

CC2640 and CC2650 SimpleLink™ *Bluetooth*® low energy Software Stack 2.2.0

Developer's Guide



Literature Number: SWRU393C
October 2010–Revised June 2016

| | |
|--|-----------|
| Preface | 11 |
| 1 Overview | 16 |
| 1.1 Introduction..... | 16 |
| 1.2 Bluetooth low energy Protocol Stack Basics | 17 |
| 2 TI Bluetooth low energy Software Development Platform | 19 |
| 2.1 Hardware and Software Architecture Overview | 20 |
| 2.1.1 ARM Cortex M0 (Radio Core) | 20 |
| 2.1.2 ARM Cortex M3 (System Core) | 21 |
| 2.2 Protocol Stack and Application Configurations..... | 21 |
| 2.3 Solution Platform | 22 |
| 2.4 Directory Structure | 22 |
| 2.4.1 Examples Folder | 23 |
| 2.4.2 Source Folder..... | 23 |
| 2.5 Sample Applications | 24 |
| 2.6 Setting up the Integrated Development Environment | 24 |
| 2.6.1 Installing the SDK | 25 |
| 2.6.2 IAR | 25 |
| 2.6.3 Code Composer Studio..... | 29 |
| 2.7 Working With Hex Files..... | 34 |
| 2.8 Accessing Preprocessor Symbols | 34 |
| 2.9 Top-Level Software Architecture | 36 |
| 2.9.1 Standard Project Task Hierarchy | 37 |
| 3 RTOS Overview | 38 |
| 3.1 RTOS Configuration | 38 |
| 3.2 Semaphores | 39 |
| 3.2.1 Initializing a Semaphore..... | 40 |
| 3.2.2 Pending a Semaphore..... | 40 |
| 3.2.3 Posting a Semaphore..... | 40 |
| 3.3 RTOS Tasks | 40 |
| 3.3.1 Creating a Task..... | 41 |
| 3.3.2 Creating the Task Function | 42 |
| 3.4 Clocks..... | 42 |
| 3.4.1 API | 43 |
| 3.4.2 Functional Example | 44 |
| 3.5 Queues | 44 |
| 3.5.1 API | 45 |
| 3.5.2 Functional Example | 46 |
| 3.6 Idle Task..... | 47 |
| 3.7 Power Management..... | 47 |
| 3.8 Hardware Interrupts | 48 |
| 3.9 Software Interrupts | 48 |
| 3.10 Flash | 49 |
| 3.10.1 Flash Memory Map | 50 |
| 3.10.2 Application and Stack Flash Boundary | 51 |
| 3.10.3 Using Simple NV for Flash Storage..... | 51 |

| | | |
|----------|--|-----------|
| 3.10.4 | Customer Configuration Area | 52 |
| 3.11 | Memory Management (RAM) | 53 |
| 3.11.1 | RAM Memory Map | 53 |
| 3.11.2 | Application and Stack RAM Boundary | 54 |
| 3.11.3 | System Stack | 54 |
| 3.11.4 | Dynamic Memory Allocation | 54 |
| 3.11.5 | Initializing RTOS Objects..... | 55 |
| 3.12 | Configuration of RAM and Flash Boundary Using the Frontier Tool | 56 |
| 3.12.1 | Frontier Tool Operation..... | 56 |
| 3.12.2 | Disabling the Frontier Tool | 57 |
| 4 | The Application | 59 |
| 4.1 | Start-Up in main() | 59 |
| 4.2 | ICall | 60 |
| 4.2.1 | Introduction..... | 60 |
| 4.2.2 | ICall Bluetooth low energy Protocol Stack Service | 61 |
| 4.2.3 | ICall Primitive Service | 61 |
| 4.2.4 | ICall Initialization and Registration | 62 |
| 4.2.5 | ICall Thread Synchronization | 63 |
| 4.2.6 | Example ICall Usage | 64 |
| 4.3 | General Application Architecture | 65 |
| 4.3.1 | Application Initialization Function..... | 65 |
| 4.3.2 | Event Processing in the Task Function | 66 |
| 4.3.3 | Callbacks | 70 |
| 5 | The Bluetooth low energy Protocol Stack..... | 71 |
| 5.1 | Generic Access Profile (GAP) | 71 |
| 5.1.1 | Connection Parameters | 72 |
| 5.1.2 | Effective Connection Interval | 73 |
| 5.1.3 | Connection Parameter Considerations | 74 |
| 5.1.4 | Connection Parameter Limitations with Multiple Connections..... | 74 |
| 5.1.5 | Connection Parameter Update | 74 |
| 5.1.6 | Connection Termination | 75 |
| 5.1.7 | GAP Abstraction..... | 75 |
| 5.1.8 | Configuring the GAP Layer | 75 |
| 5.2 | GAPRole Task..... | 75 |
| 5.2.1 | Peripheral Role | 76 |
| 5.2.2 | Central Role..... | 79 |
| 5.3 | Generic Attribute Profile (GATT) | 82 |
| 5.3.1 | GATT Characteristics and Attributes | 82 |
| 5.3.2 | GATT Services and Profile..... | 83 |
| 5.3.3 | GATT Client Abstraction..... | 85 |
| 5.3.4 | GATT Server Abstraction..... | 89 |
| 5.3.5 | Allocating Memory for GATT Procedures | 100 |
| 5.3.6 | Registering to Receive Additional GATT Events in the Application | 101 |
| 5.3.7 | GATT Security..... | 102 |
| 5.4 | GAP Bond Manager and LE Secure Connections | 104 |
| 5.4.1 | Overview | 104 |
| 5.4.2 | Selection of Pairing Mode | 104 |
| 5.4.3 | Using GAPBondMgr | 107 |
| 5.4.4 | GAPBondMgr Examples for Different Pairing Modes | 109 |
| 5.4.5 | LE Privacy 1.2..... | 117 |
| 5.5 | Logical Link Control and Adaptation Layer Protocol (L2CAP) | 119 |
| 5.5.1 | General L2CAP Terminology..... | 120 |
| 5.5.2 | Maximum Transmission Unit (MTU)..... | 121 |

| | | |
|----------|--|------------|
| 5.5.3 | L2CAP Channels..... | 122 |
| 5.5.4 | L2CAP Connection-Oriented Channel (CoC) Example | 122 |
| 5.6 | LE Data Length Extension..... | 123 |
| 5.6.1 | Summary..... | 123 |
| 5.6.2 | Data Length Update Procedure | 123 |
| 5.6.3 | Initial Values | 124 |
| 5.6.4 | Data Length Extension HCI Commands and Events | 124 |
| 5.6.5 | Enabling Extended Packet Length Feature | 125 |
| 5.7 | HCI..... | 126 |
| 5.7.1 | Using HCI and HCI Vendor-Specific Commands in the Application | 126 |
| 5.7.2 | Standard LE HCI Commands | 126 |
| 5.7.3 | HCI Vendor-Specific Commands | 128 |
| 5.8 | Run-Time Bluetooth low energy Protocol Stack Configuration | 132 |
| 5.9 | Configuring Bluetooth low energy Protocol Stack Features..... | 133 |
| 6 | Peripherals and Drivers | 135 |
| 6.1 | Adding a Driver..... | 135 |
| 6.2 | Board File | 136 |
| 6.3 | Board Level Drivers | 136 |
| 6.4 | Creating a Custom Board File | 137 |
| 6.5 | Available Drivers | 137 |
| 6.5.1 | PIN..... | 137 |
| 6.5.2 | UART and SPI | 139 |
| 6.5.3 | Other Drivers | 139 |
| 6.6 | Using 32-kHz Crystal-Less Mode..... | 139 |
| 7 | Sensor Controller | 140 |
| 8 | Startup Sequence | 141 |
| 8.1 | Programming Internal Flash With the ROM Bootloader | 141 |
| 8.2 | Resets | 141 |
| 9 | Development and Debugging | 141 |
| 9.1 | Debug Interfaces..... | 142 |
| 9.1.1 | Connecting to the XDS Debugger | 142 |
| 9.2 | Breakpoints | 143 |
| 9.2.1 | Breakpoints in CCS..... | 143 |
| 9.2.2 | Breakpoints in IAR | 143 |
| 9.2.3 | Considerations When Using Breakpoints With an Active Bluetooth low energy Connection | 144 |
| 9.2.4 | Considerations no Breakpoints and Compiler Optimization..... | 145 |
| 9.3 | Watching Variables and Registers | 145 |
| 9.3.1 | Variables in CCS..... | 145 |
| 9.3.2 | Variables in IAR..... | 146 |
| 9.3.3 | Considerations When Viewing Variables | 146 |
| 9.4 | Memory Watchpoints | 147 |
| 9.4.1 | Watchpoints in CCS | 147 |
| 9.4.2 | Watchpoints in IAR | 147 |
| 9.5 | TI-RTOS Object Viewer | 148 |
| 9.5.1 | Scanning the BIOS for Errors | 148 |
| 9.5.2 | Viewing the State of Each Task | 149 |
| 9.5.3 | Viewing the System Stack | 149 |
| 9.5.4 | Viewing Power Manager Information..... | 150 |
| 9.6 | Profiling the ICall Heap Manager (heapmgr.h)..... | 150 |
| 9.6.1 | Determining the Auto Heap Size | 150 |
| 9.7 | Optimizations | 151 |
| 9.7.1 | Project-Wide Optimizations..... | 152 |

| | | |
|-----------|---|------------|
| 9.7.2 | Single-File Optimizations | 153 |
| 9.7.3 | Single-Function Optimizations | 153 |
| 9.7.4 | Loading RTOS in ROM Symbols | 154 |
| 9.8 | Deciphering CPU Exceptions | 157 |
| 9.8.1 | Exception Cause | 157 |
| 9.8.2 | Using TI-RTOS and ROV to Parse Exceptions | 157 |
| 9.9 | Debugging a Program Exit | 160 |
| 9.10 | Assert Handling | 160 |
| 9.10.1 | Catching Stack Asserts in the Application | 160 |
| 9.10.2 | Catching App Asserts in the Application | 161 |
| 9.11 | Debugging Memory Problems | 162 |
| 9.11.1 | Task and System Stack Overflow | 162 |
| 9.11.2 | Dynamic Allocation Errors | 162 |
| 9.12 | Preprocessor Options | 162 |
| 9.12.1 | Modifying | 162 |
| 9.12.2 | Options | 162 |
| 9.13 | Check System Flash and RAM Usage With Map File | 164 |
| 10 | Creating a Custom Bluetooth low energy Application | 165 |
| 10.1 | Adding a Board File | 165 |
| 10.2 | Configuring Parameters for Custom Hardware | 165 |
| 10.3 | Creating Additional Tasks | 165 |
| 10.4 | Optimizing Bluetooth low energy Stack Memory Usage | 166 |
| 10.4.1 | Additional Memory Configuration Options | 166 |
| 10.5 | Defining Bluetooth low energy Behavior | 167 |
| 11 | Porting from CC254x to CC2640 | 168 |
| 11.1 | Introduction | 168 |
| 11.2 | OSAL | 168 |
| 11.3 | Application and Stack Separation With ICall | 168 |
| 11.4 | Threads, Semaphores, and Queues | 168 |
| 11.5 | Peripheral Drivers | 169 |
| 11.6 | Event Processing | 169 |
| 12 | Sample Applications | 170 |
| 12.1 | Blood Pressure Sensor | 170 |
| 12.1.1 | Interface | 170 |
| 12.1.2 | Operation | 170 |
| 12.2 | Heart Rate Sensor | 171 |
| 12.2.1 | Interface | 171 |
| 12.2.2 | Operation | 171 |
| 12.3 | Cycling Speed and Cadence (CSC) Sensor | 171 |
| 12.3.1 | Interface | 172 |
| 12.3.2 | Operation | 172 |
| 12.3.3 | Neglect Timer | 172 |
| 12.4 | Running Speed and Cadence (RSC) Sensor | 172 |
| 12.4.1 | Interface | 173 |
| 12.4.2 | Operation | 173 |
| 12.4.3 | Neglect Timer | 173 |
| 12.5 | Glucose Collector | 173 |
| 12.5.1 | Interface | 174 |
| 12.5.2 | Record Access Control Point | 174 |
| 12.6 | Glucose Sensor | 174 |
| 12.6.1 | Interface | 174 |
| 12.6.2 | Operation | 175 |
| 12.7 | HID-Emulated Keyboard | 175 |

| | | |
|----------|---|------------|
| 12.7.1 | Interface | 175 |
| 12.7.2 | Operation | 175 |
| 12.8 | HostTest–Bluetooth low energy Network Processor | 176 |
| 12.9 | KeyFob | 176 |
| 12.9.1 | Interface | 176 |
| 12.9.2 | Battery Operation | 176 |
| 12.9.3 | Accelerometer Operation | 176 |
| 12.9.4 | Keys | 177 |
| 12.9.5 | Proximity | 177 |
| 12.10 | SensorTag | 177 |
| 12.10.1 | Operation | 177 |
| 12.10.2 | Sensors | 178 |
| 12.11 | Simple BLE Central | 178 |
| 12.11.1 | Interface | 178 |
| 12.12 | Simple BLE Peripheral | 178 |
| 12.13 | Simple Application Processor | 178 |
| 12.14 | Simple Network Processor | 179 |
| 12.15 | TimeApp | 179 |
| 12.15.1 | Interface | 179 |
| 12.15.2 | Operation | 179 |
| 12.16 | Thermometer | 180 |
| 12.16.1 | Interface | 180 |
| 12.16.2 | Operation | 181 |
| A | GAP API | 182 |
| A.1 | Commands | 182 |
| A.2 | Configurable Parameters | 185 |
| A.3 | Events | 187 |
| B | GAPRole Peripheral Role API | 191 |
| B.1 | Commands | 191 |
| B.2 | Configurable Parameters | 193 |
| B.3 | Callbacks | 196 |
| B.3.1 | State Change Callback (pfnStateChange) | 196 |
| C | GAPRole Central Role API | 197 |
| C.1 | Commands | 197 |
| C.2 | Configurable Parameters | 201 |
| C.3 | Callbacks | 201 |
| C.3.1 | Central Event Callback (eventCB) | 202 |
| D | GATT and ATT API | 203 |
| D.1 | General Commands | 203 |
| D.2 | Server Commands | 203 |
| D.3 | Client Commands | 204 |
| D.4 | Return Values | 212 |
| D.5 | Events | 212 |
| D.6 | GATT Commands and Corresponding ATT Events | 215 |
| D.7 | ATT_ERROR_RSP errCodes | 215 |
| E | GATTServApp API | 215 |
| E.1 | Commands | 216 |
| F | GAPBondMgr API | 218 |
| F.1 | Commands | 218 |
| F.2 | Configurable Parameters | 224 |
| F.3 | Callbacks | 227 |
| F.3.1 | Passcode Callback (passcodeCB) | 227 |

| | | |
|----------|--|------------|
| | F.3.2 Pairing State Callback (pairStateCB) | 227 |
| G | L2CAP API | 229 |
| | G.1 Commands | 229 |
| H | HCI API | 233 |
| | H.1 HCI Commands | 233 |
| | H.2 Vendor-Specific HCI Commands | 234 |
| | H.3 Host Error Codes | 246 |
| I | ICall API | 247 |
| | I.1 Commands | 247 |
| | I.2 Error Codes | 247 |
| | Revision History | 248 |
| | Revision History | 249 |

List of Figures

| | | |
|-------|---|-----|
| 1. | Suggested Workflow | 13 |
| 2. | Project Zero on CCS Cloud | 14 |
| 1-1. | Bluetooth low energy Protocol Stack | 17 |
| 2-1. | SimpleLink CC2640 Block Diagram | 20 |
| 2-2. | Single-Device Processor Configuration | 21 |
| 2-3. | Simple Network Processor Configuration | 21 |
| 2-4. | Bluetooth low energy Stack Development System | 22 |
| 2-5. | Full Verbosity | 26 |
| 2-6. | Custom Argument Variables | 27 |
| 2-7. | IAR Workspace Pane | 28 |
| 2-8. | Installation Details | 30 |
| 2-9. | Installation Details | 30 |
| 2-10. | Import CCS Projects | 32 |
| 2-11. | Project Explorer Structure | 33 |
| 2-12. | IAR Defined Symbols Box | 35 |
| 2-13. | CCS Predefined Symbols | 36 |
| 2-14. | Top-Level Software Architecture | 37 |
| 3-1. | RTOS Execution Threads | 38 |
| 3-2. | Semaphore Functionality | 39 |
| 3-3. | General Task Topology | 42 |
| 3-4. | Queue Messaging Process | 45 |
| 3-5. | Preemption Scenario | 49 |
| 3-6. | System Flash Map | 50 |
| 3-7. | System Memory Map | 53 |
| 3-8. | Disabling Frontier Tool from Stack Project in IAR | 57 |
| 3-9. | Disabling Frontier Tool from Stack Project in CCS | 58 |
| 4-1. | ICall Application – Protocol Stack Abstraction | 60 |
| 4-2. | ICall Messaging Example | 64 |
| 4-3. | SBP Task Flow Chart | 66 |
| 5-1. | GAP State Diagram | 71 |
| 5-2. | Connection Event and Interval | 72 |
| 5-3. | Slave Latency | 73 |
| 5-4. | GAP Abstraction | 75 |
| 5-5. | Application Using GAPRole_TerminateConnection() | 77 |
| 5-6. | Tracing the GAP_LINK_TERMINATED_EVENT | 78 |
| 5-7. | Application Using GAPCentralRole_StartDiscovery() | 79 |
| 5-8. | Tracing the GAP_DEVICE_DISCOVERY_EVENT | 81 |
| 5-9. | GATT Client and Server | 82 |
| 5-10. | Simple GATT Profile Characteristic Table from BTool | 83 |
| 5-11. | GATT Client Abstraction | 85 |
| 5-12. | GATT Server Abstraction | 89 |
| 5-13. | Attribute Table Initialization | 90 |
| 5-14. | Get and Set Profile Parameter | 98 |
| 5-15. | Sniffer Capture Example | 103 |
| 5-16. | Parameters With Secure Connections | 105 |
| 5-17. | Parameters Without Secure Connections | 105 |
| 5-18. | Parameters With IO Capabilities | 106 |

| | | |
|-------|---|-----|
| 5-19. | Flow Diagram Example | 108 |
| 5-20. | Just Works Pairing..... | 110 |
| 5-21. | Interaction Between the GAPBondMgr and the Application | 112 |
| 5-22. | Numeric Comparison | 114 |
| 5-23. | GAPBondMgr Example With Bonding Enabled..... | 116 |
| 5-24. | Resolving List | 118 |
| 5-25. | L2CAP Architectural Blocks | 120 |
| 5-26. | L2CAP Packet Fragmentation | 121 |
| 5-27. | Sample Connection and Data Exchange Between a Master and Slave Device Using a L2CAP Connection-Oriented Channel in LE Credit Based Flow Control Mode | 123 |
| 5-28. | PDU Sizes | 124 |
| 5-29. | Read RSSI Command | 126 |
| 5-30. | RSSI Event | 127 |
| 5-31. | PER Command | 129 |
| 5-32. | PER Event | 130 |
| 5-33. | Memory Mapping | 132 |
| 6-1. | Drivers Folder | 135 |
| 9-1. | Break on Read Access..... | 148 |
| 9-2. | Error Scan | 148 |
| 9-3. | Viewing State of RTOS Tasks | 149 |
| 9-4. | Viewing the System Stack in Hwi..... | 149 |
| 9-5. | Adding RTOS ROM Symbol in IAR Project | 154 |
| 9-6. | IAR Disassembly Without ROM Symbols | 155 |
| 9-7. | IAR Disassembly With ROM | 155 |
| 9-8. | Adding RTOS ROM Symbol in CCS Project | 155 |
| 9-9. | CCS Disassembly Without ROM Symbols | 156 |
| 9-10. | CCS Disassembly With ROM Symbols | 156 |
| 9-11. | Exception Information | 158 |
| 9-12. | PC Exception Example..... | 159 |

List of Tables

| | | |
|------|---|-----|
| 2-1. | SDK Parent Folders..... | 22 |
| 2-2. | Source Folders | 24 |
| 2-3. | Supported Tools and Software | 25 |
| 3-1. | Flash System Map Definitions | 50 |
| 3-2. | OSAL_SNV Values | 51 |
| 3-3. | Boundary Address Symbols | 56 |
| 5-1. | GAP Bond Manager Terminology | 104 |
| 5-2. | Definition of Terms..... | 118 |
| 5-3. | Data Length Update Procedure Sizes and Times | 124 |
| 5-4. | Bluetooth low energy Stack Configuration Parameters | 133 |
| 5-5. | Bluetooth low energy Protocol Stack Features | 133 |
| 9-1. | Application Preprocessor Symbols | 162 |
| 9-2. | Stack Preprocessor Symbols..... | 164 |
| H-1. | API Function Map..... | 233 |
| H-2. | API Function Map..... | 235 |

References

1. *TI Bluetooth low energy Vendor-Specific HCI Reference Guide v2.2*,
C:\ti\simplelink\ble_sdk_2_02_00_xxxxx\TI_BLE_Vendor_Specific_HCI_Guide.pdf
 2. *TI CC26xx Technical Reference Manual*, ([SWCU117](#))
 3. *Measuring Bluetooth Smart Power Consumption Application Report*, ([SWRA478](#))
 4. *TI-RTOS Documentation Overview*,
C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\docs\Documentation_Overview_cc13xx_cc26xx.html
 5. *TI-RTOS Getting Started Guide*,
C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\docs\Getting_Started_Guide_cc13xx_cc26xx.pdf
 6. *TI-RTOS User's Guide*, C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\docs\Users_Guide.pdf
 7. *TI-RTOS SYS/BIOS Kernel User's Guide*,
C:\ti\tirtos_cc13xx_cc26xx_2_18_00_03\products\bios_6_45_02_31\docs\Bios_User_Guide.pdf
 8. *TI-RTOS Power Management for CC26xx*,
C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\docs\Power_Management.pdf
 9. *TI SYS/BIOS API Guide*,
C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\products\bios_6_45_02_31\docs\Bios_APIs.html
 10. *CC26xxware DriverLib API*,
C:\ti\tirtos_cc13xx_cc26xx_2_18_00_03\products\cc26xxware_2_23_03_17162\doc\doc_overview.html
 11. *Sensor Controller Studio*,<http://www.ti.com/tool/sensor-controller-studio>
 12. *TI-RTOS Drivers API Reference*,
C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\products\tidrivertoc13xx_cc26xx_2_16_01_13\docs\doxygen\html\index.html
 13. *CC2640 Simple Network Processor API Guide*,
C:\ti\simplelink\ble_sdk_2_02_00_xxxxx\sap_3_00_01_07\docs\CC2640 Simple Network Processor API Guide.pdf
 14. *ARM Cortex-M3 Devices Generic User's Guide*,
http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf
Available for download from the Bluetooth Special Interest Group (SIG) website.
 15. *Specification of the Bluetooth System, Covered Core Package, Version: 4.2 (02-Dec-2014)*,
https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439
 16. *Device Information Service (Bluetooth Specification), Version 1.0 (24-May-2011)*,
https://www.bluetooth.org/docman/handlers/downloadaddoc.ashx?doc_id=238689
- Links**
17. *TI Bluetooth low energy Wiki*, www.ti.com/ble-wiki
 18. *Latest Bluetooth low energy Stack Download*, www.ti.com/ble-stack

LaunchPad, TI Designs, SimpleLink, Code Composer Studio, SmartRF are trademarks of Texas Instruments.
ARM, Cortex are registered trademarks of ARM Limited.
CCS Cloud is a trademark of Apaliala LLC.
iBeacon is a trademark of Apple Inc.
Bluetooth is a registered trademark of Bluetooth SIG.
iOS is a registered trademark of Cisco.
Android is a trademark of Google Inc.
Eddystone is a trademark of Google Inc..
Intel is a registered trademark of Intel Corporation.
Windows 7 is a registered trademark of Microsoft Inc.
Python is a registered trademark of PSF.
ZigBee is a registered trademark of ZigBee Alliance.

19. *TI E2E Support Forum*, www.ti.com/ble-forum
20. *TI Designs Reference Library*, <http://www.ti.com/general/docs/refdesignsearch.tsp>
21. *TI SimpleLink GitHub Code Examples*: https://github.com/ti-simplelink/ble_examples

Getting Started with Bluetooth LE Development

This section serves as a roadmap for users developing applications and products using the TI SimpleLink Bluetooth low energy CC2640 wireless MCU platform. Whether a seasoned developer or just getting started, TI has created a variety of resources to simplify development on the CC2640 platform. These resources will enhance your experience with the SimpleLink Bluetooth low energy Software Development Kit (SDK) from the out-of-the-box demo to production.

Figure 1 shows the suggested workflow for getting started with TI's SimpleLink Bluetooth low energy development environment.

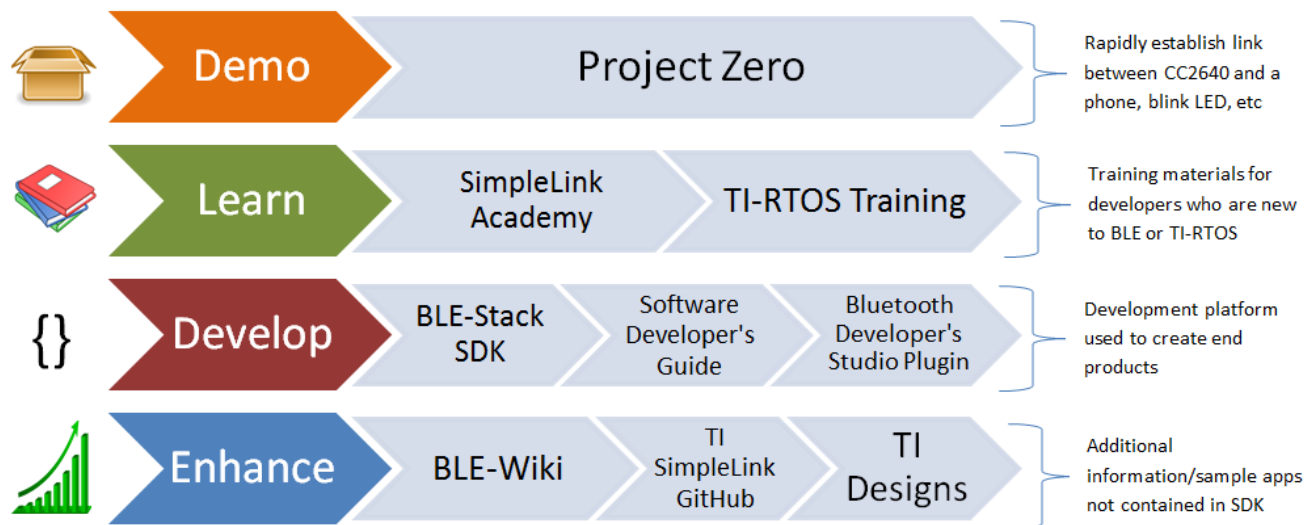


Figure 1. Suggested Workflow

Demo: Project Zero

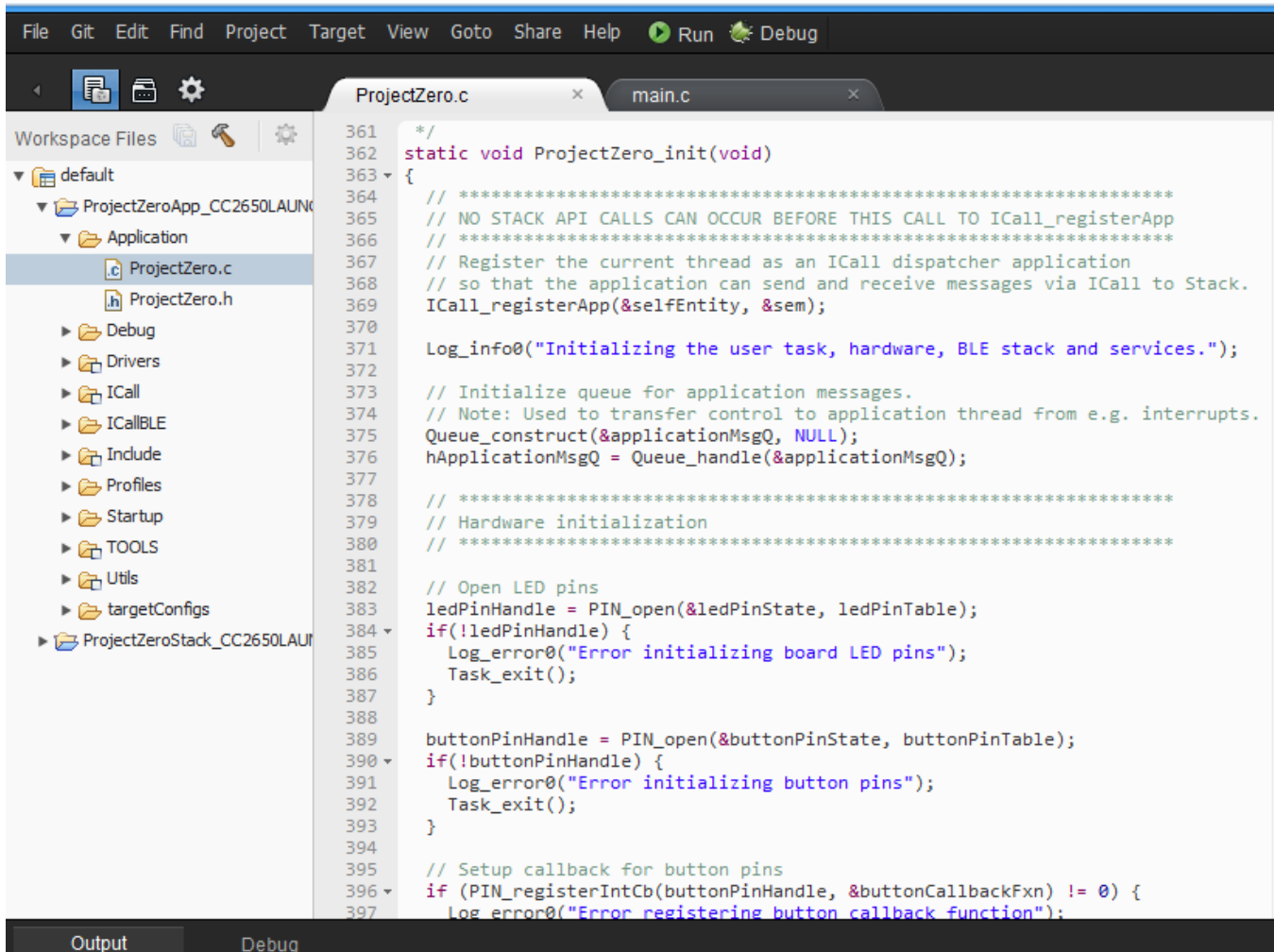
The [CC2650 LaunchPad™](#) is the main development kit for Project Zero and developing applications with the BLE-Stack v2.2 SDK. To quickly get started on a simple project with the CC2650 LaunchPad, see the Project Zero overview page: www.ti.com/ble-project-zero.

Project Zero uses a modified version of the Simple BLE Peripheral (simple_peripheral) sample application from this SDK to demonstrate and evaluate some of the most commonly used Bluetooth low energy features. Using CCS Cloud™, TI's web based IDE, Project Zero can be built, downloaded, and debugged directly from a supported browser without the need to install a full desktop IDE. With Project Zero running on the CC2650 LaunchPad, you can connect with your smart phone to remotely blink on-board LEDs, send text strings to the LaunchPad's serial port, and receive button press notifications, all straight out of the box.

The following sections in this document focus on developing a custom application with the BLE-Stack SDK.

To summarize, Project Zero allows you to quickly:

- Run software without installing any tools
- Flash the device with a single-button click from the browser
- Import a project to the cloud editor and develop, build, and debug
- Download all required project files in one archive file for CCS Desktop development



```

361  /*
362  static void ProjectZero_init(void)
363  {
364  // *****
365  // NO STACK API CALLS CAN OCCUR BEFORE THIS CALL TO ICall_registerApp
366  // *****
367  // Register the current thread as an ICall dispatcher application
368  // so that the application can send and receive messages via ICall to Stack.
369  ICall_registerApp(&selfEntity, &sem);
370
371  Log_info0("Initializing the user task, hardware, BLE stack and services.");
372
373  // Initialize queue for application messages.
374  // Note: Used to transfer control to application thread from e.g. interrupts.
375  Queue_construct(&applicationMsgQ, NULL);
376  hApplicationMsgQ = Queue_handle(&applicationMsgQ);
377
378  // *****
379  // Hardware initialization
380  // *****
381
382  // Open LED pins
383  ledPinHandle = PIN_open(&ledPinState, ledPinTable);
384  if(!ledPinHandle) {
385      Log_error0("Error initializing board LED pins");
386      Task_exit();
387  }
388
389  buttonPinHandle = PIN_open(&buttonPinState, buttonPinTable);
390  if(!buttonPinHandle) {
391      Log_error0("Error initializing button pins");
392      Task_exit();
393  }
394
395  // Setup callback for button pins
396  if (PIN_registerIntCb(buttonPinHandle, &buttonCallbackFxn) != 0) {
397      Log_error0("Error registering button callback function");
    
```

Figure 2. Project Zero on CCS Cloud

Learn

Resources contained in the learn tract of [Figure 1](#) are intended for users who are new to BLE or TI-RTOS. These modules demonstrate how to create custom applications with the BLE-Stack SDK and TI's Real Time Operation System (TI-RTOS).

