program ROM, see Chapter 5, “Program ROM” on page 43.
- A set of functions for erasing and writing the contents of flash memory provided through the Flash Write Support Library. Reference material for this library is provided in Chapter 11, “Flash Library Reference” on page 388.
- A set of macros and functions that can be used to calibrate system components to non-standard target values to supplement the calibration settings calculated during device manufacturing and provided in NVR4. This functionality is provided by the Supplemental Calibration Library, and reference material for this library is provided in Chapter 12, “Calibration Library Reference” on page 394.

Bluetooth Protocol Stack and Profiles and Custom Protocols

This set of firmware implements a Bluetooth low energy stack and a set of Bluetooth profiles. This firmware consists of a collection of include files and pre-compiled libraries that are designed to make Bluetooth-based application development simpler and quicker, while meeting the interoperability test requirements required for Bluetooth certification. Information about the Bluetooth support firmware is provided in Chapter 6, “Bluetooth Stack and Profiles” on page 50.

This set of firmware also contains include files and libraries implementing ON Semiconductor defined custom protocols. These protocols can be used as is, or can be used as the starting point for user-defined custom protocols.

All firmware components listed above execute on the Arm Cortex-M3 processor, and all of these components are CMSIS-compatible.

2.2.1 Firmware Files

The firmware files consist of include files (denoted with *.h* extensions), and precompiled library binaries (denoted with *.a* extensions). Some precompiled libraries are also provided in source code format.

The Arm Cortex-M3 processor firmware include files and libraries for each of the three groups of firmware are listed in Table 1, Table 2, and Table 3/Table 4. Applications that use the libraries provided must:

- Include the associated firmware include file.
- Link against any dependencies of the desired library.
- Link against a version of the desired library.

Table 1. Arm Cortex-M3 Processor System Firmware Include Files and Libraries

Firmware Component	Include Files	Library	Dependencies
Hardware Definition	*rsl10_map.h* *rsl10_map_nvr.h* *rsl10_reg.h* *rsl10_vectors.h*	N/A	N/A
CMSIS Include Files and library	*arm_common_tables.h* *arm_math.h* *core_cm3.h* *core_cmFunc.h* *core_cmInstr.h* *rsl10.h* *rsl10_start.h* *system_rsl10.h*	*libcmsis.a*	-

Table 1. Arm Cortex-M3 Processor System Firmware Include Files and Libraries (Continued)

Firmware Component	Include Files	Library	Dependencies
System Macros and Library	*rsl10_romvect.h* *rsl10_sys_aes.h* *rsl10_sys_asrc.h* *rsl10_sys_audio.h* *rsl10_sys_audiosink.h* *rsl10_sys_bbif.h* *rsl10_sys_clocks.h* *rsl10_sys_cm3.h* *rsl10_sys_crc.h* *rsl10_sys_dio.h* *rsl10_sys_dma.h* *rsl10_sys_flash.h* *rsl10_sys_gpio.h* *rsl10_sys_i2c.h* *rsl10_sys_ip.h* *rsl10_sys_lpdsp32.h* *rsl10_sys_adc.h* *rsl10_sys_mem.h* *rsl10_sys_pcm.h* *rsl10_sys_power.h* *rsl10_sys_power_modes.h* *rsl10_sys_pwm.h* *rsl10_sys_rffe.h* *rsl10_sys_rtc.h* *rsl10_sys_spi.h* *rsl10_sys_timers.h* *rsl10_sys_uart.h* *rsl10_sys_watchdog.h*	*libsyslib.a*	CMSIS library (or equivalent)
Math Library	*rsl10_math.h*	*libmathlib.a*	-
Flash Library	*rsl10_flash.h*	*libflashlib.a*	-

Table 2. Arm Cortex-M3 Processor Support Firmware Include Files and Libraries

Firmware Component	Include Files	Library	Dependencies
Event Kernel	*rsl10_ke.h*	*libkelib.a*	-
Calibration Library	*rsl10_calibrate.h*	*libcalibratelib.a*	System library

Table 3. Bluetooth Support Firmware Include Files and Libraries

Firmware Component	Include Files	Library	Dependencies
Bluetooth Stack	*rsl10_bb.h* *rsl10_ble.h*	*libblelib.a*	Event kernel
Bluetooth Profiles	*rsl10_profiles.h*	*libanpc.a* *libanps.a* *libbasc.a* *libbass.a* *libblpc.a* *libblps.a* *libcppc.a* *libcpps.a* *libcscpc.a* *libcscps.a* *libdisc.a* *libdiss.a* *libfindl.a* *libfindt.a* *libglpc.a* *libglps.a* *libhogpbh.a* *libhogpd.a* *libhogprh.a* *libhrpc.a* *libhrps.a* *libhtpc.a* *libhtpt.a* *liblanc.a* *liblans.a* *libpaspc.a* *libpasps.a* *libproxm.a* *libproxr.a* *librscpc.a* *librscps.a* *libscppc.a* *libscpps.a* *libtipc.a* *libtips.a* *libwptc.a* *libwpts.a*	Event kernel, Bluetooth stack

Table 4. Custom Protocol Firmware Include Files and Libraries

Firmware Component	Include Files	Library	Dependencies
Custom Low-Latency Audio Profile	*cp_pkt.h*	*libcustom_protocolLib.a*	System library

The event kernel support firmware and the Bluetooth protocol stack are available in the following formats:

- Debug, debug with HCI access, debug (light)
- Release, release with HCI access, release (light)

The Bluetooth profiles are available in the following formats:

- Debug, debug with HCI access
- Release, release with HCI access

The remaining libraries are available in the following formats:

- Source code
- Debug
- Debug for RSL10 beta revision 1.02 (RSL10_CID = 8102)
- Release

2.2.2 Compliance Exceptions

The firmware provided for the Arm Cortex-M3 processor is generally compliant with the MISRA-C:2004 rules, as required by the CMSIS standard. The RSL10 firmware exceptions in compliance are the same compliance exceptions that are part of the CMSIS Core standard.

The RSL10 firmware and CMSIS-CORE violate the following MISRA-C:2004 rules:

- Required Rule 8.5, object/function definition in header file. Violated because function definitions in header files are used to allow “inlining” of functions.
- Required Rule 18.4, declaration of union type or object of union type: { . . . }. Violated because unions are used for effective representation of core registers.
- Advisory Rule 19.7, Function-like macro defined. Violated because function-like macros are used to allow for more efficient code.

2.3 Firmware Naming Conventions

For clarity and ease of use, the firmware follows several naming conventions for library functions and macros. These conventions are compatible with the CMSIS naming requirements.

The macros provided for the Arm Cortex-M3 processor control the registers and peripherals of the processor and surrounding system. Arm Cortex-M3 processor macros that are implemented using a `#define` statement use all capitals in the macro name. These macros are prefixed with an all-capital prefix indicating the library they are supporting (e.g., `SYS_`). If the macro supports a specific target component, this prefix is followed by the name of the component it supports. The rest of the macro name indicates the intended functionality of the macro.

Inline and standard firmware functions for the Arm Cortex-M3 processor use camel-case function names (e.g., `CalcPhaseCnt`). All functions use a prefix to indicate which library provides the function (e.g., `Sys_`). The remainder of a function’s name indicates the block they affect and their intended functionality.

Table 5 lists the prefixes for each of the firmware libraries.

Table 5. Library Function Naming Convention

Library	Macro Prefix	Function Prefix
CMSIS Implementation Library	None[1]	None
System Library	SYS_	Sys_
Math Library	N/A	Math_
Flash Write Support Library	N/A	Flash_
Supplemental Calibration Library	N/A	Calibrate_

1. The CMSIS standard provides standard names for all CMSIS macros and functions, so no prefixes are used.

2.4 Firmware Resource Usage

The firmware uses the Arm Cortex-M3 processor system stack. It expects that the Arm Cortex-M3 processor stack pointer points to a valid stack that grows downward (i.e., decreasing memory addresses).

The firmware and sample code also recommend use cases for the non-volatile memory records, as described in Section 3.3, “Non-Volatile Record Memory Map” on page 24. To take advantage of the firmware design, we recommend that applications follow the use cases provided by these non-volatile memory records.

2.5 Versions

2.5.1 Hardware Variants and Firmware Compatibility

To simplify identification of systems-on-chips that are compatible with a given set of firmware, the RSL10 SoC provides chip version information using the AHBREGS_CHIP_ID_NUM register which can be used to calculate the chip identifier (CID) used by the firmware. Devices that share a CID are compatible with firmware built for that CID.

The CID is a two-byte value with the most significant byte set to the value of the AHBREGS_CHIP_ID_NUM_CHIP_VERSION bit-field, and the least significant byte set to the value of the AHBREGS_CHIP_ID_NUM_CHIP_MAJOR_REVISION bit-field. Devices with the same CID but different minor revisions (readable from the AHBREGS_CHIP_ID_NUM_CHIP_MINOR_REVISION bit-field) are compatible with the same firmware packages. See the *RSL10 Hardware Reference* for more information.

When including *rsl10.h*, specify the CID by using the symbol RSL10_CID. Set this symbol to the target platform either in the source files or through the project build settings. The following example instructs the system libraries to provide the CID 101 variant of the constants:

```
#define RSL10_CID 101              // Target RSL10 version 1.01.xx
#include <rsl10.h>
```

By default, new projects include the correct settings to target CID 101 for the RSL10 chip. To modify the target CID, update the build settings for your project. You can change the chip ID using build properties or preprocessor settings.

2.5.2 Firmware Versions

Version symbols are provided for each major system firmware component and most support firmware components. The version symbols can be used directly or indirectly to verify the version of the components used to build an application. There are two types of version symbols available:

Define A preprocessor define containing the version information

Constants A constant global variable value included in each library, which contains the version information for that library

The available version information for each firmware component is listed in Table 6.

Table 6. Firmware Version Symbols

Component	Define	Constant
CMSIS and System Library	`RSL10_SYS_VER`	`RSL10_Sys_Version`
Flash Library	`FLASH_FW_VER`	`RSL10_FlashLib_Version`
Math Library	`MATH_FW_VER`	`RSL10_MathLib_Version`
Calibration Library	`CALIBRATE_FW_VER`	`RSL10_CalibrateLib_Version`
Program ROM	N/A	`PROGRAM_ROM_VERSION`[1]

1. The Program ROM version can be read from 0x0000 001C

The version information contains a major version, minor version and revision. The major version is updated to indicate significant changes to the component. Significant changes can involve a total redesign of the component, including its interfaces and functionality. The minor version is updated to indicate minor changes or additions that are usually backward-compatible. This would generally indicate a change to the interfaces or underlying functionality, including different register modifications or leaving a different system state. The revision is updated to indicate a fairly insignificant change to the component. For example, the revision is updated when the interfaces and functionality of the component remain the same, and some other change has occurred. This can include non-functional changes to the source code, or optimizations that do not affect the calling application.

The major version, minor version and revision are contained in a 16-bit value. The version is described as Major.Minor.Revision. For example, Math Library v1.1.0 (0x1100) indicates major version 1, minor version 1, and revision 0. Table 7 shows the bit fields in the version symbols.

Table 7. Firmware Version Bit Fields

Bits 15-12	Bits 11-8	Bits 7-0
Major Version	Minor Version	Revision

CHAPTER 3

Hardware Definitions

The Arm Cortex-M3 processor on the RSL10 chip is supported by a set of header files and system library functions. These provide a description that defines the Arm Cortex-M3 processor subsystem of the RSL10 SoC. This includes:

- Register and bit descriptions for control and configuration registers in the Arm Cortex-M3 core memory map
- A memory map for the non-volatile records (NVR*) areas accessible to the Arm Cortex-M3 processor
- Interrupt vector table descriptions
- Macros that support basic Arm Cortex-M3 core functionality

The format and configuration of all of these firmware components conform to CMSIS compatibility requirements. Therefore, most of the system library consists of inline functions that are defined within the library header files.

The top-level include file for the system library, *rsl10.h*, combines all of the system hardware definition, system support macro and system library firmware components provided for the Arm Cortex-M3 processor subsystem of the RSL10 chip. If an application includes this file and defines `SL1_CID` to match the chip identifier of RSL10, then all of the support macros and system library functions that are available to support Arm Cortex-M3 processor development on the RSL10 chip are made accessible to that application.

Hardware definition files are integral to the system firmware. The hardware definitions apply a layer of data structures and address mappings to the underlying hardware, so that every control register and bit field in the system is easily accessible from C code.

3.1 Register and Register Bit-field Definition

Using the hardware definition files allows you to refer to system components by C structures and preprocessor symbols instead of by addresses and bit fields. This greatly improves the readability, reliability and maintainability of your application code. The use of hardware definitions in an application also means that some hardware changes, such as changes to addresses or bit field values, are transparent to your application code.

Hardware register descriptions for components that are linked to the RSL10 Arm Cortex-M3 processor peripheral bus are defined in the file *rsl10_hw.h* and *rsl10_hw_cid*.h*. Register descriptions for standard Arm Cortex-M3 processor peripherals, such as NVIC and SysTick, are defined in the core CMSIS header file, core_cm3.h.

Hardware descriptions in the register include files provide definitions for the components listed in Table 8.

Table 8. Hardware Register Components

Item	Example	Description
Component Register Structure	`DIO_Type`	Provides a list of all of the registers that support a specified component, and the read/write types for those registers.
Component Register Instance	`DIO`	Link the component register structure to the underlying hardware or sets of hardware
Bit-Field Positions	`DIO_CFG_DRIVE_Pos`	Defines the base position for any bit-field within a register
Bit-Field Mask	`DIO_CFG_DRIVE_Mask`	Defines a bit mask for any bit-field of more than one bit within a register

Table 8. Hardware Register Components (Continued)

Item	Example	Description
Register Structure	`DIO_CFG_Type`	Provides a list of all of the sub-registers and alias structures. • Sub-registers are defined byte (8-bit) or short (16-bit) access interfaces to part of a register that includes all elements belonging to the same configuration area. • Aliases are Arm Cortex-M3 processor bitband aliases that provide bit access to individual single-bit bit-fields where the underlying hardware supports this single-bit access.
Register Instance	`DIO_CFG`	Link the register structure to the underlying hardware or sets of hardware for sub-registers
Bit-Setting	`DIO_MODE_GPIO_IN_0`	Defines providing human-readable equivalents to settings that can be applied to a register bit-field to obtain the desired behavior.
Bit-Setting (bitband)	`DIO_LPF_DISABLE_BITBAND`	Alternate forms of a bit-setting that apply when using bitband aliases to read/write single register bits.
Bit-Field Sub-Register Positions	`DIO_CFG_IO_MODE_BYTE_Pos`	Defines the base position for any bit-field within a register's sub-register
Bit-Field Sub-Register Mask	`DIO_CFG_IO_MODE_BYTE_Mask`	Defines a bit mask for any bit-field of more than one bit within a register's sub-register
Sub-Register Bit-Setting	`DIO_MODE_GPIO_IN_0_BYTE`	Defines providing human-readable equivalents to settings that can be applied to a register's sub-register bit-field to obtain the desired behavior.

3.2 Memory Map Definition

Memory map definitions from the perspective of the Arm Cortex-M3 processor are provided in *rsl10_map.h*. Specifically, this file defines the locations of the following structures:

- Instruction and data bus memory structures
- System bus memory structures
- Peripheral bus memory-mapped control registers (including the base of control register groups for each system component)
- Private peripheral bus internal memory-mapped control registers
- System variables

For more information on the Arm Cortex-M3 processor memory map see the *RSL10 Hardware Reference*.

3.3 Non-Volatile Record Memory Map

A second set of memory map definitions are provided in *rsl10_map_nvr.h* for the non-volatile records (NVR) sections of flash that are used to hold system information, including:

- Application specific information (NVR1)
- Address and key information for bonded devices (NVR2)
- Device configuration information (NVR3)
  - The local device's Bluetooth address information
  - IP protection configuration
  - An initialization function that can be called by the ROM to load calibrated settings to their desired registers

- Manufacturing information (NVR4)
  - Calibration settings for power supplies and clocks
  - The delays needed to write to the local flash instance
  - Manufacturing and test information

3.3.1 Application Specific Record

We recommend that information stored to non-volatile record 1 (NVR1) relate to a single user application or user application set. The defined fields described in Table 9 must be defined for the user application or user application set. Data in all other locations from NVR1 are not used by the firmware and should be application defined.

Table 9. Application Specific Information

Address	Field	Description
0x00080000	SYS_INFO_START_ADDR	Location of the base of the default user application to be booted by the ROM (if no valid vector table exists at the application address, fails back to attempting to boot from the base of flash; if this attempt fails, any previous error from attempting to boot the application at SYS_INFO_START_ADDR is overwritten by the failure code for the attempt to boot from the base of flash).
0x00080004	SYS_INFO_START_MEM_CFG	Bit-field indicating which memories to enable for this application when rebooting from sleep mode. Use the defines for SYSCTRL_MEM_POWER_CFG, with the specified memory configuration used if at least PROM, flash, and DRAM0 are enabled and the system is waking up from sleep mode.

3.3.2 Bond Information Record

Information stored to non-volatile record 2 (NVR2) includes data to be used for bonded device information. Each bonded device has a record of BondInfo_Type, accessible through BOND_INFO as defined in *rsl10_map_nvr.h*. This record contains all of the stored values needed to create a whitelist or otherwise identify a bonded device. This is limited by SIZEOF_WHITELIST = 28. A description of the records stored is provided in Table 10. See the *Bluetooth Core Specification* v5.0 (https://www.bluetooth.com/specifications/adopted-specifications) for more information.

Table 10. Bond Information

Address Offset	Field	Description
0x00	STATE	State for this bonding record; 0xFF indicates an unused record, 0x00 indicates a used record that is no longer valid. Other settings for STATE can be used as an index for the record.
0x04	LTK	Long Term Key established with the bonded device
0x14	EDIV	Encrypted diversifier used to identify the LTK distributed during legacy pairing
0x18	ADDR	Address of the bonded device
0x1E	ADDR_TYPE	Address type for this bond record (public or static random address)
0x20	CSRK	Connection signature resolving key for the bonded device; used to authenticate signed data is received from this bonded device.
0x30	IRK	Identity Resolving Key used to generate and resolve random private addresses for this bonded device.
0x40	RAND	Stored random number used to identify the LTK distributed during legacy pairing

The stored bond information is valid if the INDEX is not all zeros or all ones.

3.3.3 Device Configuration Record

Information stored to non-volatile record 3 (NVR3) includes data to be used for device configuration. This includes the device's Bluetooth (MAC) address, the device lock configuration and key, and an optional initialization function that can be used to load configurations calibrated during manufacturing. A description of the records stored is provided in Table 11. The use of data in all unused locations in NVR3 are defined by the user's device and application.

Table 11. Device Configuration

Address	Field	Description
0x00081000	DEVICE_INFO_BLUETOOTH_ADDR	EUI-48 MAC address, to be used as the device's Bluetooth public address (see caution note below)
0x00081010	DEVICE_INFO_BLUETOOTH_IRK	128-bit identity resolving key (IRK), used to generate resolvable private addresses (RPA) when using Bluetooth privacy. If the device is communicating with more than one set of devices, where the device's privacy should be maintained between groups, additional IRKs can be created and stored in a separate user application defined location.
0x00081020	DEVICE_INFO_BLUETOOTH_CSRK	128-bit connection signature resolving key (CSRK), used to sign data when using signed data. If the device is signing data that will be provided to more than one set of devices, additional CSRKs can be created and stored in a separate user application defined location.
0x00081040	LOCK_INFO_SETTING	Value to be written to the debug lock configuration register; to restrict access after boot, set to DBG_ACCESS_LOCK. [1]
0x00081044	LOCK_INFO_KEY	128-bit key that can be used to override the DBG_ACCESS_LOCK configuration.
0x00081080	MANU_INFO_INIT	Manufacturing initialization function definition; requires a length field indicating the length of the function, the function implementation, and a CRC-CCITT calculated over the length field and the function's implementation code.
	MANU_INFO_LENGTH	Length of the version identifier and initialization function in bytes. The manufacturing information function must fit in NVR3.
0x00081082	-	Manufacturing initialization function version identifier (if used; otherwise, set to 0x0000)
0x00081084	-	Manufacturing initialization function implementation
0x00081082 + MANU_INFO_LENGTH	-	CRC-CCITT calculated over MANU_INFO_LENGTH and the manufacturing initialization function's implementation.

Table 11. Device Configuration (Continued)

Address	Field	Description
0x000817A0	DEVICE_INFO_ECDH_PRIVATE	256-bit private key from a locally generated Elliptic Curve Diffie-Hellman (ECDH) public-private key pair. This is used to generate keys needed for Bluetooth secure connections.
0x000817C0	DEVICE_INFO_ECDH_PUBLIC_X	256-bit public key (X) from a locally generated Elliptic Curve Diffie-Hellman (ECDH) public-private key pair. This is used to generate keys needed for Bluetooth secure connections.
0x000817E0	DEVICE_INFO_ECDH_PUBLIC_Y	256-bit public key (Y) from a locally generated Elliptic Curve Diffie-Hellman (ECDH) public-private key pair. This is used to generate keys needed for Bluetooth secure connections.

1. CAUTION: If the LOCK_INFO_SETTING is set to 0x0000 0000 or 0xFFFF FFFF, the device may not boot properly when VBAT < 1.25 V, as the ROM cannot differentiate between unreadable data and unwritten flash contents.

NOTE: A default implementation of the manufacturing initialization function is written during manufacturing to load the default calibrated settings from the manufacturing records (see Table 13 on page 29). The source for this default initialization function and the code needed to load this to NVR3 is provided as part of the sample code in the default MANU_INFO_INIT application. This application can be used to update this initialization function to provide other initialization behaviors prior to application boot.

CAUTION: The DEVICE_INFO_BLUETOOTH_ADDR value is set during testing to a unique EUI-48 MAC address for each device. Take care when erasing and writing NVR3 to preserve and restore this address if the device expects to use it as its Bluetooth device address.

3.3.4 Manufacturing Records

Information stored to non-volatile record 4 (NVR4) consists of information from test and manufacturing. This data is stored with hardware redundancy and cannot be written outside of manufacturing. Data stored to the manufacturing records includes:

1. Calibration settings used by firmware and user applications
2. Manufacturing and test records for traceability
3. Flash startup configuration

A description of the calibration records stored is provided in Table 12. The Availability Version column identifies the version of the manufacturing calibration process in which that field became available. Your version of the manufacturing calibration process is always obtainable from the MANU_INFO_VERSION field. Calibration records are generally split between a 16-bit calibration target value and a 16-bit trim setting that should be applied to the appropriate register to get the target calibration for a power supply or clock. Exceptions are the MANU_INFO_VCC field where each 16-bit trim setting is divided into two 8-bit fields for DCDC and LDO trimmings, and the MANU_INFO_ADC bit field structure which does not contain target calibration values. Up to four records can be stored for each element that is calibrated.

Table 12. Calibration Settings

Address	Field	Target Units	Trimmed Bit-field	Description	Availability Version
0x00081800	MANU_INFO_BANDGAP	mV / 10	VTRIM from ACS_BG_CTRL	Bandgap trim settings	0[1]
0x00081810	MANU_INFO_VDDRF	mV / 10	VTRIM from ACS_VDDRF_CTRL	VDDRF trim settings	21
0x00081820	MANU_INFO_VDDPA	mV / 10	VTRIM from ACS_VDDPA_CTRL	VDDPA trim settings; only used if VDDRF requirements exceed VCC supply	21
0x00081830	MANU_INFO_VDDC	mV / 10	VTRIM from ACS_VDDC_CTRL	VDDC trim settings	0
0x00081840	MANU_INFO_VDDC_STANDBY	mV / 10	STANDBY_VTRIM from ASC_VDDC_CTRL	VDDC trim settings for use in standby mode	0
0x00081850	MANU_INFO_VDDM	mV / 10	VTRIM from ACS_VDDM_CTRL	VDDM trim settings	0
0x00081860	MANU_INFO_VDDM_STANDBY	mV / 10	STANDBY_VTRIM from ACS_VDDM_CTRL	VDDM trim settings for use in standby mode	0
0x00081870	MANU_INFO_VCC[2]	mV / 10	VTRIM from ASC_VCC_CTRL	VCC trim settings for the LDO mode and for the DC-DC buck converter mode. Bits 0 - 4 are for LDO mode, and bits 8 - 12 are for DC-DC buck converter mode.	22
0x00081880	MANU_INFO_OSC_32K	Hz	FTRIM_32K from ACS_RCOSC_CTRL	32 kHz RC oscillator trim settings	0
0x00081890	MANU_INFO_OSC_RC	kHz	FTRIM_START from ACS_RCOSC_CTRL	RC start oscillator trim settings to be used without the RC oscillator multiplier	0
0x000818B0	MANU_INFO_OSC_RC_MULT	kHz	FTRIM_START from ACS_RCOSC_CTRL	RC start oscillator trim settings to be used with the RC oscillator multiplier	0
0x000818C0	MANU_INFO_ADC	N/A	DATA from ADC_OFFSET	ADC trim settings; the gain factor is calculated by ADC_DATA_AUDIO_CH[7:0] * gain/65536	27
0x000818F8	MANU_INFO_VERSION	-	N/A	Version of the manufacturing calibration record	0
0x000818FC	MANU_INFO_CRC	-	N/A	CRC-CCITT calculated over the calibration settings	0

1. If you need to distinguish between versions that are 0, contact your ON Semiconductor Customer Support representative.

2. For MANU_INFO_VERSION 21 or lower, this field was called MANU_INFO_DCDC, and it contained only the VCC trim settings for the DC-DC buck converter in bits 0 - 4.

Table 13 lists the pre-loaded calibration records that are calculated for each part during manufacturing. The default values from this table are loaded by the default implementation of the manufacturing initialization function stored to the device configuration record (see Table 11 on page 26).

Table 13. Manufacturing Calibrated Settings

Field	Targets	Description	Default
MANU_INFO_BANDGAP	0.75 V	Default bandgap trim	X
MANU_INFO_VDDRF	1.10 V	Default VDDRF trim setting; minimum setting for optimal RX sensitivity.	X
	1.07 V	Average VDDRF trimming for 0 dBm output power on channel 19 (VDDRF selected and VDDPA disabled)	
	1.20 V	Average VDDRF trimming for 2 dBm output power on channel 19 (VDDRF selected and VDDPA disabled)	
MANU_INFO_VDDPA	1.30 V	Default VDDPA trim setting	X
	1.26 V	Average VDDPA trimming for 3 dBm output power on channel 19	
	1.60 V	Average VDDPA trimming for 6 dBm output power on channel 19	
MANU_INFO_VDDC	1.15 V	Default VDDC trim setting; minimum value to allow for a safe boot across all VBAT supply levels and temperature	X
	0.92 V	Minimum VDDC voltage to ensure chip functionality at 16 MHz, with reduced ADC functionality	
	1.00 V	Minimum VDDC voltage to ensure accurate ADC functionality across temperature	
	1.05 V	Minimum VDDC voltage to ensure chip functionality at 48 MHz	
MANU_INFO_VDDC_STANDBY	0.8 V	Default VDDC standby trim setting	X
MANU_INFO_VDDM	1.15 V	Default VDDM trim setting; minimum value to allow for a safe boot across all VBAT supply levels and temperature	X
	1.05 V	Minimum VDDM voltage to ensure memory functionality	
	1.10 V	Minimum VDDM voltage to ensure chip functionality with VDDO = 3 V	
MANU_INFO_VDDM_STANDBY	0.8 V	Default VDDM standby trim setting	X
MANU_INFO_VCC or MANU_INFO_DCDC	1.20 V	Default DCDC trim setting; minimum required to guarantee VDDC, VDDM can reach 1.15 V	X
	1.12 V	Minimum required to guarantee VDDRF can reach 1.07 V (0 dBm output power)	
	1.15 V	Minimum required to guarantee VDDC and VDDM can reach 1.1 V	
	1.25 V	Minimum required to guarantee VDDRF can reach 1.20 V (2 dBm output power)	
MANU_INFO_OSC_32K	32768 Hz	Default trim setting for the 32 kHz RC oscillator	X
MANU_INFO_OSC_RC	3.00 MHz	Default trim setting for the startup RC oscillator (un-multiplied)	
MANU_INFO_OSC_RC_MULT	10.00 MHz	Default trim setting for the startup RC oscillator (multiplied)	X

Table 13. Manufacturing Calibrated Settings (Continued)

Field	Targets	Description	Default
MANU_INFO_ADC	N/A	ADC offset (low/high frequency mode) for VCC = 1.2 V	
		ADC gain (low frequency mode) for VCC = 1.2 V	
		ADC gain (high frequency mode) for VCC = 1.2 V	
		ADC offset (low/high frequency mode) for VCC = 1.12 V	
		ADC gain (low frequency mode) for VCC = 1.12 V	
		ADC gain (high frequency mode) for VCC = 1.12 V	
		ADC offset (low/high frequency mode) for VCC = 1.15 V	
		ADC gain (low frequency mode) for VCC = 1.15 V	
		ADC gain (high frequency mode) for VCC = 1.15 V	
		ADC offset (low/high frequency mode) for VCC =1.25V	
		ADC gain (low frequency mode) for VCC =1.25V	
		ADC gain (high frequency mode) for VCC =1.25V	

The bit fields for MANU_INFO_ADC are illustrated in Figure 1.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0008 18C0	ADC offset (high frequency mode) for VCC @1.2 V																ADC offset (low frequency mode) for VCC @1.2 V															
0x0008 18C4	0x0000																ADC gain (low frequency mode) for VCC @ 1.2 V															
0x0008 18C8	0x0000																ADC gain (high frequency mod) for VCC @ 1.2 V															
0x0008 18CC	ADC offset (high frequency mode) for VCC @1.12 V																ADC offset (low frequency mode) for VCC @1.12 V															
0x0008 18D0	0x0000																ADC gain (low frequency mode) for VCC @ 1.12 V															
0x0008 18D4	0x0000																ADC gain (high frequency mode) for VCC @ 1.12 V															
0x0008 18D8	ADC offset (high frequency mode) for VCC @1.15 V																ADC offset (low frequency mode) for VCC @1.15 V															
0x0008 18DC	0x0000																ADC gain (low frequency mode) for VCC @ 1.15 V															
0x0008 18E0	0x0000																ADC gain (high frequency mode) for VCC @ 1.15 V															
0x0008 18E4	ADC offset (high frequency mode) for VCC @1.25 V																ADC offset (low frequency mode) for VCC @1.25 V															
0x0008 18E8	0x0000																ADC gain (low frequency mode) for VCC @ 1.25 V															
0x0008 18EC	0x0000																ADC gain (high frequency mode) for VCC @ 1.25 V															

Figure 1. ADC Bit Fields in NVR4

The manufacturing records include MANU_INFO_BLUETOOTH_ADDR, which provides a copy of the Bluetooth public address that was written to DEVICE_INFO_BLUETOOTH_ADDR during manufacturing as a backup in case the information stored to the device information sector was accidentally erased.

Information stored to other manufacturing records is not intended for direct use by user applications, and their records are not described here.

3.4 Interrupt Vector Definition

Interrupt vector definitions are defined in *rsl10_vectors.h* for both internal and external interrupts. These definitions have the form **<interrupt_name>**_IRQn. You can use these definitions with the NVIC-related functions included with the core CMSIS (and defined in *core_cm3.h*). For a complete list of interrupts, see the *RSL10 Hardware Reference*.

The CMSIS Implementation library also provides default weakly defined interrupt handlers for each of these vectors. These interrupt handlers have the form **<interrupt_name>**_IRQHandler(). If a user application defines a function with this same name, the user application's definition of the interrupt handler will replace the default (empty) handlers.

CHAPTER 4

Event Kernel

4.1 Overview

4.1.1 Feature List

The RSL10 Kernel is a small and efficient event and message handling system that can be used as a Real Time Operating System (RTOS) or as a process executed under an RTOS, offering the following features:

- Exchange of messages
- Message saving
- Timer functionality
- The kernel also provides an event functionality used to defer actions

The purpose of the event kernel is to provide messages (such as the ones in Section 4.2, “Messages” on page 34) and timed tasks to keep RF traffic on schedule and aligned with the specification requirements.

4.1.2 Top-Level Objects

To use the services offered by the kernel, include the header file *rsl10_ke.h*.

In addition to the header file, always include the object file *libkelib.a*.

4.1.3 Include Files

Table 14. Kernel File List

File	Description
ke.h	Contains the kernel environment definition
ke_config.h	Contains all the constants that can be changed in order to tailor the kernel
ke_event.h	Contains the event handling primitives
ke_mem.h	Contains the implementation of the heap management module
ke_misc.h	This file contains the kernel initialization function and defines related to the environment definition.
ke_msg.h	This file contains the scheduler primitives called to create or delete a task. It also contains the scheduler itself
ke_task.h	Contains the implementation of the kernel task management
ke_timer.h	Contains the scheduler primitives called to create or delete a timer task. It also contains the timer scheduler itself

4.1.4 API Functions

The event kernel is supported by two primary functions as described in Table 15.

Table 15. Event Kernel Support Functions

Function	Description	Reference
`Kernel_Init`	Initialize the event kernel for use within an application.	4.1.4.1 on p. 33
`Kernel_Schedule`	Execute any pending events that have been scheduled with the event kernel.	4.1.4.2 on p. 33

4.1.4.1 Kernel_Init

Initialize the event kernel for use within an application.

Type	Function
Include File	`#include <rsl10_ke.h>`
Template	`void Kernel_Init(uint32_t mode)`
Description	Initialize the event kernel for use within an application.
Inputs	mode = Kernel initialization mode; set to 1 if using the kernel without the Bluetooth low energy stack, 0 otherwise.
Outputs	None
Assumptions	None

Example	`/* Initialize the kernel and Bluetooth stack */`
	`Kernel_Init(0);`
	`BLE_InitNoTL(0);`
	`BLE_Reset();`

4.1.4.2 Kernel_Schedule

Execute any pending events that have been scheduled with the event kernel.

Type	Function
Include File	`#include <rsl10_ke.h>`
Template	`void Kernel_Schedule(void)`
Description	Execute any pending events that have been scheduled with the event kernel.
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Main application loop:`
	` * - Run the kernel scheduler`
	` * - Refresh the watchdog and wait for an interrupt before continuing */`
	`while (1)`
	`{`
	`  Kernel_Schedule();`
	
	`  /* Refresh the watchdog timer */`
	`  Sys_Watchdog_Refresh();`
	
	`  /* Wait for an event before executing the scheduler again */`
	`  SYS_WAIT_FOR_EVENT;`
	
	`}`

4.1.5 Kernel Environment

The kernel environment structure contains the queues used for event, timer and message management:

queue_sent Queue of sent messages but not yet delivered to receiver

queue_saved Queue of messages delivered but not consumed by receiver

queue_timer Queue of timer

mblock_first Pointer to first element of linked list

If kernel profiling is enabled, the following fields are added:

max_heap_used Maximum heap memory used by the kernel

queue_timer Queue of messages delivered but not consumed by receiver

4.2 MESSAGES

4.2.1 Overview

Message queues provide a mechanism to transmit one or more messages to a task. (Queue names and purposes are defined in Section 4.1.5, “Kernel Environment” on page 33.)

Transmission of messages is performed in 3 steps:

- Sender task allocates a message structure
- Message parameters are filled
- Message structure is pushed in the kernel

A message is identified by a unique ID composed of the task type and an increasing number. The macro in Figure 2 builds the first message ID of a task.

```
#define KE_FIRST_MSG(task) ((ke_msg_id_t)((task) << 8))
```



Figure 2. First Message ID of Task

A message has a list of parameters that is defined in a structure (see Section 4.2.2, “Message Format”).

4.2.2 Message Format

The structure of the message contains:

- **`id`**: Message identifier
- **`dest_id`**: Destination kernel identifier
- **`src_id`**: Source kernel identifier
- **`param_len`**: Parameter embedded structure length
- **`param`**: Parameter embedded structure. Must be word-aligned.

4.2.3 Message Identifier

- Message identifier is defined as follows:

```
typedef unit16_t_ke_msg_id_t;
```

- The message identifier must be defined by task type in one file only to avoid multiple identical definitions.

In `xx_task.h` for XX task.

4.2.4 Parameter Management

During message allocation, the size of the parameter is passed and memory is allocated in the kernel heap. In order to store this data, the pointer on the parameters is returned. The scheduler frees this memory after the transition completion. For example:

```
void *ke_msg_alloc(ke_msg_id_t const id,
ke_task_id_t const dest_id,
ke_task_id_t const src_id,
uint16_t const param_len)
{
    struct ke_msg *msg = (struct ke_msg*) ke_malloc(sizeof(struct ke_msg) +
    param_len - sizeof (uint32_t));
    ...
    return param_ptr;
}
```

4.2.5 Message Queue Object

A Message queue is defined as a linked list composed of message elements:

- *first: pointer to first element of the list
- *last: pointer to the last element

If kernel profiling is enabled, these following fields are added:

- **`cnt`**: number of elements in the list
- **`maxcnt`**: maximum number of elements in the list
- **`mincnt`**: minimum number of elements in the list

4.2.6 Message Queue Primitives

4.2.6.1 Message Allocation

Prototype:

```
void *ke_msg_alloc(ke_msg_id_t const id, ke_task_id_t const dest_id, ke_task_id_t const
   src_id, uint16_t const param_len)
```

Parameters:

Table 16. Message Allocation Parameters

Type	Parameters	Description
`ke_msg_id_t`	`id`	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier
`unit16_t`	`param_len`	Length of Parameter

Return: Pointer to the parameter member of the `ke_msg`. If the parameter structure is empty, the pointer will point to the end of the message and must not be used (except to retrieve the message pointer or to send the message).

Description: This primitive allocates memory for a message that has to be sent. The memory is allocated dynamically on the heap, and the length of the variable parameter structure must be provided in order to allocate the correct size.

4.2.6.2 Message Send

Prototype:

```
void ke_msg_send(void const *param_ptr)
```

Parameters:

Table 17. Message Send Parameters

Type	Parameters	Description
`void const *`	`param_ptr`	Pointer to the parameter member of the message that will be sent

Return: None

Description: Send a message previously allocated with any `ke_msg_alloc()`-like functions. The kernel will take care of freeing the message memory.

Once the function has been called, it is not possible to access its data any more as the kernel might have copied the message and freed the original memory.

4.2.6.3 Message Send Basic

Prototype:

```
void ke_msg_send_basic(ke_msg_id_t const id, ke_task_id_t const dest_id, ke_task_id_t
    const src_id)
```

Parameters:

Table 18. Message Send Parameters

Type	Parameters	Description
ke_msg_id_t	id	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier

Return: None

Description: Send a message that has a zero length parameter member. No allocation is required as this will be performed internally.

4.2.6.4 Message Forward

Prototype:

```
void
ke_msg_forward(void const *param_ptr, ke_task_id_t const dest_id, ke_task_id_t const src_id)
```

Parameters:

Table 19. Message Send Parameters

Type	Parameters	Description
void const *	param_ptr	Pointer to the parameter member of the message that will be sent
ke_task_id_t	dest_id	Destination Task Identifier
ke_task_id_t	src_id	Source Task Identifier

Return: None

Description: Forward a message to another task by changing its destination and source task IDs.

4.2.6.5 Message Free

Prototype:

```
void ke_msg_free(struct ke_msg const *msg)
```

Parameters:

Table 20. Message Send Parameters

Type	Parameters	Description
struct ke_msg const *	msg	Pointer to the message to be freed
ke_task_id_t	dest_id	Destination Task Identifier
ke_task_id_t	src_id	Source Task Identifier

Return: None

Description: Free allocated message.

4.3 SCHEDULER

4.3.1 Overview

- The scheduler is called in the main loop of the user application using the `Kernel_Schedule()` function.
- In the user application's main loop, the kernel checks if the event field is non-null, and executes the event handlers for which the corresponding event bit is set.

4.3.2 Requirements

4.3.2.1 Scheduling Algorithm

Figure 3 shows how the scheduler handles messages. The message handler pops messages from the message queue, passes them to the pre-defined message handler, and then handles either releasing or saving those messages based on the responses from those handlers.

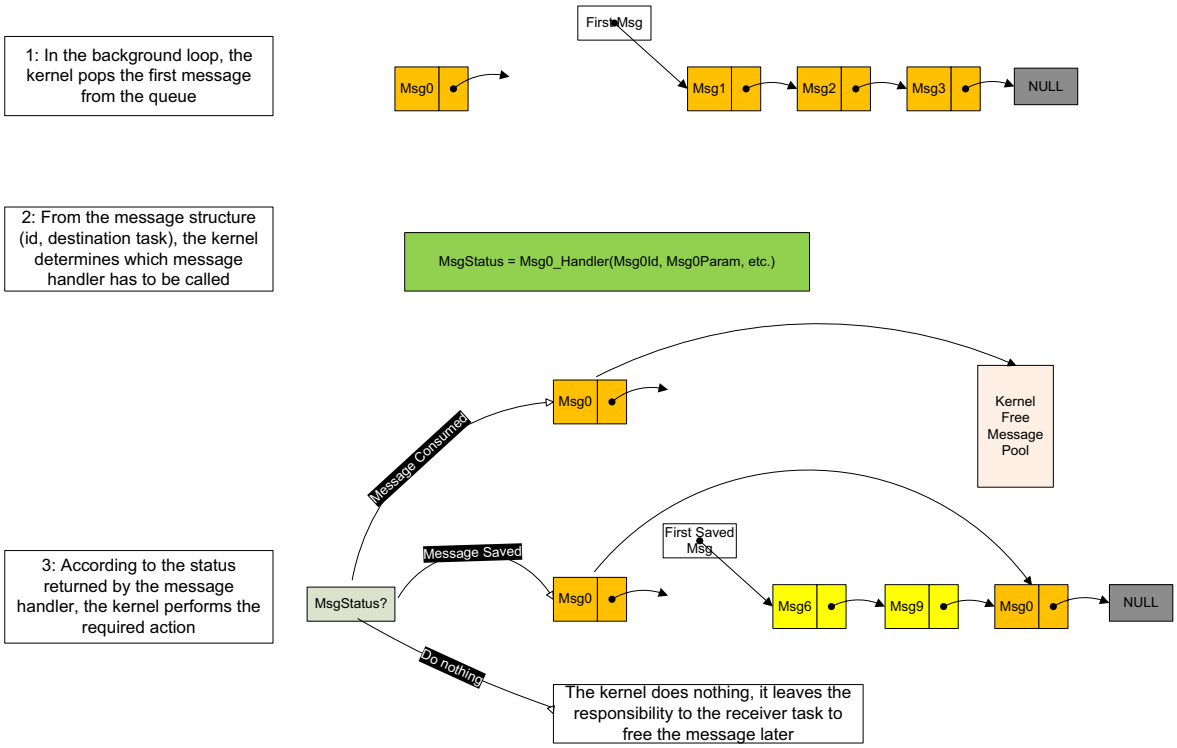


Figure 3. Scheduling Algorithm

4.3.2.2 Save Service

The Save service can SAVE a message, i.e. store it in memory without it being consumed. If the task state changes after a message is received, the scheduler will try to handle the saved message before scheduling any other signals.

4.4 TASKS

4.4.1 Definition

A kernel task is defined by:

- Its task type, i.e. a constant value defined by the kernel, unique for each task
- Its task descriptor, which is a structure containing all the information about the task:
  - The messages that it is able to receive in each of its states
  - The messages that it is able to receive in the default state
  - The number of instances of the task
  - The number of states of the task
  - The current state of each instance of the task

The kernel keeps a pointer to each task descriptor, which is used to handle the scheduling of the messages transmitted from one task to another.

4.5 KERNEL TIMER

4.5.1 Overview

- RW Kernel provides a Time reference (absolute time counter.)
- RW Kernel provides timer services: Start, Stop timer.
- Timers are implemented by means of a reserved queue of delayed messages.
- Timer messages do not have parameters.

4.5.2 Time Definition

Time is defined as duration; the minimum step is 10 ms.

4.5.3 Timer Object

The structure of the timer message contains:

- **`*next`**: Pointer on the next timer
- **`id`**: Message identifier
- **`task`**: Destination task identifier
- **`time`**: Duration

4.5.4 Timer Setting

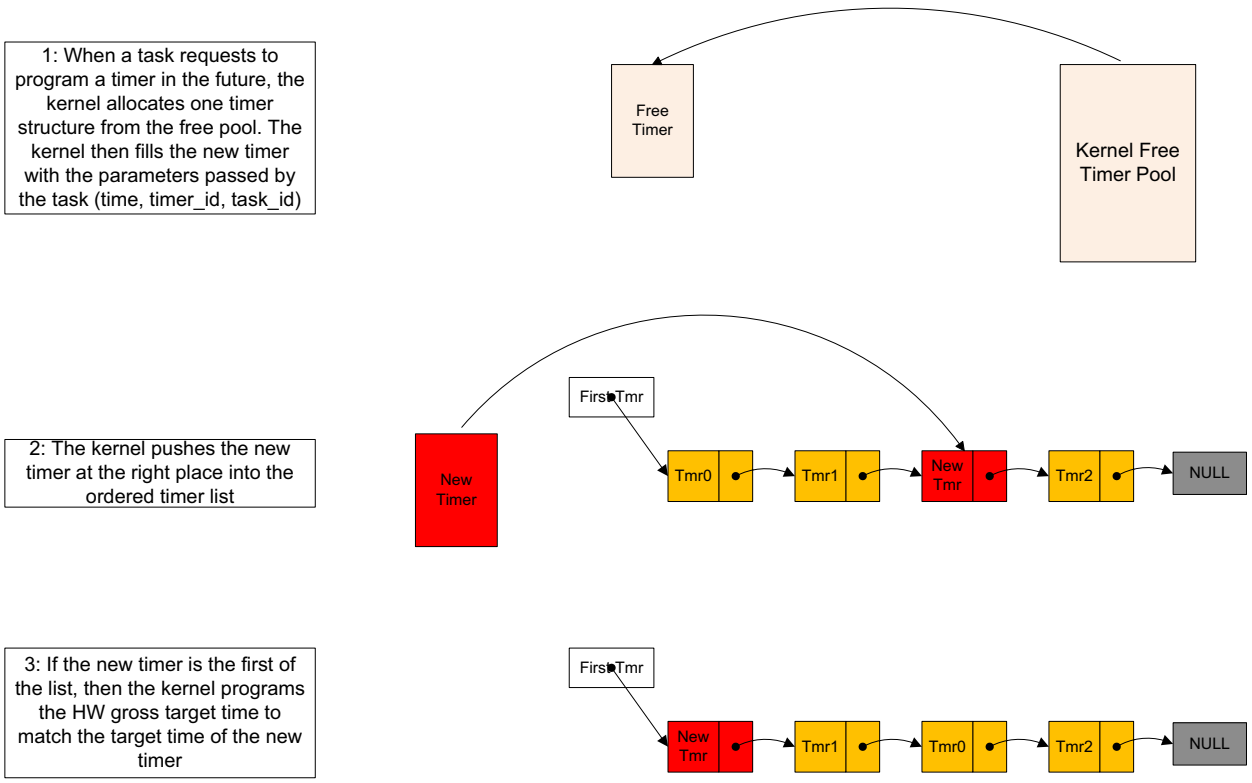


Figure 4. Timer Setting Flow

4.5.5 Time Primitives

4.5.5.1 Timer Set

Start or restart a timer.

Prototype:

```
void ke_timer_set(ke_msg_id_t const timer_id, ke_task_id_t const task, uint32_t const
    delay);
```

Parameters:

Table 21. Timer Set Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task_id`	Task identifier
`uint16_t`	`delay`	Timer duration (multiple of 10 ms)

Return: None

Description: The function first cancels the timer if it exists; then it creates a new one. The timer can be one-shot, or periodic (i.e. it will be automatically set again after each trigger).

4.5.5.2 Timer Clear

Remove a registered timer.

Prototype:

```
void ke_timer_clear(ke_msg_id_t const timer_id, ke_task_id_t const task);
```

Parameters:

Table 22. Timer Set Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task id`	Task identifier

Return: None

Description: This function searches for the timer element identified by its timer and task identifiers. If found, it is stopped and freed, otherwise an error message is returned.

4.5.5.3 Timer Activity

Check if a requested timer is active.

Prototype:

```
bool ke_timer_active(ke_msg_id_t const timer_id, ke_task_id_t const task);
```

Parameters:

Table 23. Timer Set Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task id`	Task identifier

Return: `TRUE` if the timer identified by Timer Id is active for the Task id, `FALSE` otherwise

Description: This function pops the first timer from the timer queue and notifies the appropriate task by sending a kernel message. If the timer is periodic, it is set again; if it is one-shot, the timer is freed. The function also checks the next timers, and processes them if they have expired or are about to expire. Figure 5 shows the process flow for handling expired timers.

4.5.5.4 Timer Expiry

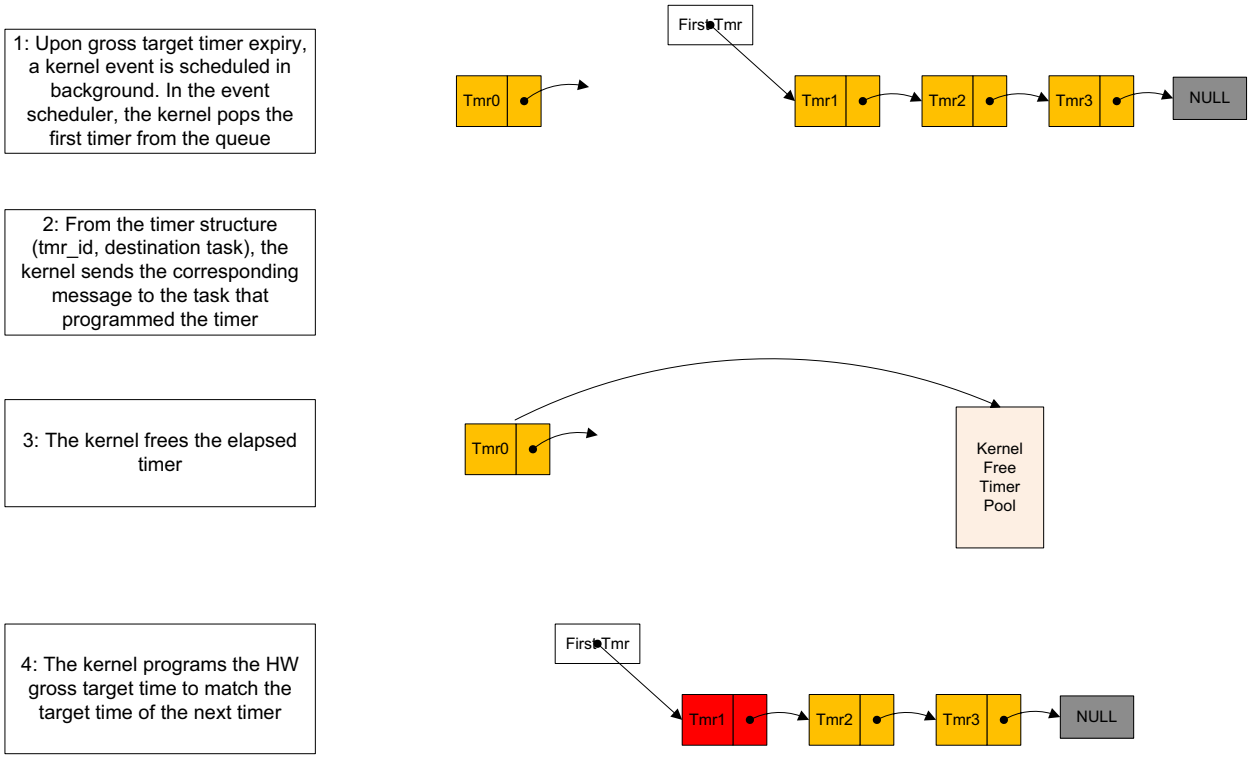


Figure 5. Timer Expiry Flow

4.6 Useful Macros

- Builds the task identifier from the type and the index of that task:

```
#define KE_BUILD_ID(type, index) ( (ke_task_id_t)(((index) << 8)|(type)) )
```

- Retrieves task type from task id:

```
#define KE_TYPE_GET(ke_task_id) ((ke_task_id) & 0xFF)
```

- Retrieves task index number from task id:

```
#define KE_IDX_GET(ke_task_id) (((ke_task_id) >> 8) & 0xFF)
```

CHAPTER 5

Program ROM

5.1 Overview

The goal of the program ROM is to efficiently boot an application or restore an application from sleep to a known state, to handle soft resets cleanly, and to ensure that all applications behave as expected once started, as the underlying portions of the system they depend on are re-initialized.

The Program ROM for the RSL10 SoC is implemented in the ROM at the base of the Arm Cortex-M3 core memory space, and contains the following features:

- A simple vector table as described in Section 5.2, “Vector Table”
- System initialization and re-initialization support as described in Section 5.3, “Initialization Support”
- Application boot and verification as described in Section 5.4, “Application Validation and Boot”
- A function table providing access to the support functions as described in Section 5.5, “Function Table”

5.2 Vector Table

The program ROM contains a minimal vector table located at the base of memory. This vector table is described in Table 24.

Table 24. Program ROM Vector Table

Address	Entry	Description
0x00000000	ROM Stack	Initial value for the Program ROM stack pointer; set to 0x20002000 (top of DRAM 0)
0x00000004	Reset Interrupt Handler	Entry point for the program ROM
0x00000008	NMI Handler	Handler that consists of a spin loop used to capture an unexpected NMI event during the execution of the ROM (waits for watchdog timer to reset the device)
0x0000000C	Hard Fault Handler	Handler that consists of a spin loop used to capture an unexpected fault event during the execution of the ROM (waits for watchdog timer to reset the device)
0x00000010	Memory Fault Handler (promoted to Hard Fault)	
0x00000014	Bus Fault Handler (promoted to Hard Fault)	
0x00000018	Usage Fault Handler (promoted to Hard Fault)	

The Program ROM version (`PROGRAM_ROM_VERSION`) is stored in, and is accessible from, the address immediately following this simple vector table (0x0000001C).

5.3 Initialization Support

The RSL10 program ROM contains three sets of initializations:

1. Base system initialization
2. User-defined system initialization
3. Boot and wakeup initialization

These three initialization sets each perform a specific task to ensure that the system is in a good state for whatever application code or other tasks will follow. A description of each initialization can be found in the following sub-sections.

5.3.1 Base System Initialization

The base system initialization function is used to ensure that key functional elements of the system are in their power-on reset state after a soft reset or other non-wakeup boot. This is generally not required, but provides an option for safely putting the system into a known good state at startup, to ensure proper behavior of everything that executes after this function.

The base system initialization function reconfigures the following components to their default configurations:

- ARM Cortex-M3 processor fault handlers promoted to the hard fault handler
- ARM Cortex-M3 processor external interrupt handlers disabled and all pending interrupts cleared
- LPDSP32 DSP disabled and reset
- All DMA channels disabled
- All digital I/Os disabled (no I/O, weak pull-up resistor enabled)
- Watchdog set to the maximum time out and refreshed
- Clock distribution divisors reset
- Flash timing reset for compatibility with the default clock settings
- Power supply configurations reset
- All memories powered and enabled in normal mode

The base system initialization function is accessible for use in a user application through the program ROM function table, as described in Section 5.5, “Function Table” on page 48.

5.3.2 User-Defined System Initialization

The user-defined system initialization function is used to verify that the manufacturing initialization function included in the device configuration record (described in Section 3.3.3, “Device Configuration Record” on page 26) is valid, and to execute that function if it is. This function is intended to load calibration information for the device to provide a calibrated environment for the application code that follows.

The manufacturing initialization function can be replaced by a user-defined manufacturing initialization function to change the default behavior of this system initialization. The default manufacturing initialization function loads default calibration values for a number of clocks and power supplies from the base of their array of calibrated trim values. Elements that are configured using this function include:

1. The bandgap, which is trimmed using the default trim setting from `MANU_INFO_BANDGAP`.
2. The DC-DC convertor, which is trimmed using the default trim setting from `MANU_INFO_DCDC`.
3. The digital power supplies VDDC and VDDM (including retention trim settings), which are trimmed using the default trim settings from `MANU_INFO_VDDC`, `MANU_INFO_VDDC_STANDBY`, `MANU_INFO_VDDM`, and `MANU_INFO_VDDM_STANDBY`.
4. The RF power supplies VDDRF and VDDPA, which are trimmed using the default trim settings from `MANU_INFO_VDDRF` and `MANU_INFO_VDDPA`.
5. The RC startup oscillator, which is trimmed to the default multiplied configuration using the default trim setting from `MANU_INFO_OSC_RC_MULT` with the flash delay timing parameters configured to match the declared frequency for proper system behavior.
6. The 32 kHz RC oscillator, which is trimmed using the default trim setting from `MANU_INFO_OSC_32K`.

NOTE: Generally, only replace the manufacturing initialization function if you want to boot with calibration targets other than the default targets.

For more information about each of these blocks, see the power supply and clocking chapters of the *RSL10 Hardware Reference*. For default trim targets for each of these power supplies and clock frequencies, refer to the datasheet for RSL10.

NOTE: No power supply or clocking elements are enabled by this initialization routine.

The manufacturing initialization function is considered valid if it:

1. Specifies a length (MANU_INFO_LENGTH) that fits within the device configuration record. No error (SYS_INIT_ERR_NONE) is reported if the 16-bit length value stored at MANU_INFO_LENGTH is set to 0x0000 or 0xFFFF to indicate that there is no manufacturing initialization function. An error code of SYS_INIT_ERR_INVALID_BLOCK_LENGTH is returned if the specified length extends beyond the end of the device configuration record sector.
2. Includes a CRC-CCITT calculated over the MANU_INFO_LENGTH field and a code section of the specified length (in bytes). If the CRC calculation indicates that the CRC value is incorrect, the SYS_INIT_ERR_BAD_CRC error code is returned to indicate this failure.

The manufacturing initialization function uses no parameters, and does not respond with any return code. If execution completes and returns, the user-defined system initialization function reports that no error was encountered using the SYS_INIT_ERR_NONE error code.

The user-defined system initialization function is accessible for use in a user application through the program ROM function table, as described in Section 5.5, “Function Table” on page 48.

5.3.3 Boot and Wakeup Initialization

The RSL10 program ROM starts execution in the reset vector after initial startup, after a soft or core reset, and after returning from a system wakeup event. This boot and wakeup initialization routine is designed to reboot and reconfigure the system as efficiently as possible for a device that quickly needs to return to sleep or another low power state.

Prior to anything else, the ROM reset vector loads the value of ASC_WAKEUP_CTRL, and clears the ACS_WAKEUP_CTRL_BOOT_SELECT bit from this register. The loaded value is used to determine what path is used through the boot and wakeup initialization routine - and this value is cleared after being loaded to ensure that the system defaults back to a regular boot from flash memory in case of a failure during boot.

The loaded value is used for the following initial sequence:

1. If the ACS_WAKEUP_CTRL_BOOT_XTAL_EN bit is set (BOOT_XTAL_ENABLE) and ACS_WAKEUP_CTRL_BOOT_SELECT is cleared (BOOT_ON_FLASH) in the loaded value, the RF block and 48 MHz crystal oscillator are enabled. This enables the system to start this oscillator with a minimum delay, allowing an application running on the RSL10 SoC to remain in sleep mode for a longer duration.

IMPORTANT: Prior to switching to the 48 MHz clock, a user application that is using BOOT_XTAL_ENABLE needs to verify that the XTAL oscillator has completed its initialization by confirming that the RF_REG39_ANALOG_INFO_CLK_DIG_READY bit from the RF_REG39 register is set (ANALOG_INFO_CLK_DIG_READY).

2. The flash is configured for correct behavior upon startup, given the current clock source configuration. The flash itself is not enabled.

3. If the ACS_WAKEUP_CTRL_BOOT_XTAL_EN bit is cleared (BOOT_XTAL_DISABLE) and ACS_WAKEUP_CTRL_BOOT_SELECT is set (BOOT_CUSTOM) in the loaded value, the program ROM attempts to restore an application running from RAM as the system is waking up from a sleep or similar low-power mode with a reboot defined.

   To restore a running application the ROM uses the following wakeup sequence:

   a. Loads the current value of the ACS_WAKEUP_GP_DATA register.
   b. If bit 0 of the ACS_WAKEUP_GP_DATA value is set, this value is written to the SYSCTRL_MEM_POWER_CFG and SYSCTRL_MEM_ACCESS_CFG registers and memory retention is disabled. Otherwise, the ROM indicates a failure and continues with step 4.
   c. Loads the value of the SYSCTRL_WAKEUP_ADDR register.
   d. If the value of SYSCTRL_WAKEUP_ADDR is non-zero, uses this as the wakeup restore address for the application being restored. Otherwise, the ROM indicates a failure and continues with step 4. If restoring, the application finds the following information starting at the specified wakeup restore address:
      i. The debug port lock configuration register (SYSCTRL_DBG_LOCK) register contents.
      ii. The four words of the lock key used with the SYSCTRL_DBG_LOCK_KEY registers to restrict or unrestrict access to the debug port.
      iii. The application's restart address, which indicates the first item executed in the restored user application (this address can be anywhere in memory, but is typically located in RAM or flash). The ROM branches to this restart address at this time.

IMPORTANT: The reset vector does not modify or access the stack prior to determining if the system is exiting a low-power mode. This will ensure that an application returning from these modes can continue from a known or fixed location without needing to reset the contents of the stack.

4. If the value loaded from ASC_WAKEUP_CTRL indicates that the device is rebooting (the ACS_WAKEUP_CTRL_BOOT_FLASH_APP_REBOOT bit is set to BOOT_FLASH_APP_REBOOT_ENABLE), the following initialization sequence is followed:
   a. Clear the ACS_WAKEUP_CTRL_BOOT_SELECT, ACS_WAKEUP_CTRL_BOOT_XTAL_EN, and ACS_WAKEUP_CTRL_BOOT_FLASH_APP_REBOOT bits from the ACS_WAKEUP_CTRL register, to ensure that if the boot fails, the system does not attempt to reboot again.
   b. Enable all of the memories in the RSL10 system, except the DSP_PRAM instances.
   c. Read the memory power configuration from SYS_INFO_START_MEM_CFG in the application specific record (described in Section 3.3.1, "Application Specific Record" on page 25).
   d. If SYS_INFO_START_MEM_CFG would enable the PRAM, flash, and DRAM0 memory instances (bits 0, 1, and 6 are set in the value read), enable the memories specified and disable all memory retention settings. Otherwise, enable all of the memories in the RSL10 system, disabling all memory retention settings.

5. If the program ROM has found that the device is not rebooting, the following initialization sequence is followed:
   a. Enable all of the memories in the RSL10 system, disabling all memory retention settings.
   b. Switch to the RC clock as the source for SYSCLK.
   c. Load the default VCC = 1.25 V trim setting.
   d. Load the calibrated bandgap and VDDM settings from the manufacturing records in NVR4. If either read reports an ECC error, repeat the read using increasing power supply values until no error is reported, and use the last loaded values as the bandgap and VDDM configuration settings.
   e. Once valid bandgap and VDDM settings are found, write them to the ACS_BG_CTRL and ACS_VDDM_CTRL_VDDM_TRIM registers. If no valid trim settings can be loaded from the manufacturing

records, the ROM defaults to a fail-safe, high trim setting for these registers that is used to guarantee that memory access is reliable after this step.

f. Execute the base system initialization function described in Section 5.3.1, “Base System Initialization”.

6. The debug port lock information is loaded using the following sequence:
   a. The value of LOCK_INFO_SETTING setting is loaded from the device configuration record (described in Section 3.3.3, “Device Configuration Record” on page 26)
   b. If an error has been observed, the VDDM setting is set to the nominal value for 1.25 V and this value is reloaded.
   c. The loaded value from LOCK_INFO_SETTING is written to SYSCTRL_DBG_LOCK. If this register is set to DBG_ACCESS_LOCK, debug port accesses remain restricted. If this register is set to any other, full access to the debug port will be available following this step.
   d. The 128-bit debug port unlock key is read from LOCK_INFO_KEY and copied to the four DBG_LOCK_KEY registers. This key will be valid once the last value is written, and can be written over the debug port while access to the debug port remains restricted to clear SYSCTRL_DBG_LOCK and unrestrict the debug port.

7. If the program ROM has found that the device is rebooting, the vector table is set to point to the value indicated by SYS_INFO_START_ADDR from the application specific record (see Section 3.3.1, “Application Specific Record” on page 25) and execution continues from this rebooted application’s reset vector.
8. If the program ROM has found that the device is not rebooting:
   a. The user-defined system initialization function described in Section 5.3.2, “User-Defined System Initialization” is executed.
   b. The application pointed to by SYS_INFO_START_ADDR is verified from the application specific record (see Section 3.3.1, “Application Specific Record”), using the application validation routine defined in Section 5.4, “Application Validation and Boot”. If this is a valid application, execution continues from this application’s reset vector.
   c. The application located at the base of flash memory is verified using the application validation routine defined in Section 5.4, “Application Validation and Boot”. If this is a valid application, execution continues from this application’s reset vector.
   d. If no valid application has been found, the error code from the application verification is written to VAR_BOOTROM_ERROR (located at the base of DRAM0) and the ROM hard fault handler is executed. The hard fault handler continues to execute until a watchdog reset triggers a power-on reset of the RSL10 system.

5.4 Application Validation and Boot

The RSL10 program ROM contains a set of functions that are used to validate and boot applications.

The program ROM only boots an application if the system is not returning from sleep mode. (If the device fails on return from sleep mode, the system triggers a reset which forces a boot.) An application is booted only after all system initialization, as described in Section 5.3, “Initialization Support”, has completed. The program ROM first attempts to boot the application pointed to by SYS_INFO_START_ADDR (see Section 3.3.1, “Application Specific Record” on page 25). If this application fails to boot, the ROM then attempts to boot an application starting from the base of flash as a fail-safe measure. If the application at the base of flash also fails, the ROM records an error code and waits in the hard-fault handler for a watchdog reset to reset the system.

IMPORTANT: If the application at SYS_INFO_START_ADDR cannot be booted, and the subsequent attempt to boot the application at the base of flash fails in any way (including a “Bad CRC” error), any previous error from the attempt to boot the application at SYS_INFO_START_ADDR is overwritten by the failure code for the attempt to boot from the base of flash.

The ROM considers an application valid if it starts with its vector table, and no errors that would prevent boot are detected. Possible errors, and the error codes reported for these errors, are described in Table 25. If neither the application pointed to by SYS_INFO_START_ADDR nor an application located at the base of flash successfully boots, the boot ROM writes this error code to VAR_BOOTROM_ERROR (located at the base of DRAM0).

Table 25. Application Validation

Error	Error Code	Description
None	0x0	No error detected
Bad Alignment	0x1	The Arm Cortex-M3 processor requires that the application's vector table is aligned to a 512-byte boundary in memory, for a device with the number of external interrupts that are included in the RSL10 SoC. The location of the specified application is not at a valid location in memory.
Bad Stack Pointer	0x2	The initial stack pointer must point to a valid memory location on the system bus or to a valid memory location in PRAM or DSP_PRAM on the D-code bus. This requires that the specified stack pointer is 32-bit aligned, and that the next address stack data will be placed at is in DRAM, DSP_DRAM, BB_DRAM, PRAM, or DSP_PRAM (remapped area).
Bad Reset Vector	0x3	The program ROM checks that the reset handler is located immediately after the vector table (or after a CRC located after the vector table). This check is performed indirectly by confirming that the reset vector points to a location that: • Provides space for at least the minimum number of entries in the vector table (a minimum valid vector table contains 4 entries; the stack pointer, reset vector, NMI handler, and hard fault handler) • Provides space for no more than the stack pointer, the 88 potential vectors, and a CRC (maximum of 90 words between the base of the application and the reset vector's location)
Failed to Start the Application	0x6	Indicates that the application has failed to boot or has returned with no identifiable cause.
Bad CRC	0x7	A CRC-CCITT value can be placed between the vector table and the reset handler. The boot validation step validates if a CRC calculated over the vector table matches the value written at this location. NOTE: This error code is considered to be a non-fatal error, since the inclusion of a CRC is optional. The first entry on the application's stack after boot will indicate whether no-error has occurred (0x0) or if a bad CRC has been discovered (0x7).

If the ROM determines that an application should be booted, the ROM:

1. Sets the VTOR bit-field in the Arm Cortex-M3 processor's SCB register to point to the application's vector table
2. Loads the initial stack pointer value from the application's vector table to the Arm Cortex-M3 processor's SP register
3. Pushes the application's status code to the top of the newly defined stack (valid error codes for a booted application are "None" and "Bad CRC" - as described in Table 25)
4. Branches to the beginning of the reset handler, as indicated by the reset vector in the application's vector table

5.5 Function Table

The Program ROM contains the implementation of a set of firmware functions that are exposed through a function table. A list of functions provided by the ROM can be found in Table 26.

Table 26. Function Table Content

Source	Function	Address	Reference
Program ROM	Reset	0x00000020	Section 9.50, “Sys_BootROM_Reset” on page 245
	System Delay	0x0000002C	Section 9.69, “Sys_Delay_ProgramROM” on page 264
	Read NVR4	0x00000058	Section 9.156, “Sys_ReadNVR4” on page 352
Program ROM (Initialization)	Initialize (Base)	0x00000024	Section 9.100, “Sys_Initialize_Base” on page 296; described in Section 5.3.1, “Base System Initialization”.
	Initialize	0x00000028	Section 9.99, “Sys_Initialize” on page 295; described in Section 5.3.2, “User-Defined System Initialization”.
	Get Trim	0x00000054	N/A - Internal function used by `Sys_Clocks_Osc*CalibratedConfig()`, `Sys_Power_*CalibratedConfig()`
	Unlock Debug	0x00000038	Section 9.150, “Sys_ProgramROM_UnlockDebug” on page 346
Program ROM (Boot ROM)	Validate Application	0x00000030	Section 9.54, “Sys_BootROM_ValidateApp” on page 249
	Start Application	0x00000034	Section 9.51, “Sys_BootROM_StartApp” on page 246, Section 9.53, “Sys_BootROM_StrictStartApp” on page 248
Flash Library	Write Word Pair	0x0000003C	Section 11.6, “Flash_WriteWordPair” on page 393
	Write Buffer	0x00000040	Section 11.3, “Flash_WriteBuffer” on page 390
	Erase Sector	0x00000044	Section 11.2, “Flash_EraseSector” on page 389
	Erase All	0x00000048	Section 11.1, “Flash_EraseAll” on page 388
	Write Command	0x0000004C	Section 11.4, “Flash_WriteCommand” on page 391
	Write Interface Control	0x00000050	Section 11.5, “Flash_WriteInterfaceControl” on page 392

All program ROM functions are exposed through the system library’s ROM vector support (*rsl10_romvect.h*). For more information on these functions and the rest of the system library, see Chapter 9, “System Library Reference” on page 195.

All flash library functions are exposed through the flash library’s ROM implementation (*rsl10_flash_rom.h*). For more information on the flash library, see Chapter 11, “Flash Library Reference” on page 388.

IMPORTANT: We recommend that all functions provided by the flash library be executed from RAM or ROM, as executing them from flash can result in hidden, flash-access-related failures. As such, the flash library is provided as part of the ROM to allow erasing and writing to the flash without having to instantiate the flash library functions in RAM.

CHAPTER 6

Bluetooth Stack and Profiles

6.1 INTRODUCTION

This chapter explains how the Bluetooth stack, including the HCI, GATT and GAP, is implemented for RSL10. This chapter also provides a description of the Bluetooth profile libraries that are provided with the RSL10 system to support standard use cases.

IMPORTANT: The default Bluetooth stack is built to support four Bluetooth links, as a balance between system flexibility and power/memory optimization. If your user application requires more links than the default Bluetooth stack library build supports, contact your ON Semiconductor Customer Service Representative for assistance.

6.1.1 Include and Object Files

In the include folder of the RSL10 installation directory, rsl10_bb.h and rsl10_ble.h list all the Bluetooth low energy technology and Baseband support header files — refer to Table 27. The object files are described in Table 28 on page 51 and Table 29 on page 53.

Table 27. RSL10 Bluetooth Low Energy and Baseband Support Files

Bluetooth	**Baseband**	
rsl10_ble.h	**rsl10_bb.h**	
#include <ble\rwble_hl_config.h>	#include <bb\rwble_config.h>	#include <bb\co_math.h>
#include <ble\rwble_hl_error.h>	#include <bb\rwble.h>	#include <bb\co_utils.h>
#include <ble\rwble_hl.h>	#include <bb\rwip.h>	#include <bb\dbg.h>
#include <ble\rwprf_config.h>	#include <bb\rwip_config.h>	#include <bb\dbg_task.h>
#include <ble\prf.h>	#include <bb_reg_ble_em_cs.h>	#include <bb\dbg_swdiag.h>
#include <ble\ahi.h>	#include <bb_reg_ble_em_ral.h>	#include <bb\dbg_mwsgen.h>
#include <ble\ahi_task.h>	#include <bb_reg_ble_em_rx_buffer.h>	#include <bb\ea.h>
#include <ble\att.h>	#include <bb_reg_ble_em_rx_desc.h>	#include <bb\em_buf.h>
#include <ble\attc.h>	#include <bb_reg_ble_em_tx_buffer_cntl.h>	#include <bb\em_map.h>
#include <ble\attm.h>	#include <bb_reg_ble_em_tx_buffer_data.h>	#include <bb\em_map_ble.h>
#include <ble\attm_db.h>	#include <bb_reg_ble_em_tx_buffer_data.h>	#include <bb\ll.h>
#include <ble\atts.h>	#include <bb_reg_ble_em_tx_desc.h>	#include <bb\llc.h>
#include <ble\ecc_p256.h>	#include <bb_reg_ble_em_wpb.h>	#include <bb\llc_ch_asses.h>
#include <ble\gap.h>	#include <bb_reg_ble_em_wpv.h>	#include <bb\llc_llcp.h>
#include <ble\gapc.h>	#include <bb\reg_blecore.h>	#include <bb\llc_task.h>
#include <ble\gapm_int.h>	#include <bb\reg_access.h>	#include <bb\llc_util.h>
#include <ble\gapc_task.h>	#include <bb\reg_assert_mgr.h>	#include <bb\lld.h>
#include <ble\gapm.h>	#include <bb\reg_common_em_et.h>	#include <bb\lld_pdu.h>
#include <ble\gapm_task.h>	#include <bb\reg_ble_em_cs.h>	#include <bb\lld_wlcoex.h>

Table 27. RSL10 Bluetooth Low Energy and Baseband Support Files (Continued)

Bluetooth	Baseband	
rsl10_ble.h	**rsl10_bb.h**	
#include <ble\gapm_util.h>	#include <bb\reg_ble_em_ral.h>	#include <bb\lld_evt.h>
#include <ble\gatt.h	#include <bb\reg_ble_em_rx_buffer.h>	#include <bb\lld_sleep.h>
#include <ble\gattc.h>	#include <bb\reg_ble_em_rx_desc.h>	#include <bb\lld_util.h>
#include <ble\gattc_task.h>	#include <bb\reg_ble_em_tx_buffer_cntl.h>	#include <bb\llm.h>
#include <ble\gattm.h>	#include <bb\reg_ble_em_tx_buffer_data.h>	#include <bb\llm_task.h>
#include <ble\gattm_task.h>	#include <bb\reg_ble_em_tx_buffer_data.h>	#include <bb\llm_util.h>
#include <ble\h4tl.h>	#include <bb\reg_ble_em_tx_desc.h>	#include <bb\rf.h>
#include <ble\hci.h>	#include <bb\reg_ble_em_wpb.h>	#include <bb\rwip_task.h>
#include <ble\l2cc.h>	#include <bb\reg_ble_em_wpv.h>	
#include <ble\l2cc_pdu.h>	#include <bb\compiler.h>	
#include <ble\l2cc_task.h>	#include <bb\arch.h>	
#include <ble\l2cm.h>	#include <bb\co_bt.h>	
#include <ble\smp_common.h>	#include <bb\co_bt_defines.h>	
#include <ble\smpc.h>	#include <bb\co_endian.h>	
#include <ble\smpc_api.h>	#include <bb\co_error.h>	
#include <ble\smpc_crypto.h>	#include <bb\co_hci.h>	
#include <ble\smpc_util.h>	#include <bb\co_list.h>	
#include <ble\smpm_api.h>	#include <bb\co_llcp.h>	
#include <ble\prf_types.h>	#include <bb\co_lmp.h>	
#include <ble\prf_utils.h>	#include <bb\co_version.h>	

Table 28. Bluetooth GATT-Based Profile and Service Object Files

Profile Name	Profile	Profile Library Name	Profile Description
Alert Notification Profile	ANP	libanpc	This profile enables a client device to receive different types of alerts and event information, as well as information on the count of new alerts and unread items, which exist in the server device.
Alert Notification Service	ANS	libanps	Alert Notification service exposes: The different types of alerts with the short text messages, The count of new alert messages, The count of unread alerts.
Battery Service	BAS	libbasc libbass	The Battery Service exposes the state of a battery within a device.
Blood Pressure Profile	BLP	libblpc	This profile enables a device to connect and interact with a Blood Pressure Sensor device for use in consumer and professional health care applications.
Blood Pressure Service	BLS	libblps	This service exposes blood pressure and other data from a blood pressure monitor for use in consumer and professional healthcare applications.

Table 28. Bluetooth GATT-Based Profile and Service Object Files (Continued)

Profile Name	Profile	Profile Library Name	Profile Description
Cycling Power Profile	CPP	libcppc	This profile enables a Collector device to connect and interact with a Cycling Power Sensor for use in sports and fitness applications.
Cycling Power Service	CPS	libcpps	This service exposes power- and force-related data and optionally speed- and cadence-related data from a Cycling Power sensor intended for sports and fitness applications.
Cycling Speed and Cadence Profile	CSCP	libcscpc	This profile enables a Collector device to connect and interact with a Cycling Speed and Cadence Sensor for use in sports and fitness applications.
Cycling Speed and Cadence Service	CSCS	libcscps	This service exposes speed-related and cadence-related data from a Cycling Speed and Cadence sensor intended for fitness applications.
Current Time Service	CTS	libtipc	This Bluetooth® service defines how the current time can be exposed using the Generic Attribute Profile (GATT).
Device Information Service	DIS	libdisc libdiss	This service exposes manufacturer and/or vendor information about a device.
Find me Profile	FMP	libfindl libfindt	The Find Me profile defines the behavior when a button is pressed on one device to cause an alerting signal on a peer device.
Glucose Profile	GLP	libglpc	This profile enables a device to connect and interact with a glucose sensor for use in consumer healthcare applications.
Glucose Service	GLS	libglps	This service exposes glucose and other data from a personal glucose sensor for use in consumer healthcare applications.
HID over GATT Profile	HOGP	libhogpd libhogpbh libhogprh	This profile defines how a device with Bluetooth low energy wireless communications can support HID services over the Bluetooth low energy protocol stack using the Generic Attribute Profile.
Heart Rate Profile	HRP	libhrpc	This profile enables a Collector device to connect and interact with a Heart Rate Sensor for use in fitness applications.
Heart Rate Service	HRS	libhrps	This service exposes heart rate and other data from a Heart Rate Sensor intended for fitness applications.
Health Thermometer Profile	HTP	libhtpc	This profile enables a Collector device to connect and interact with a Thermometer sensor for use in healthcare applications.
Health Thermometer Service	HTS	libhtpt	This service exposes temperature and other data from a Thermometer intended for healthcare and fitness applications.
Immediate Alert Service	IAS	libproxm libproxr	This service exposes a control point to allow a peer device to cause the device to immediately alert.
Link Loss Service	LLS	libproxm libproxr	This service defines behavior when a link is lost between two devices.
Location and Navigation Profile	LNP	liblanc	This profile enables a Collector device to connect and interact with a Location and Navigation Sensor for use in outdoor activity applications.
Location and Navigation Service	LNS	liblans	This service exposes location and navigation-related data from a Location and Navigation sensor intended for outdoor activity applications.
Next DST Change Service	NDCS	libtipc libtips	This service defines how the information about an upcoming DST change can be exposed using the Generic Attribute Profile (GATT)
Phone Alert Status Profile	PASP	libpaspc	This profile enables a PUID device to alert its user about the alert status of a phone connected to the PUID device.

Table 28. Bluetooth GATT-Based Profile and Service Object Files (Continued)

Profile Name	Profile	Profile Library Name	Profile Description
Phone Alert Status Service	PASS	libpasps	This service exposes the phone alert status when in a connection.
Proximity Profile	PXP	libproxm libproxr	The Proximity profile enables proximity monitoring between two devices.
Running Speed and Cadence Profile	RSCP	librscpc	This profile enables a Collector device to connect and interact with a Running Speed and Cadence Sensor for use in sports and fitness applications.
Running Speed and Cadence Service	RSCS	librscps	This service exposes speed, cadence and other data from a Running Speed and Cadence sensor intended for fitness applications.
Reference Time Update Service	RTUS	libtipc libtips	This service defines how a client can request an update from a reference time source from a time server using the Generic Attribute Profile (GATT).
Scan Parameters Profile	SCPP	libscppc	This profile defines how a Scan Client device with Bluetooth low energy wireless communications can write its scanning behavior to a Scan Server, and how a Scan Server can request updates of a Scan Client scanning behavior.
Scan Parameters Service	SCPS	libscpps	This service enables a GATT Client to store the LE scan parameters it is using on a GATT Server device so that the GATT Server can utilize the information to adjust behavior to optimize power consumption and/or reconnection latency.
Time Profile	TIP	libtipc libtips	The Time profile enables the device to get the date, time, time zone, and DST information and control the functions related the time.

Table 29. Wireless Power Transfer Profiles

Profile Name	Profile	Profile Library Name	Profile Description
Wireless Power Transfer Profile	WPTC WPTS	libwptc libwpts	This profile implements the Alliance for Wireless Power (A4WP) wireless power transfer system for transferring power from a single Power Transmitter Unit (PTU) to one or more Power Receiver Units (PRUs).

All of the individual profile libraries use the Bluetooth low energy stack through the profile's specified interfaces. These interfaces are documented in the interface specifications. Because the Bluetooth low energy stack itself requires a reciprocal link in order to find all of the profile components, the stack library has been built with an object factory that instantiates calls to each of the profiles. If a profile is used by an application, the Bluetooth stack should use the specified profile library. If a profile is not used by an application, an empty templated version of the necessary functions that the Bluetooth stack's factory is looking for is provided by the weak profile (*weakprf.a*) library.

CAUTION: The weak profile library must be the last library linked into an application, to prevent the weakly-defined function definitions provided by this library from overriding the complete function definitions provided by the individual profile libraries.

6.1.2 Bluetooth Stack

The RSL10 device supports a Bluetooth stack through a combination of hardware and firmware resources. The hardware components of the Bluetooth stack are described in the *RSL10 Hardware Reference*. The firmware components of the Bluetooth stack are accessible through a Bluetooth library and associated header files.

The Bluetooth stack is accessible at three layers:

- The Host-Controller Interface (HCI)
- The Generic Attribute Protocol (GATT)
- The Generic Access Profile (GAP)

Table 30 describes the Bluetooth stacks provided with RSL10 and their associated object files.

Table 30. Bluetooth Stack and Kernel Object Files

Kernel and Stack Type	Library Name	Description
Full-featured	Release\libblelib Release\libkelib	These libraries can be used by any application that wants to implement a full stack of Bluetooth 5 with all RSL10 supported features. This stack version supports up to four instances of the GAP state machine (four peripheral or central devices), supporting transmission of a maximum of eight packets per connection interval.
Full-featured with HCI support	Release_HCI\libblelib Release_HCI\libkelib	These libraries can be used when an HCI interface over UART or an external application over UART implementation is required with RSL10 supported features of Bluetooth 5.
Light	Release_Light\libblelib Release_Light\libkelib	These libraries can be used by an application that wants to implement a light version of the Bluetooth 5 stack with these limitations: • No client role • No central role • Only one peripheral device • No coexistence support • A maximum of three packets per connection interval The profile-based database and environment will not exceed 1.2 KB.

6.1.3 Stack Support Functions

The Bluetooth stack library includes a set of support functions that augment the stack firmware, as described in Table 31. All other stack APIs are described in their reference documentation, with support for specific Bluetooth layers described in the following documents:

GAP *RW-BLE-GAP-IS_2mbps.pdf*

GATT *RW-BLE-GATT-IS.pdf*

L2CAP *RW-BLE-L2C-IS.pdf*

Profiles *RW-BLE-PRF-*-IS.pdf*

Host (errors) *RW-BLE-HOST-ERR-CODE-IS.pdf*

Table 31. Bluetooth Stack Support Functions

Function	Description	Reference
BLE_Init	Initialize the Bluetooth stack for use within an application.	6.1.3.1 on p. 55
BLE_InitNoTL	Initialize the Bluetooth stack for use within an application.	6.1.3.2 on p. 55
BLE_Reset	Reset the underlying Bluetooth hardware.	6.1.3.3 on p. 56
BLE_Power_Mode_Enter	Safely enter into a non-running power mode, maintaining the existing Bluetooth stack state and adhering to required Bluetooth low energy timing.	6.1.3.4 on p. 56

6.1.3.1 BLE_Init

Initialize the Bluetooth stack for use within an application.

Type	Function
Include File	`#include <rsl10_bb.h>` `#include <rsl10_ble.h>`
Template	`void BLE_Init(uint32_t error)`
Description	Initialize the Bluetooth stack for use within an application. If the stack is being re-initialized due to an error, raise a message to the local Bluetooth host to help ensure that error recovery is handled well. NOTE: This initialization function also initializes the custom application host interface (AHI) and the related required transport layer hooks. If not using this interface, use `BLE_InitNoTL`() (see Section 6.1.3.2, “BLE_InitNoTL”).
Inputs	error = Indicate why the Bluetooth stack is being re-initialized; should be set to 0 (RESET_NO_ERROR) if no error had occurred, and otherwise should use one of the defined reset errors from the *bb\arch.h* include file.
Outputs	None
Assumptions	None

Example	`/* Initialize the kernel and Bluetooth stack */`
	`Kernel_Init(0);`
	`BLE_Init(0);`
	`BLE_Reset();`

6.1.3.2 BLE_InitNoTL

Initialize the Bluetooth stack for use within an application.

Type	Function
Include File	`#include <rsl10_bb.h>` `#include <rsl10_ble.h>`
Template	`void BLE_InitNoTL(uint32_t error)`
Description	Initialize the Bluetooth stack for use within an application. If the stack is being re-initialized due to an error, raise a message to the local Bluetooth host to help ensure that error recovery is handled well.
Inputs	error = Indicate why the Bluetooth stack is being re-initialized; should be set to 0 (RESET_NO_ERROR) if no error had occurred, and otherwise should use one of the defined reset errors from the *bb\arch.h* include file.
Outputs	None

Assumptions	None

Example	`/* Initialize the kernel and Bluetooth stack */`
	`Kernel_Init(0);`
	`BLE_InitNoTL(0);`
	`BLE_Reset();`

6.1.3.3 BLE_Reset

Reset the underlying Bluetooth hardware.

Type	Function
Include File	`#include <rsl10_bb.h>` `#include <rsl10_ble.h>`
Template	`void BLE_Reset(void)`
Description	Reset the underlying Bluetooth hardware. Ensures that the Bluetooth stack firmware is synchronized to the hardware.
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Initialize the kernel and Bluetooth stack */`
	`Kernel_Init(0);`
	`BLE_Init(0);`
	`BLE_Reset();`

6.1.3.4 BLE_Power_Mode_Enter

Safely enter into a non-running power mode, maintaining the existing Bluetooth stack state and adhering to required Bluetooth low energy timing.

Type	Function	
Include File	`#include <rsl10_bb.h>` `#include <rsl10_ble.h>`	
Template	`bool BLE_Power_Mode_Enter(void *power_mode_env, uint8_t power_mode)`	
Description	Safely enter into a non-running power mode, maintaining the existing Bluetooth stack state and adhering to required Bluetooth low energy timing.	
Inputs	power_mode_env power_mode	= Parameters and configurations for the desired power mode = Desired power mode; use POWER_MODE_[SLEEP | STANDBY]
Outputs	return value	= Returns true if it was safe to switch to the desired power mode, false otherwise.

Assumptions	None

Example	/* Main application loop:
	 * - Run the kernel scheduler
	 * - Refresh the watchdog, switch to sleep mode until the next event
	 * should occur, and wait for an interrupt after waking up before
	 * continuing */
	while (1)
	{
	    Kernel_Schedule();
	
	    /* Refresh the watchdog timer */
	    Sys_Watchdog_Refresh();
	
	    /* Sleep until the next event needs to be processed */
	    GLOBAL_INT_DISABLE();
	    BLE_Power_Mode_Enter(&sleep_mode_env, POWER_MODE_SLEEP);
	    GLOBAL_INT_RESTORE();
	
	    /* Wait for an event before executing the scheduler again */
	    SYS_WAIT_FOR_EVENT;
	
	}

6.2 HCI

The role of the HCI is to provide a uniform interface method of accessing a Bluetooth low energy controller's capabilities from the host. The HCI layer is part of the Bluetooth low energy 4.2 protocol stack, as shown in Figure 6 on page 57.

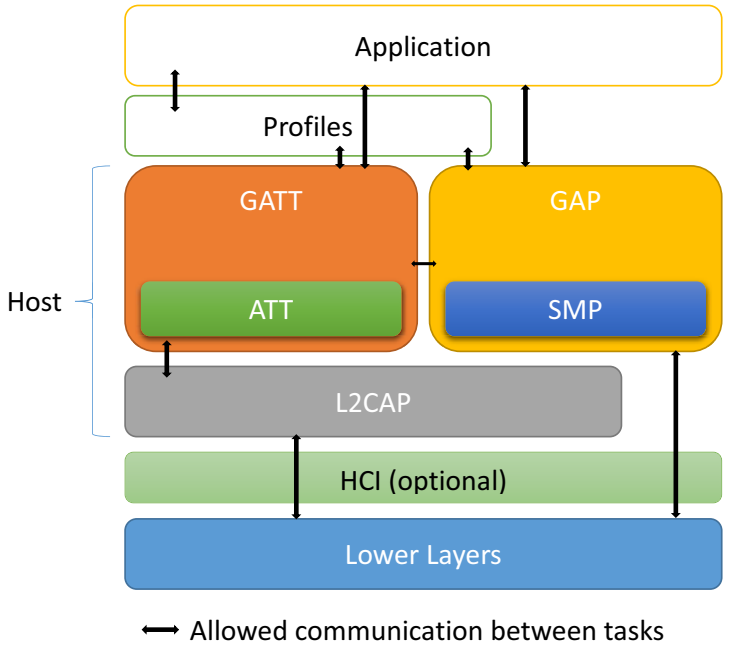


Figure 6. Bluetooth Low Energy Protocol Stack

The HCI layer can be included in three kinds of systems: a full stack system, a host, or a controller. The full stack system contains both host and controller layers. In this case, the role of the HCI is to convey the information from one part to the other by following the rules defined in the HCI standard. For a host or controller only system, the HCI will need to interface with a transport layer that manages the reception and transmission of messages over a physical interface, such as USB or UART.

As shown in Figure 7 on page 58, the two main configurations are supported by the HCI software.

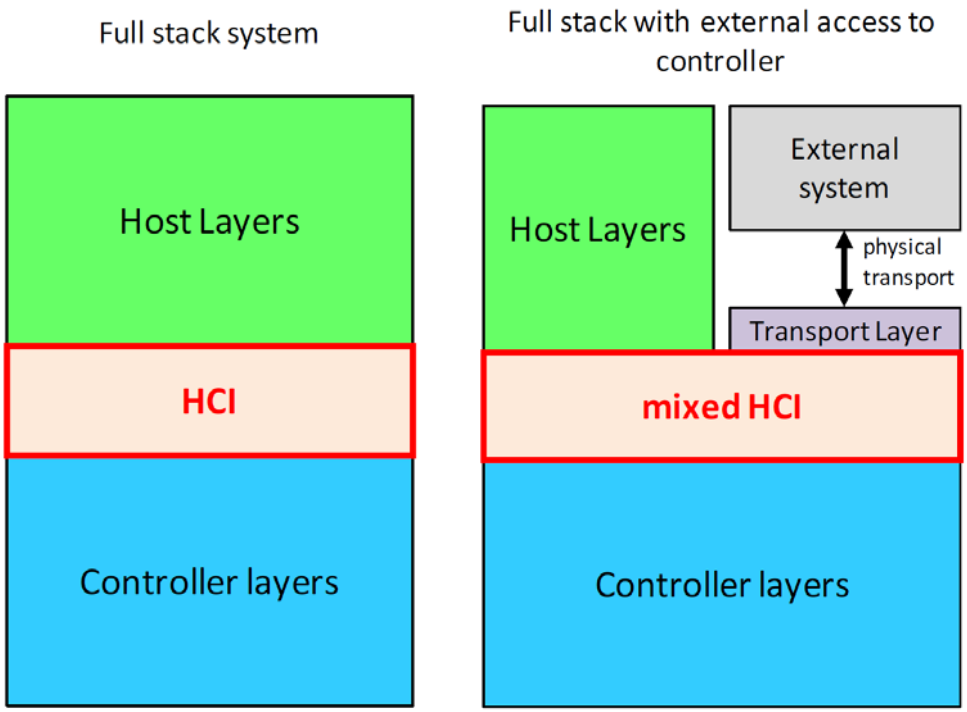


Figure 7. HCI Working Modes

The HCI software supports the two working modes as illustrated in Figure 7. RSL10 has both a full stack system, and compatibility with an external system.

6.2.1 HCI Software Architecture

The HCI software is an interface communication block (depicted in Figure 8 on page 59) that can be used for 3 main purposes:

1. Communication between internal host and external controller
2. Communication between internal controller and external host
3. Communication between internal controller and internal host

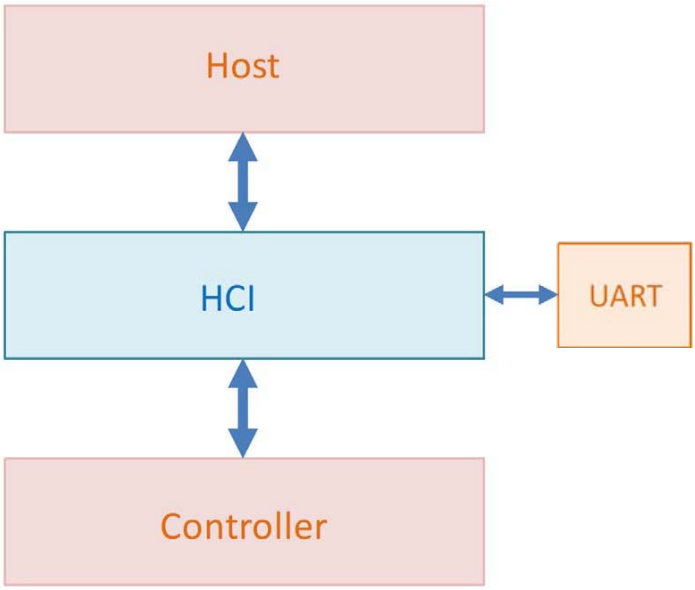


Figure 8. HCI Software Interfaces

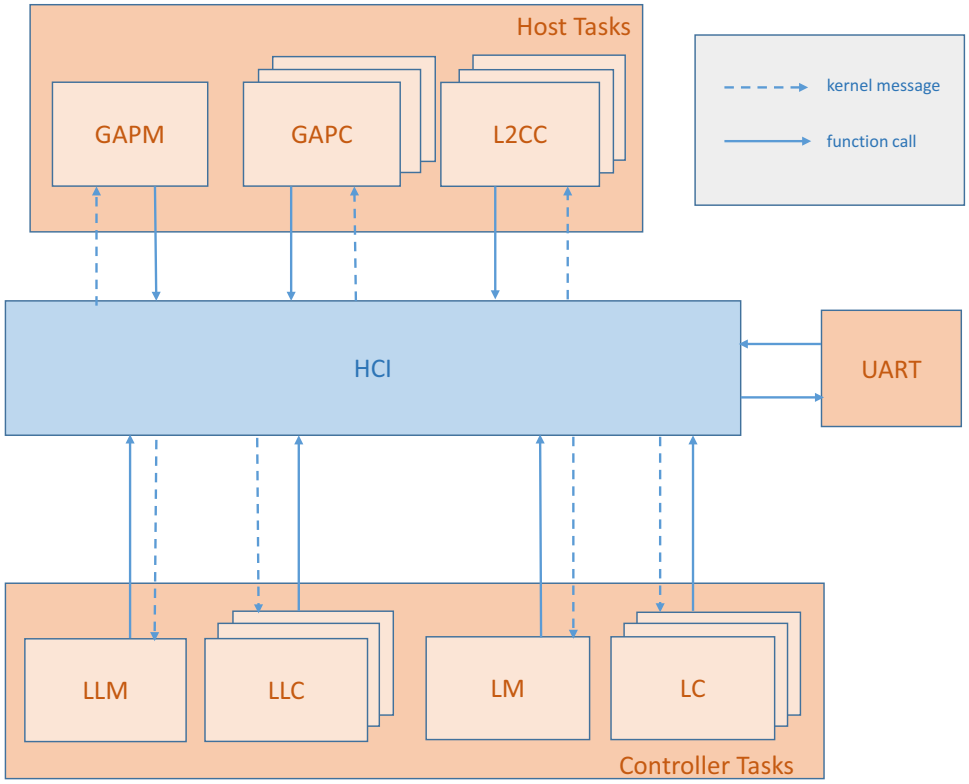


Figure 9. HCI Software Architecture

As described in Figure 9 on page 59, the HCI provides two main processing blocks for RSL10: routing, and external interface packet management. When both host and controller stack parts are present (full stack mode), the external interface feature is optional and the routing system auto-detects whether the lower layers are used by the internal or the external host. The first main challenge of the HCI software is to route the messages to/from the internal task and to/from the external interface. Several types of messages are used to carry the information. These messages might carry some basic control information for the Bluetooth low energy technology operations, in which case they will be conveyed to the main management tasks (LLM/GAPM - Link Layer Manager/ Generic Access Profile Manager). But the messages can also carry link dedicated information, and in that case they will need to be conveyed to the link specific tasks (LLC/GAPC/L2CC - Link Layer Controller/ Generic Access Profile Controller/ Logical Link Protocol).

For RSL10, the communication blocks do not have communication with external controllers; however, the capability of communication with an external host is possible. During reception from the external interface, the HCI also manages the buffer allocation with the kernel memory heap. This is so that after unpacking, an internal kernel message is ready for processing by an internal task.

The huge number of control messages means that the HCI software defines descriptor tables, so that each message's descriptor is referred to during processing. The data packets need a specific buffer allocation policy managed by the IP software. More details are given in the following sections.

6.2.1.1 HCI Control Messages Descriptors

Each HCI command is associated with a command descriptor. The command descriptor is a structure (illustrated in Figure 10) that contains complete information to:

- Route the message or its response within internal stack tasks
- Manipulate the message's parameters or return parameters when dealing with an external interface (only if an external interface is supported)

Table 32 describes each of the fields in the command descriptor.

	Routing		Packing/Unpacking				
2 bytes	1 byte				1 byte	4 bytes	4 bytes
	3:0	5:4	6	7			
Opcode	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT

Figure 10. HCI Commands Descriptor Format

Table 32. Command Descriptor Field Definitions

Field Name	Sub-field Name	Size	Description
Opcode		2 bytes	Command opcode

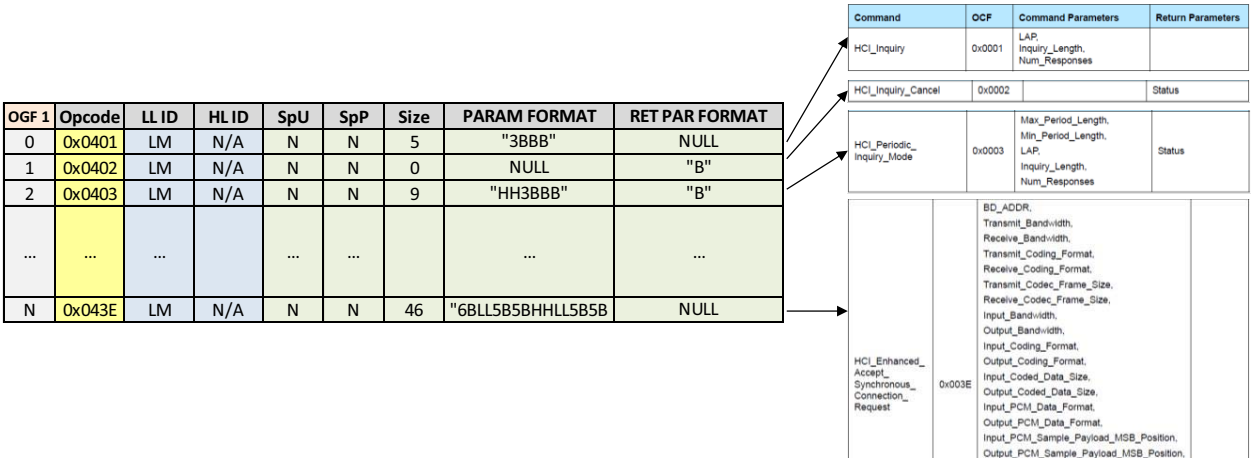
Table 32. Command Descriptor Field Definitions (Continued)

Field Name	Sub-field Name	Size	Description
Destination ID	LL ID	4 bits	Identifier of the task that receives the command
	HL ID	2 bits	Identifier of the task that receives the response event (Command Complete or Command Status)
	Special Return Params packing	1 bit	Flag indicating that the return parameters are packed/unpacked via a special function
	Special Params packing	1 bit	Flag indicating that the parameters are packed/unpacked via a special function
Maximum parameters size		1 bytes	Maximum size of the command parameters
Parameters Format		4 bytes	String representing the parameters format (NULL if no parameter), used by the generic parameter unpacker. In case of special parameter unpacking, this field points to the dedicated unpacker function.
Return Parameters Format		4 bytes	String representing the return parameters format (NULL if no parameter), used by the generic parameter packer. In case of special return parameter packing, this field points to the dedicated packer function.

IMPORTANT: For standard commands, all fields used for parameter packing or unpacking relate directly to the Bluetooth standard specification.

The fields for parameter packing or unpacking are present only if an external interface is supported. In a full stack system that does not support an external interface, only the routing fields are present in the command descriptors. The command group (OGF) allows classifying the descriptors in separate tables.

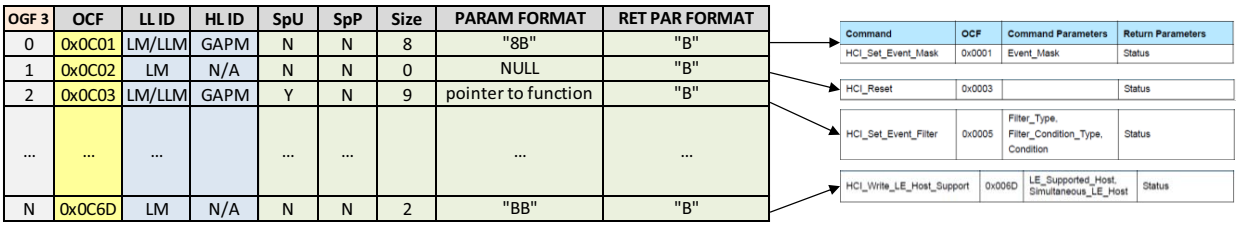
For instance, Figure 11 on page 62 shows some examples of HCI command descriptors within the Link Control commands group.



OGF 1	Opcode	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0	0x0401	LM	N/A	N	N	5	"3BBB"	NULL
1	0x0402	LM	N/A	N	N	0	NULL	"B"
2	0x0403	LM	N/A	N	N	9	"HH3BBB"	"B"
...	...	...		...	...		...	...
N	0x043E	LM	N/A	N	N	46	"6BLL5B5BHHLL5B5B	NULL

Figure 11. Link Control Group Descriptor Table

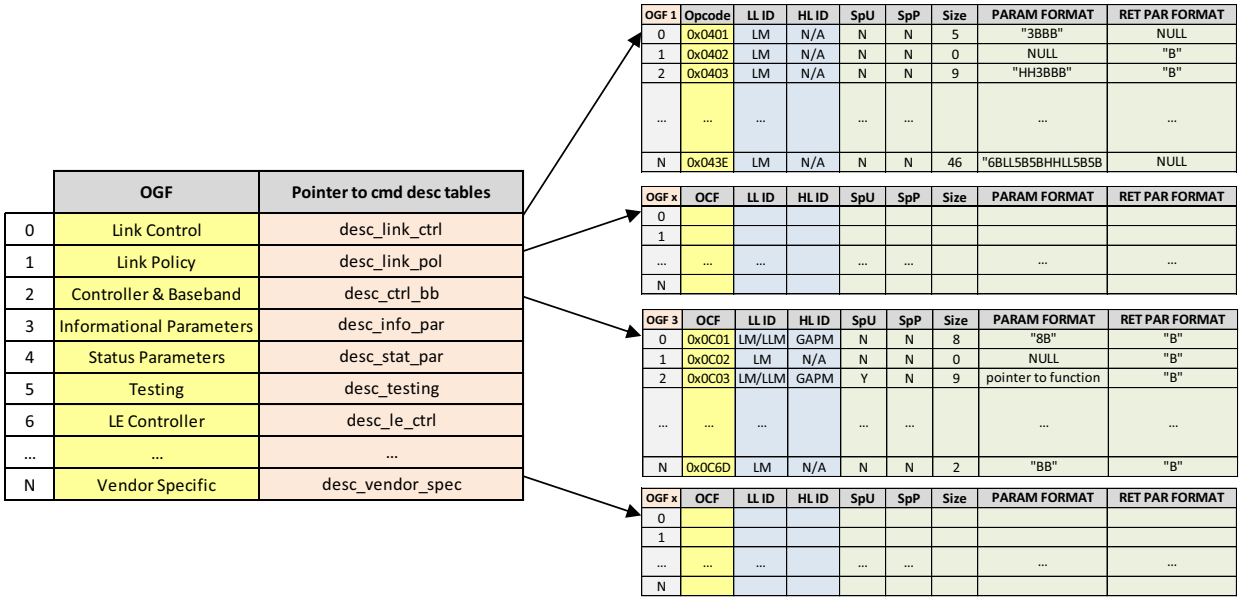
Another example of HCI command descriptors within the controller and baseband command group is shown in Figure 12.



OGF 3	OCF	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0	0x0C01	LM/LLM	GAPM	N	N	8	"8B"	"B"
1	0x0C02	LM	N/A	N	N	0	NULL	"B"
2	0x0C03	LM/LLM	GAPM	Y	N	9	pointer to function	"B"
...	...	...		...	...		...	...
N	0x0C6D	LM	N/A	N	N	2	"BB"	"B"

Figure 12. Controller and Baseband Group Descriptor Table

At any time, the HCI software can obtain a descriptor associated with a command by using a unique common table referencing all the groups present in the HCI software, as shown in Figure 13 on page 63.



	OGF	Pointer to cmd desc tables
0	Link Control	desc_link_ctrl
1	Link Policy	desc_link_pol
2	Controller & Baseband	desc_ctrl_bb
3	Informational Parameters	desc_info_par
4	Status Parameters	desc_stat_par
5	Testing	desc_testing
6	LE Controller	desc_le_ctrl
...	...	...
N	Vendor Specific	desc_vendor_spec

OGF 1	Opcode	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0	0x0401	LM	N/A	N	N	5	"3BBB"	NULL
1	0x0402	LM	N/A	N	N	0	NULL	"B"
2	0x0403	LM	N/A	N	N	9	"HH3BBB"	"B"
...	...	...		...	...		...	...
N	0x043E	LM	N/A	N	N	46	"6BLL5B5BHHLL5B5B	NULL

OGF x	OCF	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0								
1								
...	...	...		...	...		...	...
N								

OGF 3	OCF	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0	0x0C01	LM/LLM	GAPM	N	N	8	"8B"	"B"
1	0x0C02	LM	N/A	N	N	0	NULL	"B"
2	0x0C03	LM/LLM	GAPM	Y	N	9	pointer to function	"B"
...	...	...		...	...		...	...
N	0x0C6D	LM	N/A	N	N	2	"BB"	"B"

OGF x	OCF	LL ID	HL ID	SpU	SpP	Size	PARAM FORMAT	RET PAR FORMAT
0								
1								
...	...	...		...	...		...	...
N								

Figure 13. Top Level Table Pointing to Group Descriptor Tables

6.2.1.2 Event Descriptors

Each HCI event is associated with an event descriptor. The event descriptor is a structure (illustrated in Figure 14) that contains complete information that could be used for:

- Routing the message within internal stack tasks
- Manipulating the message's parameters when dealing with an external interface

Table 33 describes each of the fields in the event descriptor.

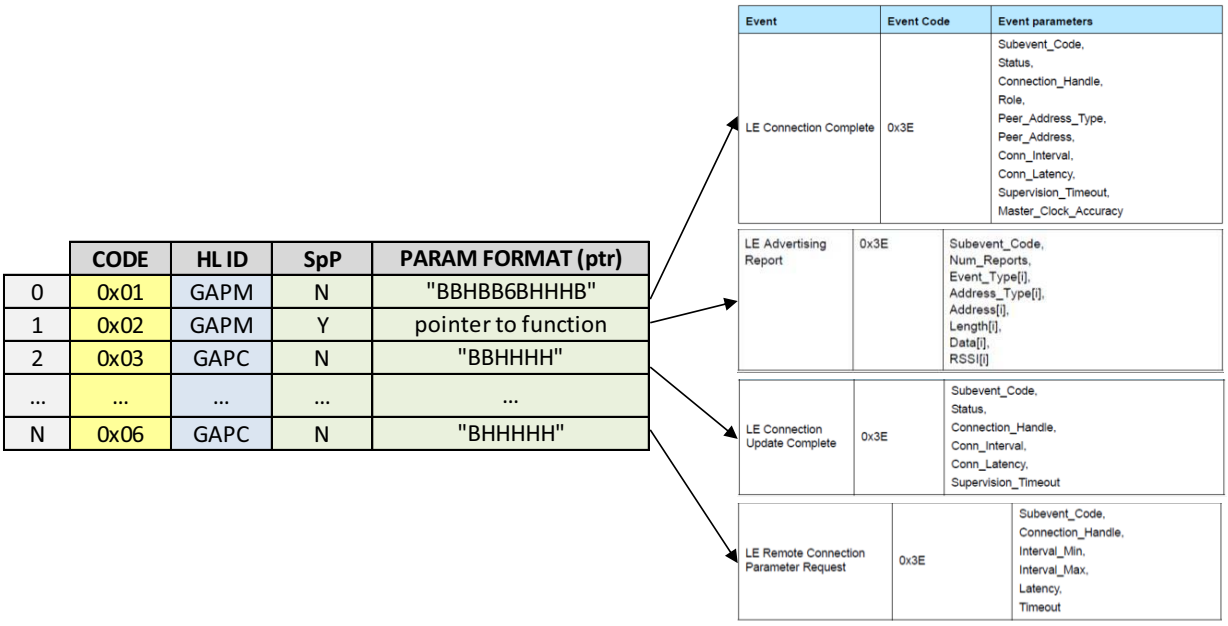
	Routing	Parameters packing	
1 byte	1 byte	1 byte	4 bytes
CODE	HL ID	SpP	PARAM FORMAT (ptr)

Figure 14. HCI Event Descriptor Format

Table 33. Event Descriptor Fields Description

Name	Size	Description
Code	1 byte	Event code or event subcode
HL ID	1 byte	Identifier of the task that receives the event
Special Parameters Packing	1 byte	Flag indicating that the parameters are packed/unpacked via a special function
Parameters Format	4 bytes	String representing the parameters format (`NULL` if no parameter), used by the generic parameter packer

The HCI software assigns one event descriptor to each of these sub-events, called LE events. As a result, a second table is present with all LE events described and indexed by their LE event subcodes, as shown in Figure 15.



	CODE	HL ID	SpP	PARAM FORMAT (ptr)
0	0x01	GAPM	N	"BBHBB6BHHHB"
1	0x02	GAPM	Y	pointer to function
2	0x03	GAPC	N	"BBHHHH"
...	...	...	...	...
N	0x06	GAPC	N	"BHHHHH"

Event	Event Code	Event parameters
LE Connection Complete	0x3E	Subevent_Code, Status, Connection_Handle, Role, Peer_Address_Type, Peer_Address, Conn_Interval, Conn_Latency, Supervision_Timeout, Master_Clock_Accuracy
LE Advertising Report	0x3E	Subevent_Code, Num_Reports, Event_Type[i], Address_Type[i], Address[i], Length[i], Data[i], RSSI[i]
LE Connection Update Complete	0x3E	Subevent_Code, Status, Connection_Handle, Conn_Interval, Conn_Latency, Supervision_Timeout
LE Remote Connection Parameter Request	0x3E	Subevent_Code, Connection_Handle, Interval_Min, Interval_Max, Latency, Timeout

Figure 15. LE Events Descriptors Table

NOTE: The fields for parameter packing or unpacking are present only if an external interface is supported. In a full stack system that does not support an external interface, only the routing fields are present in the event descriptors.

6.2.1.3 Internal Messages Definition

A kernel message is a basic exchange element used by software tasks to communicate with each other. The information carried by each HCI message is processed internally using a kernel message. However, the kernel message carrying an HCI message is not sent directly between two internal tasks. The HCI software can thus reuse some of the fields normally reserved for kernel use to organize an efficient routing and manipulation of the HCI messages. The following sections describe how the HCI software, and the blocks of user software, use the kernel message to transfer HCI messages in RSL10.

All HCI commands are internally carried through a unique kernel message filled with the following data shown in Figure 16:

MSG ID	DEST ID	SRC ID	MSG LENGTH	N + padding
CMD	Con Idx	Opcode	Param Length	PARAMS unpk

Figure 16. Kernel Message for Carrying HCI Commands

Table 34. Kernel Message Content

KE Message Field	Values
Message ID	HCI Command Message ID
Destination Task	Connection Index (only for connection oriented commands)
Source Task	Opcode
Parameters Length	Unpacked parameters length (0 for parameter-less commands)
Parameters	Unpacked parameters

Table 34 shows the kernel message contents. Thanks to the information contained in the kernel messages, each task receiving such messages can retrieve the HCI command information.

NOTE: Each lower layer task that might receive HCI commands must implement one HCI command message handler as a unique entry point. The HCI command message is responsible for processing and freeing the kernel message, and is also responsible for replying to each HCI command it receives.

6.2.1.4 Events

The controller stack can send an event to the host at any moment. It sends a kernel message that can be one of four types:

- Command Status event: in response to a procedure start
- Command Complete event: in response to a completed action
- LE event: message from Bluetooth low energy LL to host
- Legacy event: message from Bluetooth low energy LL to host

6.2.1.4.1 Legacy Events

The default container for HCI legacy events is a kernel message filled with the following data shown in Figure 17 and Table 35:

MSG ID	DEST ID	SRC ID	MSG LENGTH	N + padding
EVT	Con Idx	Event Code	Param Length	PARAMS unpk

Figure 17. Kernel Message for Carrying HCI Events

Table 35. Legacy Events Kernel Message Content

KE message field	Values
Message ID	HCI Event Message ID
Destination Task	Connection Index (only for connection oriented events)
Source Task	Event Code
Parameters Length	Unpacked parameters length (0 for parameter-less events)
Parameters	Unpacked parameters

6.2.1.4.2 LE Event

All HCI meta events are internally carried through a unique kernel message filled with the following data shown in Figure 18, and in Table 36:

MSG ID	DEST ID	SRC ID	MSG LENGTH	1	N + padding - 1
LE EVT	Con Idx	-	Param Length	SUB	PARAMS unpk

Figure 18. Kernel Message for Carrying HCI LE Events

Table 36. LE Event Kernel Message Content

KE message field	Values
Message ID	HCI LE Event Message ID
Destination Task	Connection Index (only for connection oriented events)
Source Task	Not filled
Parameters Length	Unpacked parameters length (1 for parameter-less LE events)
Parameters	Unpacked parameters

6.2.1.4.3 Command Complete Event

The HCI command complete event is internally carried through a kernel message filled with the following data shown in Figure 19 and Table 37:

MSG ID	DEST ID	SRC ID	MSG LENGTH	N + padding
CC EVT	Con Idx	Opcode	Param Length	RET PARAMS unpk

Figure 19. Kernel Message for Carrying HCI Command Complete Events

Table 37. Command Complete Event Kernel Message

KE message field	Values
Message ID	HCI CC Event Message ID
Destination Task	Connection Index (only for connection oriented events)
Source Task	Original Command Opcode
Parameters Length	Unpacked parameters length
Parameters	Unpacked parameters

6.2.1.4.4 Command Status Event

The HCI command status event is internally carried through a kernel message filled with the following data shown in Figure 20 on page 67, and in Table 38 on page 67:

MSG ID	DEST ID	SRC ID	MSG LENGTH	1
CS EVT	Con Idx	Opcode	1	STAT

Figure 20. Kernel Message for Carrying HCI Command Status Events

Table 38. Command Status Event Kernel Message

KE message field	Values
Message ID	HCI CS Event Message ID
Destination Task	Connection Index (only for connection oriented events)
Source Task	Original Command Opcode
Parameters Length	1 (Length of the parameter Status)
Parameters	Status of the command processing

6.2.1.4.5 LE ACL RX Data

The information related to HCI LE ACL RX data (received from the peer device on the Bluetooth low energy link) is carried through a unique message filled with the following data shown in Figure 21 and in Table 39:

MSG ID	DEST ID	SRC ID	MSG LENGTH	2	1	1	2	1
ACL DATA RX	Con Idx	-	LEN	CONHDL	F	Res	LEN	HDL

Figure 21. Kernel Message for Carrying HCI LE ACL RX Data Information

Table 39. LE ACL RX Data Kernel Message

KE message field	Values
Message ID	HCI CS Event Message ID
Destination Task	Connection Index (only for connection oriented events)
Source Task	Original Command Opcode
Parameters Length	1 (Length of the parameter Status)
Parameters	
	Connection handle
	Packet boundary and packet broadcast flags
	Reserved
	Data Length
	Handle of the RX buffer containing the data

6.2.1.4.6 LE ACL TX Data

The information related to HCI LE ACL TX data (sent to the peer device on the Bluetooth low energy link) is carried through a unique message filled with the following data shown in Figure 22 on page 68, and in Table 40 on page 68:

MSG ID	DEST ID	SRC ID	MSG LENGTH	2	1	1	2	4
ACL DATA TX	Con Idx	-	LEN	CONHDL	F	Res	LEN	TX descriptor

Figure 22. Kernel Message for Carrying HCI LE ACL TX Data Information

Table 40. LE ACL TX Data Kernel Message

KE message field	Values
Message ID	HCI CS Event Message ID
Destination Task	Connection Index
Source Task	Not filled
Parameters Length	Length of the parameters
Parameters	
	Connection handle
	Packet boundary and packet broadcast flags
	Reserved
	Data Length
	TX descriptor of the data to send

6.2.1.5 Internal Messages Routing

For each HCI message transferred, the HCI software decides whether to route the message internally (software task) or externally (through the transport layer). The features related to communication with external systems (host or controller), such as the reception state machine, packet TX queuing, and packet packing or unpacking, are described in Section 6.2.4, “Communication with External Host” on page 71. This section focuses on finding the internal destination of HCI messages within the internal host or controller.

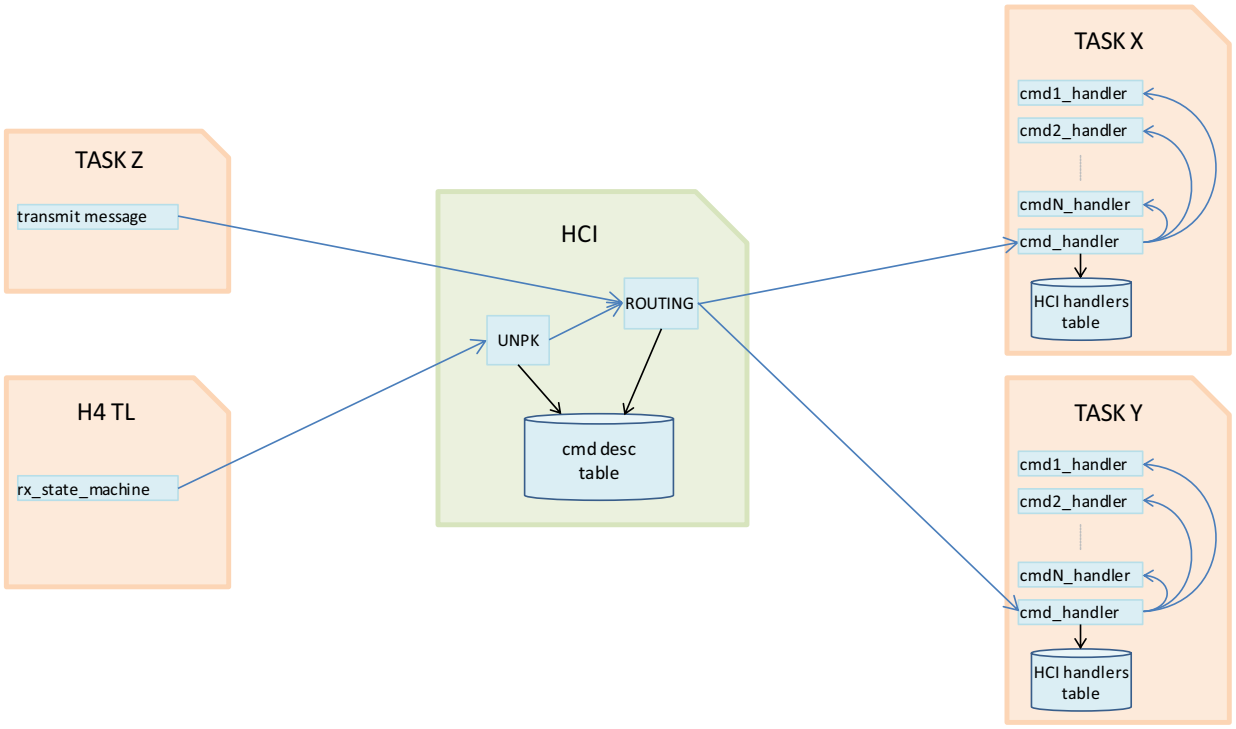


Figure 23. Message Transferring through the HCI

As seen in Figure 23, for each message transiting through the HCI (command, event, RX data, TX data), the HCI software needs to find the destination task within lower or higher layers. For example, UART Transport Layer sends the state machine to be unpacked and then rerouted to its respective task.

6.2.1.5.1 For External Host to Internal Controller

The HCI retrieves the command opcode from the HCI packet, which is used for retrieving its associated command descriptor. The descriptor contains the internal identifier that allows the HCI to associate a destination task with the message.

Control messages that are not dedicated to a specific Bluetooth low energy connection are sent to the main LL manager task (LM/LLM), which is a single instantiated task. Bluetooth low energy technology implements a manager task and is able to handle the messages specific to its own protocol. The messages related to common management of the device (e.g. `HCI_Reset_Cmd`, `HCI_Read_Local_Version_Information_Cmd`) are sent to the Bluetooth low energy controller task in Bluetooth low energy technology stand-alone configuration.

When a message is specific to a Bluetooth low energy connection (ACL data or link-specific control messages), the HCI needs to find the associated instance of the Bluetooth low energy controller task. The mechanism is mainly based on a per-connection value named “connection handle”, which is allocated by the controller at link establishment, and freed at link disconnection. Link-specific messages generally include the connection handle as part of their parameters.

The connection handle is indicated to the host by the controller when the connection has been established, thanks to the HCI LE Connection Complete event (Bluetooth low energy asynchronous connection).

A connection is considered closed by the HCI when the HCI Disconnection Complete event is transferred.

This section assumes that the connection handle is chosen by the internal controller so that it is possible for a link identifier to derive a connection handle.

To be able to route all link-oriented messages to the right Bluetooth low energy controller task instance, the HCI maintains internal data organized as shown in Figure 24.

	idx	State
BLE links	0	...
	1	...
	...	...
	M-1	...

Figure 24. Table for Link Identification (Messages Received from External Host)

The purpose of associating a status with each link is to filter the potential wrong connection handles received from the host. A message is transferred to a Bluetooth low energy controller task instance if and only if the connection handle is in the possible range and the associated link exists.

The filling of these tables is accomplished by the controller tasks themselves at link establishment or disconnection, as shown in Figure 25.

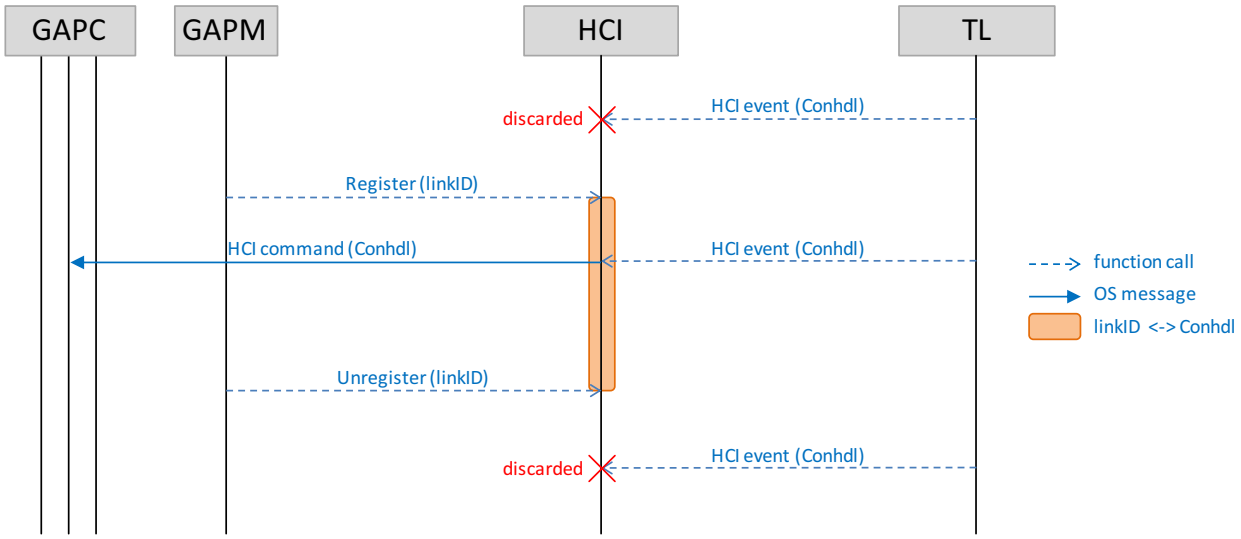


Figure 25. Bluetooth Low Energy Connection-Oriented Message Routing

As seen in Figure 25, when a link-oriented command is transferred through the HCI, the HCI checks whether there is an active link that could match on the based “State” flag and the connection handle or BD address. If no link identifier matches, a command complete event or command status is sent back to the host with the error code `Unknown Link Identifier`. If a matching link identifier is found, the destination task instance is built from the associated link identifier.

6.2.2 Between Internal Host and Controller

Communication between the internal host and the controller implies that the device is in full stack configuration, and then in Bluetooth low energy single mode, as the full stack mode is supported in Bluetooth low energy single mode only. In both directions, the HCI retrieves the command opcode or event code from the kernel message, and translates it to a higher layers or lower layers destination type. The manager task just depends on the direction (LLM task in controller, or GAPM task in host).

As aforementioned, the full stack configuration involves an internal controller only, where the connection handle allocation rules are considered known (see Section 6.2.3, “Proprietary Rules for Connection Handle Allocation” on page 71). Then, the connection handle can be directly associated with a link identifier without the need of any association table, and it is assumed that the internal host or controller never tries to transmit a message with an incorrect connection handle. Therefore, when composing the controller task destination (LLC task in controller, or GAPC task in host), the instance selection is the link ID derived from the connection handle.

6.2.3 Proprietary Rules for Connection Handle Allocation

The Bluetooth low energy controller IP internally allocates a link identifier in the range `[0 : M-1]`, where `M` is the number of Bluetooth low energy links supported. The proprietary rule to create a connection handle from the link ID is:

```
Bluetooth low energy conhdl = Bluetooth low energy link ID
```

For example, `0x02` refers to Bluetooth low energy link number 2.

These rules are given as information; they are not standard. To be compatible with third-party systems, the HCI software stores any connection handle in the link identification table as described above.

6.2.4 Communication with External Host

The HCI software handles the message routing of any message received by the transport layer to a destination block within the controller layers. Additionally, it handles command parameter unpacking, depending on the receiving system structure padding and endianness policies.

When receiving an HCI command from an external system, the transport might proceed in one or several steps. After a complete packet has been received, packet management is delegated to the HCI layer. For example, to receive a command over UART, TL gets a packet in two steps for commands with no parameters, or three steps for commands with parameters, as shown in Figure 26 on page 72.

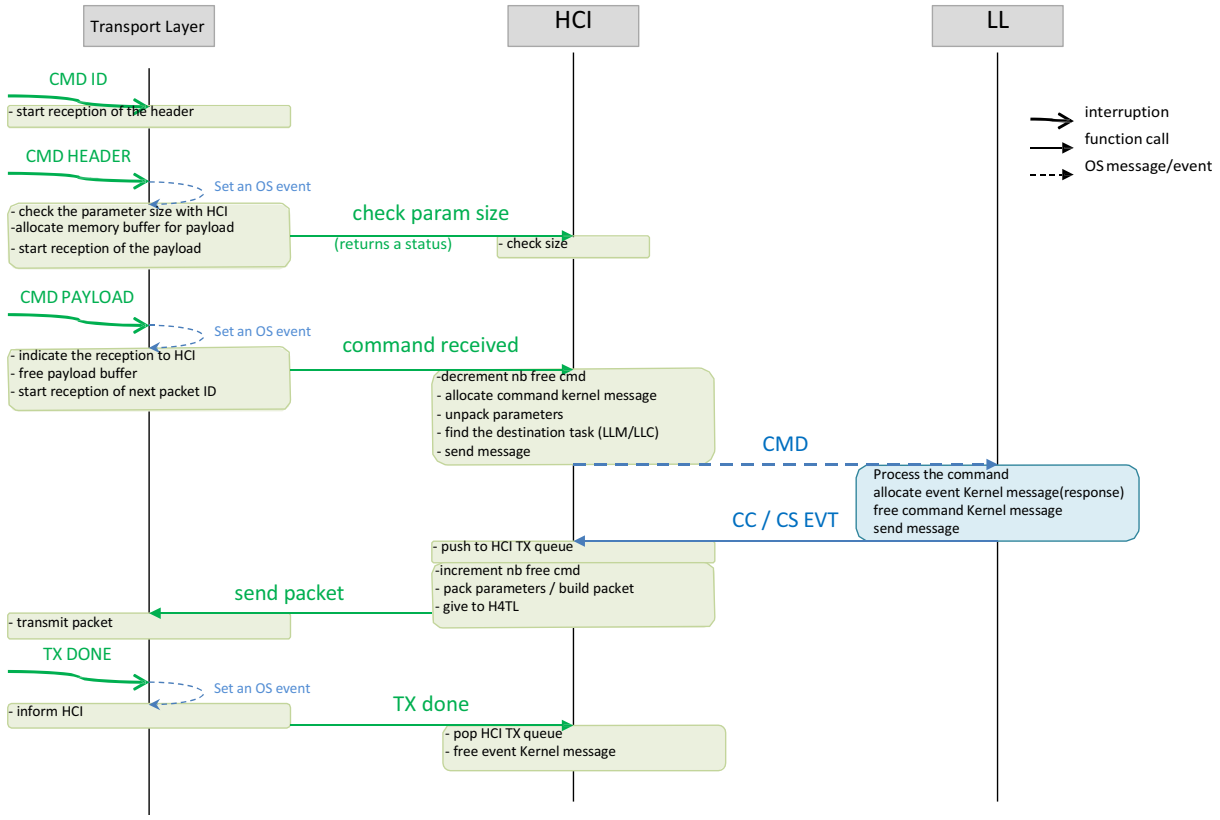


Figure 26. HCI Command Reception Flow Over UART (Command with Parameters)

As seen in Figure 23 on page 69, the UART transport layer generally works under interrupt. The command header and payload reception triggers an OS event for background processing. In background, the TL calls the HCI software to delegate the header and payload reception. Then it restarts the reception over the physical interface. A packet is considered fully received at header reception for parameter-less commands, otherwise it is considered received once the payload is received. For each command which has parameters and is checked as `valid` by the HCI, the transport Layer must allocate a memory with the appropriate space for receiving the payload.

The processing performed by the HCI at packet reception is based on the HCI command opcode. For each known packet, the HCI builds a kernel message and sends it to the right task within the Bluetooth low energy controller stack.

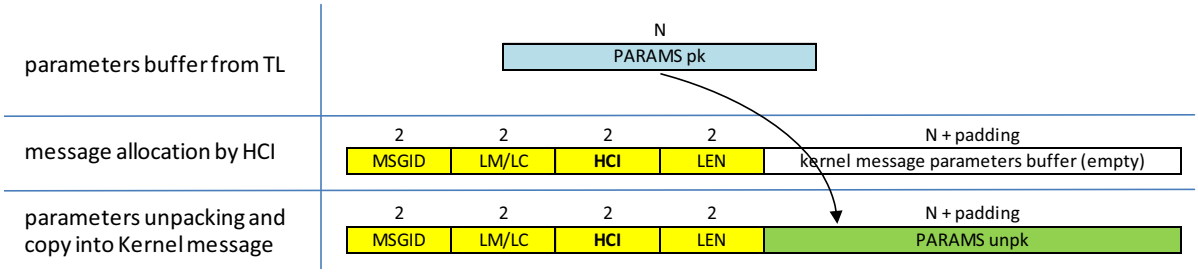


Figure 27. Data Manipulation During HCI Commands Reception

NOTE: the kernel message parameters size handles the space needed by the parameters on a C structure basis. This means that for any compiler, the space reserved is the size of the final structure. Some compilers include padding between structure fields. For that reason, the allocated size is based on the parameters format string available in the descriptor rather than the received parameters size.

Each HCI command will be replied to with an HCI command status or command complete event. These two events are particularly selected responses to HCI commands. Then their transmission through the HCI increments the current number of HCI commands the system can handle. Their special parameters manipulation is explained in the following section.

6.2.5 HCI Events

In the case of external routing, the HCI pushes the message in a transmission queue. Once a transport layer TX channel is available, the HCI builds the HCI packet and transmits the buffer to the TL. The kernel message buffer is used by the HCI to build the HCI event packet, to transmit over the transport layer. It is not freed right after being posted to the HCI, but only after the TL has confirmed the transmission (`HCI TX Done`), as shown in Figure 28.

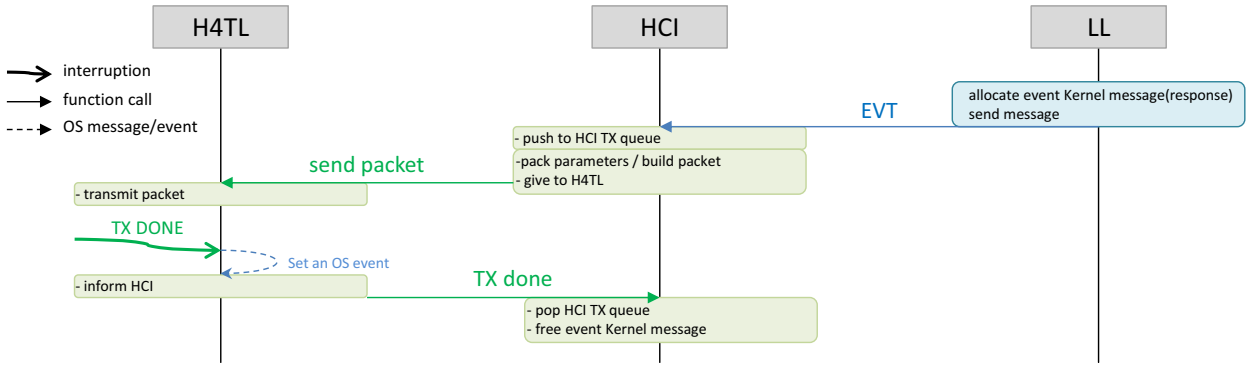


Figure 28. HCI Event Transmission Flow over UART

The events are classified in four different categories: Legacy, Command Complete, Command Status, and LE events. Each has a specific packet format and potentially specific parameter manipulation.

The HCI manages a TX FIFO for queueing several events/data for transmission. When several events are queued, the completion of one event transmission triggers the transmission of the next event. The HCI always works in OS background. The `end of transfer` interrupt from the physical layer triggers an OS event. Then the TL calls the HCI from the background.

6.2.5.1 Legacy Events

All legacy events are managed in a common way. The controller task that needs to send an event to the host uses the legacy HCI event message. When receiving this message, the HCI software will proceed to the parameter packing and sending to the transport layer, as shown in Figure 29 on page 74:

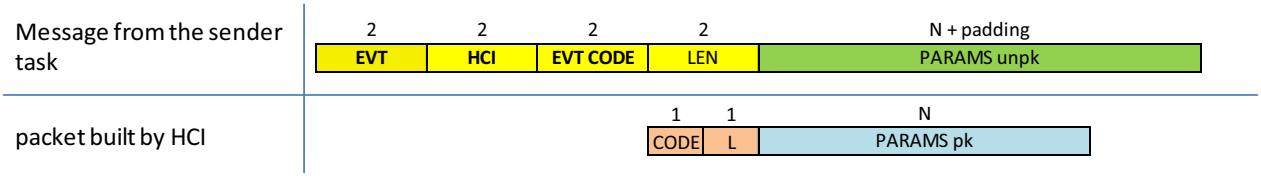


Figure 29. HCI Events Packet Building

The packet building is performed thanks to the legacy event descriptor table that contains descriptors for each supported event.

6.2.5.2 Command Complete Events

The command complete (CC) event is managed separately as it is particularly intended to reply to an HCI command (see Figure 30). It contains the original command opcode and the number of HCI commands that the controller can receive, for HCI flow control. The command complete event also contains the return parameters of the original command.

To send a CC event, a controller task composes a CC event message to the HCI. When receiving this message, the HCI performs the following actions:

- Increments the number of free commands the HCI can receive (HCI flow control)
- Packs return parameters
- Fills other fields
- Pushes to the HCI TX queue

Data manipulation over the kernel message buffer is shown in Figure 27 on page 72.

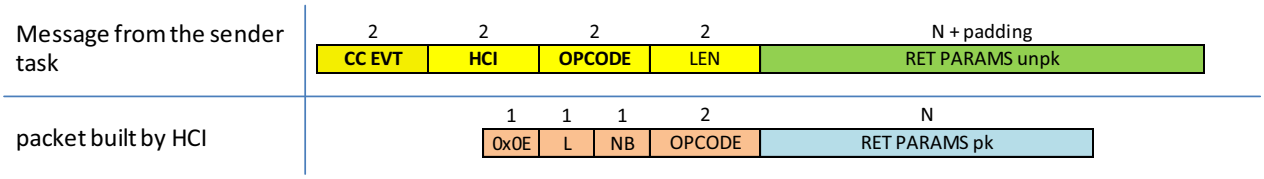


Figure 30. HCI CC Event Packet Building

The packet parameter unpacking is performed thanks to the original command descriptor found in the command.

6.2.5.3 Command Status Events

The command status (CS) event is managed separately, as it is particularly intended to reply to an HCI command. It contains the original command opcode, and the number of HCI commands that the controller can receive, for HCI flow control.

To send a CS event, a controller task composes a CS event message to the HCI. When receiving this message, the HCI performs the following actions:

- Increments the number of free commands the HCI can receive (HCI flow control)
- Builds the packet
- Pushes to the HCI TX queue

HCI CS event packet building is shown below in Figure 31:

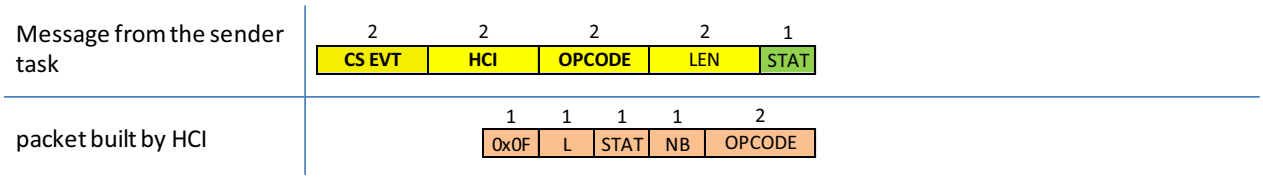


Figure 31. HCI CS Event Packet Building

6.2.5.4 LE Events

All LE events are managed in a common way. The controller task that needs to send an LE event to the host uses the LE event message. When receiving this message, the HCI performs the following actions:

- Packs parameters
- Builds the packet
- Pushes to the HCI TX queue

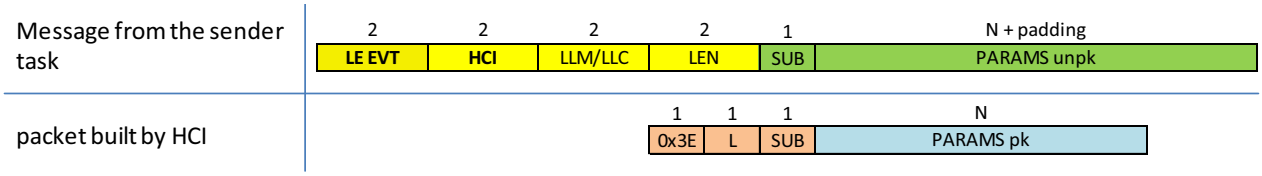


Figure 32. HCI LE Events Packet Building

The packet building is performed thanks to the LE event descriptor table that contains descriptors for each supported LE event. HCI LE events packet building is shown in Figure 32above.

6.2.5.5 HCI ACL TX Data

The data given by an external host to be transmitted over the air triggers the mechanism shown in Figure 33 on page 76:

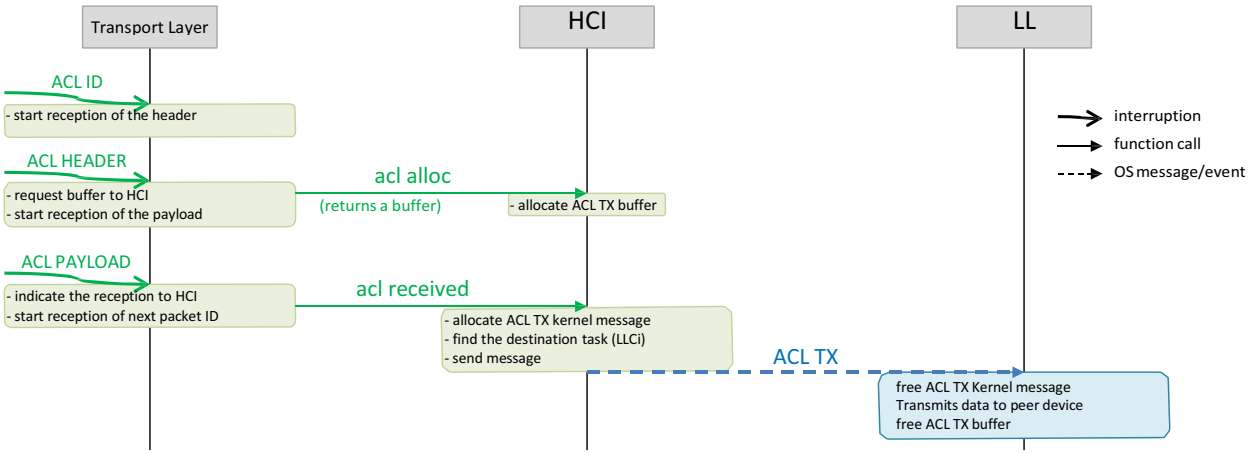


Figure 33. Reception of HCI ACL TX Data from External Host

Figure 30 on page 74 shows the behavior of the HCI in a normal case, when a correct packet is received from the host, and buffers are available. However, two error cases are possible when the transport layer receives the HCI data packet header:

1. Data length error:

   If the field received in the HCI header exceeds the maximum buffer size, the reception over the physical interface is considered erroneous. In this case, the HCI returns a `NULL` pointer, and the TL resets its reception path.

2. Buffer overflow:

   If there are no more available buffers within the stack, the HCI allocates a buffer from the RAM heaps. It frees the buffer once the TL indicates the payload reception.

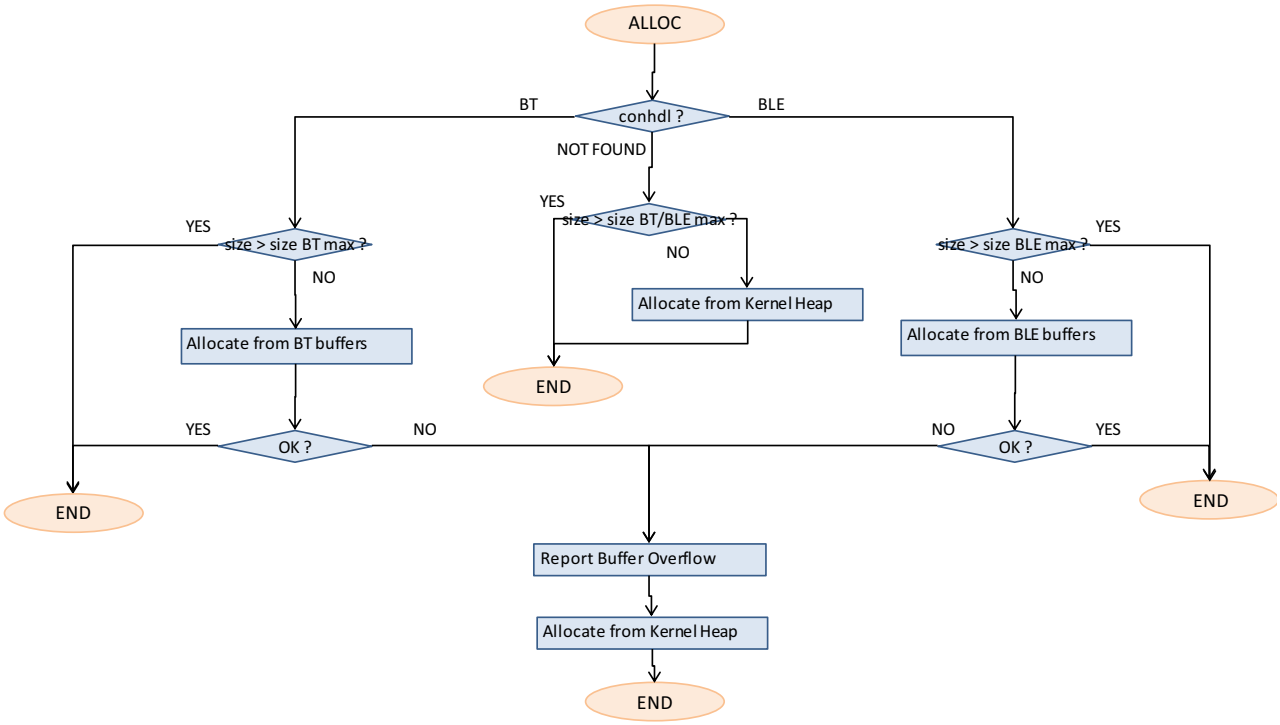


Figure 34. HCI ACL TX Data Buffer Allocation Algorithm

Figure 34 shows the algorithm executed when trying to allocate a buffer for TX data. Possible results are:

- If the payload size is higher than expected, no buffer is allocated.
- If the connection handle does not match with any active connection, or there are no more Bluetooth low energy buffers, a buffer is allocated from the heap.
- In normal cases, Bluetooth low energy technology's respective buffer management systems provide a buffer able to receive the packet payload.

Then, after reception of the payload through the TL, the action taken by the HCI follows the result of the buffer allocation, as shown in Figure 35 on page 78:

- Bluetooth low energy buffer: sends a message to LLC
- Kernel heap buffer: frees the buffer

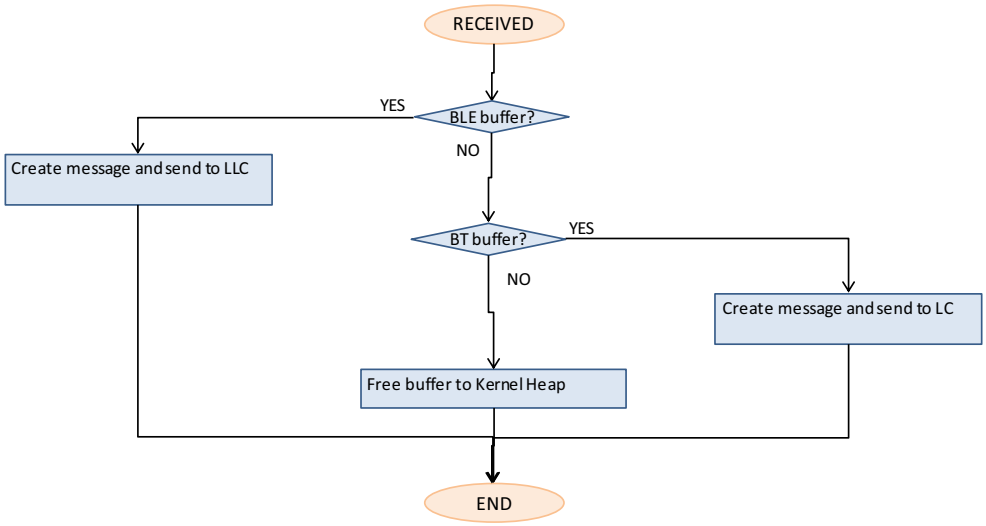


Figure 35. HCI ACL TX Data Received Algorithm

6.2.5.6 HCI ACL RX Data

The data received from the air is given to an external host, according to the following mechanism shown in Figure 36:

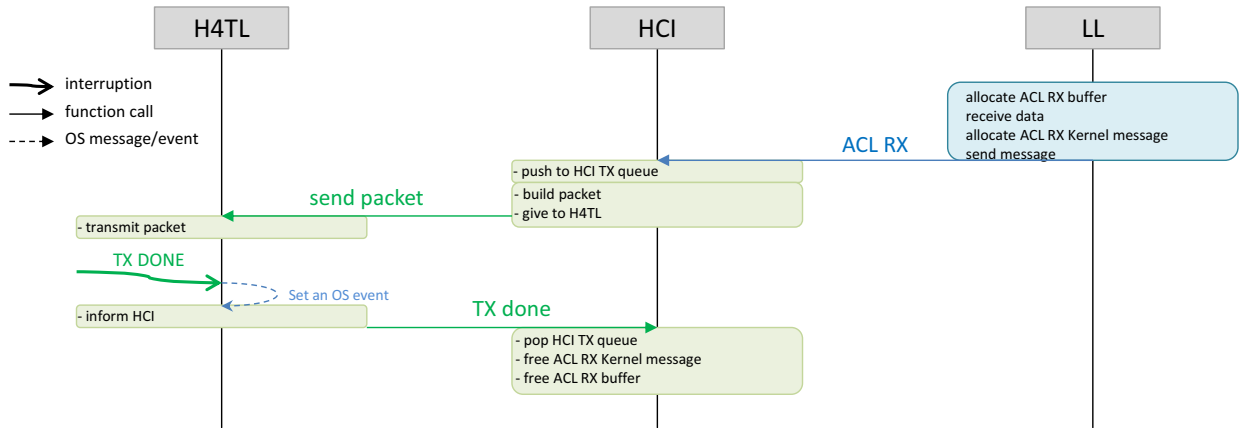


Figure 36. Transmission of HCI ACL RX Data to External Host

The kernel message used for managing the ACL packet transmission and its associated data buffer is freed when the packet has been confirmed by the physical interface.