- **SimpleLink Academy**
 - Using introductory material and labs, learn the fundamentals of BLE and how to develop a custom BLE profile.
 - Contains a single TI-RTOS module written for the CC2640 to demonstrate the rich debug environment and peripheral driver capability provided by the RTOS.
- **TI-RTOS Kernel Workshop**
 - This material teaches users who are new to TI-RTOS or RTOS programming in general about TI's RTOS kernel implementation.
 - Modules within the kernel training greatly expand the information presented in [Chapter 3](#).
 - Learn how the TI-RTOS provides the most optimal power management and design flexibly.

Develop

The develop tract of [Figure 1](#) is intended for programmers who are ready to begin developing an end product using the CC2640.

- **BLE-Stack SDK**
 - The BLE-Stack SDK contains both library and example code to create a complete BLE end application. The library code implements the BLE protocol stack on the CC2640, and the various example projects use the stack library to implement end devices. These examples should be considered as starting points for end product designs.
 - TI recommends that users start their development on a project starting with `simple_`, unless a specific function (such as heart rate, cycling sensor, and so forth) is required.
- **Software Developer's Guide**
 - This document is meant to be used alongside the BLE-Stack SDK when developing an end product. It contains documentation on the stack architecture, APIs, and suggestions for developing applications
- **Bluetooth Developer's Studio Plugin**
 - TI offers a plugin for the [Bluetooth Developer Studio](#) tool. Developers can use this tool, developed by the Bluetooth Special Interest Group (SIG), with TI's SimpleLink plugin to reduce development time by automatically generating compilable code for proprietary and adopted profiles running on the CC2640.

Enhance

The enhance tract of [Figure 1](#) is intended to take your product to the next level by leveraging TI's web-based collateral. Additional resources found in these pages include application-specific source code examples, smart phone source code, and complete sub-system designs.

- **BLE-Wiki**
 - A collection of webpages that contain step-by-step guides and code snippets that enable certain features and enhancements to the CC2640, such as production test mode and certification.
- **TI SimpleLink GitHub**
 - Additional sample applications created to implement specific use cases and examples. These are made to work in conjunction with the SDK.
- **TI Designs™**
 - Examples of system designs containing complete hardware and software examples, using the CC2640.

Overview

The purpose of this document is to give an overview of the TI SimpleLink™ Bluetooth® low energy CC2640 wireless MCU software development kit to begin creating a Bluetooth low energy custom application. This document also introduces the Bluetooth low energy specification. Do not use this document as a substitute for the complete specification. For more details, see the [Specification of the Bluetooth System](#) or some introductory material at the [TI Bluetooth low energy Wiki](#).

1.1 Introduction

Version 4.2 of the Bluetooth specification allows for two systems of wireless technology: Basic Rate (BR: BR/EDR for Basic Rate/Enhanced Data Rate) and Bluetooth low energy. The Bluetooth low energy system was created to transmit small packets of data, while consuming significantly less power than BR/EDR devices.

The TI Bluetooth low energy protocol stack (BLE-Stack) v2.2 includes these new features from Version 4.2 of the specification:

- LE Secure Connections
- LE Data Length extension
- LE Privacy 1.2

The stack also supports the following 4.1 features:

- LE L2CAP Connection-Oriented Channel Support
- LE Link Layer Topology
- LE Ping
- Slave Feature Exchange
- Connection Parameter Request

These features are optional in the 4.2 specification, and can be selectively enabled at build time.

1.2 Bluetooth low energy Protocol Stack Basics

Figure 1-1 shows the Bluetooth low energy protocol stack architecture.

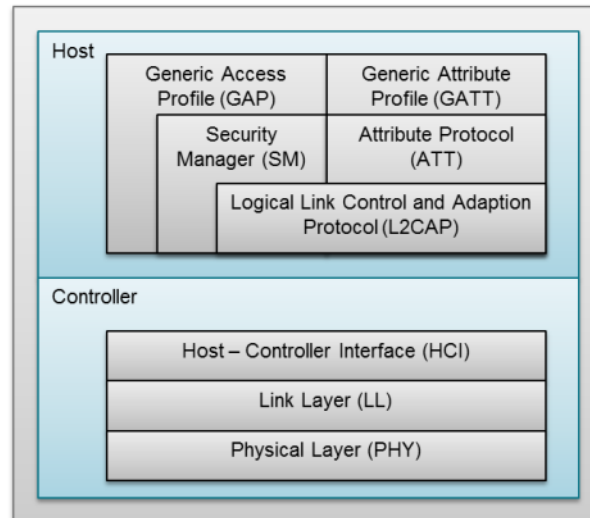


Figure 1-1. Bluetooth low energy Protocol Stack

The Bluetooth low energy protocol stack (or protocol stack) consists of the controller and the host. This separation of controller and host derives from the implementation of classic Bluetooth BR/EDR devices, where the two sections are implemented separately. Any profiles and applications sit on top of the GAP and GATT layers of the protocol stack.

The physical layer (PHY) is a 1-Mbps adaptive frequency-hopping GFSK (Gaussian frequency-shift keying) radio operating in the unlicensed 2.4-GHz ISM (industrial, scientific, and medical) band.

The link layer (LL) controls the RF state of the device, with the device in one of five states:

- Standby
- Advertising
- Scanning
- Initiating
- Connected

Advertisers transmit data without connecting, while scanners scan for advertisers. An initiator is a device that responds to an advertiser with a request to connect. If the advertiser accepts the connection request, both the advertiser and initiator enter a connected state. When a device is connected, it connects as either master or slave. The device initiating the connection becomes the master and the device accepting the request becomes the slave.

The host control interface (HCI) layer provides communication between the host and controller through a standardized interface. This layer can be implemented either through a software API or by a hardware interface such as UART, SPI, or USB. Standard HCI commands and events are specified in the [Specification of the Bluetooth System](#). TI's proprietary commands and events are specified in [TI Bluetooth low energy Vendor-Specific HCI Reference Guide v2.2](#).

The link logical control and adaptation protocol (L2CAP) layer provides data encapsulation services to the upper layers, allowing for logical end-to-end communication of data. The security manager (SM) layer defines the methods for pairing and key distribution, and provides functions for the other layers of the protocol stack to securely connect and exchange data with another device. See [Section 5.3.5](#) for more information on TI's implementation of the SM layer. The generic access protocol (GAP) layer directly interfaces with the application and/or profiles, to handle device discovery and connection-related services for the device. GAP handles the initiation of security features. See [Section 5.1](#) for more information on TI's implementation of the GAP layer.

The attribute protocol (ATT) layer protocol allows a device to expose certain pieces of data or *attributes*, to another device. The generic attribute protocol (GATT) layer is a service framework that defines the sub-procedures for using ATT. Data communications that occur between two devices in a Bluetooth low energy connection are handled through GATT sub-procedures. The application and/or profiles will directly use GATT. See [Section 5.3](#) for more information on TI's implementation of the ATT and GATT layers.

TI Bluetooth low energy Software Development Platform

The TI royalty-free Bluetooth low energy software development kit (SDK) is a complete software platform for developing single-mode Bluetooth low energy applications. This kit is based on the SimpleLink CC2640, complete System-on-Chip (SoC) Bluetooth low energy solution. The CC2640 combines a 2.4-GHz RF transceiver, 128-KB in-system programmable memory, 20KB of SRAM, and a full range of peripherals. The device is centered on an ARM® Cortex®-M3 series processor that handles the application layer and Bluetooth low energy protocol stack and an autonomous radio core centered on an ARM Cortex®-M0 processor that handles all the low-level radio control and processing associated with the physical layer and parts of the link layer. The sensor controller block provides additional flexibility by allowing autonomous data acquisition and control independent of the Cortex-M3 processor, further extending the low-power capabilities of the CC2640. [Figure 2-1](#) shows the block diagram. For more information on the CC2640, see the [CC26xx Technical Reference Manual \(TRM\)](#).

NOTE: This kit supports development of Bluetooth low energy applications on the following SimpleLink wireless MCUs: CC2640 and CC2650. The multi-standard CC2650 wireless MCU supports Bluetooth low energy as well as other wireless protocols, such as ZigBee® and 6LoWPAN. The CC2640 supports Bluetooth low energy only. All code generated from the BLE-Stack v2.x SDK is binary compatible and exchangeable with both the CC2650 and CC2640 wireless MCUs. Throughout this document, CC2640 and CC2650 may be used interchangeably.

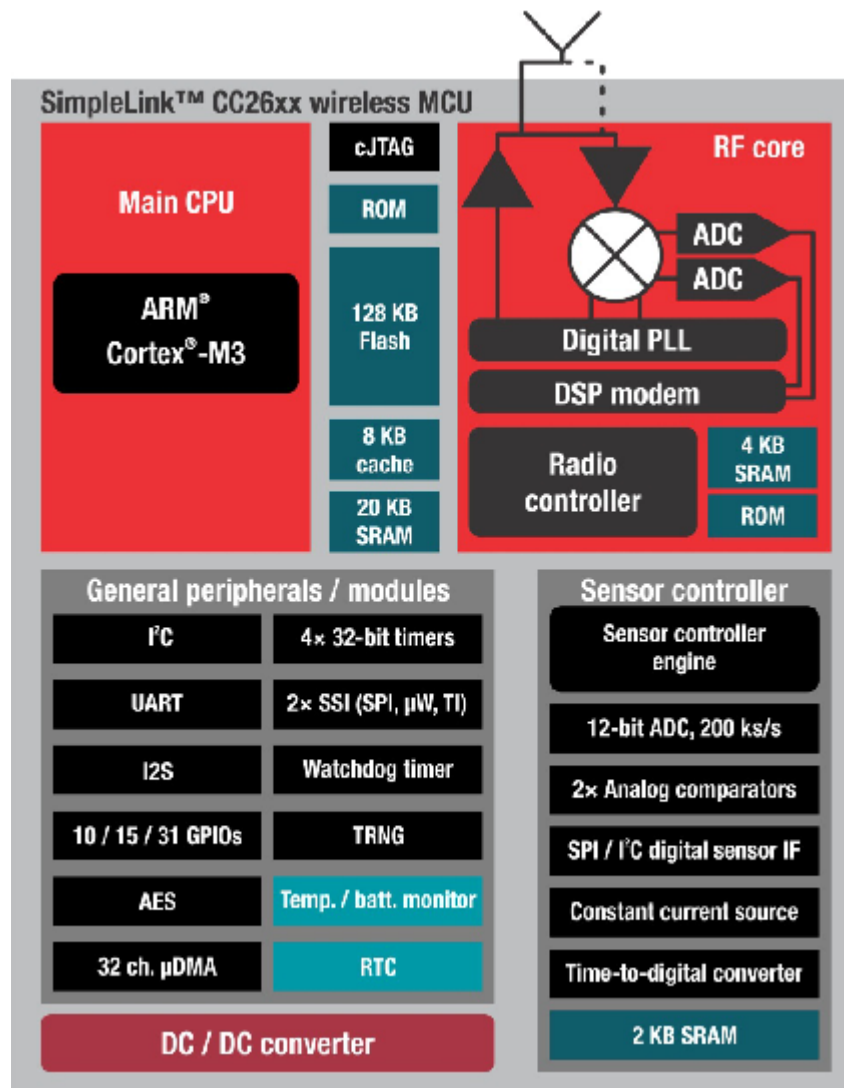


Figure 2-1. SimpleLink CC2640 Block Diagram

2.1 Hardware and Software Architecture Overview

This section aims to introduce the different cores within the CC2640, how they interact, and the firmware that runs on them. The information presented here should be considered as an overview. For detailed descriptions of the hardware described here, refer to the chapter 23 of the *CC26xx Technical Reference Manual (TRM)* ([SWCU117](#)).

2.1.1 ARM Cortex M0 (Radio Core)

The Cortex M0 (CM0) core within the CC2640 is responsible for both interfacing to the radio hardware, and translating complex instructions from the Cortex M3 (CM3) core into bits that are sent over the air using the radio. For the Bluetooth low energy protocol, the CM0 implements the PHY layer of the protocol stack. Often, the CM0 is able to operate autonomously, which frees up the CM3 for higher-level protocol and application-layer processing.

The CM3 communicates with the CM0 through a hardware interface called the RF doorbell, which is documented in section 23.2 of the *CC26xx Technical Reference Manual (TRM)* ([SWCU117](#)). The radio core firmware is not intended to be used or modified by the application developer.

2.1.2 ARM Cortex M3 (System Core)

The system core (CM3) is designed to run the Bluetooth low energy protocol stack from the link layer up to the user application. The link layer interfaces to the radio core through a software module called the RF driver, which sits above the RF doorbell. The RF driver runs on the CM3 and acts as an interface to the radio on the CC2640, and also manages the power domains of the radio hardware and core.

Documentation for the RF driver can be found within the TI-RTOS installation. Above the RF driver is the TI Bluetooth low energy protocol stack, which is implemented in library code.

The application developer should interface with the stack through a set of APIs (ICall) to implement an end use case. The rest of this document intends to document application development on the CC2640 using the Bluetooth low energy stack.

2.2 Protocol Stack and Application Configurations

Figure 2-2 shows the platform that supports two different protocol stack and application configurations.

- Single device:** The controller, host, profiles, and application are all implemented on the CC2640 as a true single-chip solution. This configuration is the simplest and most common when using the CC2640. This configuration is used by most of TI's sample projects. This configuration is the most cost-effective technique and provides the lowest-power performance.
- Simple network processor:** The Simple Network Processor (SNP) implements the controller and host layers of the BLE-Stack. Additionally, the SNP exposes an interface for scheduling communication between the stack and an external MCU. This accelerates dual MCU designs because the application processor (AP) is only responsible for managing custom profiles and application code. Stack-related functionality, such as security, is implemented on the SNP. The SNP currently supports the peripheral and broadcaster GAP roles. Communication with the SNP is carried out through the SNP API. The SNP API is built on the Unified Network Processor Interface (UNPI), which supports UART and SPI transport layers. For more information, reference the [Unified Network Processor Interface wiki page](#). TI also provides the SAP library, which implements a UNPI master and the SNP API. The SAP library can be ported to any TI-RTOS-capable processor, or used as a reference for developing a custom dual MCU solution. For a description of the SNP, see the [CC2640 Simple Network Processor API Guide](#).

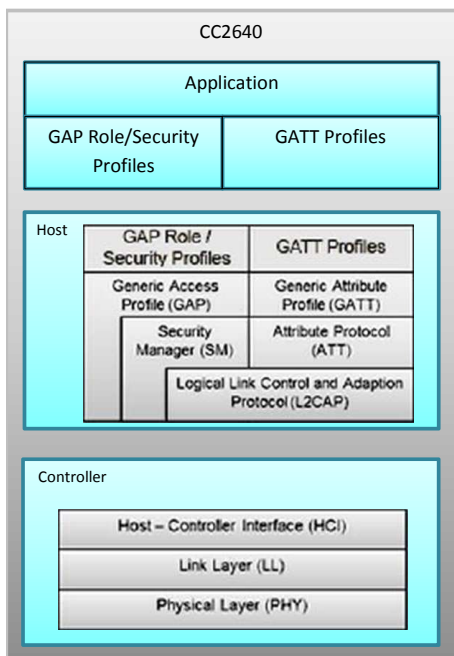


Figure 2-2. Single-Device Processor Configuration

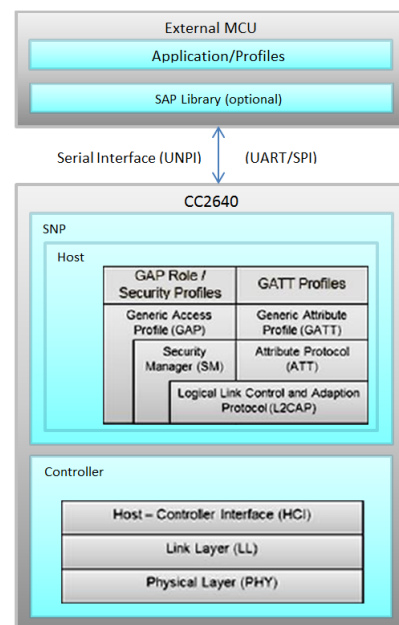


Figure 2-3. Simple Network Processor Configuration

2.3 Solution Platform

This section describes the various components that are installed with the Bluetooth low energy stack SDK and the directory structure of the protocol stack and any tools required for development. Figure 2-4 shows the *Bluetooth* low energy stack development system.

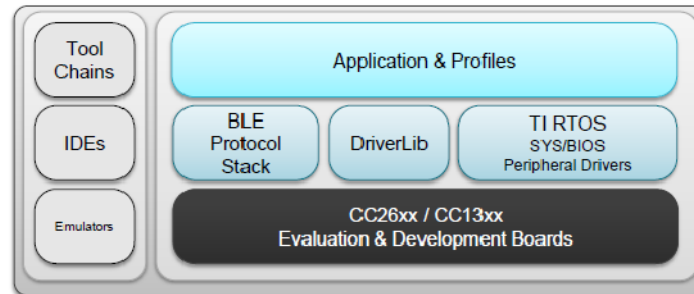


Figure 2-4. Bluetooth low energy Stack Development System

The solution platform includes the following components:

- **Real-time operating system (RTOS)** with the TI-RTOS SYS/BIOS kernel, optimized power management support, and peripheral drivers (SPI, UART, and so forth)
- **CC26xxware driverLib** provides a register abstraction layer and is used by software and drivers to control the CC2640 SoC.
- **The Bluetooth low energy protocol stack** is provided in library form with parts of the protocol stack in the CC2640 ROM.
- **Sample applications and profiles** make starting development using both proprietary and generic solutions easier. Certain applications and profiles (see Chapter 12 and <https://developer.bluetooth.org/gatt/profiles/Pages/ProfilesHome.aspx>) in the Bluetooth low energy SDK are fully qualified by Bluetooth SIG.

The following integrated development environments (IDEs) are supported:

- IAR Embedded Workbench for ARM
- Code Composer Studio™ (CCS)

Refer to the SDK release notes for the specific IDE versions supported by this release.

2.4 Directory Structure

The BLE-Stack v2.2.0 folder structure has departed from the layout used in previous releases, to make the structure easier to use and learn. If you use other TI MCU solutions, the layout should look and feel familiar. This section maps critical sections of the new SDK folder structure to the legacy structure.

The default install location is still: `C:\ti\simplelink\ble_sdk_2_02_xx_xxxx`.

The SDK installs to this location by default. For the purposes of this document, consider the above path to the BLE-Stack root directory; it will be omitted. All paths will be relative to the BLE-Stack root directory.

Opening up the root install directory shows the new parent folders within the SDK, as shown in Table 2-1.

Table 2-1. SDK Parent Folders

| SDK Folder | Pre-2.2 SDK Folder | Purpose |
|----------------|---|--|
| blelib | \Projects\ble\Libraries\CC26xx\IAR\CC2640\bin | Contains BLE-Stack code provided in library form (such as host, controller, HCI, and so forth). |
| sap_3_00_01_07 | N/A | Simple application processor (SAP) libraries and documentation for interfacing to the simple network processor (SNP) |

Table 2-1. SDK Parent Folders (continued)

| SDK Folder | Pre-2.2 SDK Folder | Purpose |
|----------------|--|---|
| examples\<kit> | \Projects\ble See Section 2.4.1 | BLE-Stack example projects, their project files, and config files organized by evaluation kit. No source files are located in this directory. |
| examples\util | \Projects\ble\util | Boot image manager (BIM) sample applications for OAD |
| src | This folder doesn't have a 1:1 mapping to a folder from the old structure. See Section 2.4.2 | All source code for the BLE-Stack, this includes example project source, header files containing library APIs, util files, ROM symbols, and so forth. |
| tools | \Projects\tools\ | Various tools used during the build process. For example, Frontier, BTool, lib_search, and so forth. |
| examples\hex | \Accessories | Prebuilt hex files for select projects |
| docs | \Documents | User guides, development guides, and so forth |

2.4.1 Examples Folder

This folder contains the files required by the supported IDEs to manage each project. Projects are organized in a two-level structure, first by evaluation board, then by project name. This is a different approach from previous stack releases. Previously, multiple configurations of a single project were created to handle supporting a project (such as `simple_peripheral`) on multiple hardware platforms. The new approach creates a separate project file for each hardware platform.

Using `simple_peripheral` on the SmartRF06 Board + CC2650EM-7ID as an example:

Old location of project file root: `\Projects\ble\simple_peripheral\CC26xx\IAR`

User first selects IAR or CCS folders as needed and imports project into their IDE of choice. The user then selects FlashROM build config to use SmartRF06. Other build configurations are available for things such as SensorTag.

New location of project file root: `\examples\cc2650em\simple_peripheral`

User first selects IAR or CCS folders as needed, then imports the project into their IDE of choice.

Because example projects are now grouped by evaluation platform, the path of the project dictates the hardware used. To switch the above project from SmartRF06 +EM to the Launchpad, load the project located at `\examples\cc2650lp\simple_peripheral`.

The general hierarchy is shown below:

- /examples
 - /eval_board1
 - supported_example1
 - supported_example2
 - /eval_board2
 - supported_example5
 - /...
 - /eval_boardN

2.4.2 Source Folder

Table 2-2. Source Folders

| SDK Folder | Pre-2.2 SDK Folder | Purpose |
|-----------------|-----------------------------|---|
| \src\common | \Projects\ble\common | Files that are shared by various examples |
| \src\components | \Components | Non-BLE-specific components used by examples (such as NPI, SBL, and so forth) |
| \src\config | \Projects\ble\config | Config files |
| \src\controller | \Components\ble\controller\ | Controller header files |
| \src\examples | \Projects\ble | Source files for example projects |
| \src\host | \Components\ble\host | Host header files |
| \src\icall | \Projects\ble\ICall | Source files for the Icall module |
| \src\inc | \Components\ble\include | Header files used to interface to the BLE stack |
| \src\profiles | \Projects\ble\Profiles | Source implementations of various BLE profiles |
| \src\rom | \Components\ble\ROM | ROM symbol file |

2.5 Sample Applications

The Bluetooth low energy stack SDK installer includes a large number of projects ranging from basic Bluetooth low energy functionality to use-case specific applications such as Heart Rate Sensor, Glucose Collector, and so forth. The following sections present the basic projects with which to begin. For more details on these and all other included projects, see [Chapter 12](#).

- The **simple_peripheral** project consists of sample code that demonstrates a simple Bluetooth low energy slave application in the single-device configuration. This project can be used as a reference for developing a slave and peripheral application.
- The **simple_central** project demonstrates the other side of the connection. This project demonstrates a simple master and central application in the single-device configuration and can be a reference for developing master and central applications.
- The **simple_broadcaster** project demonstrates the only role for implementing nonconnectable Beacon applications, such as Apple iBeacon™ and Google Eddystone™, an open beacon format from Google. See *Bluetooth low energy Beacons Application Note (SWRA475)* for more information about Bluetooth low energy Beacons.
- The **sensortag** project is a peripheral application that is configured to run on the CC2650 SensorTag reference hardware platform and communicate with the various peripheral devices of the SensorTag (for example, temperature sensor, gyro, magnetometer, and so forth).
- The **simple_np** project builds the Simple Network Processor (SNP) software for the CC2640. The SNP is the easiest way to add BLE connectivity to an existing host processor or MCU. The SNP implements the peripheral and broadcaster roles. This is the recommended starting point for dual MCU designs. Refer to the [CC2640 Simple Network Processor API Guide](#) for APIs available in the SNP implementation.
- The **simple_ap** project demonstrates an application (host) MCU using the SAP lib to communicate with the CC2640, running the SNP network processor application. See [CC2640 Simple Network Processor API Guide](#) in the Documents folder for APIs available for the SNP implementation.
- The **host_test** project builds the Bluetooth low energy HCI-based network processor software for the CC2640. This project can be configured for master and slave roles and can be controlled by the BTool PC application. See the [TI BLE Vendor-Specific HCI Reference Guide v2.2](#) in the Documents folder for APIs available for configuring and controlling the HostTest application.

2.6 Setting up the Integrated Development Environment

The IDE must be set up to browse through the relevant projects and view code. All embedded software for the CC2640 is developed using IAR Embedded Workbench for ARM (from IAR software) or CCS from TI on a Windows 7® or later PC. This section provides information on where to find this software and how to configure the workspace for each IDE.

All path and file references in this document assume that the Bluetooth low energy SDK is installed to the default path (**\$BLE_INSTALL\$**). TI recommends making a working copy of the Bluetooth low energy SDK before to making any changes. The Bluetooth low energy SDK uses relative paths and is designed to be portable, allowing the copying of the top-level directory to any valid path.

NOTE: If installing the Bluetooth low energy SDK to a nondefault path, do not exceed the maximum length of the file system namepath. Actual paths may differ from the figures.

2.6.1 Installing the SDK

To install the Bluetooth low energy stack SDK, run the ble_sdk_2_02_00_xxxx_setup.exe installer:

- xxxx is the SDK build revision number.
- The default SDK install path is C:\ti\simplelink\ble_sdk_2_02_00_xxxx. Throughout the rest of this document, this path will be referred to as \$BLE_INSTALL\$.

This installer also installs the TI-RTOS bundle, XDC tools, and the BTool PC application (if not already installed). Table 2-3 lists the software and tools are supported and tested with this Bluetooth low energy stack SDK. Newer versions of tools may not be compatible with this SDK release. Check the [TI Bluetooth LE Wiki](#) for the latest supported tool versions.

Table 2-3. Supported Tools and Software

| Tool or Software | Version | Install Path |
|---|-----------------|---------------------------------------|
| Bluetooth low energy Stack SDK Installer | 2.2.0 | C:\ti\simplelink\ble_sdk_2_02_00_xxxx |
| IAR EW ARM IDE | 7.50.3 | Windows default |
| Code Composer Studio IDE | 6.1.2 | Windows default |
| TI-RTOS | 2_18_00_03 | C:\ti\rtos_simplelink_2_18_00_03 |
| XDC Tools | 3_32_00_06_core | C:\ti\xdctools_3_32_00_06_core |
| Sensor Controller Studio | 1.2.1 | Windows default |
| BTool PC Application | 1.41.09 | \tools\btool |
| SmartRF™ Flash Programmer 2 | 1.7.3 | Windows default |
| SmartRF Studio 7 | 2.3.1 | Windows default |

2.6.2 IAR

IAR contains many features beyond the scope of this document. More information and documentation can be found at www.iar.com.

2.6.2.1 Configuring IAR Embedded Workbench for ARM

To configure the IAR Embedded Workbench for ARM do the following.

1. Download and install IAR EW ARM version 7.50.3 from <https://www.iar.com/iar-embedded-workbench/partners/texas-instruments/ti-wireless/>. (This is the official version of IAR supported and tested for this release. Opening IAR project files with a previous version of IAR may cause project file corruption.) To get IAR, choose one of the following methods:
 - Download the IAR Embedded Workbench 30-Day Evaluation Edition – This version of IAR is free, has full functionality, and includes all of the standard features. The size-limited Kickstart evaluation option is not compatible with this SDK.
 - Purchase the full-featured version of IAR Embedded Workbench – For complete BLE application development using the CC2640, TI recommends purchasing the complete version of IAR without any restrictions. You can find the information on purchasing the complete version of IAR at <https://www.iar.com/buy>.
2. Run the ti_emupack_setup.exe file in the IAR installation, <iar_install>\arm\drivers\ti-xds.

NOTE: IAR is usually installed to C:\Program Files (x86)\IAR Systems. The full, non-code size restricted version of IAR is required.

3. Select Run as Administrator when installing the emupack file.
4. Install to C:\ti (default).

NOTE: For full verbosity during building, TI recommends showing all the build output messages.

5. Navigate to Set Tools→ Options→ Messages.
6. Toggle Show Build Messages to All (see [Figure 2-5](#)).

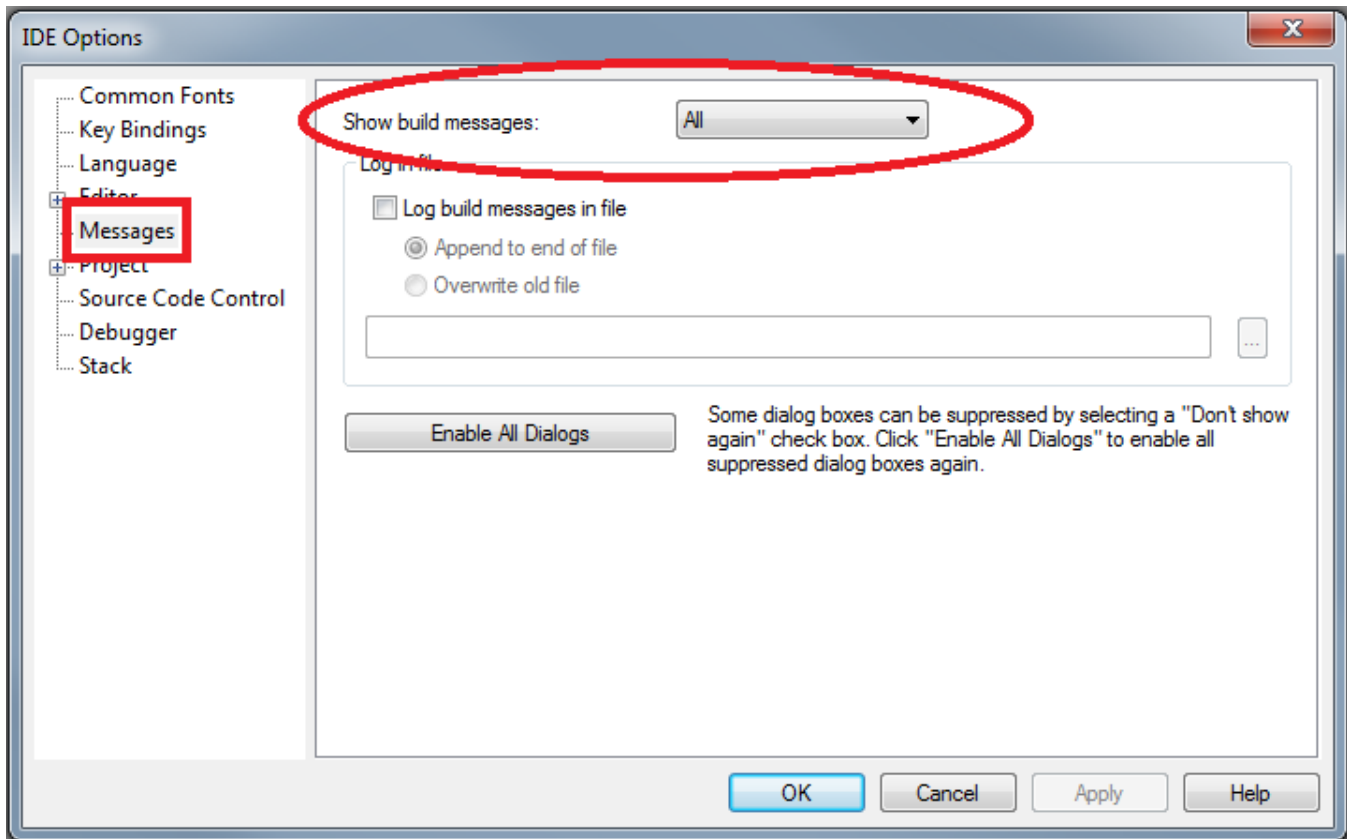


Figure 2-5. Full Verbosity

7. Navigate to Tools→ Custom Argument Variables.
8. Verify Custom Argument Variables points to the installed TI-RTOS and XDC tool paths set in the CC26xx TI-RTOS group (see [Figure 2-6](#) for the TI-RTOS and XDC default tool paths).

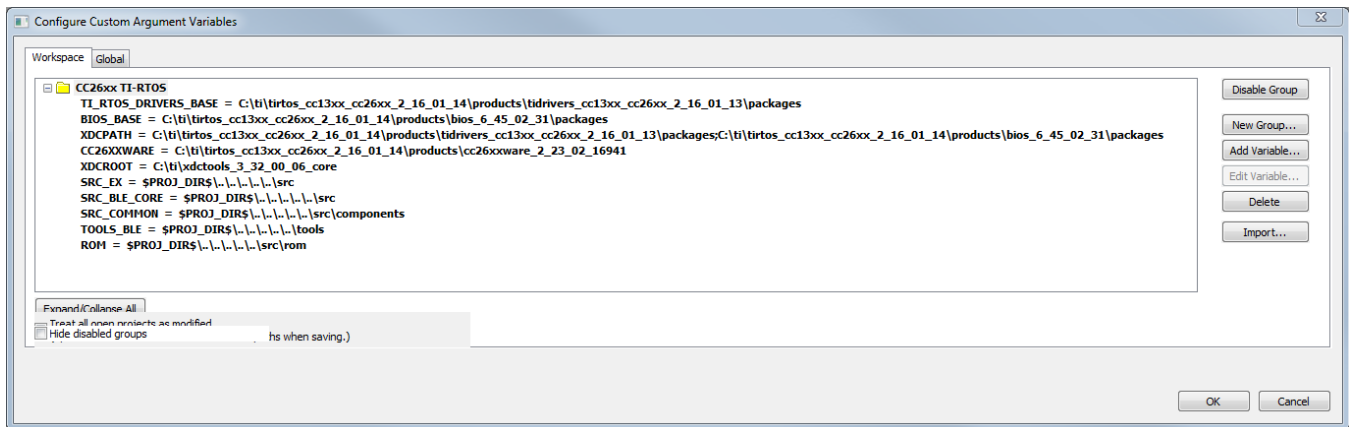


Figure 2-6. Custom Argument Variables

NOTE: If any additional argument groups on the Workspace or Global tabs are present that conflict with the CC26xx TI-RTOS group, disable the groups. If these tools are installed to a non-default location, these variables must be manually updated.