6.2.6 Generic Parameter Packing - Unpacking

For several reasons, including portability, code size and flexibility, the HCI software preferentially uses a common method of packing and unpacking the parameters according to the needs of both sides:

- The HCI interface, which deals with byte streams where the parameters are packed and the bytes are serialized in a specific order
- The internal system, which has its own processor and memory constraints (endianness, data alignment, structure padding)

An SW utility package is included within the HCI layer. It defines generic packer and unpacker functions explained below.

6.2.6.1 Parameters Format Definition

Both the packer and unpacker take as input a string representing the parameters format. The string is a concatenation of elements that describes the parameters one-by-one.

Table 41 lists the supported format elements:

Table 41. Format Elements Definition

Element	Packed format	Unpacked format
B	1 byte	1 x 8-bits variable
H	2 bytes	1 x 16-bits variable
L	4 bytes	1 x 32-bits variable
nB	n bytes	table of n x 8-bits values * Example: “2B”, “16B”, “128B”
nH	n x 2 bytes	table of n x 16-bits values * Example: “2H”, “16H”, “124H”
nL	n x 4 bytes	table of n x 32-bits values *
NOTE: Table sizes must respect the maximum buffer size		

6.2.6.2 Generic Packer

The generic packer takes a format string as input. It also takes the parameter buffer that initially contains unpacked data. It is able to work directly within the unpacked parameter buffer.

It parses the input format string up to the end. For each element, it computes the `read` position (where the unpacked data is located), taking into account the current compiler alignment constraint. Then it copies the data to the `write` position within the restrictions of the processor endianness. The `write` location is incremented by the length of the copied data. An example of data packing for an Arm processor is shown in Figure 37 on page 80.

NOTE: The generic unpacker can also be used to determine the size of the packed data. If no buffer is given to the function, the algorithm performs a space computation without any data copy. This can be useful to check packet consistency when the TL has received the header.

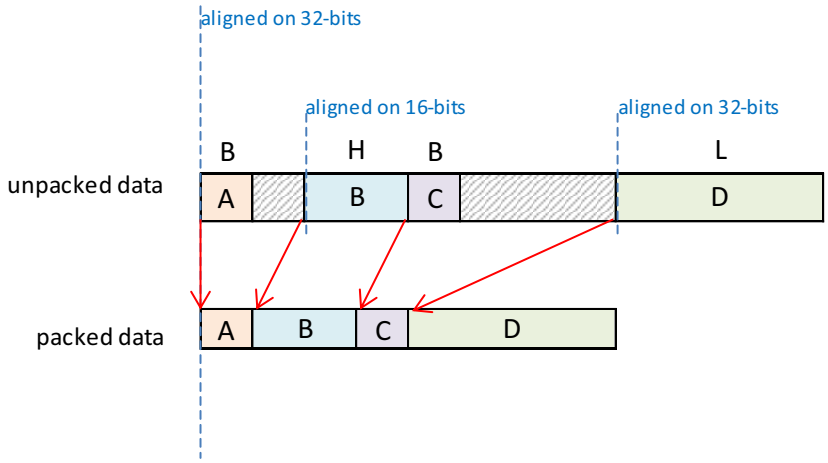


Figure 37. Example of Data Packing for an Arm Processor

6.2.6.3 Generic Unpacker

The generic unpacker takes a format string as input. It also takes the input buffer containing packed data, and the output buffer for delivering the unpacked data.

The unpacker parses the input format string up to the end. For each element, it computes the `write` position (where the unpacked data has to be written), taking into account the current compiler alignment constraint. Then it copies the data to the `write` position within the restrictions of the processor endianness. The `read` location is incremented by the length of the copied data. An example of data unpacking for an Arm processor is shown in Figure 38.

NOTE: The generic unpacker can also be used to determine the size of the unpacked data. If no buffer is given to the function, the algorithm performs a space computation without any data copy. This can be useful at buffer allocation time before receiving the data from the TL.

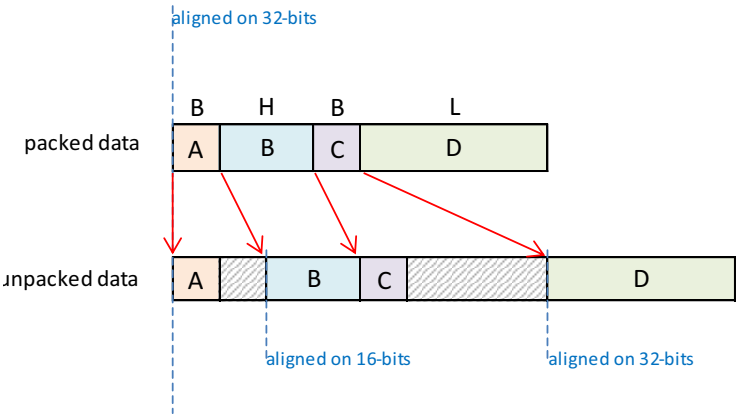


Figure 38. Example of Data Unpacking for an Arm Processor

6.2.6.4 Alignment and Data Copy Primitives

The primitives used for address alignment and data copy are located in a utility package common for all the FW (common).

Here is a list of the primitives used for HCI packing-unpacking:

- `CO_ALIGN2_HI(val)` -> align address to the following 16-bit address
- `CO_ALIGN4_HI(val)` -> align address to the following 32-bit address
- `co_read16p(ptr)` -> return a 16-bit value read at `ptr` position
- `co_read32p (ptr)` -> return a 32-bit value read at `ptr` position
- `co_write16p (ptr, val)` -> write `val` as a 16-bit value to `ptr` position
- `co_write32p (ptr, val)` -> write `val` as a 32-bit value to `ptr` position

These macros or functions must be adapted to each compiler/processor on which they are used.

6.3 GATT

The GATT is the gateway used by the Attribute Protocol to discover, read, write and obtain indications of the attributes present in the server attribute, and to configure the broadcasting of attributes. The GATT lies above the Attribute Protocol and communicates with the Generic Access Profile (GAP), higher layer profiles, and applications. The architecture of the GATT is shown in Figure 39.

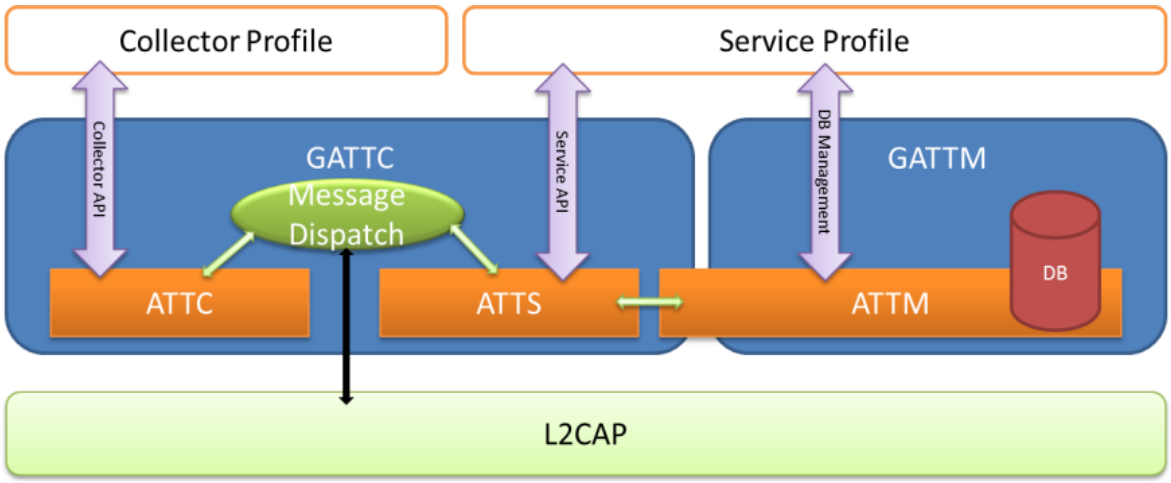


Figure 39. GATT Architecture

6.3.1 GATT Fundamentals

6.3.1.1 Roles

The GATT client is the device that initiates commands and requests to the GATT server, and can receive responses, indications and notifications from the GATT server. The GATT server is the device that accepts incoming commands and requests from the GATT client, and sends responses, indications and notifications to the GATT client. These roles are not fixed to the devices on which they run, and a device's affiliation to the role is stopped as soon as the role-specific procedure ends. A device can act in both roles simultaneously.

6.3.1.2 Security Features

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

6.3.1.3 Attribute Grouping

The GATT defines groupings of attributes to improve attribute discovery and access manipulation. The three groups are defined in Figure 40.

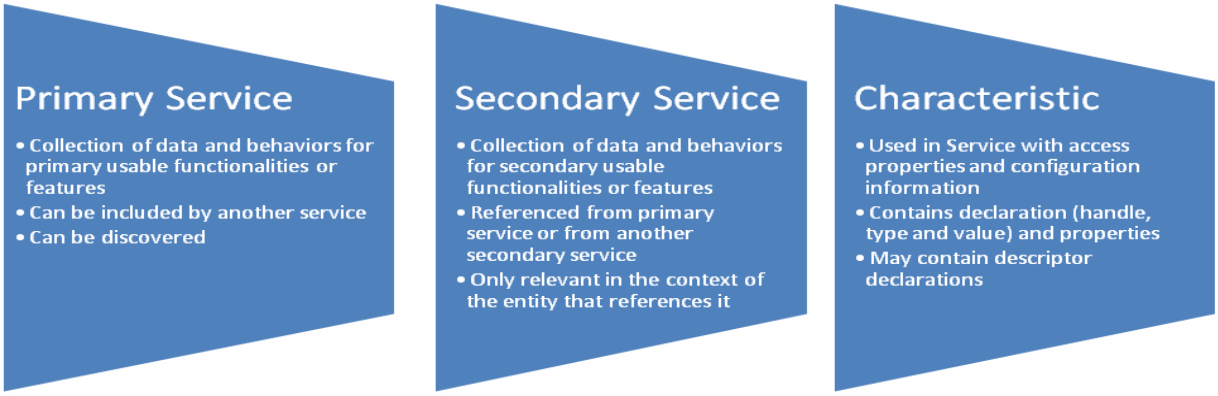


Figure 40. ATT Grouping

6.3.1.3.1 Service

The service definition contains a service declaration, and contains both include and characteristic definitions. The service declaration is an attribute with the attribute type set to UUID for primary service (as shown in Figure 41), or secondary service (as seen in Figure 42 on page 83).

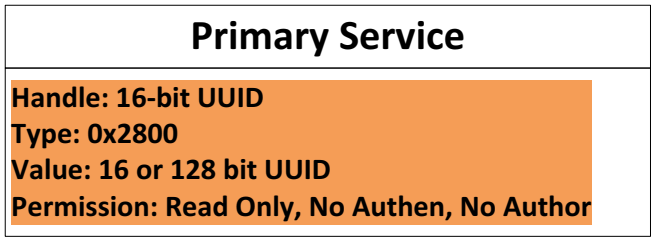


Figure 41. Primary Service Declaration

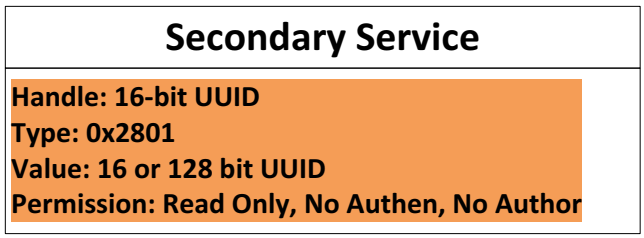


Figure 42. Secondary Service Declaration

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

6.3.1.3.2 Included Service

Include definition contains only one include declaration, as shown in Figure 43.

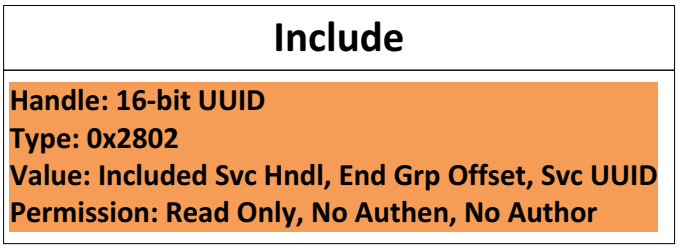


Figure 43. Include Declaration

The Include declaration is an attribute with its attribute type set to `0x2802`. This value is set to the attribute handle, End group offset and UUID for the service (2, 4 or 16 octets). If the attribute client detects a circular reference or nested include declarations to a greater level than it expects, it will terminate the ATT Bearer.

6.3.1.3.3 Characteristics

The characteristic definition contains a characteristic declaration, value, and might contain a characteristic descriptor declaration, as seen in Figure 44.

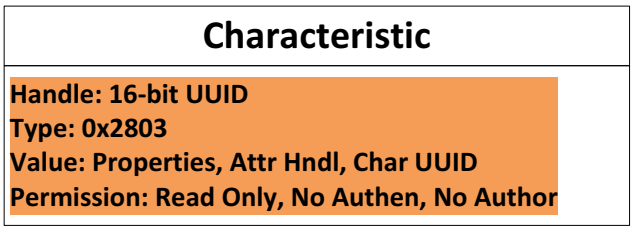


Figure 44. Characteristic Declaration

The characteristic declaration is an attribute with the attribute UUID type set to `0x2803`, and the attribute value set to the characteristic properties, value attribute handle, and value UUID (2, 4 or 16 octets).

6.3.1.3.3.1 Characteristic Extended Properties (CEP)

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

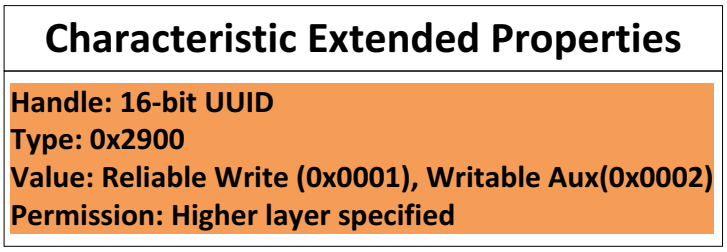


Figure 45. Characteristic Extended Properties Declaration

The characteristic extended properties declaration is a descriptor that gives more characteristic information, as shown in Figure 45. The descriptor is an attribute with type set to `0x2900`, and the attribute value equal to a set characteristic extended properties bit field.

6.3.1.3.3.2 Characteristic User Description

The characteristic user description declaration is an optional characteristic descriptor of a UTF-8 string of variable sized textual description of the characteristic value.

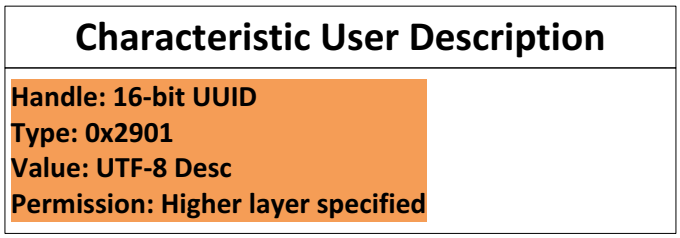


Figure 46. Characteristic User Description Declaration

The descriptor is an attribute with type set to `0x2901`, and the value set to user description UTF-8 format, as seen above in Figure 46.

6.3.1.3.3.3 Client Characteristic Configuration (CCC)

An attribute client can write a pre-configured descriptor to control the configuration of a characteristic on the server for the client. The declaration of the client characteristic configuration is readable and writable.

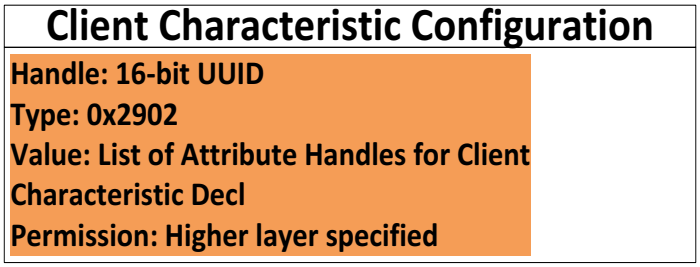


Figure 47. Client Characteristic Configuration Declaration

The descriptor is an attribute with type set to `0x2902`, and the value set to characteristic descriptor value, as seen above in Figure 47.

6.3.1.3.3.4 Server Characteristic Configuration (SCC)

An attribute client can write a pre-configured descriptor to control the configuration of a characteristic on the server for all attribute clients.

The declaration of the server characteristic configuration is readable and writable.

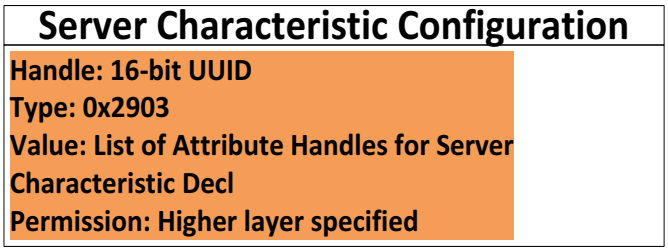


Figure 48. Server Characteristic Configuration Declaration

The descriptor is an attribute with type set to `0x2903`, and the value set to characteristic descriptor value, as shown above in Figure 48.

NOTE: Service data in advertising data is managed by application using the GAP interface.

6.3.1.3.3.5 Characteristic Presentation Format

The characteristic presentation format declaration is an optional characteristic descriptor that describes the characteristic value format. The value is composed of five parts: format, exponent, unit, name space and description, as seen below in Figure 49 on page 86.

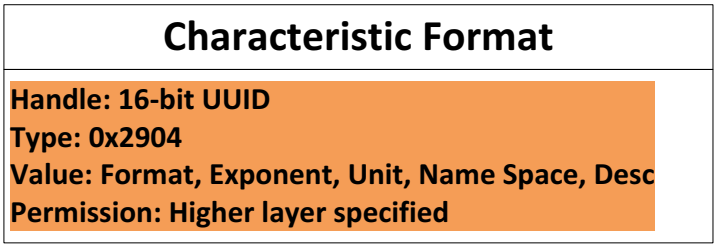


Figure 49. Characteristic Format Declaration

The access permissions are profile- or implementation-defined. The bit ordering is little-endian. Format components are shown below in Figure 50.

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

Figure 50. Format Components

6.3.1.3.3.6 Characteristic Aggregate Format

The characteristic aggregate format declaration is an optional characteristic descriptor that defines the format of an aggregated characteristic value, composed of a list of attribute handles of characteristic format declarations, as seen in Figure 51.

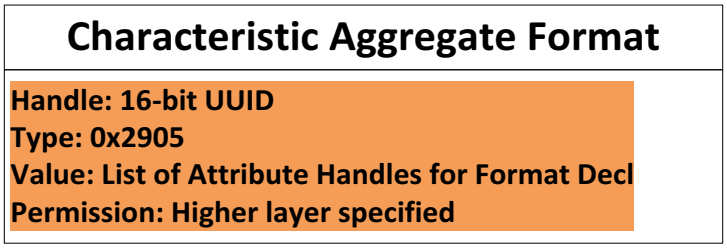


Figure 51. Characteristic Aggregate Format Declaration

The attribute value is a list of attribute handles, which is the concatenation of multiple 16-bit attribute handle values. The list contains at least two attribute handles for characteristic presentation format declaration.

6.3.1.4 L2CAP

Table 42 on page 87 shows the GATT requirements for L2CAP.

Table 42. GATT Requirements for L2CAP

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

6.3.2 Attribute Protocol Toolbox

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

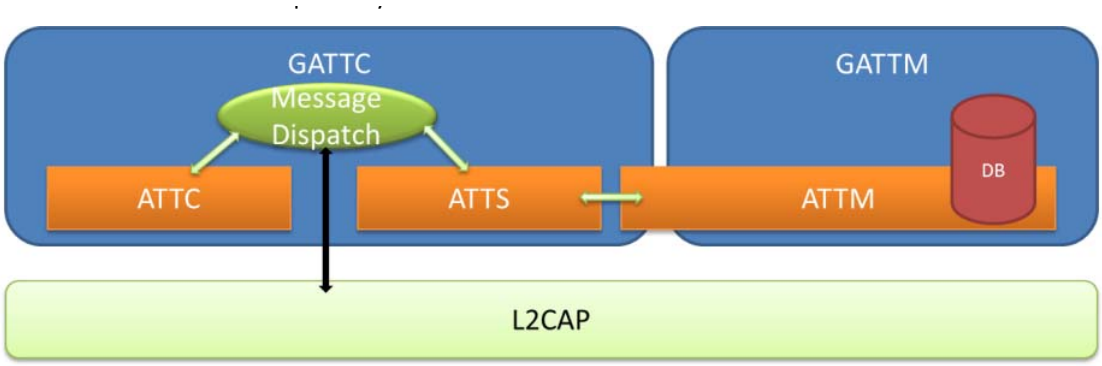


Figure 52. Attribute Module Toolbox Overview

The Attribute Block is composed of three entities: attribute server, attribute client and attribute manager, as show above in Figure 52.

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

The Attribute Client (ATTC) handles the client-related request messages for the attribute server.

The Attribute Manager (ATTM) is responsible for storing the attribute database of the device.

NOTE: The attribute toolbox is not task oriented, and can only be used by generic attribute tasks.

6.3.2.1 Basic Attribute Concepts

6.3.2.1.1 Attribute

An attribute is the basic block of the attribute protocol. It is composed of four items: attribute handle, type, value and permission property, as shown in Table 43. The access permission of the attribute is defined by the higher layer, and is not accessible through the attribute protocol.

Table 43. Attribute Description

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

6.3.2.1.2 Protocol Methods

Examples of protocol methods are request, response, command, notification, indication and confirmation. These are used by the attribute protocol to find, read, write, notify and indicate attributes, as shown in Figure 53 on page 88.

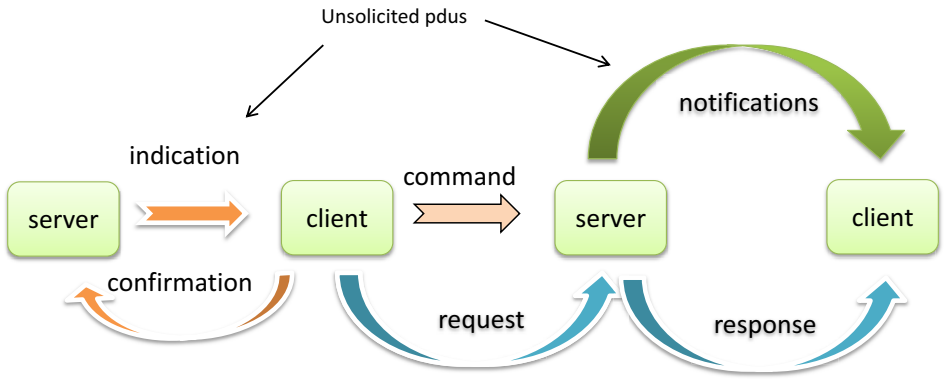


Figure 53. Overview of ATT Protocol Messages

6.3.2.2 Attribute Protocol Packet Data Unit Format

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

There are 6 types of attribute protocol PDUs (protocol data units):

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

Multi-octet fields within the attribute protocol are transmitted with the least significant octet first (little endian). Attribute PDUs may or may not contain signatures, as shown in Figure 54 (without a signature), and in Figure 55 (with a signature).

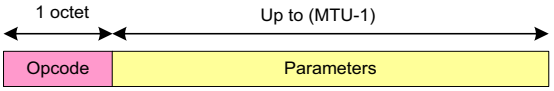


Figure 54. ATT PDU Without Signature

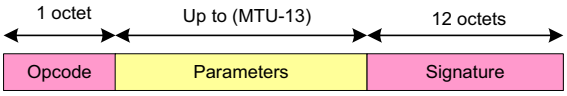


Figure 55. ATT PDU With Signature

L2CAP attribute protocol PDU messages are described in the Core Specification.

6.3.2.3 Attribute Protocol Operations

6.3.2.3.1 Atomic Operations

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

6.3.2.3.2 Flow Control

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

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

6.3.2.3.3 Transaction

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

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

Similarly, a transaction starts when an indication is sent by the attribute server. A transaction is completed when the confirmation is received by the attribute server. A transaction must be completed within 30 seconds, or else it is considered to have timed out. If a transaction has not completed before it times out, then this transaction is considered to have failed, and the local higher layers are informed of this failure. No more ATT transactions will be accepted for the link.

6.3.2.4 Attribute Protocol Module Interfaces

6.3.2.4.1 Interface with Upper Layers

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

- Reading/writing attributes, and receive notifications and indications (client side)
- Sending notifications/indications, and being notified when a client reads or writes an attribute (server side)

This API is implemented as functions available for GATT Modules

6.3.2.4.2 Interface with L2CAP

The interface with L2CAP is handled by the GATT task. It then uses the attribute toolbox to process them.

6.3.2.5 Attribute Manager (Database Owner)

Managed by the Attribute Manager module, the attribute database is composed of a list of services dynamically allocated. A service is a memory block allocated from the kernel attribute heap, and available for the attribute manager as a list of services sorted by handles (see figure Figure 56 on page 90).The Attribute manager provides a function API available for the GATTM to allocate new services with a specific start handle. If not set, the start handle is dynamically allocated.

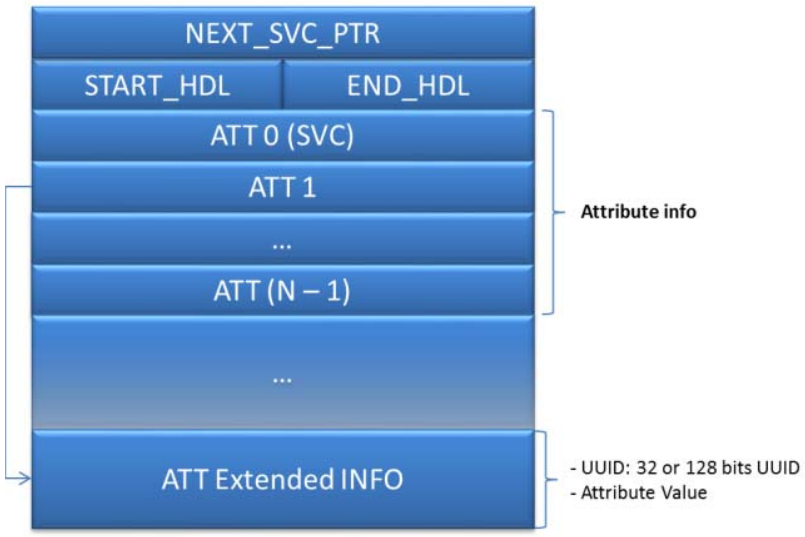


Figure 56. Service Description Block of ATT Database

This memory block contains a pointer to the next service (NEXT_SVC_PTR), its start handle, and the last handle value, followed by an array of attribute definitions (Section 6.3.2.5.1, “Attribute Definition” on page 91). The first attribute in the service memory block describes the services (see Section 6.3.2.5.3, “Service Permission Field” on page 92). It is used to determine service permissions and the number of attributes present in the service. It is forbidden

to have multiple services attributes in a service memory block. Finally, the end of memory block is used to retrieve 32-bit or 128-bit UUIDs and attribute values that can be read from the database.

NOTE: Attribute handles are unique; services handles have to be exclusive.

NOTE: Services handle mapping must be fixed to prevent the collector from performing discovery at each connection.

6.3.2.5.1 Attribute Definition

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

- Service task ID
- Pointed handle
- Maximum attribute length
- Value
- Value offset

NOTE: If the attribute UUID is a 32- or 128-bit UUID, the UUID value contains the offset where it can be found in the service block.

NOTE: Figure 57 on page 91 describes the attribute types specified by the Core Specification.

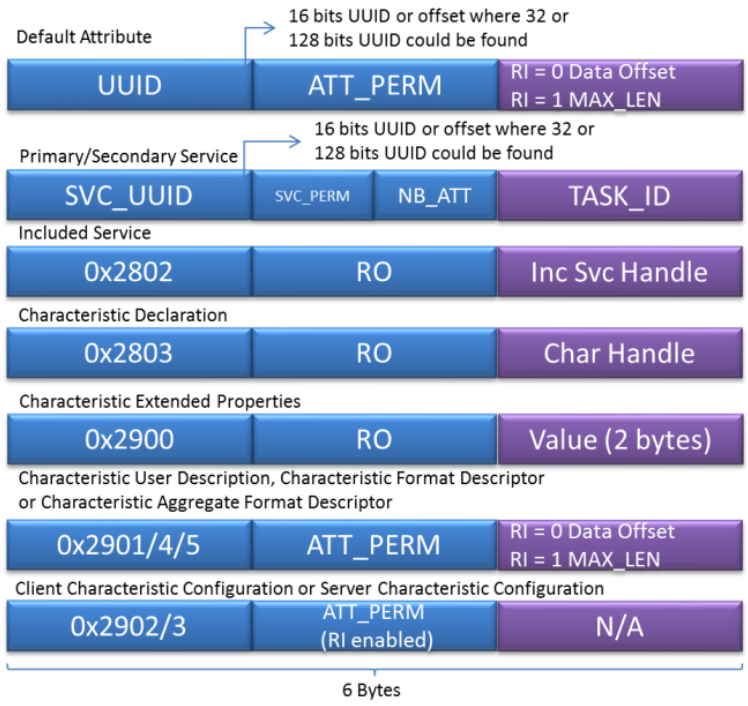


Figure 57. Attributes Types

6.3.2.5.2 Service Definition

A service is described with a 6-byte field, as shown below in Figure 58:

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

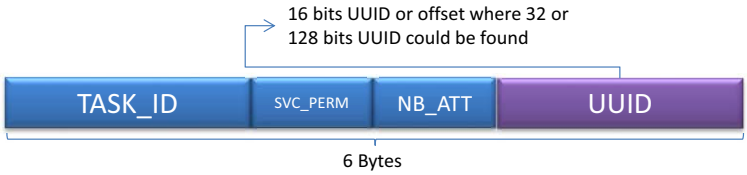


Figure 58. Service Definition

NOTE: If service UUID is a 32- or 128-bit UUID, the UUID value contains the offset where it can be found in the service block.

6.3.2.5.3 Service Permission Field

Service permission is an 8-bit field used to describe service.

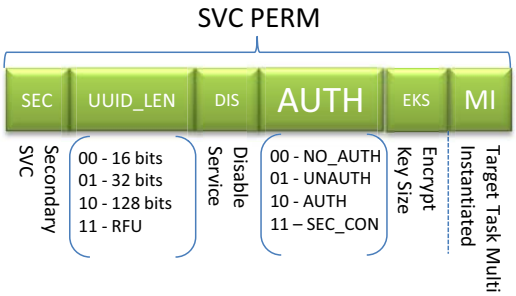


Figure 59. Service Permission Field

Service Permission Field Definitions, as seen above in Figure 59:

- SEC: used to know if the service is a primary or a secondary service
- UUID_LEN: get service UUID length (16, 32 or 128 bits)
- DIS: disables the service
- AUTH: force a level of authentication for attributes present in the service. Note, this has no impact on attributes which are read-only mandatory.
- EKS: requires an encryption key of 16 bits for an attribute requiring an authentication level.
- MI: shows whether the task that manages a service is multi-instantiated or mono-instantiated

6.3.2.5.4 Attribute Permission Field

Attribute permission is a 16-bit field, as show below in Figure 60:

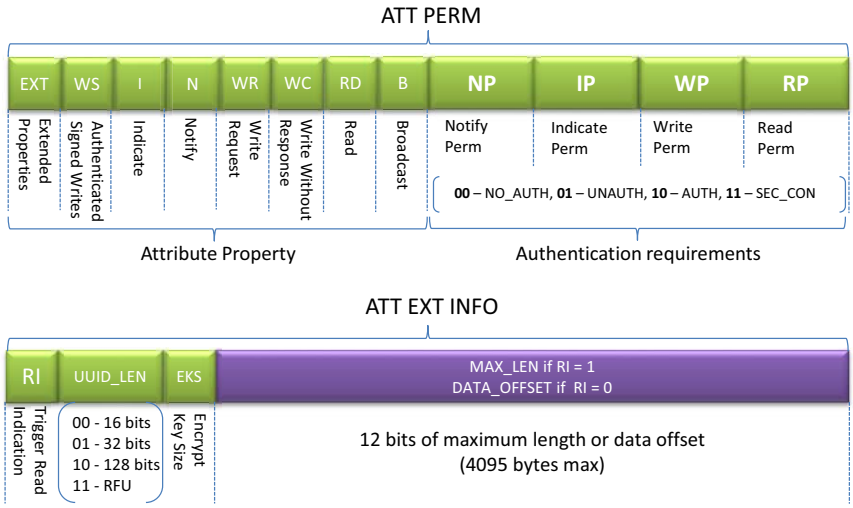


Figure 60. Attribute Permission Field

The following field is used to generate the value of the characteristic declaration property:

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

Attribute Authentication Requirements

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

NOTE: For an attribute value, permissions are used to generate the characteristic description property value.

Extended Attribute Information

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

OR

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

6.3.2.5.5 Data Caching

To ease database browsing, since several searches can be performed on the same service, keeping the pointer to the last search service in the environment variable speeds up the service and attribute discovery.

6.3.2.5.6 Attribute Database Example

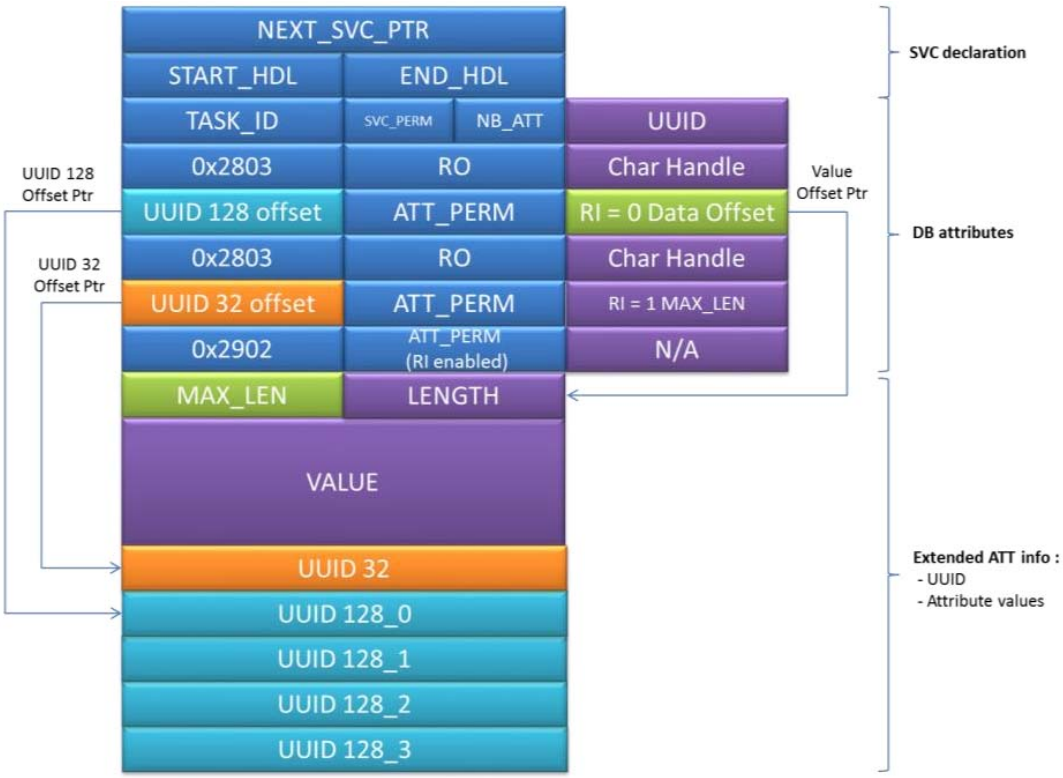


Figure 61. Attribute Database Example

6.3.2.6 Attribute Server

The attribute server has direct (function call) interface with the attribute database present in the attribute manager. It uses this interface to browse services and read characteristic values. (An example of the attribute database is shown in Figure 61.)

6.3.2.6.1 Attribute Discovery / Read

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

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

The discovery procedure is rescheduled in the kernel several times before completion. An incomplete response is kept in a cache variable and is fulfilled at the end of the search, or if the MTU is exceed.

This procedure also uses data caching of the ATTS to accelerate the read (6.3.2.6.1.4).

The search algorithm is described in Figure 62 on page 95.

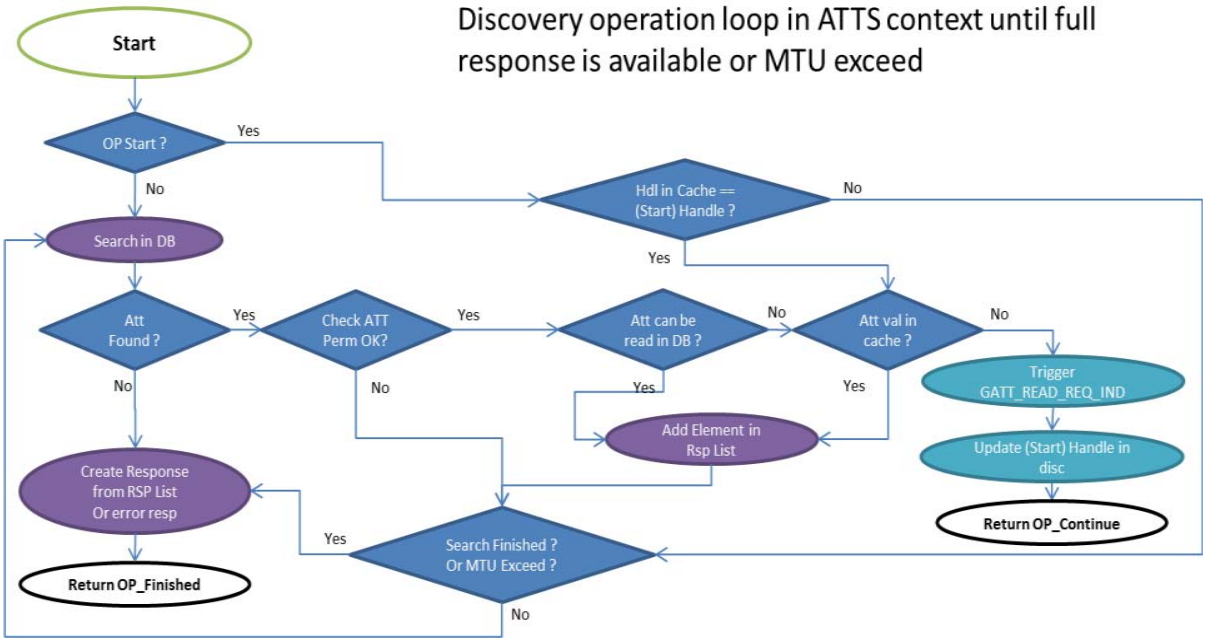


Figure 62. Attribute Discovery State Machine

6.3.2.6.2 Attribute Write

Figure 63, Figure 64 on page 96, Figure 65 on page 96, Figure 66 on page 97, and Figure 67 on page 98 describe different types of write procedures in ATT.

- Write Request

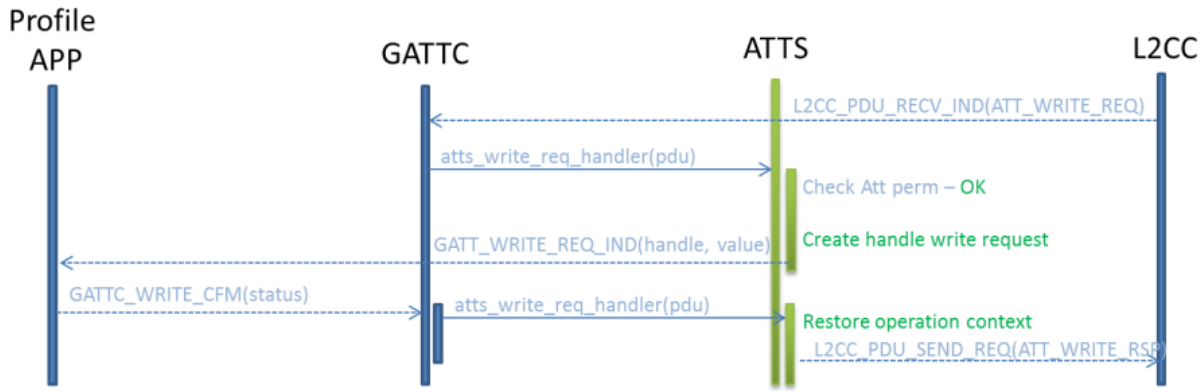


Figure 63. Write Request MSC

- Write Command

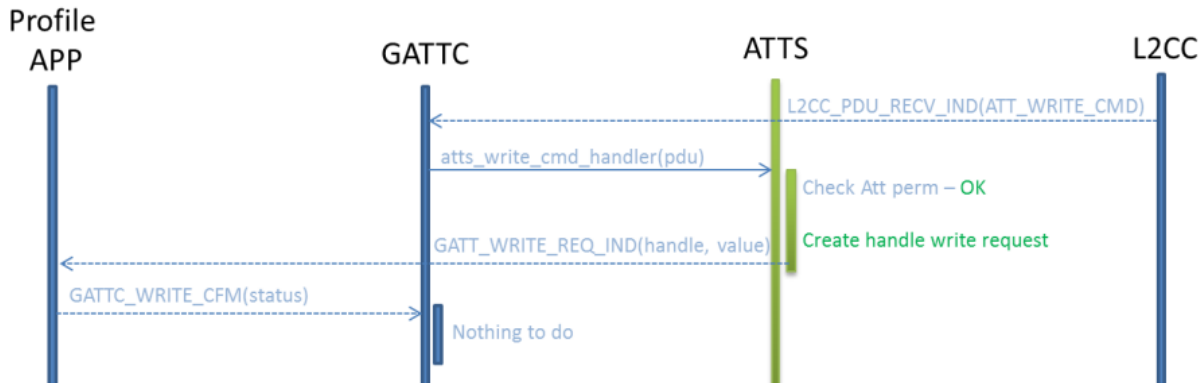


Figure 64. Write Request MSC

- Write Signed

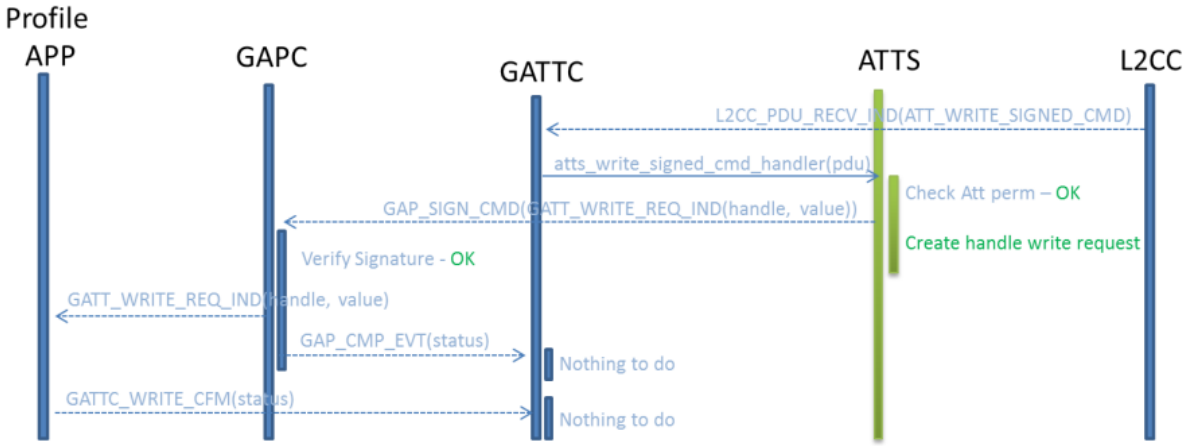


Figure 65. Write Signed MSC

- Write Long/Multiple

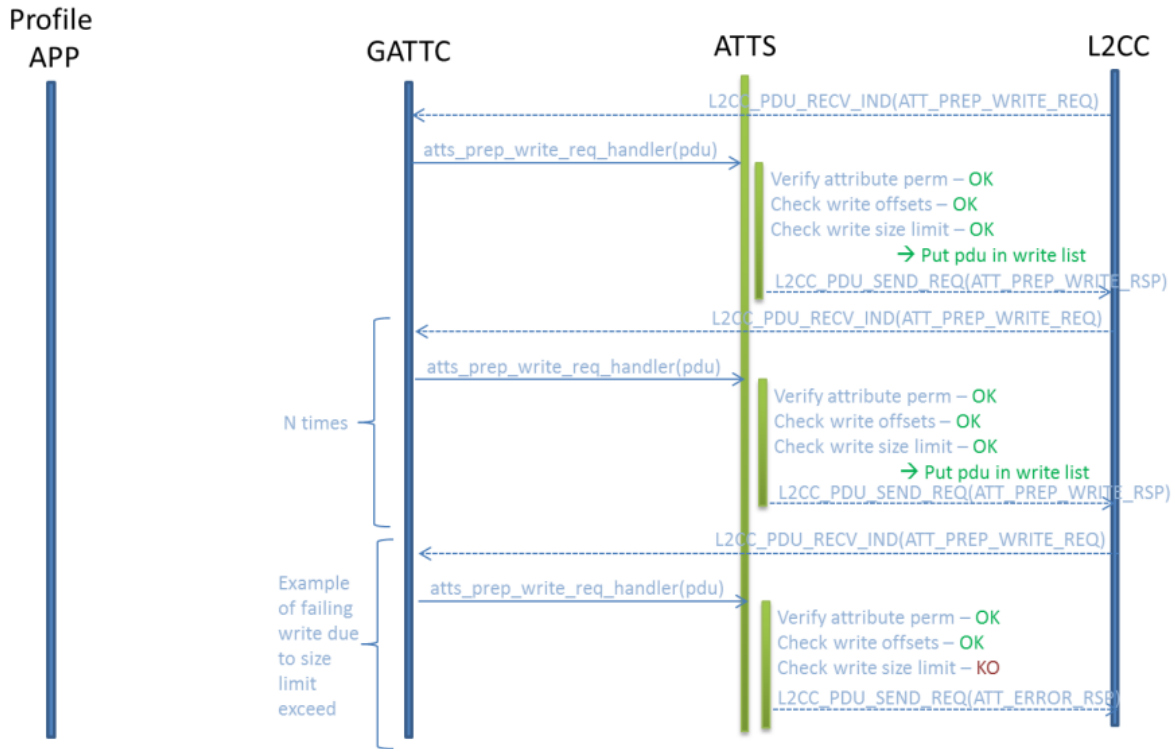


Figure 66. Multiple Prepare Write MSC

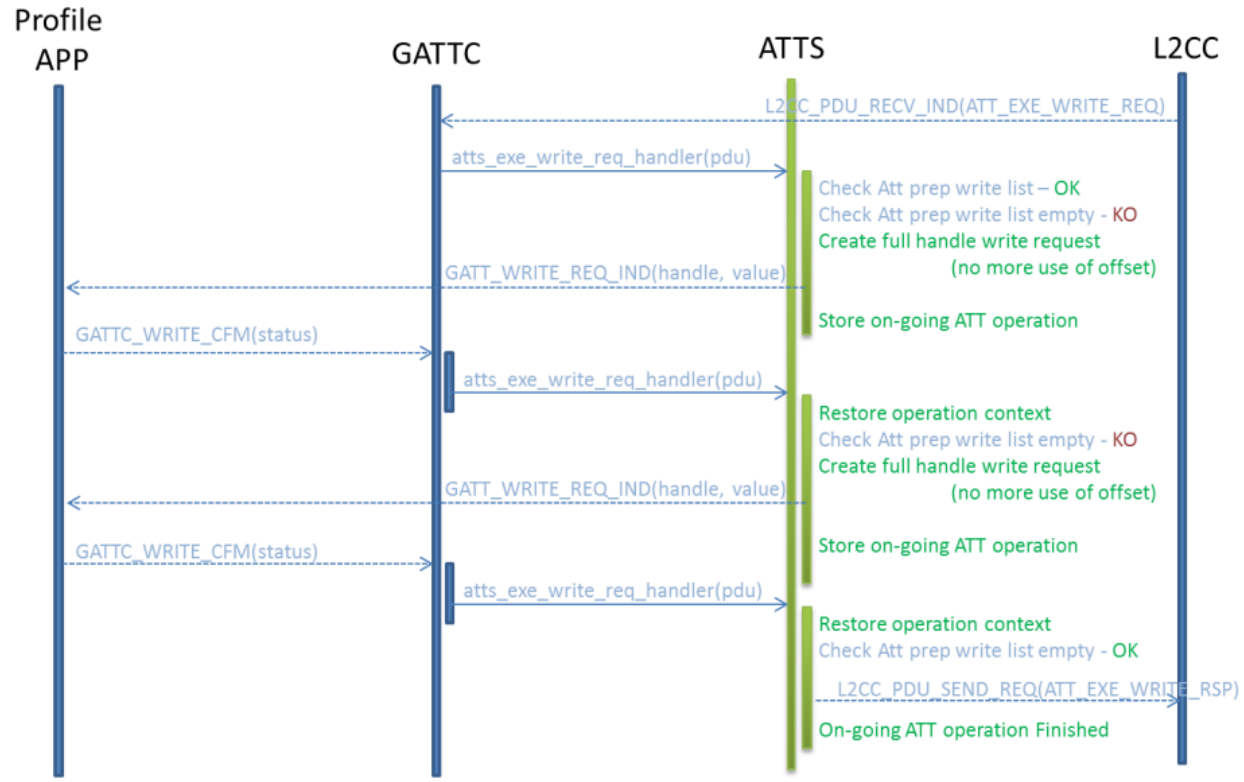


Figure 67. Execute Write MSC

NOTE: The write request is always send to the profile that manages the handle. It requires a confirmation of write event whether a message is triggered to a peer device or not.

6.3.2.6.3 Server Initiated Events

The attribute server can be used to trigger some indications or notifications, as shown in Figure 68, and in Figure 69 on page 99:

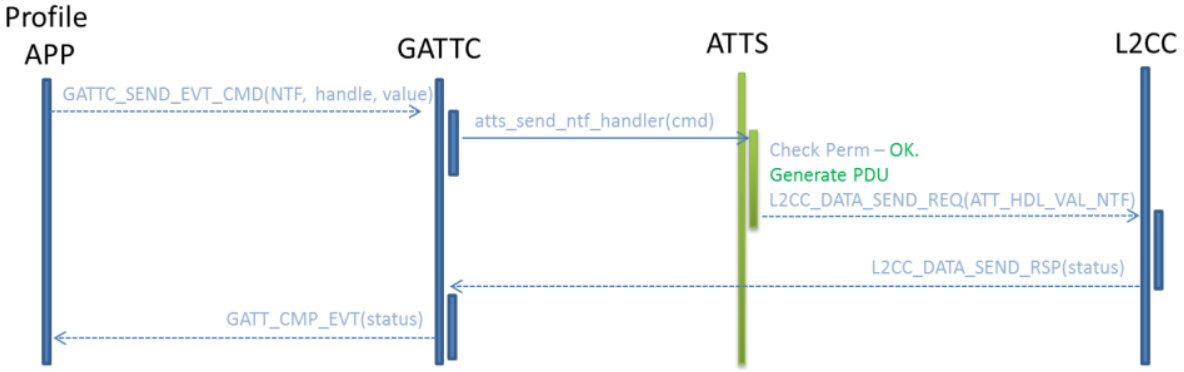


Figure 68. Trigger Notification MSC

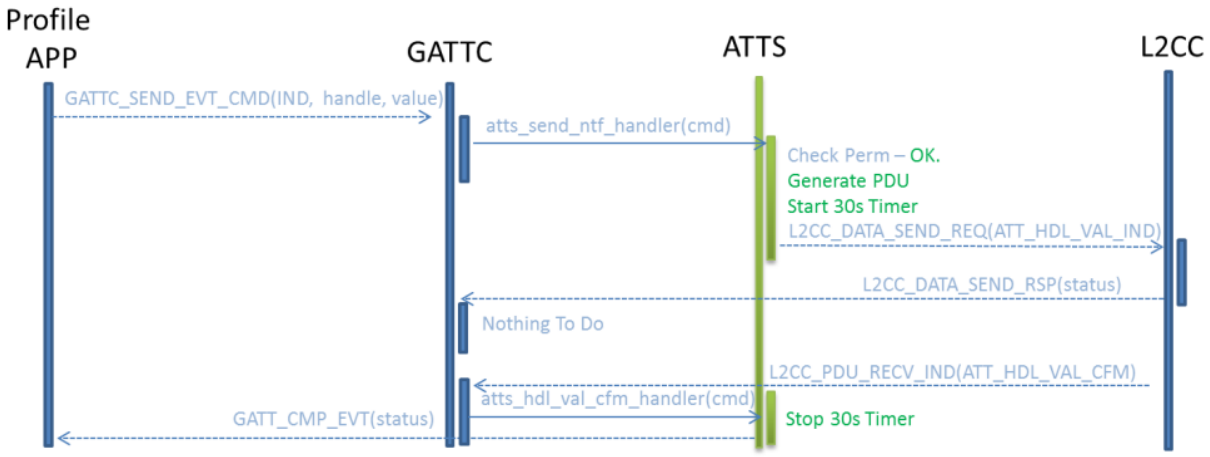


Figure 69. Trigger Indication MSC

NOTE: Notification/Indication data is present in the event message. This event message can be used to update the database value (if the attribute value is present in the database).

6.3.2.6.4 Data Caching

Ongoing procedure: The ongoing procedure pointer (L2CAP message) is kept to be rescheduled by the kernel until the operation is finished, and to perform flow control on the requests.

Response cache: Until the executed procedure is finished, a partial procedure response is stored in the attribute server environment.

Prepare write cache: For a non-atomic write, a cache is required. This cache is fed by the prepare write and flushed during the execute write requests.

Read attribute cache: In the attribute database, to speed up read procedures, the value of the latest attribute read is kept until:

- A write request is accepted for this attribute
- Notification/indication is triggered for this attribute
- The attribute is fully read by a peer device
- Disconnection

(See figure Figure 70 on page 100.)

NOTE: The attribute value is put in the cache if the value is not present in the attribute database.

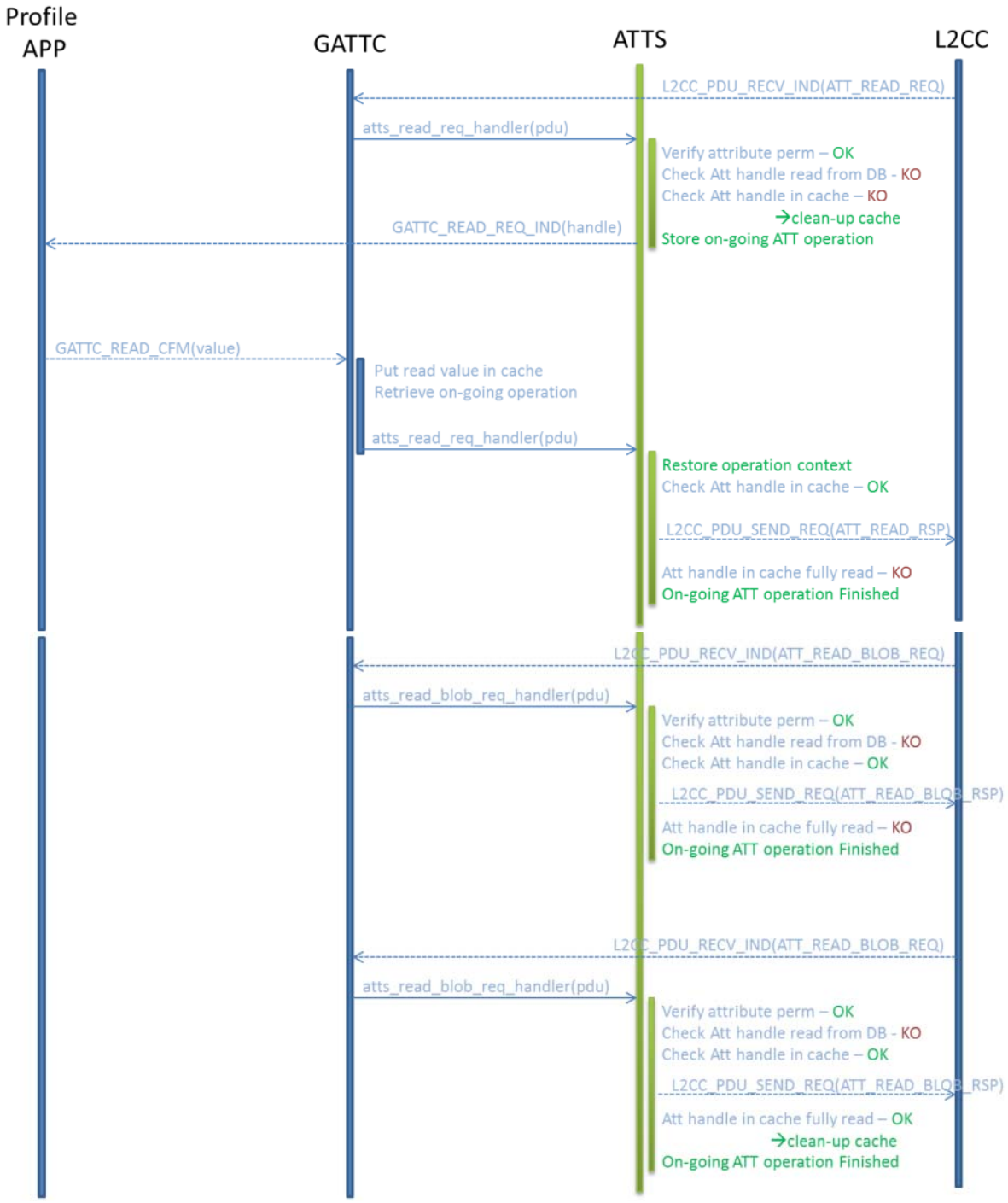


Figure 70. Data Caching of Latest Read Attribute MSC

6.3.2.7 Attribute Client

The attribute client role is very simple; it conveys requests from the GATT client to the L2CAP, managing transaction atomicity and maximum duration using a timer.

NOTE: Discovery and read procedures, using UUID as input, support 16-, 32- and 128-bit UUIDs.

6.3.2.7.1 Discovery Command

Discovery of peer services, peer characteristics, and peer descriptions is illustrated in Figure 71, Figure 72 on page 102, Figure 73 on page 103, Figure 74 on page 104, and Figure 75 on page 105.

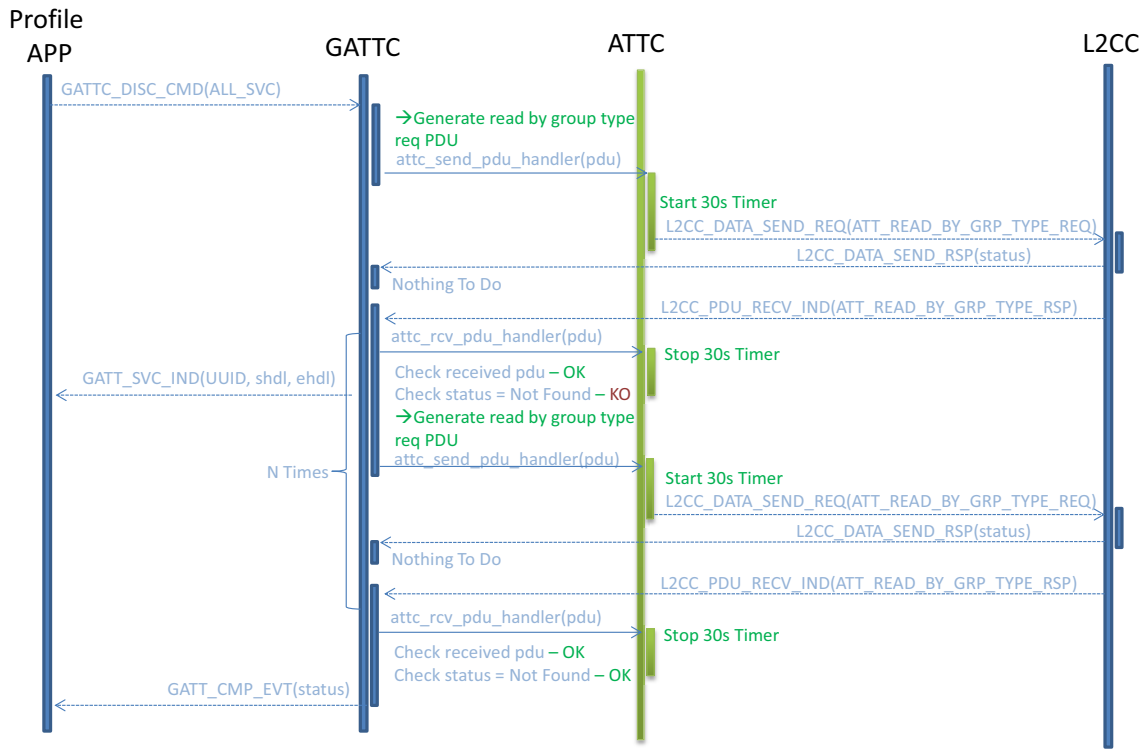


Figure 71. Discover All Peer Services MSC

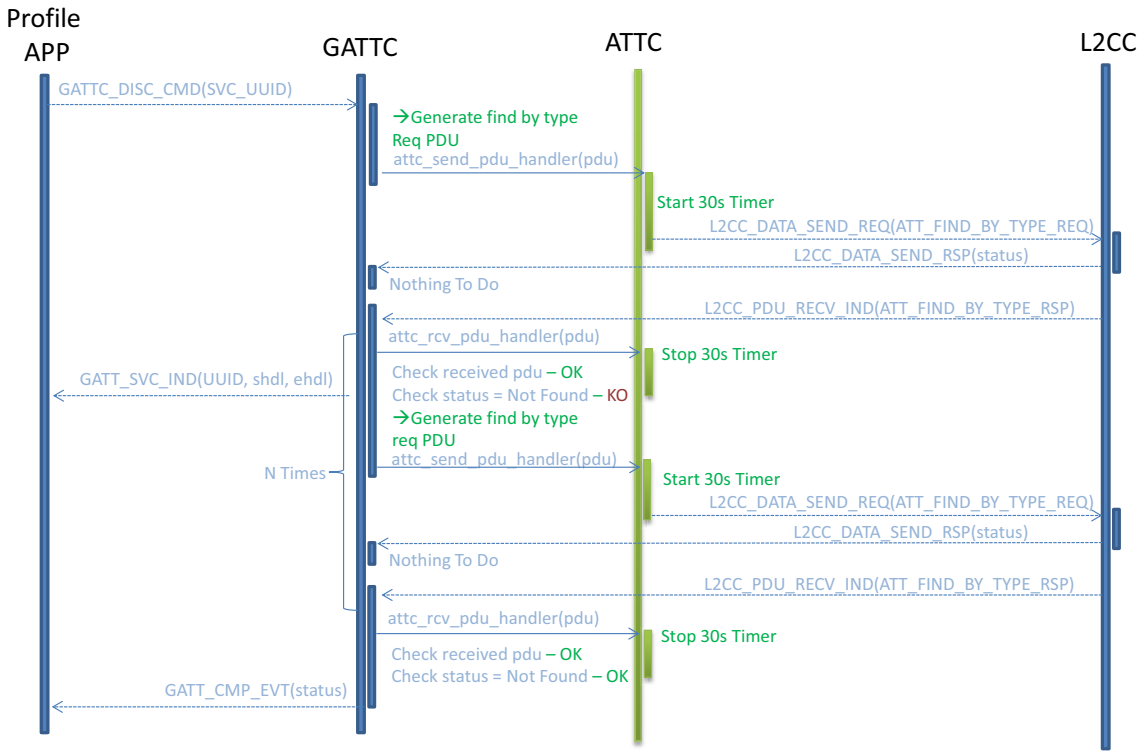


Figure 72. Discover Peer Services with Specific UUID MSC

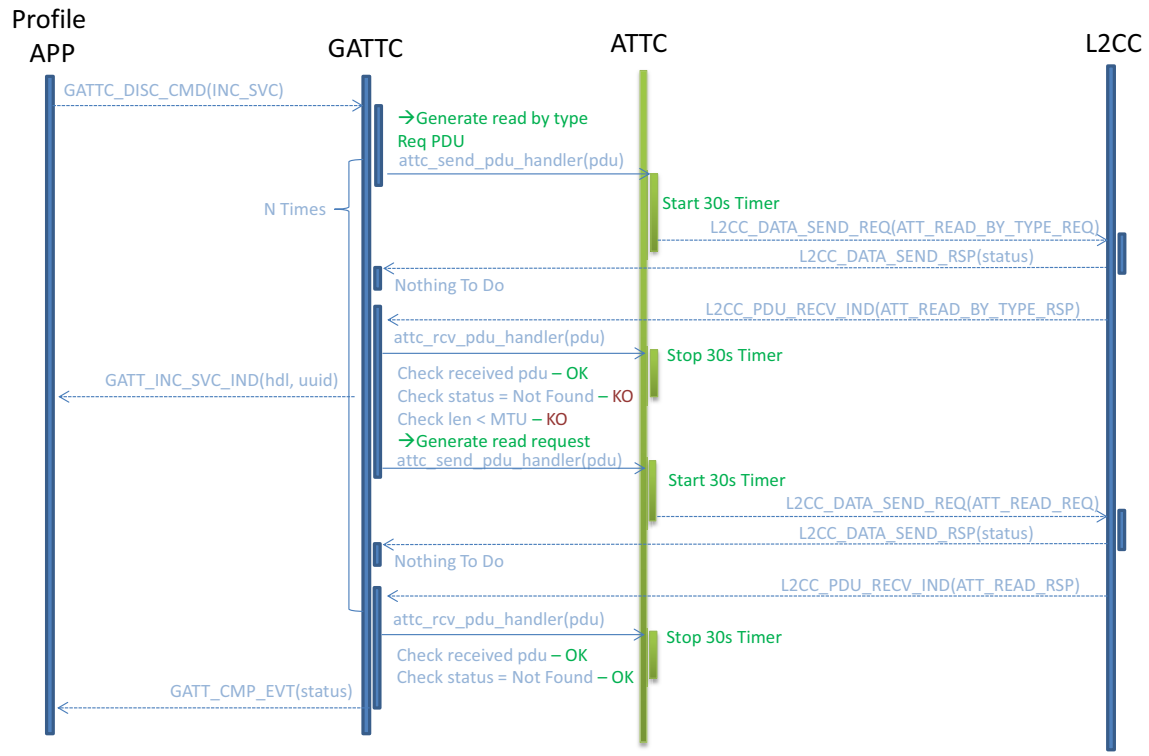


Figure 73. Discover Peer Included Services UUID MSC

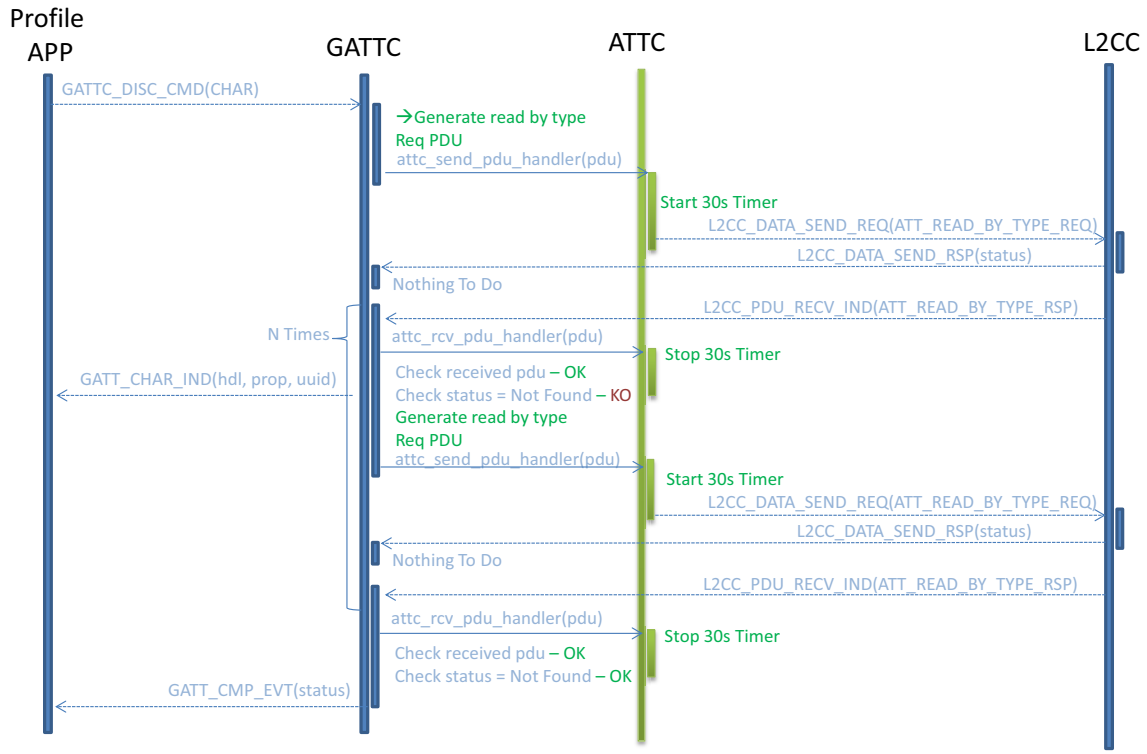


Figure 74. Discover Peer Characteristics (All or With Specific UUID) MSC

NOTE: The same procedure is used both when discovering all characteristics, and with a specific UUID. The filtering of the UUID is performed by the client side and not by the service side.

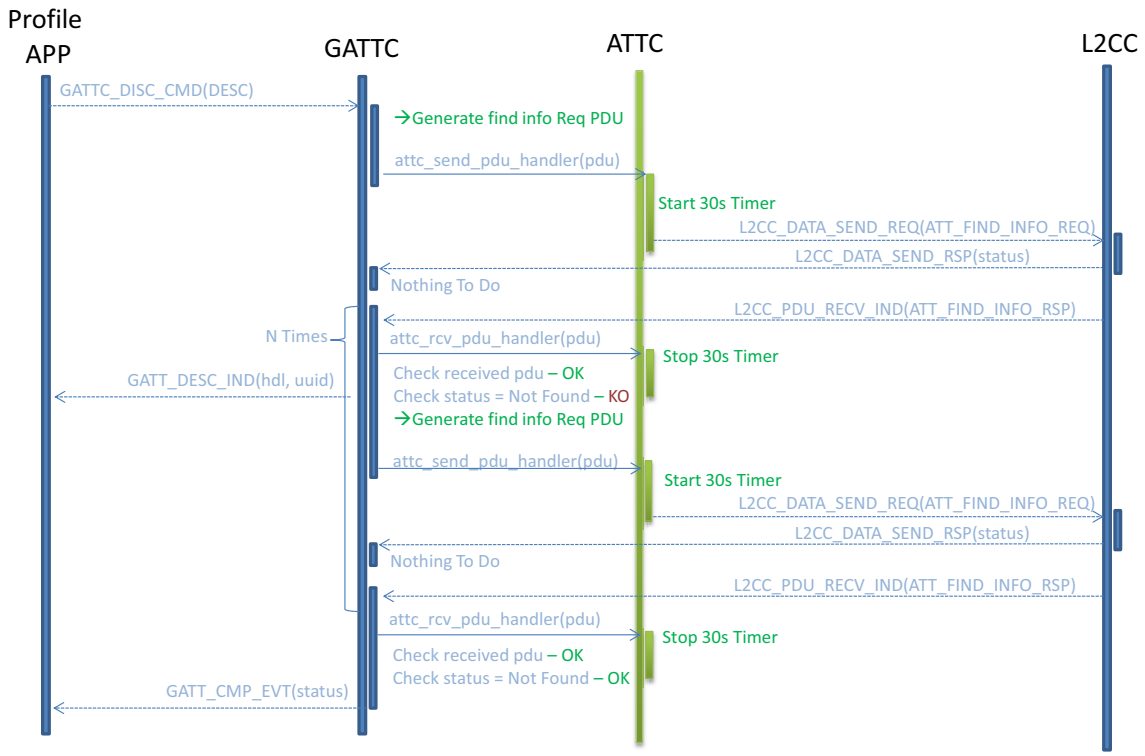


Figure 75. Discover Peer Descriptors MSC

6.3.2.7.2 Read Command

Read of a simple request, and read of a UUID request, are show in Figure 76 on page 106, and Figure 77 on page 107.

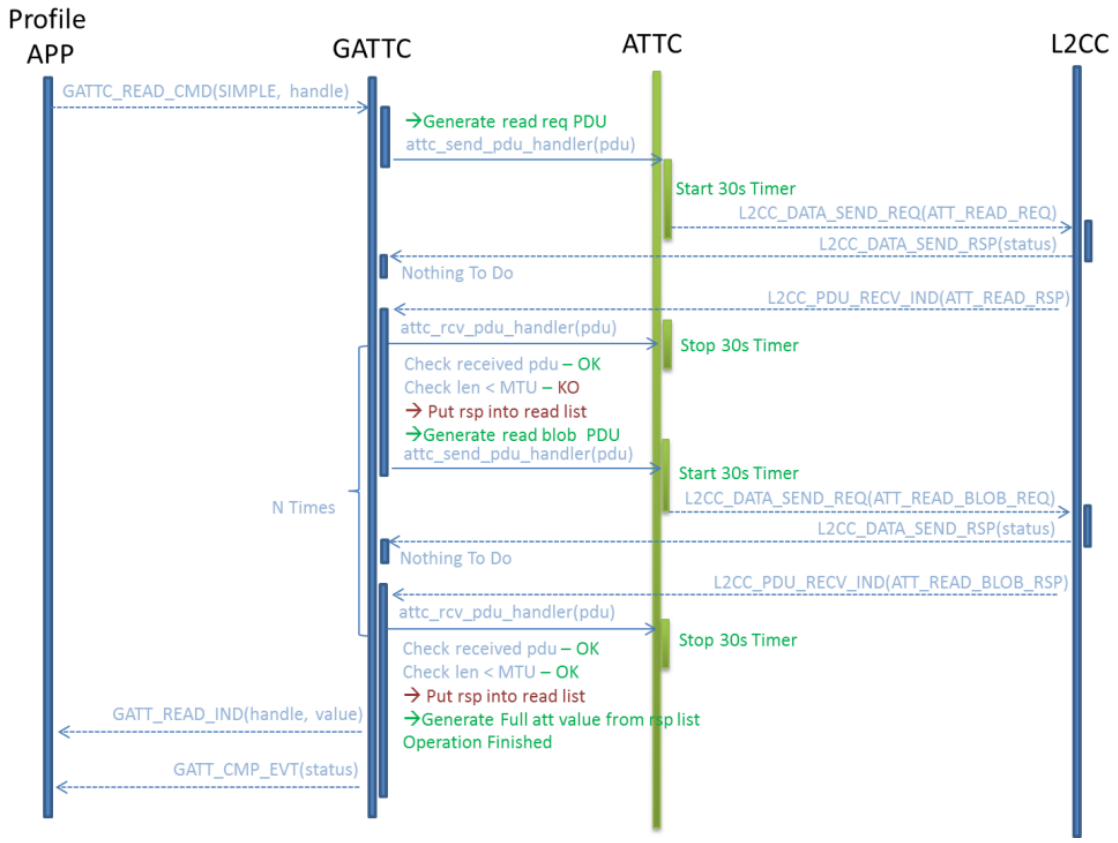


Figure 76. Read Simple Request MSC

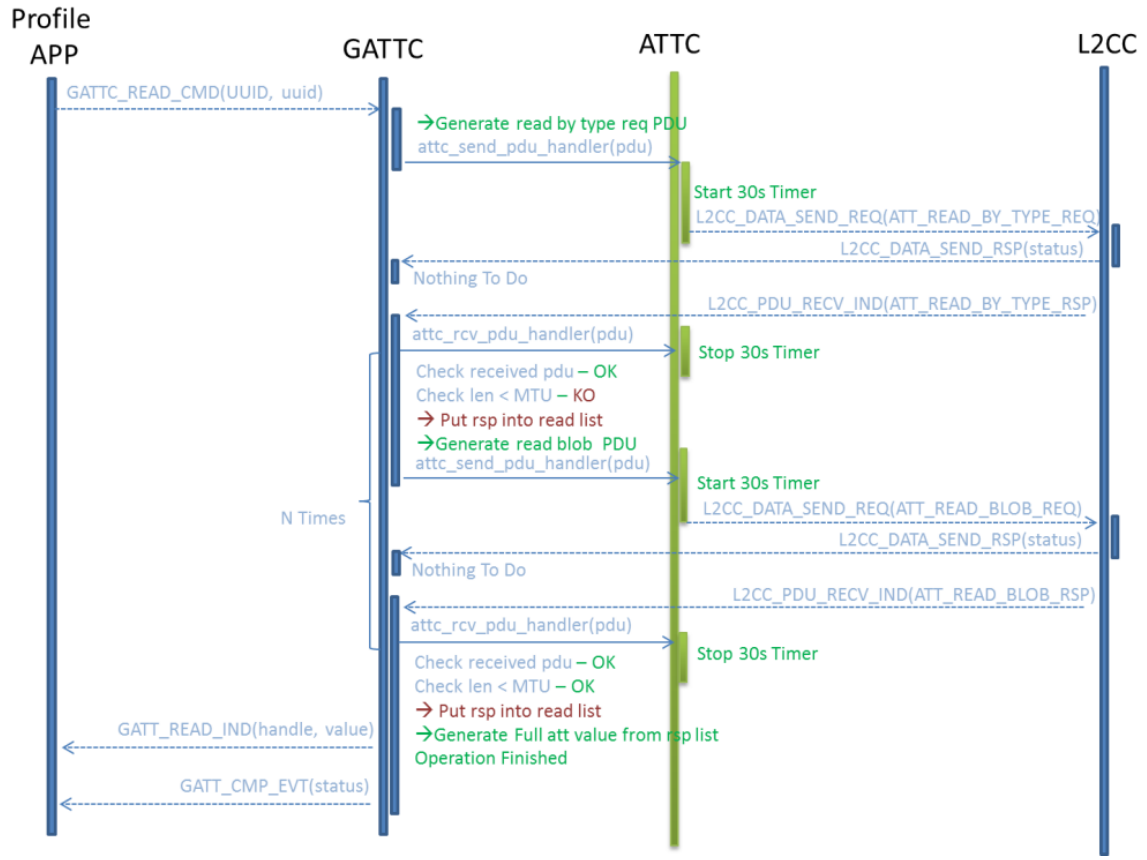


Figure 77. Read By UUID Request MSC

6.3.2.7.3 Write Command

The write command, write request, write of a long/multiple, and write signed are shown in Figure 78, Figure 79 on page 108, Figure 80 on page 109, and Figure 81 on page 110.

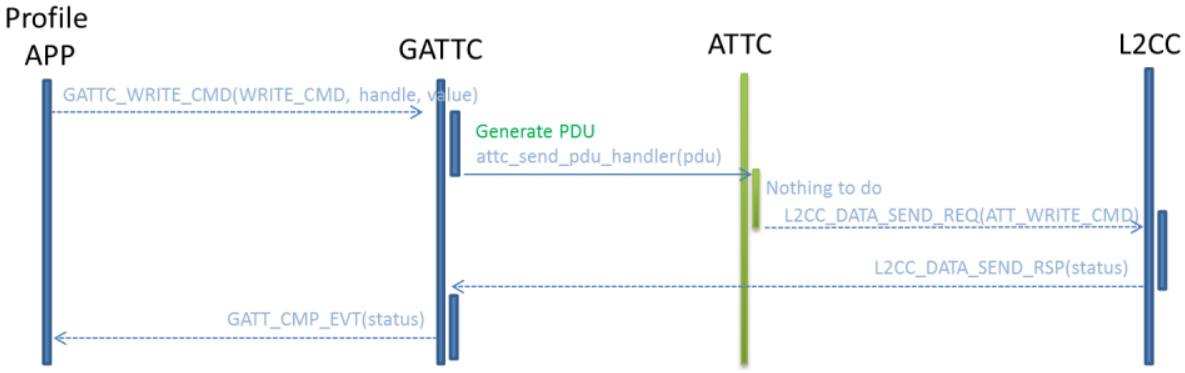


Figure 78. Write Command MSC

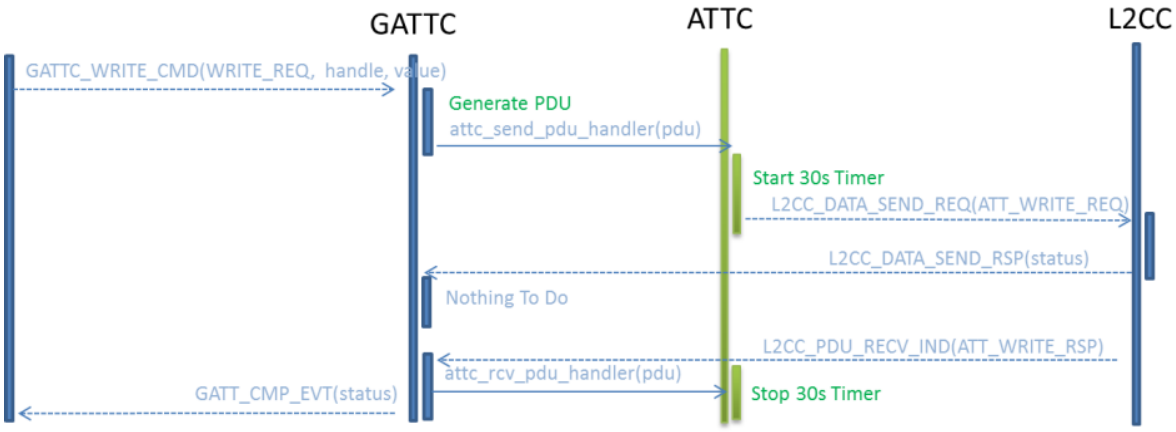


Figure 79. Write Request MSC

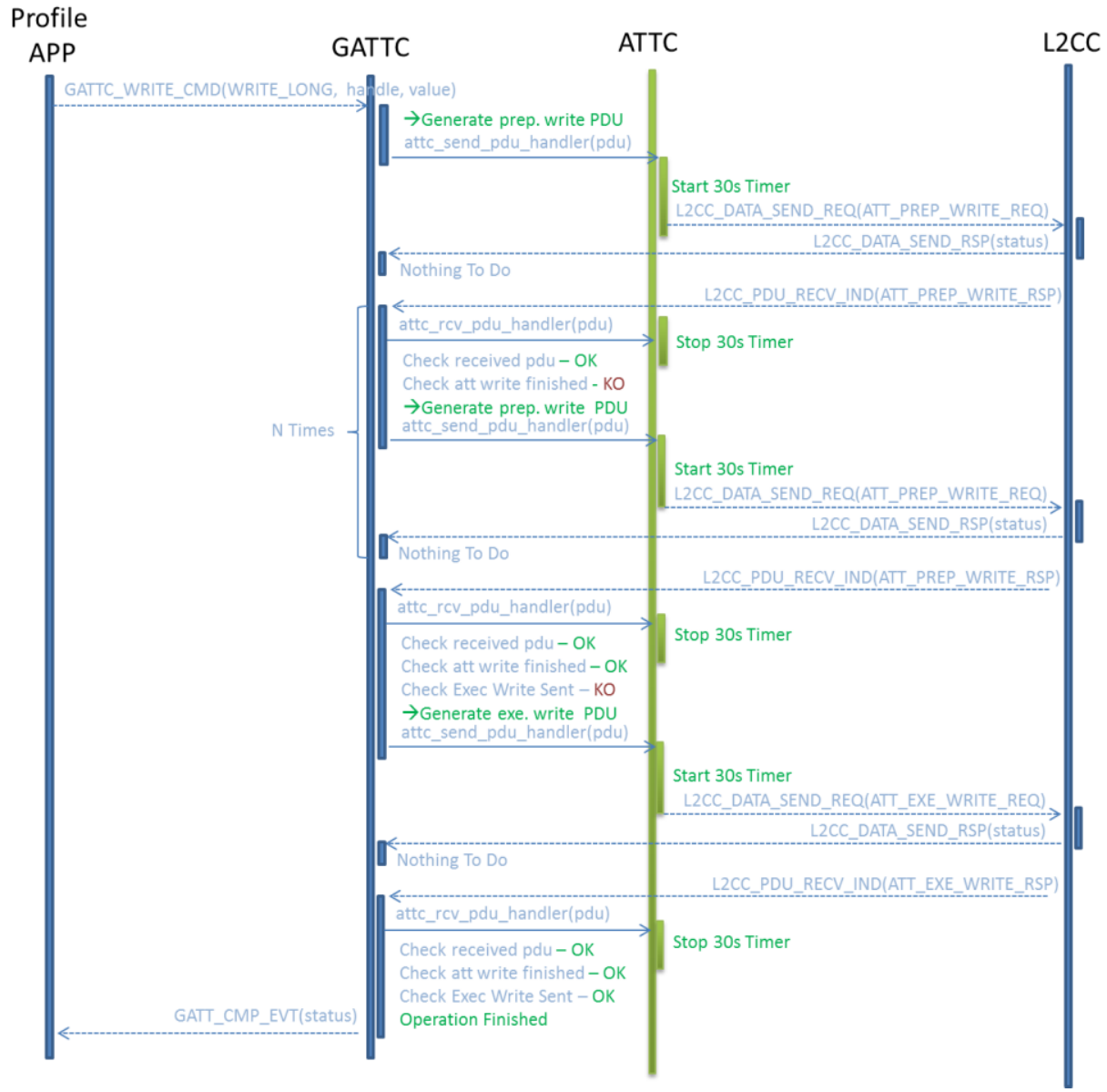


Figure 80. Write Long/Multiple MSC

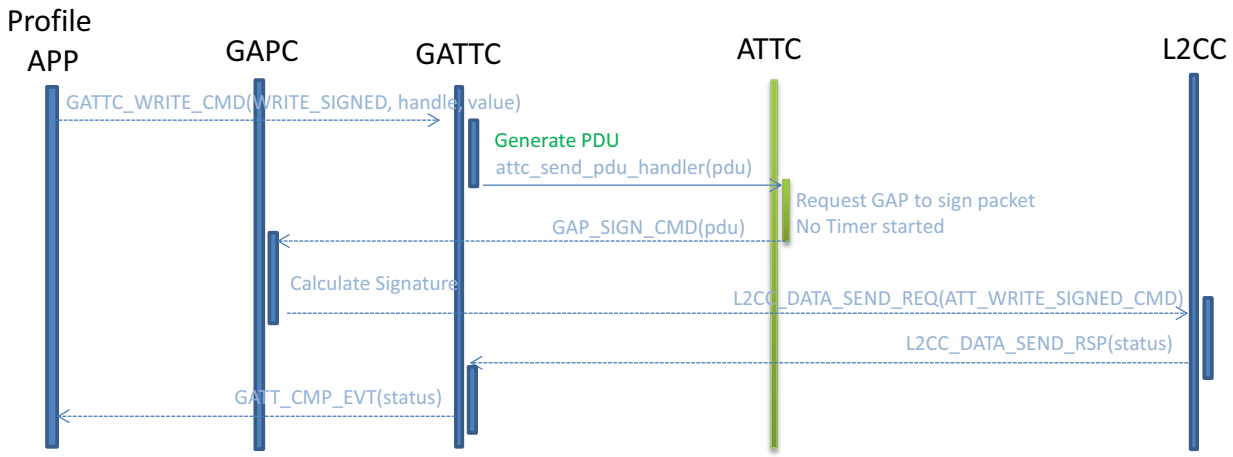


Figure 81. Write Signed MSC

6.3.2.7.4 Reception of Notification or Indications

To allow a profile to receive notification or indication of a peer device, the profile must be registered to service events. This can be accomplished with the provided peer service handle range (see Figure 82, Figure 83, and Figure 84 on page 111).

NOTE: By default the application is informed of any received events if no registration has been performed.

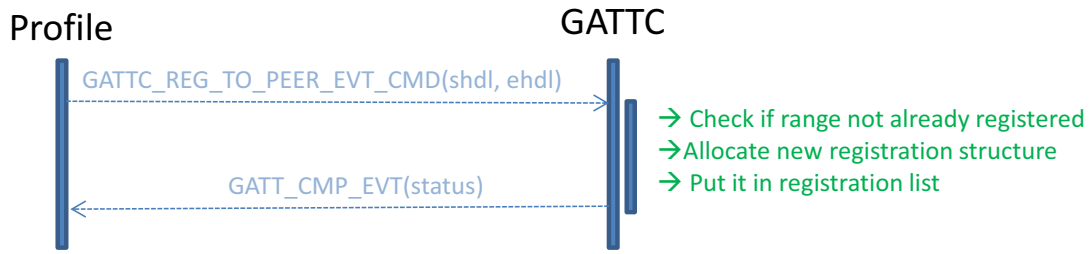


Figure 82. Event Handle Range Registration MSC

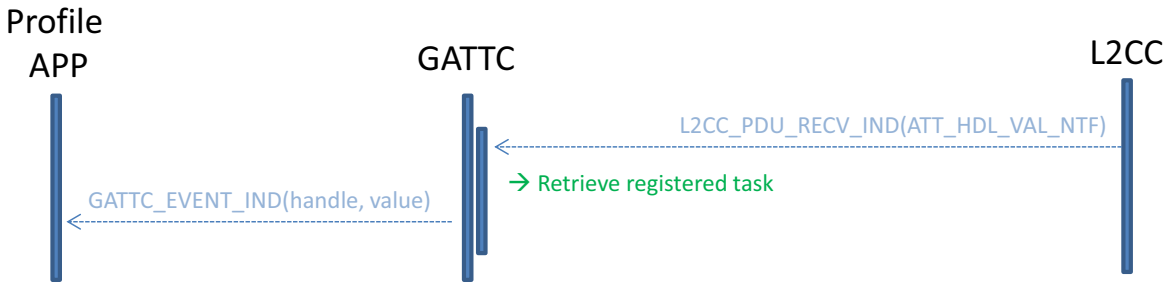


Figure 83. Reception of a Notification from Peer Device MSC

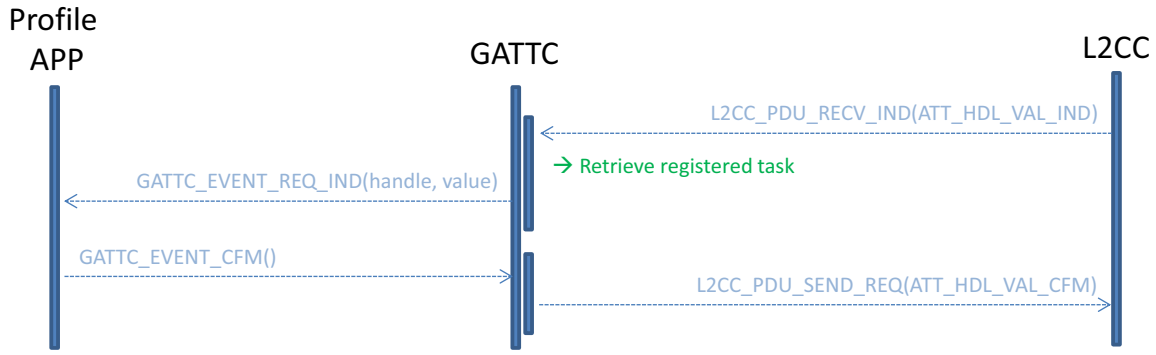


Figure 84. Reception of an Indication from Peer Device MSC

6.3.3 Features and Functions

6.3.3.1 Attribute Packet Size Negotiation

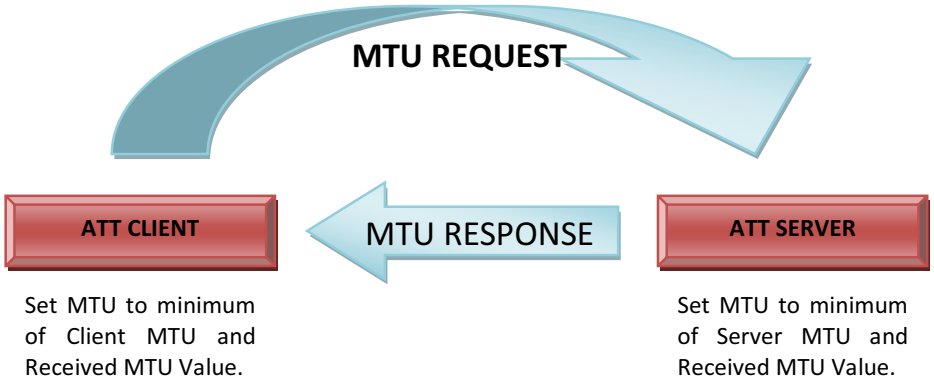


Figure 85. MTU Exchange Procedure

The MTU exchange procedure, shown above in Figure 85, is a sub-procedure of the server configuration. This is launched by the attribute client to configure the attribute protocol. At the end of the exchanges, both the attribute client and server will have a common set MTU, which is the minimum value exchanged.

6.3.3.2 Primary Service Discovery

The primary service discovery procedure is used by an attribute client to discovery primary services on a server. Once these services are discovered, additional information like characteristics and secondary services can be retrieved. There are two sub-procedures for primary service discovery, as shown in Table 44 on page 112.

Table 44. Primary Service Discovery Sub-Procedures

Sub-Procedure	ATT Operation Code

Discover All Primary Services	`Read By Group Type Request`
	`Read By Group Type Response`
	`Error Response`

Discover Primary Services By Service UUID	`Find By Type Request`
	`Find By Type Response`
	`Error Response`

The “Discover All Primary Services” sub-procedure is used by the client to discover all the primary services on a server. The “Discover Primary Services by Service UUID” sub-procedure is used by the client to discover a specific primary service on a server when only the service UUID is identified. The functions are completed in two ways: either the application receives an error code (`Attribute Not Found`) or the application cancels the search (in case the desired primary service is already found).`Insufficient Authentication` errors and `Read Not Permitted` errors shall not occur (service declaration is readable and requires no authentication or authorization).

6.3.3.3 Relationship Discovery

This procedure is used by an attribute client to discover service relationships to other services.

Table 45. Relationship Discovery Sub-Procedure

Sub-Procedure	ATT Operation Code
Find Included Services	`Read By Type Request`
	`Read By Type Response`
	`Error Response`

The “Find Included Services” sub-procedure is used by the client to find include service declarations in the attribute server database, as shown in Table 45. The function is completed in two ways: either the application receives an error code (`Attribute Not Found`) or the read by type response has enough unused data to contain another result indicating that no further results exist.`Insufficient Authentication` errors and `Read Not Permitted` errors shall not occur (Include declaration is readable and requires no authentication or authorization).

6.3.3.4 Characteristic Discovery

The characteristic discovery procedure is used by an attribute client to discover service characteristics present on the attribute server (see Table 46 on page 112).

Table 46. Characteristic Discovery Sub-Procedures

Sub-Procedure	ATT Operation Code
Discover All Characteristic of a Service	`Read By Type Request`
	`Read By Type Response`
	`Error Response`

Table 46. Characteristic Discovery Sub-Procedures

Sub-Procedure	ATT Operation Code
Discover Characteristic by UUID	`Read By Type Request`
	`Read By Type Response`
	`Error Response`

The “Discover All Characteristic of a Service” sub-procedure is used to find all the characteristic declarations within a service definition on a server, when only the service handle range is known. The “Discover Characteristic by UUID” sub-procedure is used to discover service characteristics on the attribute server, when only the service handle range and characteristic UUID are known. The functions are completed in two ways: either the application receives an error code (`Attribute Not Found`) or the application cancels the search (in case the desired characteristic is already found). I`nsufficient Authentication` errors and `Read Not Permitted` errors shall not occur (characteristic declaration is readable and requires no authentication or authorization).

6.3.3.5 Characteristic Descriptor Discovery

The characteristic descriptor discovery procedure is used by an attribute client to discover the characteristic descriptors of a characteristic.

Table 47. Characteristic Descriptor Discovery Sub-Procedure

Sub-Procedure	ATT Operation Code
Discover All Characteristic Descriptors	`Find Information Request`
	`Find Information Response`
	`Error Response`

The “Discover All Characteristic Descriptors” sub-procedure is used by a client to find all the attribute handles and types of the characteristic descriptor within the characteristic definition, and only when the handle range is known. (See Table 47, above.) The function is completed in two ways: either the application receives an error code (`Attribute Not Found`) or the application cancels the search (in case the desired characteristic descriptor is already found).

6.3.3.6 Characteristic Value Read

The characteristic value read procedure is used by an attribute client to read a characteristic value from a server. See Table 48 on page 113.

Table 48. Characteristic Value Read Sub-Procedures

Sub-Procedure	ATT Operation Code
Read Characteristic Value	`Read Request`
	`Read Response`
	`Error Response`
Read Using Characteristic UUID	`Read By Type Request`
	`Read By Type Response`
	`Error Response`

Table 48. Characteristic Value Read Sub-Procedures

Sub-Procedure	ATT Operation Code
Read Long Characteristic Values	`Read Blob Request`
	`Read Blob Response`
	`Error Response`
Read Multiple Characteristic Value	`Read Multiple Request`
	`Read Multiple Response`
	`Error Response`

The “Read Characteristic Value” sub-procedure is used to read a characteristic value from a server when the client knows the Characteristic Value Handle. The “Read Using Characteristic UUID” sub-procedure is used to read a Characteristic Value from a server when the client only knows the characteristic UUID and does not know the handle of the characteristic. The “Read Long Characteristic values” sub-procedure is used to read a characteristic value from a server when the client knows the Characteristic Value Handle, and the length of the characteristic value is longer than can be sent in a single read response attribute protocol message. The “Read Multiple Characteristic values” sub-procedure is used to read multiple characteristic values from an attribute server when the client knows the Characteristic Value Handles.

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

6.3.3.7 Characteristic Value Write

The characteristic value write procedure is used by the client to write a characteristic value to an attribute server. See Table 49 on page 114.

Table 49. Characteristic Value Write Sub-Procedures

Sub-Procedure	ATT Operation Code
Write Without Response	`Write Command`
Signed Write Without Response	`Write Command`
Write Characteristic Value	`Write Request`
	`Write Response`
	`Error Response`
Write Long Characteristic Values	`Prepare Write Request`
	`Prepare Write Response`
	`Execute Write Request`
	`Execute Write Response`
	`Error Response`

Table 49. Characteristic Value Write Sub-Procedures (Continued)

Sub-Procedure	ATT Operation Code
Characteristic Value Reliable Writes	Prepare Write Request
	Prepare Write Response
	Execute Write Request
	Execute Write Response
	Error Response

The “Write Without Response” sub-procedure is used to write a Characteristic value to a server when the client knows the Characteristic Value Handle and the client does not need an acknowledgement that the write was successfully done. The “Signed Write without Response” sub-procedure is used to write a Characteristic value to a server when the client knows the Characteristic Value Handle and the ATT Bearer is not encrypted. The “Write Characteristic Value” sub-procedure is used to write a Characteristic value to a server when the client knows the Characteristic Value Handle. The “Write Long Characteristic Values” sub-procedure is used to write a Characteristic value to a server when the client knows the Characteristic Value Handle, but the length of the Characteristic value is longer than can be sent in a single write request attribute protocol message. The “Characteristic Value Reliable Writes” sub-procedure is used to write a characteristic value to an attribute server when the client knows the Characteristic Value Handle, and assurance is required that the correct characteristic value is going to be written by transferring the characteristic value to be written in both directions before the write is performed.

6.3.3.8 Characteristic Value Notification

The characteristic value notification procedure is used to notify a client of the value of a characteristic value from a server, as shown in Table 50.

Table 50. Characteristic Value Notification Sub-Procedure

Sub-Procedure	ATT Operation Code
Notifications	Handle Value Notification

The “Notifications” sub-procedure is used when a server is configured to notify a characteristic value to a client without expecting any attribute protocol layer acknowledgement that the notification was successfully received.

6.3.3.9 Characteristic Value Indication

The characteristic value indication procedure is used to indicate the characteristic value from a server to a client as shown in Table 51.

Table 51. Characteristic Value Indication Sub-Procedure

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

The “Indications” sub-procedure is used when a server is configured to indicate a characteristic value to a client and expects an attribute protocol layer acknowledgement that the indication was successfully received.

6.3.3.10 Characteristic Descriptor Value Read

The characteristic descriptor value read procedure is used to read a characteristic descriptor on a server, as shown in Table 52.

Table 52. Characteristic Descriptor Value Read Sub-Procedures

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

The “Read Characteristic Descriptor Value Read” sub-procedure is to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic declaration. The “Read Long Characteristic Descriptors” sub-procedure is used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration, and the length of the characteristic value is more than will fit in a single read response attribute protocol message.

6.3.3.11 Characteristic Descriptor Value Write

The characteristic descriptor value write procedure is used to write a characteristic descriptor on a server, as shown in Table 53.

Table 53. Characteristic Descriptor Value Write Sub-Procedures

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

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

The “Write Long Characteristic Descriptors” sub-procedure is used to write a characteristic descriptor value to a server when the clients knows the characteristic descriptor handle of the characteristic descriptor declaration, and the length of the characteristic value is more than will fit in a single write response attribute protocol message.

6.3.4 Service Discovery Procedure

The service discovery must be a generic feature used by client profiles to discover a peer device database, illustrated in Figure 86 on page 117. By using a generic method of service discovery, it prevents code duplication in

client profiles. This discovery will be able to be performed for all services types, or for only some of them. With this feature, an application can decide if discovery is performed by the client profiles or by the application itself.

NOTE: The client profile verifies whether the peer device service can be used by its implementation.

For each discovered service, this procedure is in charge of finding included services, characteristics and descriptors. (See Figure 87 on page 118.) When a full service is discovered, this operation triggers a message containing all the information.

NOTE: Since this procedure can be very long, it can be aborted by the application through a `Cancel` API.



Figure 86. Service Discovery Procedure

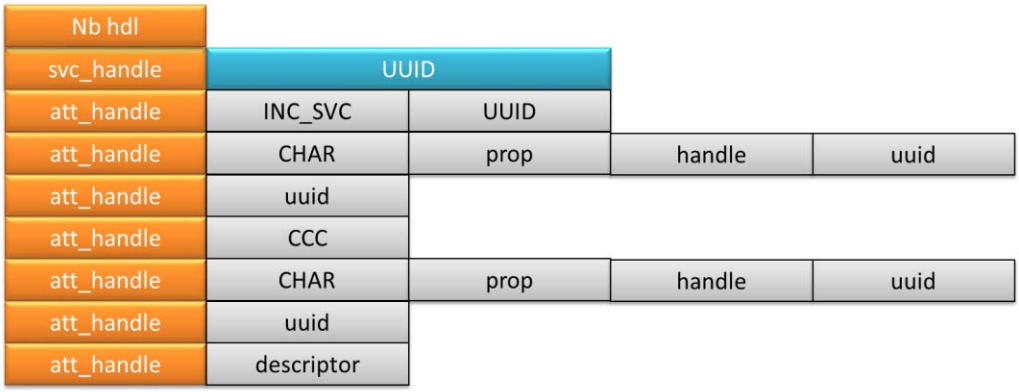


Figure 87. Overview of Information Present in Discovered Service

6.3.5 GATT Profile Service

The GATT profile service, shown in Figure 88 below, is a single-instantiated primary service which is exposed on a GATT server. The profile service has a service changed characteristic.

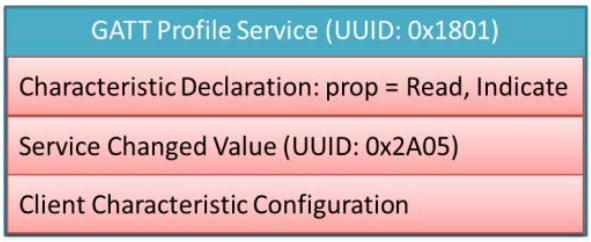


Figure 88. GATT Profile Service

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

6.3.6 GATT Environment Variables

Table 54 and Table 55 explain the environment variables associated with both the GATT manager and controller respectively. By accessing these values, you can make modifications to the structure as needed.

6.3.6.1 GATT Manager Environment

GATT Manager environment variables are shown in Table 54, below.

Table 54. GATTM Environment Variables

Type	Value	Comment
`uint16_t`	`svc_start_hdl`	GATT service start handle
`uint16_t`	`max_mtu`	Maximum device MTU size

Table 54. GATTM Environment Variables

Type	Value	Comment
attm_svc_db*	db	Attribute database pointer
attm_svc_db*	last_svc	Last attribute service searched

6.3.6.2 GATT Controller Environment

GATT Controller environment variables are shown in Table 55 on page 119.

Table 55. GATTC Environment Variables

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

6.4 GAP FUNCTIONALITY

This profile states the requirements on names, values and coding schemes used for names of parameters and procedures experienced on the user interface level. This profile describes the general procedures that can be used for establishing connections to other Bluetooth low energy technology devices that are able to accept connections and service requests.

GAP defines two parties (A and B) in establishing Bluetooth low energy technology communication:

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

The GAP allows minimal functionality in absence of other profiles and provides an API when other profiles are present.

6.4.1 Modes and Profile Roles

GAP introduces three device types based on supported Core Configurations, as shown below in Figure 89.

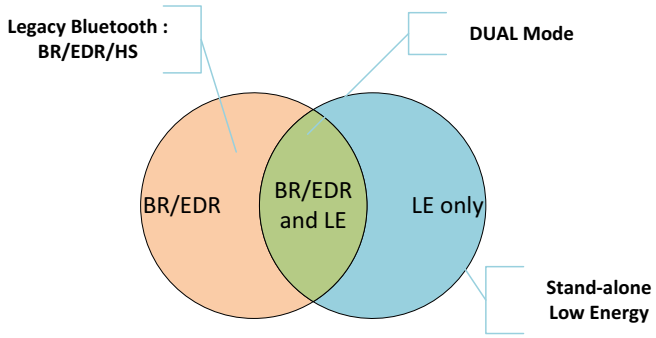


Figure 89. Devices Types

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

NOTE: Our implementation of GAP supports only LE only mode.

Moreover, GAP defines different modes of operation which are generic and can be used by profiles and by devices implementing multiple profiles (see Table 56 on page 120).

Table 56. Discoverability and Connectability Modes to Advertising Capability

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

In addition to functions shown in Figure 90, a peripheral is able to broadcast data, and a central is able to enter in observable mode. A device can support all roles at the same time, so that it can act both as a central (scan + master of a link) and peripheral (advertise + slave of a link).

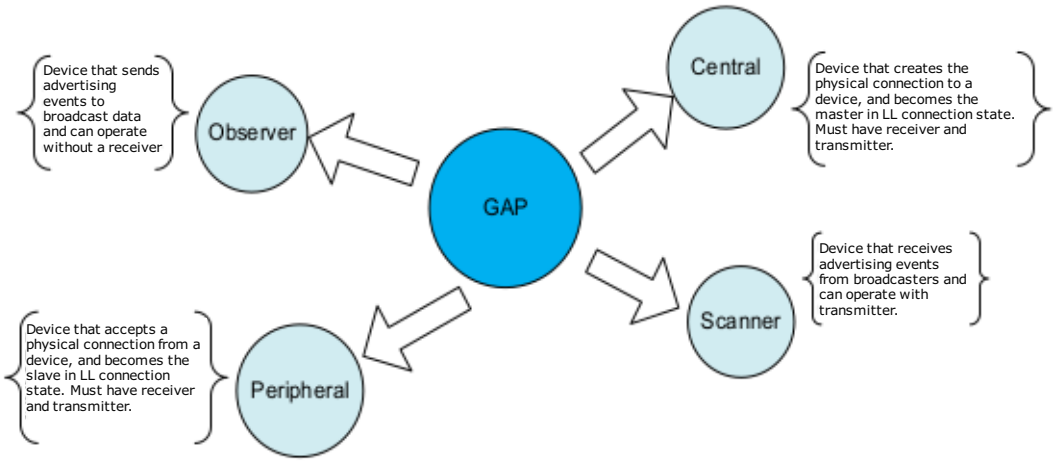


Figure 90. GAP Roles

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

6.4.2 General LE Procedures

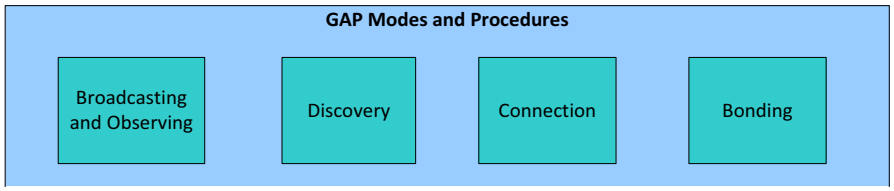


Figure 91. LE Operational Modes

GAP defines the general procedures that can be used for discovering identities, names, and basic capabilities of other Bluetooth low energy technology devices that are discoverable. It also describes the ability of a device to be connected and discovered by another device. See Figure 91 on page 121 for low energy operational modes.

6.4.2.1 Broadcasting and Observing

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

6.4.2.1.1 Conditions

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

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

6.4.2.2 Advertising Modes

A device can perform an advertising procedure in a connectable or non-connectable mode. A whitelist can be used to filter a device that can receive scan responses and initiate a connection. See Figure 92 on page 122.

NOTE: When this operation is on-going, an application can modify advertise and scan response data to update ongoing broadcast data.

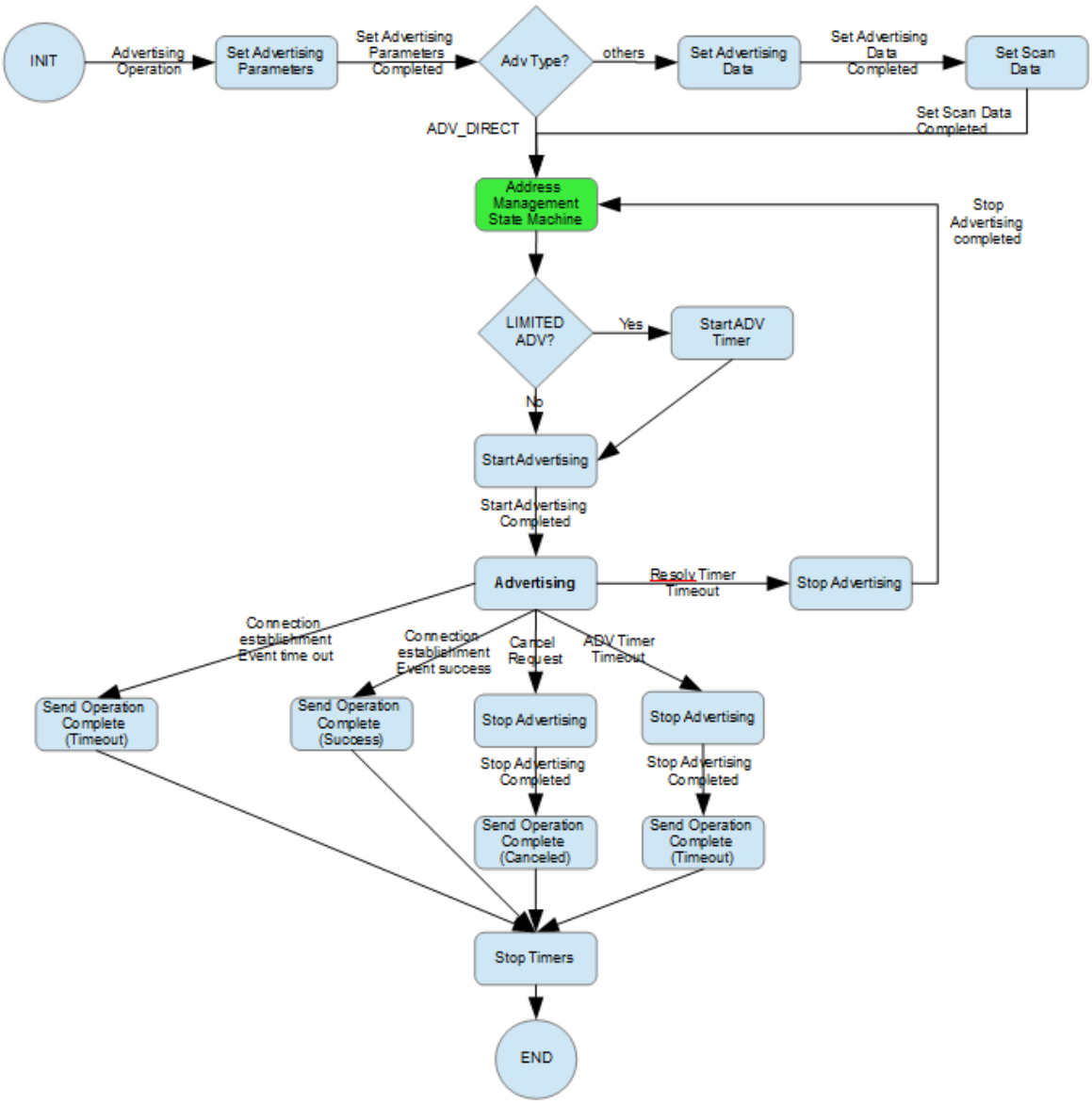


Figure 92. Advertise Air Operation State Machine

6.4.2.2.1 Broadcast Mode

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

NOTE: This is the only mode that can be used by a broadcaster device.

6.4.2.2.2 Non-Discoverable Mode

Non-discoverable mode is a connectable or non-connectable procedure without duration limitation. In the AD_TYPE flag of advertising data, LE General and the LE Limited Discoverable flag are set to zero.

6.4.2.2.3 General Discoverable

General discoverable mode is a connectable or non-connectable procedure without duration limitation. In the AD_TYPE flag of advertising data, LE General is set to 1 and the LE Limited Discoverable flag is set to zero.

6.4.2.2.4 Limited Discoverable

Limited discoverable mode is a connectable or non-connectable procedure with a limited duration. In the AD_TYPE flag of advertising data, LE General is set to zero and the LE Limited discoverable flag is set to 1.

6.4.2.2.5 Direct Mode

Direct mode is used to perform a direct connection. Advertising data contains only the targeted device. Advertising data cannot be dynamically changed in this mode.

6.4.2.3 Scan Modes

6.4.2.3.1 Device Discovery

The device discovery has two parts: procedures and modes. (See Figure 93 on page 124) A device that is searching for other devices performs one of the discovery procedures. A device that is the target of the search is operating in one of the discoverable modes. A device in the non-discoverable mode is configured to not be discovered. All devices are in either non-discoverable mode or one of the discoverable modes (general and limited). A typical example of a device that need not be in discoverable mode is an observer. A device that operates in an observer profile role requires no transmitter.

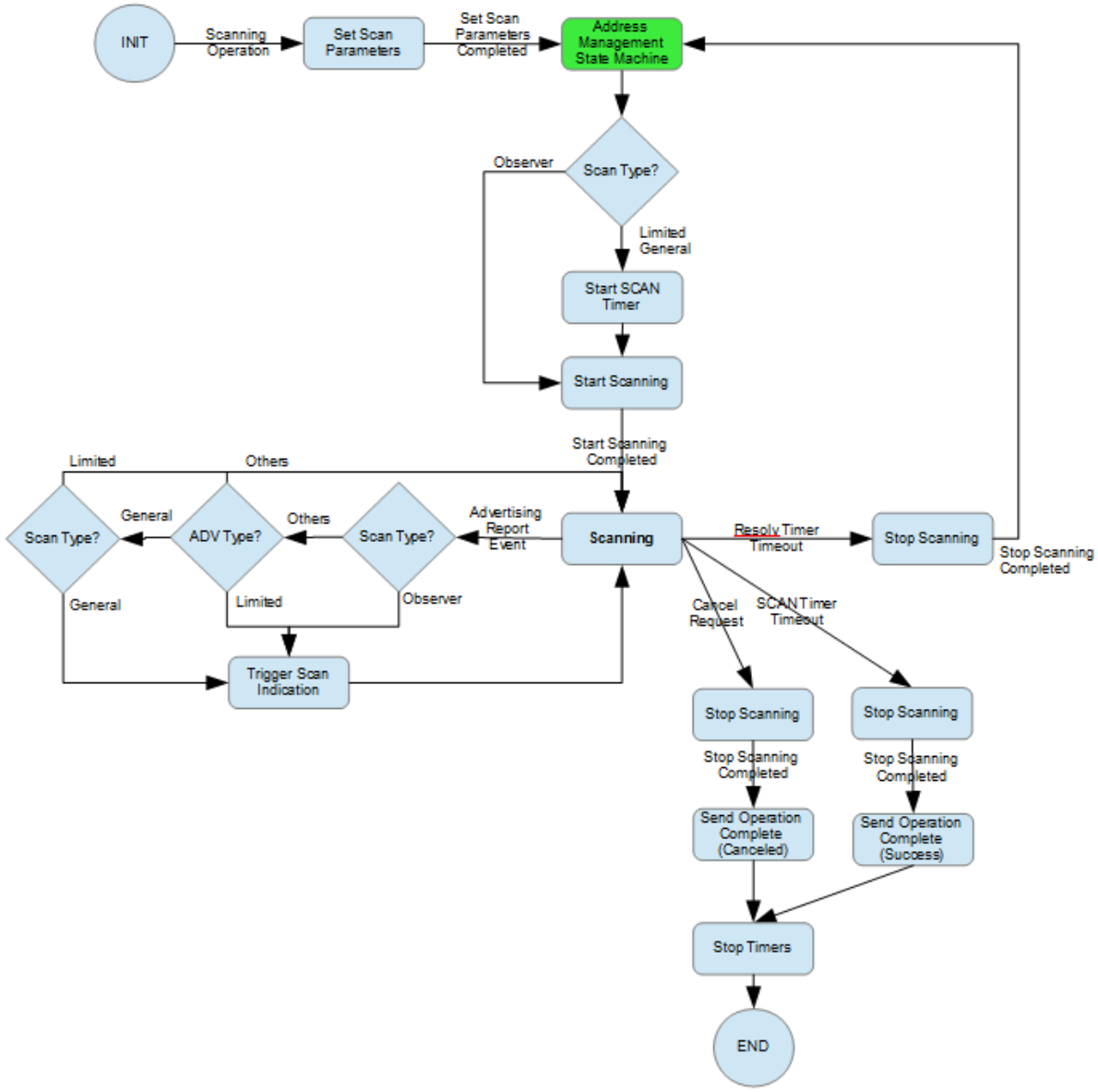


Figure 93. Scan Air Operation State Machine

6.4.2.3.2 Observer Mode

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

NOTE: This is the only mode that can be used by an observer device.

6.4.2.3.3 General Discovery

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

6.4.2.3.4 Limited Discovery

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

6.4.2.3.5 Name Discovery

Another aspect of discoverability is device name discovery, wherein the user-friendly name of the remote device is retrieved. This is performed by a device that can scan remote connectable devices – a central (illustrated in Figure 94). The discovery procedure involves three fundamental steps:

1. Search and connect to a connectable device (advertising device).
2. Perform read by characteristic UUID (Device Name: `0x2A00`).
3. Terminate the link.

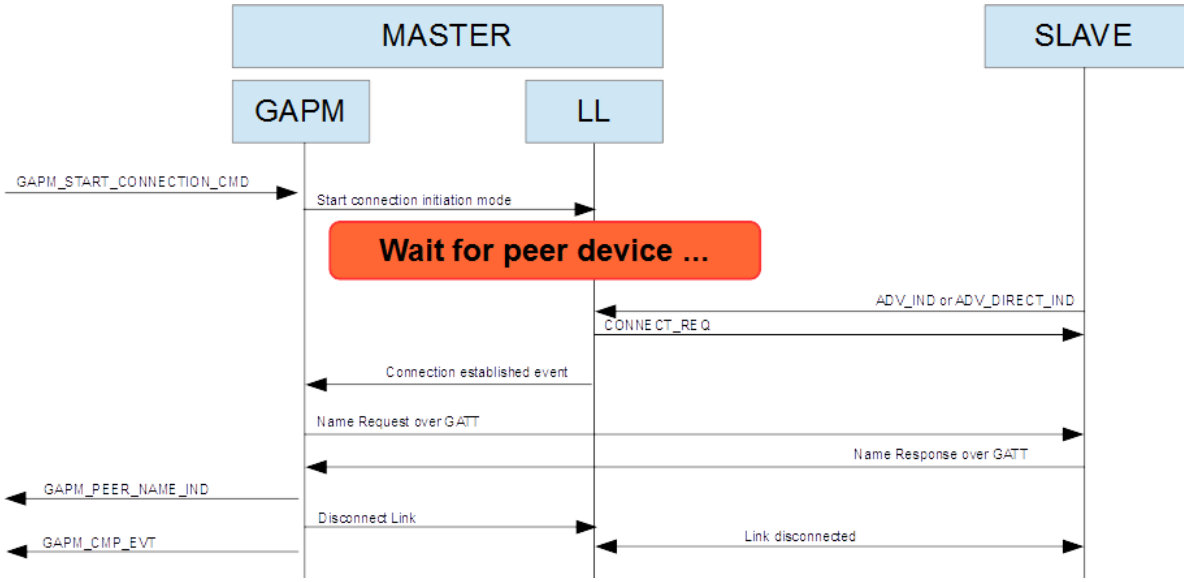


Figure 94. Name Request Procedure

6.4.2.4 Connection

There are two modes for LE connections defined in GAP (see Figure 95).

- Connectable: permits a device to make connections to or accept connections from another device.
- Non-connectable: prohibits a device from accepting connections from another device.

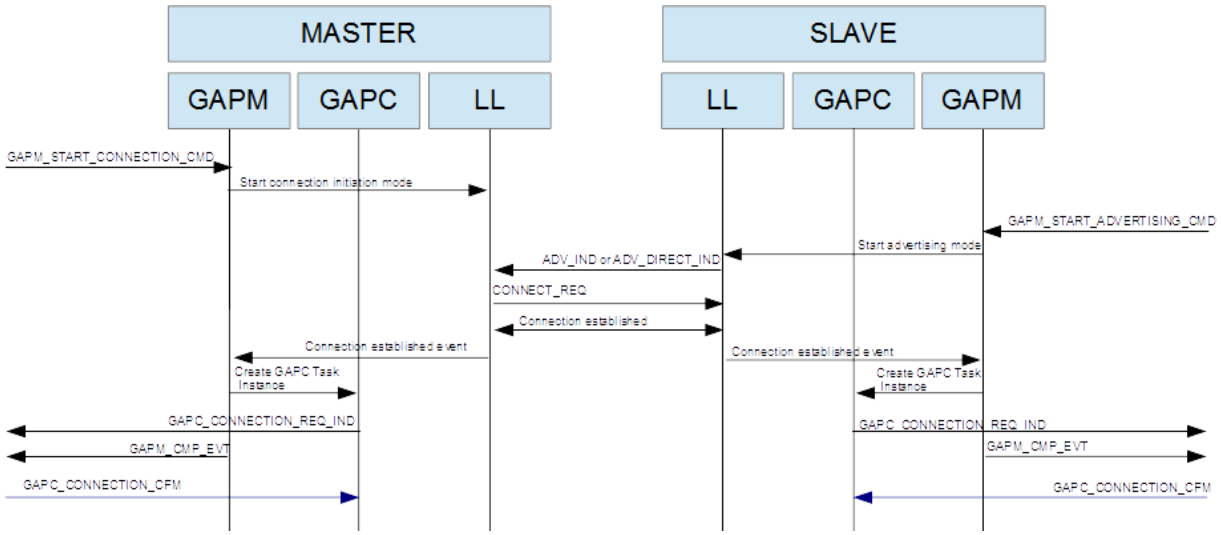


Figure 95. Connection Establishment Overview

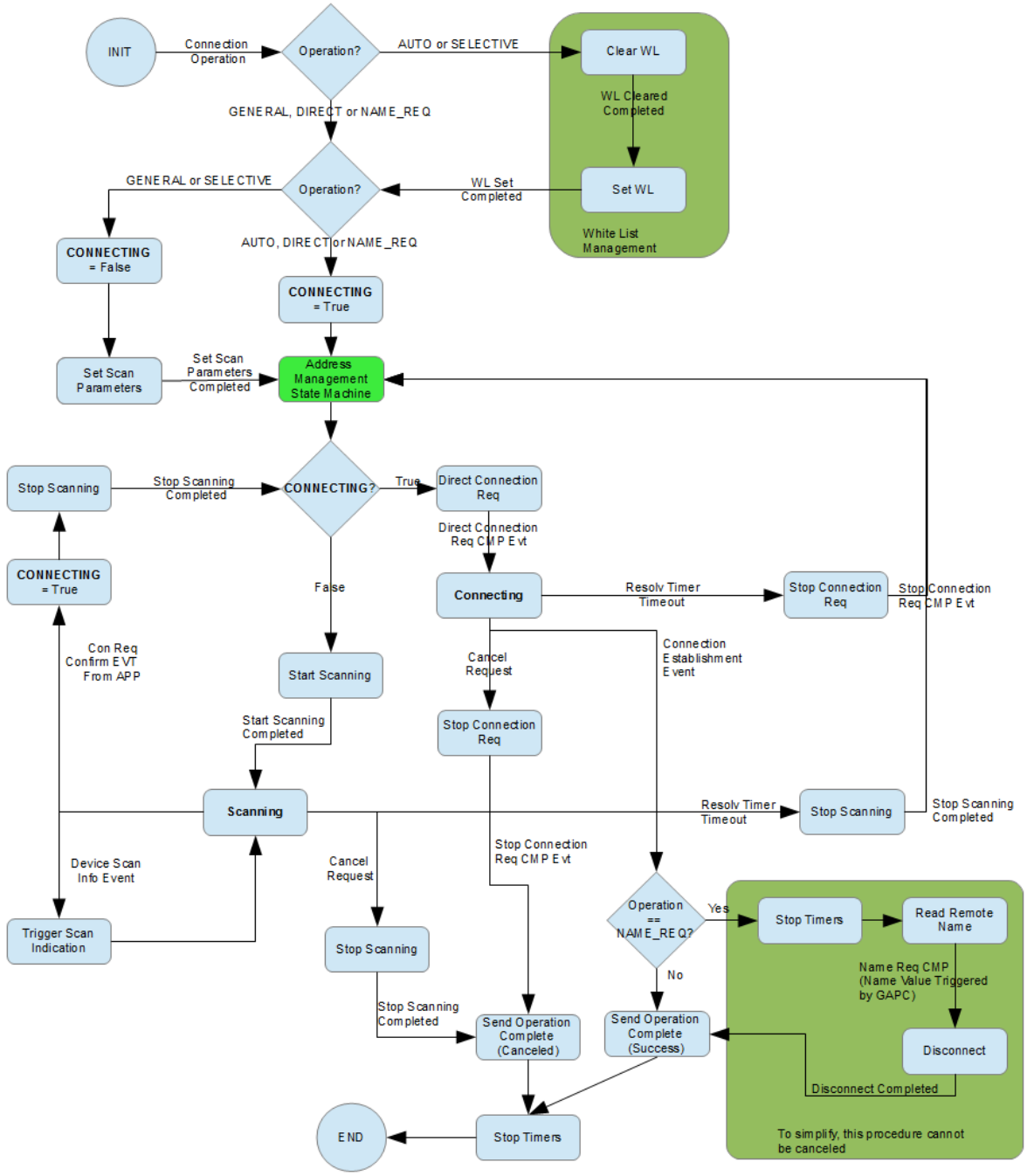


Figure 96. Connection Establishment State Machine

6.4.2.4.1 Direct Connection Establishment

To be able to establish a link between two devices, one device must be in connectable mode, and the other device would be performing the connection establishment procedure, as shown in Figure 96.

6.4.2.4.2 General Connection Establishment

Use the general discovery procedure and then the direct connection establishment procedure to perform a general connection establishment, as shown in Figure 97.

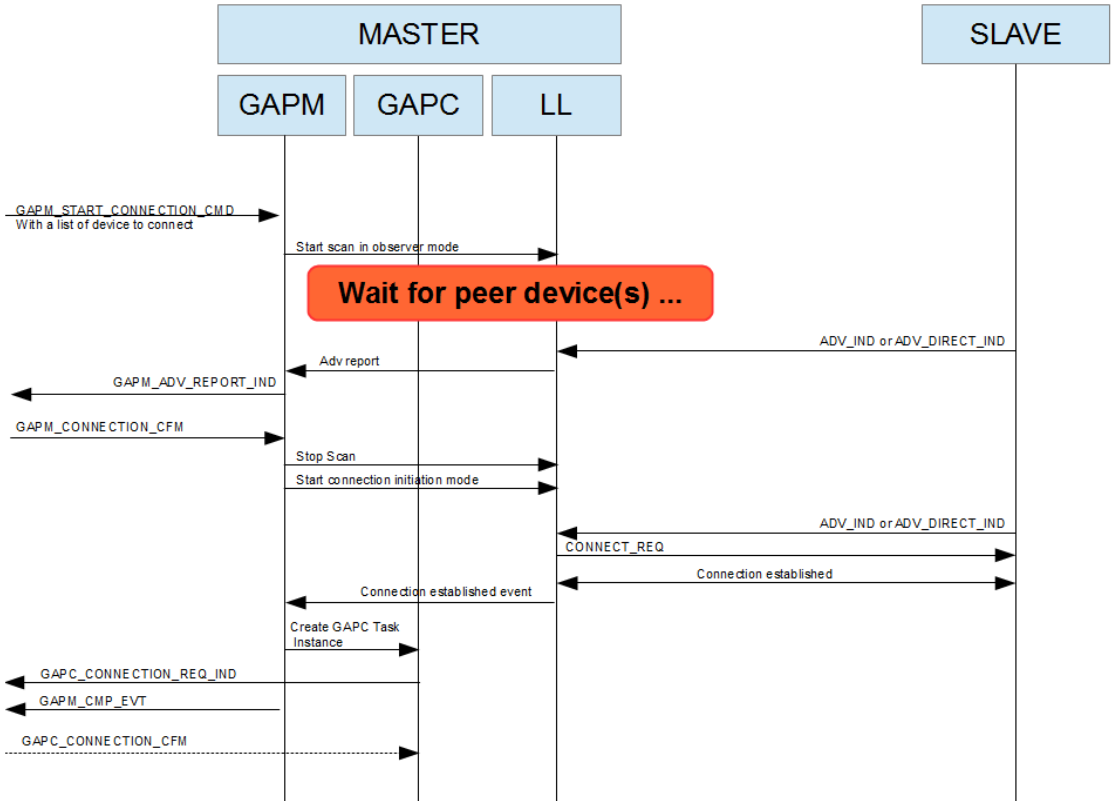


Figure 97. General Connection Procedure

6.4.2.4.3 Automatic Connection Establishment

The automatic connection establishment procedure uses a whitelist in connection mode to find any known device. As soon as a known device is found, it uses a direct connection to connect to the peer device, as shown in Figure 98 on page 129.

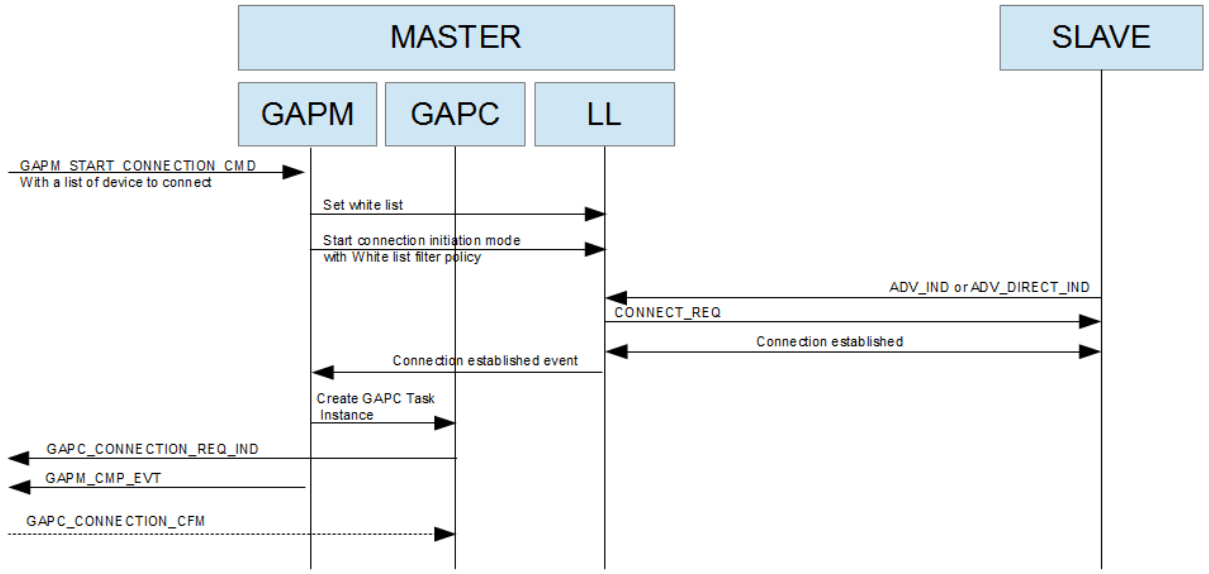


Figure 98. Automatic Connection Procedure

NOTE: When device is in this mode, it is not possible to modify the whitelist.

6.4.2.4.4 Selective Connection Establishment

The automatic connection establishment procedure uses a whitelist and observer mode to find any known device. As soon as a known device is found, the application is notified, and must reply with some connection parameters to use with this device. Finally, it uses a direct connection to connect to the peer device. (See Figure 99 on page 130.)

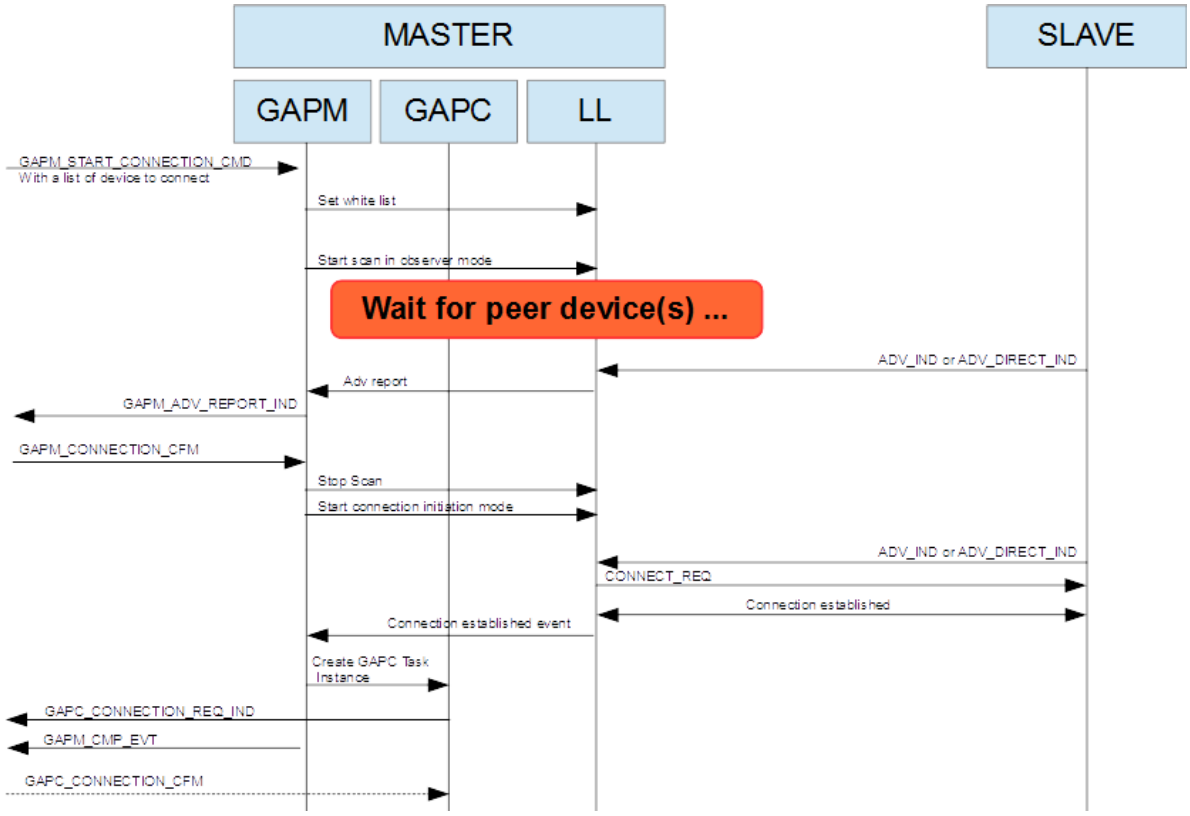


Figure 99. Selective Connection Procedure

NOTE: When device is in this mode, it is not possible to modify the whitelist.

6.4.2.4.5 Update Connection Parameters

The parameter update procedure, an operation that can be started over an LE link, is used to update link parameters. If the operation is initiated by a master on a 4.0 (Legacy) device (see Figure 100 on page 131), there is no link negotiation, and new link parameters are automatically applied.

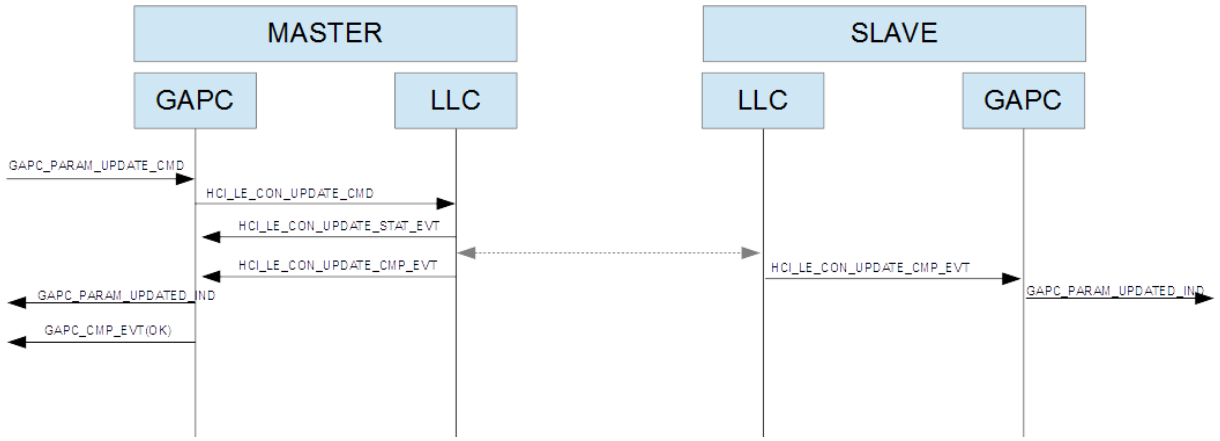


Figure 100. Parameter Update Initiated by Master

When a slave initiates a connection parameter update without knowing the remote features, a parameter update must first be started through the HCI. If it fails due to any of the following reasons, legacy negotiation over L2CAP must be used instead (see Figure 101 on page 132):

- `Unknown HCI Command`
- `Command Disallowed`
- `Unsupported Command`
- `Unknown LMP PDU`
- `Unsupported Remote Feature`
- `LMP PDU Not Allowed`

NOTE: Operation completion messages for a legacy parameter update initiated by a slave (on a slave device) have to be triggered before `GAPM_PARAM_UPDATE_IND`, to ensure that the master device did not use the connection param request.

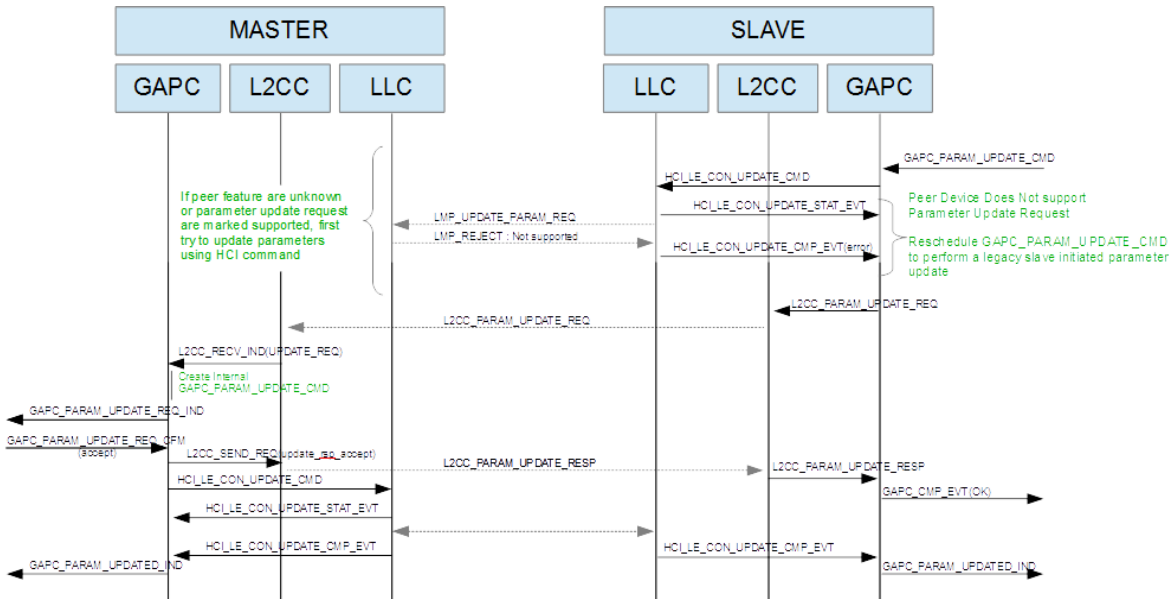


Figure 101. Legacy Parameter Update Initiated By Slave

Figure 102 shows legacy parameter update negotiation initiated by a slave and rejected by a master.

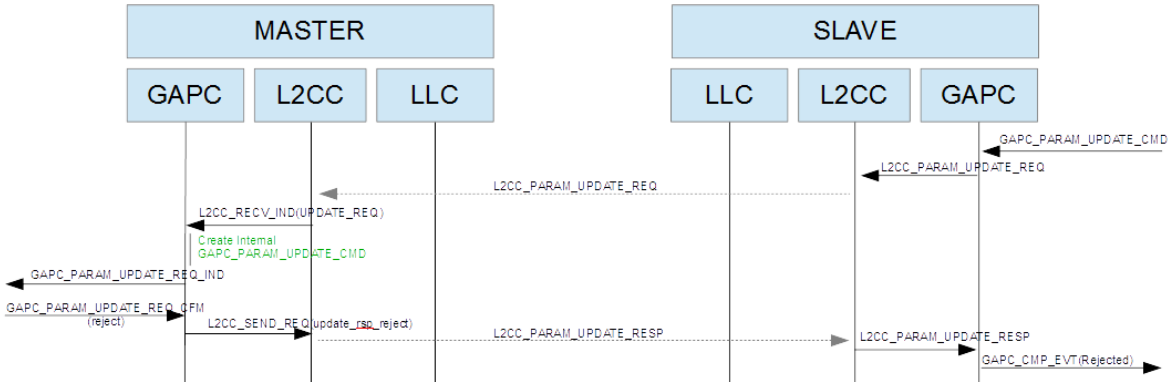


Figure 102. Legacy Parameter Update Initiated By Slave, Rejected By Master

If a parameter update request is supported by a peer device, the connection parameter initiated by a master or a slave is a little bit different. It is requested for the peer device to accept or reject new parameters. This parameter update is only performed through LLCP even if it is initiated by a slave or a master of the link. Figure 103 on page 133 shows a parameter update with remote update request support accepted by the responder. In Figure 104 on page 133, the update is rejected by the responder.

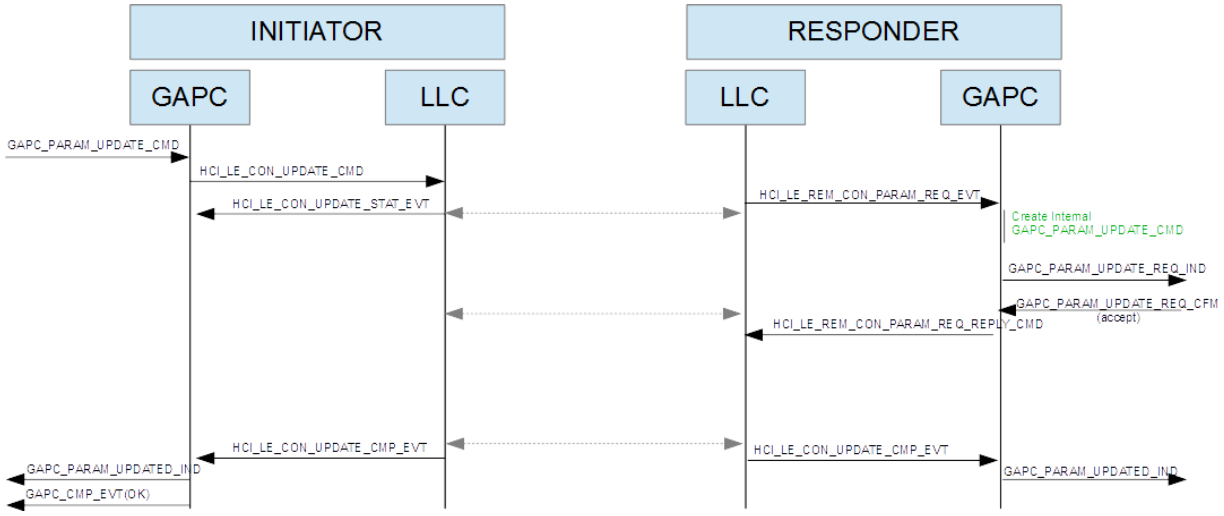


Figure 103. Parameter Update with Remote Update Request Support Accepted by Responder

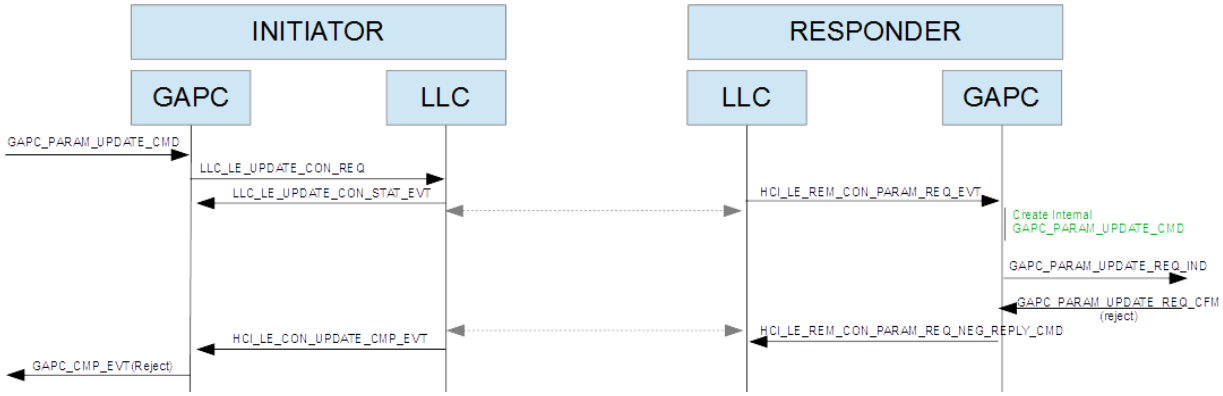


Figure 104. Parameter Update with Remote Update Request Support Rejected by Responder

6.4.2.5 Bonding

Bonding is the function where devices exchange and store security and identity information to create a secure relationship. It occurs at the first connection between devices or the first service that requires security or authorization. (See Figure 105 on page 134, and Figure 106 on page 134.) Two types of bonding procedures are defined:

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

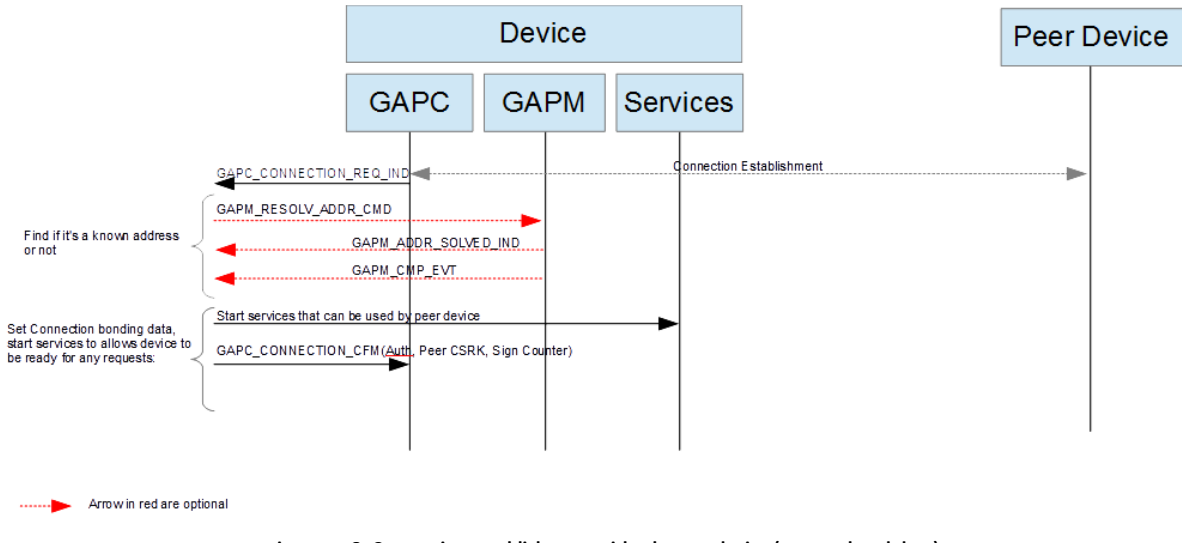


Figure 105. Connection Establishment With a Known Device (Recover Bond Data)

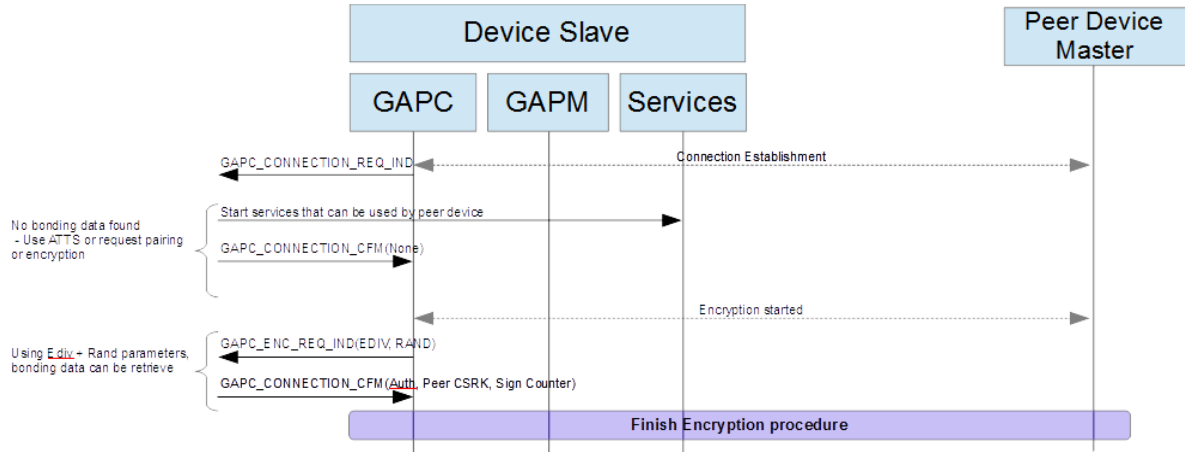


Figure 106. Recover Bond Data of a Peer Device with a Random Address

6.4.3 Low Energy Security

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

6.4.3.1 Security Modes

The LE security is expressed in modes and levels. There are two security modes: Sec 1 (encryption) and Sec 2 (signing), as shown in Figure 107 on page 135.