2.6.2.2 Using IAR Embedded Workbench

This section describes how to open and build an existing project.

NOTE: The simple_peripheral project is referenced throughout this document. All of the Bluetooth low energy projects included in the development kit have a similar structure.

2.6.2.2.1 Open an Existing Project

To open an existing project, do the following.

1. Open the IAR Embedded Workbench IDE from the Start Menu.
2. Navigate to File→ Open→ Workspace.
3. Select \$BLE_INSTALL\$\examples\cc2650em\simple_peripheral\iar\simple_peripheral.eww

NOTE: This workspace file is for the simple_peripheral project. When selected, the files associated with the workspace become visible in the Workspace pane on the left side of the screen. See [Figure 2-7](#).

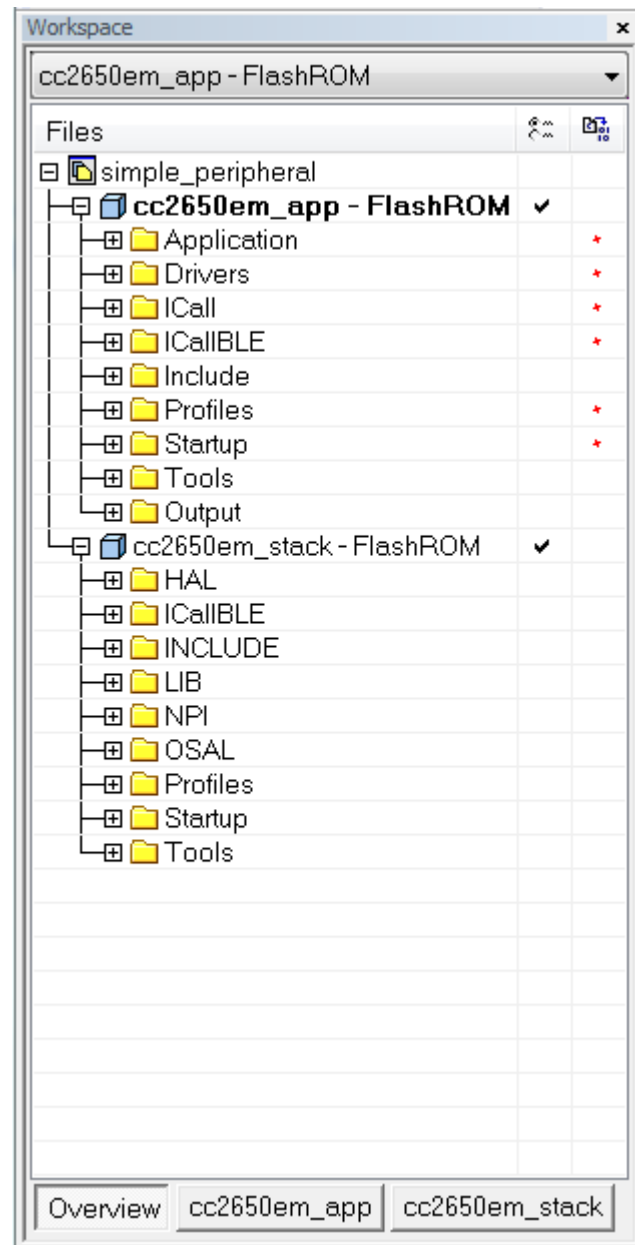


Figure 2-7. IAR Workspace Pane

This and all CC2640 project workspaces contain the following projects. The stack and application project names vary based on the evaluation board.

- The application project (cc2650em_app)
- The Bluetooth low energy protocol stack project (cc2650em_stack)

Select either project as the active project by clicking the respective tab at the bottom of the workspace pane. In [Figure 2-7](#), the Overview tab is selected. This tab displays the file structure for both projects simultaneously. In this case, use the drop-down menu at the top of the workspace pane to select the active project. Each of these projects produces a separate downloadable object. TI chose this dual-image architecture so that the application can be updated independent of the stack. The simple_peripheral sample project is the primary reference target for the description of a generic application in this guide. The simple_peripheral project implements a basic Bluetooth low energy peripheral device including a GATT server with GATT services. This project can be a framework for developing peripheral-role applications.

2.6.2.2.2 Compile and Download

NOTE:

- Do not modify the CPU Variant in the project settings. All sample projects are configured with a CPU type, and changing this setting (that is, from CC2640 to CC2650) may result in build errors.
 - All CC2640 and CC2650 code is binary-compatible and interchangeable for Bluetooth low energy-Stack software builds.
 - The CPU type is the same for all silicon package types.
-

Because the workspace is split into two projects (application and stack), the following is the specific sequence for compilation and download.

1. Select the new stack project.
2. Select Project→ Download→ Download Active Application to download the stack project.
3. Select the application project.

NOTE: The stack project defines the flash and RAM boundary parameters used by the application project. Any modifications to the stack project require a rebuild of the stack project, followed by a rebuild of the application project to use the new boundary settings. See Section [3.12 Frontier Tool].

After the initial build, if the stack project is not modified, do the following:

4. Select Project→ Make to build the application.
 5. Select Project→ Download and Debug to download the application.
-

NOTE: When the application is downloaded (that is, flash memory programmed), you can use Project→ Debug without Downloading.

Sample applications that implement the Over the Air Download (OAD) firmware update capability require the Boot Image Manager (BIM) project to be built. Refer to the CC2640 BLE OAD User's Guide for more details.

2.6.3 Code Composer Studio

CCS contains many features beyond the scope of this guide. For more information and documentation, see <http://www.ti.com/tool/CCSTUDIO>.

Check the Bluetooth low energy SDK release notes to see which CCS version to use and any required workarounds. Object code produced by CCS may differ in size and performance as compared to IAR produced object code.

2.6.3.1 Configure CCS

The following procedure describes installing and configuring the correct version of CCS and the necessary tools.

1. Download version 6.1.2 (or later) of CCS with TI ARM Compiler 5.2.6+ from http://processors.wiki.ti.com/index.php/Download_CCS. If the required TI ARM Compiler, as specified in the release notes, is not installed by default, refer to [Section 2.6.3.2](#) for the procedure to install the required TI ARM Compiler version.
2. Launch setup_ccs_win32.exe.
3. Select Processor Support→ Simplelink Wireless MCUs→ CC26xx Device Support and TI ARM Compiler.
4. Select Next to complete the installation.
5. After installation completes, select Help→ About Code Composer Studio to verify the installation

details (see [Figure 2-8](#) and [Figure 2-9](#)).

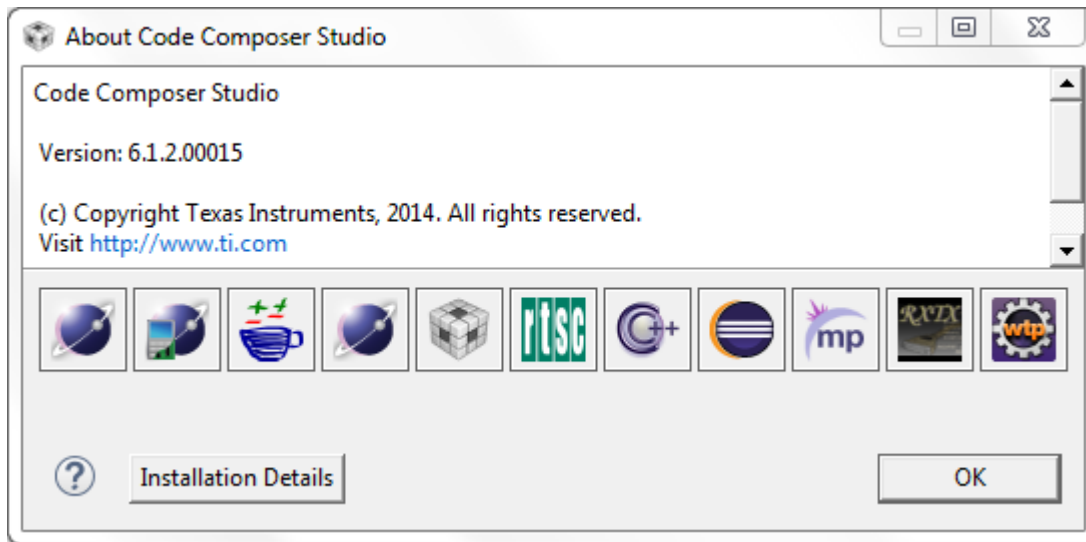


Figure 2-8. Installation Details

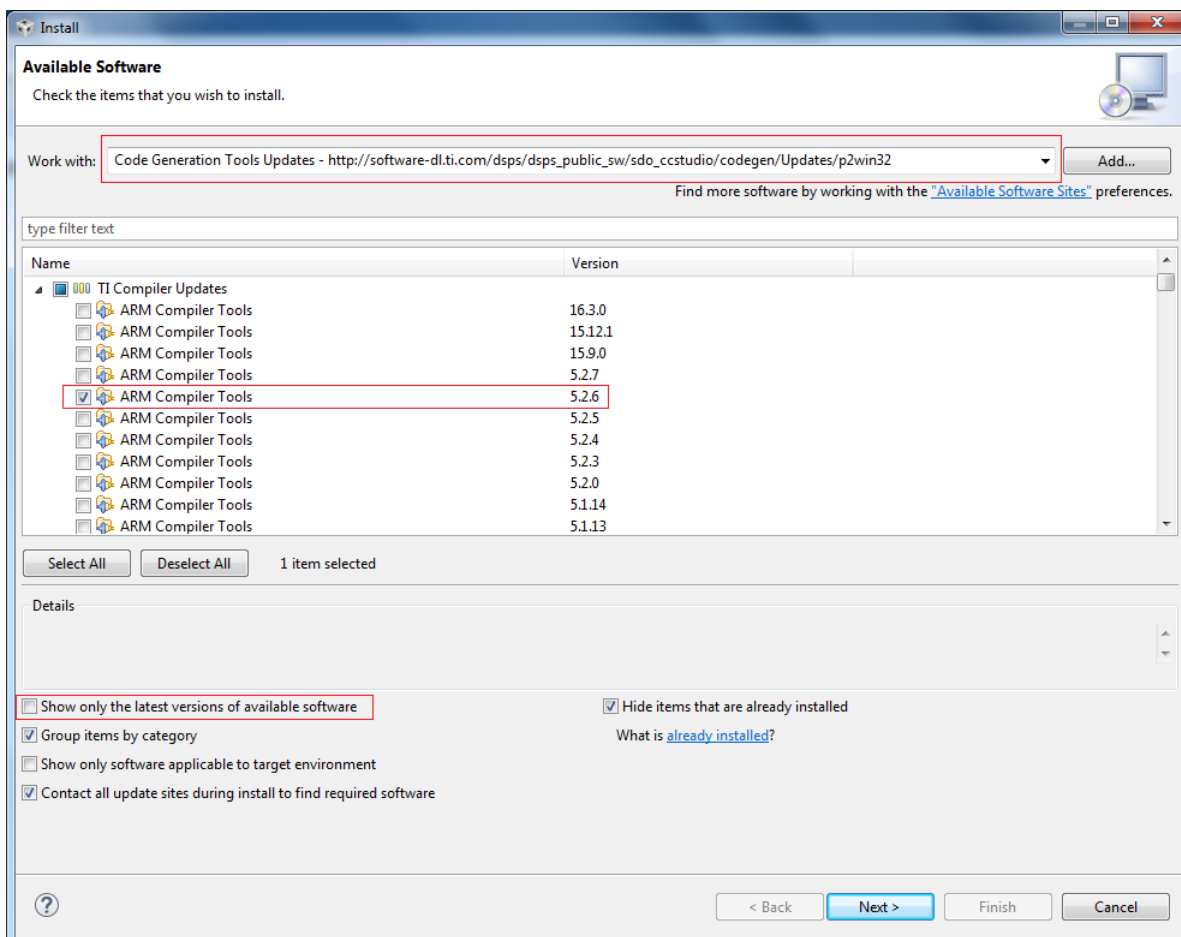


Figure 2-9. Installation Details

- Verify that the ARM Compiler Tools is version 5.2.6 (or later) and that CC26xx Device Support is version 1.16.2.00 (or later). Use the Help -> Check for Updates menu to check for updated

components. Refer to [Section 2.6.3.2](#) to install a specific TI ARM Compiler version.

2.6.3.2 Installing a Specific TI ARM Compiler

To install a specific TI ARM Compiler, refer to the following steps and [Figure 2-9](#)

1. Help -> Install New Software
2. Select Code Generation Tools Update from the Works With drop down menu
3. Expand the TI Compiler Update.
4. Uncheck "Show only the latest versions of software"
5. Select the desired ARM Compiler Tools version.
6. Press Next to complete the installation.

2.6.3.3 Using CCS

This section describes how to open and build an existing project and references the simple_peripheral project. All of the CCS Bluetooth low energy projects included in the development kit have a similar structure.

2.6.3.3.1 Import an Existing Project

To import an existing project, do the following.

1. Open the CCS IDE from the Start Menu.
2. Select Project→ Import CCS Project.
3. Select `$BLE_INSTALL$\examples\cc2650em\simple_peripheral\ccs`.

NOTE: This is the CCS directory for the simple_peripheral project. CCS discovers two projects (the application project and the stack project).

4. Click the box next to an application project (depending on your development platform) and the stack project to select them.
5. Select Copy projects into workspace.

6. Click Finish to import (see [Figure 2-10](#)).

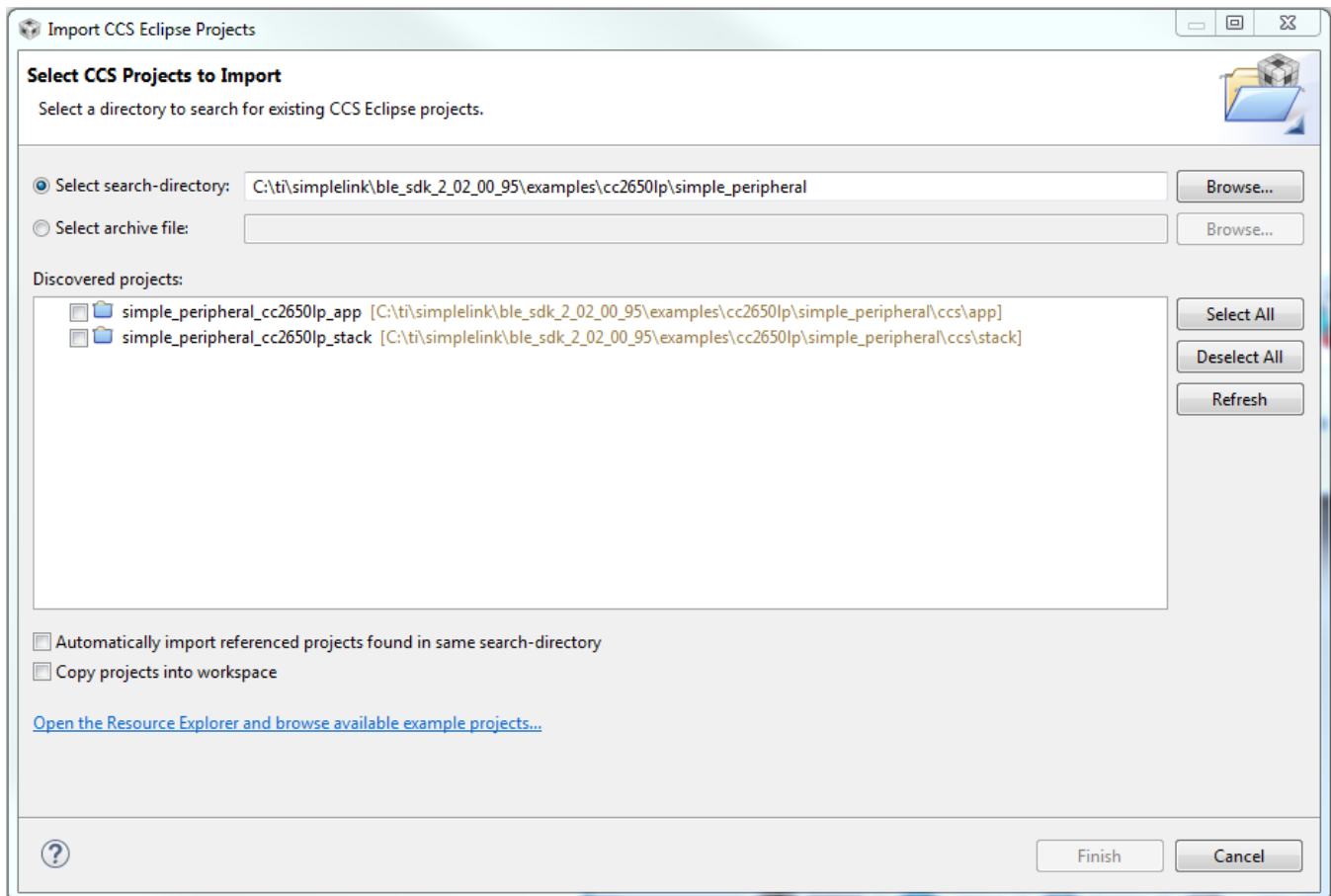


Figure 2-10. Import CCS Projects

2.6.3.3.2 Workspace Overview

This workspace and all CC2640 project workspaces contain the following projects:

- The application project (simple_peripheral_cc2650p_app)
- The stack project (simple_peripheral_cc2650p_stack)

Click the project name in the file explorer to select the project as the active project. In [Figure 2-11](#), the application is selected as the active project. Each of these projects produces a separate, downloadable image. TI chose this dual-image architecture so that the application can be updated independent of the stack.

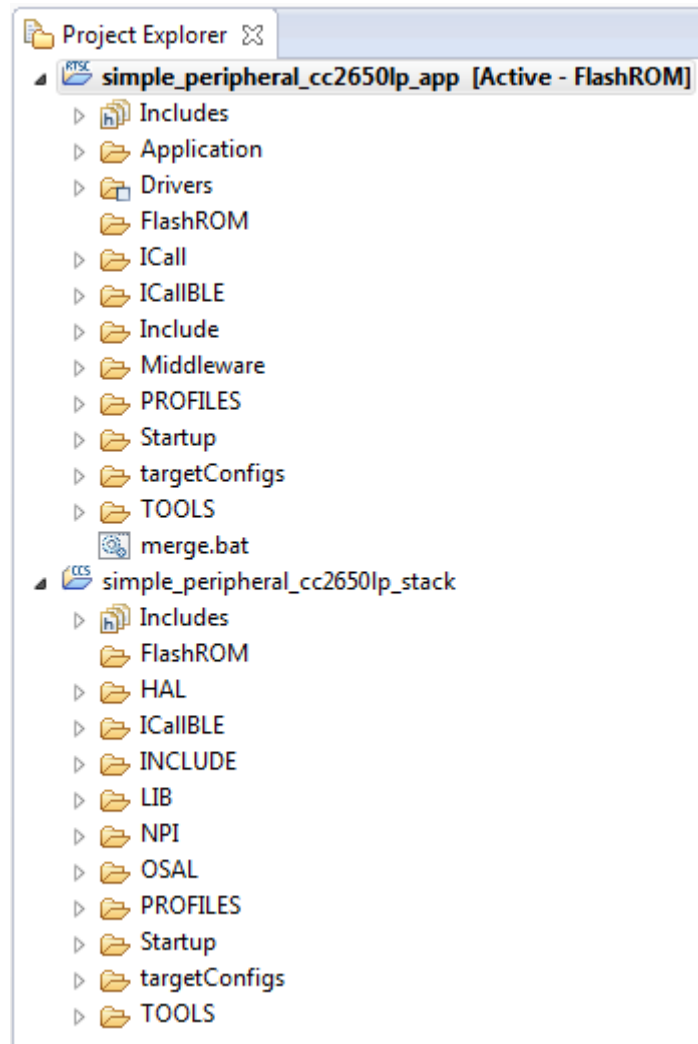


Figure 2-11. Project Explorer Structure

The `simple_peripheral` sample project is the primary sample application for the description of a generic application in this guide. The `simple_peripheral` project implements a basic Bluetooth low energy peripheral device including a GATT server with GATT services. This project can be used as a framework for developing peripheral-role applications.

2.6.3.3.3 Compiling and Downloading

NOTE:

- Do not modify the CPU Variant in the project settings.
- All sample projects are configured with a CPU type and changing this setting (that is, from CC2640 to CC2650) may result in build errors.
- All CC2640/CC2650 code is binary compatible and interchangeable for Bluetooth low energy software stack builds.
- The CPU type is the same for all silicon package types.

Because the workspace is split into two projects (application and stack), the following is the specific sequence for compilation and download.

1. Set the stack project as the active project.

2. Select Project→ Build All to build the stack project.
3. Set the application project as the active project.
4. Select Project→ Build All to build the application project.
5. Select the stack project as the active project.
6. Select Run→ Debug to download the stack.
7. Select the application project as the active project.
8. Select Run→ Debug to download the application.

NOTE: The stack project defines flash and RAM boundary parameters used by the application project. Any modifications to the stack project will require a rebuild of the stack project followed by a rebuild of the application project to use the new boundary settings. See [Section 3.12](#).

After the initial build and download, if the stack project is not modified, do the following:

1. Build the application.
2. Download the application.

Sample applications that implement the Over the Air Download (OAD) firmware update capability require the Boot Image Manager (BIM) project to be built. Refer to the CC2640 BLE OAD User's Guide for more details.

2.7 Working With Hex Files

TI configured the application and stack projects to produce an Intel®-extended hex file in their respective output folders. These hex files lack overlapping memory regions and can be programmed individually with a flash programming tool, such as SmartRF Flash Programmer 2. You can combine the application and stack hex files into a super hex file manually or using free tools. For information on the Intel Hex standard, refer to https://en.wikipedia.org/wiki/Intel_HEX.

One example for creating the super hex file is with the IntelHex python script `hex_merge.py`, available at <https://launchpad.net/intelhex/+download>. To merge the hex files, install Python® 2.7.x and add it to your system path environment variables.

The following is an example usage to create a merged `simple_peripheral_merged.hex` file consisting of the individual application and stack hex files:

```
C:\Python27\Scripts>python hexmerge.py -o .\simple_peripheral_merged.hex -r 0000:1FFFF
simple_peripheral_cc2650em_app.hex:0000:1FFFF simple_peripheral_cc2650em_stack.hex --
overlap=error
```

2.8 Accessing Preprocessor Symbols

Various C preprocessor symbols may need to be set or adjusted at the project level. The procedure to access the preprocessor symbols (predefined symbols) is based on the IDE being used. The following procedure describes how to access and modify preprocessor symbols using IAR and CCS.

NOTE: In IAR:

1. Open the Project Options for either project in the C/C++ Compiler Category.
2. Open the Preprocessor tab.
3. View the Defined symbols box (see [Figure 2-12](#)).

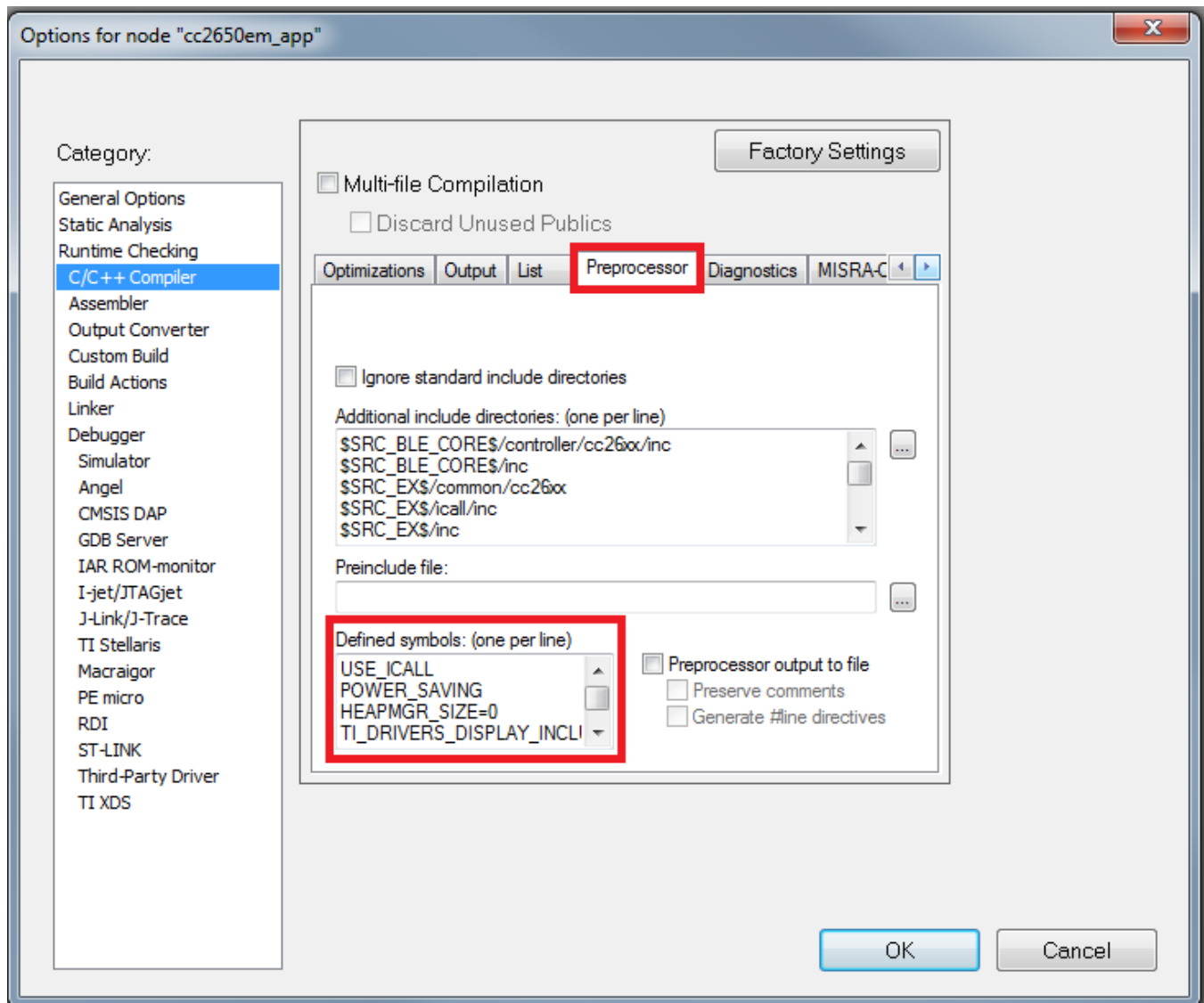


Figure 2-12. IAR Defined Symbols Box

4. Add or edit the preprocessor symbols.

In CCS, access preprocessor symbols by doing the following.

1. Open the Project Properties for either project.
2. Navigate to CCS Build→ ARM Compiler→ Advanced Options→ Predefined Symbols.
3. Use the buttons highlighted in [Figure 2-13](#) to add, delete, or edit a preprocessor.

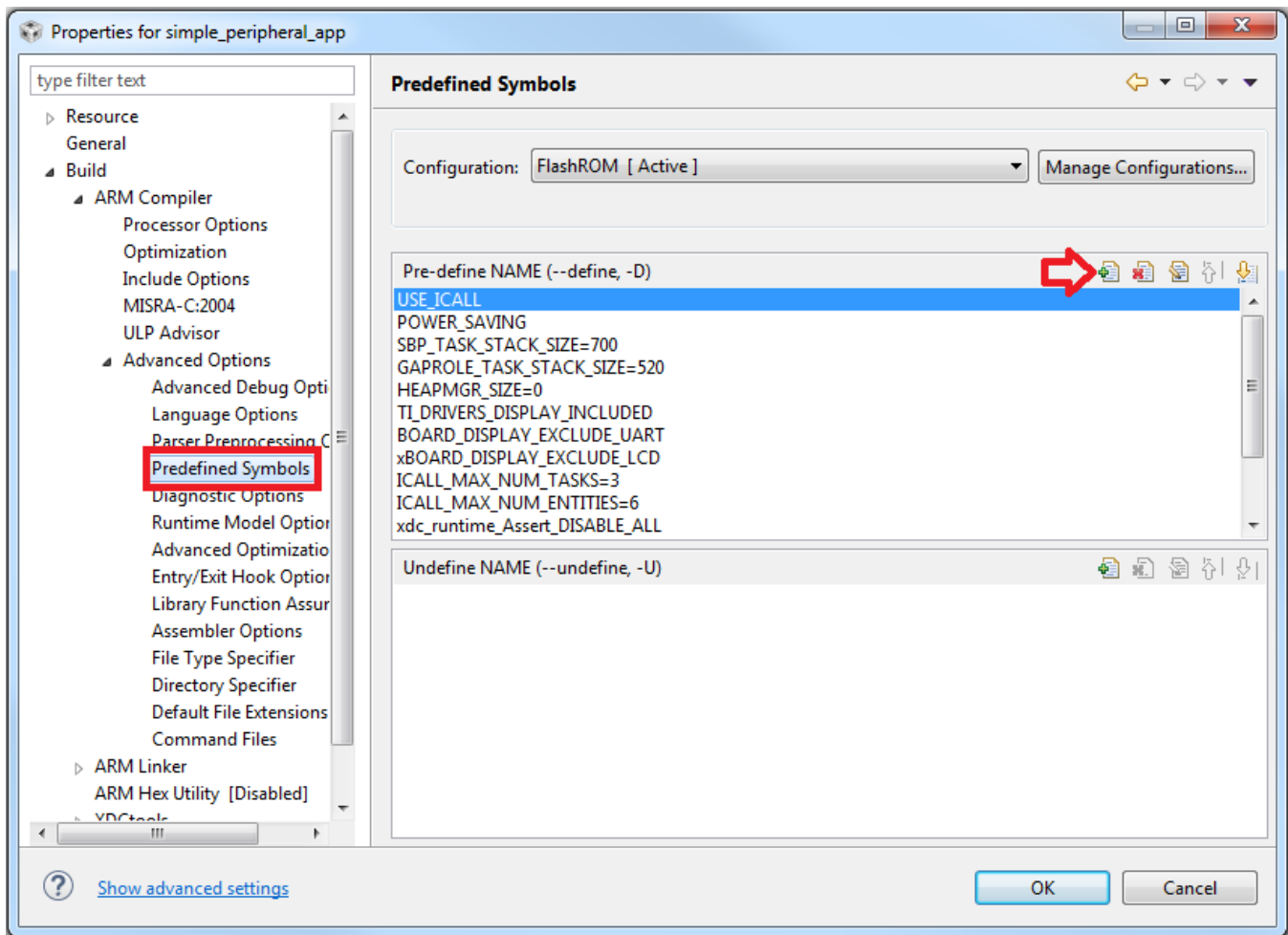


Figure 2-13. CCS Predefined Symbols

2.9 Top-Level Software Architecture

The CC2640 Bluetooth low energy software environment consists of the following parts:

- An RTOS
- An application image
- A stack image

The TI-RTOS is a real-time, pre-emptive, multithreaded operating system that runs the software solution with task synchronization. Both the application and Bluetooth low energy protocol stack exist as separate tasks within the RTOS. The Bluetooth low energy protocol stack has the highest priority. A messaging framework called indirect call (ICall) is used for thread-safe synchronization between the application and stack. [Figure 2-14](#) shows the architecture.

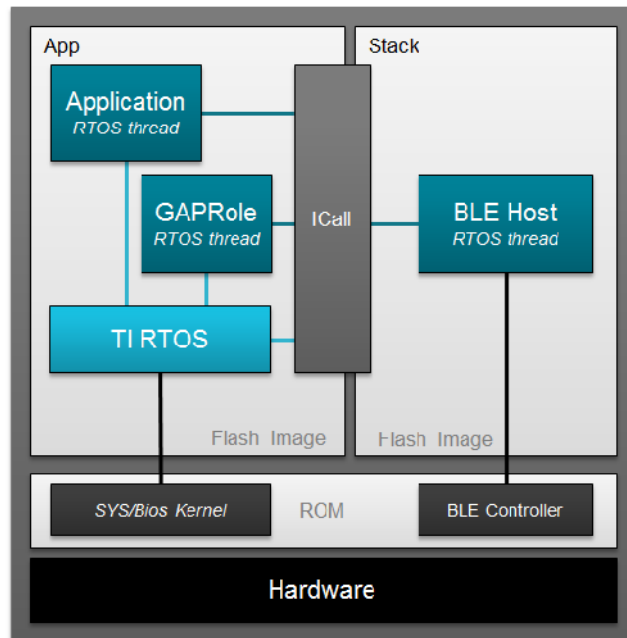


Figure 2-14. Top-Level Software Architecture

- The stack image includes the lower layers of the Bluetooth low energy protocol stack from the LL up to and including the GAP and GATT layers (see Figure 1-1). Most of the Bluetooth low energy protocol stack code is provided as a library.
- The application image includes the profiles, application code, drivers, and the ICall module.

2.9.1 Standard Project Task Hierarchy

Considering the simple_peripheral project as an example, these tasks are listed by priority. A higher task number corresponds to a higher priority task:

- 5: Bluetooth low energy protocol stack task
- 3: GapRole task (peripheral role)
- 1: Application task (simple_peripheral)

Section 3.3 introduces RTOS tasks. Chapter 5 describes interfacing with the Bluetooth low energy protocol stack. Section 5.2 describes the GapRole task. Section 4.2.1 describes the application task.

RTOS Overview

TI-RTOS is the operating environment for Bluetooth low energy projects on CC2640 devices. The TI-RTOS kernel is a tailored version of the SYS/BIOS kernel and operates as a real-time, pre-emptive, multithreaded operating system with tools for synchronization and scheduling (XDCTools). The SYS/BIOS kernel manages four distinct levels of execution threads (see [Figure 3-1](#)).