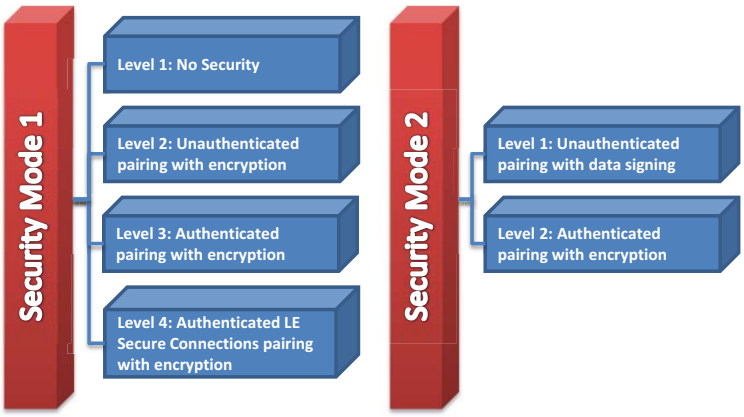


Figure 107. LE Security Modes

6.4.3.2 Authentication Procedure

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

There are two types of pairing:

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

6.4.3.3 Authorization Procedure

The authorization procedure allows the continuation of service access by a remote device. This is a confirmation by the user for continuance of the procedure. Authorization might be granted after successful authentication.

6.4.3.4 Data Signing

The data signing procedure is used to transfer authenticated data between two devices in an unencrypted connection, as shown in Figure 108. This is used by services that require fast connection setup and data transfer. If data signing is used, security mode 2 is a must.

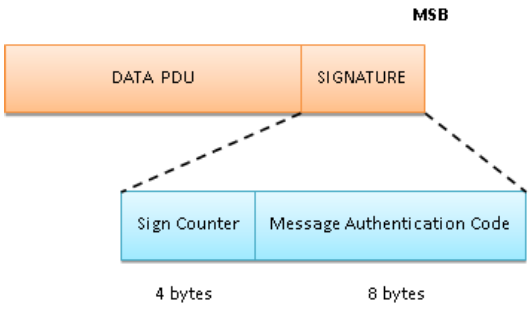


Figure 108. Packet Signature

6.4.3.5 Privacy

6.4.3.5.1 Host Managed Privacy (1.1)

The host managed privacy feature provides a specific level of security from attackers, to keep them from tracking an LE device over a certain period of time. This is an optional feature for all GAP roles. (See Table 57.)

Table 57. Device address type according to privacy configuration

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

NOTE: For passive scans, the Privacy feature is ignored

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

6.4.3.5.2 Controller Managed Privacy (1.2)

With controller managed privacy, the application will set the resolving address list (RAL) using the `GAPM_RAL_MGMT_CMD` command. This resolving address list can be managed like a whitelist, and is used as a complement to the whitelist. When controller managed privacy is enabled, the scan, advertise and initiating parameters are set to use the resolving list. If this feature is enabled when setting device configuration (see Section 6.4.5.11.2, “Device Configuration” on page 174), then the central address resolution characteristic becomes present in the GAP service (see Section 6.4.5.9, “GAP service database” on page 172). (See Figure 109 on page 137, Figure 110 on page 137, Figure 111 on page 137, and Figure 112 on page 138.)

6.4.3.5.3 LE Address

There are two types of Bluetooth low energy addresses:

1. Static Address
   - Two most significant bits are equal to 1
   - All the other bits are neither “all 0s” nor “all 1s”

2. Private Address
   a. Non-resolvable Address
      - Two most significant bits are equal to 0
      - All the other bits are neither “all 0s” nor “all 1s”

   a. Resolvable Address
      - Two most significant bits are equal to 01
      - 22 remaining bits of prand are neither “all 0s” nor “all 1s”
      - 24-bit hash section is derived from IRK, prand and ah func

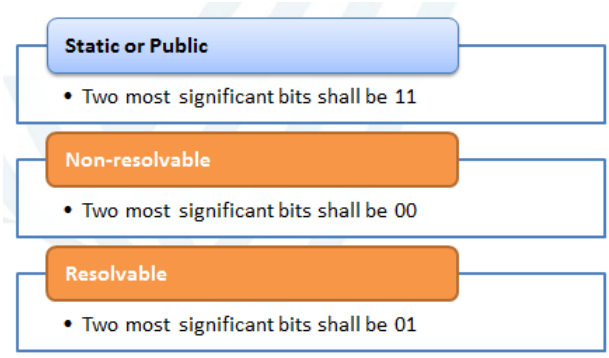


Figure 109. LE Address

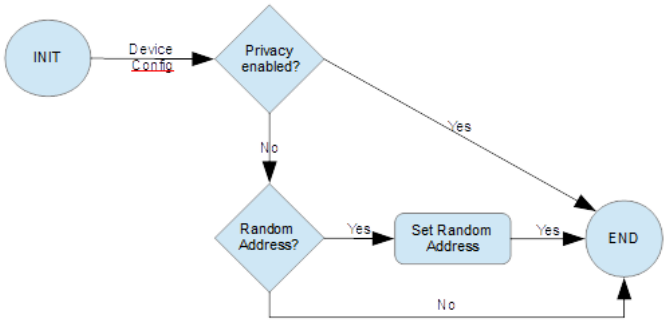


Figure 110. Initialize Device Address FW State Machine

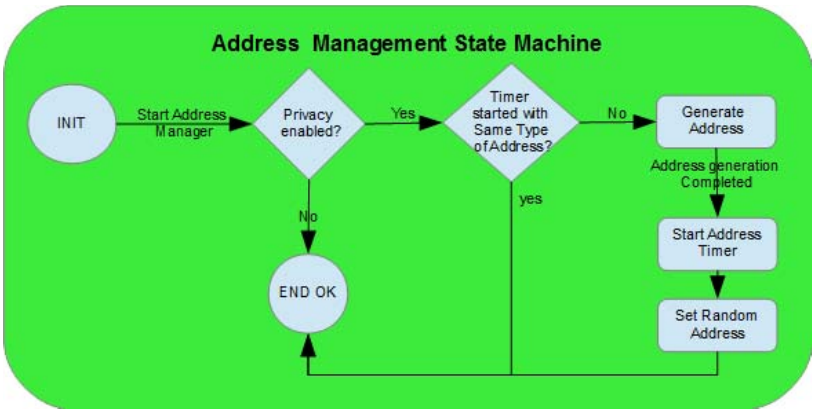


Figure 111. Air Operation Address Management FW State Machine

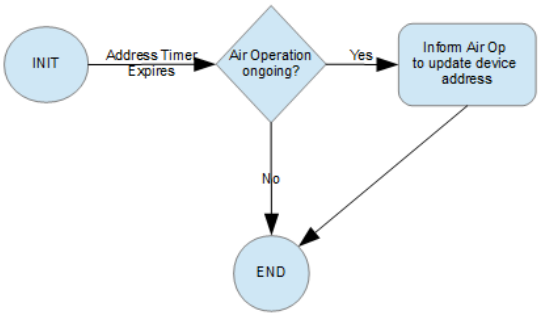


Figure 112. Privacy Address Management FW State Machine

6.4.4 Security Manager Toolbox

The Bluetooth low energy security manager allows two devices to set up a secure relationship, either by encrypting a link, by bonding (exchanging information about each other), or by signature use over a plain link. (See Figure 113, below.) Refer to the *Bluetooth Core Specification* v5.0 for the SM requirements and protocol methods.

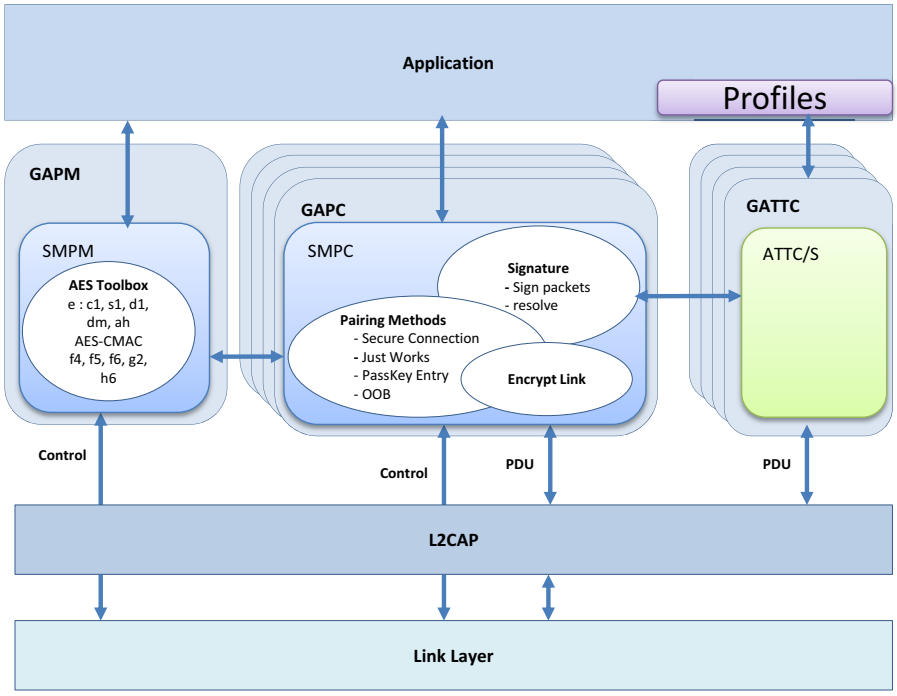


Figure 113. SMP Block Overview

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

- Pairing: this procedure allows two devices to agree upon features that will allow them to establish a certain level of security.

- Bonding: this procedure involves at least one device sending some sort of identification or security information to the other device, to be used in future connections. This can be an encryption key, signature key, or identification resolution key. If both devices are bondable, the transport key distribution phase following pairing will occur. Otherwise no bonding information will be exchanged, and if any is sent, it is a violation of protocol. Pairing might occur without necessarily bonding, but the features exchanged during pairing are essential to the existence of a bonding stage. If one of the devices is not bondable, no information about the peer should be stored (not even the BD address or other non-security related information).
- Unauthentication/authentication: unauthentication is NOT lack of any security, but an intermediary level between no security and the authenticated security level. The relationship between two devices is said to be (un)authenticated when the key(s) being used for their link encryption/signing/etc. have a security property that confirms (un)authentication. This security property is bestowed on a key during pairing, as a function of the STK method generation used. For both Passkey Entry and OOB Methods, all keys generated and exchanged afterwards have the authenticated (MITM) property (a pin key/ larger OOB key was used, which enforces security). If the Just Works method was used, all keys will have the unauthenticated (NO MITM) property. There can also be the no security property, which applies when the link is plain.
- LE secure connection: This pairing method allows a greater security level than the normal pairing method. It uses the private/public keys (P-256 elliptic curve) security algorithm to prevent any man-in-the-middle attack. This secure connection is a fully new pairing method that can be used for Just Works pairing, OOB or pin code entry. With this method, except Just Works pairing, the security level of the link is considered a “secure connection authenticated” link.

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

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

However, certain higher and lower layer modules have a unique instance, handled by the GAPM task through its API, SMPM toolbox – Security Manager Protocol Manager – which will monitor SMPC’s requests and responses without overloading those modules.

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

6.4.4.1 Keys Definition

There are several important types of keys in Bluetooth security, as shown in Table 58 below.

Table 58. Bluetooth Keys

Key Type	Description
Identity Root (IR)	• 128-bit key generated for LE device • Only for devices that support encryption or use random addresses • Device can have multiple IR keys, but will only use one per connection • Used to generate IRK and DHK
Encryption Root (ER)	• 128-bit random generated • Used to generate CSRK and LTK.
Identity Resolving Key (IRK)	• 128-bit key • Used to resolve random addresses
Diversifier Hiding Key (DHK)	• 128-bit key • Used to encrypt DIV during encryption connection setup
Connection Signature Resolving Key (CSRK)	• 128-bit key • Used to sign and verify signatures on the receiving device
Long Term Key (LTK)	• 128-bit key, used partially depending on agreed key size • Used to generate contributory session key for an encrypted connection
Diversifier (DIV)	• 128-bit stored value, used to calculate LTK • A new DIV is generated each time a unique LTK is distributed. • The DIV value is masked to the 2 octet EDIV distributed value.
Short Term Key (STK)	• Generated at the end of Phase 2 using TK • Used to encrypt link after Phase 2 (according to agreed key size)
Temporary Key (TK)	• Either 0, Pass Key or OOB depending on STK generation method • Used to calculate STK

6.4.4.2 AES-CMAC Algorithm

RFC-4493[1] defines the Cipher-based Message Authentication Code (CMAC) that uses AES-128 as the block cipher function, also known as AES-CMAC. The inputs to AES-CMAC are:

- m is the variable length data to be authenticated
- k is the 128-bit key

The 128-bit message authentication code (MAC) is generated as follows: MAC = AES-CMACk(m)

A device can implement AES functions in the Host or can use the HCI_LE_Encrypt command (see Bluetooth [Vol 2] Part E, Section 7.8.22) to use the AES function in the Controller.

6.4.4.3 Identity Root Generation

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

6.4.4.3.1 Identity Resolving Key Generation

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

6.4.4.3.2 Diversifier Hiding Key Generation

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

6.4.4.3.3 Connection Signature Resolving Key Generation

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

6.4.4.3.4 Long Term Key and Diversifier Generation

Devices supporting encrypted links in the slave role are capable of generating unique LTK and DIV values. The DIV is used by the slave device to regenerate a previously shared LTK to start an encrypted connection with a previously paired master device. Any method of generation of LTK can be used as it is not visible outside the slave device. New values of LTK and DIV are generated each time they are distributed.

6.4.4.3.5 Encrypted Session Setup

Establishing an encrypted link requires that both devices use the same key, which has either been generated on both devices using the same base parameters (reference to STK) or previously distributed. Both devices always use the slave distributed LTK if the link is to be encrypted using LTK. The host of the master provides the link layer with the long term key to use when setting up the encrypted session, together with the EDIV and RAND numbers that correspond to it. The EDIV and RAND are two ‘identifiers’ for the LTK and they allow retrieval of the same key on both devices without actually exchanging it. During the encryption session setup the master device sends the EDIV and the random number to the slave device. The host of the slave receives the EDIV and Rand values and provides the corresponding long term key to the slave’s link layer to use when setting up the encrypted link. The encrypted session can be setup either by using STK or LTK. The procedure is the same, the only difference being that when using STK, EDIV=RAND=0.

6.4.4.3.6 Link Layer Encryption

As described in the Bluetooth Specification (Version 6.0, Vol 6, Part E), the Link Layer provides encryption and authentication using Counter with Cipher Block Chaining-Message Authentication Code (CCM) Mode, which shall be implemented consistent with the algorithm as defined in IETF RFC 3610 (*http://www.ietf.org/rfc/rfc3610.txt*) in conjunction with the AES-128 block cipher as defined in NIST Publication FIPS-197 (*http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*). A description of the CCM algorithm can also be found in the NIST Special Publication 800-38C (*http://csrc.nist.gov/publications/PubsSPs.html*).

This specification uses the same notation and terminology as the IETF RFC except for the Message Authentication Code (MAC) that in this specification is called the Message Integrity Check (MIC) to avoid confusion with the term Media Access Controller.

CCM has two size parameters, M and L. The Link Layer defines these to be:

- M = 4; indicating that the MIC (authentication field) is 4 octets
- L = 2; indicating that the Length field is 2 octets

CCM requires a new temporal key whenever encryption is started. CCM also requires a unique nonce value for each Data Channel PDU protected by a given temporal key. The CCM nonce shall be 13 octets.

6.4.4.3.7 Signing Algorithm

An LE device can send signed data without having to establish an encrypted session with a peer device. Data is signed using CSRK. The signing algorithm is used in two situations:

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

6.4.4.3.8 Slave Initiated Security

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

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

6.4.4.4 Procedure Details

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

6.4.4.4.1 Random Address Generation

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

- Static address
- Private address

A private address can be either of the following types:

- Non-resolvable private address
- Resolvable private address

The three figures below (Figure 114, Figure 115, and Figure 116) give the structure of each kind of private address:



Figure 114. Static Random Address Structure



Figure 115. Private Non-Resolvable Random Address Structure

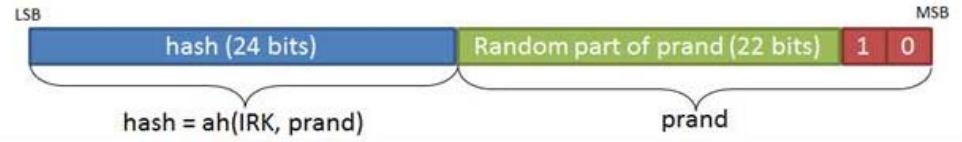


Figure 116. Private Resolvable Random Address Structure

The random address generation procedure, shown in Figure 117 on page 143, will be the same, whatever kind of random address is requested. However, in the case of a resolvable private address, the IRK used to generate the address shall be kept by the higher layers so that it can be distributed to a peer device.

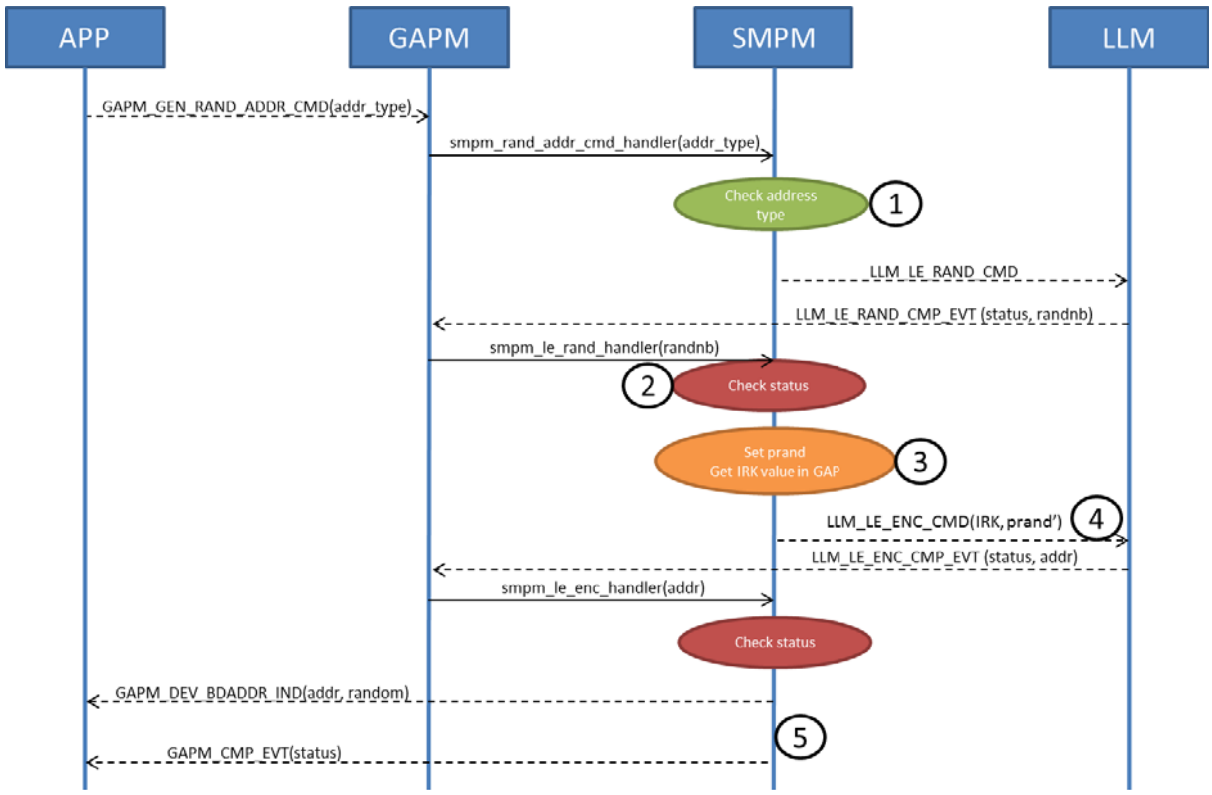


Figure 117. Random Address Generation Procedure

1. If the address type is not valid, a `GAPM_CMP_EVT` message with a `GAP_ERR_INVALID_PARM` status error is sent.
2. If an error status is returned by the controller, a `GAPM_CMP_EVT` message with a `GAP_ERR_LL_ERROR` status error is sent.
3. `prand = LSB22(randnb) || LSB2(addr_type)`
4. `prand' = 0`^{104} `|| prand`
5. If an error status is returned by the controller, a `GAPM_CMP_EVT` message with a `GAP_ERR_LL_ERROR` status error is sent, else the status will be `GAP_ERR_NO_ERROR`.

6.4.4.4.2 Address Resolution

The address resolution procedure, as shown in Figure 118 on page 144, is used to identify a device which would use a resolvable private random address. The structure of this kind of address is defined in Section Figure 116., “Private Resolvable Random Address Structure” on page 143.

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

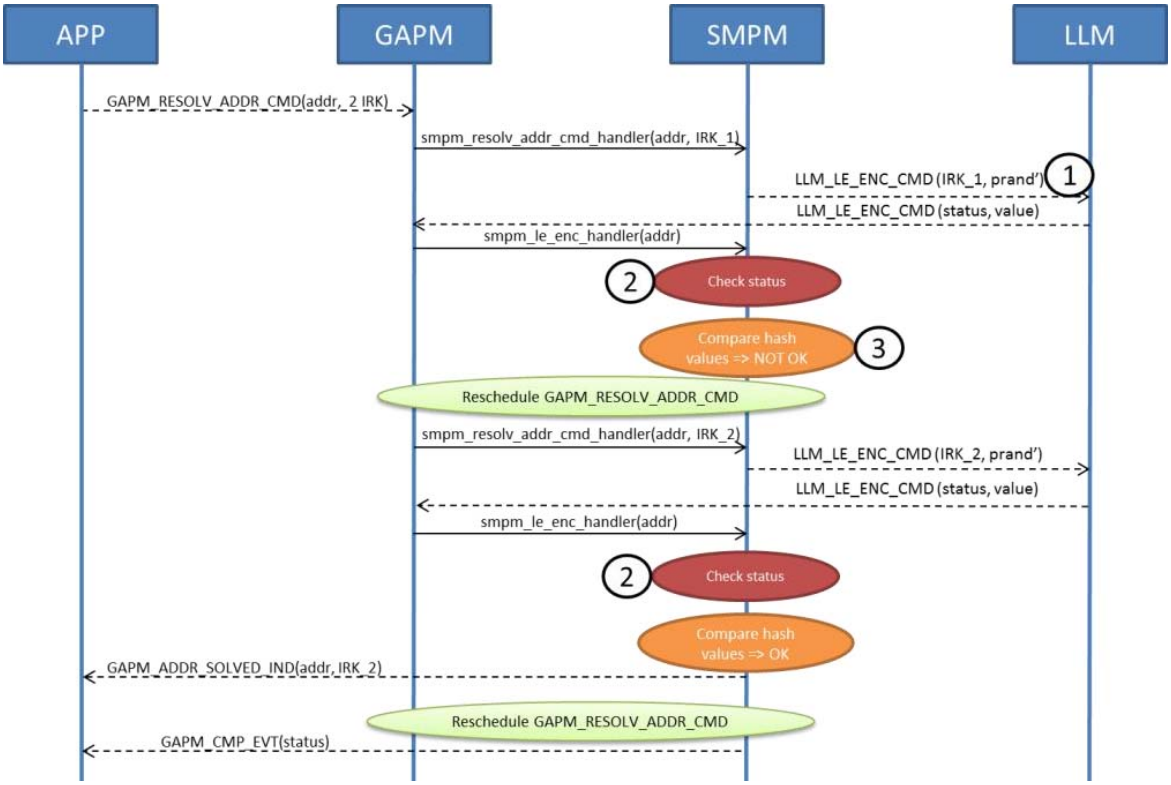


Figure 118. Address Resolution Procedure

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

6.4.4.4.3 Encryption Toolbox Access

The encryption toolbox access provides a way for a host layer to use the hardware encryption block. This block can be accessed using the LLM API, as shown in Figure 119.

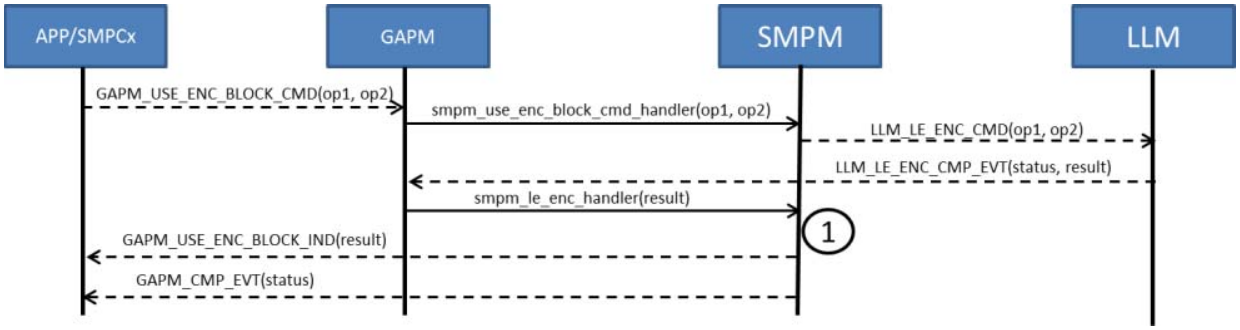


Figure 119. Encryption Toolbox Access

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

6.4.4.4.4 Pairing

- Phase 1 – pairing feature exchange: It is used to exchange IO capabilities, OOB authentication data, authentication requirements and which keys to distribute.
- Legacy phase 2 – authentication and encryption: information exchanged during the phase 1 is used to determine which method will be used to encrypt the link (Just Works, Passkey Entry, Out Of Band).
- LE secure connections Phase 2: – authentication and encryption: information exchanged during the phase 1 is used to determine which method will be used to encrypt the link (Just Works, Numeric Comparison, Passkey Entry, Out Of Band). The outcome of this pairing is Long Term Key (LTK) generation.
- Phase 3 – transport keys distribution: This phase is optional and depends on the key distribution features shared during phase 1.

6.4.4.4.4.1 Phase 1: Pairing Feature Exchange (Initiated by Master)

The pairing is always initiated by the master device by sending a pairing request message, as shown in Figure 120.

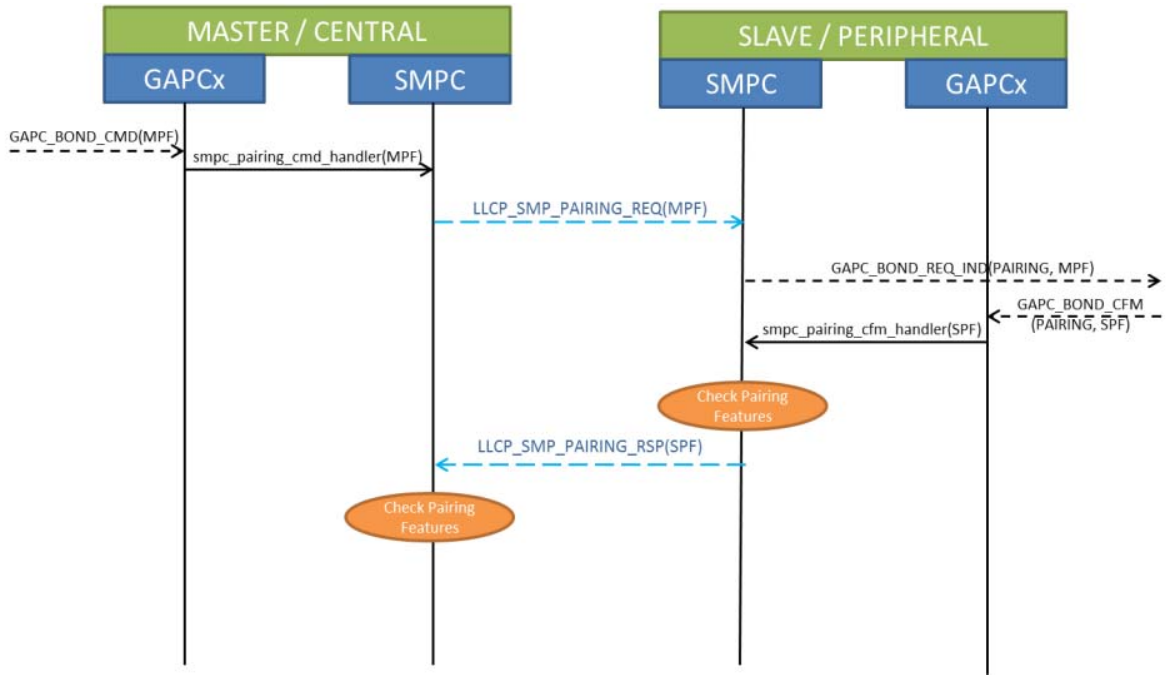


Figure 120. Pairing Phase 1: Pairing Features Exchange (Initiated by Master)

If the slave device doesn't support pairing, it responds using the pairing failed message with the error code `Pairing Not Supported` upon reception of a pairing request message. If a device receives a command with invalid parameters, it responds with a pairing failed command with the error code `Invalid Parameters`. The minimum and maximum encryption key length parameters shall be between 7 bytes and 16 bytes in 1 byte steps. The smaller value of the initiating and the responding device's maximum encryption key length will be used as the encryption key size. If the resultant encryption key size is smaller than the minimum key size parameter, the device responds with pairing failed command with the error code `Encryption Key Size`.

6.4.4.4.4.2 Phase 1: Pairing Feature Exchange (Initiated by Slave)

A slave device requires that the master initiates a pairing procedure by sending a security request, as shown in Figure 121.

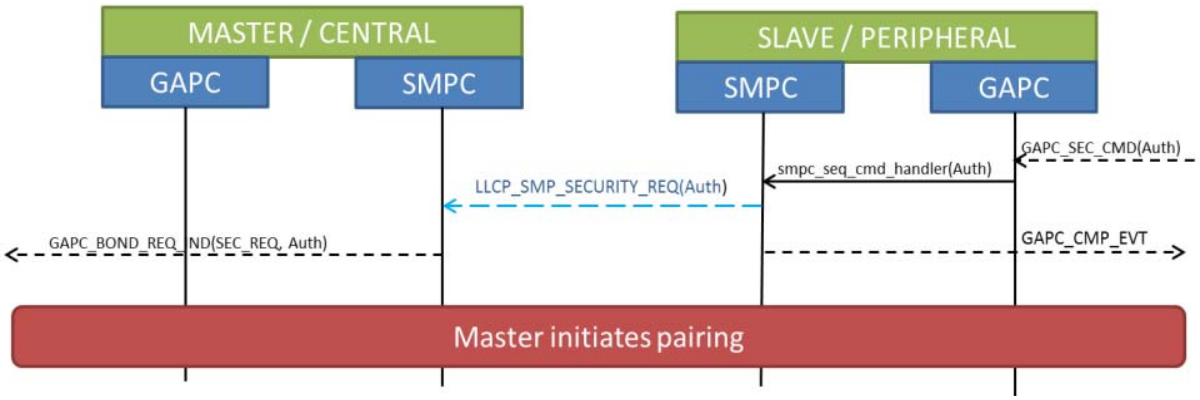


Figure 121. Pairing Phase 1: Pairing Features Exchange (Initiated by Slave)

6.4.4.4.4.3 Legacy Phase 2: Authentication and Encryption

The information exchanged in Phase 1 is used to select which STK generation method is used in Phase 2, as shown in Figure 122 on page 147.

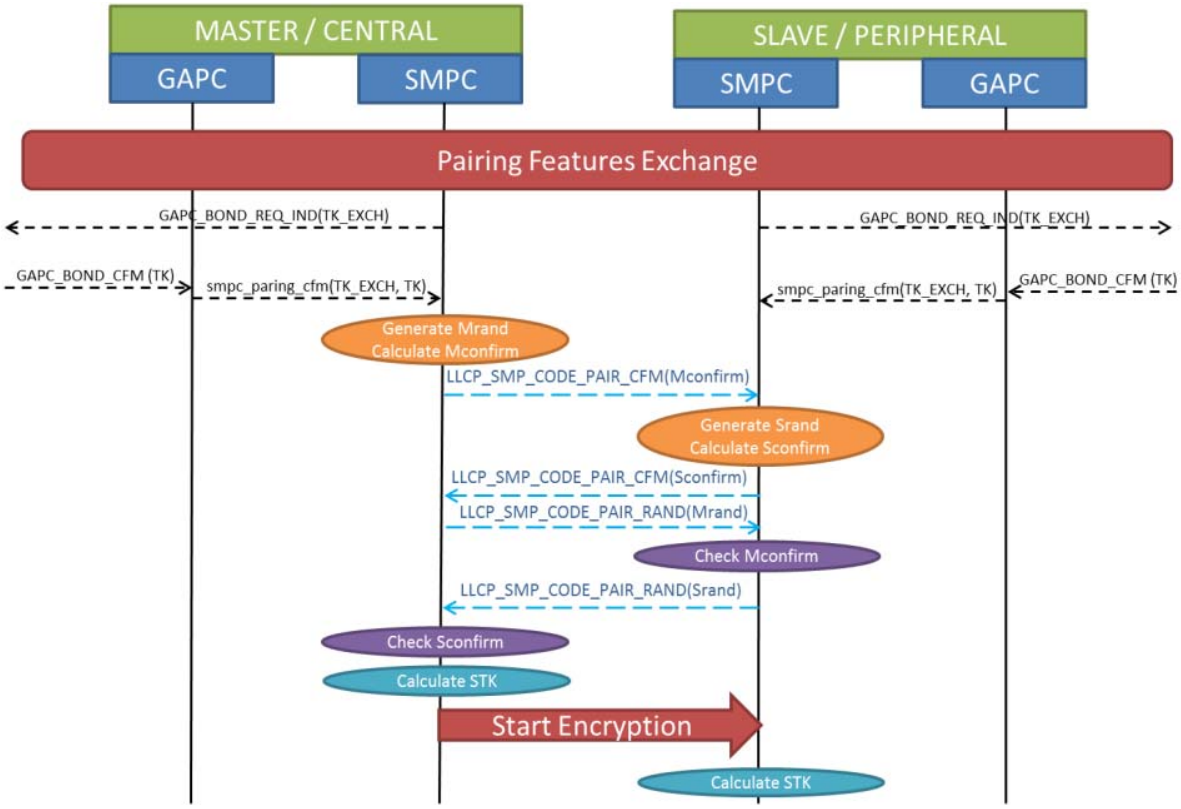


Figure 122. Phase 2: Authentication and Encryption

If the Just Works method is used, no TK will be required (0 is used) from the application. If a generated Sconfirm or Mconfirm value doesn't match with the received confirm value from the peer device, the device aborts the pairing procedure by sending a pairing failed message with a `Confirm Value Failed` error code.

6.4.4.4.4.4 LE Secure Connection Phase 2: Authentication and Encryption

Authentication Stage 1: Just Works Method

If it is not possible to enter a passkey or do a numeric comparison, this method applies, as seen in Figure 123 on page 148:

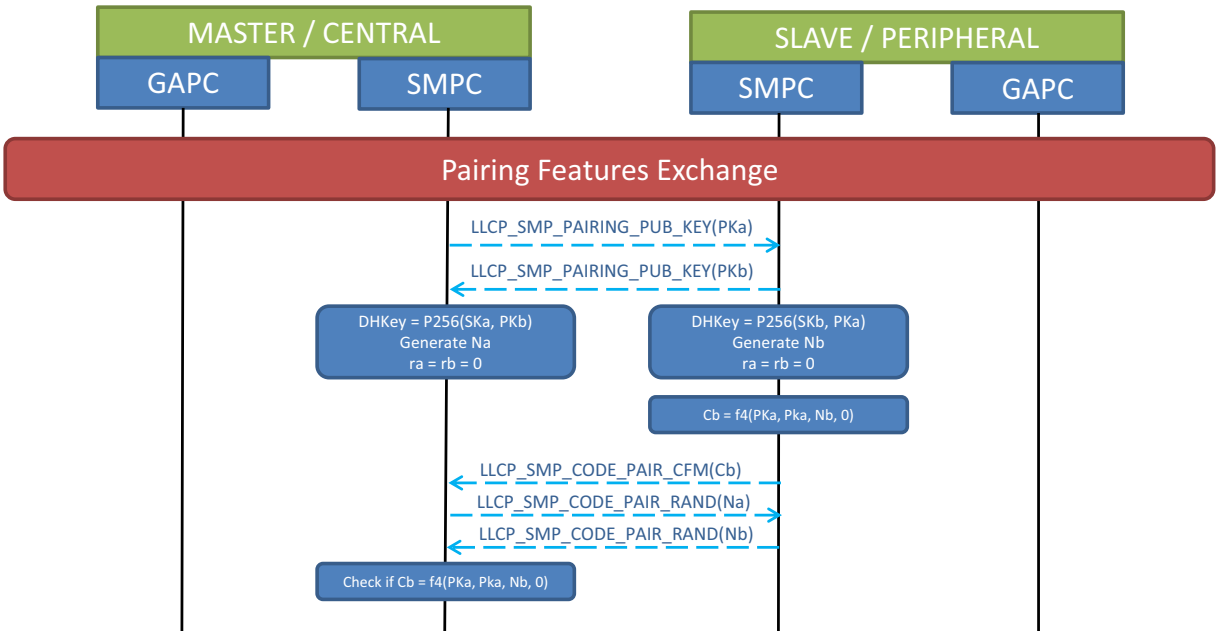


Figure 123. Phase 2: LE Secure Connection Just Works Pairing

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

Authentication Stage 1: Numeric Comparison Method

If both devices have display capability, numeric comparison must be chosen, as seen in Figure 124 on page 149.

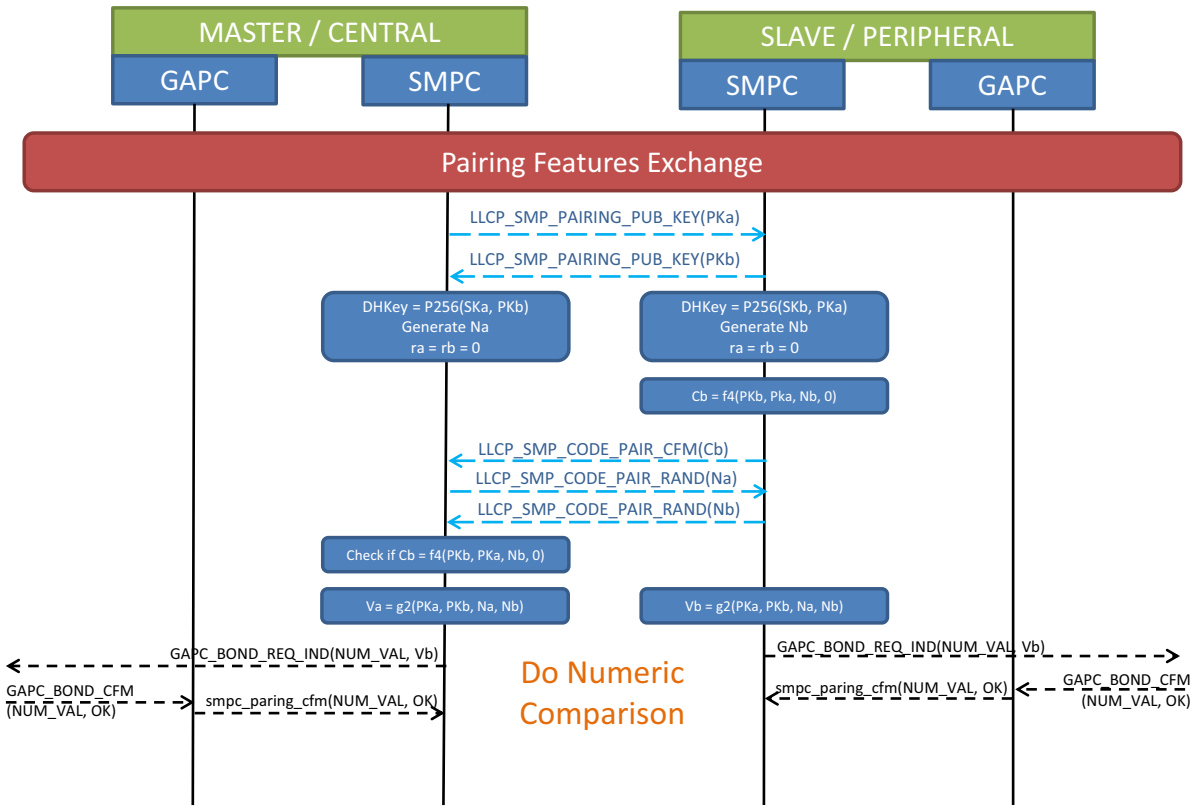


Figure 124. Phase 2: LE Secure Connection Numeric Comparison Pairing

At the end of the pairing, the link is considered secure connection authenticated.

Authentication Stage 1: Passkey Entry Method

If both devices have pin code entry possible, passkey entry is chosen, as shown in Figure 125 on page 150:

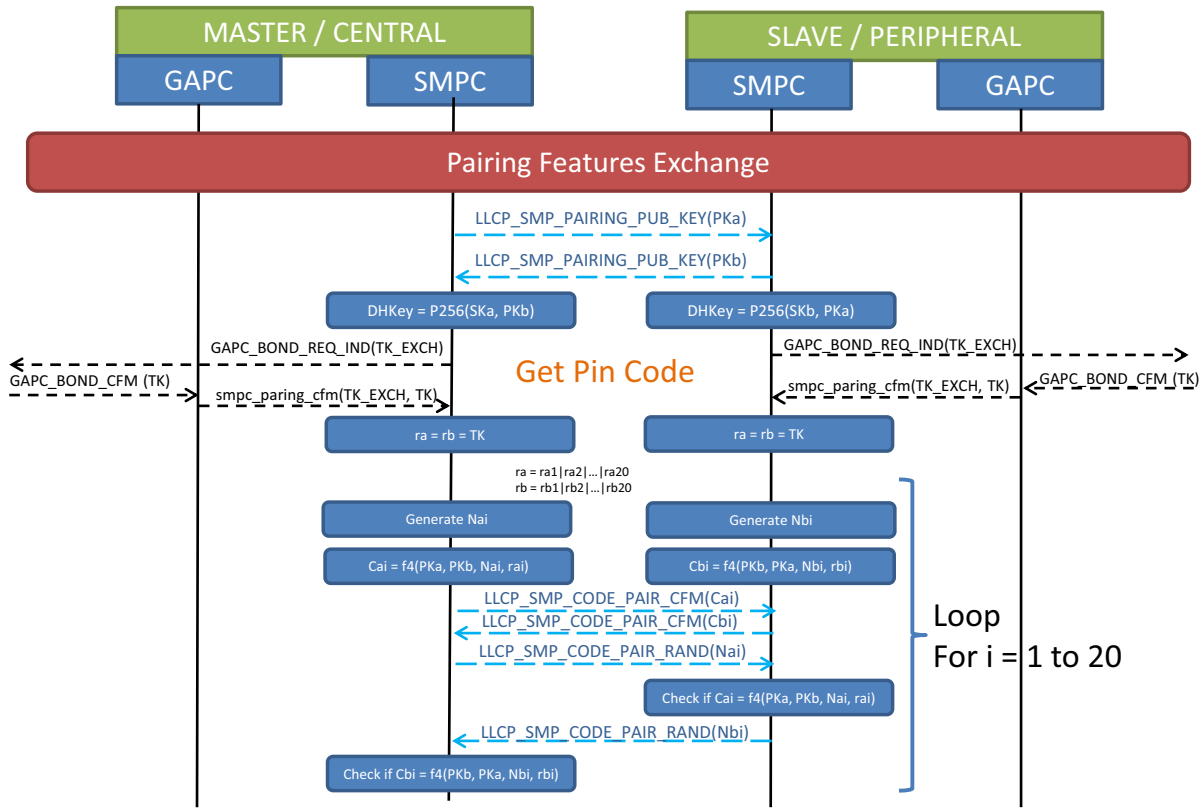


Figure 125. Phase 2: LE Secure Connection Passkey Entry Pairing

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

At the end of the pairing, the link is considered secure connection authenticated.

Authentication Stage 1: Out of Band Method

If OOB Data can be sent by one or both devices, the Out Of Band pairing method is chosen, as seen in Figure 126 on page 151:

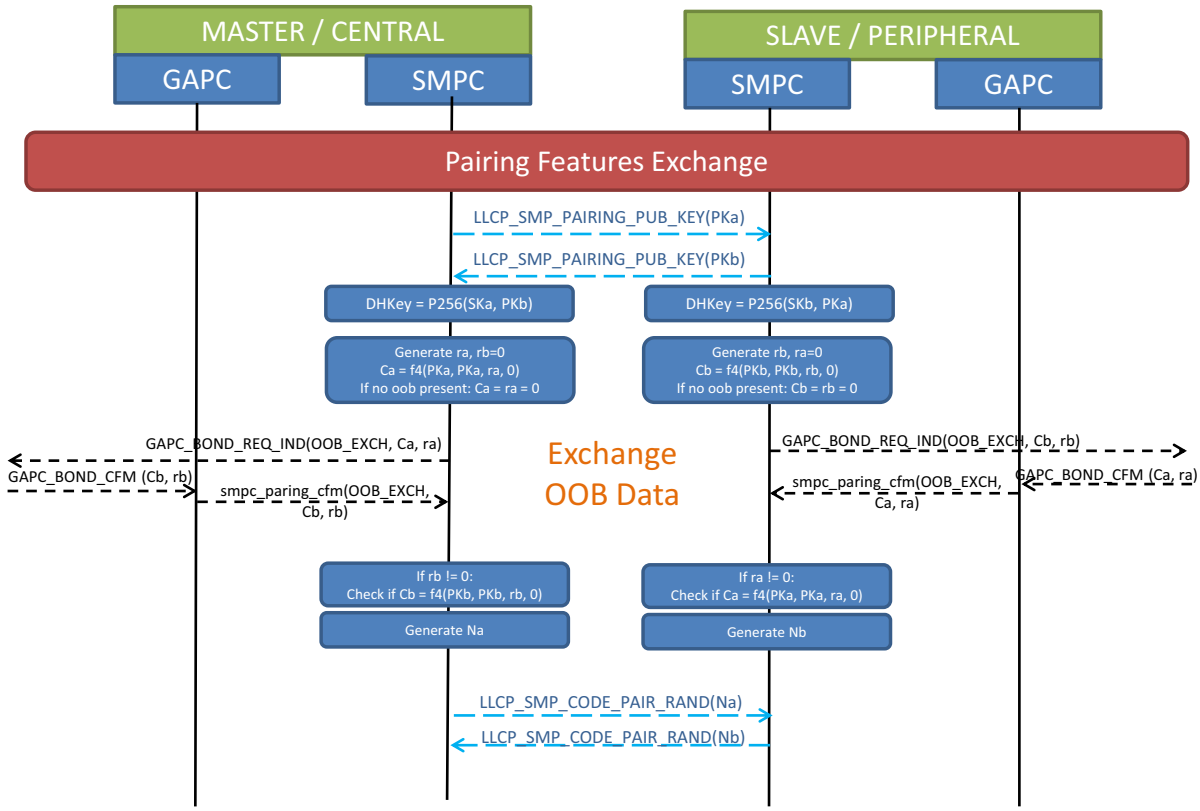


Figure 126. Phase 2: LE Secure Connection Out of Band Pairing

At the end of the pairing, the link is considered secure connection authenticated.

Authentication Stage 2: Generation of LTK

After the LE secure connection authentication Stage 2, the LTK is generated according to pairing information, as seen in Figure 127 on page 152. Then the link is encrypted. If encryption succeeds and the BOND bit is present in the pairing feature exchange, the generated LTK is provided to the upper application.

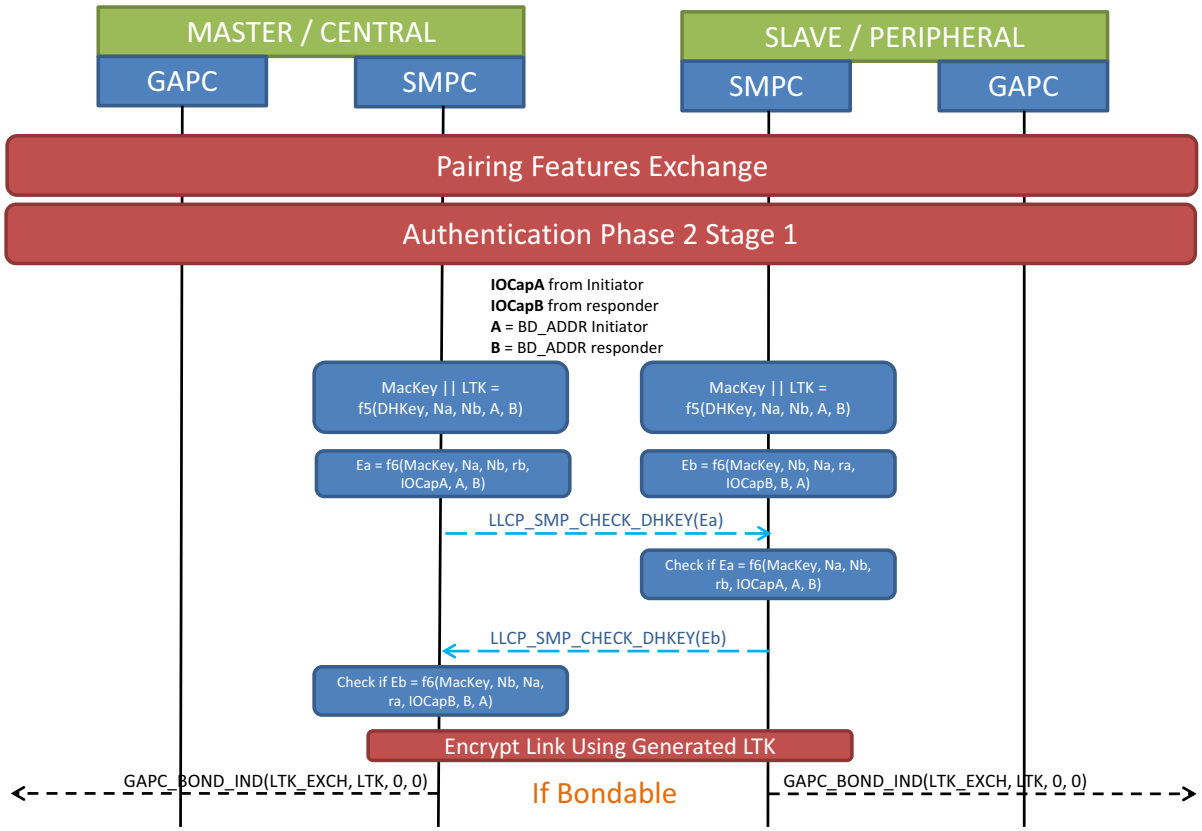


Figure 127. Phase 2: LE Secure Connection LTK Generation

6.4.4.4.4.5 Phase 3: Transport Keys Distribution

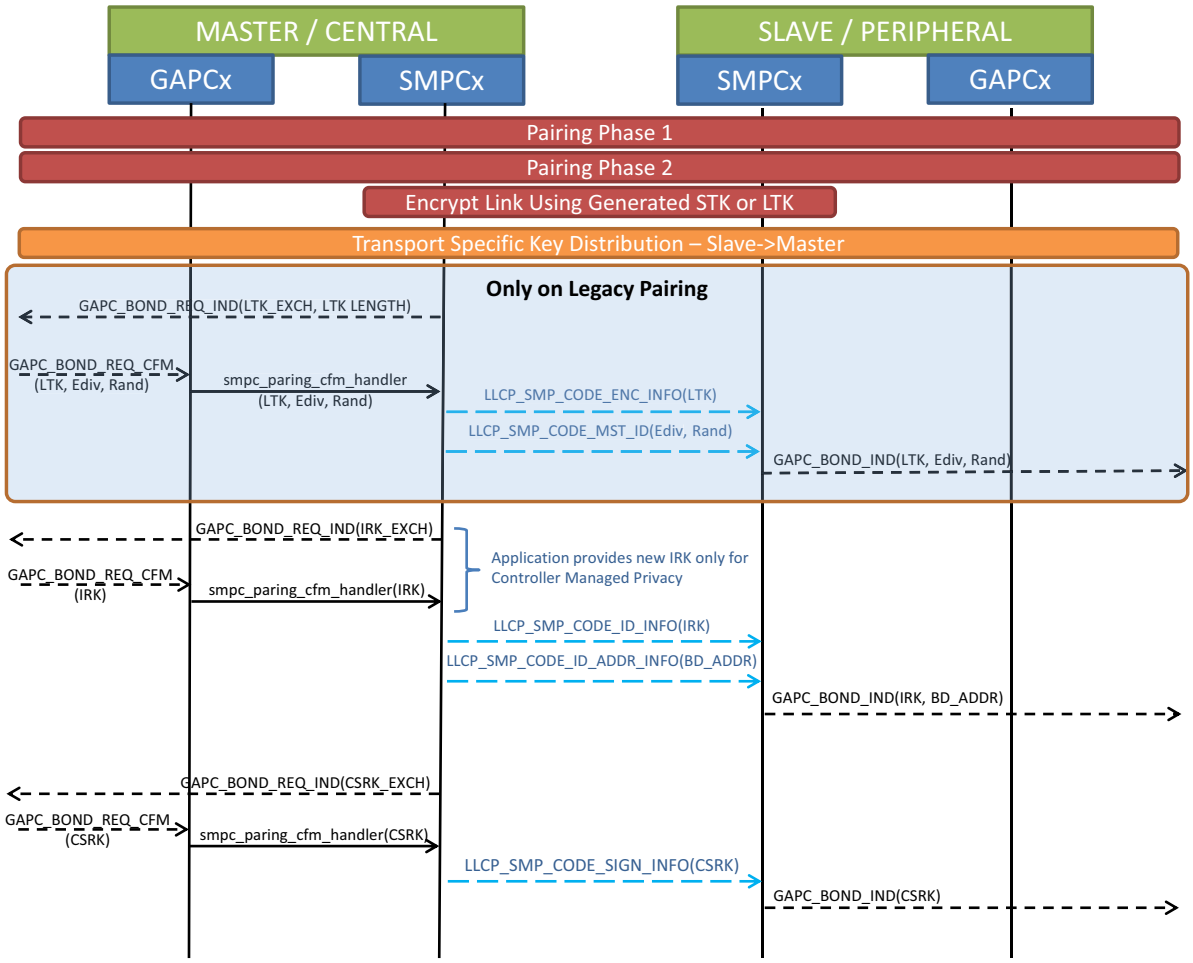


Figure 128. Phase 3: Transport Keys Distribution

When Privacy is managed by the host (privacy 1.1), the IRK value is already set in the GAP environment. But for a controller managed privacy (privacy 1.2), the IRK will be unique for each bonded device, and so the new IRK will be generated and retrieved from the application. The LTK and the CSRK need to be retrieved from the application. On legacy paring, the application is responsible for generating the transport keys (CSRK, IRK, LTK, Ediv, Rand) by any means. Figure 128 shows the distribution of the transport keys. The `GAPM_USE_ENC_BLOCK_CMD` message can be used through the GAP API. On secure connection pairing, the application is responsible for generating only CSRK and IRK; the LTK is generated by the pairing algorithm.

6.4.4.4.4.6 End of Pairing Procedure

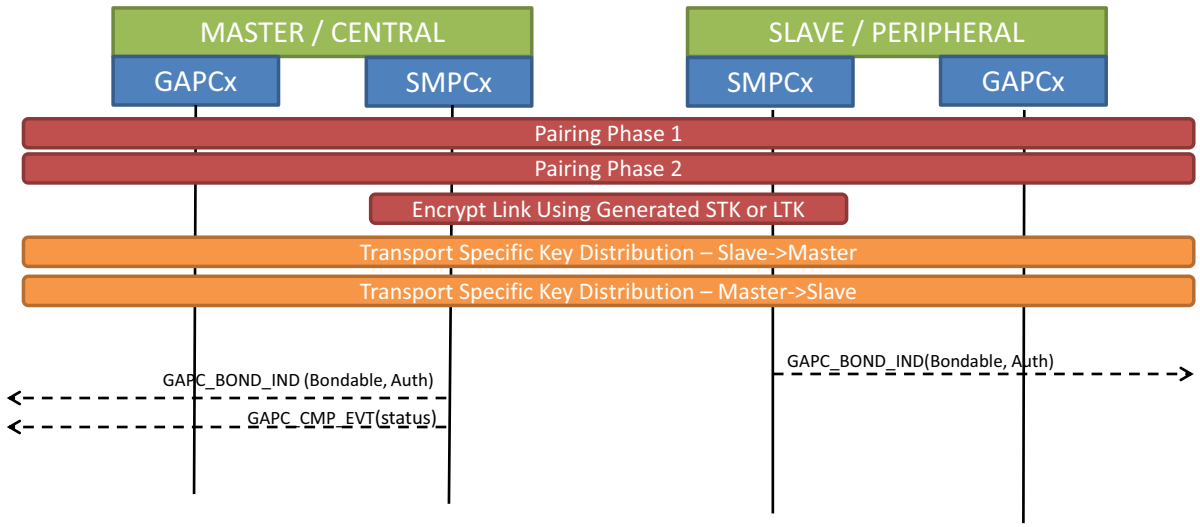


Figure 129. End of Pairing Procedure

The pairing procedure is considered to be over in the following cases, as shown in Figure 129:

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

6.4.4.4.5 Encryption

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

6.4.4.4.5.1 Case 1: Both devices have LTK

If a master already knows the encryption keys of the slave device it is connected with, it can initiate the creation of an encrypted link, as shown in Figure 130 on page 155.

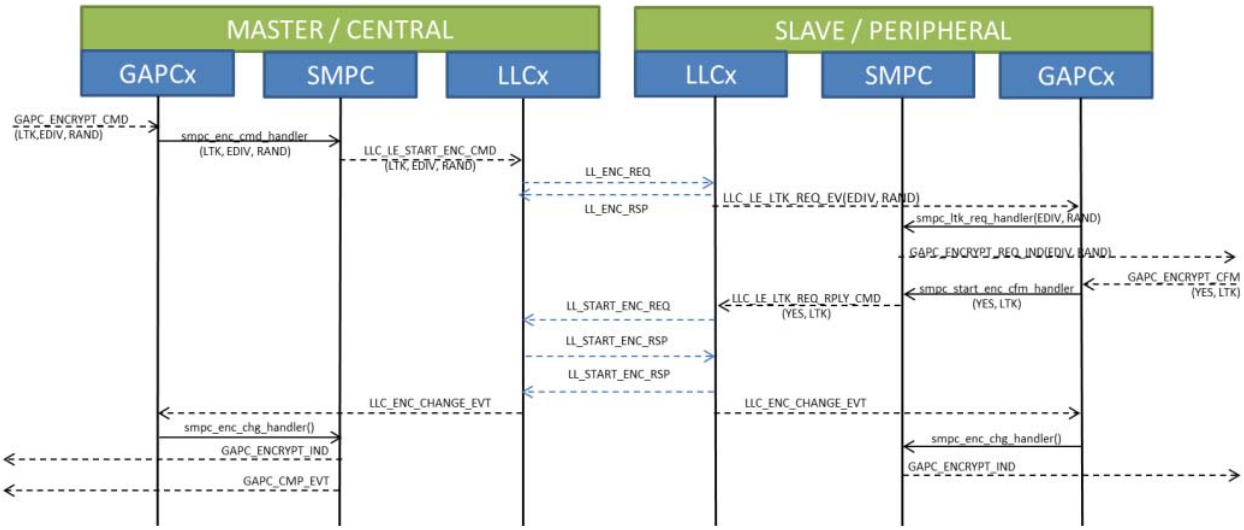


Figure 130. Start Encryption Procedure (Both Devices Have Keys)

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

6.4.4.4.5.2 Case 2: Slave forgot the LTK

If the slave forgot the LTK distributed by the master device during a previous bonding procedure, it will reject the encryption request with a `Pin Key Missing` error, as shown below in Figure 131. Upon reception of this error, the master can initiate a new pairing procedure with the slave device.

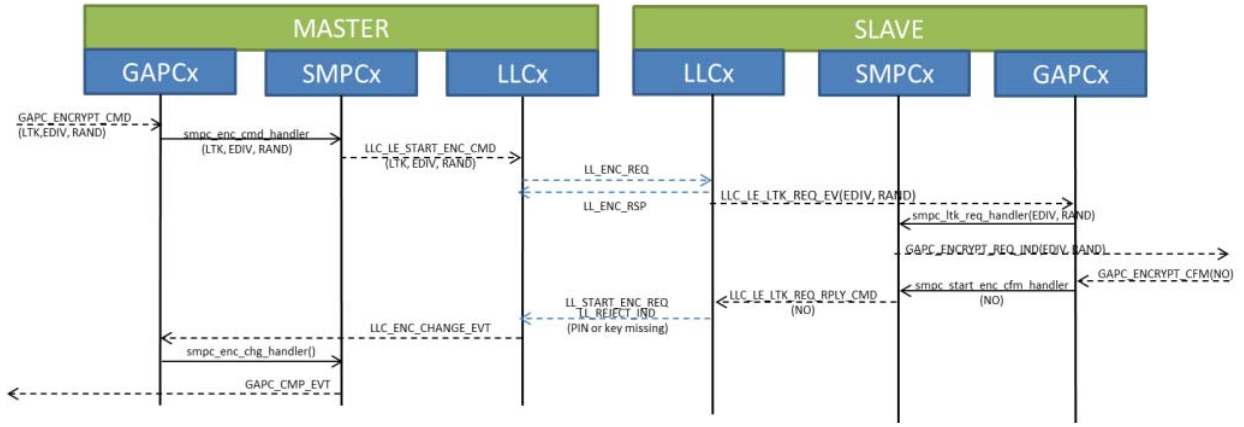


Figure 131. Start Encryption Procedure (Slave Forgot Keys)

6.4.4.4.5.3 Case 3: Slave doesn't support encryption

This case is illustrated in Figure 132 on page 156.

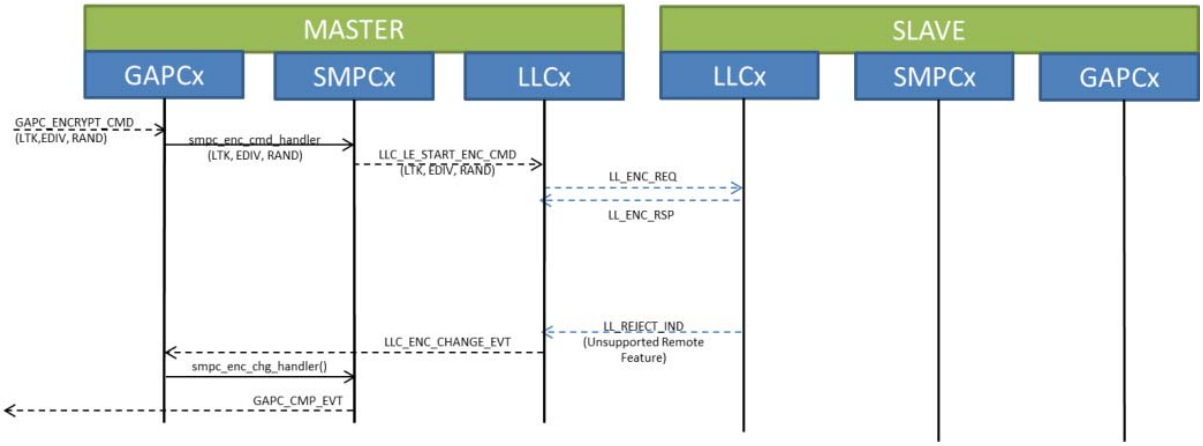


Figure 132. Start Encryption Procedure (Slave Does not Support Encryption)

6.4.4.4.6 Data Signing

The data signing procedure is used to authenticate a data PDU sent over a non-encrypted link. More details about the generation of the signature can be found in 6.4.4.2.6.

6.4.4.4.6.1 Subkeys Generation

An illustration of subkeys generation can be found in Figure 133, below.

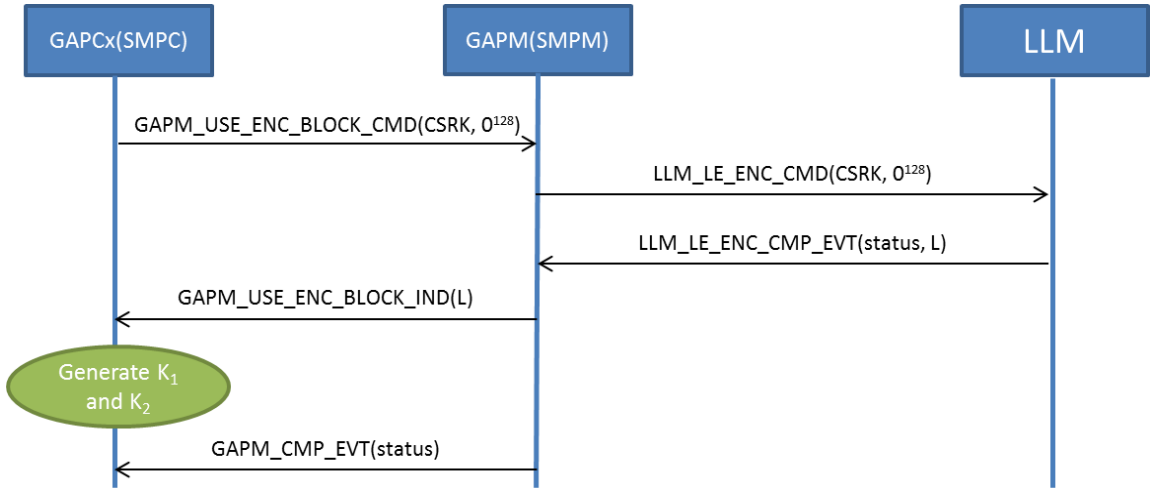


Figure 133. Data Signing: Subkeys Generation

6.4.4.4.6.2 MAC Generation

This refers to the data to be signed in the concatenation of the data PDU and the `SignCounter` value, as shown in Figure 134 on page 157.

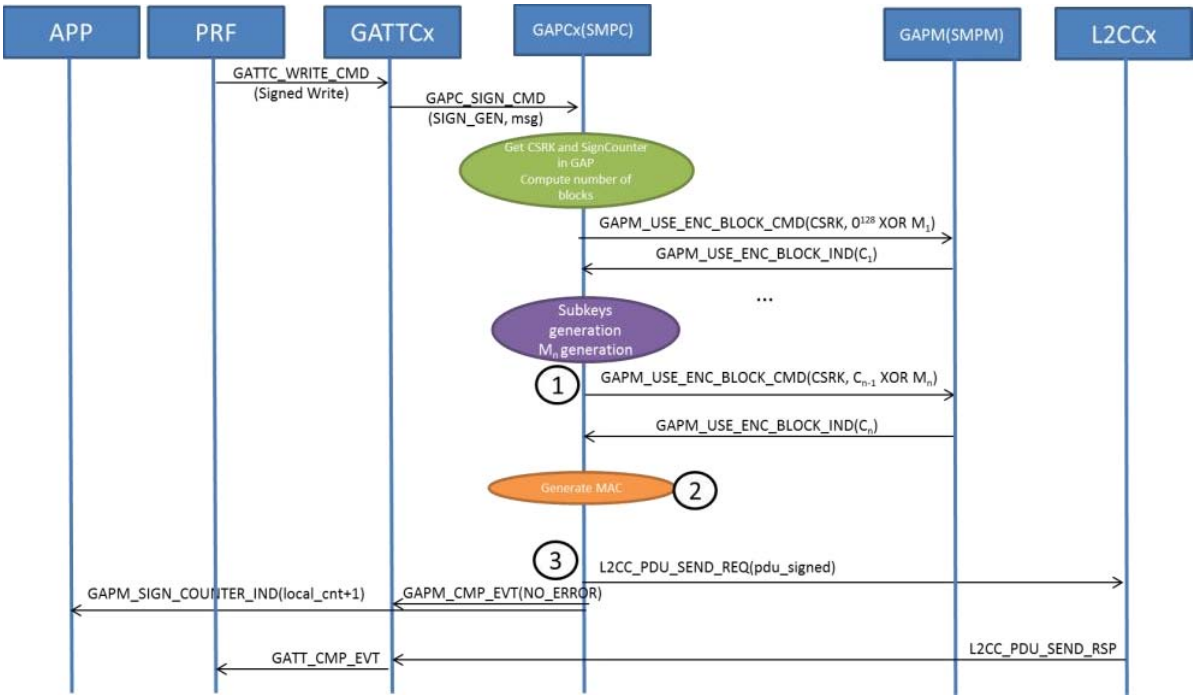


Figure 134. Data Signing: MAC Generation

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

6.4.4.4.6.3 MAC Verification

The verification of the received MAC is done by generating a MAC value based on the received data PDU and `SignCounter` values. If the generated MAC value matches with the received one, the signature is accepted, as seen in Figure 135 on page 158.

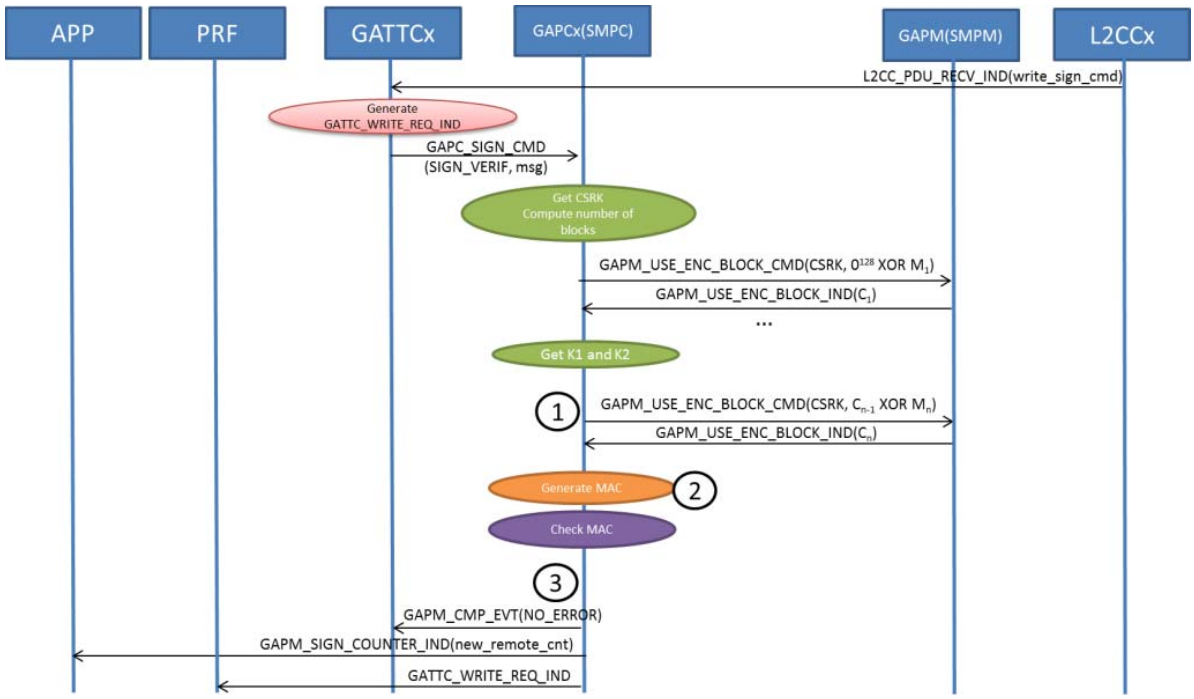


Figure 135. Data Signing: MAC Verification

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

6.4.4.4.7 Pairing Repeated Attempts

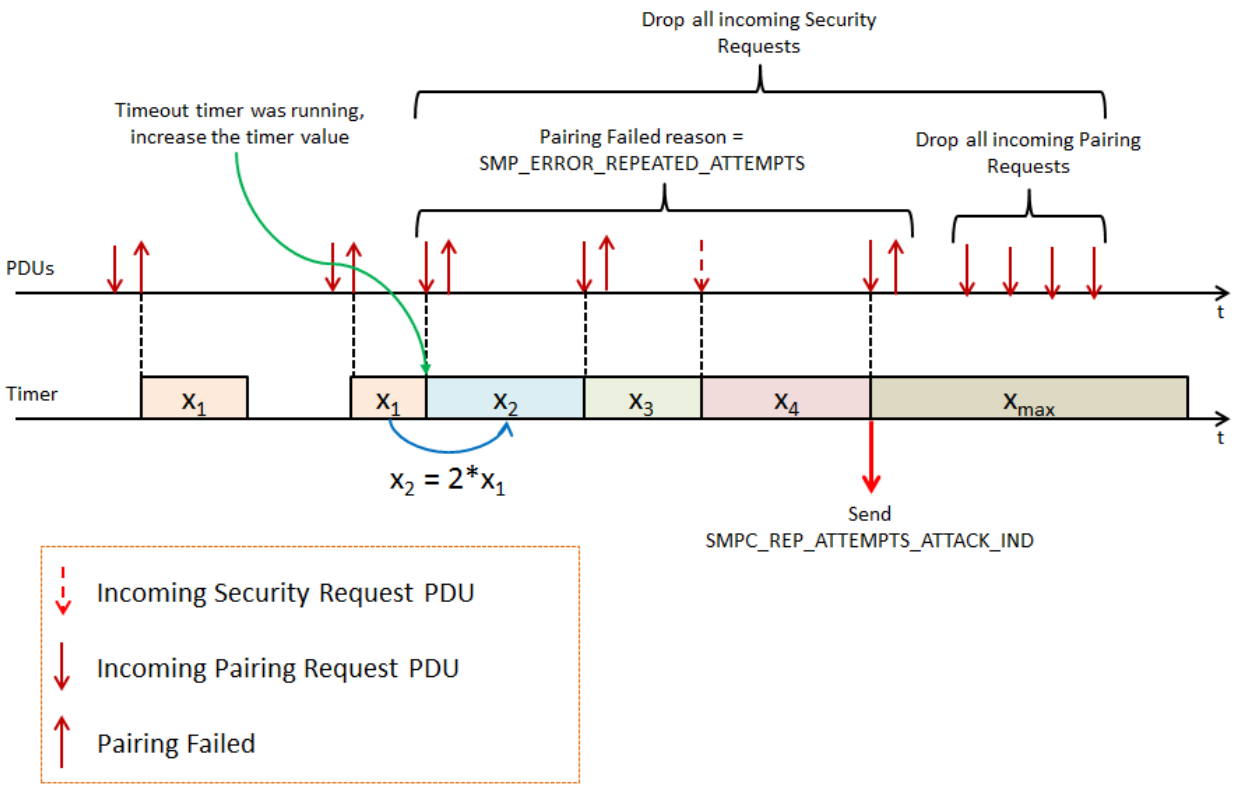


Figure 136. Repeated Attempts Protection

The Bluetooth specification [1] requires the implementation of a mechanism: “When a pairing procedure fails a waiting interval shall pass before the verifier will initiate a new Pairing Request command or Security Request command to the same claimant, or before it will respond to a Pairing Request command or Security Request command initiated by a device claiming the same identity as the failed device. For each subsequent failure, the waiting interval shall be increased exponentially.” Figure 136 presents the mechanism implemented to rapidly detect an attack from a malicious device. The minimal interval value is set to 2 s and the maximal interval value is set to 30 s. Thus, according to the procedure described in Figure 136, a repeated attempt attack will be detected after five attempts.

6.4.4.5 Security Manager Protocol Data Unit Format

All SMP commands are transmitted over L2CAP using fixed channel with CID `0x0006` in Basic L2CAP mode. SMP has a fixed L2CAP MTU size of 23 octets. Only a single SMP command is sent per L2CAP frame. (See Figure 137 on page 159.)

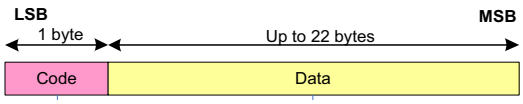


Figure 137. SMP Command PDU

6.4.4.5.1 SMP PDU Codes

Table 59 below specifies the SMP codes. A packet with a code not included in the list below is ignored.

Table 59. SMP Codes

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

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

6.4.5 LE Credit Based Channel

The LE credit based connection, also called the connection oriented channel (COC), is an L2CAP feature managed by GAP. It allows an LE service to create a dedicated channel on a specific link. The Peer service client must connect to this LE credit based connection before exchanging any packets.

The GAPM manages the list of LE credit based channels created by a profile service. (A peer device cannot connect to an LE credit based channel if it does not exist on manager.)