- Hardware interrupt service routines (ISRs)
- Software interrupt routines
- Tasks
- Background idle functions

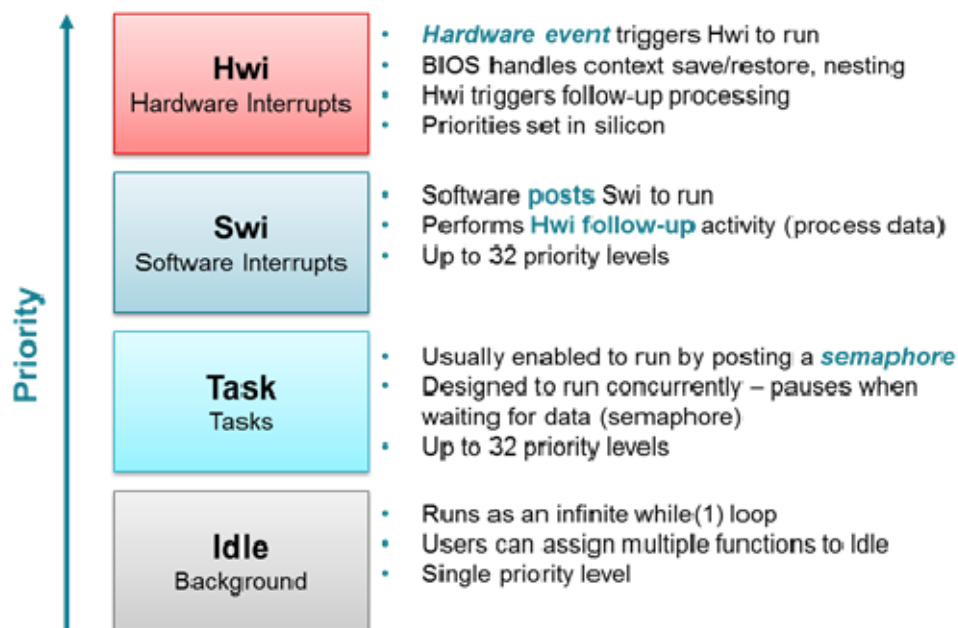


Figure 3-1. RTOS Execution Threads

This section describes these four execution threads and various structures used throughout the RTOS for messaging and synchronization. In most cases, the underlying RTOS functions have been abstracted to higher-level functions in the util.c file. The lower-level RTOS functions are described in the SYS/BIOS module section of the [TI SYS/BIOS API Guide](#). This document also defines the packages and modules included with the TI-RTOS.

3.1 RTOS Configuration

The SYS/BIOS kernel provided with the installer can be modified using the RTOS configuration file (that is, appBLE.cfg for the simple_peripheral project). In the IAR project, this file is in the application project workspace TOOLS folder. This file defines the various SYS/BIOS and XDCTools modules in the RTOS compilation, as well as system parameters such as exception handlers and timer-tick speed. The RTOS must then be recompiled for these changes to take effect by recompiling the project.

The default project configuration is to use elements of the RTOS from the CC26xx ROM. In this case, some RTOS features are unavailable. If any ROM-unsupported features are added to the RTOS configuration file, use an RTOS in flash configuration. Using RTOS in flash consumes additional flash memory. The default RTOS configuration supports all features required by the respective example projects in the SDK.

See the TI-RTOS documentation for a full description of configuration options.

NOTE: If the RTOS configuration file is changed, do the following.

1. Delete the configPkg folder to force a rebuild of the RTOS.
For example:
\$PROJ_DIR\$\CC26xx\IAR\Application\CC2650\ConfigPkg
2. Select Project→ Rebuild All to rebuild the application project and build the RTOS. (The RTOS library is compiled as part of the Pre-Build phase of the Application Project.)

3.2 Semaphores

The kernel package provides several modules for synchronizing tasks such as the semaphore. Semaphores are the prime source of synchronization in the CC2640 software and are used to coordinate access to a shared resource among a set of competing tasks (that is, the application and Bluetooth low energy stack). Semaphores are used for task synchronization and mutual exclusion.

Figure 3-2 shows the semaphore functionality. Semaphores can be counting semaphores or binary semaphores. Counting semaphores keep track of the number of times the semaphore is posted with Semaphore_post(). When a group of resources are shared between tasks, this function is useful. Such tasks might call Semaphore_pend() to see if a resource is available before using one. Binary semaphores can have only two states: available (count = 1) and unavailable (count = 0). Binary semaphores can be used to share a single resource between tasks or for a basic-signaling mechanism where the semaphore can be posted multiple times. Binary semaphores do not keep track of the count; they track only whether the semaphore has been posted.

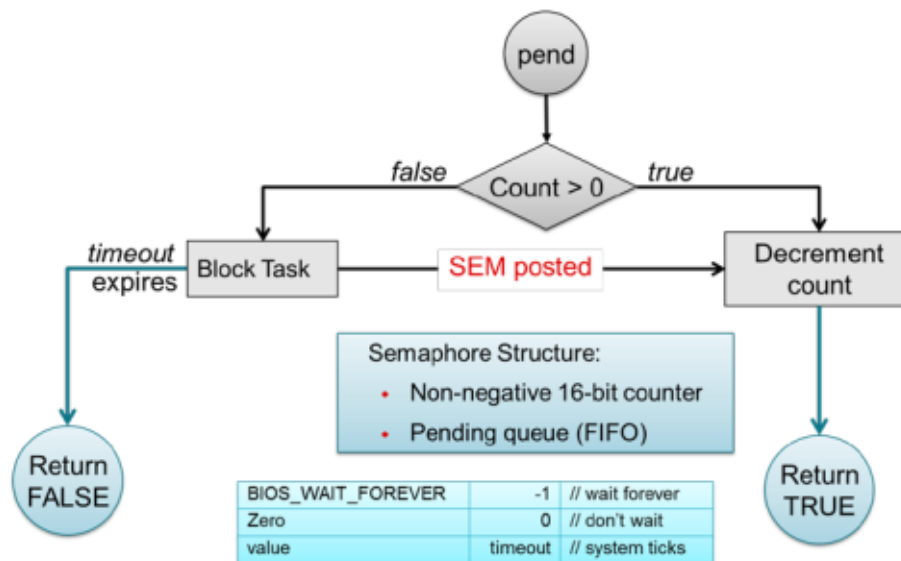


Figure 3-2. Semaphore Functionality

3.2.1 Initializing a Semaphore

The following code depicts how a semaphore is initialized in RTOS. An example of this in the `simple_peripheral` project is when a task is registered with the ICall module: `ICall_registerApp()` which eventually calls `ICall_primRegisterApp()`. These semaphores coordinate task processing. [Section 4.2](#) describes this coordination further.

```
Semaphore_Handle sem;
sem = Semaphore_create(0, NULL, NULL);
```

3.2.2 Pending a Semaphore

`Semaphore_pend()` is a blocking call that allows another task to run while waiting for a semaphore. The time-out parameter allows the task to wait until a time-out, wait indefinitely, or not wait at all. The return value indicates if the semaphore was signaled successfully.

```
Semaphore_pend(sem, timeout);
```

3.2.3 Posting a Semaphore

`Semaphore_post()` signals a semaphore. If a task is waiting for the semaphore, this call removes the task from the semaphore queue and puts it on the ready queue. If no tasks are waiting, `Semaphore_post()` increments the semaphore count and returns. For a binary semaphore, the count is always set to 1.

```
Semaphore_post(sem);
```

3.3 RTOS Tasks

RTOS *tasks* are equivalent to independent threads that conceptually execute functions in parallel within a single C program. In reality, switching the processor from one task to another helps achieve concurrency.

Each task is always in one of the following modes of execution:

- Running: task is currently running
- Ready: task is scheduled for execution
- Blocked: task is suspended from execution
- Terminated: task is terminated from execution
- Inactive: task is on inactive list

One (and only one) task is always running, even if only the idle task (see [Figure 3-1](#)). The current running task can be blocked from execution by calling certain task module functions, as well as functions provided by other modules like semaphores. The current task can also terminate itself. In either case, the processor is switched to the highest priority task that is ready to run. See the task module in the package `ti.sysbios.knl` section of the [TI SYS/BIOS API Guide](#) for more information on these functions.

Numeric priorities are assigned to tasks, and multiple tasks can have the same priority. Tasks are readied to execute by highest to lowest priority level; tasks of the same priority are scheduled in order of arrival. The priority of the currently running task is never lower than the priority of any ready task. The running task is preempted and rescheduled to execute when there is a ready task of higher priority. In the `simple_peripheral` application, the Bluetooth low energy protocol stack task is given the highest priority (5) and the application task is given the lowest priority (1).

Each RTOS task has an initialization function, an event processor, and one or more callback functions.

3.3.1 Creating a Task

When a task is created, it has its own runtime stack for storing local variables as well as further nesting of function calls. All tasks executing within a single program share a common set of global variables, accessed according to the standard rules of scope for C functions. This set of memory is the context of the task. The following is an example of the application task being created in the simple_peripheral project.

```
void SimpleBLEPeripheral_createTask(void)
{
    Task_Params taskParams;

    // Configure task
    Task_Params_init(&taskParams);
    taskParams.stack = sbpTaskStack;
    taskParams.stackSize = SBP_TASK_STACK_SIZE;
    taskParams.priority = SBP_TASK_PRIORITY;

    Task_construct(&sbpTask, SimpleBLEPeripheral_taskFxn, &taskParams, NULL);
}
```

The task creation is done in the main() function, before the SYS/BIOS scheduler is started by BIOS_start(). The task executes at its assigned priority level after the scheduler is started.

TI recommends using the existing application task for application-specific processing. When adding an additional task to the application project, the priority of the task must be assigned a priority within the RTOS priority-level range, defined in the appBLE.cfg RTOS configuration file.

```
/* Reduce number of Task priority levels to save RAM */
Task.numPriorities = 6;
```

Do not add a task with a priority equal to or higher than the Bluetooth low energy protocol stack task and related supporting tasks (for example, the GapRole task). See [Section 2.9.1](#) for details on the system task hierarchy.

Ensure the task has a minimum task stack size of 512 bytes of predefined memory. At a minimum, each stack must be large enough to handle normal subroutine calls and one task preemption context. A task preemption context is the context that is saved when one task preempts another as a result of an interrupt thread readying a higher priority task. Using the TI-RTOS profiling tools of the IDE, the task can be analyzed to determine the peak task stack usage.

NOTE: The term *created* describes the instantiation of a task. The actual TI-RTOS method is to construct the task. See [Section 3.11.5](#) for details on constructing RTOS objects.

3.3.2 Creating the Task Function

When a task is constructed, a function pointer to a task function (for example, `simple_peripheral_taskFxn`) is passed to the `Task_Construct` function. When the task first gets a chance to process, this is the function which the RTOS runs. Figure 3-3 shows the general topology of this task function.

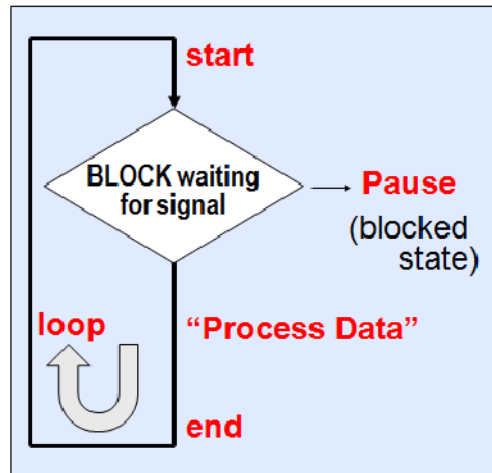


Figure 3-3. General Task Topology

In the `simple_peripheral` task, the task spends most of its time in the blocked state, where it is pending a semaphore. When the semaphore of the task is posted to from an ISR, callback function, queue, and so forth, the task becomes ready, processes the data, and returns to this paused state. See Section 4.2.1 for more detail on the functionality of the `simple_peripheral` task.

3.4 Clocks

Clock instances are functions that can be scheduled to run after a certain number of clock ticks. Clock instances are either one-shot or periodic. These instances start immediately upon creation, are configured to start after a delay, and can be stopped at any time. All clock instances are executed when they expire in the context of a software interrupt. The following example shows the minimum resolution is the RTOS clock tick period set in the RTOS configuration.

```
/* 10 us tick period */
Clock.tickPeriod = 10;
```

Each tick, which is derived from the RTC, launches a clock SWI that compares the running tick count with the period of each clock to determine if the associated function should run. For higher-resolution timers, TI recommends using a 16-bit hardware timer channel or the sensor controller.

3.4.1 API

You can use the RTOS clock module functions directly (see the clock module in the SYS/BIOS API 0). For usability, these functions have been extracted to the following functions in util.c.

Clock_Handle Util_constructClock (Clock_Struct *pClock, Clock_FuncPtr clockCB, uint32_t clockDuration, uint32_t clockPeriod, uint8_t startFlag, UArg arg)

Initialize a TIRTOS Clock instance.

Parameters

- pClock – pointer to clock instance structure
- clockCB – function to be called upon clock expiration
- clockDuration – length of first expiration period
- clockPeriod – length of subsequent expiration periods. If set to 0, clock is a one-shot clock.
- startFlag – TRUE to start immediately, FALSE to wait. If FALSE, Util_startClock() must be called later.
- arg – argument passed to callback function

Returns

Handle to the Clock instance

void Util_startClock(Clock_Struct *pClock)

Start an (already constructed) clock.

Parameters

pClock – pointer to clock structure

bool Util_isActive(Clock_Struct *pClock)

Determine if a clock is currently running.

Parameters

pClock – pointer to clock structure

Returns

TRUE: clock is running.

FALSE: clock is not running.

void Util_stopClock(Clock_Struct *pClock)

Stop a clock.

Parameters:

pClock – pointer to clock structure

3.4.2 Functional Example

The following example from the simple_peripheral project details the creation of a clock instance and how to handle the expiration of the instance.

1. Define the clock handler function to service the clock expiration SWI.

simple_peripheral.c:

```
//clock handler function
static void SimpleBLEPeripheral_clockHandler(UArg arg)
{
    //Store the event.
    events |= arg;

    // Wake up the application.
}
```

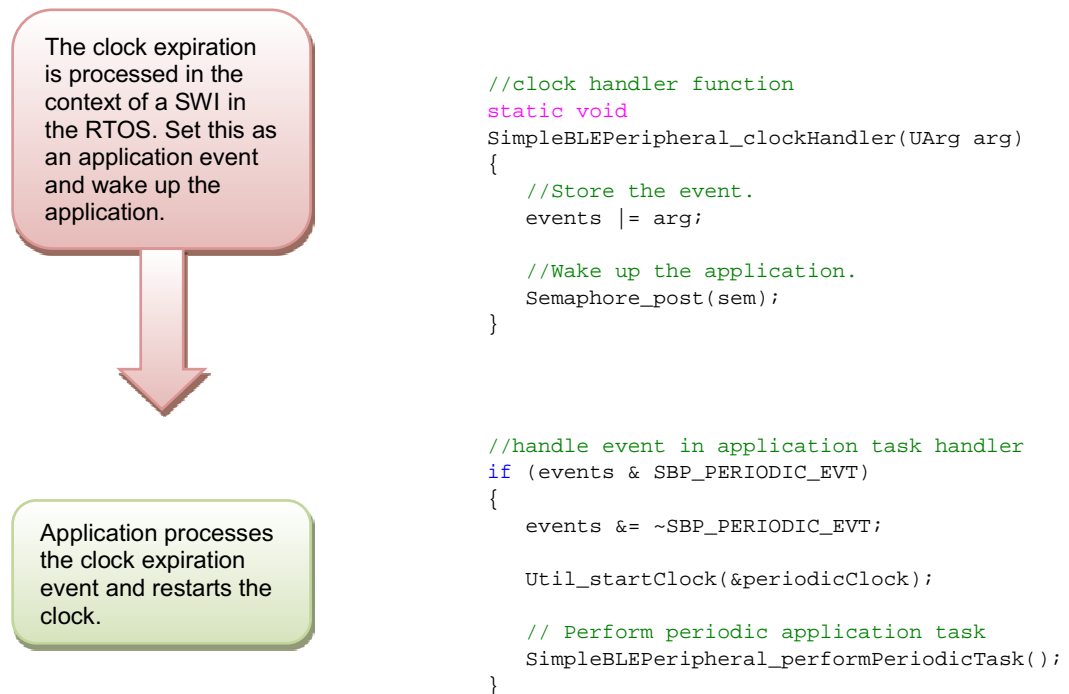
2. Create the clock instance.

simple_peripheral.c:

```
// Clock instances for internal periodic events.
static Clock_Struct periodClock;

//Create one-shot clocks for internal periodic events.
Util_constructClock(&periodicClock, SimpleBLEPeripheral_clockHandler,
                  SBP_PERIODIC_EVT_PERIOD, 0, false, SBP_PERIODIC_EVT);
```

3. Wait for the clock instance to expire and process in the application context (in the following flow diagram, green corresponds to the processor running in the application context and red corresponds to an SWI). Do not call any driver, protocol stack APIs, or blocking RTOS functions in an SWI.



3.5 Queues

Queues let applications process events in order by providing a FIFO ordering for event processing. A project may use a queue to manage internal events coming from application profiles or another task. Clocks must be used when an event must be processed in a time-critical manner. Queues are more useful for events that must be processed in a specific order.

The Queue module provides a unidirectional method of message passing between tasks using a FIFO. In [Figure 2-14](#), a queue is configured for unidirectional communication from task A to task B. Task A pushes messages onto the queue and task B pops messages from the queue in order. [Figure 3-4](#) shows the queue messaging process.



Figure 3-4. Queue Messaging Process

3.5.1 API

The RTOS queue functions have been abstracted into functions in the `util.c` file. See the Queue module in the [TI SYS/BIOS API Guide](#) for the underlying functions. These utility functions combine the Queue module with the ability to notify the recipient task of an available message through semaphores. In CC2640 software, the semaphore used for this process is the same semaphore that the given task uses for task synchronization through `ICall`. For an example of this, see the `SimpleBLECentral_enqueueMsg()` function. Queues are commonly used to limit the processing time of application callbacks in the context of the higher level priority task. In this manner, the higher priority task pushes a message to the application queue for processing later instead of immediate processing in its own context. [Section 3.5](#) further describes this process.

Queue_Handle Util_constructQueue(Queue_Struct *pQueue)

Initialize an RTOS queue.

Parameters: pQueue – pointer to queue instance

Returns Handle to the Queue instance

uint8_t Util_enqueueMsg(Queue_Handle msgQueue, Semaphore_Handle sem, uint8_t *pMsg)

Creates a queue node and puts the node in an RTOS queue.

Parameters msgQueue – queue handle

sem – event processing semaphore of the task with which the queue is associated

pMsg – pointer to message to be queued

Returns TRUE – Message was successfully queued.

FALSE – Allocation failed and message was not queued.

uint8_t *Util_dequeueMsg(Queue_Handle msgQueue)

De-queues the message from an RTOS queue.

Parameters msgQueue – queue handle

Returns NULL: no message to dequeue

Otherwise: pointer to dequeued message

3.5.2 Functional Example

The following queue example from the simple_peripheral project follows the handling of a button press from HWI ISR to processing in the application context.

1. Define the enqueue function of the task so that it uses the semaphore to wake.

```
static uint8_t SimpleBLECentral_enqueueMsg(uint8_t event, uint8_t status, uint8_t *pData)
{
    sbcEvt_t *pMsg;

    // Create dynamic pointer to message.
    if (pMsg = ICall_malloc(sizeof(sbcEvt_t)))
    {
        pMsg->event = event;
        pMsg->status = status;
        pMsg->pData = pData;

        //Enqueue the message.
        return Util_enqueueMsg(appMsgQueue, sem, (uint8_t *)pMsg);
    }

    return FALSE;
}
```

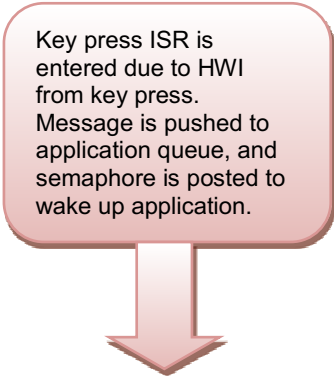
2. Statically allocate and then construct queue.

```
// Queue object used for app messages
static Queue_Struct appMsg;
static Queue_Handle appMsgQueue;

...

// Create an RTOS queue for messages to be sent to app.
appMsgQueue = Util_constructQueue(&appMsg);
```

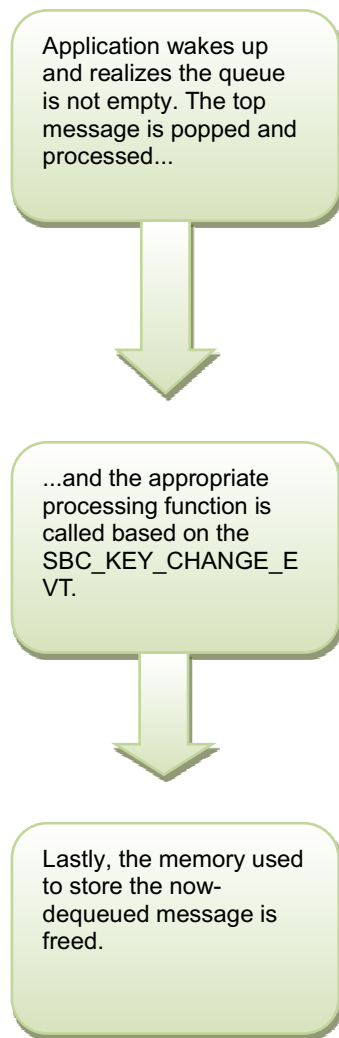
3. Wait for button to be pressed and processed in application context (in the following diagram, green corresponds to the processor running in the application context and red corresponds to an HWI).



Key press ISR is entered due to HWI from key press. Message is pushed to application queue, and semaphore is posted to wake up application.

```
void SimpleBLECentral_keyChangeHandler(uint8
keys)
{

SimpleBLECentral_enqueueMsg(SBC_KEY_CHANGE_EVT,
keys, NULL);
}
```



```

// If RTOS queue is not empty, process app
message
if (!Queue_empty(appMsgQueue))
{
    sbvEvt_t *pMsg =
    (sbvEvt_t*)Util_dequeueMsg(appMsgQueue);
    if (pMsg)
    {
        // Process message
        SimpleBLECentral_processAppMsg(pMsg);
        ...
    }
}

static void
SimpleBLECentral_processAppMsg(sbvEvt_t *pMsg)
{
    switch (pMsg->event)
    {
        ...
        case SBC_KEY_CHANGE_EVT:
            SimpleBLECentral_handleKeys(0, pMsg->status);
            break;
        ...
    }
}

...
// Free the space from the message
ICall_free(pMsg);
}
  
```

3.6 Idle Task

The Idle module specifies a list of functions to be called when no other tasks are running in the system. In the CC2640 software, the idle task runs the Power Policy Manager.

3.7 Power Management

All power-management functionality is handled by the peripheral drivers and the Bluetooth low energy protocol stack. This feature can be enabled or disabled by including or excluding the `POWER_SAVING` preprocessor-defined symbol. When `POWER_SAVING` is enabled, the device enters and exits sleep as required for Bluetooth low energy events, peripheral events, application timers, and so forth. When `POWER_SAVING` is undefined, the device stays awake. See [Section 9.2](#) for steps to modify preprocessor-defined symbols.

More information on power-management functionality, including the API and a sample use case for a custom UART driver, can be found in the [TI-RTOS Power Management for CC26xx](#) included in the RTOS install. These APIs are required only when using a custom driver.

Also see *Measuring Bluetooth Smart Power Consumption (SWRA478)* for steps to analyze the system power consumption and battery life.

3.8 Hardware Interrupts

Hardware interrupts (HWIs) handle critical processing that the application must perform in response to external asynchronous events. The SYS/BIOS device-specific HWI modules are used to manage hardware interrupts. Specific information on the nesting, vectoring, and functionality of interrupts can be found in the *TI CC26xx Technical Reference Manual* ([SWCU117](#)). The SYS/BIOS User Guide details the HWI API and provides several software examples.

HWIs are abstracted through the peripheral driver to which they pertain to (see the relevant driver in [Chapter 6](#)). [Chapter 9](#) provides an example of using GPIOs as HWIs. Abstracting through the peripheral driver to which they pertain is the preferred method of using interrupts. Using the `Hwi_plug()` function, ISRs can be written which do not interact with SYS/BIOS. These ISRs must do their own context preservation to prevent breaking the time-critical Bluetooth low energy stack.

For the Bluetooth low energy protocol stack to meet RF time-critical requirements, all application-defined HWIs execute at the lowest priority. TI does not recommend modifying the default HWI priority when adding new HWIs to the system. No application-defined critical sections should exist to prevent breaking the RTOS or time-critical sections of the Bluetooth low energy protocol stack. Code executing in a critical section prevents processing of real-time interrupt-related events.

3.9 Software Interrupts

See [TI SYS/BIOS API Guide](#) for detailed information about the SWI module. Software interrupts have priorities that are higher than tasks but lower than hardware interrupts (see [Figure 3-5](#)). The amount of processing in a SWI must be limited as this processing takes priority over the Bluetooth low energy protocol stack task. Just as with HWIs, blocking API calls, including calls to the protocol stack APIs, cannot be made in an SWI. As described in [Section 3.4](#), the clock module uses SWIs to preempt tasks. The only processing the clock handler SWI does is set an event and post a semaphore for the application to continue processing outside of the SWI. Whenever possible, the Clock module should be used to implement SWIs. A SWI can be implemented with the SWI module as described in [TI SYS/BIOS API Guide](#).

NOTE: To preserve the RTOS heap, the amount of dynamically created SWIs must be limited as described in [Section 3.11.5](#).

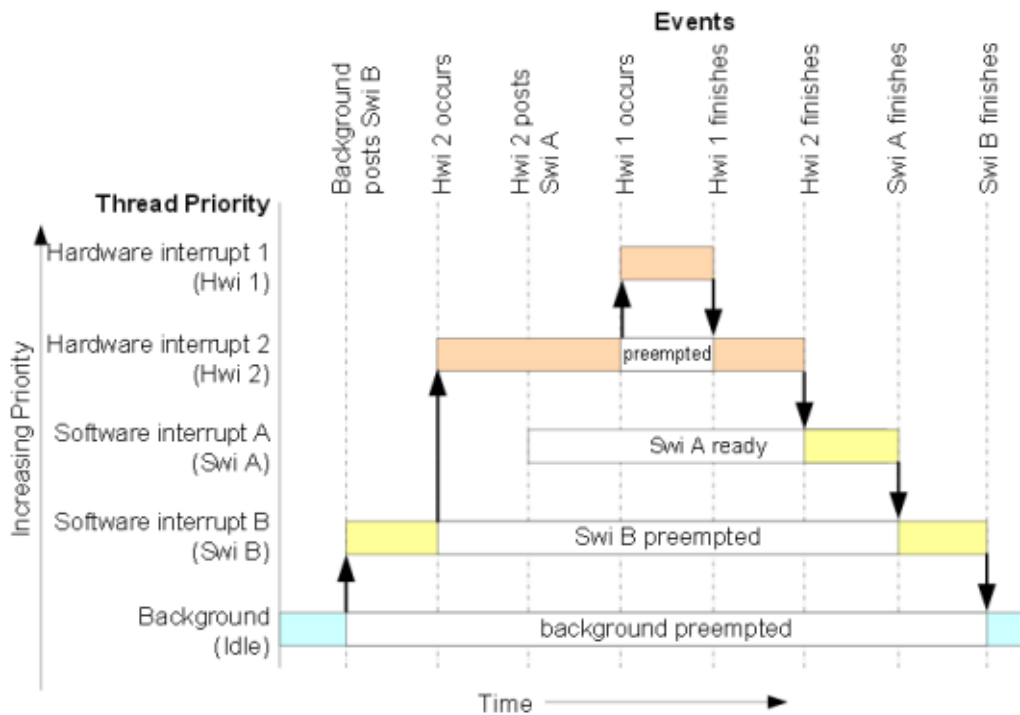


Figure 3-5. Preemption Scenario

3.10 Flash

The flash is split into erasable pages of 4kB. The application and stack projects must each start on a 4kB aligned flash address. The various sections of flash and their associate linker files are as follows.

- Application Image: code space for the application project. This image is configured in the linker configuration file of the application: `cc26xx_app.icf` (IAR) and `cc26xx_app.cmd` (CCS).
- Stack Image: code space for the stack project. This image is configured in the linker configuration file of the stack: `cc26xx_stack.icf` (IAR) and `cc26xx_stack.cmd` (CCS).
- Simple NV (SNV): area used for nonvolatile memory storage by the GAP Bond Manager and also available for use by the application. See [Section 3.10.3](#) for configuring SNV. When configured, the SNV flash storage area is part of the stack image.
- Customer Configuration Area (CCA): the last sector of flash used to store customer-specific chip configuration (CCFG) parameters. The unused space of the CCA sector is allocated to the application project. See [Section 3.10.4](#).

3.10.1 Flash Memory Map

This section describes the flash memory map at the system level. As [Figure 3-6](#) shows, the application linker file point to symbols with a solid arrow and the stack linker file point to symbols with a dashed arrow.

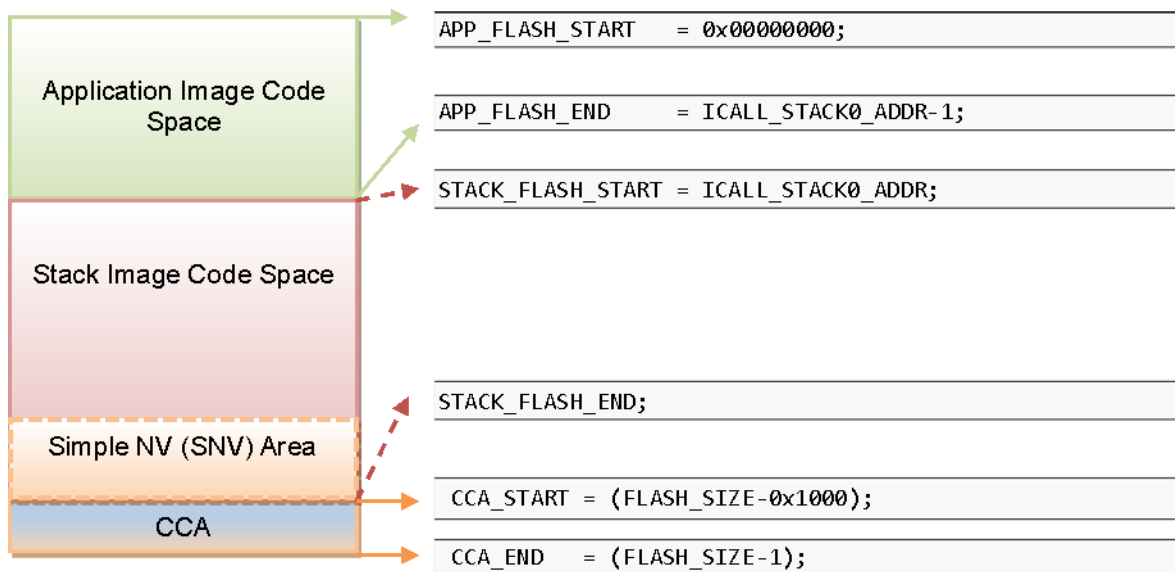


Figure 3-6. System Flash Map

[Table 3-1](#) summarizes the Flash System Map definitions from [Figure 3-6](#) and provides the associated linker definitions or symbols that can be found in the respective IDE linker files.

Table 3-1. Flash System Map Definitions

| Symbol/Region | Meaning | Project | CCS Definition | IAR Definition |
|-------------------|---|---------|--|----------------------|
| APP_FLASH_START | Start of flash/Start of App code image | App | APP_BASE | FLASH_START |
| APP_FLASH_END | End of App code image. (ICALL_STACK0_ADDR-1) | App | ICALL_STACK0_ADDR - APP_BASE - 1 | FLASH_END |
| STACK_FLASH_START | Start of Stack code image (ICALL_STACK0_ADDR) | Stack | ICALL_STACK0_ADDR | FLASH_START |
| STACK_FLASH_END | End of Stack flash code image, including SNV | Stack | FLASH_SIZE - RESERVED_SIZE - ICALL_STACK0_ADDR | FLASH_END |
| CCA sector | Last sector of flash. Contains the CCFG. | App | FLASH_LAST_PAGE | FLASH_LAST_PAGE |
| CCFG region | Location in CCA where Customer Configuration (CCFG) parameters are stored | App | Last 86 bytes of CCA | Last 86 bytes of CCA |

3.10.2 Application and Stack Flash Boundary

The application and stack code images are based on the common `ICALL_STACK0_ADDR` and `ICALL_STACK0_START` predefined symbols. These values define the hardcoded flash address (4kB aligned) of the entry function for the stack image: it is essentially the flash address of the application–stack project boundary. To ensure proper linking, both the application and stack projects must use the same defined symbols. By default, the linker is configured to allocate unused flash to the application project but can be modified manually or automatically through the frontier tool. For information on using the frontier tool to configure the flash boundary address, see [Section 3.12](#).

3.10.3 Using Simple NV for Flash Storage

The Simple NV (SNV) area of flash is used for storing persistent data, such as encryption keys from bonding or to store custom defined parameters. The protocol stack can be configured to reserve up to two 4kB flash pages for SNV, although valid data is only stored in one active flash page. To minimize the number of erase cycles on the flash, the SNV manager performs compactions on the flash sector (or sectors) when the sector has 80% invalidated data. A compaction is the copying of valid data to a temporary area followed by an erase of the sector where the data was previously stored. Depending on the `OSAL_SNV` value as described in [Table 3-2](#), this valid data is then either placed back in the newly erased sector or remains in a new sector. The number of flash sectors allocated to SNV can be configured by setting the `OSAL_SNV` preprocessor symbol in the stack project. [Table 3-2](#) lists the valid values that can be configured as well as the corresponding trade-offs.

Table 3-2. OSAL_SNV Values

| OSAL_SNV Value | Description |
|----------------|---|
| 0 | SNV is disabled. Storing of bonding keys in NV is not possible. Maximizes code space for the application and/or stack project. GAP Bond Manager must be disabled. In the Stack project, set preprocessor symbol <code>NO_OSAL_SNV</code> and disable GAP Bond Manager. See Section 10.4 for configuring Bluetooth low energy protocol stack features. |
| 1 (default) | One flash sector is allocated to SNV. Bonding info is stored in NV. Flash compaction uses flash cache RAM for intermediate storage, thus a power-loss during compaction results in SNV data loss. Also, due to temporarily disabling the cache, a system performance degradation may occur during the compaction. Set preprocessor symbol <code>OSAL_SNV=1</code> in the Stack project. |
| 2 | Two flash sectors are allocated to SNV. Bonding information is stored in NV. SNV data is protected against power-loss during compaction. |

Other values for `OSAL_SNV` are invalid. Using less than the maximum value has the net effect of allocating more code space to the application or stack project. SNV can be read from or written to using the following APIs.

uint8 osal_snv_read(osalSnvId_t id, osalSnvLen_t len, void *pBuf)

Read data from NV.

Parameters

`id` – valid NV item
`len` – length of data to read
`pBuf` – pointer to buffer to store data read

Returns

SUCCESS: NV item read successfully
 NV_OPER_FAILED: failure reading NV item

uint8 osal_snv_write(osalSnvId_t id, osalSnvLen_t len, void *pBuf)

Write data to NV

Parameters

`id` – valid NV item
`len` – length of data to write
`pBuf` – pointer to buffer containing data to be written. All contents are updated at once.

Returns **SUCCESS:** NV item read successfully
 NV_OPER_FAILED: failure reading NV item

Because SNV is shared with other modules in the *Bluetooth* low energy SDK such as the GapBondMgr, carefully manage the NV item IDs. By default, the IDs available to the customer are defined in bcomdef.h.

```
// Customer NV Items - Range 0x80 - 0x8F - This must match the number of Bonding entries
#define BLE_NVID_CUST_START 0x80 //!< Start of the Customer's NV IDs
#define BLE_NVID_CUST_END 0x80 //!< End of the Customer's NV IDs
```

The following example shows how to read and write a byte using SNV flash:

```
#define BUF_LEN 1
#define SNV_ID_APP 0x80
uint8 buf[BUF_LEN] = {0,};

// Initialize application
simple_peripheral_init();
uint8 status = SUCCESS;

//Read from SNV flash
status = osal_snv_read(SNV_ID_APP, BUF_LEN, (uint8 *)buf);
if(status != SUCCESS)
{
    DISPLAY_WRITE_STRING_VALUE("SNV READ FAIL: %d", status, LCD_PAGE5);
    //Write first time to initialize SNV ID
    osal_snv_write(SNV_ID_APP, BUF_LEN, (uint8 *)buf);
}

//Increment value and write to SNV flash
buf[0]++;
status = osal_snv_write(SNV_ID_APP, BUF_LEN, (uint8 *)buf);
if(status != SUCCESS)
    DISPLAY_WRITE_STRING_VALUE("SNV WRITE FAIL: %d", status, LCD_PAGE6);
else
    DISPLAY_WRITE_STRING_VALUE("Num of Resets: %d", buf[0], LCD_PAGE6);
```

No prior initialization of a NV item ID is required; the OSAL SNV manager initializes the NV ID when first accessed by a successful `osal_snv_write()` call.

When reading or writing large amounts of data to SNV, TI recommends placing the read/write data in statically (linker) allocated arrays or buffers allocated from the heap. Placing large amounts of data in local arrays may result in a task stack overflow.

By default, `osalSnvId_t` and `osalSnvLen_t` are type defined as `uint8`. To use `uint16`-type definitions, define the preprocessor symbol `OSAL_SNV_UINT16_ID` in both the application and stack projects.

3.10.4 Customer Configuration Area

The Customer Configuration Area (CCA) occupies the last page of flash and lets a customer configure various chip and system parameters in the Customer Configuration (CCFG) table. The CCFG table is defined in `ccfg_app_ble.c`, which can be found in the Startup folder of the application project. The last 86 bytes of the CCA sector are reserved by the system for the CCFG table. By default, the linker allocates the unused flash of the CCA sector to the application image for code and data use. The linker can be modified to reserve the entire sector for customer parameter data (for example, board serial number and other identity parameters).

The CCA region is defined linker file of the application by the `FLASH_LAST_PAGE` symbol; placement is based on the IDE:

For CCS:

```
FLASH_LAST_PAGE (RX) : origin = FLASH_SIZE - 0x1000, length = 0x1000
...
.ccfg : > FLASH_LAST_PAGE (HIGH)
```

For IAR:

```
define region FLASH_LAST_PAGE = mem:[from(FLASH_SIZE) - 0x1000 to FLASH_SIZE-1];
```

```
...
place at end of FLASH_LAST_PAGE { readonly section .ccfg };
```

See the *TI CC26xx Technical Reference Manual (SWCU117)* for details on CCFG fields and related configuration options, including how to set the CCFG to disable access to internal flash memory contents.

3.11 Memory Management (RAM)

Similar to flash, the RAM is shared between the application and stack projects. The RAM sections are configured in their respective linker files.

- Application Image: RAM space for the application and shared heaps. This image is configured in the linker configuration file of the application: `cc26xx_app.icf` (IAR) and `cc26xx_app.cmd` (CCS).
- Stack Image: RAM space for the `.bss` and `.data` sections of the stack. This image is configured in the linker configuration file of the stack: `cc26xx_stack.icf` (IAR) and `cc26xx_stack.cmd` (CCS).

3.11.1 RAM Memory Map

Figure 3-7 shows the system memory map for the default `simple_peripheral` project. This is a summary and the exact memory placement for a given compilation can be found in the `simple_peripheral_app.map` and `simple_peripheral_stack.map` files in the output folder in IAR or the FlashROM folder in CCS. See Section 0 for more information about these files. In Section 9.13, the application linker file contains symbols pointed with a solid arrow and the stack linker file contains symbols pointed with a dashed arrow.

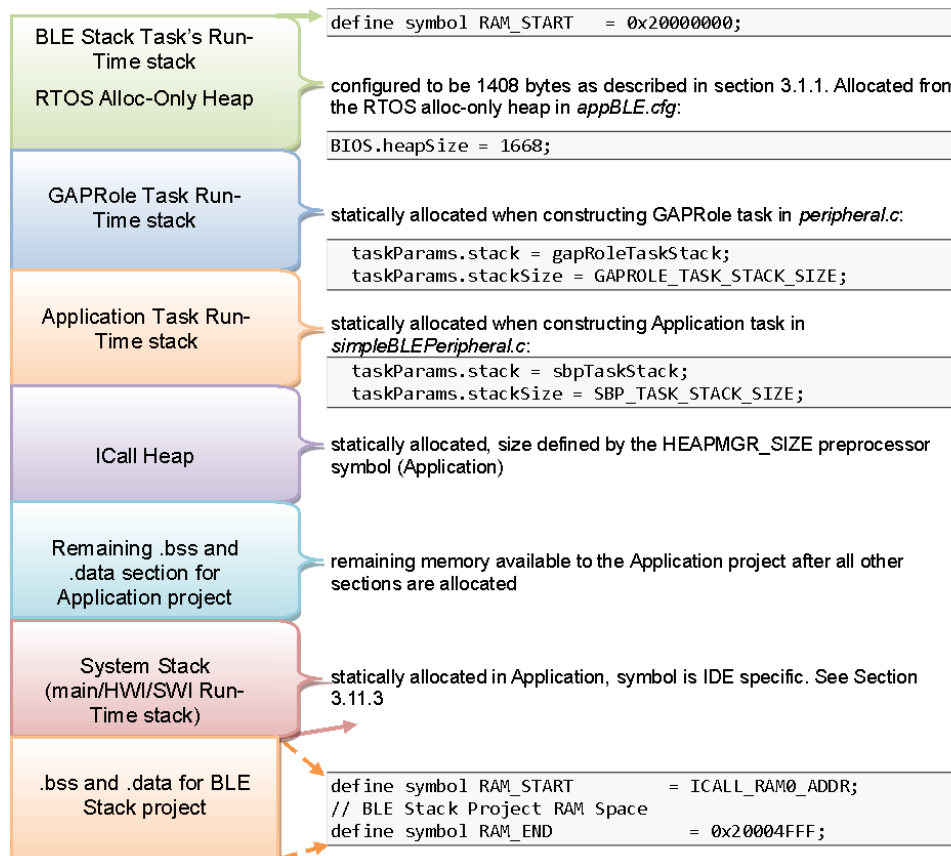


Figure 3-7. System Memory Map

3.11.2 Application and Stack RAM Boundary

The application and stack RAM memory maps are based on the common ICALL_RAM0_START defined symbol. This value defines the hardcoded RAM boundary for the end of the RAM space of the application and the start of the image of the stack .BSS and .DATA sections. Unlike the flash boundary, elements of the stack project (such as task stacks and heaps) are allocated in the application project. To ensure proper linking, both the application and stack projects must use the same ICALL_RAM0_START value. By default, ICALL_RAM0_START is configured to allocate unused RAM to the application project through the frontier tool. For information on using the frontier tool to configure the RAM boundary address, see [Section 3.12](#).

3.11.3 System Stack

Besides the RTOS and ICall heaps, consider other sections of memory. As described in [Section 3.3.1](#), each task has its own runtime stack for context switching. Another runtime stack is used by the RTOS for main(), HWIs, and SWIs. This system stack is allocated in the application linker file to be placed at the end of the RAM of the application.

For IAR, this RTOS system stack is defined by the CSTACK symbol:

```

////////////////////////////////////
// Stack
//

define symbol STACK_SIZE      = 0x400
define symbol STACK_START = RAM_END + 1;
define symbol STACK_END = STACK_START - STACK_SIZE;
define block CSTACK with alignment = 8, size = STACK_SIZE { section .stack };
//
define symbol STACK_TOP = RAM_END + 1;
export symbol STACK_TOP;
//
place at end of RAM { block CSTACK };

```

In IAR, to change the size of the CSTACK, adjust the STACK_SIZE symbol value in the linker file of the application.

For CCS, the RTOS system stack is defined by the Program.stack parameter in the appBLE.cfg RTOS configuration file:

```

/* main() and Hwi, Swi stack size */
Program.stack = 1024;

```

and placed by the linker in the RAM space of the application:

```

/* Create global constant that points to top of stack */
/* CCS: Change stack size under Project Properties */
__STACK_TOP = __stack + __STACK_SIZE;

```

3.11.4 Dynamic Memory Allocation

The system uses two heaps for dynamic memory allocation. The application designer must understand the use of each heap to maximize the use of available memory.

The RTOS is configured with a small heap in the app_ble.cfg RTOS configuration file:

```

var HeapMem = xdc.useModule('xdc.runtime.HeapMem');

BIOS.heapSize = 1668;

```

This heap (HeapMem) is used to initialize RTOS objects and allocate the task runtime stack of the Bluetooth low energy protocol stack. TI chose this size of this heap to meet the system initialization requirements. Due to the small size of this heap, TI does not recommend allocating memory from the RTOS heap for general application use. For more information on the TI-RTOS heap configuration, see the Heap Implementations section of the [TI-RTOS SYS/BIOS Kernel User's Guide](#).

The application must use a separate heap. The ICall module uses an area of application RAM which can be used by the various tasks. The size of this ICall heap is defined by the `HEAPMGR_SIZE` preprocessor definition in the application project. Using a non-zero value sets the ICall heap to the specified value, while a `HEAPMGR_SIZE` value of zero (0) auto sizes the heap to a size equal to the amount of available free RAM not allocated by the linker. By default, the `simple_peripheral` project uses the auto size feature. Although the ICall heap is defined in the application project, this heap is also shared with the Bluetooth low energy protocol stack. APIs that allocate memory (such as `GATT_bm_alloc()`) allocate memory from the ICall heap.

To profile the amount of ICall heap used, define the `HEAPMGR_METRICS` preprocessor symbol in the application project. Refer to [Section 9.6](#) to determine the size of the ICall heap when the auto heap size feature is enabled..

NOTE: The auto heap size feature does not determine the amount of heap needed for the application. The system designer must ensure that the heap has the required space to meet the application's runtime memory requirements.

The following is an example of dynamically allocating a variable length (n) array using the ICall heap:

```
//define pointer
uint8_t *pArray;

// Create dynamic pointer to array.
if (pArray = (uint8_t*)ICall_malloc(n*sizeof(uint8_t)))
{
    //fill up array
}
else
{
    //not able to allocate
}
```

The following is an example of freeing the previous array:

```
ICall_free(pMsg->payload);
```

3.11.5 Initializing RTOS Objects

Due to the limited size of the RTOS heap, TI recommends constructing and not creating RTOS objects. Consider the difference between the `Clock_construct()` and `Clock_create()` functions. The following shows their definitions from the SYS/BIOS API:

```
Clock_Handle Clock_create(Clock_FuncPtr clockFxn, UInt timeout, const Clock_Params *params,
Error_Block *eb);
    // Allocate and initialize a new instance object and return its handle

Void Clock_construct(Clock_Struct *structP, Clock_FuncPtr clockFxn, UInt timeout, const
Clock_Params *params);
    //Initialize a new instance object inside the provided structure
```

By declaring a static `Clock_Struct` object and passing this object to `Clock_construct()`, the `.DATA` section for the actual `Clock_Struct` is used; not the limited RTOS heap. `Clock_create()` would cause the RTOS to allocate the `Clock_Struct` using the limited heap of the RTOS.

This example shows how clocks and RTOS objects should be initialized throughout a project. If creating RTOS objects, the size of the RTOS heap may require adjustment in `app_ble.cfg`.

3.12 Configuration of RAM and Flash Boundary Using the Frontier Tool

The frontier tool is a utility to automatically adjust the respective RAM and flash boundary address symbols shared between the application and stack projects. Frontier runs as a post-build step of the stack project, and adjusts the respective RAM and flash boundaries based on analysis of the stack linker and map files. No project files are modified by the frontier tool. The frontier tool does not modify any source code or perform any compiler or linker optimization; the tool adjusts and updates the respective flash and RAM boundary addresses, located in the compiler and linker configuration files used by the application and stack project.

The frontier tool is installed to the following path within the SDK:

```
$BLE_INSTALL\tools\frontier\frontier.exe
```

The python source for this tool is also included.

Table 3-3 shows the boundary address symbols updated by the frontier tool.

Table 3-3. Boundary Address Symbols

| Boundary Address Symbol | Definition |
|-------------------------|--|
| ICALL_STACK0_START | Boundary flash address between application and stack images. Represents the end of the application image and the beginning of the stack image. |
| ICALL_STACK0_ADDR | Stack entry address (flash) |
| ICALL_RAM0_START | Boundary RAM address between application and stack images. Represents the end of the application RAM and the beginning of the stack RAM. |

All sample application projects are, by default, configured to use the frontier tool; thus, no user configuration of the frontier tool is required. The boundary files may be updated when the stack configuration is changed, or when any files are updated in the stack project that result in a change in the size of the stack image. It is therefore required to rebuild the application project anytime the stack project is built.

Note for previous SDK users: The frontier tool replaces the boundary tool used in earlier SDKs.

3.12.1 Frontier Tool Operation

The frontier tool (frontier.exe) is invoked as a CCS or IAR IDE post-build operation of the stack project. If an adjustment to the RAM or flash boundary is required, the frontier tool updates the boundary linker configuration and C definition files listed below. To incorporate the updated configuration values, perform a Project → Rebuild All on the application project. The stack project must build and link correctly before the application can be rebuilt.

Each project in the SDK has a set of configuration files that the linker and compiler of the IDE use to set or adjust the respective flash and RAM values. These configuration files are shared between the application and stack workspaces, and are stored at the following location:

```
$BLE_INSTALL\examples\<EVAL_BOARD>\<PROJECT>\<IDE>\config
```

Where <EVAL_BOARD> is the evaluation platform, <PROJECT> is the sample application (for example, simple_peripheral), and <IDE> is either IAR or CCS.

For the simple_peripheral sample application running on the CC2650 LaunchPad, the boundary config files are located at the following path:

```
CCS: $BLE_INSTALL\examples\cc2650\simple_peripheral\ccs\config
```

```
IAR: $BLE_INSTALL\examples\cc2650\simple_peripheral\iar\config
```

- Boundary linker configuration file: iar_boundary.xcl [IAR] or ccs_linker_defines.cmd [CCS]. Defines the boundary addresses for the linker. This file is in the TOOLS IDE folder and is updated by the frontier tool when an adjustment is required.
- Boundary C definition file: iar_boundary.cdef [IAR] or ccs_compiler_defines.bcfg [CCS]. Defines the boundary addresses for the compiler. This file is in the TOOLS IDE folder and is updated by the frontier tool when an adjustment is required.

NOTE: The values in boundary linker configuration file and Boundary C definition file must match.

3.12.2 Disabling the Frontier Tool

Under normal conditions, it is not recommended to disable the frontier tool. To disable the frontier tool, follow these steps:

1. Open the project options for the stack project.
2. Select Build Actions (IAR) or Steps in the CCS build window (CCS).
3. Remove the post-build command line (see [Figure 3-8](#) and [Figure 3-9](#)).

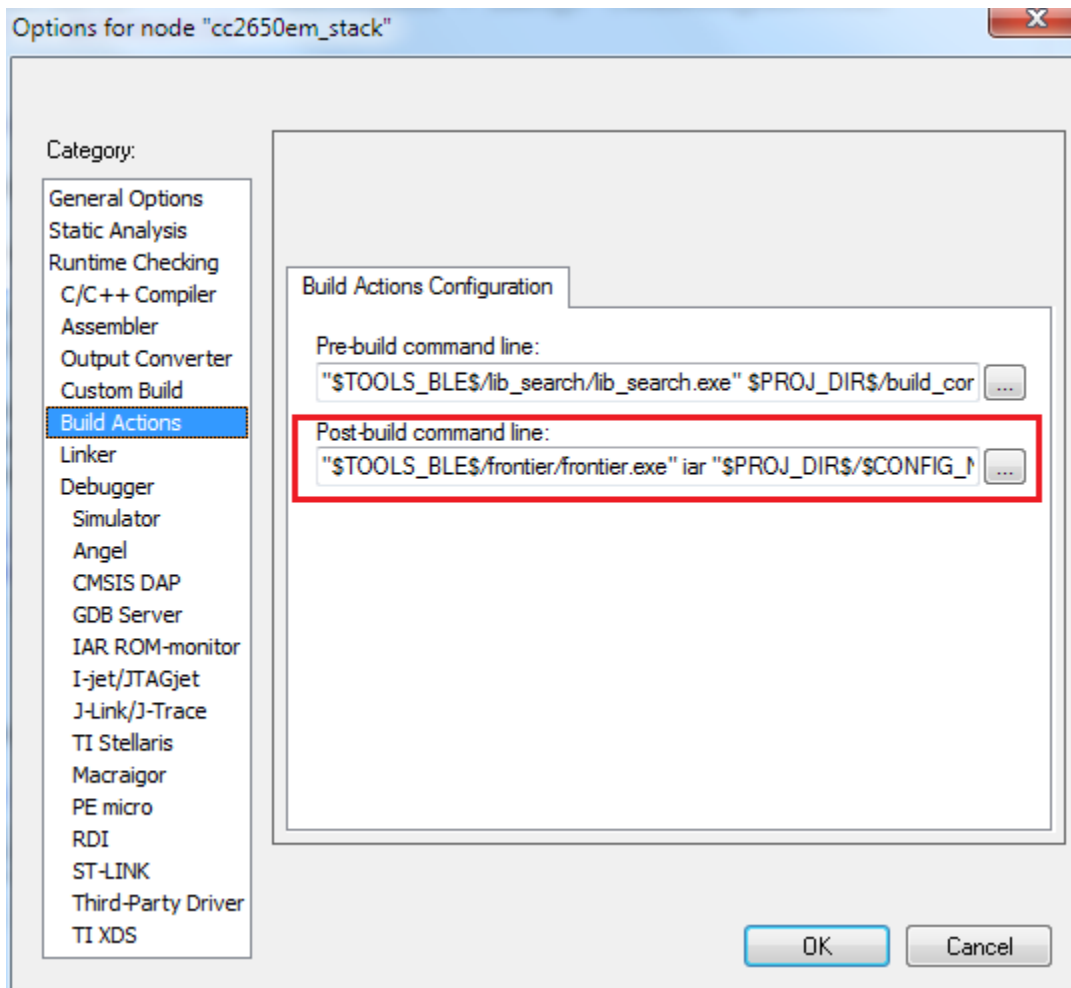


Figure 3-8. Disabling Frontier Tool from Stack Project in IAR

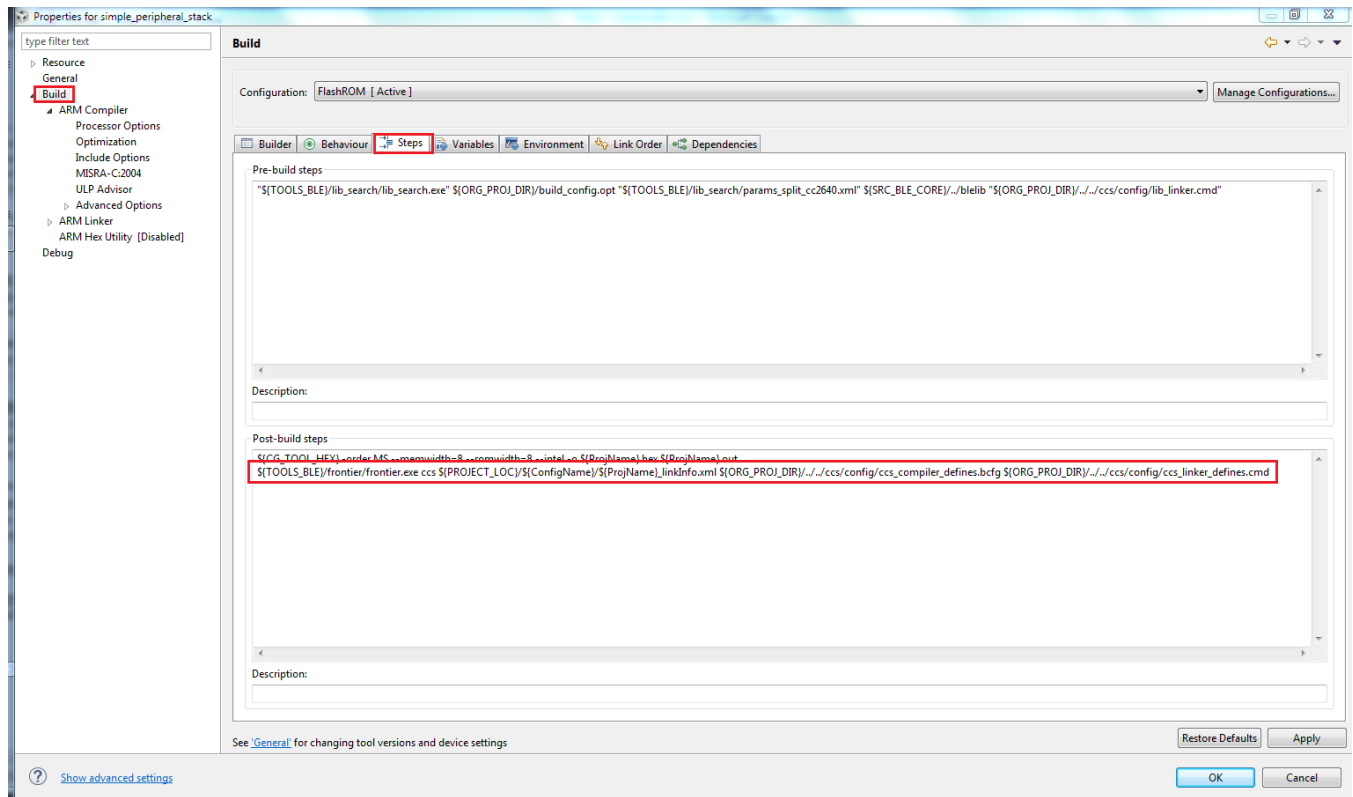


Figure 3-9. Disabling Frontier Tool from Stack Project in CCS

The Application

This section describes the application portion of the simple_peripheral project, which includes the following:

- Pre-RTOS initialization
- simple_peripheral task: This is the application task, which is the lowest priority task in the system. The code for this task is in simple_peripheral.c and simple_peripheral in the Application IDE folder.
- ICall: This interface module abstracts communication between the stack and other tasks.

NOTE: The GAPRole task is also part of the application project workspace. The functionality of this task relates more closely to the protocol stack.

[Section 5.2](#) describes this functionality.

4.1 Start-Up in main()

The main() function inside of main.c in the IDE Start-up folder is the starting point at run time. This point is where the board is brought up with interrupts disabled and drivers are initialized. Also in this function, power management is initialized and the tasks are created or constructed. In the final step, interrupts are enabled and the SYS/BIOS kernel scheduler is started by calling BIOS_start(), which does not return. See [Chapter 8](#) for information on the start-up sequence before main() is reached.

```
int main()
{
    PIN_init(BoardGpioInitTable);

#ifdef POWER_SAVING
    /* Set constraints for Stanby, powerdown and idle mode */
    Power_setConstraint(Power_SB_DISALLOW);
    Power_setConstraint(Power_IDLE_PD_DISALLOW);
#endif // POWER_SAVING

    /* Initialize ICall module */
    ICall_init();

    /* Start tasks of external images - Priority 5 */
    ICall_createRemoteTasks();

    /* Kick off profile - Priority 3 */
    GAPRole_createTask();

    SimpleBLEPeripheral_createTask();

    /* enable interrupts and start SYS/BIOS */
    BIOS_start();

    return 0;
}
```

[Chapter 3](#) describes how the application and GAPRole tasks are constructed. The stack task is created here as well in ICall_createRemoteTasks(). The ICall module is initialized through ICall_init(). In terms of the IDE workspace, main.c exists in the application project (when the project is compiled and placed in the allocated section of flash of the application).

4.2 ICall

4.2.1 Introduction

Indirect Call Framework (ICall) is a module that provides a mechanism for the application to interface with the Bluetooth low energy protocol stack services (that is, Bluetooth low energy stack APIs) as well as certain primitive services provided by the RTOS (for example, thread synchronization). ICall allows the application and protocol stack to operate efficiently, communicate, and share resources in a unified RTOS environment.

The central component of the ICall architecture is the dispatcher, which facilitates the application program interface between the application and the Bluetooth low energy protocol stack task across the dual-image boundary. Although most ICall interactions are abstracted within the Bluetooth low energy protocol stack APIs (for example, GAP, HCI, and so forth), the application developer must understand the underlying architecture for the Bluetooth low energy protocol stack to operate properly in the multithreaded RTOS environment.

The ICall module source code is provided in the ICall and ICall Bluetooth low energy IDE folders in the application project.

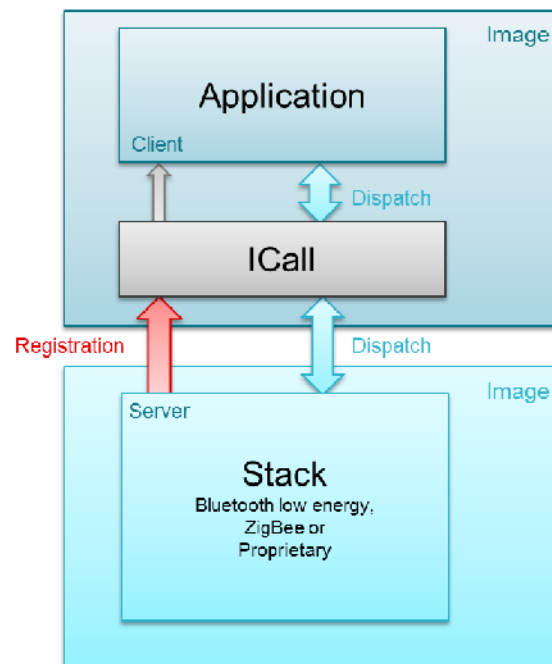


Figure 4-1. ICall Application – Protocol Stack Abstraction

4.2.2 ICall Bluetooth low energy Protocol Stack Service

As [Figure 4-1](#) shows, the ICall core use case involves messaging between a server entity (that is, the Bluetooth low energy stack task) and a client entity (for example, the application task).

NOTE: The ICall framework is not the GATT server and client architecture as defined by the Bluetooth low energy protocol.

The reasoning for this architecture is as follows:

- To enable independent updating of the application and Bluetooth low energy protocol stack
- To maintain API consistency as software is ported from legacy platforms (that is, OSAL for the CC254x) to the TI-RTOS of the CC2640

The ICall Bluetooth low energy protocol stack service serves as the application interface to Bluetooth low energy stack APIs. When a Bluetooth low energy protocol stack API is called by the application internally, the ICall module routes (that is, dispatches) the command to the Bluetooth low energy protocol stack and routes messages from the Bluetooth low energy protocol stack to the application when appropriate.

Because the ICall module is part of the application project, the application task can access ICall with direct function calls. Because the Bluetooth low energy protocol stack executes at the highest priority, the application task blocks until the response is received. Certain protocol stack APIs may respond immediately, but the application thread blocks as the API is dispatched to the Bluetooth low energy protocol stack through ICall. Other Bluetooth low energy protocol stack APIs may also respond asynchronously to the application through ICall (for example, event updates) with the response sent to the event handler of the application task.

4.2.3 ICall Primitive Service

ICall includes a primitive service that abstracts various operating system-related functions. Due to shared resources and to maintain interprocess communication, the application must use the following ICall primitive service functions:

- Messaging and Thread Synchronization
- Heap Allocation and Management

Some of these are abstracted to Util functions (see the relevant module in [Chapter 3](#)).

4.2.3.1 Messaging and Thread Synchronization

The Messaging and Thread Synchronization functions provided by ICall enable designing an application to protocol stack interface in the multithreaded RTOS environment.

In ICall, messaging between two tasks occurs by sending a block of message from one thread to the other through a message queue. The sender allocates a memory block, writes the content of the message into the memory block, and then sends (that is, enqueues) the memory block to the recipient. Notification of message delivery occurs using a signaling semaphore. The receiver wakes up on the semaphore, copies the message memory block (or blocks), processes the message, and returns (frees) the memory block to the heap.

The stack uses ICall for notifying and sending messages to the application. ICall delivers these service messages, the application task receives them, and the messages are processed in the context of the application.

4.2.3.2 Heap Allocation and Management

ICall provides the application with global heap APIs for dynamic memory allocation. The size of the ICall heap is configured with the `HEAPMGR_SIZE` preprocessor-defined symbol in the application project. See [Section 3.11.4](#) for more details on managing dynamic memory. ICall uses this heap for all protocol stack messaging and to obtain memory for other ICall services. TI recommends that the application uses these ICall APIs to allocate dynamic memory.

4.2.4 ICall Initialization and Registration

To instantiate and initialize the ICall service, the application must call the following functions in main() before starting the SYS/BIOS kernel scheduler:

```
/* Initialize ICall module */
ICall_init();
/* Start tasks of external images - Priority 5 */
ICall_createRemoteTasks();
```

Calling ICall_init() initializes the ICall primitive service (for example, heap manager) and framework. Calling ICall_createRemoteTasks() creates but does not start the Bluetooth low energy protocol stack task. Before using ICall protocol services, the server and client must enroll and register with ICall. The server enrolls a service, which is defined at build time. Service function handler registration uses a globally defined unique identifier for each service. For example, Bluetooth low energy uses ICALL_SERVICE_CLASS_BLE for receiving Bluetooth low energy protocol stack messages through ICall.

The following is a call to enroll the Bluetooth low energy protocol stack service (server) with ICall in osal_icall_ble.c:

```
// ICall enrollment
/* Enroll the service that this stack represents */
ICall_enrollService(ICALL_SERVICE_CLASS_BLE, NULL, &entity, &sem);
```

The registration mechanism is used by the client to send and/or receive messages through the ICall dispatcher.

For a client (for example, application task) to use the Bluetooth low energy stack APIs, the client must first register its task with ICall. This registration usually occurs in the task initialization function of the application. The following is an example from simple_peripheral_int() in simple_peripheral:

```
// Register the current thread as an ICall dispatcher application
// so that the application can send and receive messages.
ICall_registerApp(&selfEntity, &sem);
```

The application supplies the selfEntity and sem inputs. These inputs are initialized for the task of the client (for example, application) when the ICall_registerApp() returns are initialized. These objects are subsequently used by ICall to facilitate messaging between the application and server tasks. The sem argument represents the semaphore for signaling and the selfEntity represents the destination message queue of the task. Each task registering with ICall have unique sem and selfEntity identifiers.

NOTE: Bluetooth low energy protocol stack APIs defined in ICallBLEApi.c and other ICall primitive services are not available before ICall registration.