NOTE: The maximum number of LECB connection that can be established for a device is configurable for the device (see Section 6.4.5.11.2, “Device Configuration” on page 174).

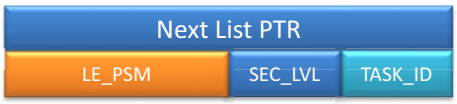


Figure 138. LE Credit Manager Environment Structure

The manager environment of variables that manages an LE credit based channel is allocated in ATT_DB heap. (See Figure 138.) It contains:

- LE_PSM (LE Protocol/Service Multiplexer)
- TASK identifier that manages the channel
- Security level requirement (authentication level and encryption key size) - see Figure 139, below

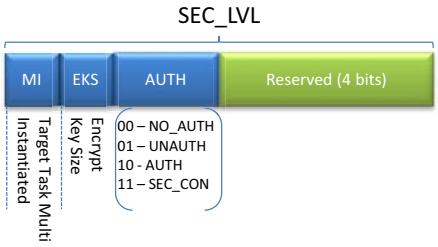


Figure 139. LE Credit Connection Security Bit Field

GAPC manages the LE credit based channels connection using a list.



Figure 140. LE Credit Connection Environment Structure

The environment of variables that manages an LE credit based connection is allocated in the ATT_DB heap. (See Figure 140.) It contains:

- LE_PSM (LE Protocol/Service Multiplexer)
- TASK identifier that manages reception of packet from peer device

- Status of the current connection
- Source and destination:
  - Channel identifier
  - Number of available credits for the channel
  - Maximum Transmit Unit (MTIU)
  - Maximum Packet Size (MPS)

6.4.5.1 Channel Registration

Registration of the LE Protocol/Service Multiplexer, as shown in Figure 141 on page 162, will be performed just after device configuration (see Section 6.4.5.11.2, “Device Configuration” on page 174). This registration ensures that no LE credit based channel will be created on an unregistered LE_PSM, and also ensures the same security level for all Bluetooth links.

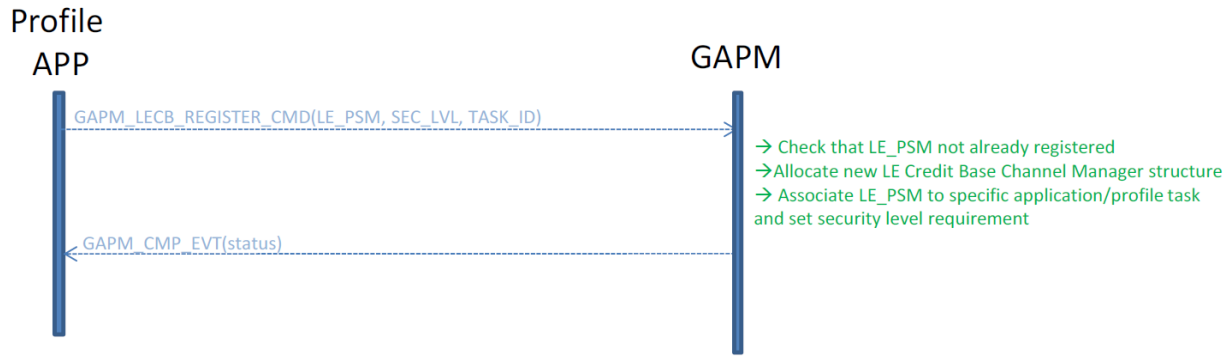


Figure 141. Registration of an LE_PSM Identifier

NOTE: The LE_PSMs are automatically unregistered when the application requests one of the device initialization procedures (see Section 6.4.5.11, “Device initialization” on page 174).

If no links are using a specified LE_PSM (no LECB connection established), the application or profile can de-register it (see Figure 142).

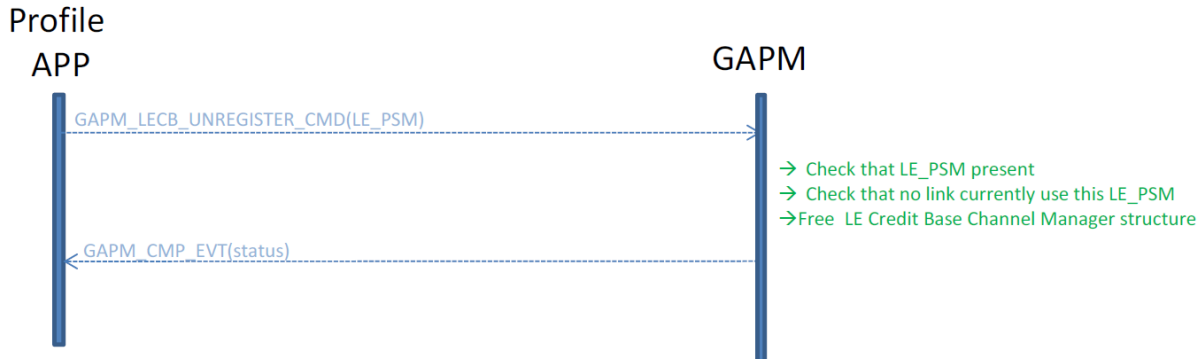


Figure 142. De-registration of an LE_PSM Identifier

6.4.5.2 Connection Creation

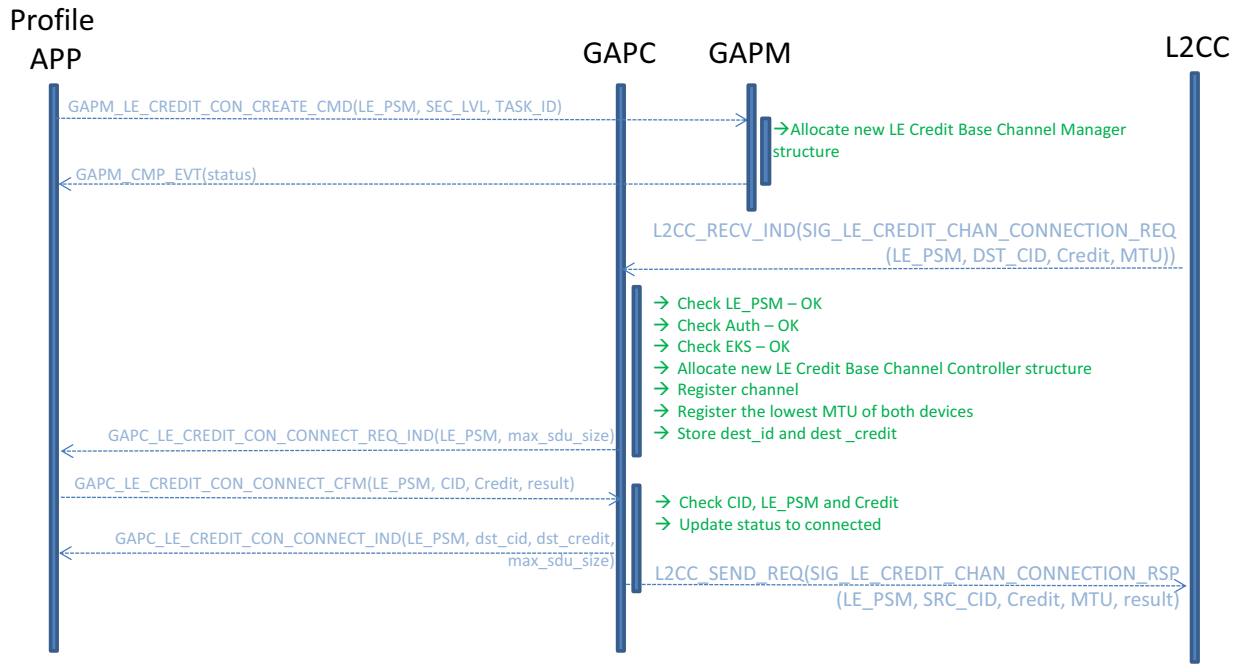


Figure 143. Service View of LE Credit Connection

Figure 143 shows different steps of LE credit based connection creation on the service side.

- LE_PSM has already been registered at GAPM Level
- Peer client then establishes the LE credit based connection using the same LE PSM, its channel ID, credit count, MTU and MPS.
- Device receives connection request and has to confirm connection with a specific result code:
  - `Accept`
  - `Reject Insufficient Resources`
  - `Reject Not Authorized`

- The initial number of credits for the local device must be at least $\text{floor}\left(\frac{\text{MTU} + (\text{MPS} - 1)}{\text{MPS}}\right) + 1$. This is for receiving at least an SDU with max packet size.
- If the local CID is set to zero, the GAPC module will find and allocate the first available LECB channel identifier.

NOTE: When the LE credit based connection is established, the application is informed about the maximum SDU size allowed (MTU - 2).

NOTE: Several connections can be opened on the same LE_PSM, but the local and peer channel identifier has to be different each time. This is the role of the application to accept or reject an incoming connection if one already exists for specific LE_PSM.

NOTE: Channel local MTU and MPS sizes for a connection cannot exceed Maximum MTU and MPS sizes configured for the device (see Section 6.4.5.11, “Device initialization” on page 174).

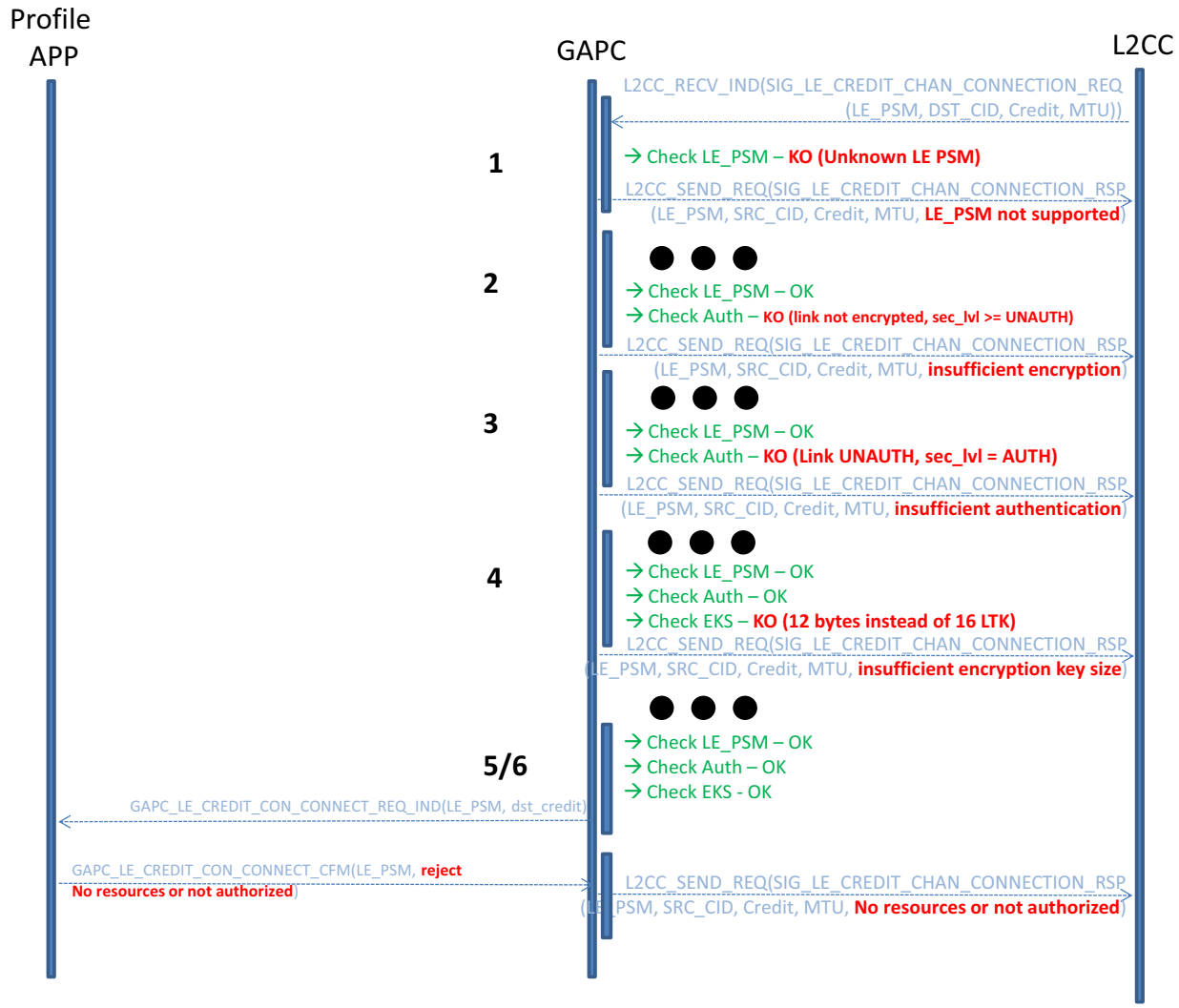


Figure 144. Service Reject Connection Creation

Figure 144 shows possible connection creation errors on the service side:

1. LE PSM is unknown.
2. Security level is set to unauthenticated, or set to authenticated but the link is not encrypted.
3. Link is encrypted with insufficient authentication level.
4. Link is encrypted with LTK key < 16 bytes but connection requires a 16-byte LTK.
5. Application cannot accept link connection due to insufficient resources.
6. Application cannot accept link connection because peer device is not authorized.

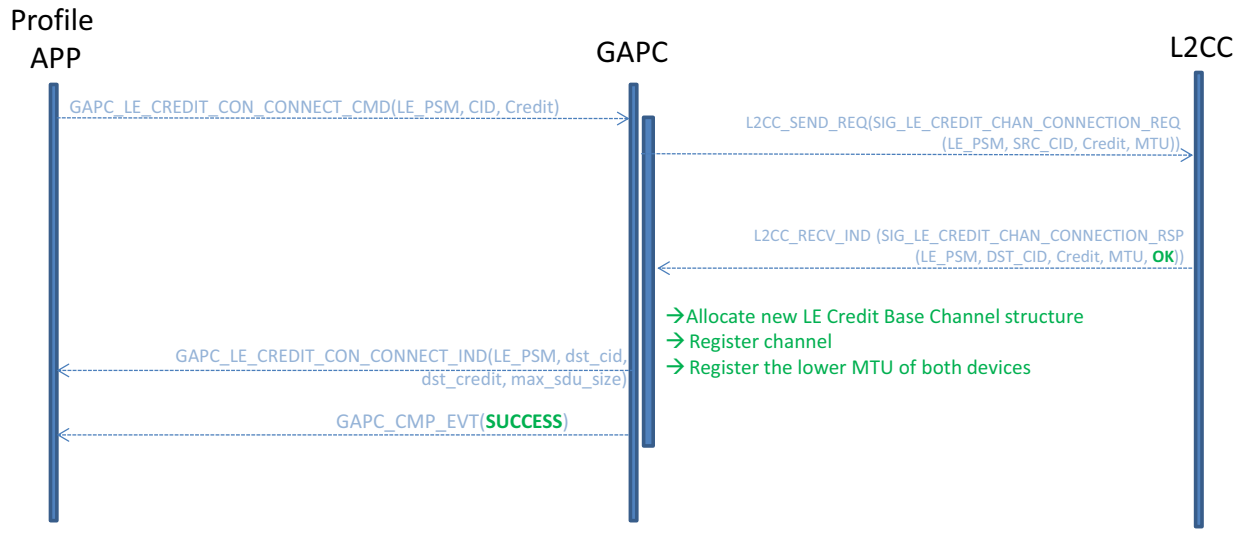


Figure 145. Client View of LE Credit Connection

Figure 145 shows the different steps of LE credit based connection creation on the client side.

- Using LE PSM, its channel ID, credit count, MTU and MPS, a client establishes an LE credit based connection created by a peer service device.
- The initial number of credits for the local device must be at least $\text{floor}\left(\frac{\text{MTU} + (\text{MPS} - 1)}{\text{MPS}}\right) + 1$. This is for receiving at least an SDU with max packet size.
- If the local CID is set to zero, the GAPC module will find and allocate the first available LECB channel identifier.

Figure 146, below, shows a client LE credit connection rejected by a peer device:

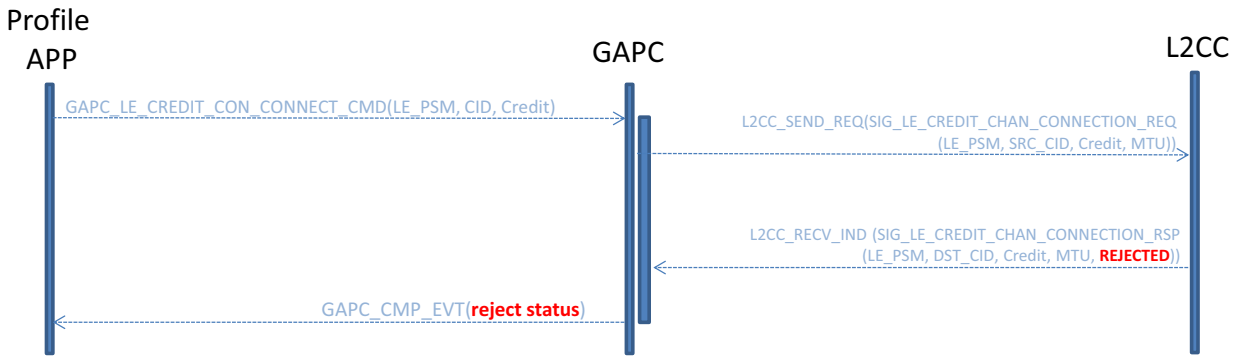


Figure 146. Client LE Credit Connection Rejected By Peer Device

6.4.5.3 Disconnection

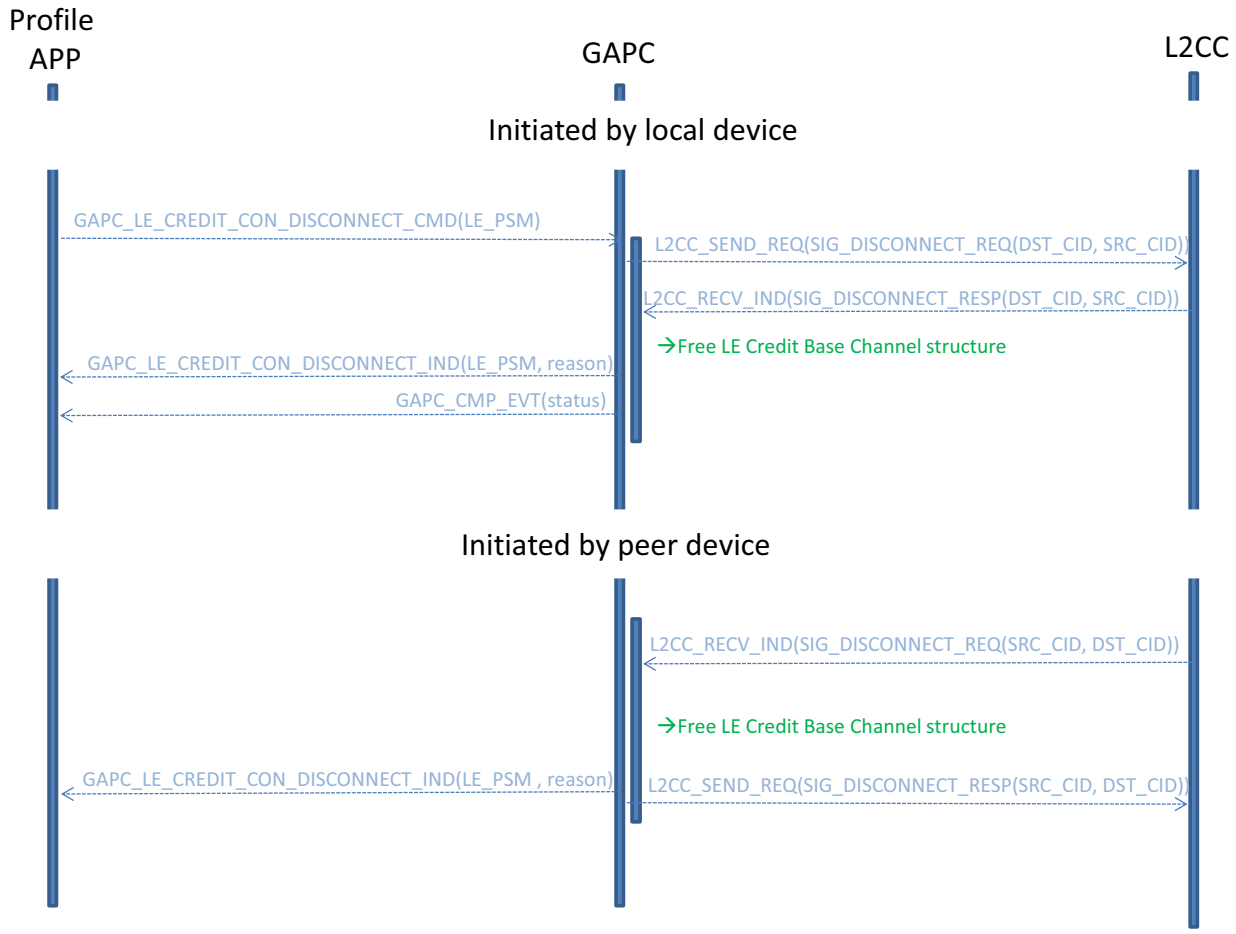


Figure 147. Disconnection Overview

Figure 147 shows how the LE credit based connection can be stopped from any device. When disconnection is performed, the corresponding environment variables are free and no more data can be sent or received on this channel.

NOTE: Reason for disconnection is provided to the upper layer, such as:

- Local device initiates disconnection (no error)
- Remote device initiates disconnection
- No more credit available
- Invalid MTU (MTU exceeded)
- Invalid packet size (MPS exceeded)
- Link connection is terminated by local or peer device

6.4.5.4 Data Exchange

Data exchange over an LE credit based channel is performed directly over an L2CC task.

- Packet transmission

When sending a packet, an L2CC Send procedure verifies with the GAPC module (using native functions) whether there is still available credit on the destination device, and whether the negotiated MTU is not exceeded. If not enough credit is available on the peer device, the packet is put into a wait queue until new credit is provided for the peer CID. When the message is in the wait queue, L2CC can send other messages (ATT, SIG, SMP, or other CID) to the peer. (See Figure 148, below.) When the full packet is transmitted, a confirmation message is sent to the application with the status of transmission and number of credits used. Until confirmation is sent, any message to send to the same CID will be rejected.

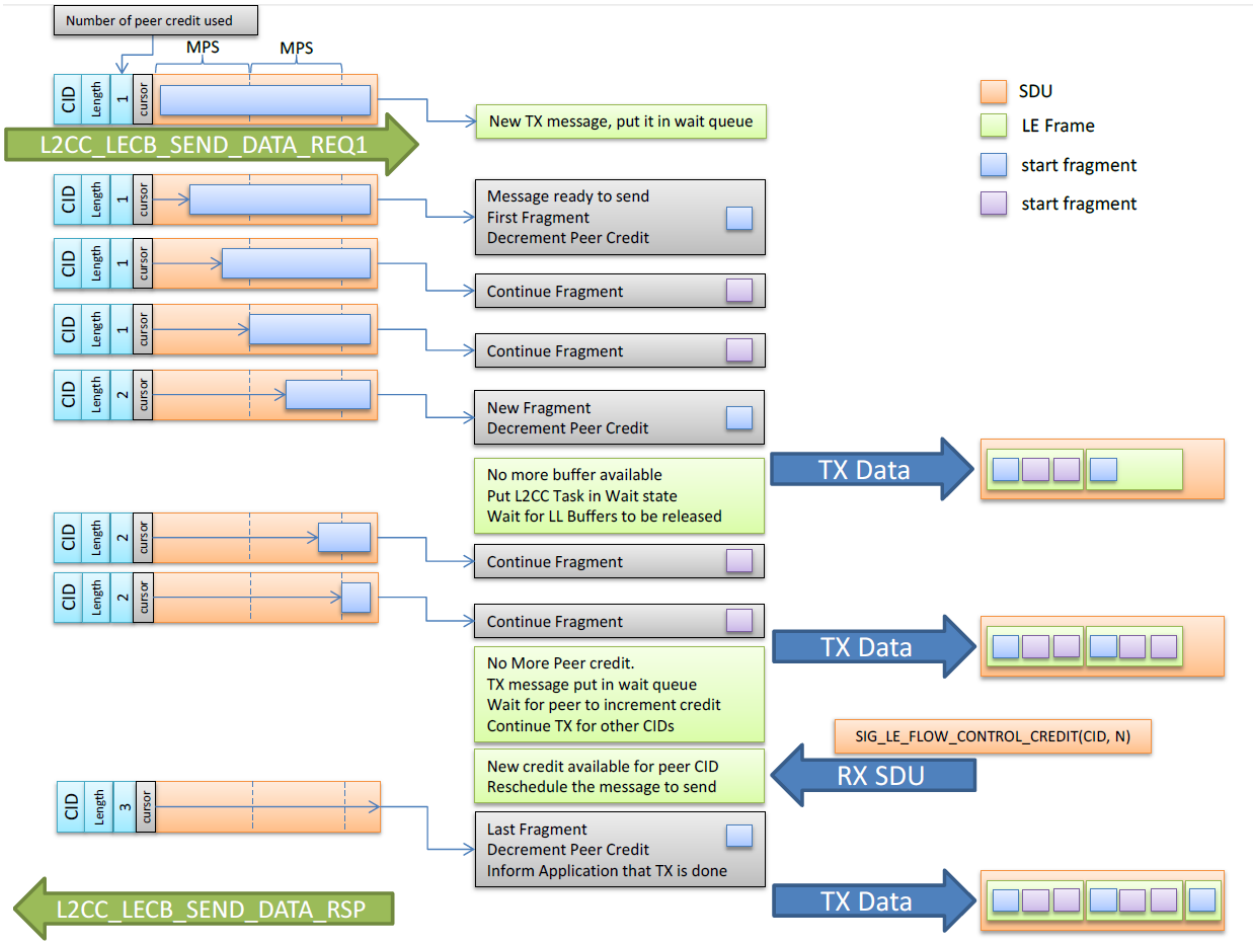


Figure 148. Transmission of an SDU to Peer Device

- Packet reception

  When receiving a packet, the L2CC receive procedure verifies with the GAPC module (using native functions) whether the CID is available. LE frame (segment) per LE frame, the number of local credits is decremented (see Figure 149 on page 168). At end of the LE frame reception, a mechanism verifies (according to local MPS) whether some credit can be automatically incremented. Condition: (total number of credits decremented) < (data length received / MPS).

Between each LE Frame received, the L2CC task can receive messages for other channels (ATT, SMP, SIG or other CID) Finally, the L2CC task informs the destination task (which registers the LE credit based channel) that a packet has been received, and how many credits have been used.

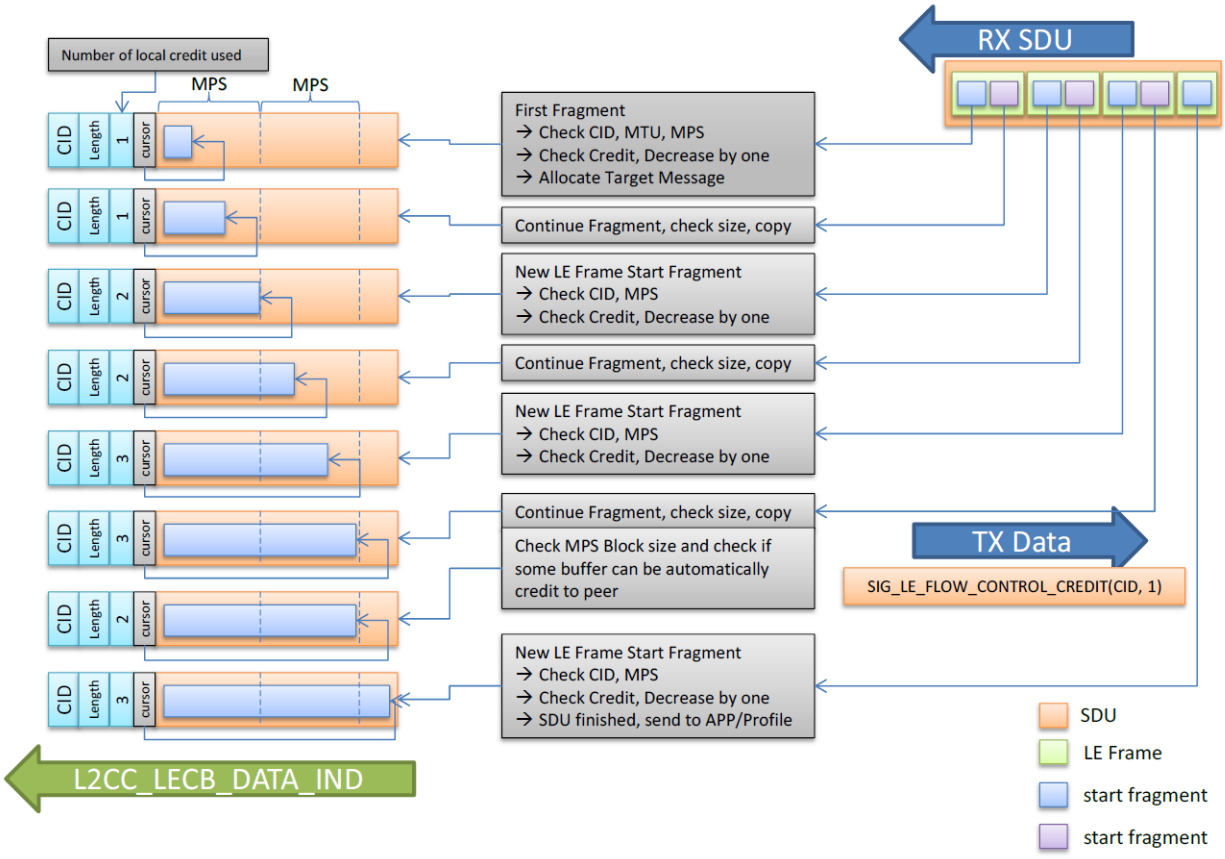


Figure 149. SDU Reception from Peer Device

6.4.5.5 Credit Management

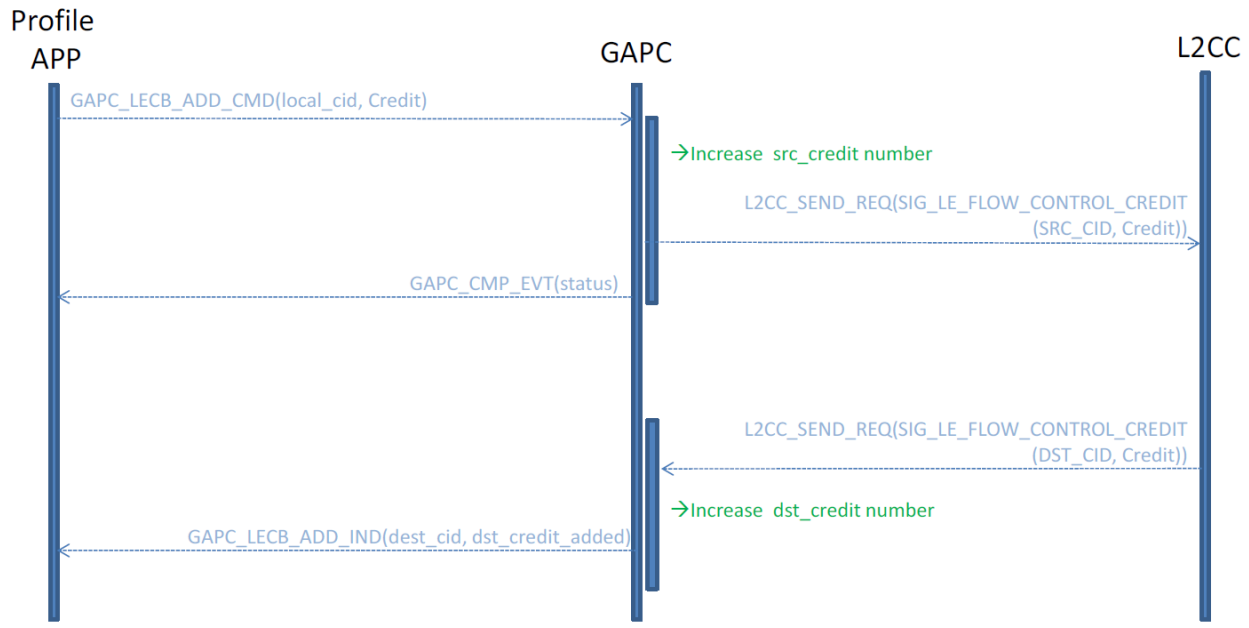


Figure 150. LE Credit Management

Figure 150 shows how to manage credit on an LE credit based connection:

- One of the devices can increase its local number of credits; when this is done, the peer device will be informed that the credit number has been updated.
- When the peer device updates its credit count, the local device increases the destination credit count, and the task that manages the LE credit based connection is informed about the number of relative credits added.

6.4.5.6 LE Ping

The LE ping feature is handled by the lower layers (see Figure 151 on page 169). The application can configure or retrieve the authenticated payload timeout (10 ms step) through the GAP interface (see Figure 152 on page 170).

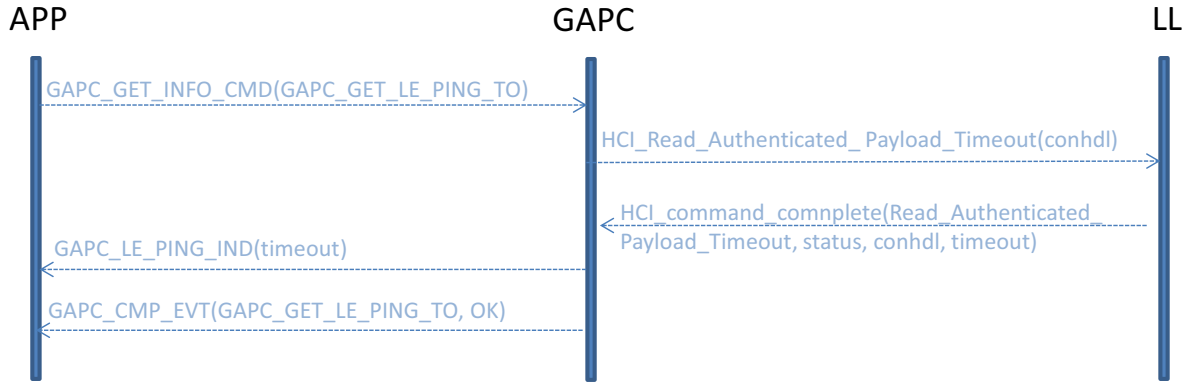


Figure 151. Retrieve LE Ping Authenticated Payload Timeout from LL

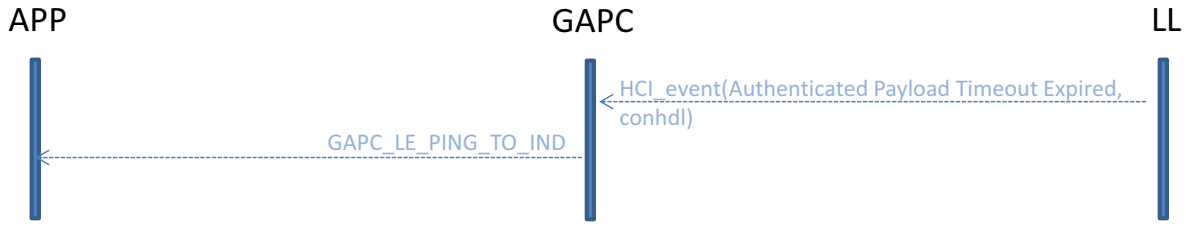


Figure 152. Inform Application about LE Ping Authenticated Payload Timeout Expiration

6.4.5.7 LE Data Packet Length Extension

The size of LE data packets can be negotiated over the Bluetooth low energy technology link. The preferred LE data packet size is set by the application when setting the device configuration (see Section 6.4.5.11.2, “Device Configuration” on page 174). When the link is established, the application can try to (re)negotiate the LE data packet size using `GAPC_SET_LE_PKT_SIZE_CMD`.

When the Link size is updated, `GAPC_LE_PKT_SIZE_IND` is triggered. It does not change the fragmentation mechanism in L2CAP since it preferentially uses the fragmentation mechanism provided by the lower layers.

6.4.5.8 Profile Management

Our stack implementation supports a large number of profiles; for each profile, a minimum of two tasks are implemented: one for the profile, and one for the client. Those tasks will support multiple connections. In a normal use case, an application will not support all profiles and services at the same time; the number of profiles must be limited to a certain number of profile tasks. To do so, an array in the generic access profile environment variable is used to manage profile tasks. This array contains the task descriptors and a pointer to the environment heap. At start-up, the application decides on the profiles that can be started (both client and services tasks). For services tasks, this means that the corresponding attribute database will be loaded, and a minimum authentication level is selected.

- No authentication required
- Unauthenticated link required
- Authenticated link required
- Secure connection link required

The profile manages allocation of its task state array, and its environment memory (static and for each links). The number of profile tasks managed by the generic access profile is managed by a compilation flag. An overview of a profile task descriptor is shown in Figure 153.

NOTE: For integration purposes, the customer must allow this to be runtime configurable.

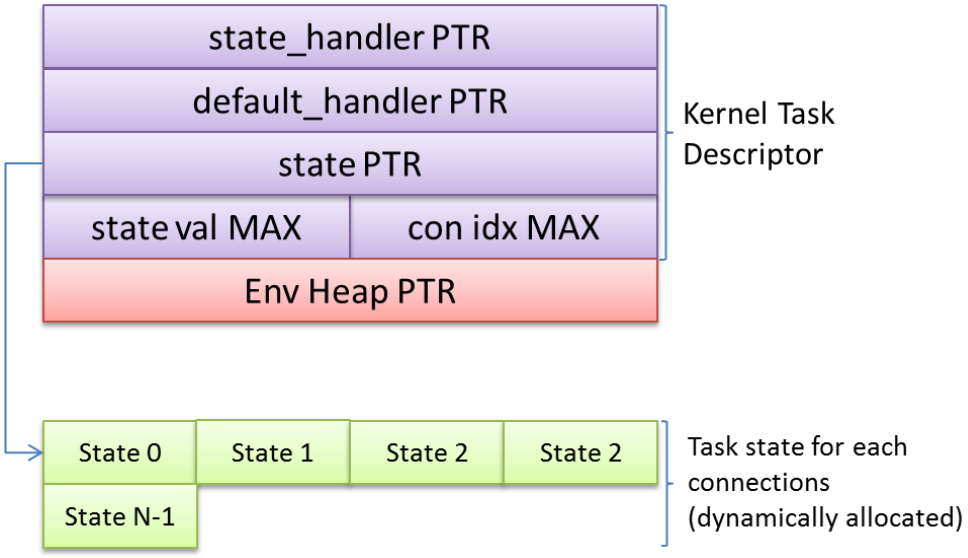


Figure 153. Overview of a Profile Task Descriptor in GAP Profile Task Array

NOTE: When all profile tasks has been affected, an application requesting to use another profile will receive an Out Of Memory error.

To fix the profile API, instead of using a task number, a profile id (statically set) is used. This ID will be unique and not be used by another task. Profile task registration is illustrated in Figure 154 on page 171.

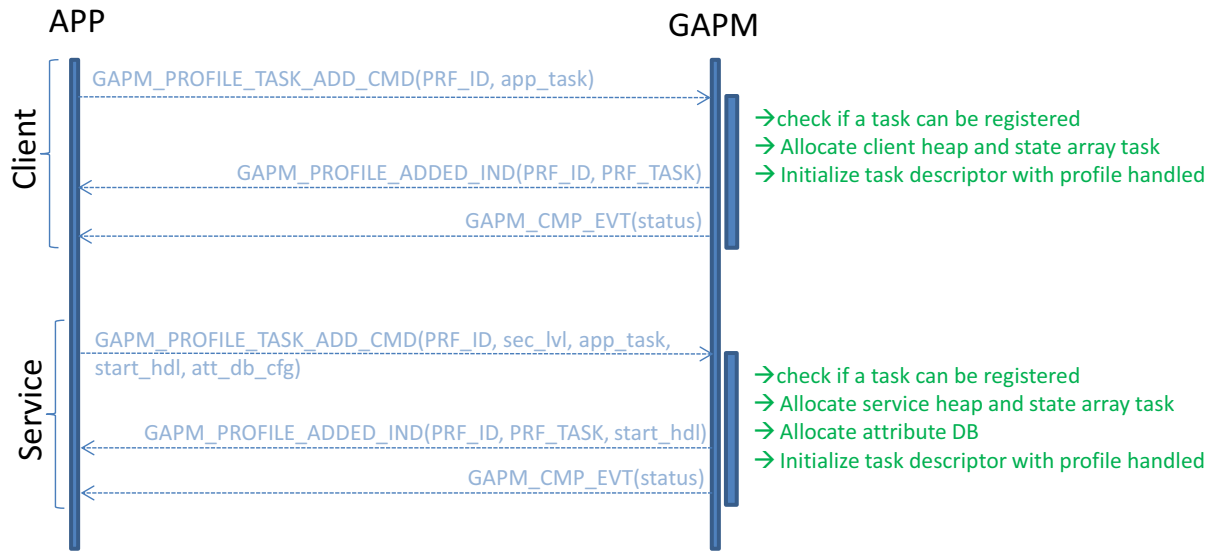


Figure 154. Profile Task Registration

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

task id from the profile id. When a profile is registered, it is natively informed about link establishment (to allocate environment) and termination. An example of profile task registration is shown in Figure 155 on page 172.

NOTE: When the system is reset, all registered tasks are remove and profiles are cleaned up.

By default, profile task descriptors are initialized without any handler and without any task id. This ensures that when a task is not registered, any message kernel to this task will be ignored.

NOTE: If GTL receives a message on a non-registered profile identifier, it will answer with a generic error message.

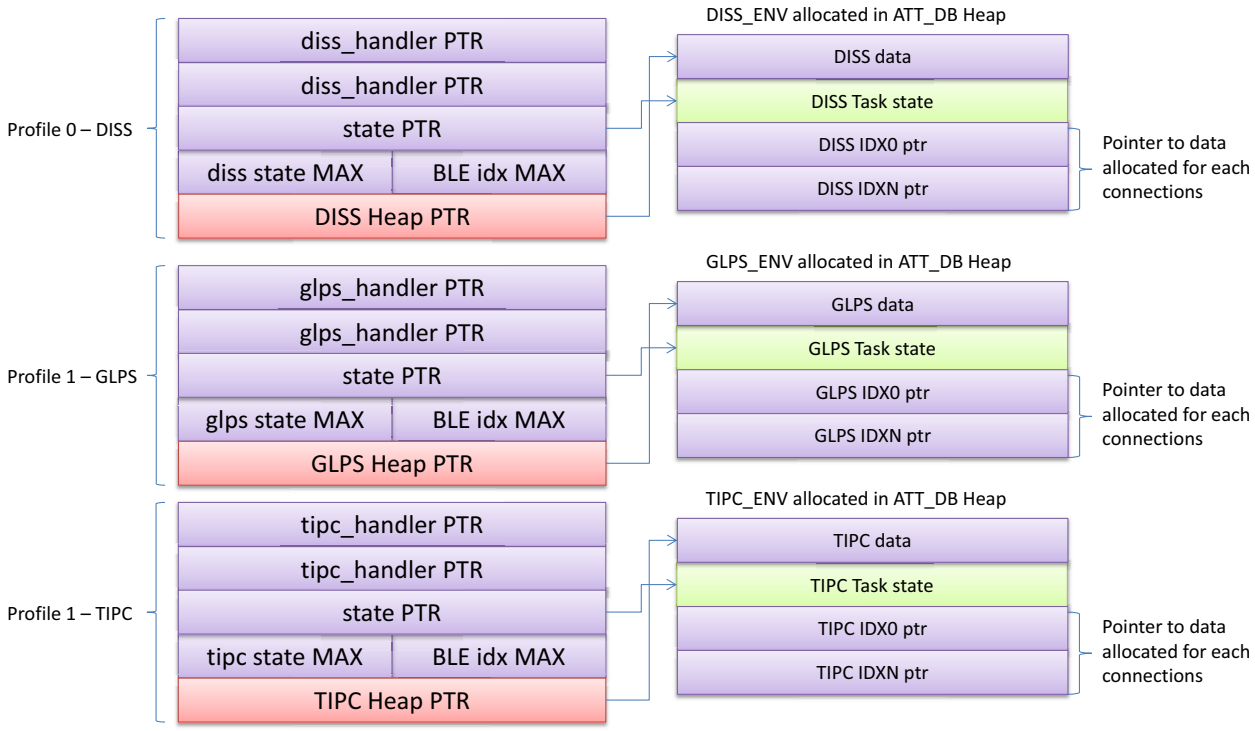


Figure 155. Example of Profile Task Registration

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

6.4.5.9 GAP service database

GAP service (UUID = 0x1800) will be represented as an attribute service in the attribute database. Depending on the role of the device, certain attribute characteristics are required in the service definition, as seen in Table 60.

Table 60. GAP Characteristics

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

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

6.4.5.10 GAP Environment Variables

6.4.5.10.1 GAP Manager Environment

GAP Manager environment variables are shown in Table 61.

Table 61. GAPM Environment variables

Type	Value	Comment
ke_msg*	CFG operation	Operation used to configure System, use encryption block, get system information
ke_msg*	AIR operation	Operation used to perform advertising, scanning or connection init activity
uint16_t	Start_hdl	GAP Service start handle
gap_sec_key	IRK	IRK used for resolvable random BD address generation
bd_addr	addr	Current BD address (private or public)
gap_bdaddr*	scan_filter	Scan filtering Array
co_list	LECB channels	List that contains list of LE Credit Based channels
uint8_t	role	Current device role
uint8_t	nb_mst_con	Number of master connections
uint8_t	nb_slave_con	Number of slave connections
uint16_t	renew_dur	Duration of resolvable address before regenerate it.
uint8_t	Flags	Flag field for: • Addr is private or public • Host Privacy Enabled • Controller Privacy Enabled • Use resolvable/non resolvable address • Slave preferred param present • Address Renew timer started

6.4.5.10.2 GAP Controller Environment

GAP Controller environment variables are shown in Table 62.

Table 62. GAPC Environment variables

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

6.4.5.10.3 GAP Profiles Environment

GAP Profile environment variables are shown in Table 63.

Table 63. GAP Profiles Environment variables

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

6.4.5.11 Device initialization

6.4.5.11.1 Software Reset

At system start-up, to initialize software state machines, a software reset command is sent. This command also initializes the attribute database; after a software reset, the device attribute database is empty, and device configuration and profile configuration are performed.

6.4.5.11.2 Device Configuration

At system start-up, after sending the software reset command, the device is set up using the set device configuration command. Configuration of a device can be updated only if there is no on-going connection.

- Role: five roles allowed, as shown in Table 64 on page 175.

NOTE: The device can support all GAP roles (advertising, scanning, initiating and connected) simultaneously, sharing use of the RF front-end between the different application use cases. For example, this allows a device to be a master of one connection and a slave of a different connection, or to start scanning and advertising activity at the same time.

Table 64. Device Roles

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

- Device Privacy:
  - Device IRK: used to generate random address (only valid for host Privacy 1.1)
  - Privacy managed by host (privacy 1.1), by controller (privacy 1.2) or disabled
  - Renew address timer duration

- Device Address: (if privacy disabled or managed by controller)
  - Device address type
  - Device static address (if address type is random)

- Packet Size: Maximum MTU allowed by device (mini = 23 bytes, max = 2048)
- GAP DB configuration:
  - GAP DB start handle (`0x0000` – dynamically allocated)
  - Appearance write permissions
  - Device name write permissions + Device name max length
  - Peripheral preferred connection parameters present + read permissions

- GATT DB Configuration:
  - GATT DB start handle (`0x0000` – dynamically allocated)
  - Service changed characteristic present

- LE Credit Based Channel:
  - Maximum number of LE credit based channel connections that can be established
  - Maximum MTU and MPS size authorized on a local device. It also limits maximum MTU and MPS size that can be transmitted to a peer device.

NOTE: The set device configuration command recreates the GAP and GATT databases.

6.4.6 Profile Functionalities

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

Support of an LE profile depends on its specification availability, from the Bluetooth Special Interest Group (SIG). The FS of these profile implementations are beyond the scope of this document.

Some guidelines for profile implementation:

- Due to Bluetooth topology, client and server profiles are multi-instantiated tasks.
- Profiles have to manage environment memory by allocating it in an ATT Heap. Memory is used for dedicated link and general configuration.
- If not enabled by GAP, the profile RAM footprint will equal zero.
- Service profile will be ready by default; enable message must be used to restore the bond data of a known device.
- A profile is not aware of its task identifier, the in message handler; the destination id must be used to retrieve its task identifier, or eventually request it to GAP through the native API.
- It is recommended to use the operation mechanism (see Section 6.4.8.2, “Operation Model” on page 177) to optimize profile memory usage.

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

6.4.7 Message API naming requirements

To have a standard message interface between each task:

- The upper layer interface uses the API from the lower layer one (it is not allowed for a lower layer interface to use an upper layer api).
- A request (_REQ suffix) or a command (_CMD suffix) from an API user needs to be answered by the task: a command (_CMD suffix) is finished by sending a complete event (_CMP_EVT suffix) (see Figure 156), and a request (_REQ suffix) is finished when a response message (_RSP suffix) is sent (see Figure 157).

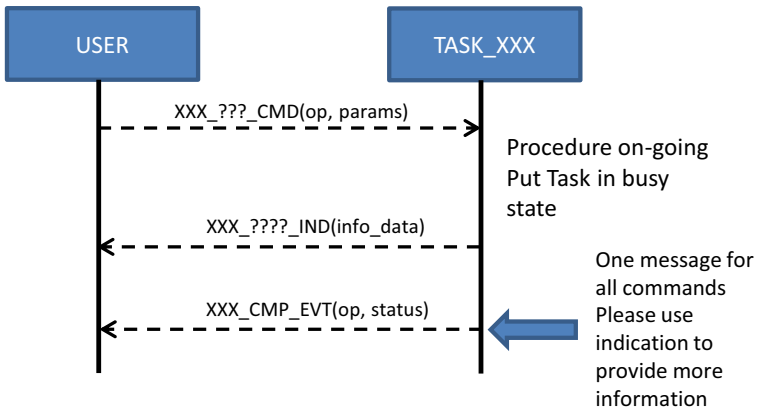


Figure 156. Command Operation Finished with a Complete Event

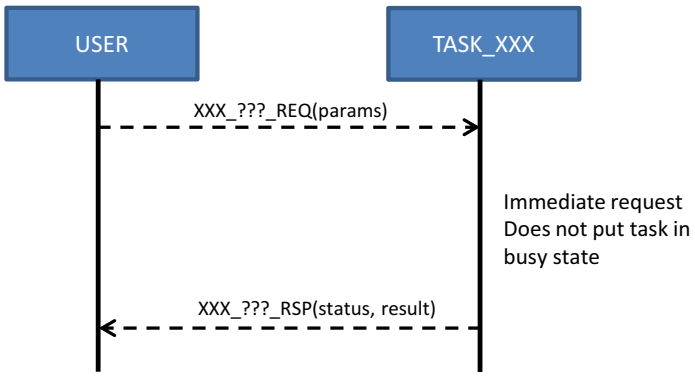


Figure 157. Request Message which is Answered By Response Message

A task can inform an upper layer task using an indication message (_IND suffix); or when information is needed by a task, a request indication message (_REQ_IND suffix) can be raised and shall be answered using a confirmation message (_CFM suffix) (see Figure 158).

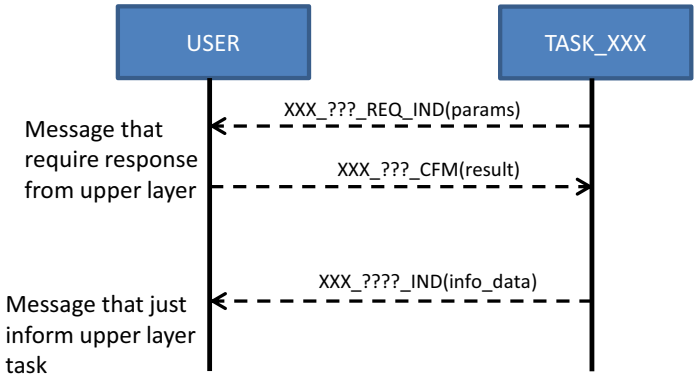


Figure 158. Message API use by a Task to Communicate with an Upper Layer

6.4.8 Memory Optimization

Bluetooth host software memory is optimized for allowing the system to shut down some memory blocks when sleeping between Bluetooth events. This feature can be used thanks to the kernel memory heap segmentation.

6.4.8.1 Connection Oriented Task

The environment variables for tasks related to a connection are allocated at connection, and removed as soon as connection is stopped. Those environment variables must not contain values used only during specific operations such as pairing or connection update. These variables will be allocated in the kernel environment heap.

6.4.8.2 Operation Model

An operation is a command that will be executed by a task. This command contains parameters that must be used during its execution. Instead of copying the parameter into the task environment, the command message is stored until its execution is finished. See Figure 159 for the operation life cycle.

Thanks to this model:

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

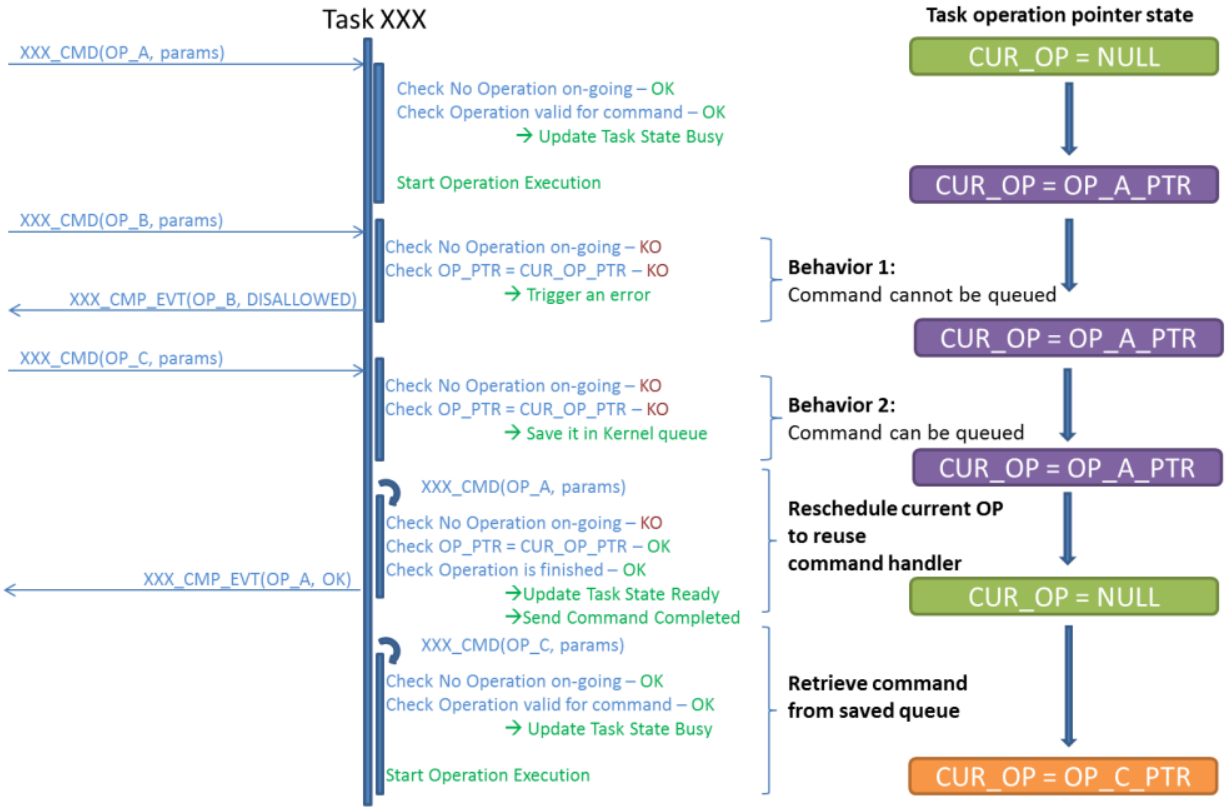


Figure 159. Operation Life Cycle

CHAPTER 7

Custom Protocols

7.1 Overview

In addition to standard RF protocol support, a number of custom protocols have been defined for the RSL10 ecosystem. These protocols are designed to handle use cases that are not typically easy to support using Bluetooth low energy technology. These protocols are supported by header files, libraries, and sample applications that demonstrate how the protocols can be used in a larger system.

Custom protocols supported include:

Audio Stream Broadcast (Remote Microphone Custom Protocol)

This is a custom audio transmission protocol that allows the broadcast of either a mono or a stereo audio stream, where the transmitting device is responsible for the majority of the RF traffic. This protocol is designed to limit the active RF time on any receiving devices, at the expense of higher traffic handled by the transmitting device.

For more information on this custom protocol, see Section 7.2, “Audio Stream Broadcast Custom Protocol”.

Low-Latency

This is a custom transmission protocol that is used to establish a low-latency bidirectional connection to provide transfers of data between two devices that contain the RSL10 SoC, with the minium feasible delay.

For more information on this custom protocol, see Section 7.3, “Low-Latency Custom Protocol”.

7.2 Audio Stream Broadcast Custom Protocol

The audio stream broadcast custom protocol enables audio transmission that can carry either a mono or a stereo audio stream to one or more devices. This protocol is supported by the firmware and sample code listed in Table 65.

Table 65. Audio Stream Broadcast Custom Protocol Objects

Object	File Name or Project	Description
Headers	*rm_pkt.h*	Header file that needs to be included to use this protocol
Source	*config_data.c, rm_event.c, rm_pkt_hdl.c*	Source files containing the implementation of this protocol

Table 65. Audio Stream Broadcast Custom Protocol Objects

Object	File Name or Project	Description
Library	*remote_micLib.a*	Library file that a user application needs to link against if using this protocol. The transmitting device side has two ways to access audio data using the payloadFlowRequest parameter: through `RM_APP_REQUEST` or `RM_PRO_REQUEST`. The corresponding delay parameter, preFetchDelay, is set according to the audio path application. When using RM_APP_REQUEST, the application can call an API function to provide data to the library. For RM_PRO_REQUEST, the protocol calls a callback function from the application to obtain its audio data.
Sample Code	*remote_mic_trx_coded*, *remote_mic_rx_raw*, *remote_mic_rx_coex*, *remote_mic_tx_raw*, *remote_mic_tx_coex*	Demonstration code showing use of this protocol as a: • Transmitter of a mono audio data channel • Receiver of an audio data channel • Transmitter of two pre-encoded audio data channels, typically for stereo audio transmission A number of data source and sink configurations (using both raw and pre-encoded data) are available. Several examples that demonstrate coexistence of this protocol with Bluetooth traffic are also provided.

7.2.1 Audio Stream Broadcast Packet Structure

The audio stream broadcast custom protocol uses a simple packet structure, which limits the additional packet transmission information to a minimum beyond what is necessary to transmit the packet payload information. The packet structure is shown in Figure 160, with information on the included components provided in Table 66.



Figure 160. Audio Stream Broadcast Packet Structure Layout

Table 66. Audio Stream Broadcast Packet Structure Details

Field	Length (bytes)	Description
Preamble	1	Bit sequence used to synchronize the demodulator to the incoming bit stream (0x55).
Address	4	Address information for the stream, used to differentiate between different stream sources.

TABLE 66. AUDIO STREAM BROADCAST PACKET STRUCTURE DETAILS

Field	Length (bytes)	Description
Header	1	Bits 1:0 - Transmission ID - Circular count that aligns an update with its transmission interval
		Bits 3:2 - Channel designation; nominal usage assigns the following designations: • 0b00 - Reserved • 0b01 - Left or Mono • 0b10 - Right • 0b11 - Other
		Bits 7:4 - RFU (reserved for future use)
Payload	Variable	Payload data to be transferred using this protocol. The payload size is assumed to be known in advance so as not to require transferring the length with each packet. NOTE: If the payload size is not known up front, it is recommended that the payload length used is included in the payload data so that receivers can determine the correct length for received packets.
CRC	2	CRC-CCITT value calculated over the header and payload from this packet

7.2.2 AUDIO STREAM BROADCAST TRANSMISSION STRUCTURE

7.2.2.1 Packet Sets

Data transmitted using the audio stream broadcast custom protocol is grouped into packets, with each sample sent as part of a redundant packet set. For each channel to be transmitted, this packet set contains a packet carrying the payload for the current transmission interval, and a packet carrying the payload for the previous transmission interval.

For a typical stereo audio transfer, the packet set will consist of four packets:

1. The left-channel's audio data for the current transmission interval
2. The left-channel's audio data for the previous transmission interval
3. The right-channel's audio data for the current transmission interval
4. The right-channel's audio data for the previous transmission interval

The left or right channel can be selected through the audioChnl parameter.

If no data is available for the previous transmission interval (as would be the case at the beginning of a transfer), the data packet for the current transmission interval is used in its place, to ensure consistency of the transmission structure.

7.2.2.2 RF Physical Layer Configuration

The audio stream broadcast radio streaming protocol is required to:

- Maximize the available RX sensitivity to provide a more robust link
- Reduce the power consumption for receiving devices (typically by reducing the radio on-time)
- Provide sufficient throughput to support the required audio data channels

To meet these somewhat conflicting requirements, the configuration described in Table 67 on page 182 is used for this custom protocol.

Table 67. Physical Layer Configuration for THE AUDIO STREAM BROADCAST CUSTOM PROTOCOL

Parameter	Value	Description and Notes
TX Power	+6 dBm	Typical; can be lowered based on the needs of the user's device network.
Modulation Scheme	GFSK	-
Modulation Index	0.32	Nominal modulation index for compatibility with Ezairo 7150 SL implementations of this protocol. For RSL10 only connections, an optional configuration that uses a modulation index of 0.5 is provided for improved performance.
Symbol Rate	2 Mbps	-
Channels	40	Aligned with Bluetooth low energy channels to simplify coexistence between this protocol and Bluetooth low energy traffic. Channels aligned between 2402 and 2480 MHz.
Channel Spacing	2 MHz	
Channel Hop Sequence	-	Predefined hop sequences are used for transmissions and retransmissions, and can be configured to ensure that all channels are used by this protocol, and all transmissions or retransmissions of a given packet are widely spaced across the channel set. For compatibility with Ezairo 7150 SL implementations, a hop sequence of 7 values is used.

NOTE: The physical layer for the audio stream broadcast custom protocol, as implemented, does not include whitening of the RF data. If you develop a use case that would include mostly ones or zeros in the RF traffic, adding data whitening to this protocol will likely improve the reliability of traffic broadcast using this protocol.

7.2.2.3 RF Transmission Structure

Audio streaming using this custom protocol is centered around an asymmetric use of resources. For this protocol, the transmitter unconditionally retransmits data four times in an attempt to improve the likelihood of a successful data reception. Transmission of data is defined by a set of RF transmission parameters as listed in Table 68.

Table 68. RF Transmission Parameters

Parameter	Value	Notes
Transmission Interval	10 ms	The time interval between primary transmissions of packet sets; the start of the transmission interval for a packet set is defined as the synchronization point for the packet set transmission.
Retransmission Interval	5 ms	The time interval between the start of a primary transmission of a packet set, and the start of the retransmission of that packet set.

Confirmation of link establishment or loss (disconnection) is provided to the application using a callback function.

Prior to transmission:

- Encoded audio data for each channel is placed into a packet (see Section Section 7.2.1, "Audio Stream Broadcast Packet Structure").
- Packets of data are collected with data from other channels and previous transfers, as a packet set (see Section Section 7.2.2.1, "Packet Sets").

Each packet set is transmitted at the synchronization point, and one retransmission interval later, as shown in the example transmission sequence provided by Figure 161 on page 183. At the start of the next transmission interval, a

new packet set is created and the transmission process repeats. Each time a packet set is transmitted (including both at the start of a transmission interval and at the retransmission interval), the channel used for the transmission is updated with a fixed spacing between the transmission and retransmission, and a pre-defined channel hopping sequence is used for each transmission interval (as described in Section 7.2.2.2, “RF Physical Layer Configuration”).

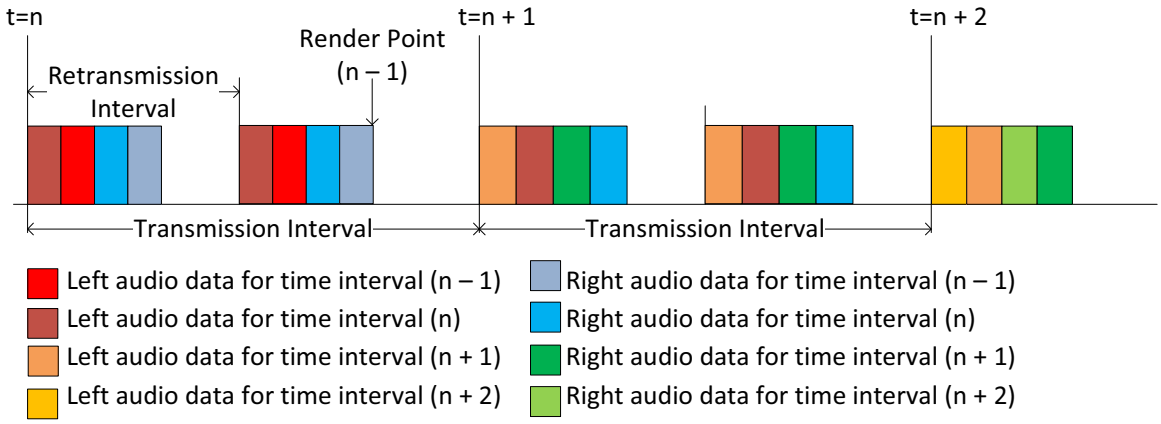


Figure 161. Example RF Transmission Sequence

To maintain an audio stream, the receiver needs to listen only to those events necessary to obtain a complete set of data for its channel.

Figure 162 on page 183 shows an example of the receiver behavior when trying to receive one packet. In this example, the receiver is attempting to receive data for the left channel, and fails to receive data for transmission interval (n) in both transmission slots of transmission interval (n). This data is then received in transmission interval (n + 1), along with the data for transmission interval (n+1). The receiver does not listen for more data during the retransmission interval, and only listens for the data for the (n + 2) interval in the subsequent interval. In this way, the receiver only listens when new data for its channel is available.

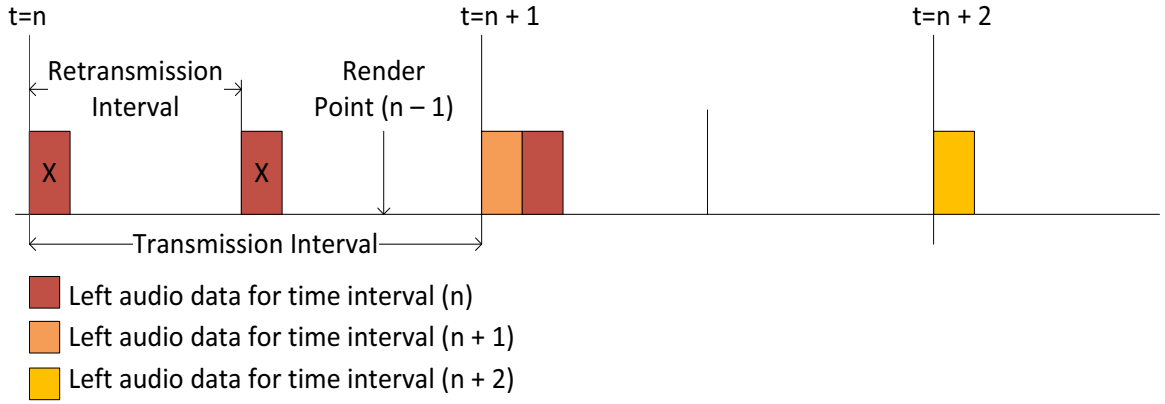


Figure 162. Example RF Reception Sequence

Only after all four potential transmissions have completed can the device then render its audio data. If this data is available earlier, it needs to be held back, to maintain consistent timing. When rendering stereo data, the receiving device also delays its rendering point, so that this point occurs after all packets from the last packet set containing a given packet have been retransmitted, as shown in Figure 161 on page 183.

The library provides audio data using a rendering delay. This delay timing can be changed using the renderDelay parameter. The library will automatically adjust the rendering time for the left and right microphones; the application is not responsible for this. Rendering delay timing allows the left and right microphones to deliver audio data to the application simultaneously. The time difference between left and right is taken into account in the protocol implementation.

The library will deliver audio data to the application through a callback function, which also provides the status of the date.The application has no need to consider timing when no packet is received, because the protocol retains the timing. At rendering time, the protocol provides one of three different status results for the packet: `good packet`, `bad CRC packet`, or `no packet` (timeout). Then the application can decide if it wants to use any of the PLC algorithms.

7.2.3 Audio Stream Broadcast API

This reference material presents a detailed description of all the external API functions in the audio stream broadcast custom protocol library (see Table 69), including calling parameters, returned values, and assumptions.

Table 69. Audio Stream Broadcast Protocol Library Reference Functions

Function	Description	Reference
RM_Configure	Configure protocol environment based on input from application	7.2.3.1 on p. 184
RM_Disable	Disable the protocol	7.2.3.2 on p. 185
RM_Enable	Enable the protocol	7.2.3.3 on p. 185
RM_EventHandler	Protocol event handler	7.2.3.4 on p. 185
RM_StatusHandler	Protocol status update handler	7.2.3.5 on p. 186

7.2.3.1 RM_Configure

Configure protocol environment based on input from application

Type	Function
Include File	`#include <rm_pkt.h>`
Source File	rm_event.c
Template	`uint8_t RM_Configure(struct rm_param_tag param, struct rm_callback callback)`
Description	Configure protocol environment based on input from application
Inputs	param = Application input parameters callback = Application call back functions
Outputs	return value = 0 if it configures successfully, error value otherwise

Assumptions	None

Example	`struct app_env_tag app_env;`
	`struct rm_callback callback;`
	
	`/* Define the application environment and callback for the`
	`* audio broadcast streaming custom protocol here... */`
	
	`/* Configure the custom protocol for use by the application */`
	`RM_Configure(&app_env.rm_param, callback);`

7.2.3.2 RM_Disable

Disable the protocol

Type	Function
Include File	`#include <rm_pkt.h>`
Source File	rm_event.c
Template	`uint8_t RM_Disable(void)`
Description	Disable the protocol
Inputs	None
Outputs	return value = 0 if it disables successfully, error value otherwise
Assumptions	None
Example	`/* Disable the custom protocol */` `RM_Disable();`

7.2.3.3 RM_Enable

Enable the protocol

Type	Function
Include File	`#include <rm_pkt.h>`
Source File	rm_event.c
Template	`uint8_t RM_Enable(uint16_t offset)`
Description	Enable the protocol
Inputs	offset = Offset instant in micro second
Outputs	return value = 0 if it enables successfully, error value otherwise
Assumptions	None

Example	`/* Configure and enable the custom protocol for use by the application */`
	`RM_Configure(&app_env.cp_param, callback);`
	`RM_Enable(500);`

7.2.3.4 RM_EventHandler

Protocol event handler

Type	Function
Include File	`#include <rm_pkt.h>`
Source File	rm_event.c

Template	`uint8_t RM_EventHandler(uint8_t type, uint8_t *length, uint8_t *ptr)`
Description	Protocol event handler
Inputs	None
Outputs	return value = 0 if it handles successfully, error value otherwise
Assumptions	None

Example	`/* Setup payload data for the left channel */`
	`RM_EventHandler(RM_TX_PAYLOAD_READY_LEFT, &length,`
	`(uint8_t *) &spi_buf[0]);`

7.2.3.5 RM_StatusHandler

Protocol status update handler

Type	Function
Include File	`#include <rm_pkt.h>`
Source File	rm_event.c
Template	`void RM_StatusHandler(void)`
Description	Protocol status update handler
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Main loop: Handle custom protocol events */`
	`while (1)`
	`{`
	` Sys_Watchdog_Refresh();`
	` RM_StatusHandler();`
	` SYS_WAIT_FOR_EVENT;`
	`}`

7.3 Low-Latency Custom Protocol

The low-latency custom protocol provides a minimum latency bidirectional connection between two devices based on the RSL10 SoC. This protocol provides a means for point-to-point audio streaming. The protocol's main use cases include, but are not limited to, the following:

- Ear-to-ear CROS and BiCROS uses
- Beamforming and directional microphones
- Quick control or algorithm data exchanges between two devices to enable coordinated signal processing

The low-latency custom protocol also demonstrates the use of the RSL10 RF front end, making it easier for users to create and implement their own individual protocols.

All parameters of the protocol are configured through APIs. The target has the lowest possible delay, and the flushable protocol cannot guarantee data transmission between points, as data that cannot be transmitted in the desired window is simply discarded. This differs from Bluetooth low energy technology, which guarantees the arrival of all data. In addition to ensuring that only data that is still relevant is received, transmitting audio data through the low-latency custom protocol instead of Bluetooth low energy ensures that the data remains time synchronized between the two sides of the link. This significantly simplifies synchronization between data sample across multiple devices.

This symmetric protocol is supported by the firmware and sample code listed in Table 70.

Table 70. Low-Latency Custom Protocol Objects

Object	File Name or Project	Description
Headers	*cp_pkt.h*	Header file that needs to be included to use this protocol
Source	*config_data.c, cp_event.c, cp_pkt_hdl.c*	Source files containing the implementation of this protocol
Library	*custom_protocolLib.a*	Library file that a user application needs to link against if using this protocol
Sample Code	*custom_protocol_trx*	Demonstration code showing use of this protocol to provide a complete audio path with the custom protocol, routing audio

7.3.1 Low-Latency Protocol Physical Layer

The Low-Latency custom protocol is designed to use the RF characteristics already qualified for use on the RSL10 device with Bluetooth low energy technologies, with some flexibility provided to enable users to meet a variety of use cases. The physical layer configuration for this protocol is described in Table 71.

Table 71. Physical Layer Configuration for the Low Latency Custom Protocol

Parameter	Value	Description and Notes
Modulation Scheme	GFSK	-
Modulation Index	0.5	Matches Bluetooth low energy configuration
Symbol Rate	500 kbps, 1 Mbps, 2 Mbps	Configurable through the protocol API
Channels	40	Aligned with Bluetooth low energy channels to simplify coexistence between this protocol and Bluetooth low energy traffic. Channels aligned between 2402 and 2480 MHz.
Channel Spacing	2 MHz	
Channel Hop Sequence	-	During data transmission, frequency hopping is used, with the number and list of channels determined through the provided API. There are two frequency hopping lists: one for the main transmissions, and one for re-transmissions. In addition, the connection establishment phase and the connected phase have different frequency hopping lists, as explained in Section 7.3.3, “Low-Latency Protocol Link Layer Structure” on page 188.

7.3.2 Low-Latency Protocol Packet Structure

The packet format for this protocol is:

Preamble (1 octet)	Sync word (4 octets)	Header (2 octets)	Data (variable)	CRC (2 octets)

- Preamble: 1 octet. Either 0x55 or 0xAA.
- Sync word: 4 octets. The Sync word has two roles: one for packet detection, and the other as an address.
- Header: 2 octets. The first octet is used for the packet control data; the second octet indicates the length of the data field.

Table 72. First Octet of Packet Header

Bit Number	Name of Field	Meaning
0	SN	Sequence number of first (0) or second (1) packet.
1	ASN	If ACK=1, the sequence number of the packet is acknowledged.
2	Data	Library file that a user application needs to link with when using this protocol. Packet includes data (1) or in-band signalling (0).
3	ACK	Indicates whether an ACK (1) is conveyed.
4-7	Hop Cluster Num	Header Cluster number of first stage channel hopping.

- Data: Any desired data sent through this protocol.

  Since the main purpose of this protocol is transmitting audio with low latency, data in the sample code is a coded audio frame (for example, 16 octets per 2 ms of data at a 64 Kbps coding rate). This audio data can be replaced as needed by control data, signaled using the one-bit Data bit-field in the header.

- CRC: 2 octets. CRC-CCITT is calculated over the header and data sections of the low-latency protocol packet.