4.2.5 ICall Thread Synchronization

The ICall module switches between application and stack threads through Preemption and Semaphore Synchronization services provided by the RTOS. The two ICall functions to retrieve and enqueue messages are not blocking functions. These functions check whether there is a received message in the queue and if there is no message, the functions return immediately with `ICALL_ERRNO_NOMSG` return value. To allow a client or a server thread to block until it receives a message, ICall provides the following function which blocks until the semaphore associated with the caller RTOS thread is posted:

```
//static inline ICall_Errno ICall_wait(uint_fast32_t milliseconds)
ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);
```

`milliseconds` is a time-out period in milliseconds. If not already returned after this time-out period, the function returns with `ICALL_ERRNO_TIMEOUT`. If `ICALL_TIMEOUT_FOREVER` is passed as `milliseconds`, the `ICall_wait()` blocks until the semaphore is posted. Allowing an application or a server thread to block yields the processor resource to other lower priority threads or conserves energy by shutting down power and/or clock domains when possible.

The semaphore associated with an RTOS thread is signaled by either of the following conditions:

- A new message is queued to the RTOS thread queue of the application.
- `ICall_signal()` is called to unblock the semaphore.

`ICall_signal()` is used so an application or a server can add its own event to unblock `ICall_wait()` and synchronize the thread. `ICall_signal()` accepts semaphore handle as its sole argument as follows:

```
//static inline ICall_Errno ICall_signal(ICall_Semaphore msgem)
ICall_signal(sem);
```

The semaphore handle associated with the thread is obtained through either `ICall_enrollService()` call or `ICall_registerApp()` call.

NOTE: Do not call an ICall function from a stack callback. This action can cause ICall to abort (with `ICall_abort()`) and break the system.

4.2.6 Example ICall Usage

Figure 4-2 shows an example command being sent from the application to the Bluetooth low energy protocol stack through the ICall framework with a corresponding return value passed back to the application. ICall_init() initializes the ICall module instance and ICall_createRemoteTasks() creates a task per external image with an entry function at a known address. After initializing ICall, the application task registers with ICall through ICall_registerApp. After the SYS/BIOS scheduler starts and the application task runs, the application sends a protocol command defined in ICallBLEAPI.c such as GAP_GetParamValue(). The protocol command is not executed in the thread of the application but is encapsulated in an ICall message and routed to the Bluetooth low energy protocol stack task through the ICall framework. This command is sent to the ICall dispatcher where it is dispatched and executed on the server side (that is, Bluetooth low energy stack). The application thread meanwhile blocks (that is, waits) for the corresponding command status message (that is, status and GAP parameter value). When the Bluetooth low energy protocol stack finishes executing the command, the command status message response is sent through ICall back to the application thread.

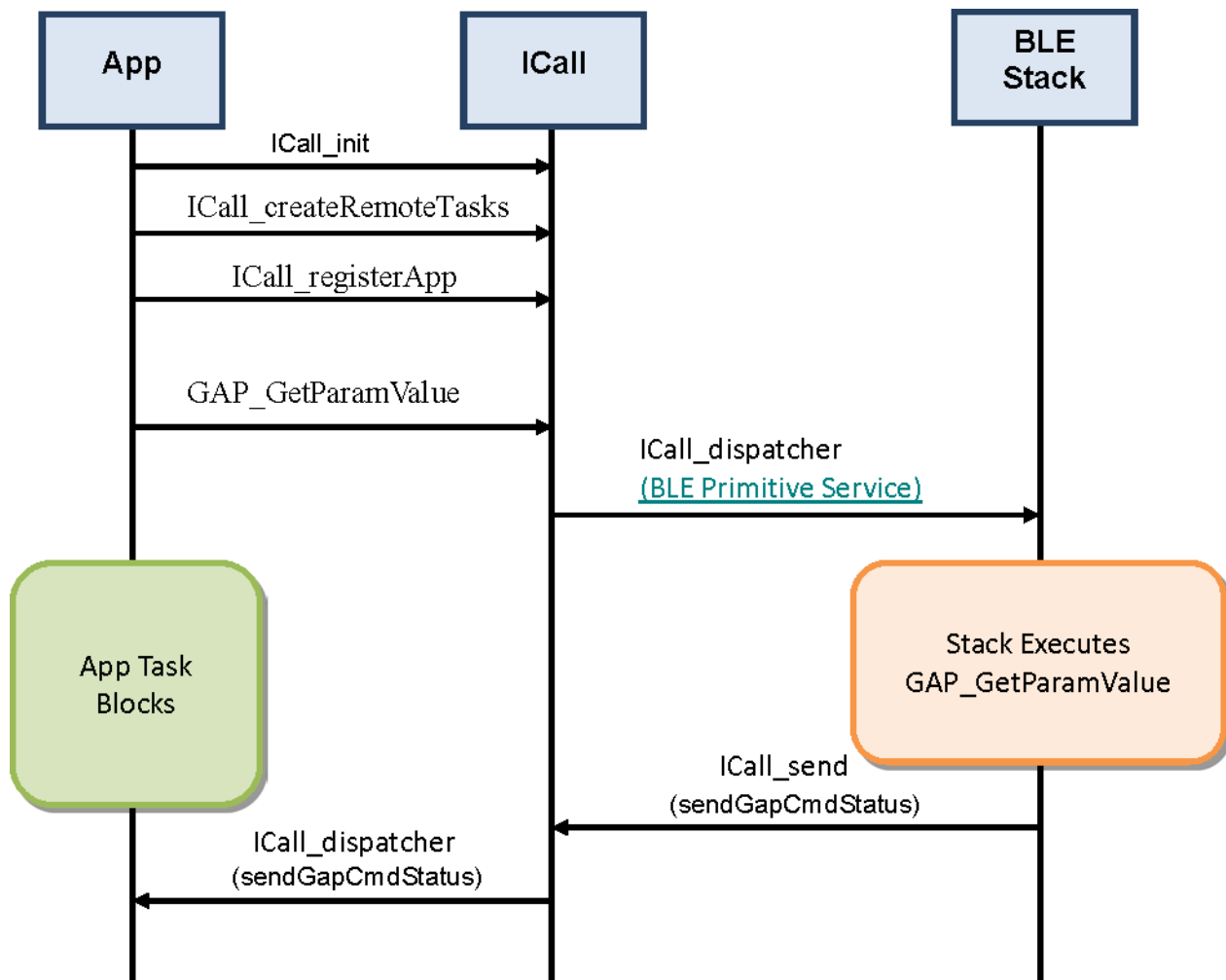


Figure 4-2. ICall Messaging Example

4.3 General Application Architecture

This section describes in detail how an application task is constructed.

4.3.1 Application Initialization Function

[Section 3.3](#) describes how a task is constructed. After the task is constructed and the SYS/BIOS kernel scheduler is started, the function that was passed during task construction is run when the task is ready (for example, `simple_peripheral_taskFxn`). This function must first call an application initialization function. For example, in `simple_peripheral.c`:

```
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
    // Initialize application
    SimpleBLEPeripheral_init();

    // Application main loop
    for (;;)
    {
        ...
    }
}
```

This initialization function (`simple_peripheral_init()`) configures several services for the task and sets several hardware and software configuration settings and parameters. The following list contains some common examples:

- Initializing the GATT client
- Registering for callbacks in various profiles
- Setting up the GAPRole
- Setting up the Bond Manager
- Setting up the GAP layer
- Configuring hardware modules such as LCD, SPI, and so forth

For more information on all of these examples, see their respective sections in this guide.

NOTE: In the application initialization function, `ICall_registerApp()` must be called before any stack API is called.

4.3.2 Event Processing in the Task Function

After the initialization function shown in the previous code snippet, the task function enters an infinite loop so that it continuously processes as an independent task and does not run to completion. In this infinite loop, the task remains blocked and waits until a semaphore signals a new reason for processing:

```

ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

if (errno == ICALL_ERRNO_SUCCESS)
{
...

```

When an event or other stimulus occurs and is processed, the task waits for the semaphore and remains in a blocked state until there is another reason to process. Figure 4-3 shows this flow.

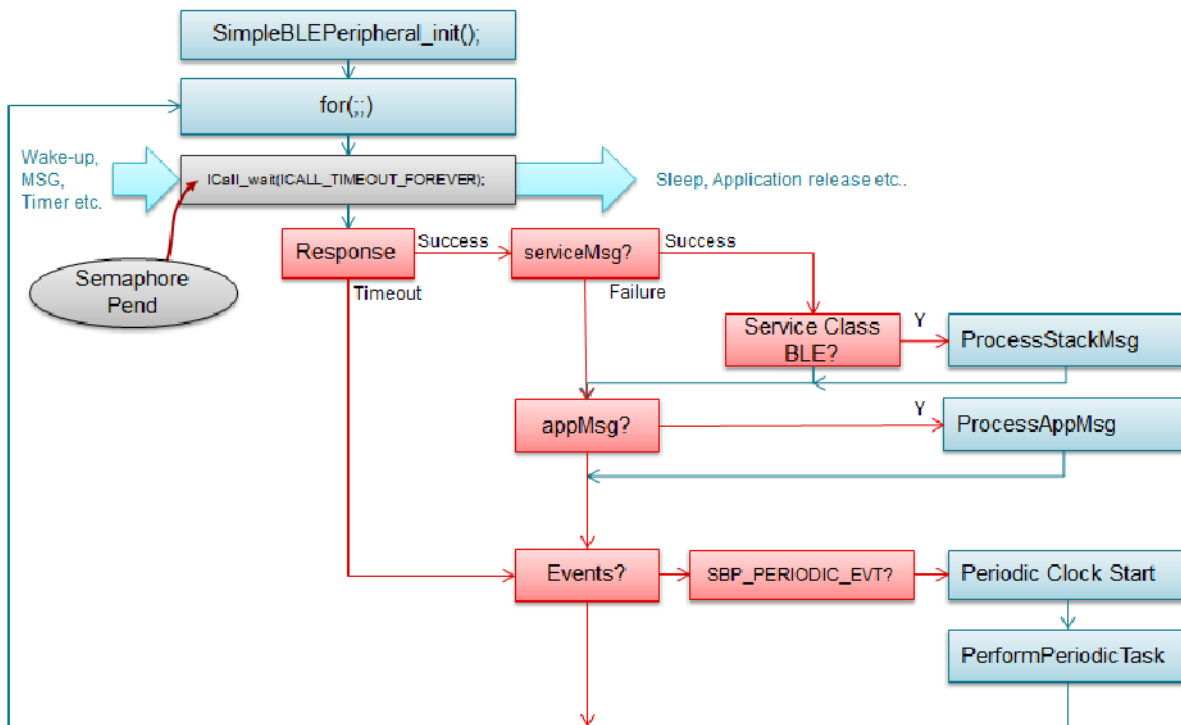


Figure 4-3. SBP Task Flow Chart

As shown in Figure 4-3, various reasons cause the semaphore to be posted to and the task to become active to process.

4.3.2.1 Task Events

Task events are set when the Bluetooth low energy protocol stack sets an event in the application task through ICall. An example of a task event is when the HCI_EXT_ConnEventNoticeCmd() is called (see [Section H.1](#)) to indicate the end of a connection event. An example of a task event that signals the end of a connection event is shown in the task function of the simple_peripheral:

```
if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
{
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
        ICall_Event *pEvt = (ICall_Event *)pMsg;

        // Check for BLE stack events first
        if (pEvt->signature == 0xffff)
        {
            if (pEvt->event_flag & SBP_CONN_EVT_END_EVT)
            {
                // Try to retransmit pending ATT response (if any)
                SimpleBLEPeripheral_sendATTRsp();
            }
            ...
        }

        if (pMsg)
        {
            ICall_freeMsg(pMsg);
        }
    }
}
```

NOTE: In the code, the pEvt->signature is always equal to 0xFFFF if the event is coming from the Bluetooth low energy protocol stack.

When selecting an event value for an intertask event, the value must be unique for the given task and must be a power of 2 (so only 1 bit is set). Because the pEvt->event variable is initialized as uint16_t, this initialization allows for a maximum of 16 events. The only event values that cannot be used are those already used for Bluetooth low energy OSAL global events (stated in bcomdef.h):

```
/******
 * BLE OSAL GAP GLOBAL Events
 */
#define GAP_EVENT_SIGN_COUNTER_CHANGED 0x4000 //!< The device level sign counter changed
```

NOTE: These intertask events are a different set of events than the intratask events mentioned in [Section 4.3.2.4](#).

4.3.2.2 Intertask Messages

These messages are passed from another task (such as the Bluetooth low energy protocol stack) through ICall to the application task. Some possible examples are as follows:

- A confirmation sent from the protocol stack in acknowledgment of a successful over-the-air indication
- An event corresponding to an HCI command (see [Section 5.7](#))
- A response to a GATT client operation (See [Section 5.3.3.1](#))

The following is an example of this from the main task loop of the simple_peripheral.

```

if (ICall_fetchServiceMsg(&src, &dest,
                        (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
{
    uint8 safeToDealloc = TRUE;

    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
        ICall_Event *pEvt = (ICall_Event *)pMsg;
        ...
        else
        {
            // Process inter-task message
            safeToDealloc = SimpleBLEPeripheral_processStackMsg((ICall_Hdr *)pMsg);
        }
    }

    if (pMsg && safeToDealloc)
    {
        ICall_freeMsg(pMsg);
    }
}

```

4.3.2.3 Messages Posted to the RTOS Queue of the Application Task

These messages have been enqueued using the simple_peripheral_enqueueMsg() function. Because these messages are posted to a queue, they are processed in the order in which they occurred. A common example of this is an event received in a callback function (see [Section 5.3.4.2.4](#)).

```

// If RTOS queue is not empty, process app message.

if (!Queue_empty(appMsgQueue))
{
    sbpEvt_t *pMsg = (sbpEvt_t *)Util_dequeueMsg(appMsgQueue);
    if (pMsg)
    {
        // Process message.
        SimpleBLEPeripheral_processAppMsg(pMsg);

        // Free the space from the message.
        ICall_free(pMsg);
    }
}

```

4.3.2.4 Events Signaled Through the Internal Event Variable

These asynchronous events are signaled to the application task for processing by setting the appropriate bit in the events variable of the application task, where each bit corresponds to a defined event.

```
// Internal Events for RTOS application
#define SBP_STATE_CHANGE_EVT      0x0001
#define SBP_CHAR_CHANGE_EVT      0x0002
#define SBP_PERIODIC_EVT         0x0004
```

The function that sets this bit in the events variable must also post to the semaphore to wake up the application for processing. An example of this process is the clock handler that handles clock timeouts (see [Section 3.4.2](#)). The following is an example of processing the periodic event from the main task function of simple_peripheral:

```
if (events & SBP_PERIODIC_EVT)
{
    events &= ~SBP_PERIODIC_EVT;

    Util_startClock(&periodicClock);

    // Perform periodic application task
    SimpleBLEPeripheral_performPeriodicTask();
}
```

NOTE: When adding an event, the event must be unique for the given task and must be a power of 2 (so that only one bit is set). Because the events variable is initialized as `uint16_t`, this initialization allows for a maximum of 16 internal events.

4.3.2.5 Events Signaled Using TI-RTOS Events Module

While most of the sample applications in the SDK use an internal event variable, as described in [Section 4.3.2.4](#), some use the TI-RTOS event module instead. The `simple_np` and `simple_ap` sample applications have adopted the TI-RTOS event method. The internal event variable approach described in [Section 4.3.2.4](#) and the TI-RTOS event module approach described in this section are functionally equivalent; the user should adopt one convention and use it throughout the application. The TI-RTOS event offers an added level of simplicity, as it handles some of the event processing for the application.

To get started using events, the application must include the event module from the RTOS.

```
#include <ti/sysbios/knl/Event.h>
```

Similar to the bitfield approach adopted by the previous section, the application can define custom events to be registered with the RTOS; instead of manually assigning the bitfields, the predefined RTOS events are used. For ARM devices such as the CC2640, up to 32 events can be used. The code snippet below shows the events defined in `simple_ap`.

```
#define AP_NONE                Event_Id_NONE    // No Event
#define AP_EVT_PUI             Event_Id_00     // Power-Up Indication
#define AP_EVT_ADV_ENB        Event_Id_01     // Advertisement Enable
#define AP_EVT_ADV_END        Event_Id_02     // Advertisement Ended
#define AP_EVT_CONN_EST       Event_Id_03     // Connection Established
#define AP_EVT_CONN_TERM      Event_Id_04     // Connection Terminated
#define AP_EVT_AUTHENTICATION Event_Id_05     // Authentication IO
#define AP_EVT_SECURITY        Event_Id_06     // Security State event
#define AP_EVT_START_PERIODIC_CLOCK Event_Id_07 // Start the periodic
#define AP_EVT_BUTTON_SELECT  Event_Id_24     // SELECT Button Press
#define AP_EVT_BUTTON_UP      Event_Id_25     // UP Button Press
#define AP_EVT_BUTTON_DOWN    Event_Id_26     // DOWN Button Press
#define AP_EVT_BUTTON_LEFT    Event_Id_27     // LEFT Button Press
#define AP_EVT_BUTTON_RIGHT   Event_Id_28     // RIGHT Button Press
#define AP_ERROR               Event_Id_29     // Error
```

Similar to the semaphore approach, the application can pend on events, allowing execution of the task to block until the event occurs. See the code snippet from the `simple_ap` below.

```
flags = Event_pend(event, Event_Id_NONE, orFlags + andFlags + SNP_ALL_EVENTS, timeout);
```

Using the andFlags and orFlags of the event module, the user can configure the pend call to wait until a specific combination of events occurs. When a certain event occurs, the application can post the event, such as the below example from simple_ap.

```
Event_post(apEvent, AP_EVT_BUTTON_LEFT);
```

Further documentation on the event module can be found in the TI-RTOS kernel documentation, located at \$TI_RTOS_INSTALL\$/docs.

4.3.3 Callbacks

The application code also includes various callbacks to protocol stack layers, profiles, and RTOS modules. To ensure thread safety, processing must be minimized in the actual callback and the bulk of the processing should occur in the application context. Two functions are defined per callback (consider the GAPRole state change callback):

NOTE: No blocking RTOS function calls or protocol stack APIs should be performed in a callback function. Such function calls may result in an abort or undefined behavior. Always perform protocol stack and RTOS blocking calls from the application task context.

- **The actual callback:** This function is called in the context of the calling task or module (for example, the GAPRole task). To minimize processing in the calling context, this function should enqueue an event to the queue of the application for processing.

```
static void SimpleBLEPeripheral_stateChangeCB(gaprole_States_t newState)
{
    SimpleBLEPeripheral_enqueueMsg(SBP_STATE_CHANGE_EVT, newState);
}
```

- **The function to process in the application context:** When the application wakes up due to the enqueue from the callback, this function is called when the event is popped from the application queue and processed.

```
static void SimpleBLEPeripheral_processStateChangeEvt(gaprole_States_t newState)
{
    ...
}
```

See [Section 5.2.1](#) for a flow diagram of this process.

The Bluetooth low energy Protocol Stack

This section describes the functionality of the Bluetooth low energy protocol stack and provides a list of APIs to interface with the protocol stack. The stack project and its associated files serve to implement the Bluetooth low energy protocol stack task. This is the highest priority task in the system and it implements the Bluetooth low energy protocol stack as shown in [Figure 1-1](#).

Most of the Bluetooth low energy protocol stack is object code in a single library file (TI does not provide the protocol stack source code as a matter of policy). A developer must understand the functionality of the various protocol stack layers and how they interact with the application and profiles. This section explains these layers.

5.1 Generic Access Profile (GAP)

The GAP layer of the Bluetooth low energy protocol stack is responsible for connection functionality. This layer handles the access modes and procedures of the device including device discovery, link establishment, link termination, initiation of security features, and device configuration. See [Figure 5-1](#) for more details.

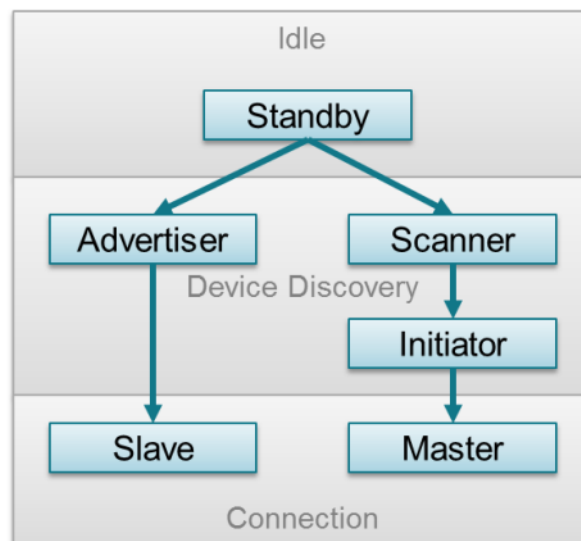


Figure 5-1. GAP State Diagram

Based on the role for which the device is configured, [Figure 5-1](#) shows the states of the device. The following describes these states.

- **Standby:** The device is in the initial idle state upon reset.
- **Advertiser:** The device is advertising with specific data letting any initiating devices know that it is a connectable device (this advertisement contains the device address and can contain some additional data such as the device name).
- **Scanner:** When receiving the advertisement, the scanning device sends a scan request to the advertiser. The advertiser responds with a scan response. This process is called device discovery. The scanning device is aware of the advertising device and can initiate a connection with it.
- **Initiator:** When initiating, the initiator must specify a peer device address to which to connect. If an advertisement is received matching that address of the peer device, the initiating device then sends

out a request to establish a connection (link) with the advertising device with the connection parameters described in [Section 5.1.1](#).

- **Slave/Master:** When a connection is formed, the device functions as a slave if the advertiser and a master if the initiator.

5.1.1 Connection Parameters

This section describes the connection parameters which are sent by the initiating device with the connection request and can be modified by either device when the connection is established. These parameters are as follows:

- **Connection Interval** – In Bluetooth low energy connections, a frequency-hopping scheme is used. The two devices each send and receive data from one another only on a specific channel at a specific time. These devices meet a specific amount of time later at a new channel (the link layer of the Bluetooth low energy protocol stack handles the channel switching). This meeting is where the two devices send and receive data is known as a *connection event*. If there is no application data to be sent or received, the two devices exchange link layer data to maintain the connection. The connection interval is the amount of time between two connection events in units of 1.25 ms. The connection interval can range from a minimum value of 6 (7.5 ms) to a maximum of 3200 (4.0 s). See [Figure 5-2](#) for more details.

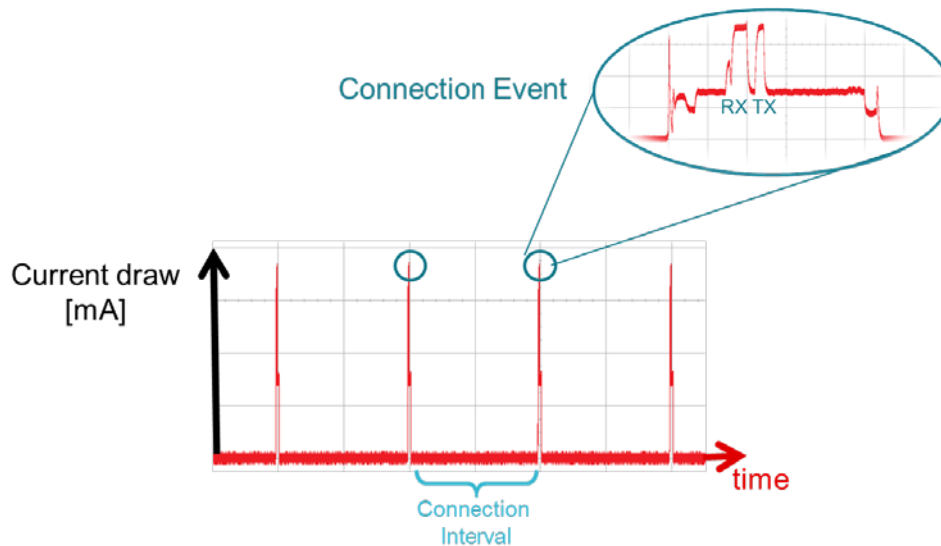


Figure 5-2. Connection Event and Interval

Different applications may require different connection intervals. As described in [Section 5.1.3](#), these requirements affect the power consumption of the device. For more detailed information on power consumption, see [Measuring Bluetooth Smart Power Consumption Application Report \(SWRA478\)](#).

- Slave Latency** – This parameter gives the slave (peripheral) device the option of skipping a number of connection events. This ability gives the peripheral device some flexibility. If the peripheral does not have any data to send, it can skip connection events, stay asleep, and save power. The peripheral device selects whether to wake or not on a per connection event basis. The peripheral can skip connection events but must not skip more than allowed by the slave latency parameter or the connection fails. See [Figure 5-3](#) for more details.

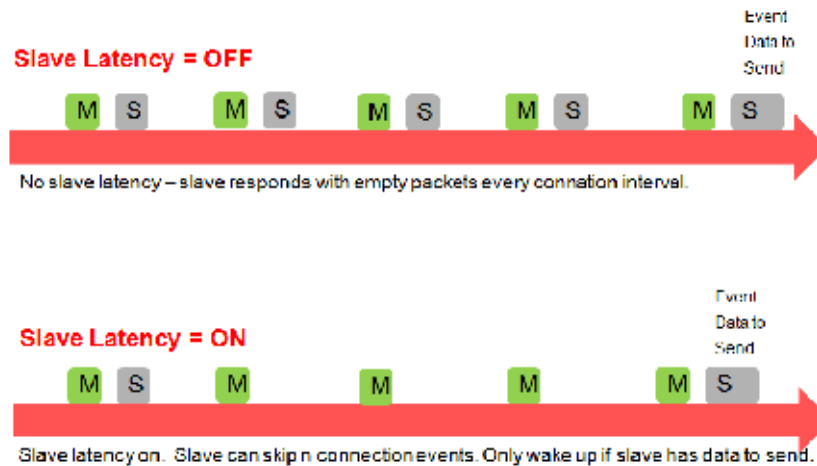


Figure 5-3. Slave Latency

- Supervision Time-out** – This time-out is the maximum amount of time between two successful connection events. If this time passes without a successful connection event, the device terminates the connection and returns to an unconnected state. This parameter value is represented in units of 10 ms. The supervision time-out value can range from a minimum of 10 (100 ms) to 3200 (32.0 s). The time-out must be larger than the effective connection interval (see [Section 5.1.2](#) for more details).

5.1.2 Effective Connection Interval

The effective connection interval is equal to the amount of time between two connection events, assuming that the slave skips the maximum number of possible events if slave latency is allowed (the effective connection interval is equal to the actual connection interval if slave latency is set to 0).

The slave latency value represents the maximum number of events that can be skipped. This number can range from a minimum value of 0 (meaning that no connection events can be skipped) to a maximum of 499. The maximum value must not make the effective connection interval (see the following formula) greater than 16 s. The interval can be calculated using the following formula:

$$\text{Effective Connection Interval} = (\text{Connection Interval}) \times (1 + [\text{Slave Latency}])$$

Consider the following example:

- Connection Interval: 80 (100 ms)
- Slave Latency: 4
- Effective Connection Interval: $(100 \text{ ms}) \times (1 + 4) = 500 \text{ ms}$

When no data is being sent from the slave to the master, the slave transmits during a connection event once every 500 ms.

5.1.3 Connection Parameter Considerations

In many applications, the slave skips the maximum number of connection events. Consider the effective connection interval when selecting or requesting connection parameters. Selecting the correct group of connection parameters plays an important role in power optimization of the Bluetooth low energy device. The following list gives a general summary of the trade-offs in connection parameter settings.

Reducing the connection interval does as follows:

- Increases the power consumption for both devices
- Increases the throughput in both directions
- Reduces the time for sending data in either direction

Increasing the connection interval does as follows:

- Reduces the power consumption for both devices
- Reduces the throughput in both directions
- Increases the time for sending data in either direction

Reducing the slave latency (or setting it to zero) does as follows:

- Increases the power consumption for the peripheral device
- Reduces the time for the peripheral device to receive the data sent from a central device

Increasing the slave latency does as follows:

- Reduces power consumption for the peripheral during periods when the peripheral has no data to send to the central device
- Increases the time for the peripheral device to receive the data sent from the central device

5.1.4 Connection Parameter Limitations with Multiple Connections

There are additional constraints that exist when connected to multiple devices or performing multiple GAP roles simultaneously. See the MultiRole example in for an example of this.

5.1.5 Connection Parameter Update

In some cases, the central device requests a connection with a peripheral device containing connection parameters that are unfavorable to the peripheral device. In other cases, a peripheral device might have the desire to change parameters in the middle of a connection, based on the peripheral application. The peripheral device can request the central device to change the connection settings by sending a Connection Parameter Update Request. For Bluetooth 4.1-capable devices, this request is handled directly by the Link Layer. For Bluetooth 4.0 devices, the L2CAP layer of the protocol stack handles the request. The Bluetooth low energy stack automatically selects the update method.

This request contains four parameters: minimum connection interval, maximum connection interval, slave latency, and time-out. These values represent the parameters that the peripheral device needs for the connection (the connection interval is given as a range). When the central device receives this request, it can accept or reject the new parameters.

Sending a Connection Parameter Update Request is optional and is not required for the central device to accept or apply the requested parameters. Some applications try to establish a connection at a faster connection interval to allow for a faster service discovery and initial setup. These applications later request a longer (slower) connection interval to allow for optimal power usage.

Depending on the GAPRole, connection parameter updates can be sent asynchronously with the GAPRole_SendUpdateParam() or GAPCentralRole_UpdateLink() command. See the API in [Section B.1](#) and [Section C.1](#), respectively. The peripheral GAPRole can be configured to automatically send a parameter update a certain amount of time after establishing a connection. For example, the simple_peripheral application uses the following preprocessor-defined symbols:

```
#define DEFAULT_ENABLE_UPDATE_REQUEST      TRUE
#define DEFAULT_DESIRED_MIN_CONN_INTERVAL  80
#define DEFAULT_DESIRED_MAX_CONN_INTERVAL  800
#define DEFAULT_DESIRED_SLAVE_LATENCY     0
#define DEFAULT_DESIRED_CONN_TIMEOUT      1000
```

```
#define DEFAULT_CONN_PAUSE_PERIPHERAL 6
```

Six seconds after a connection is established, the GAP layer automatically sends a connection parameter update. See [Section 5.2.1](#) for an explanation of how the parameters are configured, and [Section B.2](#) for a more detailed description of these parameters. This action can be disabled by setting `DEFAULT_ENABLE_UPDATE_REQUEST` to `FALSE`.

5.1.6 Connection Termination

Either the master or the slave can terminate a connection for any reason. One side initiates termination and the other side must respond before both devices exit the connected state.

5.1.7 GAP Abstraction

The application and profiles can directly call GAP API functions to perform Bluetooth low energy-related functions such as advertising or connecting. Most of the GAP functionality is handled by the GAPRole Task. [Figure 5-4](#) shows this abstraction hierarchy.

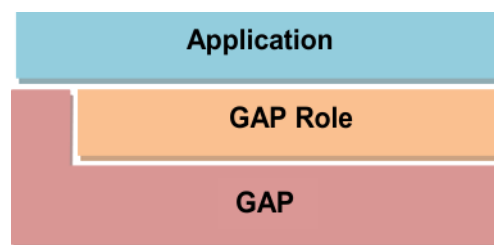


Figure 5-4. GAP Abstraction

Access the GAP layer through direct calls or through the GAPRole task as described in [Section 5.2](#). Use the GAPRole task rather than direct calls when possible. [Section 5.1.8](#) describes the functions and parameters that are not handled or configured through the GAPRole task and must be modified directly through the GAP layer.

5.1.8 Configuring the GAP Layer

The GAP layer functionality is mostly defined in library code. The function headers can be found in `gap.h` in the protocol stack project. Most of these functions are used by the GAPRole and do not need to be called directly. For reference, the GAP API is defined in [Appendix D](#). Several parameters exist which may be desirable to modify before starting the GAPRole. These parameters can be set or get through the `GAP_SetParamValue()` and `GAP_GetParamValue()` functions and include advertising and scanning intervals, windows, and so forth (see the API for more information). The following is the configuration of the GAP layer in `simple_peripheral_init()`:

```
// Set advertising interval
{
    uint16_t advInt = DEFAULT_ADVERTISING_INTERVAL;

    GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MIN, advInt);
    GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MAX, advInt);
    GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MIN, advInt);
    GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MAX, advInt);
}
```

5.2 GAPRole Task

The GAPRole task is a separate task which offloads the application by handling most of the GAP layer functionality. This task is enabled and configured by the application during initialization. Based on this configuration, many Bluetooth low energy protocol stack events are handled directly by the GAPRole task and never passed to the application. Callbacks exist that the application can register with the GAPRole task so that the application task can be notified of certain events and proceed accordingly.

Based on the configuration of the device, the GAP layer always operates in one of four roles:

- Broadcaster – The advertiser is nonconnectable.
- Observer – The device scans for advertisements but cannot initiate connections.
- Peripheral – The advertiser is connectable and operates as a slave in a single link-layer connection.
- Central – The device scans for advertisements and initiates connections and operates as a master in a single or multiple link-layer connections. The Bluetooth low energy central protocol stack supports up to three simultaneous connections.

The Bluetooth low energy specification allows for certain combinations of multiple-roles, which are supported by the Bluetooth low energy protocol stack. For configuration of the Bluetooth low energy stack features, see [Section 10.4](#).

5.2.1 Peripheral Role

The peripheral GAPRole task is defined in `peripheral.c` and `peripheral.h`. [Section A.1](#) describes the full API including commands, configurable parameters, events, and callbacks. The steps to use this module are as follows:

1. Initialize the GAPRole parameters (see [Appendix B](#)). This initialization should occur in the application initialization function (that is `simple_peripheral_init()`).

```
// Setup the GAP Peripheral Role Profile
{
    uint8_t initialAdvertEnable = TRUE;

    uint16_t advertOffTime = 0;

    uint8_t enableUpdateRequest = DEFAULT_ENABLE_UPDATE_REQUEST;
    uint16_t desiredMinInterval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
    uint16_t desiredMaxInterval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
    uint16_t desiredSlaveLatency = DEFAULT_DESIRED_SLAVE_LATENCY;
    uint16_t desiredConnTimeout = DEFAULT_DESIRED_CONN_TIMEOUT;

    // Set the GAP Role Parameters
    GAPRole_setParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t), &initialAdvertEnable);
    GAPRole_setParameter(GAPROLE_ADVERT_OFF_TIME, sizeof(uint16_t), &advertOffTime);
    GAPRole_setParameter(GAPROLE_SCAN_RSP_DATA, sizeof(scanRspData), scanRspData);
    GAPRole_setParameter(GAPROLE_ADVERT_DATA, sizeof(advertData), advertData);
    GAPRole_setParameter(GAPROLE_PARAM_UPDATE_ENABLE, sizeof(uint8_t), &enableUpdateRequest);
    GAPRole_setParameter(GAPROLE_MIN_CONN_INTERVAL, sizeof(uint16_t), &desiredMinInterval);
    GAPRole_setParameter(GAPROLE_MAX_CONN_INTERVAL, sizeof(uint16_t), &desiredMaxInterval);
    GAPRole_setParameter(GAPROLE_SLAVE_LATENCY, sizeof(uint16_t), &desiredSlaveLatency);
    GAPRole_setParameter(GAPROLE_TIMEOUT_MULTIPLIER, sizeof(uint16_t), &desiredConnTimeout);
}
```

2. Initialize the GAPRole task and pass application callback functions to GAPRole (see [Section B.3](#)). This should also occur in the application initialization function.

```
// Start the Device
VOID GAPRole_StartDevice(&SimpleBLEPeripheral_gapRoleCBs);
```

3. Send GAPRole commands from the application. [Figure 5-5](#) is an example of the application using `GAPRole_TerminateConnection()`. Green corresponds to the app context and red corresponds to the protocol stack context.

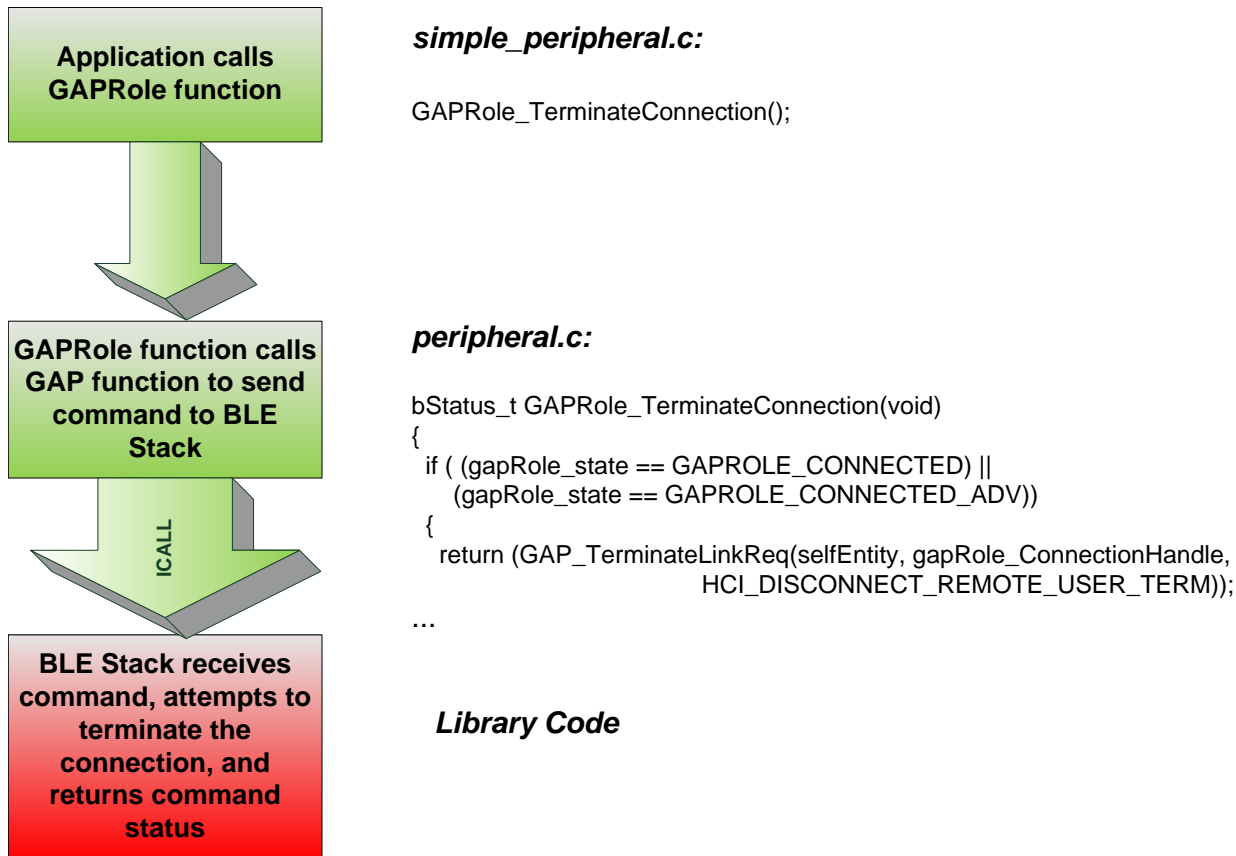
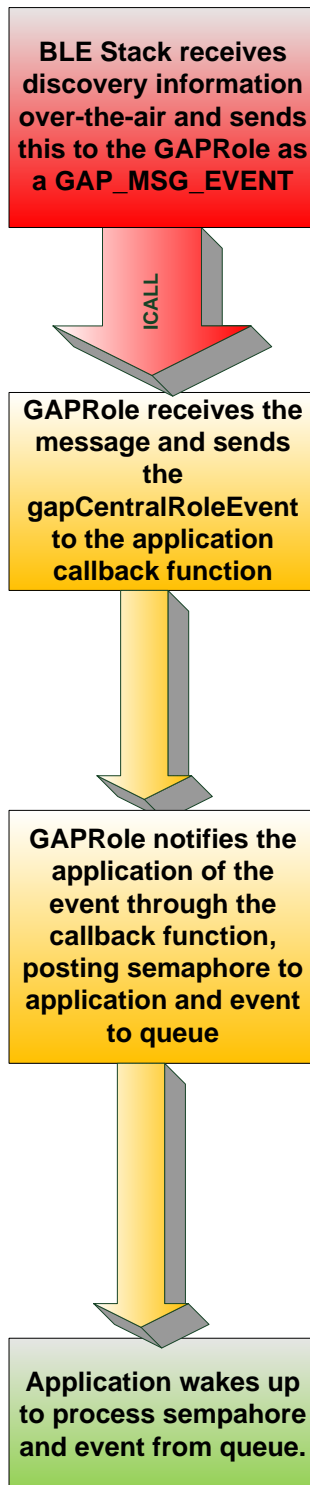


Figure 5-5. Application Using GAPRole_TerminateConnection()

NOTE: The return value only indicates whether the attempt to terminate the connection initiated successfully. The actual termination of connection event is returned asynchronously and is passed to the application through a callback. The API in [Section B.3](#) lists the return parameters for each command and associated callback function events.

- The GAPRole task processes most of the GAP-related events passed to it from the Bluetooth low energy protocol stack. The GAPRole task also forwards some events to the application. [Figure 5-6](#) is an example tracing the GAP_LINK_TERMINATED_EVENT from the Bluetooth low energy protocol stack to the application. Green corresponds to the app context. Orange corresponds to the GAPRole context. Red corresponds to the protocol stack context.)



Library Code

peripheral.c:

```

static uint8_t gapCentralRole_ProcessGAPMsg(gapEventHdr_t
*pMsg)
{
    // Pass event to app
    if (pGapCentralRoleCB && pGapCentralRoleCB->eventCB)
    {
        return (pGapCentralRoleCB->eventCB((gapCentralRoleEvent_t
*)pMsg));
    }
}
    
```

...

peripheral.c:

```

// Notify the application with the new state change
if (pGapRoles_AppCGs && pGapRoles_AppCGs->pfnStateChange)
{
    pGapRoles_AppCGs->pfnStateChange(gapRole_state);
}
    
```

simple_peripheral.c:

```

static void SimpleBLEPeripheral_stateChangeCB(gaprole_States_t
newState)
{
    SimpleBLEPeripheral_enqueueMsg(SBP_STATE_CHANGE_EVT,
newState);
}
    
```

simple_peripheral.c:

```

static void SimpleBLEPeripheral_processAppMsg(sbpEvt_t *pMsg)
{
    switch (pMsg->hdr.event)
    {
        case SBP_STATE_CHANGE_EVT:
            SimpleBLEPeripheral_processStateChangeEvt((gaprole_States_t)
pMsg->hdr.state);
    }
}
    
```

...

```

static void impleBLEPeripheral_processStateChangeEvt(gaprole_States_t
newState)
    
```

```

{
    switch ( newState )
    {
        case GAPROLE_WAITING:
    }
}
    
```

Figure 5-6. Tracing the GAP_LINK_TERMINATED_EVENT

5.2.2 Central Role

The central GAPRole task is defined in central.c and central.h. [Appendix C](#) describes the full API including commands, configurable parameters, events, and callbacks. See the SimpleBLECentral project for an example of implementing the central GAPRole. The steps to use this module are as follows.

1. Initialize the GAPRole parameters. [Appendix C](#) defines these parameters. This initialization should occur in the application initialization function (that is, SimpleBLECentral_init()).

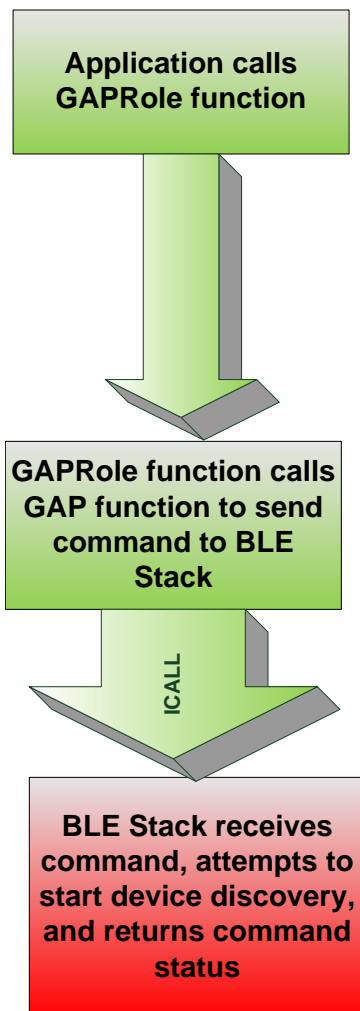
```
{
    uint8_t scanRes = DEFAULT_MAX_SCAN_RES;

    GAPCentralRole_SetParameter(GAPCENTRALROLE_MAX_SCAN_RES, sizeof(uint8_t), &scanRes);
}
```

2. Start the GAPRole task. This involves passing function pointers to application callback function to the central GAPRole. See [Section C.3](#) for a detailed description of the callbacks. This should also occur in the application initialization function.

```
VOID GAPCentralRole_StartDevice(&SimpleBLECentral_roleCB);
```

3. Send GAPRole commands from the application. [Figure 5-7](#) is an example of the application using GAPCentralRole_StartDiscovery(). Green corresponds to the app context and red corresponds to the Bluetooth low energy protocol stack context.



simple_central.c:

```
GAPCentralRole_StartDiscovery(DEFAULT_DISCOVERY_MODE,
    DEFAULT_DISCOVERY_ACTIVE_SCAN,
    DEFAULT_DISCOVERY_WHITE_LIST);
```

central.c:

```
bStatus_t GAPCentralRole_StartDiscovery(uint8_t mode, uint8_t
activeScan, uint8_t whiteList)
{
    gapDevDiscReq_t params;

    params.taskID = Call_getLocalMsgEntityId(
        ICALL_SERVICE_CLASS_BLE_MSG, selfEntity);
    params.mode = mode;
    params.activeScan = activeScan;
    params.whiteList = whiteList;

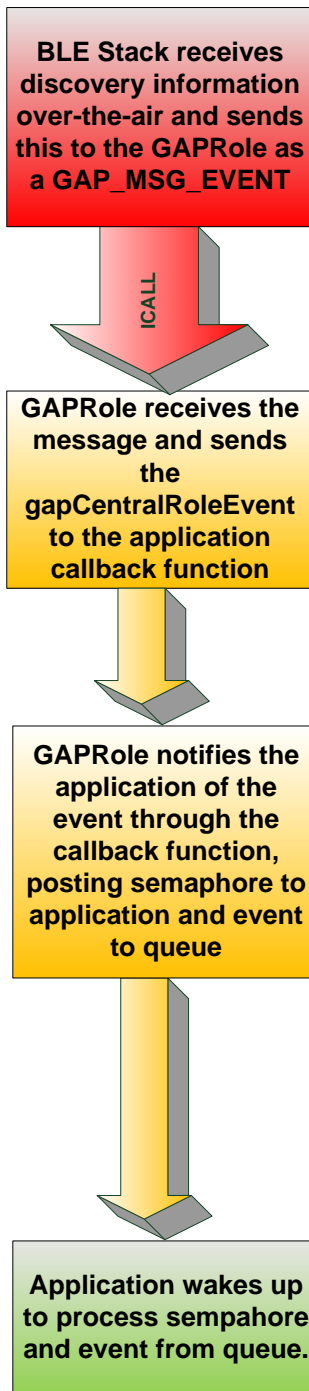
    return GAP_DeviceDiscoveryRequest(&params);
}
```

Library Code

Figure 5-7. Application Using GAPCentralRole_StartDiscovery()

NOTE: The return value from the protocol stack indicates only whether or not the attempt to perform device discovery was initiated. The actual termination of connection event is returned asynchronously as a GAP event forwarded through the central GAPRole callbacks as described below. [Section C.1](#) lists the return parameters for each command and associated callback events.

4. The GAPRole task performs some processing on the GAP events it receives from the protocol stack. The task also forwards some events to the application. [Figure 5-8](#) is an example tracing the GAP_DEVICE_DISCOVERY_EVENT, prompted by the command from step 3, from the protocol stack to the application. Green corresponds to the app context. Orange corresponds to the GAPRole context. Red corresponds to the protocol stack context.



Library Code

central.c:

```

static uint8_t gapCentralRole_ProcessGAPMsg(gapEventHdr_t *pMsg)
{
    // Pass event to app
    if (pGapCentralRoleCB && pGapCentralRoleCB->eventCB)
    {
        return (pGapCentralRoleCB->eventCB((gapCentralRoleEvent_t *)pMsg));
    }
    ...
}
  
```

simple_central.c:

```

static uint8_t SimpleBLECentral_eventCB(gapCentralRoleEvent_t *pEvent)
{
    // Forward the role event to the application
    if (SimpleBLECentral_enqueueMsg(SBC_STATE_CHANGE_EVT, SUCCESS, (uint8_t *)pEvent))
    {
        ...
    }
}
  
```

simple_central.c:

```

static void SimpleBLECentral_processAppMsg(sbcEvt_t *pMsg)
{
    switch (pMsg->hdr.event)
    {
        case SBC_STATE_CHANGE_EVT:
            SimpleBLECentral_processStackMsg((ICall_Hdr *)pMsg->pData);
            ...
    }
}

static void SimpleBLECentral_processStackMsg(ICall_Hdr *pMsg)
{
    switch (pMsg->event)
    {
        case GAP_MSG_EVENT:
            SimpleBLECentral_processRoleEvent((gapCentralRoleEvent_t *)pMsg);
            ...
    }
}

static void SimpleBLECentral_processRoleEvent(gapCentralRoleEvent_t *pEvent)
{
    switch (pEvent->gap.opcode)
    {
        case GAP_DEVICE_INFO_EVENT:
            ...
    }
}
  
```

Figure 5-8. Tracing the GAP_DEVICE_DISCOVERY_EVENT

5.3 Generic Attribute Profile (GATT)

Just as the GAP layer handles most connection-related functionality, the GATT layer of the Bluetooth low energy protocol stack is used by the application for data communication between two connected devices. Data is passed and stored in the form of characteristics which are stored in memory on the Bluetooth low energy device. From a GATT standpoint, when two devices are connected they are each in one of two roles.

- The **GATT server** is the device containing the characteristic database that is being read or written by a GATT client.
- The **GATT client** is the device that is reading or writing data from or to the GATT server.

Figure 5-9 shows this relationship in a sample Bluetooth low energy connection where the peripheral device (that is, a CC2650 LaunchPad) is the GATT server and the central device (that is, a smart phone) is the GATT client.

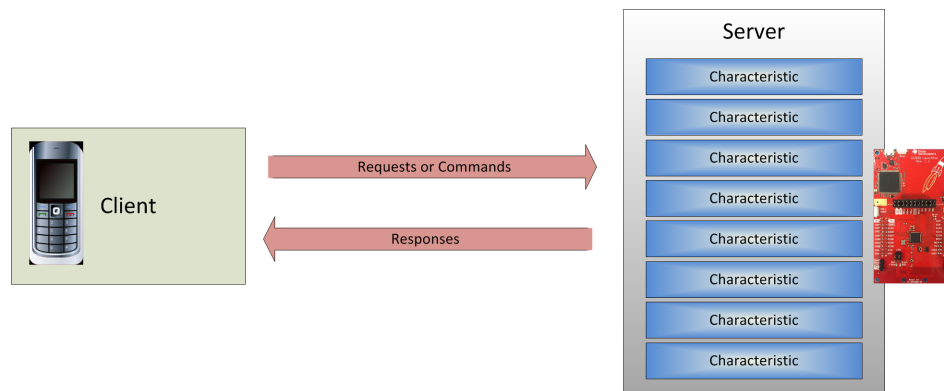


Figure 5-9. GATT Client and Server

The GATT roles of client and server are independent from the GAP roles of peripheral and central. A peripheral can be either a GATT client or a GATT server, and a central can be either a GATT client or a GATT server. A peripheral can act as both a GATT client and a GATT server. For a hands-on review of GATT services and characteristics, see [SimpleLink Academy Module 1](#).

5.3.1 GATT Characteristics and Attributes

While characteristics and attributes are sometimes used interchangeably when referring to Bluetooth low energy, consider characteristics as groups of information called attributes. Attributes are the information actually transferred between devices. Characteristics organize and use attributes as data values, properties, and configuration information. A typical characteristic is composed of the following attributes.

- **Characteristic Value:** data value of the characteristic
- **Characteristic Declaration:** descriptor storing the properties, location, and type of the characteristic value
- **Client Characteristic Configuration:** a configuration that allows the GATT server to configure the characteristic to be notified (send message asynchronously) or indicated (send message asynchronously with acknowledgment)
- **Characteristic User Description:** an ASCII string describing the characteristic

These attributes are stored in the GATT server in an attribute table. In addition to the value, the following properties are associated with each attribute.

- **Handle:** the index of the attribute in the table (Every attribute has a unique handle.)
- **Type:** indicates what the attribute data represents (referred to as a UUID [universal unique identifier]. Some of these are Bluetooth SIG-defined and some are custom.)
- **Permissions:** enforces if and how a GATT client device can access the value of an attribute

5.3.2 GATT Services and Profile

A GATT service is a collection of characteristics. For example, the heart rate service contains a heart rate measurement characteristic and a body location characteristic, among others. Multiple services can be grouped together to form a profile. Many profiles only implement one service so the two terms are sometimes used interchangeably. There are four GATT profiles for the simple_peripheral application.

- **GAP GATT Service:** This service contains device and access information such as the device name, vendor identification, and product identification. See [Section 5.3.2.1](#).

The following characteristics are defined for this service:

- Device name
- Appearance
- Peripheral preferred connection parameters

See Vol. 3 Part C, Ch. 12 in the Bluetooth low energy specification for more information on these characteristics.

- **Generic Attribute Service:** This service contains information about the GATT server, is a part of the Bluetooth low energy protocol stack, and is required for every GATT server device as per the Bluetooth low energy specification.
- **Device Info Service:** This service exposes information about the device such as the hardware, software version, firmware version, regulatory information, compliance information, and manufacturer name. The Device Info Service is part of the Bluetooth low energy protocol stack and configured by the application.
- **simple_gatt_profile Service:** This service is a sample profile for testing and for demonstration. The full source code is provided in the simple_gatt_profile.c and simple_gatt_profile.h files.

[Figure 5-10](#) shows the portion of the attribute table in the simple_peripheral project corresponding to the simple_gatt_profile service. TI intends this section as an introduction to the attribute table. For information on how this profile is implemented in the code, see [Section 5.3.4.2](#).

| Handle | Uuid | Uuid Description | Value | Properties |
|--------|--------|-------------------------------------|------------------|------------|
| 0x001F | 0x2800 | GATT Primary Service Declaration | 00:FF | |
| 0x0020 | 0x2803 | GATT Characteristic Declaration | 0A:21:00:F1:FF | |
| 0x0021 | 0xFFF1 | Simple Profile Char 1 | 01 | Rd Wr 0x0A |
| 0x0022 | 0x2901 | Characteristic User Description | Characteristic 1 | |
| 0x0023 | 0x2803 | GATT Characteristic Declaration | 02:24:00:F2:FF | |
| 0x0024 | 0xFFF2 | Simple Profile Char 2 | 02 | Rd 0x02 |
| 0x0025 | 0x2901 | Characteristic User Description | Characteristic 2 | |
| 0x0026 | 0x2803 | GATT Characteristic Declaration | 08:27:00:F3:FF | |
| 0x0027 | 0xFFF3 | Simple Profile Char 3 | | Wr 0x08 |
| 0x0028 | 0x2901 | Characteristic User Description | Characteristic 3 | |
| 0x0029 | 0x2803 | GATT Characteristic Declaration | 10:2A:00:F4:FF | |
| 0x002A | 0xFFF4 | Simple Profile Char 4 | | Nfy 0x10 |
| 0x002B | 0x2902 | Client Characteristic Configuration | 00:00 | |
| 0x002C | 0x2901 | Characteristic User Description | Characteristic 4 | |
| 0x002D | 0x2803 | GATT Characteristic Declaration | 02:2E:00:F5:FF | |
| 0x002E | 0xFFF5 | Simple Profile Char 5 | | Rd 0x02 |
| 0x002F | 0x2901 | Characteristic User Description | Characteristic 5 | |

Figure 5-10. Simple GATT Profile Characteristic Table from BTool

The simple_gatt_profile contains the following characteristics:

- **SIMPLEPROFILE_CHAR1** – a 1-byte value that can be read or written from a GATT client device
- **SIMPLEPROFILE_CHAR2** – a 1-byte value that can be read from a GATT client device but cannot be written
- **SIMPLEPROFILE_CHAR3** – a 1-byte value that can be written from a GATT client device but cannot be read

- **SIMPLEPROFILE_CHAR4** – a 1-byte value that cannot be directly read or written from a GATT client device (This value is notifiable: This value can be configured for notifications to be sent to a GATT client device.)
- **SIMPLEPROFILE_CHAR5** – a 5-byte value that can be read (but not written) from a GATT client device

The following is a line-by-line description of the simple profile attribute table, referenced by the following handle.

- 0x001F is the `simple_gatt_profile` service declaration. This declaration has a UUID of 0x2800 (Bluetooth-defined `GATT_PRIMARY_SERVICE_UUID`). The value of this declaration is the UUID of the `simple_gatt_profile` (custom-defined).
- 0x0020 is the `SimpleProfileChar1` characteristic declaration. This declaration can be thought of as a pointer to the `SimpleProfileChar1` value. The declaration has a UUID of 0x2803 (Bluetooth-defined `GATT_CHARACTER_UUID`). The value of the declaration characteristic, as well as all other characteristic declarations, is a 5-byte value explained here (from MSB to LSB):
 - Byte 0: the properties of the `SimpleProfileChar1` as defined in the Bluetooth specification (The following are some of the relevant properties.)
 - 0x02: permits reads of the characteristic value
 - 0x04: permits writes of the characteristic value (without a response)
 - 0x08: permits writes of the characteristic value (with a response)
 - 0x10: permits of notifications of the characteristic value (without acknowledgment)
 - 0x20: permits notifications of the characteristic value (with acknowledgment)

The value of 0x0A means the characteristic is readable (0x02) and writeable (0x08).

 - Bytes 1–2: the byte-reversed handle where the `SimpleProfileChar1` value is (handle 0x0021)
 - Bytes 3–4: the UUID of the `SimpleProfileChar1` value (custom-defined 0xFFF1)
- 0x0021 is the `SimpleProfileChar1` value. This value has a UUID of 0xFFF1 (custom-defined). This value is the actual payload data of the characteristic. As indicated by its characteristic declaration (handle 0x0020), this value is readable and writeable.
- 0x0022 is the `SimpleProfileChar1` user description. This description has a UUID of 0x2901 (Bluetooth-defined). The value of this description is a user-readable string describing the characteristic.
- 0x0023 – 0x002F are attributes that follow the same structure as the `simpleProfileChar1` described previously with regard to the remaining four characteristics. The only different attribute, handle 0x002B, is described as follows.
- 0x002B is the `SimpleProfileChar4` client characteristic configuration. This configuration has a UUID of 0x2902 (Bluetooth-defined). By writing to this attribute, a GATT server can configure the `SimpleProfileChar4` for notifications (writing 0x0001) or indications (writing 0x0002). Writing 0x0000 to this attribute disable notifications and indications.

5.3.2.1 GAP GATT Service

The GAP GATT Service (GGS) is required for low-energy devices that implement the central or peripheral role. Multirole devices that implement either of these roles must also contain the GGS. The purpose of the GGS is to aide in the device discovery and connection initiation process. For more information about the GGS, refer to the *Bluetooth Specification Version 4.2 [Vol 3, Part C] Section 12*.

5.3.2.1.1 Using the GGS

This section describes what the application must do to configure, start, and use the GAP Gatt Service. The GGS is implemented as part of the Bluetooth Low Energy library code, the API can be found in `gapgattserver.h`. [Appendix F](#) describes the full API, including commands, configurable parameters, events, and callbacks.

1. Include headers (found at `$install/src/inc`)

```
#include "gapgattserver.h"
```

2. Initialize GGS parameters.

```
// GAP GATT Attributes
static uint8_t attDeviceName[GAP_DEVICE_NAME_LEN] = "Simple BLE Peripheral";
GGS_SetParameter(GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, attDeviceName);
```

3. Initialize application callbacks with GGS (optional). This notifies the application when any of the characteristics in the GGS have changed.

```
GGS_RegisterAppCBs (&appGGSCBs);
```

4. Add the GGS to the GATT server.

```
bStatus_t GGS_AddService(GATT_ALL_SERVICES);
```

5.3.2.2 Generic Attribute Profile Service

The Generic Attribute Profile (GATT) Service provides information about the GATT services registered with a device. For more information, refer to Bluetooth Specification Version 4.2 [Vol 3, Part G] Section 7.

The service changed characteristic is used to inform bonded devices that services have changed on the server upon reconnection. Service changed updates are sent in the form of GATT indications, and the service changed characteristic is not writeable or readable. In the TI Bluetooth low energy stack, the service changed characteristic is implemented as part of the gattservapp, which is part of library code.

Per the TI Bluetooth low energy stack spec, it is safe for server devices whose characteristic tables do not change over their lifetime to exclude the service changed characteristic. Support for indications from this characteristic must be supported on GATT client devices.

5.3.2.2.1 Using the GATT Service

This section describes what the user must do to enable the GATT service changed feature in the Bluetooth low energy stack. Once the service changed feature is enabled, the GAPBondMgr will handle sending the service changed indication to clients who have enabled it using the CCCD.

1. Use a supported build config for the stack; only stack libraries with v4.1 features and L2CAP connection-oriented channels will support the service changed characteristic.

Enable this feature with the project's buildConfig.opt file, by uncommenting the following line:

```
-DBLE_V41_FEATURES=L2CAP_COC_CFG+V41_CTRL_CFG
```

2. From this point, the GAPBondMgr handles sending an indication to connected clients when the service has changed and the CCCD is enabled. If the feature is enabled, the peripheral role invokes the necessary APIs to send the indication through the GAPBondMgr.
3. On the client side, service changed indications can be registered using the same method as registering for other indications (see [Section 5.3.3.1](#)).

5.3.3 GATT Client Abstraction

Like the GAP layer, the GATT layer is also abstracted. This abstraction depends on whether the device is acting as a GATT client or a GATT server. As defined by the Bluetooth Specification, the GATT layer is an abstraction of the ATT layer.

GATT clients do not have attribute tables or profiles as they are gathering, not serving, information. Most of the interfacing with the GATT layer occurs directly from the application. In this case, use the direct GATT API described in [Appendix D](#). [Figure 5-11](#) shows the abstraction.

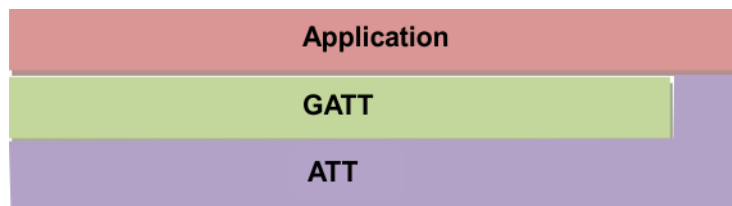


Figure 5-11. GATT Client Abstraction

5.3.3.1 Using the GATT Layer Directly

This section describes how to use the GATT layer in the application. The functionality of the GATT layer is implemented in the library but header functions can be found in the `gatt.h` file. [Appendix D](#) has the complete API for the GATT layer. [Specification of the Bluetooth System, Covered Core Package, Version: 4.2](#) provides more information on the functionality of these commands. These functions are used primarily for GATT client applications. A few server-specific functions are described in the API. Most of the GATT functions return ATT events to the application so consider the ATT API in [Appendix D](#). The general procedure to use the GATT layer when functioning as a GATT client (that is, in the SimpleBLECentral project) is as follows:

1. Initialize the GATT client.

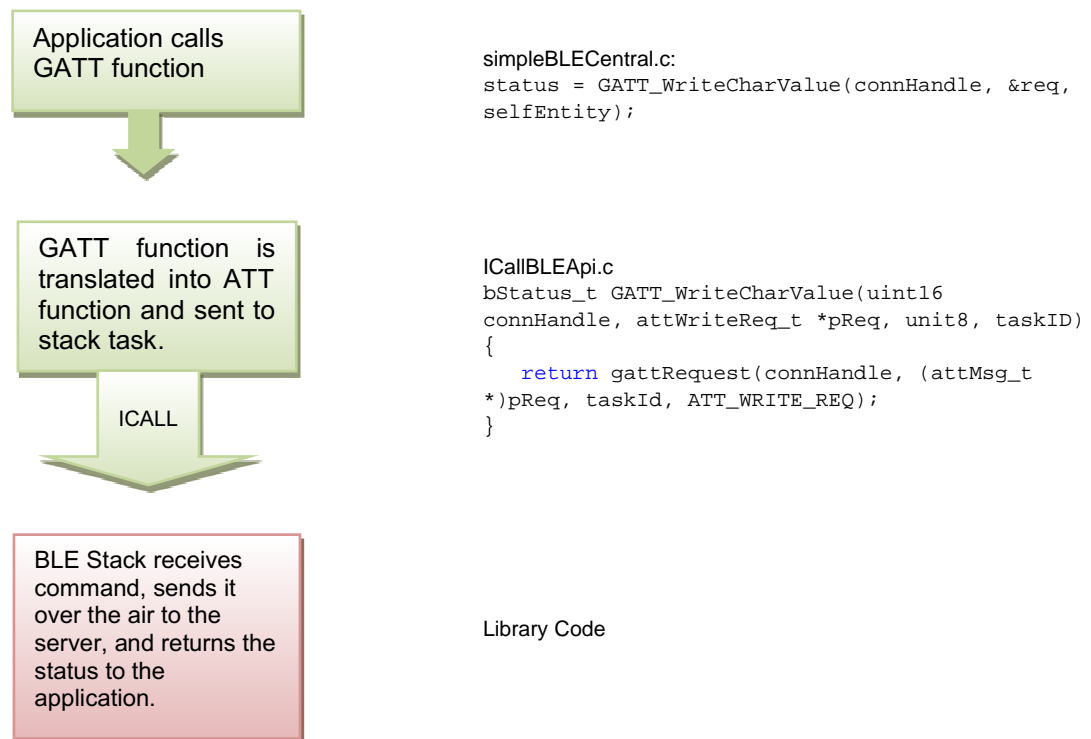
```
VOID GATT_InitClient();
```

2. Register to receive incoming ATT indications and notifications.

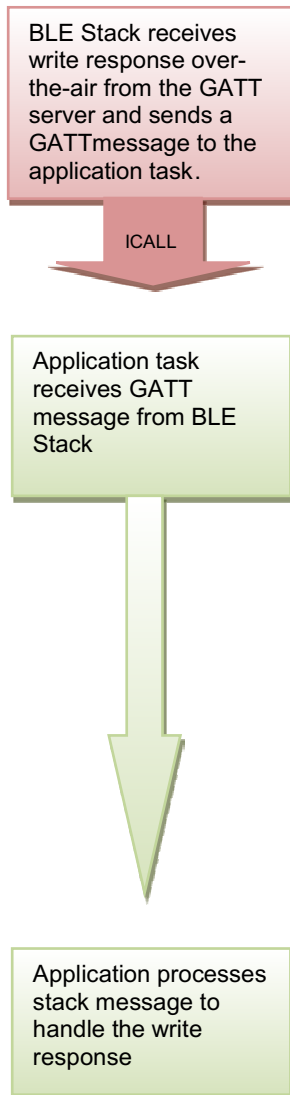
```
GATT_RegisterForInd(selfEntity);
```

3. Perform a GATT client procedure.

NOTE: The example uses `GATT_WriteCharValue()`, which is triggered by a left key press in the SimpleBLECentral application. Green corresponds to the app context and red corresponds to the protocol stack context.