7.3.3 Low-Latency Protocol Link Layer Structure

The two peer devices that communicate using this protocol are the master and slave devices. The master device sends data/audio packets in consecutive transmission intervals, while the channel frequency changes for each interval. Once the slave receives the master's audio/data packet, the slave sends back an acknowledgment packet at the same channel frequency. If the master device does not receive the acknowledgement as expected, the master retransmits the payload on another frequency channel. Figure 163 illustrates this mechanism.

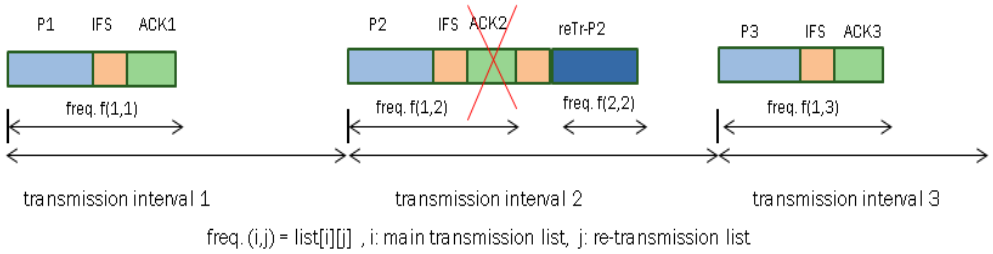


Figure 163. Data Packet Reception and Acknowledgement

The low-latency custom protocol uses two sets, each with two lists of frequencies, for controlling the frequency hopping configuration. In each set, one list is used to select the frequency for the main transmissions, and the second is used for re-transmissions. The first set includes two lists of four frequencies each, which are used during link establishment. The second set includes two lists of up to a maximum of 36 frequency channels, which are used while streaming data.

At the beginning of link establishment, the master device sends packets via channels present on the first list of the first set, and increments the frequency hop number at every connection interval. During each connection interval following the main transmission, after a pre-defined inter frame space (IFS), the master device waits to receive an acknowledgement from the slave device at the same frequency. If no acknowledgement is received after the IFS, the

main transmission packet will be sent again, on the same channel index but from the re-transmission list. The master device continues sending the transmission, cycling through the 8 channels from the transmission and retransmission lists, and as long as it does not receive any response from the slave device, it repeats the same sequence.

The frequency hop counter is sent in the packet header. Once the slave device receives it, the hop counter sequence enables the slave device to synchronize with the master device. When the master receives the acknowledgment from the slave device, it stops transmitting using the first set of channels, and switches to the second set of transmission lists to send packets (and re-transmit packets) using entries from lists in the second channel frequency set.

While sending data packets, a pre-configured frequency list is used for frequency hopping. Each frequency in this list is used for one transmission, with the list wrapping back to the start after all frequencies have been exhausted. Since retransmission happens on a different frequency from initial transmission, the effects of interference and fading are mitigated.

The intent behind this two-stage frequency hopping list is that this scheme makes link establishment faster, and lowers power consumption, because the receiver has no need to deal with a long channel frequency list.

7.3.4 Low-Latency Protocol Application Program Interface

The low-latency custom protocol is implemented in a static library that can be linked to any application. The interface between the library and the application is handled through the following steps:

1. Protocol configuration: At the beginning, the application provides the desired parameters to configure the protocol (library). The application can change the following parameters:
   - Protocol role: master or slave
   - Mono-directional or bidirectional (currently only mono-directional is supported)
   - Four frequency lists for the first and second phases of connection, including main transmission and retransmission. The first list of the first phase, which can have a maximum of four channels, cannot have any common channel with the other three lists.
   - Transmission interval
   - Radio data rate
   - Audio data rate
   - Access word and preamble
   - Link management parameters (packet lost low and high thresholds)
   - Status update callback function (at any change in the link status, the allocated callback function is called)
   - Event handling callback function

2. Protocol-application interface: when the protocol needs to obtain data from the application for transmission, the regarding callback function is called. Once data is received, the registered callback function is called. The application can be informed of data packet reception through three events: main transmission, re-transmission, and unsuccessful transmission (timeout). Being informed of data packet reception enables the application to assess the link. In the case of timeout, the application can call any PLC (packet loss concealment) algorithm, or receive the previous packet. On the master side, once the protocol requests data from the application for transmission, the application is responsible for managing its timing to synchronize with the activity of the radio.

   For sampling clock synchronization, the required signals for the audio sink clock counter are generated in the protocol library, and based on the phase and period interrupts. On the master side, these interrupts can be used for sampling clock calculations based on the radio frame sync, The application can leverage this to synchronize its audio peripheral interface, sampling rate converter, and encoding timing.

On the slave side, these interrupts and signals can be used in the same way. Additionally, once a phase interrupt occurs, a rendering timer can run such that audio is rendered after its expiry. Rendering time needs to be configured in free run mode, and can be re-synchronized with the ASCC phase interrupt anytime it occurs. In the case of a missed signal (timeout, retransmission), it continues rendering based on the receiver clock.

7.3.5 Low-Latency Protocol Modules/Peripheral Usage

The low-latency custom protocol library uses several system blocks as part of the protocol implementation. For proper system functionality, the user cannot use these blocks elsewhere in their application when the low-latency custom protocol is active without potentially disrupting the low-latency custom protocol. These blocks include:

- The RF front end module, which prevents the Bluetooth low energy BB (HW) from accessing the RF front end. The radio block works in the RF front-end's packet handling mode.
- Two of the general purpose timers (timers 0, 1).

7.3.6 Low-Latency Custom Protocol API

This reference material presents a detailed description of all the external API functions in the low-latency custom protocol library, including calling parameters, returned values, and assumptions.

Table 73. Low-Latency Protocol Library Reference Functions

Function	Description	Reference
CP_Configure	Configure protocol environment based on input from application	7.3.6.1 on p. 190
CP_Disable	Disable the protocol	7.3.6.2 on p. 191
CP_Enable	Enable the protocol	7.3.6.3 on p. 191
CP_EventHandler	Protocol event handler	7.3.6.4 on p. 191

7.3.6.1 CP_Configure

Configure protocol environment based on input from application

Type	Function
Include File	`#include <cp_pkt.h>`
Source File	cp_event.c
Template	`uint8_t CP_Configure(struct cp_param_tag param, struct cp_callback callback)`
Description	Configure protocol environment based on input from application
Inputs	param = Application input parameters callback = Application call back functions
Outputs	return value = 0 if it configures successfully, error value otherwise
Assumptions	None

Example	`struct app_env_tag app_env;`
	`struct cp_callback callback;`
	
	`/* Define the application environment and callback for the`
	`* low-latency custom protocol here... */`
	
	`/* Configure the custom protocol for application use */`
	`CP_Configure(&app_env.cp_param, callback);`

7.3.6.2 CP_Disable

Disable the protocol

Type	Function
Include File	`#include <cp_pkt.h>`
Source File	cp_event.c
Template	`uint8_t CP_Disable(void)`
Description	Disable the protocol
Inputs	None
Outputs	return value = 0 if it disables successfully, error value otherwise
Assumptions	None
Example	`/* Disable the custom protocol */` `CP_Disable();`

7.3.6.3 CP_Enable

Enable the protocol

Type	Function
Include File	`#include <cp_pkt.h>`
Source File	cp_event.c
Template	`uint8_t CP_Enable(uint16_t offset)`
Description	Enable the protocol
Inputs	offset = Offset instant in micro second
Outputs	return value = 0 if it enables successfully, error value otherwise
Assumptions	None

Example	`/* Configure and enable the custom protocol for application use */`
	`CP_Configure(&app_env.cp_param, callback);`
	`CP_Enable(500);`

7.3.6.4 CP_EventHandler

Protocol event handler

Type	Function
Include File	`#include <cp_pkt.h>`
Source File	cp_event.c
Template	`uint8_t CP_EventHandler(void)`
Description	Protocol event handler
Inputs	None
Outputs	return value = 0 if it handles successfully, error value otherwise

Assumptions	None

Example	`/* Main loop: Handle custom protocol events */`
	`while (1)`
	`{`
	`  Sys_Watchdog_Refresh();`
	`  CP_EventHandler();`
	`  SYS_WAIT_FOR_EVENT;`
	`}`

CHAPTER 8

CMSIS Implementation Library Reference

This reference chapter presents a description of the functions implemented in the standards-compliant CMSIS library. This includes calling parameters, returned values, and assumptions. These functions implement the CMSIS-required device specific functions, and extend the generic function implementations provided for the Arm Cortex-M3 processor. The generic functions provided by CMSIS are included in the CMSIS header files, and reference documentation is provided by the standard Arm CMSIS documentation (http://arm-software.github.io/CMSIS_5/Core/html/modules.html).

8.1 SystemCoreClockUpdate

Updates the variable SystemCoreClock and the FLASH_DELAY_CTRL register

Type	Function
Include File	`#include <rsl10.h>`
Source File	system_rsl10.c
Template	`void SystemCoreClockUpdate(void)`
Description	Updates the variable SystemCoreClock and the FLASH_DELAY_CTRL register. This function must be called whenever the core clock is changed during program execution.
Inputs	None
Outputs	None
Assumptions	It is safe to treat undefined clock configurations as if they are sourced from the RC oscillator. EXTCLK and JTCK should be scaled from their maximum frequencies. It is safe to assume a STANDBYCLK frequency of 32768 Hz

Example	`/* Switch the system clock source to RF clock (clearing the`
	` * EXTCLK/JTCK divisors), and refresh the system core clock`
	` * variable. */`
	`Sys_Clocks_SystemClkConfig(SYSCLK_CLKSRC_RFCLK);`
	`SystemCoreClockUpdate();`

8.2 SYSTEMINIT

Setup the system core clock variable; assumes the ROM has previously initialized the system

Type	Function
Include File	`#include <rsl10.h>`
Source File	system_rsl10.c
Template	`SystemInit`
Description	Setup the system core clock variable; assumes the ROM has previously initialized the system.
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Initialize the system */` `SystemInit();`

CHAPTER 9

System Library Reference

This reference chapter presents a detailed description of all the macros, functions and inline functions defined in the ARM® Cortex®-M3 processor's system library. For each macro, function or inline function, it describes calling parameters, modified registers, and returned values.

9.1 BLE_DeviceParam_Set_ADV_IFS

Definition of the function to set advertisement inter-frame space

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`BLE_DeviceParam_Set_ADV_IFS(uint32_t adv_ifs)`
Description	Definition of the function to set advertisement inter-frame space
Inputs	adv_ifs = A inter-frame space in us
Outputs	None
Assumptions	None
Example	

9.2 BLE_DEVICEPARAM_SET_ADVDELAY

Enables a fixed value for advertisement intervals by setting advDelay to zero

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`BLE_DeviceParam_Set_AdvDelay(uint8_t fixedDelayEnable)`
Description	Enables a fixed value for advertisement intervals by setting advDelay to zero. If enabled, this feature will violate the Bluetooth Low Energy specification
Inputs	fixedDelayEnable = Set to non-zero to enable a zero random AdvDelay value
Outputs	None
Assumptions	None
Example	

9.3 BLE_DEVICEPARAM_SET_CLOCKACCURACY

Definition of the function to set clock accuracy according to XTAL 48 MHz or low power clock accuracy for sleep applications

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`BLE_DeviceParam_Set_ClockAccuracy(uint16_t clockAccuracy)`
Description	Definition of the function to set clock accuracy according to XTAL 48 MHz or low power clock accuracy for sleep applications
Inputs	clockAccuracy = Clock accuracy in ppm
Outputs	None
Assumptions	None
Example	

9.4 BLE_DEVICEPARAM_SET_FORCEDCLOCKACCURACY

Definition of the function to set the sum of clock accuracy of master and slave devices

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`BLE_DeviceParam_Set_ForcedClockAccuracy(uint32_t forcedClockAccuracy)`
Description	Definition of the function to set the sum of clock accuracy of master and slave devices
Inputs	forcedClockAccuracy = The sum of clock accuracy of devices in ppm
Outputs	None
Assumptions	None
Example	

9.5 BLE_DEVICEPARAM_SET_MAXNUMRAL

The size of resolving address list

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`BLE_DeviceParam_Set_MaxNumRAL(uint8_t maxNumRAL)`
Description	The size of resolving address list
Inputs	maxNumRAL = Maximum number of devices that can be set for RAL. By default it is set to 3. For a baseband clock equal or greater than 16 MHz, it can be set up to 6.
Outputs	None
Assumptions	None
Example	

9.6 BLE_DEVICEPARAM_SET_MAXRXOCTET

Setting of default data length parameters

Type	Function	
Include File	`#include <rsl10.h>`	
Source File	rsl10_protocol.c	
Template	`BLE_DeviceParam_Set_MaxRxOctet(uint8_t maxRxOctet, uint16_t maxRxTime)`	
Description	Setting of default data length parameters	
Inputs	maxRxOctet	= Supported maximum number of bytes for RX : - maxRxTime - Supported maximum time in microsecond for RX
Outputs	return value	= Returns zero if parameters are set successfully
Assumptions	None	
Example		

9.7 DEVICE_PARAM_PREPARE

Weak definition of the function in case that application doesn't define it

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`Device_Param_Prepare(app_device_param_t *param)`
Description	Weak definition of the function in case that application doesn't define it
Inputs	param = Parameter identifier
Outputs	None
Assumptions	None
Example	`/* This weakly defined function must be replaced by any application * that wishes to use the Device_Param_Read() function */`

9.8 DEVICE_PARAM_READ

Read Bluetooth low energy parameters, security keys, and channel assessment parameters that are provided by the application or NVR3

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_protocol.c
Template	`Device_Param_Read(uint8_t requestedId, uint8_t *buf)`
Description	Read Bluetooth low energy parameters, security keys, and channel assessment parameters that are provided by the application or NVR3
Inputs	requestedId = Parameter identifier buf = Pointer to the returned value
Outputs	Return value = Indicate if requested parameter exists in flash memory
Assumptions	Application has declared Device_Param_Prepare function

Example	`/* Read the public Bluetooth address from the device parameters */`
	
	`if (Device_Param_Read(PARAM_ID_PUBLIC_BLE_ADDRESS, (uint8_t *) &tempAddr))`
	`{`
	`  /* Use the address that was read to set up the device */`
	`}`

9.9 SYS_ADC_CLEAR_BATMONSTATUS

Clear the battery monitor status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`void Sys_ADC_Clear_BATMONStatus(void)`
Description	Clear the battery monitor status
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Clear ADC new sample ready, overrun condition and battery`
	` * monitoring alarm status. */`
	`Sys_ADC_Clear_BATMONStatus();`

9.10 SYS_ADC_GET_BATMONSTATUS

Get the battery monitor status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`uint32_t Sys_ADC_Get_BATMONStatus(void)`
Description	Get the battery monitor status
Inputs	None
Outputs	return value = Current ADC_BATMON_STATUS status; compare with BATMON_ALARM_[FALSE | TRUE], ADC_OVERRUN_[FALSE | TRUE], and ADC_READY_[FALSE | TRUE]
Assumptions	None
Example	`/* Read status of ADC and battery monitoring alarm. */` `status = Sys_ADC_Get_BATMONStatus();`

9.11 SYS_ADC_GET_CONFIG

Get the control register values from ADC_CFG

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`uint32_t Sys_ADC_Get_Config(void)`
Description	Get the control register values from ADC_CFG
Inputs	None
Outputs	return value = Current ADC control setting; compare with ADC_VBAT_DIV2_[NORMAL | DUTY], ADC_[NORMAL | CONTINUOUS], and ADC_[DISABLE | PRESCALE_*]
Assumptions	None
Example	`/* ADC configuration is read. */` `status = Sys_ADC_Get_Config();`

9.12 SYS_ADC_INPUTSELECTCONFIG

Configure the input selection for an ADC channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`void Sys_ADC_InputSelectConfig(uint32_t num, uint32_t cfg)`
Description	Configure the input selection for an ADC channel
Inputs	num = Channel number; use [0 to 7] cfg = Input selection configuration; use ADC_POS_INPUT_* | ADC_NEG_INPUT_*
Outputs	None
Assumptions	None
Example	`/* Configure the input selection of ADC. */` `Sys_ADC_InputSelectConfig(0, ADC_POS_INPUT_DIO1);`

9.13 SYS_ADC_SET_BATMONCONFIG

Set the battery monitor configuration

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`void Sys_ADC_Set_BATMONConfig(uint32_t cfg)`
Description	Set the battery monitor configuration.
Inputs	cfg = ADC_BATMON configuration; use BATMON_ALARM_[NONE | COUNT1 | COUNT255] or other values shifted to ADC_BATMON_CFG_ALARM_COUNT_VALUE_Pos, SUPPLY_THRESHOLD_[LOW | MID | HIGH] or other values shifted to ADC_BATMON_CFG_SUPPLY_THRESHOLD_Pos, and BATMON_CH[6 | 7]
Outputs	None
Assumptions	None

Example	`/* Configure the battery monitoring alarm count value to 1 and low`
	` * voltage threshold to 1V and monitor channel 6. */`
	`Sys_ADC_Set_BATMONConfig(BATMON_ALARM_COUNT1 |`
	` SUPPLY_THRESHOLD_MID |`
	` BATMON_CH6);`

9.14 SYS_ADC_SET_BATMONINTCONFIG

Set the battery monitor interrupt configuration

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`void Sys_ADC_Set_BATMONIntConfig(uint32_t cfg)`
Description	Set the battery monitor interrupt configuration
Inputs	cfg = ADC_BATMON_INT_ENABLE configuration; use INT_[DIS | EBL]_ADC ADC_INT_CH*, and INT_[DIS | EBL]_BATMON_ALARM,
Outputs	None
Assumptions	None
Example	`/* Enable the ADC and BATMON alarm interrupts assigning channel * number 6 to trigger the ADC interrupt. */ Sys_ADC_Set_BATMONIntConfig(INT_EBL_ADC | ADC_INT_CH6 | INT_EBL_BATMON_ALARM);`

9.15 SYS_ADC_SET_CONFIG

Set the ADC configuration

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_adc.h
Template	`void Sys_ADC_Set_Config(uint32_t cfg)`
Description	Set the ADC configuration
Inputs	cfg = The ADC configuration; use ADC_VBAT_DIV2_[NORMAL | DUTY], ADC_[NORMAL | CONTINUOUS], and ADC_[DISABLE | PRESCALE_*]
Outputs	None
Assumptions	None
Example	`/* Configure ADC to normal mode VBAT dividing and sample the 8 * channels in rate of 200Hz*/ Sys_ADC_Set_Config(ADC_VBAT_DIV2_DUTY | ADC_NORMAL | ADC_PRESCALE_800);`

9.16 SYS_AES_CIPHER

Run AES-128 cipher engine

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_aes.h
Template	`void Sys_AES_Cipher(void)`
Description	Run AES-128 cipher engine
Inputs	None
Outputs	None

Assumptions	The baseband block should be enabled. Sys_AES_Config() has been called.

Example	/*
	Key (16-octet value MSO to LSO):
	 0x4C68384139F574D836BCF34E9DFB01BF
	Plaintext_Data (16-octet value MSO to LSO):
	 0x0213243546576879ACBDCEDFE0F10213
	Encrypted_Data (16-octet value MSO to LSO):
	 0x99AD1B5226A37E3E058E3B8E27C2C666
	*/
	
	/* Definitions from the BLE stack */
	#define EM_BASE_ADDR 0x20012000
	#define EM_BLE_ENC_PLAIN_OFFSET 0x1D0
	#define EM_BLE_ENC_CIPHER_OFFSET (EM_BLE_ENC_PLAIN_OFFSET + 0x10)
	
	/* Plain-text data */
	uint32_t plaintext[4] = {
	 0XE0F10213,
	 0XACBDCEDF,
	 0X46576879,
	 0X02132435
	};
	
	uint32_t key[4] = {
	 0x9DFB01BF,
	 0x36BCF34E,
	 0x39F574D8,
	 0x4C683841,
	};
	
	/* Enable and configure the base band block */
	BBIF->CTRL = BB_CLK_ENABLE | BBCLK_DIVIDER_8 | BB_WAKEUP;
	
	/* Copy in the exchange memory */
	memcpy((void *)(EM_BLE_ENC_PLAIN_OFFSET + EM_BASE_ADDR),
	 &plaintext[0],
	 0x10 * sizeof(uint8_t));
	
	/* Configure the AES-128 engine for ciphering with the key and the memory
	 * zone */
	Sys_AES_Config (key, EM_BLE_ENC_PLAIN_OFFSET);
	
	/* Run AES-128 encryption block */
	Sys_AES_Cipher();
	
	/* Access to the cipher-text at EM_BLE_ENC_CIPHER_OFFSET address */

9.17 SYS_AES_CONFIG

Configure AES-128 engine for a ciphering method

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_aes.h
Template	`void Sys_AES_Config(uint32_t * key, uint32_t mem_zone)`
Description	Configure AES-128 engine for a ciphering method
Inputs	key = Pointer to AES encryption 128-bit key mem_zone = Memory offset from the top of the exchange memory; points to a 32-byte array consisting of the 16-byte plain-text input, followed by the 16-byte cipher-text output
Outputs	None
Assumptions	The baseband block should be enabled.

Example	`/* Configure the AES-128 engine for ciphering with the key and the memory * zone */`
	`Sys_AES_Config (key, EM_BLE_ENC_PLAIN_OFFSET);`
	`/* Access to the cipher-text at EM_BLE_ENC_CIPHER_OFFSET address */`

9.18 SYS_ASRC_CALCPHASECNT

Calculate the phase increment value according to the mode and the input frequencies

Type	Macro
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`Sys_ASRC_CalcPhaseCnt(mode, f_src, f_sink)`
Description	Calculate the phase increment value according to the mode and the input frequencies
Inputs	mode = Configuration of the ASRC mode value; use ASRC_INT_MODE | ASRC_DEC_MODE* f_src = Source frequency or source sample number f_sink = Sink frequency or sink sample number
Outputs	return value = The phase increment value
Assumptions	None
Example	`/* Calculate the phase increment value when f_src = 4 kHz and * f_sink = 16 kHz are in mode 0. */` `result = Sys_ASRC_CalcPhaseCnt(ASRC_INT_MODE, 4, 16);`

9.19 SYS_ASRC_CHECKINPUTCONFIG

Check that the input frequencies or sample numbers are valid in the range depending on the selected mode

Type	Macro	
Include File	`#include <rsl10.h>`	
Source File	rsl10_sys_asrc.h	
Template	`Sys_ASRC_CheckInputConfig(mode, f_src, f_sink)`	
Description	Check that the input frequencies or sample numbers are valid in the range depending on the selected mode	
Inputs	mode f_src f_sink	= Configuration of the mode value; use ASRC_INT_MODE | ASRC_DEC_MODE* = Source frequency or source sample number = Sink frequency or sink sample number
Outputs	return value	= 0 if frequency range check failed; 1 otherwise
Assumptions	None	
Example	`/* Check if f_sink = 16 kHz and f_src = 4 kHz are valid pair of * frequencies in mode 0. */` `result = Sys_ASRC_CheckInputConfig(ASRC_INT_MODE, 4, 16);`	

9.20 SYS_ASRC_CONFIG

Configure the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_Config(uint32_t phase_inc, uint32_t cfg)`
Description	Configure the ASRC block
Inputs	phase_inc = The phase increment value cfg = The WDF type and ASRC mode; use ASRC_[INT_MODE | DEC_MODE*], and [LOW_DELAY | WIDE_BAND]
Outputs	None
Assumptions	None

Example	`/* Configure ASRC block for f_sink = 7 kHz and f_src = 16 kHz in`
	`* mode = 3. Coefficient setting for WDF1 is for wide band`
	`* response. */`
	`Sys_ASRC_Config(0x2492792, ASRC_DEC_MODE3 | WIDE_BAND);`

9.21 SYS_ASRC_CONFIGRUNTIME

Configure the phase increment value according to the WDF selection and the input frequencies

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.c
Template	`void Sys_ASRC_ConfigRunTime(uint32_t cfg, uint32_t f_src, uint32_t f_sink)`
Description	Configure the phase increment value according to the WDF selection and the input frequencies. The mode is calculated automatically.
Inputs	cfg = The WDF type and the ASRC mode; use [LOW_DELAY | WIDE_BAND], [ASRC_INT_MODE | ASRC_DEC_MODE*] f_src = Source frequency or source sample number f_sink = Sink frequency or sink sample number diff_bit = Number of shifts on the numerator of the ASRC formula after the subtraction of (f_src) and (x*f_sink). It must be calculated by the user to prevent overflow and have the maximum precision
Outputs	None
Assumptions	The ASRC mode must be selected according to f_src and f_sink mode == ASRC_INT_MODE where (f_sink > f_src) mode == ASRC_DEC_MODE1 where (f_sink < f_src * 1.20 && f_sink > f_src * 0.8) mode == ASRC_DEC_MODE2 where (f_sink > f_src * 0.4 && f_sink < f_src)) mode == ASRC_DEC_MODE2 where f_sink < f_src * 0.4
Example	`/* Configure the ASRC block to convert 16 kHz sample rate to * 16.12 kHz at run time. Zero bit shift is performed*/ Sys_ASRC_ConfigRunTime(ASRC_INT_MODE | WIDE_BAND, 16000, 16120, 0);`

9.22 SYS_ASRC_INPUTDATA

Send a sample to the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_InputData(uint16_t data)`
Description	Send a sample to the ASRC block
Inputs	data = Value of the input sample
Outputs	None
Assumptions	None
Example	`/* Send 0xAA as sample in the ASRC block. */` `Sys_ASRC_InputData (0xAA);`

9.23 SYS_ASRC_INTENABLECONFIG

Configure the interrupt enable register of the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_IntEnableConfig(uint32_t cfg)`
Description	Configure the interrupt enable register of the ASRC block
Inputs	cfg = Interrupt register value; use INT_EBL_ASRC_IN, INT_EBL_ASRC_OUT, INT_EBL_ASRC_IN_ERR, and INT_EBL_ASRC_UPDATE_ERR
Outputs	None
Assumptions	None

Example	`/* Enable asrc_in and asrc_out interrupts. */`
	`Sys_ASRC_IntEnableConfig(INT_EBL_ASRC_IN |`
	`INT_EBL_ASRC_OUT);`

9.24 SYS_ASRC_OUTPUTCOUNT

Read the ASRC output counter value

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`uint32_t Sys_ASRC_OutputCount(void)`
Description	Read the ASRC output counter value
Inputs	None
Outputs	return value = The output counter value
Assumptions	None
Example	`/* Read the number of output samples. */` `result = Sys_ASRC_OutputCount();`

9.25 SYS_ASRC_OUTPUTDATA

Read a sample from the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`uint16_t Sys_ASRC_OutputData(void)`
Description	Read a sample from the ASRC block
Inputs	None
Outputs	return value = The output value of the block
Assumptions	None
Example	`/* Read sample out of the ASRC block. */` `result = Sys_ASRC_OutputData();`

9.26 SYS_ASRC_PHASEINCCONFIG

Calculate the phase increment value in m8p24

Type	Macro	
Include File	`#include <rsl10.h>`	
Source File	rsl10_sys_asrc.h	
Template	`Sys_ASRC_PhaseIncConfig(mode, f_src, f_sink)`	
Description	Calculate the phase increment value in m8p24	
Inputs	mode f_src f_sink	= Configuration value; use ASRC_INT_MODE | ASRC_DEC_MODE* = Source frequency or source samples = Sink frequency or sink samples
Outputs	return value	= 0 if frequency range check failed; otherwise return the phase increment value
Assumptions	None	

Example	`/* Calculate the phase increment value when f_src = 4 kHz and`
	` * f_sink = 16 kHz are in mode 0. */`
	`result = Sys_ASRC_PhaseIncConfig(ASRC_INT_MODE, 4, 16);`	

9.27 SYS_ASRC_RESET

Reset the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_Reset(void)`
Description	Reset the ASRC block
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset the ASRC block. */` `Sys_ASRC_Reset();`

9.28 SYS_ASRC_RESETOUTPUTCOUNT

Reset the ASRC block output counter

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_ResetOutputCount(void)`
Description	Reset the ASRC block output counter
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset the ASRC output counter. */` `Sys_ASRC_ResetOutputCount();`

9.29 SYS_ASRC_STATUS

Read status of the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`uint32_t Sys_ASRC_Status(void)`
Description	Read status of the ASRC block
Inputs	None
Outputs	return value = The value of the ASRC_CTRL register
Assumptions	None
Example	`/* Read status of the ASRC block. */` `result = Sys_ASRC_Status();`

9.30 SYS_ASRC_STATUSCONFIG

Configure the status of the ASRC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_asrc.h
Template	`void Sys_ASRC_StatusConfig(uint32_t cfg)`
Description	Configure the status of the ASRC block
Inputs	cfg = The value of the ASRC_CTRL register
Outputs	None
Assumptions	None
Example	`/* Reset the ASRC block. */` `Sys_ASRC_StatusConfig(ASRC_RESET);`

9.31 SYS_AUDIO_DMICDIOCONFIG

Configure two DIOs for the specified DMIC data input selection

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.h
Template	`void Sys_Audio_DMICDIOConfig(uint32_t cfg, uint32_t clk, uint32_t data, uint32_t clk_ext)`
Description	Configure two DIOs for the specified DMIC data input selection
Inputs	cfg = DIO pin configuration for the DMIC input clk = DIO to use as the DMIC clock out pad data = DIO to use as the DMIC input pad clk_ext = Clock source for external clock on DMIC clock pad; use DIO_MODE_[AUDIOCLK | AUDIOSLOWCLK]
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1 and 4 as the DMIC interface. */` `Sys_Audio_DMICDIOConfig(APP_DIO_CFG, 1, 4, DIO_MODE_AUDIOCLK);`

9.32 SYS_AUDIO_ODDIOCONFIG

Configure two DIOs for the specified OD data output selection

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.h
Template	`void Sys_Audio_ODDIOConfig(uint32_t cfg, uint32_t od_p, uint32_t od_n)`
Description	Configure two DIOs for the specified OD data output selection
Inputs	cfg = DIO pin configuration for the OD outputs od_p = DIO to use as the OD positive pin od_n = DIO to use as the OD negative pin
Outputs	None
Assumptions	None
Example	`/* Configure pin 0 as OD positive and 1 as OD negative. */` `Sys_Audio_ODDIOConfig(DIO_NO_PULL, 0, 1);`

9.33 SYS_AUDIO_ODDIOCONFIGMULT

Configure multiple sets of DIOs for the specified OD data output selection

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.c
Template	`void Sys_Audio_ODDIOConfigMult(uint32_t cfg, uint32_t * od_p, uint32_t * od_n, uint32_t num)`
Description	Configure multiple sets of DIOs for the specified OD data output selection
Inputs	cfg = DIO pin configuration for the OD outputs od_p = Pointer to the DIOs array to use as the OD positive pins od_n = Pointer to the DIOs array to use as the OD negative pins num = Number of pairs of DIO pins used for the OD
Outputs	None
Assumptions	The arrays od_p and od_n are the same length

Example	`/* Configure 3 pins as OD positive and 3 others as OD negative. */`
	`#define NUM_OD_DIO 3`
	
	`uint32_t P[NUM_OD_DIO] = {0, 1, 2};`
	`uint32_t N[NUM_OD_DIO] = {3, 4, 5};`
	`Sys_Audio_ODDIOConfigMult(DIO_NO_PULL, &P[0], &N[0], NUM_OD_DIO);`

9.34 SYS_AUDIO_SET_CONFIG

Configure the audio block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.h
Template	`void Sys_Audio_Set_Config(uint32_t cfg)`
Description	Configure the audio block
Inputs	cfg = The DMIC and OD configuration values; use OD_[AUDIOCLK | AUDIOSLOWCLK], DMIC_[AUDIOCLK | AUDIOSLOWCLK], DECIMATE_BY_*, OD_UNDERRUN_PROTECT_[ENABLE | DISABLE], OD_DMA_REQ_[ENABLE | DISABLE], OD_INT_GEN_[ENABLE | DISABLE], OD_DATA_[LSB | MSB]_ALIGNED, OD_[ENABLE | DISABLE], DMIC*_DMA_REQ_[ENABLE | DISABLE], DMIC*_INT_GEN_[ENABLE | DISABLE], DMIC*_DATA_[LSB | MSB]_ALIGNED, and DMIC*_[ENABLE | DISABLE]
Outputs	None
Assumptions	None
Example	`/* Enable OD with LSB aligned setting. */` `Sys_Audio_Set_Config(OD_ENABLE | OD_DATA_LSB_ALIGNED);`

9.35 SYS_AUDIO_SET_DMICCONFIG

Configure the DMIC

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.h
Template	`void Sys_Audio_Set_DMICConfig(uint32_t cfg, uint32_t frac_delay)`
Description	Configure the DMIC
Inputs	cfg = The DMIC configuration; use DMIC*_DCRM_CUTOFF_[*HZ | DISABLE], DMIC1_DELAY_[*P* | DISABLE], and DMIC*_[FALLING | RISING]_EDGE frac_delay = DMIC1 fractional delay; use a 5- bit number
Outputs	None
Assumptions	None

Example	`/* Enable DMIC0 for a 5 Hz cutoff removal frequency and decimation`
	` * rate of 64, sampled on the falling edge of the input clock */`
	`Sys_Audio_Set_DMICConfig(DMIC0_DCRM_CUTOFF_5HZ |`
	` DECIMATE_BY_64 |`
	` DMIC0_FALLING_EDGE |`
	` DMIC0_ENABLE,`
	` 0);`

9.36 SYS_AUDIO_SET_ODCONFIG

Configure the OD block and sigma-delta modulator for normal operation

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audio.h
Template	`void Sys_Audio_Set_ODConfig(uint32_t cfg)`
Description	Configure the OD block and sigma-delta modulator for normal operation
Inputs	cfg = The OD configuration; use DCRM_CUTOFF_[*HZ | DISABLE], DITHER_[ENABLE | DISABLE], and OD_[RISING | FALLING]_EDGE
Outputs	None
Assumptions	None
Example	`/* Configure OD to enable dithering, DC removal with a 10 Hz cut off * frequency and output data clock edge on rising. */ Sys_Audio_Set_ODConfig (DCRM_CUTOFF_10HZ | DITHER_ENABLE | OD_RISING_EDGE);`

9.37 SYS_AUDIOSINK_CONFIG

Configure the audio sink block and set values for clock counter, clock phase counter and clock period counter

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`void Sys_Audiosink_Config(uint32_t cfg, uint32_t phasecnt, uint32_t periodcnt)`
Description	Configure the audio sink block and set values for clock counter, clock phase counter and clock period counter
Inputs	cfg = The number of the audio sink Clock periods over which the period counter measures; use AUDIO_SINK_PERIODS_* phasecnt = The sink clock phase counter initial value periodcnt = The sink clock period counter initial value
Outputs	None
Assumptions	None
Example	`/* Measure 1 audio sink clock period. The initial value for the phase * and period counter is 0. */` `Sys_Audiosink_Config(AUDIO_SINK_PERIODS_1, 0, 0);`

9.38 SYS_AUDIOSINK_COUNTER

Read the value of the audio sink Clock counter

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`uint32_t Sys_Audiosink_Counter(void)`
Description	Read the value of the audio sink Clock counter
Inputs	None
Outputs	return value = The current value of the audio sink Clock counter
Assumptions	None
Example	`/* Read audio sink clock counter value. */` `result = Sys_Audiosink_Counter();`

9.39 SYS_AUDIOSINK_INPUTCLOCK

Configure a source for the audio sink input

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`void Sys_Audiosink_InputClock(uint32_t cfg, uint32_t sink)`
Description	Configure a source for the audio sink input
Inputs	cfg = DIO pin configuration for the audio sink input sink = Source to use as the audio sink input pad; use AUDIOSINK_CLK_SRC_DIO_*, AUDIOSINK_CLK_SRC_CONST_[LOW | HIGH], or AUDIOSINK_CLK_SRC_[STANDBYCLK | DMIC_OD]
Outputs	None
Assumptions	None
Example	`/* Configure DIO 0 as Audio Sink pad. */` `Sys_Audiosink_InputClock(APP_DIO_CFG, AUDIOSINK_CLK_SRC_DIO_0);`

9.40 SYS_AUDIOSINK_PERIODCOUNTER

Read the value of the audio sink Clock period counter

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`uint32_t Sys_Audiosink_PeriodCounter(void)`
Description	Read the value of the audio sink Clock period counter
Inputs	None
Outputs	return value = The current value of the audio sink Clock period counter
Assumptions	None
Example	`/* Read the audio sink period counter value. */` `result = Sys_Audiosink_PeriodCounter();`

9.41 SYS_AUDIOSINK_PHASECOUNTER

Read the value of the audio sink Clock phase counter

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`uint32_t Sys_Audiosink_PhaseCounter(void)`
Description	Read the value of the audio sink Clock phase counter
Inputs	None
Outputs	return value = The current value of the audio sink Clock phase counter
Assumptions	None
Example	`/* Read the audio sink phase counter value. */` `result = Sys_Audiosink_PhaseCounter();`

9.42 SYS_AUDIOSINK_RESETCOUNTERS

Reset counter, phase counter and period counter mechanism

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`void Sys_Audiosink_ResetCounters(void)`
Description	Reset counter, phase counter and period counter mechanism.
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset all Audio Sink counters. */` `Sys_Audiosink_ResetCounters();`

9.43 SYS_AUDIOSINK_SET_CTRL

Configure the audio sink Clock control

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_audiosink.h
Template	`void Sys_Audiosink_Set_Ctrl(uint32_t cfg)`
Description	Configure the audio sink Clock control
Inputs	cfg = The control value for the audio sink; use PHASE_CNT_[STOP | START], and CNT_RESET
Outputs	None
Assumptions	None

Example	`/* Reset PERIOD_CNT and start audio sink clock period counter,`
	` * stop audio sink clock phase counter. */`
	`Sys_Audiosink_Set_Ctrl(PHASE_CNT_STOP |`
	` CNT_RESET);`

9.44 SYS_BBIF_CONNECTRFFE

Internally connect the baseband to the RF front-end

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`Sys_BBIF_ConnectRFFE(void)`
Description	Internally connect the baseband to the RF front-end.
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Connect baseband to RF front-end. */` `Sys_BBIF_ConnectRFFE();`

9.45 SYS_BBIF_DIOCONFIG

Configure DIO pads connected to radio pins of baseband controller

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`void Sys_BBIF_DIOConfig(uint32_t cfg, uint32_t gpi_rx_clk, uint32_t gpi_rx_data, uint32_t tx_data_valid, uint32_t tx_data, uint32_t sync_p)`
Description	Configure DIO pads connected to radio pins of baseband controller
Inputs	cfg = DIO pin configuration for the output pads rx_clk = DIO to use as the BB_RX_CLK pad rx_data = DIO to use as the BB_RX_DATA pad tx_data_valid = DIO to use as the BB_TX_DATA_VALID pad tx_data = DIO to use as the BB_TX_DATA pad sync_p = DIO to use as the BB_SYNC_P pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1, 2, 3, 4, 5 as data for baseband. */` `Sys_BBIF_DIOConfig(APP_DIO_CFG, 1, 2, 3, 4, 5);`

9.46 SYS_BBIF_RFFE

Configure a DIO as a source for RF front-end audio synchronization pulse

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`void Sys_BBIF_RFFE(uint32_t gpio_num)`
Description	Configure a DIO as a source for RF front-end audio synchronization pulse
Inputs	gpio_num = GPIO number used for SYNC_PULSE generation
Outputs	None
Assumptions	None

Example	`/* Configure RF front-end audio synchronization pulse with link`
	` * label 0x00`
	` */`
	`Sys_BBIF_RFFE(0x00);`

9.47 SYS_BBIF_RFFEDRIVENEXTERNAL

Configure DIO pads connected to the RF frontend interface to be driven from an external device

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`Sys_BBIF_RFFEDrivenExternal(uint32_t cfg, uint32_t clk, uint32_t mosi, uint32_t miso, uint32_t csn, uint32_t rx_clk, uint32_t rx_data, uint32_t tx_data_val, uint32_t tx_data, uint32_t sync_p)`
Description	Configure DIO pads connected to the RF frontend interface to be driven from an external device
Inputs	cfg = DIO pin configuration for the output pads clk = DIO to use as the clock pad mosi = DIO to use as the MOSI pad miso = DIO to use as the MISO pad csn = DIO to use as the chip select pad rx_clk = DIO to use as the BB_RX_CLK pad rx_data = DIO to use as the BB_RX_DATA pad tx_data_valid = DIO to use as the BB_TX_DATA_VALID pad tx_data = DIO to use as the BB_TX_DATA pad sync_p = DIO to use as the BB_SYNC_P pad
Outputs	None
Assumptions	None

Example	`/* Connect RF and digital pins to external GPIOs. */`
	`Sys_BBIF_RFFEDrivenExternal(DIO_WEAK_PULL_DOWN,`
	`0, 1, 2, 3, 4, 5, 6, 7, 8);`

9.48 SYS_BBIF_SPICONFIG

Configure DIOs as an SPI slave for the Bluetooth baseband controller; disable the RF SPI slave interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`void Sys_BBIF_SPIConfig(uint32_t cfg, uint32_t miso, uint32_t csn, uint32_t mosi, uint32_t clk)`
Description	Configure DIOs as an SPI slave for the Bluetooth baseband controller; disable the RF SPI slave interface
Inputs	cfg = DIO pin configuration for the output pads miso = DIO to use as the MISO pad csn = DIO to use as the CSN pad mosi = DIO to use as the MOSI pad clk = DIO to use as the CLK pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1, 2, 3, 4 as the baseband SPI interface in slave * mode */` `Sys_BBIF_SPIConfig(APP_DIO_CFG, 1, 2, 3, 4);`

9.49 SYS_BBIF_SYNCCONFIG

Configure the link synchronization

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_bbif.h
Template	`void Sys_BBIF_SyncConfig(uint32_t cfg, uint32_t linklbl, uint32_t linkformat)`
Description	Configure the link synchronization
Inputs	cfg = Configuration of the link synchronization mechanism mode; use RX_[IDLE | ACTIVE], SYNC_[DISABLE | ENABLE], [IDLE | ACTIVE], and SYNC_SOURCE_[BLE_RX | BLE_RX_AUDIO* | RF_RX | BLE_TX] linklbl = The BLE link label for synchronization; use a 5-bit number linkformat = Configure the BLE link format for synchronization; use [SLAVE, MASTER]_CONNECT
Outputs	None
Assumptions	None
Example	`/* Configure the baseband synchronization signal to trigger from the * RX signal. */` `Sys_BBIF_SyncConfig(SYNC_ENABLE | SYNC_SOURCE_BLE_TX, 0, SLAVE_CONNECT);`

9.50 SYS_BOOTROM_RESET

Reset the system by executing the reset vector in the Boot ROM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`void Sys_BootROM_Reset(void)`
Description	Reset the system by executing the reset vector in the Boot ROM
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset the system by executing the reset vector in the Boot ROM. */` `Sys_BootROM_Reset();`

9.51 SYS_BOOTROM_STARTAPP

Validate and start up an application using the Boot ROM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`BootROMStatus Sys_BootROM_StartApp(uint32_t* vect_table)`
Description	Validate and start up an application using the Boot ROM.
Inputs	vect_table = Pointer to the vector table at the start of an application that will be validated and then run.
Outputs	return value = Status code indicating application validation error if application cannot be started. If not returning, the status code is written to the top of the started application's stack to capture non-fatal validation issues.
Assumptions	None

Example	`/* Checks if application is valid and starts it (if possible).`
	`* Returns status code. */`
	`isValid = Sys_BootROM_StartApp(vect_table);`

9.52 SYS_BOOTROM_STARTAPP_RETURN

Read the start application return code from the application stack for the current application

Type	Macro
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_cm3.h
Template	`SYS_BOOTROM_STARTAPP_RETURN`
Description	Read the start application return code from the application stack for the current application
Inputs	None
Outputs	return value = The value stored on the top of the stack (uint32_t); compare against BOOTROM_ERR_* or SYS_INIT_ERR_*
Assumptions	None
Example	`/* Check the boot ROM start application error code return. */` `result = SYS_BOOTROM_STARTAPP_RETURN;`

9.53 SYS_BOOTROM_STRICTSTARTAPP

Validate and start up an application using the Boot ROM

Type	Function	
Include File	`#include <rsl10.h>`	
Source File	rsl10_romvect.c	
Template	`BootROMStatus Sys_BootROM_StrictStartApp(uint32_t* vect_table)`	
Description	Validate and start up an application using the Boot ROM. Only start the application if application validation returns BOOTROM_ERR_NONE.	
Inputs	vect_table	= Pointer to the vector table at the start of an application that will be validated and then run.
Outputs	return value	= Status code indicating application validation error if application cannot be started. If not returning, the status code is written to the top of the started application's stack to capture non-fatal validation issues.
Assumptions	None	

Example	`/* Checks if application is valid and starts it (if possible).`
	` * Returns status code if any errors at all occur. */`
	`isValid = Sys_BootROM_StrictStartApp(vect_table);`	

9.54 SYS_BOOTROM_VALIDATEAPP

Validate an application using the Boot ROM application checks

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`BootROMStatus Sys_BootROM_ValidateApp(uint32_t* vect_table)`
Description	Validate an application using the Boot ROM application checks.
Inputs	vect_table = Pointer to the vector table at the start of an application that will be validated.
Outputs	return value = Status code indicating whether a validation error occurred or not; compare against BOOTROM_ERR_*
Assumptions	None
Example	`/* Checks if application is valid. Returns status code. */` `isValid = Sys_BootROM_ValidateApp(vect_table);`

9.55 SYS_CLOCKS_CLKDETENABLE

Enable/Disable the external clock detector

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_ClkDetEnable(void)`
Description	Enable/Disable the external clock detector
Inputs	cfg = Configuration of the clock detector enable value; use CLK_DET_[DISABLE | ENABLE]_BITBAND
Outputs	None
Assumptions	None
Example	`/* Enable the clock detector. */` `Sys_Clocks_ClkDetEnable(CLK_DET_ENABLE_BITBAND);`

9.56 SYS_CLOCKS_OSC

Configure the RC oscillator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_Osc(uint32_t cfg)`
Description	Configure the RC oscillator
Inputs	cfg = Configuration for 3 MHz/12 MHz RC oscillator; use RC_START_OSC_[3 | 12]MHZ RC_START_OSC_[M48 | M46P5 | NOM | P46P5]
Outputs	None
Assumptions	None

Example	`/* Enable the RC oscillator at a nominal 3 MHz frequency. */`
	`Sys_Clocks_Osc(RC_OSC_ENABLE |`
	`RC_START_OSC_3MHZ);`

9.57 SYS_CLOCKS_OSC32KCALIBRATEDCONFIG

Set the standby clock frequency to the given target based on a calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.c
Template	`unsigned int Sys_Clocks_Osc32kCalibratedConfig(uint16_t target)`
Description	Set the standby clock frequency to the given target based on a calibration trim value specified in NVR4. The 32k oscillator is not enabled. This function will only load the trim register, the user is responsible for enabling the oscillator if desired.
Inputs	target = The target 32k oscillator frequency in Hz
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the standby oscillator trim register to target 32768 Hz */` `result = Sys_Clocks_Osc32kCalibratedConfig(32768);`

9.58 SYS_CLOCKS_OSC32KHZ

Configure the 32 kHz RC oscillator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_Osc32kHz(uint32_t cfg)`
Description	Configure the 32 kHz RC oscillator
Inputs	cfg = Configuration for 32 kHz RC oscillator; use RC_OSC_[DISABLE | ENABLE] RC_OSC_RANGE_[NOM | M25] RC_OSC_[M48 | M46P5 | NOM | P46P5]
Outputs	None
Assumptions	None
Example	`/* The 32kHz RC Oscillator frequency trimming set to the nominal. */` `Sys_Clocks_Osc32kHz(RC_OSC_NOM);`

9.59 SYS_CLOCKS_OSCRCCALIBRATEDCONFIG

Set the start oscillator frequency to the given target based on a calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.c
Template	`unsigned int Sys_Clocks_OscRCCalibratedConfig(uint16_t target)`
Description	Set the start oscillator frequency to the given target based on a calibration trim value specified in NVR4. This function only loads the trim register and multiplier bit if necessary.
Inputs	target = The target start oscillator frequency in kHz
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the start oscillator trim register for a target of 3 MHz */` `result = Sys_Clocks_OscRCCalibratedConfig(3000);`

9.60 SYS_CLOCKS_SET_CLKDETCONFIG

Configure the external clock detector

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_Set_ClkDetConfig(uint32_t cfg)`
Description	Configure the external clock detector
Inputs	cfg = The external clock detector configuration; use CLK_DET_[DISABLE | ENABLE], CLK_DET_SLOWCLK_DIV*, CLK_DET_INT_[DISABLE | ACTIVATED | DEACTIVATED | ACTIVITY_CHANGE], and CLK_DET_SEL_[EXT | SW]CLK
Outputs	None
Assumptions	None

Example	`/* Enable the clock detector to monitor EXTCLK, setting the`
	` * clock detector divider to 32. */`
	`Sys_Clocks_Set_ClkDetConfig(CLK_DET_ENABLE |`
	` CLK_DET_SLOWCLK_DIV32 |`
	` CLK_DET_INT_ACTIVATED |`
	` CLK_DET_SEL_EXTCLK);`

9.61 SYS_CLOCKS_SYSTEMCLKCONFIG

Configure System Clock

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_SystemClkConfig(uint32_t cfg)`
Description	Configure System Clock
Inputs	cfg = Configuration of the system clock source and prescale value; use SYSCLK_CLKSRC_[RCCLK | STANDBYCLK | RFCLK | EXTCLK | JTCK], EXTCLK_PRESCALE_*, and JTCK_PRESCALE_*
Outputs	None
Assumptions	None

Example	`/* Configure the system clock source to RF clock with default`
	` * prescale values.`
	` */`
	`Sys_Clocks_SystemClkConfig(SYSCLK_CLKSRC_RFCLK |`
	` EXTCLK_PRESCALE_1 |`
	` JTCK_PRESCALE_1);`

9.62 SYS_CLOCKS_SYSTEMCLKPRESCALE0

Configure prescale register number 0

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_SystemClkPrescale0(uint32_t cfg)`
Description	Configure prescale register number 0
Inputs	cfg = Configuration of the prescale value for the slow, user and baseband peripheral clocks; use SLOWCLK_PRESCALE_*, BBCLK_PRESCALE_*, and USRCLK_PRESCALE_*
Outputs	None
Assumptions	None

Example	`/* Configure prescale of slow clock to 1, baseband clock to 2 and`
	` * user clock to 3. */`
	`Sys_Clocks_SystemClkPrescale0(SLOWCLK_PRESCALE_1 |`
	` BBCLK_PRESCALE_2 |`
	` USRCLK_PRESCALE_3);`

9.63 SYS_CLOCKS_SYSTEMCLKPRESCALE1

Configure prescale register number 1

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_SystemClkPrescale1(uint32_t cfg)`
Description	Configure prescale register number 1
Inputs	cfg = Configuration of the prescale value for the PWM0, PWM1, UART and AUDIO input peripheral clocks; use PWM0CLK_PRESCALE_*, PWM1CLK_PRESCALE_*, UARTCLK_PRESCALE_*, and AUDIOCLK_PRESCALE_*
Outputs	None
Assumptions	None

Example	`/* Configure prescale of PWM0 clock to 1, PWM1 clock to 2,`
	` * UART clock to 31 and AUDIO clock to 63. */`
	`Sys_Clocks_SystemClkPrescale1(PWM0CLK_PRESCALE_1 |`
	` PWM1CLK_PRESCALE_2 |`
	` UARTCLK_PRESCALE_31 |`
	` AUDIOCLK_PRESCALE_63);`

9.64 SYS_CLOCKS_SYSTEMCLKPRESCALE2

Configure prescale register number 2

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_clocks.h
Template	`void Sys_Clocks_SystemClkPrescale2(uint32_t cfg)`
Description	Configure prescale register number 2
Inputs	cfg = Configuration of the prescale value for the charge pump and DC-DC converter clocks; use CPCLK_PRESCALE_*, and DCCLK_PRESCALE_*
Outputs	None
Assumptions	None
Example	`/* Configure the prescalers of the clocks used by the charge pump * and DC-DC converters */ Sys_Clocks_SystemClkPrescale2(CPCLK_PRESCALE_1 | DCCLK_PRESCALE_2);`

9.65 SYS_CRC_CALC

Calculate the CRC over the specified range

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_crc.c
Template	`uint32_t Sys_CRC_Calc(uint32_t type, uint32_t base, uint32_t top, uint32_t* crc_val)`
Description	Calculate the CRC over the specified range
Inputs	type = CRC mode; use CRC_32 or CRC_CCITT base = Base of the range to be verified top = Last byte in the range to be verified crc_val = Address to store the calculated value at
Outputs	return value = 0 if invalid input; 1 otherwise
Assumptions	None

Example	`/* Calculate the CRC for a sample buffer */`
	`crc_val = 0x1234;`
	`result = Sys_CRC_Calc(CRC_CCITT, (uint32_t)&sample,`
	`(uint32_t)&sample[7], &crc_val);`
	
	`/* Save the calculated CRC at the end of the block (Assumes CRC-CCITT`
	`* algorithm and Little Endian mode) */`
	`sample[8] = (uint8_t) (crc_val & 0xFF);`
	`sample[9] = (uint8_t) ((crc_val >> 8) & 0xFF);`

9.66 SYS_CRC_CHECK

Check the CRC over the specified range assuming the last bytes of the defined block contain the previously calculated CRC

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_crc.c
Template	`uint32_t Sys_CRC_Check(uint32_t type, uint32_t base, uint32_t top)`
Description	Check the CRC over the specified range assuming the last bytes of the defined block contain the previously calculated CRC
Inputs	type = CRC mode; use CRC_32 or CRC_CCITT base = Base of the range to be verified top = Last byte in the range to be verified
Outputs	return value = 0 if CRC check failed, 1 if CRC check passed, 2 if there's an error
Assumptions	None

Example	`/* Verify the CRC for a sample array of ten 8-bit elements */`
	`result = Sys_CRC_Check(CRC_CCITT, (uint32_t)&sample,`
	`(uint32_t)&sample[9]);`

9.67 SYS_CRC_GET_CONFIG

Get the CRC generator configuration

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_crc.h
Template	`uint32_t Sys_CRC_Get_Config(void)`
Description	Get the CRC generator configuration
Inputs	None
Outputs	return value = CRC generator configuration; compare with CRC_[CCITT | 32], CRC_[BIG | LITTLE]_ENDIAN, CRC_BIT_ORDER_[STANDARD | NON_STANDARD], CRC_FINAL_REVERSE_[STANDARD | NON_STANDARD], and CRC_FINAL_XOR_[STANDARD | NON_STANDARD]
Assumptions	None
Example	`/* Get current CRC generator configuration. */` `curr_config = Sys_CRC_Get_Config();`

9.68 SYS_CRC_SET_CONFIG

Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_crc.h
Template	`void Sys_CRC_Set_Config(uint32_t cfg)`
Description	Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior
Inputs	cfg = CRC generator configuration; use CRC_[CCITT | 32], CRC_[BIG | LITTLE]_ENDIAN, CRC_BIT_ORDER_[STANDARD | NON_STANDARD], CRC_FINAL_REVERSE_[STANDARD | NON_STANDARD], and CRC_FINAL_XOR_[STANDARD | NON_STANDARD]
Outputs	None
Assumptions	None
Example	`/* Enable CRC-CCITT (16-bit) algorithm, Little Endian mode, standard * bit order, standard CRC reversal and standard CRC XOR. */ Sys_CRC_Set_Config(CRC_CCITT | CRC_LITTLE_ENDIAN | CRC_BIT_ORDER_STANDARD | CRC_FINAL_REVERSE_STANDARD | CRC_FINAL_XOR_STANDARD);`

9.69 SYS_DELAY_PROGRAMROM

Delay by at least the specified number of clock cycles

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`void Sys_Delay_ProgramROM(uint32_t cycles)`
Description	Delay by at least the specified number of clock cycles
Inputs	cycles = Number of system clock cycles to delay
Outputs	None
Assumptions	The requested delay is at least 32 cycles (32 us at 1 MHz) and fits in a uint32_t (0xFFFFFFFF cycles is approximately 214.75 s at 20 MHz). A delay between cycles and (cycles + 3) provides a sufficient delay resolution. The requested delay does not exceed the watchdog timeout. If the delay resolution is required to be exact, disable interrupts.
Example	`/* Delay by 100 clock cycles. */` `Sys_Delay_ProgramROM(100);`

9.70 SYS_DIO_CONFIG

Configure the specified digital I/O

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dio.h
Template	`void Sys_DIO_Config(uint32_t pad, uint32_t cfg)`
Description	Configure the specified digital I/O
Inputs	pad = Digital I/O pad to configure; use a constant between 0 and 15 cfg = I/O configuration; use DIO_*X_DRIVE, DIO_LPF_[ENABLE | DISABLE], DIO_[NO_PULL | STRONG_PULL_UP | WEAK_PULL_UP | WEAK_PULL_DOWN], and DIO_MODE_*
Outputs	None
Assumptions	None
Example	`/* Configure DIO 4 as a GPIO input with a weak pull-up resistor. */` `Sys_DIO_Config(4, DIO_WEAK_PULL_UP | DIO_MODE_GPIO_IN_0);`

9.71 SYS_DIO_GET_MODE

Get the DIO mode (DIO or GPIO)

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dio.h
Template	`uint32_t Sys_DIO_Get_Mode(void)`
Description	Get the DIO mode (DIO or GPIO)
Inputs	None
Outputs	return value = Current DIO/GPIO mode
Assumptions	None
Example	`/* Get current DIO mode for DIOs 0 to 15. */` `dio_mode = Sys_DIO_Get_Mode();`

9.72 SYS_DIO_INTCONFIG

Configure a DIO interrupt source

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dio.h
Template	`void Sys_DIO_IntConfig(uint32_t index, uint32_t cfg, uint32_t dbnc_clk, uint32_t dbnc_cnt)`
Description	Configure a DIO interrupt source
Inputs	index = DIO interrupt source to configure; use [0- 3] cfg = DIO interrupt configuration; use DIO_DEBOUNCE_[DISABLE | ENABLE], DIO_SRC_DIO_*, and DIO_EVENT_[NONE | HIGH_LEVEL | LOW_LEVEL | RISING_EDGE | FALLING_EDGE | TRANSITION] dbnc_clk = Interrupt button debounce filter clock; use DIO_DEBOUNCE_SLOWCLK_DIV[32 | 1024] dbnc_cnt = Interrupt button debounce filter count
Outputs	None
Assumptions	None

Example	`/* Configure 0 Interrupt source with pad 0 and high level event in * active debounce with slow clock divider 32. */`
	`Sys_DIO_IntConfig(0,`
	`DIO_DEBOUNCE_ENABLE |`
	`DIO_SRC_DIO_0 |`
	`DIO_EVENT_HIGH_LEVEL,`
	`DIO_DEBOUNCE_SLOWCLK_DIV32,`
	`0);`

9.73 SYS_DIO_NMICONFIG

Configure a DIO for the specified NMI input selection

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dio.h
Template	`void Sys_DIO_NMIConfig(uint32_t cfg, uint32_t nmi)`
Description	Configure a DIO for the specified NMI input selection
Inputs	cfg = DIO pin configuration for the NMI input pad nmi = DIO to use as the NMI input pad
Outputs	None
Assumptions	None
Example	`/* Configure DIO 1 as the NMI interface. */` `Sys_DIO_NMIConfig(APP_DIO_CFG, 1);`

9.74 SYS_DIO_SET_DIRECTION

Set the input/output direction for any DIOs configured as GPIOs

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dio.h
Template	`void Sys_DIO_Set_Direction(uint32_t dir)`
Description	Set the input/output direction for any DIOs configured as GPIOs
Inputs	dir = Input/output configuration for those DIOs configured as GPIOs; use DIO*_INPUT, and DIO*_OUTPUT
Outputs	None
Assumptions	None
Example	`/* Set DIO0, DIO2 as inputs; set DIO1 as an output. */` `Sys_DIO_Set_Direction(DIO0_INPUT | DIO1_OUTPUT | DIO2_INPUT);`

9.75 SYS_DMA_CHANNELCONFIG

Configure the DMA channels for a data transfer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.c
Template	`void Sys_DMA_ChannelConfig(uint32_t num, uint32_t cfg, uint32_t transferLength, uint32_t counterInt, uint32_t srcAddr, uint32_t destAddr)`
Description	Configure the DMA channels for a data transfer
Inputs	num = DMA channel number cfg = Configuration of the DMA transfer behavior; use DMA_DEST_ADDR_STEP_SIZE_*, DMA_SRC_ADDR_STEP_SIZE_*, DMA_DEST_ADDR_[POS | NEG], DMA_SRC_ADDR_[POS | NEG], DMA_[LITTLE | BIG]_ENDIAN, DMA_DISABLE_INT_[DISABLE | ENABLE], DMA_ERROR_INT_[DISABLE | ENABLE], DMA_COMPLETE_INT_[DISABLE | ENABLE], DMA_COUNTER_INT_[DISABLE | ENABLE], DMA_START_INT_[DISABLE | ENABLE], DMA_DEST_WORD_SIZE_*, DMA_SRC_WORD_SIZE_*, DMA_DEST_[I^2C | SPI0 | SPI1 | PCM | UART | ASRC], DMA_SRC_[I^2C | SPI0 | SPI1 | PCM | UART | ASRC], DMA_PRIORITY_*, DMA_TRANSFER_[P | M]_TO_[P | M] DMA_DEST_ADDR_[STATIC | INC], DMA_SRC_ADDR_[STATIC | INC], DMA_ADDR_[CIRC | LIN], DMA_[DISABLE | ENABLE] transferLength = Configuration of the DMA transfer length counterInt = Configuration of when the counter interrupt will occur during the transfer srcAddr = Base source address for the DMA transfer destAddr = Base destination address for the DMA transfer
Outputs	None

Assumptions	None
Example	/* Configure DMA channel 0 for a transfer from the PCM interface to * a 16-word buffer in memory. Clear any previous DMA status bit * settings. */ Sys_DMA_ChannelConfig(0, (DMA_DEST_ADDR_STEP_SIZE_1 | DMA_DEST_ADDR_POS | DMA_SRC_ADDR_STEP_SIZE_1 | DMA_SRC_ADDR_POS | DMA_LITTLE_ENDIAN | DMA_DISABLE_INT_DISABLE | DMA_ERROR_INT_ENABLE | DMA_COMPLETE_INT_ENABLE | DMA_COUNTER_INT_ENABLE | DMA_START_INT_DISABLE | DMA_DEST_WORD_SIZE_32 | DMA_SRC_WORD_SIZE_32 | DMA_SRC_PCM | DMA_PRIORITY_0 | DMA_TRANSFER_P_TO_M | DMA_SRC_ADDR_STATIC | DMA_DEST_ADDR_INC | DMA_ADDR_CIRC | DMA_DISABLE) , 16, 0, (uint32_t)&PCM->RX_DATA, (uint32_t)buffer);

9.76 SYS_DMA_CHANNELDISABLE

Disable the DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_ChannelDisable(uint32_t num)`
Description	Disable the DMA channel
Inputs	num = The DMA channel number
Outputs	None
Assumptions	None
Example	`/* Disable DMA channel 0. */` `Sys_DMA_ChannelDisable(0);`

9.77 SYS_DMA_CHANNELENABLE

Enable the DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_ChannelEnable(uint32_t num)`
Description	Enable the DMA channel
Inputs	num = The DMA channel number
Outputs	None
Assumptions	None
Example	`/* Enable DMA channel 0. */` `Sys_DMA_ChannelEnable(0);`

9.78 SYS_DMA_CLEARALLCHANNELSTATUS

Clear the current status for the DMA on all channels

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_ClearAllChannelStatus(void)`
Description	Clear the current status for the DMA on all channels
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Clear the current status for DMA on all channels. */` `Sys_DMA_ClearAllChannelStatus();`

9.79 SYS_DMA_CLEARCHANNELSTATUS

Clear the current status for the specified DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_ClearChannelStatus(uint32_t num)`
Description	Clear the current status for the specified DMA channel
Inputs	num = The DMA channel number; use 0- 7
Outputs	None
Assumptions	None
Example	`/* Clear the current status for DMA channel 0. */` `Sys_DMA_ClearChannelStatus(0);`

9.80 SYS_DMA_GET_CHANNELSTATUS

Get the current status of the DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`uint8_t Sys_DMA_Get_ChannelStatus(uint32_t num)`
Description	Get the current status of the DMA channel
Inputs	num = The DMA channel number
Outputs	return value = The current status of the specified DMA channel
Assumptions	None
Example	`/* Get the current status of DMA channel 0. */` `status = Sys_DMA_Get_ChannelStatus(0);`

9.81 SYS_DMA_SET_CHANNELDESTADDRESS

Set the base destination address for the specified DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_Set_ChannelDestAddress(uint32_t num, uint32_t destAddr)`
Description	Set the base destination address for the specified DMA channel
Inputs	num = The DMA channel number destAddr = Base destination address for the DMA transfer
Outputs	None
Assumptions	None

Example	`/* Set buffer as the base destination address for DMA channel 0. */`
	`uint32_t buffer[1] = {0};`
	`Sys_DMA_Set_ChannelDestAddress(0, (uint32_t) buffer);`

9.82 SYS_DMA_SET_CHANNELSOURCEADDRESS

Set the base source address for the specified DMA channel

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_dma.h
Template	`void Sys_DMA_Set_ChannelSourceAddress(uint32_t num, uint32_t srcAddr)`
Description	Set the base source address for the specified DMA channel
Inputs	num = The DMA channel number srcAddr = Base source address for the DMA transfer
Outputs	None
Assumptions	None
Example	`/* Set PCM_RX_DATA as the base source address for DMA channel 0. */` `Sys_DMA_Set_ChannelSourceAddress(0, (uint32_t)&PCM->RX_DATA);`

9.83 SYS_FLASH_COMPARE

Compare data in the flash to a pre-specified value

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_flash.c
Template	`uint32_t Sys_Flash_Compare(uint32_t cfg, uint32_t addr, uint32_t length, uint32_t value, uint32_t value_ecc)`
Description	Compare data in the flash to a pre-specified value
Inputs	cfg = Flash comparator configuration; use COMP_MODE_[CONSTANT | CHBK]_BYTE, COMP_ADDR_[DOWN | UP]_BYTE, and COMP_ADDR_STEP_*_BYTE addr = Base address of the area to verify length = Number of words to verify value = Value that the words read from flash will be compared against value_ecc = Value that the error-correction coding bits from the extended words read from flash will be compared against
Outputs	return value = 0 if comparison succeeded, 1 if the comparison failed.
Assumptions	addr points to an address in flash memory

Example	`/* Check that the 10 words this application needs are still erased */`
	`result = Sys_Flash_Compare((COMP_MODE_CONSTANT_BYTE |`
	`COMP_ADDR_UP_BYTE |`
	`COMP_ADDR_STEP_1_BYTE),`
	`FLASH_MAIN_TOP - 0x100, 10,`
	`0xFFFFFFFF, 0xF);`
	
	`if (result == 0)`
	`{`
	`/* Use the flash here since it has been validated as erased. */`
	`}`

9.84 SYS_FLASH_COPY

Copy data from the flash memory to a RAM memory instance

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_flash.c
Template	`void Sys_Flash_Copy(uint32_t src_addr, uint32_t dest_addr, uint32_t length, uint32_t cpy_dest)`
Description	Copy data from the flash memory to a RAM memory instance
Inputs	src_addr = Source address in flash to copy data from dest_addr = Destination address in RAM to copy data to length = Number of words to copy cpy_dest = Destination copier is CRC or memories; use COPY_TO_[CRC | MEM]_BITBAND
Outputs	None
Assumptions	src_addr points to an address in flash memory dest_addr points to an address in RAM memory If dest_addr points to an area in DSP_PRAM memory, the copy will write all 40 bits of the PRAM memory The flash copy does not need to be complete before returning If CRC is selected as the destination, dest_addr is ignored and 32-bit copy mode is selected automatically.
Example	`/* Copy 10 words from data to the base of DSP_PRAM0. */` `Sys_Flash_Copy(data, DSP_PRAM0_BASE, 10, COPY_TO_MEM_BITBAND);`

9.85 SYS_FLASH_ECC_CONFIG

Configure the flash error-correction control support

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_flash.h
Template	`void Sys_Flash_ECC_Config(uint32_t cfg)`
Description	Configure the flash error-correction control support
Inputs	cfg = Configuration for the flash error-correction control block; use FLASH_IDBUS_ECC_[ENABLE | DISABLE], FLASH_DMA_ECC_[ENABLE | DISABLE], FLASH_CMD_ECC_[ENABLE | DISABLE], FLASH_COPIER_ECC_[ENABLE | DISABLE], and FLASH_ECC_COR_INT_THRESHOLD_* or a constant shifted to FLASH_ECC_CTRL_ECC_COR_CNT_INT_THRESHOLD_Pos
Outputs	None
Assumptions	None

Example	`/* Enable the ECC blocks for all users of flash. */`
	`Sys_Flash_ECC_Config(FLASH_IDBUS_ECC_ENABLE | FLASH_CMD_ECC_ENABLE |`
	`FLASH_COPIER_ECC_ENABLE |`
	`FLASH_ECC_COR_INT_THRESHOLD_1);`

9.86 SYS_GPIO_SET_HIGH

Set the specified GPIO output value to high

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_gpio.h
Template	`void Sys_GPIO_Set_High(uint32_t gpio_pin)`
Description	Set the specified GPIO output value to high
Inputs	gpio_pin = GPIO pin to set high
Outputs	None
Assumptions	None
Example	`/* Set GPIO 0 high. */` `Sys_GPIO_Set_High(0);`

9.87 SYS_GPIO_SET_LOW

Set the specified GPIO output value to low

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_gpio.h
Template	`void Sys_GPIO_Set_Low(uint32_t gpio_pin)`
Description	Set the specified GPIO output value to low
Inputs	gpio_pin = GPIO pin to set low
Outputs	None
Assumptions	None
Example	`/* Set GPIO 0 low. */` `Sys_GPIO_Set_Low(0);`

9.88 SYS_GPIO_TOGGLE

Toggle the current value of the specified GPIO output

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_gpio.h
Template	`void Sys_GPIO_Toggle(uint32_t gpio_pin)`
Description	Toggle the current value of the specified GPIO output
Inputs	gpio_pin = GPIO pin to toggle
Outputs	None
Assumptions	None
Example	`/* Toggle GPIO 0. */` `Sys_GPIO_Toggle(0);`

9.89 SYS_I2C_ACK

Manually acknowledge the latest transfer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_ACK(void)`
Description	Manually acknowledge the latest transfer
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Acknowledge the latest byte transfer. */` `Sys_I2C_ACK();`

9.90 SYS_I2C_CONFIG

Configure the I²C interface for operation

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_Config(uint32_t cfg)`
Description	Configure the I²C interface for operation
Inputs	cfg = I²C interface configuration; use I2C_MASTER_SPEED_*, a slave address constant shifted to I2C_CTRL0_SLAVE_ADDRESS_Pos, I2C_CONTROLLER_[CM3 | DMA], I2C_STOP_INT_[ENABLE | DISABLE], I2C_AUTO_ACK_[ENABLE | DISABLE], I2C_SAMPLE_CLK_[ENABLE | DISABLE], and I2C_SLAVE_[ENABLE | DISABLE],
Outputs	None
Assumptions	None

Example	`/* Configure the I2C interface to communicate as a slave at address * 0x40 in auto-acknowledgement mode. */`
	`Sys_I2C_Config( (0x40 << I2C_CTRL0_SLAVE_ADDRESS_Pos) |`
	`I2C_CONTROLLER_CM3 |`
	`I2C_STOP_INT_DISABLE |`
	`I2C_AUTO_ACK_ENABLE |`
	`I2C_SAMPLE_CLK_ENABLE |`
	`I2C_SLAVE_ENABLE );`

9.91 SYS_I2C_DIOCONFIG

Configure two DIOs for the specified I²C interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_DIOConfig(uint32_t cfg, uint32_t scl, uint32_t sda)`
Description	Configure two DIOs for the specified I²C interface
Inputs	cfg = DIO pin configuration for the I²C pads scl = DIO to use as the I²C SCL pad sda = DIO to use as the I²C SDA pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1 and 4 as the I2C interface. */` `Sys_I2C_DIOConfig(APP_DIO_CFG, 1, 4);`

9.92 SYS_I2C_GET_STATUS

Get the current I²C interface status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`uint32_t Sys_I2C_Get_Status(void)`
Description	Get the current I²C interface status
Inputs	None
Outputs	status = Current I²C interface status; compare with I2C_[NO_ERROR | ERROR]_S, I2C_[BUS | NO_BUS]_ERROR_S, I2C_START_[PENDING | NOT_PENDING], I2C_MASTER_[ACTIVE | INACTIVE], I2C_[DMA | NO_DMA]_REQUEST, I2C_[STOP | NO_STOP]_DETECTED, I2C_[DATA | NON_DATA]_EVENT, I2C_[ERROR | NO_ERROR], I2C_[BUS_ERROR | NO_BUS_ERROR], I2C_BUFFER_[FULL | EMPTY], I2C_CLK_[STRETCHED | NOT_STRETCHED], I2C_BUS_[FREE | BUSY], I2C_DATA_IS_[ADDR | DATA], I2C_IS_[READ | WRITE], I2C_ADDR_[GEN_CALL | OTHER] and I2C_HAS_[NACK | ACK]
Assumptions	None
Example	`/* Check if a repeated start condition occurred, indicating that the * most recently received data will be treated as an address. */ if ((Sys_I2C_Get_Status() & (1 << I2C_STATUS_ADDR_DATA_Pos)) == I2C_DATA_IS_ADDR) { /* Initialize a new slave transfer over the I2C interface. */ }`

9.93 SYS_I2C_LASTDATA

Indicate that this is the last byte in the transfer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_LastData(void)`
Description	Indicate that this is the last byte in the transfer
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Send LAST_DATA control bit to stop a transaction automatically. */` `Sys_I2C_LastData();`

9.94 SYS_I2C_NACK

Manually not-acknowledge the latest transfer (releases the bus to continue with a transfer)

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_NACK(void)`
Description	Manually not-acknowledge the latest transfer (releases the bus to continue with a transfer)
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Do not acknowledge the latest byte transfer. */` `Sys_I2C_NACK();`

9.95 SYS_I2C_NACKANDSTOP

Manually not-acknowledge the latest transfer and send a stop condition (Master mode only)

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_NACKAndStop(void)`
Description	Manually not-acknowledge the latest transfer and send a stop condition (Master mode only)
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Not-acknowledge the latest byte transfer and issue a stop`
	` * condition on the I2C interface. */`
	`Sys_I2C_NACKAndStop();`

9.96 SYS_I2C_RESET

Reset the I²C interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_Reset(void)`
Description	Reset the I²C interface
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset the I2C Interface. */` `Sys_I2C_Reset();`

9.97 SYS_I2C_STARTREAD

Initialize an I²C master read transfer on the I²C interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_StartRead(uint32_t addr)`
Description	Initialize an I²C master read transfer on the I²C interface
Inputs	addr = I²C slave address to initiate a transfer with
Outputs	None
Assumptions	None
Example	`/* Initialize a read from address 0x40 over the I2C interface. */` `Sys_I2C_StartRead(0x40);`

9.98 SYS_I2C_STARTWRITE

Initialize an I²C master write transfer on the I²C interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_i2c.h
Template	`void Sys_I2C_StartWrite(uint32_t addr)`
Description	Initialize an I²C master write transfer on the I²C interface
Inputs	addr = I²C slave address to initiate a transfer with
Outputs	None
Assumptions	None
Example	`/* Initialize a write to the general call address over the I2C * interface. */ Sys_I2C_StartWrite(0x00);`