4. Receive and handle the response to the GATT client procedure in the application. In this example, the application receives an ATT_WRITE_RSP event. See Section D.6 for a list of GATT commands and their corresponding ATT events. Green corresponds to the app context and red corresponds to the protocol stack context.



Library Code

```

simpleBLECentral.c:
if (ICall_fetchServiceMsg(&src, &dest, (void
**) &pMsg) == ICALL_ERRNO_SUCCESS)
{
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest ==
selfEntity))
    {
        // Process inter-task message
        SimpleBLECentral_processStackMsg((ICall_Hdr
*)pMsg);
        ...
SimpleBLECentral.c:
static void
SimpleBLECentral_processStackMsg(ICall_Hdr *pMsg)
{
    switch (pMsg->event)
    {
        case GATT_MSG_EVENT:

SimpleBLECentral_processGATTMsg((gattMsgEvent_t
*)pMsg);
        break;
        ...
static void
SimpleBLECentral_processGATTMsg(gattMsgEvent_t *pMsg)
{
    ...
    else if ((pMsg->method == ATT_WRITE_RSP) ||
              ((pMsg->method == ATT_ERROR_RSP) &&
               (pMsg->
>msg.errorRsp.reqOpcode == ATT_WRITE_REQ)))
    {
        ...
        else
        {
            // After a successful write, display the
value that was // written and increment
value
            LCD_WRITE_STRING_VALUE("Write sent:",
charVal++, 10, LCD_PAGE2);
            ...
        }
    }
}
  
```

NOTE: Even though the event sent to the application is an ATT event, it is sent as a GATT protocol stack message (GATT_MSG_EVENT).

5. A GATT client may also receive asynchronous data from the GATT server as indications or notifications other than receiving responses to its own commands. Registering to receive these ATT

notifications and indications is required as in Step 2. These notifications and indications are also be sent as ATT events in GATT messages to the application and must be handled as described in [Section 5.3.2](#).

5.3.4 GATT Server Abstraction

As a GATT server, most of the GATT functionality is handled by the individual GATT profiles. These profiles use the GattServApp (a configurable module that stores and manages the attribute table). Figure 5-12 shows this abstraction hierarchy.

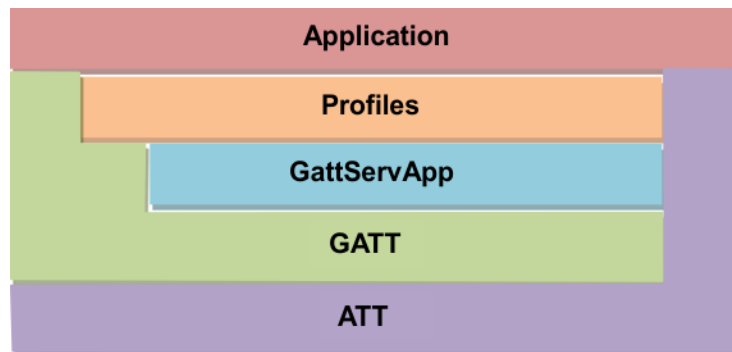


Figure 5-12. GATT Server Abstraction

The design process involves creating GATT profiles that configure the GATTServApp module and use its API to interface with the GATT layer. In this case of a GATT server, direct calls to GATT layer functions are unnecessary. The application then interfaces with the profiles.

5.3.4.1 GATTServApp Module

The GATTServApp stores and manages the application-wide attribute table. Various profiles use this module to add their characteristics to the attribute table. The Bluetooth low energy stack uses this module to respond to discovery requests from a GATT client. For example, a GATT client may send a Discover all Primary Characteristics message. The Bluetooth low energy stack on the GATT server receives this message and uses the GATTServApp to find and send over-the-air all of the primary characteristics stored in the attribute table. This type of functionality is beyond the scope of this document and is implemented in the library code. The GATTServApp functions accessible from the profiles are defined in gattservapp_util.c and described in the API in Appendix E. These functions include finding specific attributes and reading or modifying client characteristic configurations. See Figure 5-13 for more information.

5.3.4.1.1 Building up the Attribute Table

Upon power-on or reset, the application builds the GATT table by using the GATTServApp to add services. Each service consists of a list of attributes with UUIDs, values, permissions, and read and write call-backs. As Figure 5-13 shows, all of this information is passed through the GATTServApp to GATT and stored in the stack.

Attribute table initialization must occur in the application initialization function, that is, simple_peripheral_init().

```
// Initialize GATT attributes
GGS_AddService(GATT_ALL_SERVICES);           // GAP
GATTServApp_AddService(GATT_ALL_SERVICES);   // GATT attributes
DevInfo_AddService();                         // Device Information Service

#ifdef FEATURE_OAD_ONCHIP
    SimpleProfile_AddService(GATT_ALL_SERVICES); // Simple GATT Profile
#endif // !FEATURE_OAD_ONCHIP
```

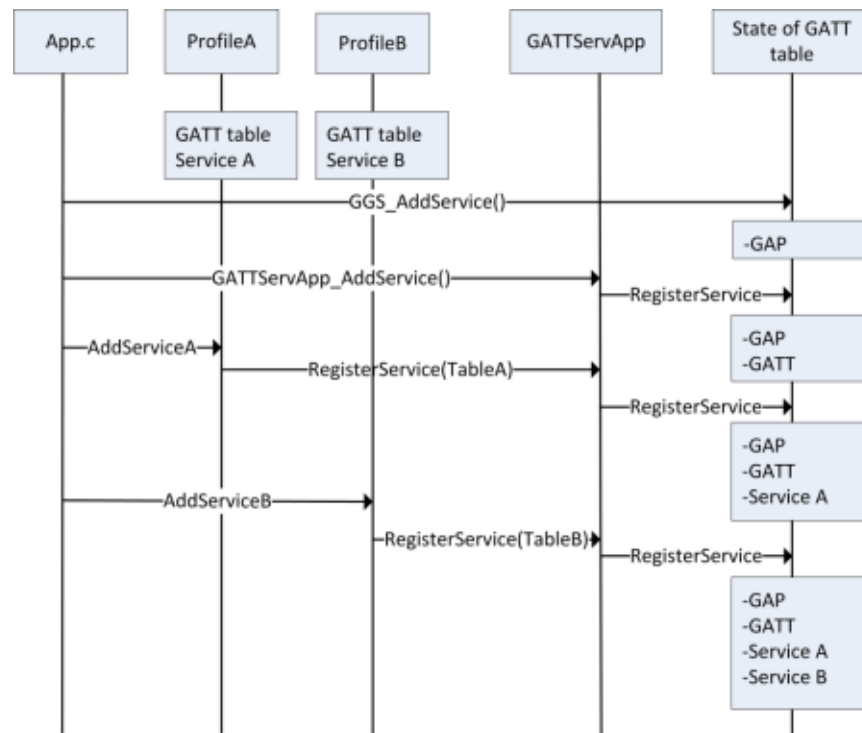


Figure 5-13. Attribute Table Initialization

5.3.4.2 Profile Architecture

This section describes the general architecture for all profiles and provides specific functional examples in relation to the `simple_gatt_profile` in the `simple_peripheral` project. See [Section 5.3.2](#) for an overview of the `simple_gatt_profile`. To interface with the application and *Bluetooth* low energy protocol stack at minimum, each profile must contain all the sub-elements of [Section 5.3.4.2.1](#).

5.3.4.2.1 Attribute Table Definition

Each service or group of GATT attributes must define a fixed size attribute table that gets passed into GATT. This table in `simple_gatt_profile.c` is defined as the following.

```
static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED]
```

Each attribute in this table is of the following type.

```
typedef struct attAttribute_t
{
    gattAttrType_t type; //!< Attribute type (2 or 16 octet UUIDs)
    uint8 permissions; //!< Attribute permissions
    uint16 handle; //!< Attribute handle - assigned internally by attribute server
    uint8* const pValue; //!< Attribute value - encoding of the octet array is defined in
                        //!< the applicable profile. The maximum length of an attribute
                        //!< value shall be 512 octets.
} gattAttribute_t;
```

The specific elements of this attribute type are detailed as follows.

- `type` is the UUID associated with the attribute and is defined as follows:

```
typedef struct
{
    uint8 len; //!< Length of UUID (2 or 6)
    const uint8 *uuid; //!< Pointer to UUID
} gattAttrType_t;
```

The length can be either `ATT_BT_UUID_SIZE` (2 bytes), or `ATT_UUID_SIZE` (16 bytes). The `*uuid` is a pointer to a number either reserved by Bluetooth SIG (defined in `gatt_uuid.c`) or a custom UUID defined in the profile.

- `permissions` enforces how and if a GATT client device can access the value of the attribute. Possible permissions are defined in `gatt.h` as follows:

```
#define GATT_PERMIT_READ    0x01 //!< Attribute is Readable
#define GATT_PERMIT_WRITE  0x02 //!< Attribute is Writable
#define GATT_PERMIT_AUTHEN_READ  0x04 //!< Read requires Authentication
#define GATT_PERMIT_AUTHEN_WRITE 0x08 //!< Write requires Authentication
#define GATT_PERMIT_AUTHOR_READ  0x10 //!< Read requires Authorization
#define GATT_PERMIT_AUTHOR_WRITE 0x20 //!< Write requires Authorization
#define GATT_PERMIT_ENCRYPT_READ  0x40 //!< Read requires Encryption
#define GATT_PERMIT_ENCRYPT_WRITE 0x80 //!< Write requires Encryption
```

[Section 5.3.5](#) further describes authentication, authorization, and encryption.

- `handle` is a placeholder in the table where `GATTServApp` assigns a handle. This placeholder is not customizable. Handles are assigned sequentially.
- `pValue` is a pointer to the attribute value. The size is unable to change after initialization. The maximum size is 512 octets.

The following sections provide examples of attribute definitions for common attribute types.

5.3.4.2.1.1 Service Declaration

Consider the following `simple_gatt_profile` service declaration attribute:

```
// Simple Profile Service
{
  { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
  GATT_PERMIT_READ, /* permissions */
  0, /* handle */
  (uint8 *)&simpleProfileService /* pValue */
},
```

The type is set to the Bluetooth SIG-defined primary service UUID (0x2800). A GATT client must read this attribute, so the permission is set to `GATT_PERMIT_READ`. The `pValue` is a pointer to the UUID of the service, custom-defined as `0xFFFF0`.

```
// Simple Profile Service attribute
static CONST gattAttrType_t simpleProfileService = { ATT_BT_UUID_SIZE,
  simpleProfileServUUID };
```

5.3.4.2.1.2 Characteristic Declaration

Consider the following `simple_gatt_profile` `simpleProfileCharacteristic1` declaration.

```
// Characteristic 1 Declaration
{
  { ATT_BT_UUID_SIZE, characterUUID },
  GATT_PERMIT_READ,
  0,
  &simpleProfileChar1Props
},
```

The type is set to the Bluetooth SIG-defined characteristic UUID (0x2803).

A GATT client must read this so the permission is set to `GATT_PERMIT_READ`.

[Section 5.3.1](#) describes the value of a characteristic declaration. For functional purposes, the only information required to be passed to the `GATTServApp` in `pValue` is a pointer to the properties of the characteristic value. The `GATTServApp` adds the UUID and the handle of the value. These properties are defined as follows.

```
// Simple Profile Service attribute
static CONST gattAttrType_t simpleProfileService = { ATT_BT_UUID_SIZE, simpleProfileServUUID };
```

NOTE: An important distinction exists between these properties and the GATT permissions of the characteristic value. These properties are visible to the GATT client stating the properties of the characteristic value. The GATT permissions of the characteristic value affect its functionality in the protocol stack. These properties must match that of the GATT permissions of the characteristic value. [Section 5.3.4.2.1.3](#) expands on this idea.

5.3.4.2.1.3 Characteristic Value

Consider the `simple_gatt_profile` `simpleProfileCharacteristic1` value.

```
// Characteristic Value 1
{
  { ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
  GATT_PERMIT_READ | GATT_PERMIT_WRITE,
  0,
  &simpleProfileChar1
},
```

The type is set to the custom-defined `simpleProfilechar1` UUID (0xFFFF1). The properties of the characteristic value in the attribute table must match the properties from the characteristic value declaration. The `pValue` is a pointer to the location of the actual value, statically defined in the profile as follows.

```
// Characteristic 1 Value
static uint8 simpleProfileChar1 = 0;
```

5.3.4.2.1.4 Client Characteristic Configuration

Consider the simple_gatt_profile simpleProfileCharacteristic4 configuration.

```
// Characteristic 4 configuration
{
    { ATT_BT_UUID_SIZE, clientCharCfgUUID },
    GATT_PERMIT_READ | GATT_PERMIT_WRITE,
    0,
    (uint8 *)&simpleProfileChar4Config
},
```

The type is set to the Bluetooth SIG-defined client characteristic configuration UUID (0x2902) GATT clients must read and write to this attribute so the GATT permissions are set to readable and writeable. The pValue is a pointer to the location of the client characteristic configuration array, defined in the profile as follows.

```
static gattCharCfg_t *simpleProfileChar4Config;
```

NOTE: Client characteristic configuration is represented as an array because this value must be cached for each connection. The catching of the client characteristic configuration is described in more detail in [Section 5.3.4.2.2](#).

5.3.4.2.2 Add Service Function

As described in [Section 5.3.4.1](#), when an application starts up it requires adding the GATT services it supports. Each profile needs a global AddService function that can be called from the application. Some of these services are defined in the protocol stack, such as GAP GATT Service and GATT Service. User-defined services must expose their own AddService function that the application can call for profile initialization. Using SimpleProfile_AddService() as an example, these functions should do as follows.

- Allocate space for the client characteristic configuration (CCC) arrays. As an example, a pointer to one of these arrays was initialized in the profile as described in [Section 5.3.4.2.1.4](#).

In the AddService function, the supported connections is declared and memory is allocated for each array. Only one CCC is defined in the simple_gatt_profile but there can be multiple CCCs.

```
// Allocate Client Characteristic Configuration table
simpleProfileChar4Config = (gattCharCfg_t *)ICall_malloc( sizeof(gattCharCfg_t) *
                                                         linkDBNumConns );

if ( simpleProfileChar4Config == NULL )
{
    return ( bleMemAllocError );
}
```

- Initialize the CCC arrays. CCC values are persistent between power downs and between bonded device connections. For each CCC in the profile, the GATTServApp_InitCharCfg() function must be called. This function tries to initialize the CCCs with information from a previously bonded connection and set the initial values to default values if not found.

```
GATTServApp_InitCharCfg( INVALID_CONHANDLE, simpleProfileChar4Config );
```

- Register the profile with the GATTServApp. This function passes the attribute table of the profile to the GATTServApp so that the attributes of the profile are added to the application-wide attribute table managed by the protocol stack and handles are assigned for each attribute. This also passes pointers to the callbacks of the profile to the stack to initiate communication between the GATTServApp and the profile.

```
// Register GATT attribute list and CBs with GATT Server App
status = GATTServApp_RegisterService( simpleProfileAttrTbl,
                                       GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                                       GATT_MAX_ENCRYPT_KEY_SIZE,
                                       &simpleProfileCBs );
```

5.3.4.2.3 Register Application Callback Function

Profiles can relay messages to the application using callbacks. In the `simple_peripheral` project, the `simple_gatt_profile` calls an application callback whenever the GATT client writes a characteristic value. For these application callbacks to be used, the profile must define a Register Application Callback function that the application uses to set up callbacks during its initialization. The register application callback function for the `simple_gatt_profile` is the following.

```
bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks )
{
    if ( appCallbacks )
    {
        simpleProfile_AppCBs = appCallbacks;

        return ( SUCCESS );
    }
    else
    {
        return ( bleAlreadyInRequestedMode );
    }
}
```

Where the callback typedef is defined as the following.

```
typedef struct
{
    simpleProfileChange_t      pfnSimpleProfileChange; // Called when characteristic value
changes
} simpleProfileCBs_t;
```

The application must then define a callback of this type and pass it to the `simple_gatt_profile` with the `SimpleProfile_RegisterAppCBs()` function. This occurs in `simple_peripheral.c` as follows.

```
// Simple GATT Profile Callbacks
#ifdef FEATURE_OAD_ONCHIP
static simpleProfileCBs_t SimpleBLEPeripheral_simpleProfileCBs =
{
    SimpleBLEPeripheral_charValueChangeCB // Characteristic value change callback
};
#endif // !FEATURE_OAD_ONCHIP
...
// Register callback with SimpleGATTprofile
SimpleProfile_RegisterAppCBs(&SimpleBLEPeripheral_simpleProfileCBs);
```

See [Section 5.3.4.2.4](#) for further information on how this callback is used.

5.3.4.2.4 Read and Write Callback Functions

The profile must define Read and Write callback functions which the protocol stack calls when one of the attributes of the profile are written to or read from. The callbacks must be registered with `GATTServApp` as mentioned in [Section 5.3.4.2.2](#). These callbacks perform the characteristic read or write and other processing (possibly calling an application callback) as defined by the specific profile.

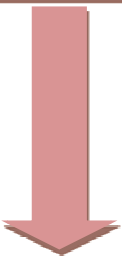
5.3.4.2.4.1 Read Request from Client

When a read request from a GATT Client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is readable, call the read call-back profile. The profile copies in the value, performs any profile-specific processing, and notifies the application if desired. This procedure is illustrated in the following flow diagram for a read of `simpleprofileChar1` in the `simple_gatt_profile`. Red corresponds to processing in the protocol stack context.

BLE Stack receives attribute read request over-the-air and sends a GATT_MSG_EVENT to GATTServApp



GATTServApp receives GATT_MSG_EVENT, processes it to call profile callback



Profile copies the value of the characteristic into the data pointer and returns this to GATTServApp



GATTServApp returns the attribute value to the stack to be returned over-the-air to the GATT Client

Library Code

```
gattservapp.c:
if ( useCB == TRUE )
{
    // Use Service's read callback to process the request
    pfnGATTReadAttrCB_t pfnCB = gattServApp_FindReadAttrCB( service
);
    if ( pfnCB != NULL )
    {
        // Read the attribute value
        status = (*pfnCB)( connHandle, pAttr, pValue, pLen, offset,
mxLen, method );
        ...
    }
}
```

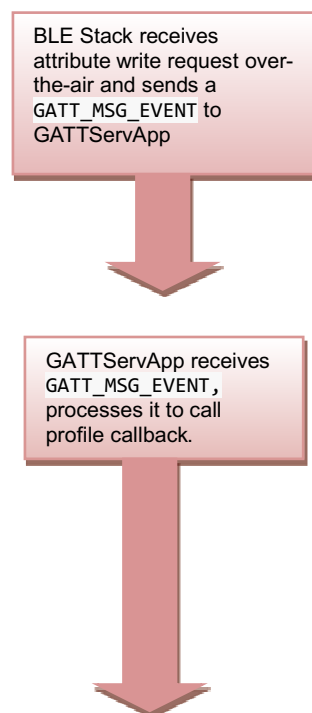
```
simpleGATTProfile.c
static bStatus_t simpleProfile_ReadAttrCB(.....)
{
    if ( pAttr->type.len == ATT_BT_UUID_SIZE )
    {
        // 16-bit UUID
        uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr-
>type.uuid[1]);
        switch (uuid )
        {
            case SIMPLEPROFILE_CHAR1_UUID:
                *pLen = 1;
                pValue[0] = *pAttr->pValue;
                break;
            ...
        }
    }
}
```

Library Code

NOTE: The processing is in the context of the protocol stack. If any intensive profile-related processing that must be done in the case of an attribute read, this should be split up and done in the context of the Application task. See the following write request for more information.

5.3.4.2.4.2 Write Request from Client

When a write request from a GATT client is received for a given attribute, the protocol stack checks the permissions of the attribute and, if the attribute is write, call the write callback of the profile. The profile stores the value to be written, performs any profile-specific processing, and notifies the application if desired. The following flow diagram shows this procedure for a write of simpleprofileChar3 in the simple_gatt_profile. Red corresponds to processing in the protocol stack context and green is processing in the application context.

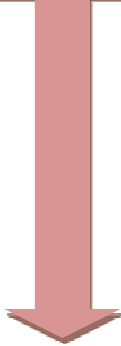


Library Code

```

gattservapp.c:
// Find the owner of the attribute
pAttr = GATT_FindHandle( handle, &service );
if ( pAttr != NULL )
{
    // Find out the owner's callback functions
    pfnGATTWriteAttrCB_t pfnCB =
    gattservapp_FindWriteAttrCB( service);
    if ( pfnCB != NULL )
    {
        //Try to write the new value
        status = (*pfnCB)( connHandle, pAttr,
        pValue, len, offset, method);
    }
}
...
  
```


Profile stores the value from the characteristic write as the new characteristic value.



Profile notifies the application by calling its callback function to enqueue message in the application and post its semaphore.



I
C
A
L
L

Status is returned through GATTServApp to stack to send over-the-air to GATT Client

```

simpleGATTProfile.c
static bStatus_t
simpleProfile_WriteAttrCB(.....)
{
    if ( pAttr->type.len == ATT_BT_UUID_SIZE )
    {
        uint16 uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr->type.uuid[1]_);
        switch ( uuid )
        {
            // Write the value
            if ( status == SUCCESS )
            {
                uint8 *pCurValue = (uint8 *)pAttr->pValue;
                *pCurValue = pValue[0];
                ...
                notifyApp = SIMPLEPROFILE_CHAR3;

                // If a characteristic value changed then callback function to notify application of change
                if ( (notifyApp !=0xFF ) && simpleProfile_AppCBs && simpleProfile_AppCBs->pfnSimpleProfileChange( notifyApp ) );
                {
                    simpleProfile_AppCBs->pfnSimpleProfileChange( notifyApp );
                }
            }
        }
    }
}

```

```

simpleBLEPeripheral.c
static void
SimpleBLEPeripheral_charValueChangeCB(uint8_t paramID)
{
    SimpleBLEPeripheral_enqueueMsg(SBP_CHAR_CHANGE_EVT, paramID);
}
Library Code

```

Application wakes up to do additional write-related processing

```

simpleBLEPeripheral.c
static void
SimpleBLEPeripheral_processAppMsg(sbpEvt_t
*pMsg)
{
    switch (pMsg->event)
    {
        case SBP_STATE_CHANGE_EVT:
            SimpleBLEPeripheral_processStateChangeEvt
            ((gaprole_States_t)pMsg->status);
            ...
        static void
SimpleBLEPeripheral_processCharValueChangeEvt(ui
nt8+t paramID)
    {
        switch (paramID)
        {
            case
SIMPLEPROFILE_CHAR3:SimpleProfile_GetParameter(S
IMPLEPROFILE_CHAR3, &newValue);

                LCD_WRITE_STRING_VALUE("Char 3:",
                (uint16_t)newValue, 10, LCD_PAGE4)1
    }
}
    
```

NOTE: Minimizing the processing in protocol stack context is important. In this example, additional processing beyond storing the attribute write value in the profile (that is, writing to the LCD) occurs in the application context by enqueueing a message in the queue of the application.

5.3.4.2.5 Get and Set Functions

The profile containing the characteristics shall provide set and get abstraction functions for the application to read and write a characteristic of the profile. The set parameter function also includes logic to check for and implement notifications and indications if the relevant characteristic has notify or indicate properties. Figure 5-14 and the following code show this example for setting simpleProfileCharacteristic4 in the simple_gatt_profile.

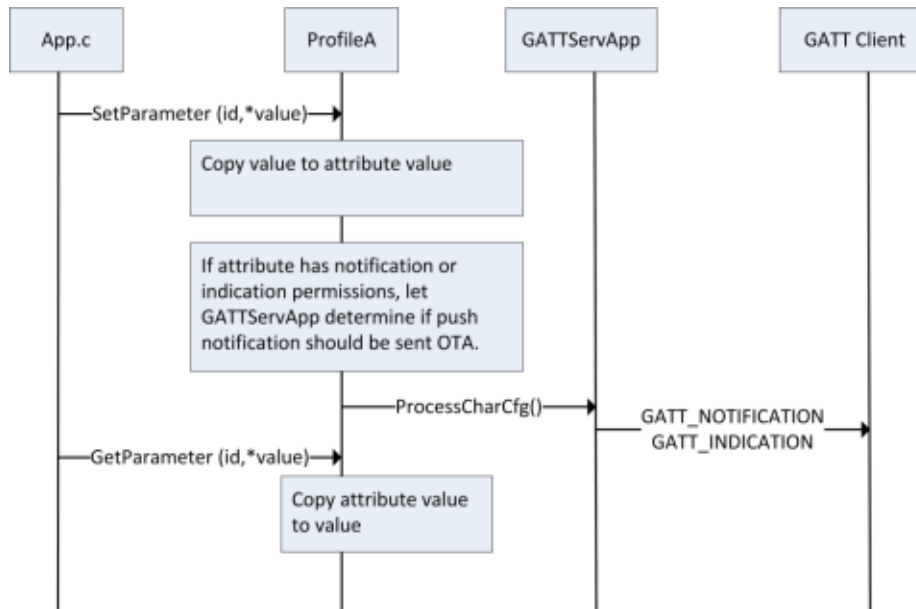


Figure 5-14. Get and Set Profile Parameter

For example, the application initializes simpleProfileCharacteristic4 to 0 in simple_peripheral.c through the following.

```
uint8_t charValue4 = 4;  
SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8_t),  
                           &charValue4);
```

The code for this function is displayed in the following code snippet (from `simple_gatt_profile.c`). Besides setting the value of the static `simpleProfileChar4`, this function also calls `GATTServApp_ProcessCharCfg` because it has notify properties. This action forces `GATTServApp` to check if notifications have been enabled by the GATT Client. If so, the `GATTServApp` sends a notification of this attribute to the GATT Client.

```
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value )
{
    bStatus_t ret = SUCCESS
    switch ( param )
    {
        case SIMPLEPROFILE_CHAR4:
            if ( len == sizeof ( uint8 ) )
            {
                simpleProfileChar4 = *((uint8*)value);

                // See if Notification has been enabled
                GATTServApp_ProcessCharCfg( simpleProfileChar4Config, &simpleProfileChar4, FALSE,
                    simpleProfileAttrTbl, GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                    INVALID_TASK_ID, simpleProfile_ReadAttrCB );
            }
    }
}
```

5.3.5 Allocating Memory for GATT Procedures

To support fragmentation, GATT and ATT payload structures must be dynamically allocated for commands sent wirelessly. For example, a buffer must be allocated when sending a `GATT_Notification`. The stack does this allocation if the preferred method to send a GATT notification or indication is used: calling a `SetParameter` function of the profile (that is, `SimpleProfile_SetParameter()`) and calling `GATTServApp_ProcessCharCfg()` as described in [Section 5.3.4.2.5](#).

If using `GATT_Notification()` or `GATT_Indication()` directly, memory management must be added as follows.

1. Try to allocate memory for the notification or indication payload using `GATT_bm_alloc()`.
2. Send notification or indication using `GATT_Notification()` or `GATT_Indication()` if the allocation succeeds.

NOTE: If the return value of the notification or indication is `SUCCESS (0x00)`, the stack freed the memory.

3. Use `GATT_bm_free()` to free the memory if the return value is something other than `SUCCESS` (for example, `blePending`).

The following is an example of this in the `gattServApp_SendNotifInd()` function in the `gattservapp_util.c` file.

```

noti.pValue = (uint8 *)GATT_bm_alloc( connHandle, ATT_HANDLE_VALUE_NOTI,
                                     GATT_MAX_MTU, &len );

if ( noti.pValue != NULL )
{
    status = (*pfnReadAttrCB)( connHandle, pAttr, noti.pValue, &noti.len,
                              0, len, GATT_LOCAL_READ );

    if ( status == SUCCESS )
    {
        noti.handle = pAttr->handle;

        if ( cccValue & GATT_CLIENT_CFG_NOTIFY )
        {
            status = GATT_Notification( connHandle, &noti, authenticated );
        }
        else // GATT_CLIENT_CFG_INDICATE
        {
            status = GATT_Indication( connHandle, (attHandleValueInd_t *)&noti,
                                     authenticated, taskId );
        }
    }

    if ( status != SUCCESS )
    {
        GATT_bm_free( (gattMsg_t *)&noti, ATT_HANDLE_VALUE_NOTI );
    }
}
else
{
    status = bleNoResources;
}

```

For other GATT procedures, take similar steps as noted in the API in [Appendix D](#)

5.3.6 Registering to Receive Additional GATT Events in the Application

Using `GATT_RegisterForMsgs()` (see [Appendix D](#)), receiving additional GATT messages to handle certain corner cases is possible. This possibility can be seen in `simple_peripheral_processGATTMsg()`. The following three cases are currently handled.

- GATT server in the stack was unable to send an ATT response (due to lack of available HCI buffers): Attempt to transmit on the next connection interval. Additionally, a status of `bleTimeout` is sent if the ATT transaction is not completed within 30 seconds, as specified in the core spec.

```

// See if GATT server was unable to transmit an ATT response
if (pMsg->hdr.status == blePending)
{
    // No HCI buffer was available. Let's try to retransmit the response
    // on the next connection event.
    if (HCI_EXT_ConnEventNoticeCmd(pMsg->connHandle, selfEntity,
                                   SBP_CONN_EVT_END_EVT) == SUCCESS)
    {
        // First free any pending response
        SimpleBLEPeripheral_freeAttRsp(FAILURE);

        // Hold on to the response message for retransmission
        pAttRsp = pMsg;

        // Don't free the response message yet
        return (FALSE);
    }
}

```

- An ATT flow control violation: The application is notified that the connected device has violated the ATT flow control specification, such as sending a Read Request before an Indication Confirm is sent.

No more ATT requests or indications can be sent wirelessly during the connection. The application may want to terminate the connection due to this violation. As an example in `simple_peripheral`, the LCD is updated.

```
else if (pMsg->method == ATT_FLOW_CTRL_VIOLATED_EVENT)
{
    // ATT request-response or indication-confirmation flow control is
    // violated. All subsequent ATT requests or indications will be dropped.
    // The app is informed in case it wants to drop the connection.

    // Display the opcode of the message that caused the violation.
    DISPLAY_WRITE_STRING_VALUE("FC Violated: %d", pMsg->msg.flowCtrlEvt.opcode,
                              LCD_PAGE5);
}
```

- An ATT MTU size is updated: The application is notified in case this affects its processing in any way. See [Section 5.5.2](#) for more information on the MTU. As an example in `simple_peripheral`, the LCD is updated.

```
else if (pMsg->method == ATT_MTU_UPDATED_EVENT)
{
    // MTU size updated
    DISPLAY_WRITE_STRING_VALUE("MTU Size: $d", pMsg->msg.mtuEvt.MTU, LCD_PAGE5);
}
```

5.3.7 GATT Security

As described in [Section 5.3.4](#), the GATT server may define permissions independently for each characteristic. The server may allow some characteristics to be accessed by any client, while limiting access to other characteristics to only authenticated or authorized clients. These permissions are usually defined as part of a higher level profile specification. For custom profiles, the user may select the permissions as they see fit. For more information about the GATT Security, refer to the Bluetooth Specification Version 4.2 [Vol 3, Part G] Section 8.

5.3.7.1 Authentication

Characteristics that require authentication cannot be accessed until the client has gone through an authenticated pairing method. This verification is performed within the stack, with no processing required by the application. The only requirement is for the characteristic to be registered properly with the GATT server.

For example, characteristic 5 of the `simple_gatt_profile` allows on authenticated reads.

```
// Characteristic Value 5
{
    { ATT_BT_UUID_SIZE, simpleProfilechar5UUID },
    GATT_PERMIT_AUTHEN_READ,
    0,
    simpleProfileChar5
},
```

When an un-authenticated client attempts to read this value, the GATT server automatically rejects it with `ERROR_INSUFFICIENT_AUTHEN` (0x41), without invoking the `simpleProfile_ReadAttrCB()`. See an example of this in [Figure 5-15](#).

| Pnbr. | Time (us) | Channel | Access Address | Direction | ACK Status | Data Type | Data Header | L2CAP Header | ATT_Read_Req | CRC | RSSI (dBm) | FCS |
|-------|---------------------|---------|----------------|-----------|------------|-----------|--|--------------------------------------|--|----------|------------|-----|
| 517 | +29772 =13535613 | 0x20 | 0x5065626C | M->S | OK | L2CAP-S | LLID NESN SN MD PDU-Length 2 0 1 0 7 | L2CAP-Length ChanId 0x0003 0x0004 | Opcode AttHandle 0x0A 0x002B | 0x9B1F71 | -53 | OK |
| 518 | +285 =13535898 | 0x20 | 0x5065626C | S->M | OK | Empty PDU | | | | 0x08E6DA | -43 | OK |
| 519 | +29716 =13565614 | 0