9.99 SYS_INITIALIZE

Run the program ROM's extended initialization functions

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`SysInitStatus Sys_Initialize(void)`
Description	Run the program ROM's extended initialization functions.
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Reinitialize the system using the initialization program stored`
	` * to the information page of flash. */`
	`Sys_Initialize();`

9.100 SYS_INITIALIZE_BASE

Run the Program ROM based basic initialization function; re-initialize all critical memory, clock, and power supply components

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`void Sys_Initialize_Base(void)`
Description	Run the Program ROM based basic initialization function; re-initialize all critical memory, clock, and power supply components
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reinitialize the system using the ROM initializtion routine. */` `Sys_Initialize_Base();`

9.101 SYS_IP_LOCK

Configure the debug lock key and set the device SWJ-DP to lock mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_ip.h
Template	`void Sys_IP_Lock(uint32_t * key)`
Description	Configure the debug lock key and set the device SWJ-DP to lock mode
Inputs	key = Pointer to the 128-bit key as a debug lock key
Outputs	None
Assumptions	None

Example	`uint32_t ip_key[4] = {0x12345678, 0x9ABCDEF0, 0xAA5500FF, 0xAABBCCDD};`
	
	`/* Set the key and limit access to the SWJ-DP interface */`
	`Sys_IP_Lock(ip_key);`

9.102 SYS_IP_UNLOCK

Set the device SWJ-DP to unlock mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_ip.h
Template	`void Sys_IP_Unlock(void)`
Description	Set the device SWJ-DP to unlock mode
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Unlock full access to the SWJ-DP interface */` `Sys_IP_Unlock();`

9.103 SYS_LPDSP32_COMMAND

Configure commands for LPDSP32

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_Command(uint32_t cfg)`
Description	Configure commands for LPDSP32
Inputs	cfg = Set LPDSP32 commands; use DSS_CMD_[0- 6]
Outputs	None
Assumptions	None
Example	`/* Send command 0 to the LPDSP32. */` `Sys_LPDSP32_Command(DSS_CMD_0);`

9.104 SYS_LPDSP32_DIOJTAG

Configure DIO pads connected to LPDSP32 JTAG

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`Sys_LPDSP32_DIOJTAG(uint32_t cfg, uint32_t tdi, uint32_t tms, uint32_t tck, uint32_t tdo)`
Description	Configure DIO pads connected to LPDSP32 JTAG. It causes the LPDSP32 to be resumed.
Inputs	cfg = DIO pin configuration for LPDSP32 JTAG pads tdi = DIO to use as the JTAG TDI pad tms = DIO to use as the JTAG TMS pad tck = DIO to use as the JTAG TCK pad tdo = DIO to use as the JTAG TDO pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1, 2 and 3 as the LPDSP32 JTAG interface. */` `Sys_LPDSP32_DIOJTAG(APP_DIO_CFG, 1, 2, 3, 4);`

9.105 SYS_LPDSP32_GET_ACTIVITYCOUNTER

Read LPDSP32 activity counter value

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`uint32_t Sys_LPDSP32_Get_ActivityCounter(void)`
Description	Read LPDSP32 activity counter value
Inputs	None
Outputs	return value = LPDSP32 activity counter value
Assumptions	None
Example	`/* Read the LPDSP32 activity counter value. */` `result = Sys_LPDSP32_Get_ActivityCounter();`

9.106 SYS_LPDSP32_INTCLEAR

Reset pending (DMA and ARM Cortex-M3) interrupts in the LPDSP32 interrupt controller

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_IntClear(void)`
Description	Reset pending (DMA and ARM Cortex-M3) interrupts in the LPDSP32 interrupt controller
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset pending interrupts in the LPDSP32 interrupt controller. */` `Sys_LPDSP32_IntClear();`

9.107 SYS_LPDSP32_PAUSE

Pause LPDSP32

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_Pause(void)`
Description	Pause LPDSP32
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Pause DSS. */` `Sys_LPDSP32_Pause();`

9.108 SYS_LPDSP32_RESET

Reset LPDSP32

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_Reset(void)`
Description	Reset LPDSP32
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset DSS. */` `Sys_LPDSP32_Reset();`

9.109 SYS_LPDSP32_RUN

Run LPDSP32

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_Run(void)`
Description	Run LPDSP32
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Run DSS. */` `Sys_LPDSP32_Run();`

9.110 SYS_LPDSP32_RUN_STATUS

LPDSP32 running status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`uint32_t Sys_LPDSP32_Run_Status(void)`
Description	LPDSP32 running status
Inputs	None
Outputs	return value = LPDSP32 status; compare with DSS_LPDSP32_STATE_[PAUSE | RUN]
Assumptions	None
Example	`/* Get the LPDSP32 running status. */` `status = Sys_LPDSP32_Run_Status();`

9.111 SYS_LPDSP32_RUNTIMEADDR

Calculate the equivalent LPDSP32 address to an ARM Cortex-M3 processor address

Type	Macro
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`Sys_LPDSP32_RuntimeAddr(Addr, prgdata)`
Description	Calculate the equivalent LPDSP32 address to an ARM Cortex-M3 processor address.
Inputs	Addr = the address in LPDSP32 prgdata = selection between program memory or data memory in LPDSP32. Value 1 indicates program section and 0 shows data section
Outputs	return value = equivalent address in ARM Cortex-M3 processor
Assumptions	For DSP_PROMx the output will contain the 32 bit LSB locations

Example	`/* Calculate the equivalent LPDSP32 address 0x4000 in the`
	` * program memory. */`
	`result = Sys_LPDSP32_RuntimeAddr(0x4000, 1);`

9.112 SYS_LPDSP32_SET_DEBUGCONFIG

Configure the LPDSP32 Debug Port

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_lpdsp32.h
Template	`void Sys_LPDSP32_Set_DebugConfig(uint32_t cfg)`
Description	Configure the LPDSP32 Debug Port
Inputs	cfg = debug port configuration; use LPDSP32_DEBUG_IN_POWERDOWN_[DISABLED | ENABLE] | LPDSP32_EXIT_POWERDOWN_WHEN_HALTED_[DISABLED | ENABLE]
Outputs	None
Assumptions	None

Example	`/* Configure the LPDSP32 to exit power-down when halted */`
	`Sys_LPDSP32_Set_DebugConfig(`
	`  LPDSP32_EXIT_POWERDOWN_WHEN_HALTED_DISABLED);`

9.113 SYS_NVIC_CLEARALLPENDINGINT

Clear the pending status for all external interrupts

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_cm3.h
Template	`void Sys_NVIC_ClearAllPendingInt(void)`
Description	Clear the pending status for all external interrupts
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Clear the pending status for all of the external interrupts. */` `Sys_NVIC_ClearAllPendingInt();`

9.114 SYS_NVIC_DISABLEALLINT

Disable all external interrupts

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_cm3.h
Template	`void Sys_NVIC_DisableAllInt(void)`
Description	Disable all external interrupts
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Disable all external interrupts. */` `Sys_NVIC_DisableAllInt();`

9.115 SYS_PCM_CLEARSTATUS

Clear the current PCM interface status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_ClearStatus(void)`
Description	Clear the current PCM interface status
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Clear the error indicators for the PCM Interface. */` `Sys_PCM_ClearStatus();`

9.116 SYS_PCM_CONFIG

Configure the PCM interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_Config(uint32_t cfg)`
Description	Configure the PCM interface
Inputs	cfg = Interface operation configuration; use PCM_SAMPLE_[FALLING | RISING]_EDGE, PCM_BIT_ORDER_[MSB | LSB]_FIRST, PCM_TX_ALIGN_[MSB | LSB], PCM_WORD_SIZE_*, PCM_FRAME_ALIGN_[LAST | FIRST], PCM_FRAME_WIDTH_[SHORT | LONG], PCM_MULTIWORD_*, PCM_SUBFRAME_[ENABLE | DISABLE], PCM_CONTROLLER_[CM3 | DMA], PCM_[DISABLE | ENABLE], and PCM_SELECT_[MASTER | SLAVE]
Outputs	None
Assumptions	None

Example	`/* Configure the PCM interface as a master for use with the DMA. */`
	`Sys_PCM_Config(PCM_SAMPLE_FALLING_EDGE |`
	`PCM_BIT_ORDER_MSB_FIRST |`
	`PCM_TX_ALIGN_MSB |`
	`PCM_WORD_SIZE_32 |`
	`PCM_FRAME_ALIGN_LAST |`
	`PCM_FRAME_WIDTH_LONG |`
	`PCM_MULTIWORD_2 |`
	`PCM_SUBFRAME_ENABLE |`
	`PCM_CONTROLLER_DMA |`
	`PCM_ENABLE |`
	`PCM_SELECT_MASTER);`

9.117 SYS_PCM_CONFIGCLK

Configure four DIOs for the PCM interface and clock source selection

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_ConfigClk(uint32_t slave, uint32_t cfg, uint32_t clk, uint32_t frame, uint32_t seri, uint32_t sero, uint32_t clksrc)`
Description	Configure four DIOs for the PCM interface and clock source selection
Inputs	slave = PCM master/slave configuration; use PCM_SELECT_[MASTER | SLAVE] cfg = DIO pin configuration for the PCM pads clk = DIO to use as the PCM clock pad frame = DIO to use as the PCM frame pad seri = DIO to use as the PCM serial input pad sero = DIO to use as the PCM serial output pad clksrc = Clock source for PCM; use DIO_MODE_[*CLK | INPUT]
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 0, 1, 2, and 3 as a master PCM interface. */` `Sys_PCM_ConfigClk(PCM_SELECT_MASTER, APP_DIO_CFG, 0, 1, 2, 3, DIO_MODE_USRCLK);`

9.118 SYS_PCM_DIOCONFIG

Configure four DIOs for the PCM interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_DIOConfig(uint32_t slave, uint32_t cfg, uint32_t clk, uint32_t frame, uint32_t seri, uint32_t sero)`
Description	Configure four DIOs for the PCM interface
Inputs	slave = PCM master/slave configuration; use PCM_SELECT_[MASTER | SLAVE] cfg = DIO pin configuration for the PCM pads clk = DIO to use as the PCM clock pad frame = DIO to use as the PCM frame pad seri = DIO to use as the PCM serial input pad sero = DIO to use as the PCM serial output pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 0, 1, 2, and 3 as a master PCM interface. */` `Sys_PCM_DIOConfig(PCM_SELECT_MASTER, APP_DIO_CFG, 0, 1, 2, 3);`

9.119 SYS_PCM_DISABLE

Disable the PCM interface without changing other PCM configuration settings

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_Disable(void)`
Description	Disable the PCM interface without changing other PCM configuration settings
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Disable the PCM Interface. */` `Sys_PCM_Disable();`

9.120 SYS_PCM_ENABLE

Enable the PCM interface without changing other PCM configuration settings

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`void Sys_PCM_Enable(void)`
Description	Enable the PCM interface without changing other PCM configuration settings
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Enable the PCM Interface. */` `Sys_PCM_Enable();`

9.121 SYS_PCM_GET_STATUS

Get the current PCM interface status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pcm.h
Template	`uint32_t Sys_PCM_Get_Status(void)`
Description	Get the current PCM interface status
Inputs	None
Outputs	Return value = The current PCM interface status
Assumptions	None

Example	`/* Check for errors on the PCM Interface. */`
	`if (Sys_PCM_Get_Status() != 0)`
	`{`
	`/* An error has occurred. Run the application's error handler. */`
	`AppErrorHandler();`
	`}`

9.122 SYS_POWER_BANDGAPCALIBRATEDCONFIG

Set the band-gap voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_BandGapCalibratedConfig(uint8_t target)`
Description	Set the band-gap voltage trim to the given target based on the calibration trim value specified in NVR4.
Inputs	target = The target band-gap voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the band gap power supply trim for a target of 750 mV */` `result = Sys_Power_BandGapCalibratedConfig(75);`

9.123 SYS_POWER_BANDGAPCONFIG

Configure the band gap supply voltage

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_BandGapConfig(uint32_t cfg_slope, uint32_t cfg_vtrim)`
Description	Configure the band gap supply voltage
Inputs	cfg_vtrim = Reference voltage trimming; use BG_TRIM_0p* cfg_slope = Temperature coefficient trimming; use a 6-bit number
Outputs	None
Assumptions	None

Example	`/* Configure temperature dependency 0 ppm/C and reference voltage`
	` * trim on 0.750V. */`
	`Sys_Power_BandGapConfig(0x6, BG_TRIM_0P750V);`

9.124 SYS_POWER_BANDGAPSTATUS

Read Bandgap status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`uint32_t Sys_Power_BandGapStatus(void)`
Description	Read Bandgap status
Inputs	None
Outputs	return value = content of the BG register; compare with BG_NOT_READY | BG_READY, and BG_TRIM_0p*V
Assumptions	None
Example	`/* Read Bandgap status. */` `result = Sys_Power_BandGapStatus();`

9.125 SYS_POWER_DCDCCALIBRATEDCONFIG

Set the DC-DC voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_DCDCCalibratedConfig(uint8_t target)`
Description	Set the DC-DC voltage trim to the given target based on the calibration trim value specified in NVR4. The DC-DC power supply is not enabled.
Inputs	target = The target DCDC voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VCC power supply trim for a target of 1.20 V */` `result = Sys_Power_DCDCCalibratedConfig(120);`

9.126 SYS_POWER_GET_RESETANALOG

Read ACS reset source status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`uint32_t Sys_Power_Get_ResetAnalog(void)`
Description	Read ACS reset source status
Inputs	None
Outputs	return value = read status of reset source; compare with CLK_DET_RESET_FLAG_SET, VDDM_RESET_FLAG_SET, VDDC_RESET_FLAG_SET, PAD_RESET_FLAG_SET, and POR_RESET_FLAG_SET
Assumptions	None
Example	`/* Read the reset source status. */` `result = Sys_Power_Get_ResetAnalog();`

9.127 SYS_POWER_GET_RESETDIGITAL

Read digital reset source status

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`uint32_t Sys_Power_Get_ResetDigital(void)`
Description	Read digital reset source status
Inputs	None
Outputs	return value = setting for reset source status; use ACS_RESET_[NOT_SET | SET], CM3_SW_RESET_[NOT_SET | SET], WATCHDOG_RESET_[NOT_SET | SET], and LOCKUP_NOT_[NOT_SET | SET]
Assumptions	None
Example	`/* Read the value of digital reset source. */` `result = Sys_Power_Get_ResetDigital();`

9.128 SYS_POWER_RESETANALOGCLEARFLAGS

Clear all the analog reset flags

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_ResetAnalogClearFlags(void)`
Description	Clear all the analog reset flags
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset POR, PAD, VDDC, VDDM and CLK_DET flags. */` `Sys_Power_ResetAnalogClearFlags();`

9.129 SYS_POWER_RESETDIGITALCLEARFLAGS

Clear all the digital reset flags

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_ResetDigitalClearFlags(void)`
Description	Clear all the digital reset flags
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Reset LOCKUP, Watchdog time out, CM3 software and ACS flags. */` `Sys_Power_ResetDigitalClearFlags();`

9.130 SYS_POWER_VCCCONFIG

Configure DC-DC/ LDO supply

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VCCConfig(uint32_t cfg)`
Description	Configure DC-DC/ LDO supply
Inputs	cfg = DC-DC/ LDO supply value; use VCC_ICHTRIM_*MA, VCC_[MULTI | SINGLE]_PULSE, VCC_CONSTANT_[CHARGE | IMAX], VCC_[BUCK | VBAT], and VCC_TRIM_1p*V_BYTE
Outputs	None
Assumptions	None

Example	`/* Configure DC-DC/ LDO supply to 1 V nominal output voltage`
	` * in 16 mA max charge current. Single pulse mode control with`
	` * constant charge transfer is chosen. The buck converter`
	` * is enabled. */`
	`Sys_Power_VCCConfig(VCC_ICHTRIM_16MA |`
	` VCC_SINGLE_PULSE |`
	` VCC_CONSTANT_CHARGE |`
	` VCC_BUCK |`
	` VCC_TRIM_1P00V);`

9.131 SYS_POWER_VDDACONFIG

Configure analog voltage maximum current and sleep mode clamp control

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VDDAConfig(uint32_t cfg)`
Description	Configure analog voltage maximum current and sleep mode clamp control
Inputs	cfg = Configuration for output power trimming; use VDDA_PTRIM_*MA
Outputs	None
Assumptions	None

Example	`/* Configure VDDA to 8 mA charge pump max current charge pump.`
	` * VCC is shorted to VDDA in sleep mode. */`
	`Sys_Power_VDDAConfig(VDDA_PTRIM_8MA);`

9.132 SYS_POWER_VDDCCALIBRATEDCONFIG

Set the VDDC voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDCCalibratedConfig(uint8_t target)`
Description	Set the VDDC voltage trim to the given target based on the calibration trim value specified in NVR4.
Inputs	target = The target VDDC voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDC power supply trim for a target of 1.15 V */` `result = Sys_Power_VDDCCalibratedConfig(115);`

9.133 SYS_POWER_VDDCCONFIG

Configure digital core voltage regulator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VDDCConfig(uint32_t cfg)`
Description	Configure digital core voltage regulator
Inputs	cfg = setting standby voltage trimming, low power mode control, sleep mode clamp control and output voltage trimming configuration; use VDDC_TRIM_*V, VDDC_SLEEP_[HIZ | GND], VDDC_[LOW| NOMINAL]_BIAS, and VDDC_STANDBY_TRIM_*V
Outputs	None
Assumptions	None

Example	`/* Configure VDDC standby voltage 0.75 V and output voltage`
	` * 1 V in nominal biasing. The clamp output is grounded`
	` * in sleep mode. */`
	`Sys_Power_VDDCConfig(VDDC_TRIM_1P00V |`
	` VDDC_SLEEP_GND |`
	` VDDC_NOMINAL_BIAS |`
	` VDDC_STANDBY_TRIM_0P75V);`

9.134 SYS_POWER_VDDCSTANDBYCALIBRATEDCONFIG

Set the VDDC standby voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDCStandbyCalibratedConfig( uint8_t target)`
Description	Set the VDDC standby voltage trim to the given target based on the calibration trim value specified in NVR4.
Inputs	target = The target VDDC standby voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDC standby power supply trim for a target of 800 mV */` `result = Sys_Power_VDDCStandbyCalibratedConfig(80);`

9.135 SYS_POWER_VDDMCALIBRATEDCONFIG

Set the VDDM voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDMCalibratedConfig(uint8_t target)`
Description	Set the VDDM voltage trim to the given target based on the calibration trim value specified in NVR4.
Inputs	target = The target VDDM voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDM power supply trim for a target of 1.15 V */` `result = Sys_Power_VDDMCalibratedConfig(115);`

9.136 SYS_POWER_VDDMCONFIG

Configure memories' voltage regulator setting

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VDDMConfig(uint32_t cfg)`
Description	Configure memories' voltage regulator setting
Inputs	cfg = value of the memory voltage regulator; use register VDDM_TRIM_*V, VDDM_SLEEP_[HIZ | GND], VDDM_[LOW| NOMINAL]_BIAS, and VDDM_STANDBY_TRIM_*V
Outputs	None
Assumptions	None

Example	`/* Configure VDDM standby voltage at 0.75 V and output voltage`
	` * at 1 V in nominal biasing. The clamp output is grounded in`
	` * sleep mode. */`
	`Sys_Power_VDDMConfig(VDDM_TRIM_1P00V |`
	` VDDM_SLEEP_GND |`
	` VDDM_NOMINAL_BIAS |`
	` VDDM_STANDBY_TRIM_0P75V);`

9.137 SYS_POWER_VDDMSTANDBYCALIBRATEDCONFIG

Set the VDDM standby voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDMStandbyCalibratedConfig( uint8_t target)`
Description	Set the VDDM standby voltage trim to the given target based on the calibration trim value specified in NVR4.
Inputs	target = The target VDDM standby voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDM standby power supply trim for a target of 800 mV */` `result = Sys_Power_VDDMStandbyCalibratedConfig(80);`

9.138 SYS_POWER_VDDPACALIBRATEDCONFIG

Set the VDDPA voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDPACalibratedConfig(uint8_t target)`
Description	Set the VDDPA voltage trim to the given target based on the calibration trim value specified in NVR4. The VDDPA power supply is not enabled.
Inputs	target = The target VDDPA voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDPA power supply trim for a target of 1.30 V */` `result = Sys_Power_VDDPACalibratedConfig(130);`

9.139 SYS_POWER_VDDPACONFIG

Configure power amplifier RF block regulator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VDDPAConfig(uint32_t cfg)`
Description	Configure power amplifier RF block regulator
Inputs	cfg = Power amplifier supply control, enable current sensing circuit, enable control and output voltage trimming configuration; use VDDPA_[DISABLE | ENABLE, VDDPA_TRIM_*V, VDDPA_ISENSE_[DISABLE | ENABLE], and VDDPA_SW_[HIZ | GND]
Outputs	None
Assumptions	None

Example	`/* Power amplifier output connected to VDDRF regulator,`
	` * the VDDPA regulator is enabled, the current sensing circuit`
	` * is disabled and VDDPA is configured for a nominal 1.05 V`
	` * output voltage trim. */`
	`Sys_Power_VDDPAConfig(VDDPA_ENABLE |`
	` VDDPA_TRIM_1P05V |`
	` VDDPA_ISENSE_ENABLE |`
	` VDDPA_SW_HIZ);`

9.140 SYS_POWER_VDDRFCALIBRATEDCONFIG

Set the VDDRF voltage trim to the given target based on the calibration trim value specified in NVR4

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.c
Template	`unsigned int Sys_Power_VDDRFCalibratedConfig(uint8_t target)`
Description	Set the VDDRF voltage trim to the given target based on the calibration trim value specified in NVR4. The VDDRF power supply is not enabled.
Inputs	target = The target VDDRF voltage in 10*mV
Outputs	return value = A code indicating whether an error has occurred.
Assumptions	None
Example	`/* Load the VDDRF power supply trim for a target of 1.10 V */` `result = Sys_Power_VDDRFCalibratedConfig(110);`

9.141 SYS_POWER_VDDRFCONFIG

Configure RF block regulator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power.h
Template	`void Sys_Power_VDDRFConfig(uint32_t cfg)`
Description	Configure RF block regulator
Inputs	cfg = set the RF block regulator; use VDDRF_TRIM_*V, VDDRF_[DISABLE | ENABLE], and VDDRF_DISABLE_[HIZ | GND]
Outputs	None
Assumptions	None

Example	`/* Configure VDDRF regulator nominal output voltage on 1.0 V.`
	` * The clamp control output is in floating. */`
	`Sys_Power_VDDRFConfig(VDDRF_TRIM_1P00V |`
	` VDDRF_ENABLE |`
	` VDDRF_DISABLE_HIZ);`

9.142 SYS_POWERMODES_SLEEP

Configure the system, save register and memory banks of the BLE, then enter Sleep Mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Sleep( struct sleep_mode_env_tag *sleep_mode_env)`
Description	Configure the system, save register and memory banks of the BLE, then enter Sleep Mode
Inputs	sleep_mode_env = Parameters and configurations for the Sleep Mode
Outputs	None
Assumptions	It is safe to enter Sleep Mode (this should be checked before calling this function), DMA channel 0 is available

Example	`/* Assume the configuration of Sleep Mode in sleep_mode_init_env`
	` * and sleep_mode_env parameters were initialized. Also,`
	` * Sys_PowerModes_Sleep_Init(&sleep_mode_init_env) or`
	` * Sys_PowerModes_Sleep_Init_2Mbps(&sleep_mode_init_env) was called. */`
	
	`/* Configure the system, save register and memory banks`
	` * of the BLE, then enter Sleep Mode. */`
	`Sys_PowerModes_Sleep(&sleep_mode_env);`

9.143 SYS_POWERMODES_SLEEP_INIT

Initialize some system blocks for Sleep Mode, save RF register and memory banks excluding 2 Mbps bank, configure retention regulators of supply voltages

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Sleep_Init( struct sleep_mode_init_env_tag *sleep_mode_env)`
Description	Initialize some system blocks for Sleep Mode, save RF register and memory banks excluding 2 Mbps bank, configure retention regulators of supply voltages
Inputs	sleep_mode_env = Parameters and configurations for the Sleep Mode
Outputs	None
Assumptions	RF bank 1 (2 Mbps) does not need to be saved

Example	`/* Assume the configuration of Sleep Mode in sleep_mode_init_env * parameter was initialized. */`
	
	`/* Initialize some system blocks for Sleep Mode, save RF * register and memory banks excluding 2 Mbps bank, configure * retention regulators of supply voltages. */`
	`Sys_PowerModes_Sleep_Init(&sleep_mode_init_env);`

9.144 SYS_POWERMODES_SLEEP_INIT_2MBPS

Initialize some system blocks for Sleep Mode, save RF register and memory banks including 2 Mbps bank, configure retention regulators of supply voltages

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Sleep_Init_2Mbps( struct sleep_mode_init_env_tag *sleep_mode_env)`
Description	Initialize some system blocks for Sleep Mode, save RF register and memory banks including 2 Mbps bank, configure retention regulators of supply voltages
Inputs	sleep_mode_env = Parameters and configurations for the Sleep Mode
Outputs	None
Assumptions	None

Example	`/* Assume the configuration of Sleep Mode in sleep_mode_init_env * parameter was initialized. */`
	
	`/* Initialize some system blocks for Sleep Mode, save RF * register and memory banks including 2 Mbps bank, configure * retention regulators of supply voltages. */`
	`Sys_PowerModes_Sleep_Init_2Mbps(&sleep_mode_init_env);`

9.145 SYS_POWERMODES_SLEEP_WAKEUPFROMFLASH

Configure the system and enter Sleep Mode (wake up from flash)

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Sleep_WakeupFromFlash( struct sleep_mode_flash_env_tag *sleep_mode_env)`
Description	Configure the system and enter Sleep Mode (wake up from flash)
Inputs	sleep_mode_env = Parameters and configurations for the Sleep Mode
Outputs	None
Assumptions	None

Example	`/* Assume the configuration of Sleep Mode in sleep_mode_flash_env`
	` * parameter was initialized and`
	` * Sleep_Mode_Configure(&sleep_mode_flash_env) was called. */`
	
	`/* Configure the system and enter Sleep Mode (wake up from flash) */`
	`Sys_PowerModes_Sleep_WakeupFromFlash(&sleep_mode_flash_env);`

9.146 SYS_POWERMODES_STANDBY

Configure the system and enter Standby Mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Standby( struct standby_mode_env_tag *standby_mode_env)`
Description	Configure the system and enter Standby Mode
Inputs	standby_mode_env = Parameters and configurations for the Standby Mode
Outputs	None
Assumptions	Any retention regulator needed has been enabled Desired wake-up source has been set up before calling this function At least one interrupt needs to be enabled before going to Standby Mode and asserted after wake-up event to wake up the ARM Cortex-M3 processor from WFI

Example	`/* Assume the configuration of Standby Mode in standby_mode_env * parameter was initialized and * Sys_PowerModes_Standby_Init(&standby_mode_env) was called. */`
	
	`/* Configure the system and enter Standby Mode */`
	`Sys_PowerModes_Standby(&standby_mode_env);`

9.147 SYS_POWERMODES_STANDBY_WAKEUP

Execute steps required to wake up the system from Standby Mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Standby_Wakeup( struct standby_mode_env_tag *standby_mode_env)`
Description	Execute steps required to wake up the system from Standby Mode
Inputs	Pre-defined = and configurations for the Standby Mode
Outputs	None
Assumptions	None

Example	`/* Execute steps required to wake up the system from Standby Mode */`
	`Sys_PowerModes_Standby_Wakeup(&standby_mode_env);`
	
	`/* Functions to be performed after waking up from Standby Mode`
	`* start here. */`

9.148 SYS_POWERMODES_WAKEUP

Execute steps required to wake up the system from Sleep Mode RF register bank 1 (2 Mbps) is not restored

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Wakeup(void)`
Description	Execute steps required to wake up the system from Sleep Mode RF register bank 1 (2 Mbps) is not restored
Inputs	None
Outputs	None
Assumptions	DMA channels 0 and 1 are available Start RC oscillator is calibrated to 3 MHz RF bank 1 (2 Mbps) does not need to be restored

Example	`/* Execute steps required to wake up the system from Sleep Mode.`
	` * RF register bank 1 (2 Mbps) is not restored. */`
	`Sys_PowerModes_Wakeup(&sleep_mode_env);`
	
	`/* Functions to be performed after waking up from Sleep Mode`
	` * start here. */`

9.149 SYS_POWERMODES_WAKEUP_2MBPS

Execute steps required to wake up the system from Sleep Mode RF register bank 1 (2 Mbps) is also restored

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_power_modes.c
Template	`void Sys_PowerModes_Wakeup_2Mbps(void)`
Description	Execute steps required to wake up the system from Sleep Mode RF register bank 1 (2 Mbps) is also restored
Inputs	None
Outputs	None
Assumptions	DMA channels 0 and 1 are available Start RC oscillator is calibrated to 3 MHz

Example	`/* Execute steps required to wake up the system from Sleep Mode.`
	` * RF register bank 1 (2 Mbps) is also restored. */`
	`Sys_PowerModes_Wakeup_2Mbps(&sleep_mode_env);`
	
	`/* Functions to be performed after waking up from Sleep Mode`
	` * start here. */`

9.150 SYS_PROGRAMROM_UNLOCKDEBUG

Run the unlock routine from the ProgramROM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`void Sys_ProgramROM_UnlockDebug(void)`
Description	Run the unlock routine from the ProgramROM. WARNING: This will unlock the device by erasing the flash and SRAM memories!
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Unlock the SWJ-DP after wiping the flash and RAM contents`
	` * to protect the application IP (does not return). */`
	`Sys_ProgramROM_UnlockDebug();`

9.151 SYS_PWM_CONFIG

Configure a pulse-width modulator

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pwm.h
Template	`void Sys_PWM_Config(uint32_t num, uint32_t period, uint32_t duty)`
Description	Configure a pulse-width modulator
Inputs	num = The PWM interface to configure; use 0 or 1 period = The period length for the PWM in cycles duty = The high part of the period for the PWM in cycles
Outputs	None
Assumptions	PWM period is (period + 1) cycles long PWM duty is high for (duty + 1) cycles

Example	`/* Set PWM0 period to (periodVal + 1). PWM0 duty cycle is high for`
	`* (dutyVal + 1) cycles. */`
	`Sys_PWM_Config(0, periodVal, dutyVal);`

9.152 SYS_PWM_CONFIGALL

Configure both pulse-width modulators with the same configuration

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pwm.h
Template	`void Sys_PWM_ConfigAll(uint32_t period, uint32_t duty)`
Description	Configure both pulse-width modulators with the same configuration
Inputs	period = The period length for the PWMs in cycles duty = The high part of the period for the PWMs in cycles
Outputs	None
Assumptions	PWM period is (period + 1) cycles long PWM duty is high for (duty + 1) cycles
Example	`/* Sets PWM0 and PWM1 periods to (periodVal + 1). PWM0 and PWM1 duty * cycles are high for (dutyVal + 1) cycles. */` `Sys_PWM_ConfigAll(periodVal, dutyVal);`

9.153 SYS_PWM_CONTROL

Set the control configuration for the two PWM interfaces

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pwm.h
Template	`void Sys_PWM_Control(uint32_t cfg)`
Description	Set the control configuration for the two PWM interfaces
Inputs	cfg = The PWM bitband enable/disable setting; use PWM*_[DISABLE | ENABLE], PWM_OFFSET_[DISABLE | ENABLE], and a constant shifted to PWM_CTRL_PWM_OFFSET_Pos
Outputs	None
Assumptions	None
Example	`/* Enable both PWM interfaces with an offset of 10 cycles between * PWM0 and PWM1. */` `Sys_PWM_Control(PWM0_ENABLE | PWM1_ENABLE | PWM_OFFSET_ENABLE | (10 << PWM_CTRL_PWM_OFFSET_Pos));`

9.154 SYS_PWM_DIOCONFIG

Configure DIO for the specified PWM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pwm.h
Template	`void Sys_PWM_DIOConfig(uint32_t numinv, uint32_t cfg, uint32_t pwm)`
Description	Configure DIO for the specified PWM
Inputs	numinv = PWM number; use DIO_MODE_[PWM0 | PWM0_INV | PWM1 | PWM1_INV] cfg = DIO pin configuration for the PWM output pwm = DIO to use as the PWM output pad
Outputs	None
Assumptions	None
Example	`/* Configure DIO 1 as the PWM0 interface and non-inverted. */` `Sys_PWM_DIOConfig(DIO_MODE_PWM0, APP_DIO_CFG, 1);`

9.155 SYS_PWM_ENABLE

Enable or disable a PWM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_pwm.h
Template	`void Sys_PWM_Enable(uint32_t num, uint32_t cfg)`
Description	Enable or disable a PWM
Inputs	num = The PWM interface to enable or diable; use 0 or 1 cfg = The PWM bitband enable/disable setting; use PWM*_[DISABLE | ENABLE]_BITBAND
Outputs	None
Assumptions	None
Example	`/* Enable PWM0 interface. */` `Sys_PWM_Enable(0, PWM0_ENABLE_BITBAND);`

9.156 SYS_READNVR4

Read data from NVR4 using a function implemented in ROM

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_romvect.h
Template	`unsigned int Sys_ReadNVR4(unsigned int calib_info_ptr,`
Description	Read data from NVR4 using a function implemented in ROM.
Inputs	info_ptr = The base register for the specified calibration information. numReads = The number of words to be read. data = A pointer to the variable that will hold the read data.
Outputs	return value = A code indicating whether an error has occurred. data = The data read from NVR4 will be contained here.
Assumptions	None
Example	`/* Read the default VDDRF configuration from NVR4 into data */` `Sys_ReadNVR4(MANU_INFO_VDDRF, 1, (unsigned int *)&data);`

9.157 SYS_RFFE_INPUTDIOCONFIG

Configure a DIO pad as an RF front-end general-purpose input

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rffe.h
Template	`void Sys_RFFE_InputDIOConfig(uint32_t num, uint32_t cfg, uint32_t gpi)`
Description	Configure a DIO pad as an RF front-end general-purpose input
Inputs	num = GPIO interface pad to configure; use 0 to 9 cfg = DIO pin configuration for the output pads gpi = DIO to use as the GPI pad; use RF_GPIO[0:9]_SRC_DIO_[0:15], and RF_GPIO[0:9]_SRC_CONST_[LOW | HIGH]
Outputs	None
Assumptions	None
Example	`/* Configure DIO 4 as an input pin of the chip and connect it to RF * GPIO3 in other-end. */` `Sys_RFFE_InputDIOConfig(3, DIO_WEAK_PULL_UP, RF_GPIO3_SRC_DIO_4);`

9.158 SYS_RFFE_OUTPUTDIOCONFIG

Configure DIO pads connected to RF front-end GPIO

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rffe.h
Template	`void Sys_RFFE_OutputDIOConfig(uint32_t num, uint32_t cfg, uint32_t gpio)`
Description	Configure DIO pads connected to RF front-end GPIO
Inputs	num = GPIO interface pad to configure; use 0 to 9 cfg = DIO pin configuration for the output pads gpo = DIO to use as the GPO pad
Outputs	None
Assumptions	None
Example	`/* Configure DIO 2 as an output pin of the chip and connect it to RF * GPIO5 in other-end. */ Sys_RFFE_OutputDIOConfig(5, APP_DIO_CFG, 2);`

9.159 SYS_RFFE_SETTXPOWER

Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm

Type	Function	
Include File	`#include <rsl10.h>`	
Source File	rsl10_sys_rffe.c	
Template	`unsigned int Sys_RFFE_SetTXPower(int target)`	
Description	Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm. This function sets VDDRF, VDDPA, and PA_PWR (RF_REG19) when applicable. Note - This function provides RF TX power configurations that match the requested levels, without considering the potential for increased power consumption due to the use of VDDPA.	
Inputs	target	= Target transmission power in the range from -17 to +6 dBm in 1 dBm increments
Outputs	return value	= ERRNO_NO_ERROR; ERRNO_RFFE_INVALIDSETTING_ERROR: if target is out of the expected range; ERRNO_RFFE_MISSINGSETTING_ERROR: if the device is missing the manufacturing reference trim values in NVR4; ERRNO_RFFE_INSUFFICIENTVCC_ERROR: if the configured VCC target may not be enough to guarantee the expected target TX power. The function might still try to reach the desired target
Assumptions	The calibrated voltage values exist in device NVR4 VCC has been configured to an appropriate level for the expected battery level	
Example	`/* Set the radio TX power to 0 dBm */` `result = Sys_RFFE_SetTXPower(0);`	

9.160 SYS_RFFE_SPIDIOCONFIG

Configure the SPI slave for the RF front-end to use DIOs as the SPI master source

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rffe.h
Template	`void Sys_RFFE_SPIDIOConfig(uint32_t cfg, uint32_t mosi, uint32_t csn, uint32_t clk, uint32_t miso)`
Description	Configure the SPI slave for the RF front-end to use DIOs as the SPI master source
Inputs	cfg = DIO pin configuration for the output pads mosi = DIO to use as the MOSI pad csn = DIO to use as the CSN pad clk = DIO to use as the CLK pad miso = DIO to use as the MISO pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 1, 2, 3, 4 as the SPI interface to the * RF front-end. */ Sys_RFFE_SPIDIOConfig(APP_DIO_CFG, 1, 2, 3, 4);`

9.161 SYS_RTC_CONFIG

Configure RTC block

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rtc.h
Template	`void Sys_RTC_Config(uint32_t start_value, uint32_t rtc_ctrl_cfg)`
Description	Configure RTC block
Inputs	start_value = Start value for the RTC timer counter; use a 32 bit value rtc_ctrl_cfg = RTC control register; use RTC_RESET, RTC_FORCE_CLOCK, RTC_ALARM_*, RTC_[DISABLE | ENABLE], and RTC_CLK_SRC_[XTAL32K | RC_OSC]
Outputs	None
Assumptions	None

Example	`/* RTC timer count period of 30.518 us, RTC is reset and enabled,`
	` * RTC alarm invoke every 1 s and the RTC is RC Oscillator. */`
	`Sys_RTC_Config(0, RTC_RESET |`
	` RTC_ALARM_1S |`
	` RTC_ENABLE |`
	` RTC_CLK_SRC_RC_OSC);`

9.162 SYS_RTC_START

Enable or disable the RTC

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rtc.h
Template	`void Sys_RTC_Start(uint32_t cfg)`
Description	Enable or disable the RTC
Inputs	cfg = Value for enabling or disabling RTC; use RTC_[DISABLE | ENABLE]_BITBAND
Outputs	None
Assumptions	None
Example	`/* Enable the RTC timer. */` `Sys_RTC_Start(RTC_ENABLE_BITBAND);`

9.163 SYS_RTC_VALUE

Read the current value of the RTC timer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_rtc.h
Template	`void Sys_RTC_Value(uint32_t cfg)`
Description	Read the current value of the RTC timer
Inputs	None
Outputs	return value = RTC timer counter current value
Assumptions	None
Example	`/* Read the value of the RTC timer. */` `result = Sys_RTC_Value();`

9.164 SYS_SPI_CONFIG

Configure the specified SPI interface's operation and controller information

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_Config(uint32_t num, uint32_t cfg)`
Description	Configure the specified SPI interface's operation and controller information
Inputs	num = SPI interface to configure; use 0 or 1 cfg = Interface operation configuration; use SPI*_OVERRUN_INT_[DISABLE | ENABLE], SPI*_UNDERRUN_INT_[DISABLE | ENABLE], SPI*_CONTROLLER_[CM3 | DMA], SPI*_SELECT_[MASTER | SLAVE], SPI*_CLK_POLARITY_[NORMAL | INVERSE], SPI*_MODE_SELECT_[MANUAL | AUTO], SPI*_[DISABLE | ENABLE] and SPI*_PRESCALE_*
Outputs	None
Assumptions	None

Example	`/* Configure SPI0 for master-mode writes of 16-bit data (controlled * by the ARM Cortex-M3 processor), running the SPI0 at 1/4 of the * system clock frequency. */`
	`Sys_SPI_Config(0, SPI0_OVERRUN_INT_DISABLE |`
	`SPI0_UNDERRUN_INT_DISABLE |`
	`SPI0_CONTROLLER_CM3 |`
	`SPI0_SELECT_MASTER |`
	`SPI0_CLK_POLARITY_NORMAL |`
	`SPI0_MODE_SELECT_AUTO |`
	`SPI0_ENABLE |`
	`SPI0_PRESCALE_4);`
	
	`Sys_SPI_TransferConfig(0, SPI0_IDLE |`
	`SPI0_WRITE_DATA |`
	`SPI0_CS_1 |`
	`SPI0_WORD_SIZE_16);`

9.165 SYS_SPI_DIOCONFIG

Configure four DIOs for the specified SPI interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_DIOConfig(uint32_t num, uint32_t slave, uint32_t cfg, uint32_t clk, uint32_t cs, uint32_t seri, uint32_t sero)`
Description	Configure four DIOs for the specified SPI interface
Inputs	num = SPI interface to configure; use 0 or 1 slave = SPI master/slave configuration; use SPI*_SELECT_[MASTER | SLAVE] cfg = DIO pin configuration for the SPI pads clk = DIO to use as the SPI clock pad cs = DIO to use as the SPI chip select pad seri = DIO to use as the SPI serial input pad sero = DIO to use as the SPI serial output pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 5, 6, 7, and 8 as SPI interface 1 in slave mode */` `Sys_SPI_DIOConfig(1, SPI1_SELECT_SLAVE, APP_DIO_CFG, 5, 6, 7, 8);`

9.166 SYS_SPI_MASTERINIT

Initialize an SPI operation on a specified SPI interface when running this interface in master mode

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_MasterInit(uint32_t num)`
Description	Initialize an SPI operation on a specified SPI interface when running this interface in master mode
Inputs	num = SPI interface to configure; use 0 or 1
Outputs	None
Assumptions	The SPI interface is currently idle The SPI interface is configured for master mode operation If writing over the SPI interface, the data to be written has been queued

Example	`/* Read the latest word from SPI0 before starting the next`
	` * transfer. */`
	`tempData = SPI0->RX_DATA;`
	`Sys_SPI_MasterInit(0);`

9.167 SYS_SPI_READ

Configure the interface to read the specified number of bits over the specified SPI interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_Read(uint32_t num, uint8_t bits)`
Description	Configure the interface to read the specified number of bits over the specified SPI interface
Inputs	num = SPI interface to read from; use 0 or 1 bits = Word size used by the SPI interface; use SPI*_WORD_SIZE_*
Outputs	None
Assumptions	The SPI interface is currently idle The SPI interface is configured for master mode operation

Example	`/* Read the latest word from SPI0 before reading the next SPI0`
	` * byte. */`
	`tempData = SPI0->RX_DATA;`
	`Sys_SPI_Read(0, 8);`

9.168 SYS_SPI_READWRITE

Configure the interface to read and write the specified number of bits over the specified SPI interface (full-duplex)

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_ReadWrite(uint32_t num, uint8_t bits)`
Description	Configure the interface to read and write the specified number of bits over the specified SPI interface (full-duplex)
Inputs	num = SPI interface to read from; use 0 or 1 bits = Number of bits to transmit and receive (between 1 and 32)
Outputs	None
Assumptions	The SPI interface is currently idle The SPI interface is configured for master mode operation The data to be written has been queued

Example	`/* Echo back to slave device the most recently received SPI0 word,`
	` * while reading the next word. */`
	`SPI0->TX_DATA = SPI0->RX_DATA;`
	`Sys_SPI_ReadWrite(0, 32);`

9.169 SYS_SPI_TRANSFERCONFIG

Configure the SPI transfer information for the specified SPI interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_TransferConfig(uint32_t num, uint32_t cfg)`
Description	Configure the SPI transfer information for the specified SPI interface
Inputs	num = SPI interface to configure; use 0 or 1 cfg = Interface transfer configuration; use SPI*_[IDLE | START], SPI*_[WRITE | READ | RW]_DATA, SPI*_CS_*, and SPI*_WORD_SIZE_* or a constant shifted to SPI*_CTRL1_SPI0_WORD_SIZE_Pos
Outputs	None
Assumptions	None

Example	`/* Configure SPI0 for master-mode writes of 32-bit data`
	` * (controlled by the ARM Cortex-M3 processor), running the SPI0`
	` * at 1/2 of the system clock frequency. */`
	`Sys_SPI_Config(0, SPI0_OVERRUN_INT_DISABLE |`
	` SPI0_UNDERRUN_INT_DISABLE |`
	` SPI0_CONTROLLER_CM3 |`
	` SPI0_SELECT_MASTER |`
	` SPI0_CLK_POLARITY_NORMAL |`
	` SPI0_MODE_SELECT_AUTO |`
	` SPI0_ENABLE |`
	` SPI0_PRESCALE_2);`
	
	`Sys_SPI_TransferConfig(0, SPI0_IDLE |`
	` SPI0_WRITE_DATA |`
	` SPI0_CS_1 |`
	` SPI0_WORD_SIZE_32);`

9.170 SYS_SPI_WRITE

Configure the interface to write the specified number of bits over the specified SPI interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_spi.h
Template	`void Sys_SPI_Write(uint32_t num, uint8_t bits)`
Description	Configure the interface to write the specified number of bits over the specified SPI interface
Inputs	num = SPI interface to read from; use 0 or 1 bits = Number of bits to transmit (between 1 and 32)
Outputs	None
Assumptions	The SPI interface is currently idle The SPI interface is configured for master mode operation The data to be written has been queued

Example	`/* Queue up and transmit the next SPI0 byte. */`
	`SPI0->TX_DATA = tempData;`
	`Sys_SPI_Write(0, 8);`

9.171 SYS_TIMER_BBCONFIG

Configure the baseband timer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_timers.h
Template	`void Sys_Timer_BBConfig(uint32_t cfg)`
Description	Configure the baseband timer
Inputs	cfg = Value for configuration of the baseband timer clock; use: BB_TIMER_[RESET | NRESET], BB_CLK_PRESCALE_*
Outputs	None
Assumptions	None
Example	`/* Reset and configure the baseband timer with prescale 1. */` `Sys_Timer_BBConfig(BB_TIMER_RESET | BB_CLK_PRESCALE_1);`

9.172 SYS_TIMER_GET_STATUS

Return the current running or stopped status of the specified general-purpose system timer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_timers.h
Template	`uint32_t Sys_Timer_Get_Status(uint32_t num)`
Description	Return the current running or stopped status of the specified general-purpose system timer
Inputs	num = Timer to read status from; use 0, 1, 2, or 3
Outputs	return value = The current timer status; value loaded from TIMER_CTRL_[*].TIMER_STATUS_ALIAS
Assumptions	None
Example	`/* Get current running or stopped status of timer 0. */` `status = Sys_Timer_Get_Status(0);`

9.173 SYS_TIMER_SET_CONTROL

Set up a general-purpose system timer

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_timers.h
Template	`void Sys_Timer_Set_Control(uint32_t num, uint32_t cfg)`
Description	Set up a general-purpose system timer
Inputs	num = Timer to configure; use 0, 1, 2, or 3 cfg = Control configuration for the specified timer; use TIMER_MULTI_COUNT_*, TIMER_[SHOT_MODE | FREE_RUN], TIMER_PRESCALE_* and a timeout count setting
Outputs	None
Assumptions	None

Example	`/* Configure general-purpose timer 0 as a free-running timer,`
	` * triggering every second for a slow clock of 1.28 MHz. */`
	`Sys_Timer_Set_Control(0, TIMER_FREE_RUN | TIMER_PRESCALE_32 | 40000);`

9.174 SYS_TIMERS_START

Start the specified general-purpose system timers

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_timers.c
Template	`void Sys_Timers_Start(uint32_t cfg)`
Description	Start the specified general-purpose system timers
Inputs	cfg = Timers to start; use the SELECT_TIMER* settings or SELECT_[ALL | NO]_TIMERS to indicate which timers to start
Outputs	None
Assumptions	None
Example	`/* Start timer 0 and timer 2. */` `Sys_Timers_Start(SELECT_TIMER0 | SELECT_TIMER2);`

9.175 SYS_TIMERS_STOP

Stop the specified general-purpose system timers

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_timers.c
Template	`void Sys_Timers_Stop(uint32_t cfg)`
Description	Stop the specified general-purpose system timers
Inputs	cfg = Timers to stop; use the SELECT_TIMER* settings or SELECT_[ALL | NO]_TIMERS to indicate which timers to stop
Outputs	None
Assumptions	None
Example	`/* Stop timer 0 and timer 2. */` `Sys_Timers_Stop(SELECT_TIMER0 | SELECT_TIMER2);`

9.176 SYS_UART_DIOCONFIG

Configure two DIOs for the specified UART interface

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_uart.h
Template	`void Sys_UART_DIOConfig(uint32_t cfg, uint32_t tx, uint32_t rx)`
Description	Configure two DIOs for the specified UART interface
Inputs	cfg = DIO pin configuration for the UART pads tx = DIO to use as the UART transmit pad rx = DIO to use as the UART receive pad
Outputs	None
Assumptions	None
Example	`/* Configure DIOs 9 and 10 as UART interface 1. */` `Sys_UART_DIOConfig(APP_DIO_CFG, 9, 10);`

9.177 SYS_UART_DISABLE

Disable the UART

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_uart.h
Template	`void Sys_UART_Disable(void)`
Description	Disable the UART
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Disable the UART0 interface. */` `Sys_UART_Disable();`

9.178 SYS_WAIT_FOR_EVENT

Hold the ARM Cortex-M3 core waiting for an event, interrupt request, abort or debug entry request (ARM Thumb-2 WFE instruction)

Type	Macro
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_cm3.h
Template	`SYS_WAIT_FOR_EVENT`
Description	Hold the ARM Cortex-M3 core waiting for an event, interrupt request, abort or debug entry request (ARM Thumb-2 WFE instruction)
Inputs	None
Outputs	None
Assumptions	None

Example	`/* Wait for an event, interrupt request, abort or debug entry`
	` * request. */`
	`SYS_WAIT_FOR_EVENT;`

9.179 SYS_WAIT_FOR_INTERRUPT

Hold the ARM Cortex-M3 core waiting for an interrupt request, abort or debug entry request (ARM Thumb-2 WFI instruction)

Type	Macro
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_cm3.h
Template	`SYS_WAIT_FOR_INTERRUPT`
Description	Hold the ARM Cortex-M3 core waiting for an interrupt request, abort or debug entry request (ARM Thumb-2 WFI instruction)
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Wait for an interrupt request, abort or debug entry request. */` `SYS_WAIT_FOR_INTERRUPT;`

9.180 SYS_WATCHDOG_REFRESH

Refresh the watchdog timer count

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_watchdog.h
Template	`void Sys_Watchdog_Refresh(void)`
Description	Refresh the watchdog timer count
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Refresh the watchdog timer count. */` `Sys_Watchdog_Refresh();`

9.181 SYS_WATCHDOG_SET_TIMEOUT

Set the watchdog timeout period

Type	Function
Include File	`#include <rsl10.h>`
Source File	rsl10_sys_watchdog.h
Template	`void Sys_Watchdog_Set_Timeout(uint32_t timeout)`
Description	Set the watchdog timeout period
Inputs	timeout = Timeout value for watchdog; use WATCHDOG_TIMEOUT_*
Outputs	None
Assumptions	None

Example	`/* Set up watchdog timeout value to 2.048ms for a 1MHz watchdog`
	` * input clock. */`
	`Sys_Watchdog_Set_Timeout(WATCHDOG_TIMEOUT_2M048);`

CHAPTER 10

Math Library Reference

This reference chapter presents a detailed description of all the functions in the Arm Cortex-M3 core math library, including calling parameters, returned values, and assumptions.

10.1 MATH_ADD_FRAC32

Add two 32-bit signed fractional numbers, then saturate

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_Add_frac32(int32_t x, int32_t y)`
Description	Add two 32-bit signed fractional numbers, then saturate. The result is one of the following: • (x + y), if MIN_FRAC32 <= (x + y) <= MAX_FRAC32 • MAX_FRAC32, if (x + y) > MAX_FRAC32 • MIN_FRAC32, if (x + y) < MIN_FRAC32
Inputs	x = Fractional number represented as a 32 bit integer y = Fractional number represented as a 32 bit integer
Outputs	z = (x + y) with saturation
Assumptions	None

Example	`/* Addition with saturation: add two 32-bit signed fractional`
	` * numbers, then saturate to 0x7FFFFFFF if positive overflow`
	` * occurs, or 0x80000000 if negative overflow occurs. */`
	`z = Math_Add_frac32(x, y);`

10.2 MATH_ATTACKRELEASE

Calculate a first-order attack-release filter

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_float.c
Template	`float Math_AttackRelease(float a, float b, float x, float y1)`
Description	Calculate a first-order attack-release filter. The outputs of the attack-release filter are calculated as: • y[n] = beta * x[n] + (1 - beta) * y[n-1] Where: • beta = a if x[n] >= y[n-1] • beta = b if x[n] < y[n-1] • a is the coefficient for attack, 0 < a < 1 • b is the coefficient for release, 0 < b < 1
Inputs	a = Filter coefficient b = Filter coefficient x = Current input y1 = Previous output
Outputs	y = New output
Assumptions	None

Example	`/* Dual-time constant attack release averaging filter: generate new`
	` * output data based on the filter coefficients a and b, new input`
	` * data data_in, and previous output data data_out. */`
	`data_out = Math_AttackRelease(a, b, data_in, data_out);`

10.3 MATH_ATTACKRELEASE_FRAC32

Calculate a fixed-point first-order attack-release filter

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_AttackRelease_frac32(int32_t a, int32_t b, int32_t x, int32_t y1)`
Description	Calculate a fixed-point first-order attack-release filter. The outputs of the attack-release filter are calculated as: • y[n] = beta * x[n] + (1 - beta) * y[n-1] Where: • beta = a if x[n] >= y[n-1] • beta = b if x[n] < y[n-1] • a is the coefficient for attack, 0 < a < 1 • b is the coefficient for release, 0 < b < 1
Inputs	a = Filter coefficient b = Filter coefficient x = Current input y1 = Previous output
Outputs	y = New output
Assumptions	None
Example	`/* Dual-time constant attack release averaging filter: generate new * output data based on the filter coefficients a and b, new input * data data_in, and previous output data data_out. All input and * output types are 32-bit signed fractional. */` `data_out = Math_AttackRelease_frac32(a, b, data_in, data_out);`

10.4 MATH_EXPAVG

Calculate a first-order exponential average

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_float.c
Template	`float Math_ExpAvg(float alpha, float x, float y1)`
Description	Calculate a first-order exponential average. The outputs of the exponential average are calculated as: • y[n] = alpha * x[n] + (1 - alpha) * y[n-1] Where: • 0 < alpha < 1
Inputs	alpha = Filter coefficient x = Current input y1 = Previous output
Outputs	y = New output
Assumptions	None

Example	`/* Exponential moving average filter: generate new output data based`
	` * on the filter coefficient alpha, new input data data_in, and`
	` * previous output data data_out. */`
	`data_out = Math_ExpAvg(alpha, data_in, data_out);`

10.5 MATH_EXPAVG_FRAC32

Calculate a fixed-point first-order exponential average

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_ExpAvg_frac32(int32_t alpha, int32_t x, int32_t y1)`
Description	Calculate a fixed-point first-order exponential average. The outputs of the exponential average are calculated as: • y[n] = alpha * x[n] + (1 - alpha) * y[n-1] Where: • 0 < alpha < 1
Inputs	alpha = Filter coefficient x = Current input y1 = Previous output
Outputs	y = New output
Assumptions	None

Example	`/* Exponential moving average filter: generate new output data based`
	` * on the filter coefficient alpha, new input data data_in, and`
	` * previous output data data_out. All input and output types are`
	` * 32-bit signed fractional. */`
	`data_out = Math_ExpAvg_frac32(alpha, data_in, data_out);`

10.6 MATH_LINEARINTERP

Calculate linear interpolation on the interval [x0, x1)

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_float.c
Template	`float Math_LinearInterp(float x0, float x1, float y0, float y1, float x)`
Description	Calculate linear interpolation on the interval [x0, x1). The interpolation is calculated as: • y = y0 + ((y1 - y0) / (x1 - x0)) * (x - x0)
Inputs	x0 = First boundary point x-axis value x1 = Second boundary point x-axis value y0 = First boundary point y-axis value y1 = Second boundary point y-axis value x = Interpolation point
Outputs	y = Interpolated value
Assumptions	x0 != x1

Example	`/* Linear interpolation on the interval [x0, x1) with boundary points`
	` * (x0, y0) and (x1, y1). */`
	`y = Math_LinearInterp(x0, x1, y0, y1, x);`

10.7 MATH_LINEARINTERP_FRAC32

Calculate fixed-point linear interpolation on the interval [0, 1)

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_LinearInterp_frac32(int32_t y0, int32_t y1, int32_t x)`
Description	Calculate fixed-point linear interpolation on the interval [0, 1). The interpolation is calculated as: • y = y0 + x * (y1 - y0)
Inputs	y0 = Left boundary point y1 = Right boundary point x = Interpolation point
Outputs	y = Interpolated value
Assumptions	0 <= x < 1

Example	`/* Linear interpolation on the interval [0, 1) with boundary points`
	` * y0 and y1. */`
	`y = Math_LinearInterp_frac32(y0, y1, x);`

10.8 MATH_MULT_FRAC32

Multiply two 32-bit signed fractional numbers, then saturate

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_Mult_frac32(int32_t x, int32_t y)`
Description	Multiply two 32-bit signed fractional numbers, then saturate. The result is either: • x * y, if x > MIN_FRAC32 or y > MIN_FRAC32 • MAX_FRAC32, if x = MIN_FRAC32 and y = MIN_FRAC32
Inputs	x = Fractional number represented as a 32 bit integer y = Fractional number represented as a 32 bit integer
Outputs	z = (x * y) with saturation
Assumptions	None

Example	`/* Multiplication with saturation: multiply two 32-bit signed`
	` * fractional numbers, then saturate to 0x7FFFFFFF if overflow`
	` * occurs. */`
	`z = Math_Mult_frac32(x, y);`

10.9 MATH_SINGLEVAR_REG

Find the least-squares solution for a single variable linear regression model

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_float.c
Template	`void Math_SingleVar_Reg(float* x, float* y, unsigned int N, float* a)`
Description	Find the least-squares solution for a single variable linear regression model. A linear regression model with a single predictor variable can be represented as: • y[i] = a0 + a1 * x[i] + e[i], i = 0, 1, 2, ..., N-1
Inputs	x = Pointer to the input variable vector x[] y = Pointer to the dependent variable vector y[] N = Length of vector x[] and y[]
Outputs	a = Pointer to the coefficient vector {a0, a1}
Assumptions	x[] and y[] are of the same length x[] is not a constant vector (constant vector here means x[0] = x[1] = ... = x[N-1]) a is a pointer to a coefficient vector of length 2

Example	`/* For an input variable vector x[] and a dependent variable vector`
	` * y[], find the least-squares solution to the linear regression`
	` * model. */`
	`Math_SingleVar_Reg(x, y, DATA_LENGTH, coeff_vector);`

10.10 MATH_SUB_FRAC32

Subtract one 32-bit signed fractional number from another, then saturate

Type	Function
Include File	`#include <rsl10_math.h>`
Source File	rsl10_math_frac32.c
Template	`int32_t Math_Sub_frac32(int32_t x, int32_t y)`
Description	Subtract one 32-bit signed fractional number from another, then saturate. The result is one of the following: • (x - y), if MIN_FRAC32 <= (x - y) <= MAX_FRAC32 • MAX_FRAC32, if (x - y) > MAX_FRAC32 • MIN_FRAC32, if (x - y) < MIN_FRAC32
Inputs	x = Fractional number represented as a 32 bit integer y = Fractional number represented as a 32 bit integer
Outputs	z = (x - y) with saturation
Assumptions	None

Example	`/* Subtract with saturation: subtract one 32-bit signed fractional`
	` * number from another, then saturate to 0x7FFFFFFF if positive`
	` * overflow occurs, or 0x80000000 if negative overflow occurs. */`
	`z = Math_Sub_frac32(x, y);`

CHAPTER 11

Flash Library Reference

This reference chapter presents a detailed description of all the functions in the flash write support library, including calling parameters, returned values, and assumptions. Warning: All functions provided by the flash library should be executed from RAM or ROM, as executing them from flash can result in hidden, flash-access-related failures.

11.1 FLASH_ERASEALL

Erase all of the sectors in the main block of the flash

Type	Function
Include File	`#include <rsl10_flash.h>`
Source File	rsl10_flash.c
Template	`unsigned int Flash_EraseAll(void)`
Description	Erase all of the sectors in the main block of the flash
Inputs	None
Outputs	return value = Status code indicating whether the requested flash operation succeeded
Assumptions	The calling application has unlocked all of the main flash instance, and any NVR or redundancy sectors that should be erased If the flash is in sequential programming mode, it is safe to exit this mode

Example	`/* Configure the flash to allow writing to the whole flash */`
	`FLASH->MAIN_WRITE_UNLOCK = FLASH_MAIN_KEY;`
	`FLASH->MAIN_CTRL = (MAIN_LOW_W_ENABLE | MAIN_MIDDLE_W_ENABLE |`
	`MAIN_HIGH_W_ENABLE);`
	
	`/* Erase the main flash */`
	`Flash_EraseAll();`

11.2 FLASH_ERASESECTOR

Erase the specified flash sector

Type	Function
Include File	`#include <rsl10_flash.h>`
Source File	rsl10_flash.c
Template	`unsigned int Flash_EraseSector(unsigned int addr)`
Description	Erase the specified flash sector. This sector could be in the main flash, one of the NVR sectors, or one of the redundancy sectors. Verify the sector was erased, progressively trying the different sector erase pulses until one successfully erases the sector.
Inputs	addr = Address of data in the sector to be erased
Outputs	return value = Status code indicating whether the requested flash operation succeeded
Assumptions	The calling application has unlocked the flash for erase If the flash is in sequential programming mode, it is safe to exit this mode None of the RECALL, VREAD0_MODE and VREAD1_MODE bits are set in FLASH_IF_CTRL

Example	`/* Configure the flash to allow writing to the lower flash area */`
	`FLASH->MAIN_WRITE_UNLOCK = FLASH_MAIN_KEY;`
	`FLASH->MAIN_CTRL = MAIN_LOW_W_ENABLE;`
	
	`/* Erase the first sector of the main flash */`
	`Flash_EraseSector(FLASH_MAIN_BASE);`

11.3 FLASH_WRITEBUFFER

Write a buffer of memory of the specified length, starting at the specified address, to flash

Type	Function
Include File	`#include <rsl10_flash.h>`
Source File	rsl10_flash.c
Template	`unsigned int Flash_WriteBuffer(unsigned int start_addr, unsigned int length, unsigned int* data)`
Description	Write a buffer of memory of the specified length, starting at the specified address, to flash
Inputs	start_addr = Start address for the write to flash length = Number of words to write to flash data = Pointer to the data to write to flash
Outputs	return value = Status code indicating whether the requested flash operation succeeded
Assumptions	The calling application has unlocked the flash for write The areas of flash to be written have been previously erased (if necessary) and are not currently write protected "data" points to a buffer of at least "length" words The address in flash is even word aligned The number of words to write is an even number If the flash is already in sequential programming mode, it is safe to exit this mode to perform the buffered write. None of the RECALL, VREAD0_MODE and VREAD1_MODE bits are set in FLASH_IF_CTRL

Example	`/* Configure the flash to allow writing to the lower flash area */`
	`FLASH->MAIN_WRITE_UNLOCK = FLASH_MAIN_KEY;`
	`FLASH->MAIN_CTRL = MAIN_LOW_W_ENABLE;`
	
	`/* Write the first words of the main flash with data from a`
	`* previously loaded buffer (assumes this sector has been`
	`* previously erased) */`
	`Flash_WriteBuffer(FLASH_MAIN_BASE, bufferLength, buffer);`

11.4 FLASH_WRITECOMMAND

Safely issue a flash command; blocks waiting for the flash to be idle before running the command and again before returning

Type	Function	
Include File	`#include <rsl10_flash.h>`	
Source File	rsl10_flash.c	
Template	`void Flash_WriteCommand(uint32_t command)`	
Description	Safely issue a flash command; blocks waiting for the flash to be idle before running the command and again before returning.	
Inputs	command	= Command to be written to FLASH_CMD_CTRL; use CMD_*
Outputs	return value	= Status code indicating whether the flash interface can be written; returns FLASH_ERR_INACCESSIBLE if the flash is isolated or not powered, otherwise returns no error.
Assumptions	None	
Example	`/* Force a wakeup the flash */` `Flash_WriteCommand(CMD_WAKE_UP);`	

11.5 FLASH_WRITEINTERFACECONTROL

Safely write the interface control register; blocks waiting for the flash to be idle before writing the interface control register, and again before returning

Type	Function	
Include File	`#include <rsl10_flash.h>`	
Source File	rsl10_flash.c	
Template	`void Flash_WriteInterfaceControl(uint32_t ctrl)`	
Description	Safely write the interface control register; blocks waiting for the flash to be idle before writing the interface control register, and again before returning.	
Inputs	ctrl	= Data to write to the FLASH_IF_CTRL register
Outputs	return value	= Status code indicating whether the flash interface can be written; returns FLASH_ERR_INACCESSIBLE if the flash is isolated or not powered and LP_MODE, RECALL, VREAD0_MODE or VREAD1_MODE are being changed, otherwise returns no error.
Assumptions	If the flash is in sequential programming mode, it is safe to exit this mode No more than two of the LP_MODE, RECALL, VREAD0_MODE and VREAD1_MODE bits are being updated in FLASH_IF_CTRL	
Example	`/* Disable the flash recall settings */` `Flash_WriteInterfaceControl(FLASH_RECALL_DISABLE);`	

11.6 FLASH_WRITEWORDPAIR

Write a word pair of flash at the specified address

Type	Function
Include File	`#include <rsl10_flash.h>`
Source File	rsl10_flash.c
Template	`unsigned int Flash_WriteWordPair(unsigned int addr, unsigned int data0, unsigned int data1)`
Description	Write a word pair of flash at the specified address
Inputs	addr = Address to write in the flash data0 = First data word to write to the specified address in flash data1 = Second data word to write to the specified (address + 4) in flash
Outputs	return value = Status code indicating whether the requested flash operation succeeded
Assumptions	The calling application has unlocked the flash for write The area of flash to be written has been previously erased (if necessary) and is not currently write protected If the flash is in sequential programming mode, it is safe to exit this mode None of the RECALL, VREAD0_MODE and VREAD1_MODE bits are set in FLASH_IF_CTRL

Example	`/* Configure the flash to allow writing to the lower flash area */`
	`FLASH->MAIN_WRITE_UNLOCK = FLASH_MAIN_KEY;`
	`FLASH->MAIN_CTRL = MAIN_LOW_W_ENABLE;`
	
	`/* Write the first word of the main flash with a test value (assumes`
	`* this sector has been previously erased) */`
	`Flash_WriteWordPair(FLASH_MAIN_BASE, 0x12345678, 0x9ABCDEF0);`

CHAPTER 12

Calibration Library Reference

This reference chapter presents a detailed description of all the functions in the calibration support library, including calling parameters, returned values, and assumptions.

12.1 CALIBRATE_CLOCK_32K_RCOSC

Used to calibrate the 32K RC oscillator to a specified frequency

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_clock.c
Template	`unsigned int Calibrate_Clock_32K_RCOSC(uint32_t target)`
Description	Used to calibrate the 32K RC oscillator to a specified frequency.
Inputs	target = Number of cycles required to achieve the Desired clock frequency in Hz
Outputs	return value = Status code indicating whether the RCOSC calibration succeeded
Assumptions	Calibrate_Clock_Initialize() has been called.
Example	`/* Calibrate the 32K RC oscillator to 30000 Hz */` `result = Calibrate_Clock_32K_RCOSC(30000);`

12.2 CALIBRATE_CLOCK_INITIALIZE

Initialize the system to support the clock calibration, consisting of the 48 MHz XTAL oscillator and RC oscillator

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_clock.c
Template	`void Calibrate_Clock_Initialize(void)`
Description	Initialize the system to support the clock calibration, consisting of the 48 MHz XTAL oscillator and RC oscillator.
Inputs	None
Outputs	None
Assumptions	None
Example	`/* Initialize the system for clock calibration. */` `Calibrate_Clock_Initialize();`

12.3 CALIBRATE_CLOCK_START_OSC

Used to calibrate the startup oscillator to a specified frequency

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_clock.c
Template	`unsigned int Calibrate_Clock_Start_OSC(uint32_t target)`
Description	Used to calibrate the startup oscillator to a specified frequency.
Inputs	target = Desired clock frequency in kHz
Outputs	return value = Status code indicating whether the clock succeeded
Assumptions	Calibrate_Clock_Initialize() has been called.
Example	`/* Calibrate the startup oscillator to 3 MHz */` `result = Calibrate_Clock_Start_OSC(3000);`

12.4 CALIBRATE_POWER_DCDC

Calibrate the DC-DC converter (DCDC)

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_DCDC(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the DC-DC converter (DCDC).
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	VBG has been calibrated. Calibrate_Power_Initialize() has been called.
Example	`/* Calibrate the DC-DC converter to 125 10*mV. */` `result = Calibrate_Power_DCDC(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 125);`

12.5 CALIBRATE_POWER_INITIALIZE

The initialization function does the following tasks: 1) Changes settings in all power supply control registers to their default values

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`void Calibrate_Power_Initialize(void)`
Description	The initialization function does the following tasks: 1) Changes settings in all power supply control registers to their default values. 2) Sets the system clock source to RFCLK/3 (16 MHz). 3) Configures the ADC to enable measurement at 100 Hz
Inputs	None
Outputs	None
Assumptions	VBAT must be less than or equal to 1.3 V
Example	`/* Initialize the system for power supply calibration and configure * ADC to be measured. */ Calibrate_Power_Initialize();`

12.6 CALIBRATE_POWER_VBG

Calibrate the bandgap voltage (VBG) against a specified VBAT supply voltage

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_VBG(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the bandgap voltage (VBG) against a specified VBAT supply voltage. VBG is the reference voltage for the ADC, so it can be calibrated based on the ADC output for a known voltage, which is VBAT.
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	The target band-gap is calibrated by reading the current VBAT supply using the ADC. The assumed VBAT supply voltage is 1.25 V. Calibrate_Power_Initialize() has been called.
Example	`/* Calibrate the bandgap voltage (VBG) to 75 10*mV.*/` `result = Calibrate_Power_VBG(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 75);`

12.7 CALIBRATE_POWER_VDDC

Calibrate the digital core voltage power supply (VDDC)

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_VDDC(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the digital core voltage power supply (VDDC).
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	VBG has been calibrated. Calibrate_Power_Initialize() has been called.
Example	`/* Calibrate the VDDC supply to 118 10*mV. */` `result = Calibrate_Power_VDDC(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 118);`

12.8 CALIBRATE_POWER_VDDM

Calibrate the digital memory voltage (VDDM)

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_VDDM(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the digital memory voltage (VDDM)
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	VBG has been calibrated. Calibrate_Power_Initialize() has been called.
Example	`/* Calibrate the VDDM supply to 118 10*mV. */` `result = Calibrate_Power_VDDM(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 118);`

12.9 CALIBRATE_POWER_VDDPA

Calibrate the radio power amplifier power supply (VDDPA)

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_VDDPA(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the radio power amplifier power supply (VDDPA).
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	VBG has been calibrated. Calibrate_Power_Initialize() has been called.
Example	`/* Calibrate the VDDPA supply to 160 10*mV. */` `result = Calibrate_Power_VDDPA(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 160);`

12.10 CALIBRATE_POWER_VDDRF

Calibrate the radio front-end power supply (VDDRF)

Type	Function
Include File	`#include <rsl10_calibrate.h>`
Source File	rsl10_calibrate_power.c
Template	`unsigned int Calibrate_Power_VDDRF(unsigned int adc_num, uint32_t *adc_ptr, uint32_t target)`
Description	Calibrate the radio front-end power supply (VDDRF).
Inputs	adc_num = ADC channel number [0-7] adc_ptr = Pointer to the ADC data register target = Target voltage readback [10*mV]
Outputs	return value = Status code indicating whether the calibration succeeded
Assumptions	VBG has been calibrated. Calibrate_Power_Initialize() has been called. VCC is sufficiently high to trim VDDRF to the desired value. This is because VCC supplies VDDRF.
Example	`/* Calibrate the VDDRF supply to 125 10*mV. */` `result = Calibrate_Power_VDDRF(0, (uint32_t *)&ADC->DATA_TRIM_CH[0], 125);`

APPENDIX A

Glossary

The following abbreviations and terms are used in this manual:

ACL	asynchronous connection-oriented logical transport
ACS	analog control system
ADC	analog-to-digital converter
AFE	analog front-end
CRC	cyclic redundancy check
CSRK	Connection Signature Resolving Key
DAC	digital-to-analog converter
DIO	digital input/output
DMA	direct memory access
ECC	error correcting code
GAP	generic access profile
GAPC	generic access profile controller
GAPM	generic access profile manager
GPIO	general-purpose input/output
HCI	host controller interface
I^2C	inter-IC communication protocol
I^2S	inter-IC sound protocol
INL	integral non-linearity
IRK	Identity Resolving Key
JTAG	joint test action group (developer of IEEE standard 1149.1-1990)
L2CAP	logical link control and adaptation protocol
L2CC	logical link control controller
LC	link controller

LL	link layer
LLC	link layer controller
LLM	link layer manager
LM	link manager
LDO	low dropout voltage regulator
LSB	least significant bit
MCU	microcontroller unit
MSB	most significant bit
MUX	multiplexer, selector of one signal from many
NVIC	nested vectored interrupt controller
PCM	pulse code modulation
PDU	packet data unit; sub-packet containing a 2-byte L2CAP header and a payload
PLL	phase-locked loop
PMU	power management unit
PWM	pulse width modulation
POR	power-on-reset
RAM	random-access memory
ROM	read-only memory
RTC	real-time clock
SCL	serial clock (part of I^2C bus)
SDA	serial data (part of I^2C bus)
SPI	serial peripheral interface
SWD	serial wire debug, two-wire interface used for communication with Arm cores
SWJ-DP	serial wire and JTAG debug port
TWI	two-wire interface
UART	universal asynchronous receiver-transmitter

VCO	voltage-controlled oscillator
VDD	system voltage
VDDA	analog voltage domain
VDDC	digital core voltage domain
VDDO	I/O supply voltage domain
VDD_XTAL	crystal voltage domain
WDF	wave digital filter
XTAL	crystal, generally quartz-based

Arm and Cortex are trademarks or registered trademarks of Arm Ltd. Bluetooth is a registered trademark of Bluetooth SIG, Inc. All other brand names and product names appearing in this document are trademarks of their respective holders.

ON Semiconductor and ON are trademarks of Semiconductor Components Industries, LLC dba ON Semiconductor or its subsidiaries in the United States and/or other countries. ON Semiconductor owns the rights to a number of patents, trademarks, copyrights, trade secrets, and other intellectual property. A listing of ON Semiconductor's product/patent coverage may be accessed at www.onsemi.com/site/pdf/Patent-Marking.pdf. ON Semiconductor reserves the right to make changes without further notice to any products herein. ON Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does ON Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation special, consequential or incidental damages. Buyer is responsible for its products and applications using ON Semiconductor products, including compliance with all laws, regulations and safety requirements or standards, regardless of any support or applications information provided by ON Semiconductor. "Typical" parameters which may be provided in ON Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. ON Semiconductor does not convey any license under its patent rights nor the rights of others. ON Semiconductor products are not designed, intended, or authorized for use as a critical component in life support systems or any FDA Class 3 medical devices or medical devices with a same or similar classification in a foreign jurisdiction or any devices intended for implantation in the human body. Should Buyer purchase or use ON Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold ON Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that ON Semiconductor was negligent regarding the design or manufacture of the part. ON Semiconductor is an Equal Opportunity/Affirmative Action Employer. This literature is subject to all applicable copyright laws and is not for resale in any manner.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for ON Semiconductor
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support: 800-282-9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

ON Semiconductor Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative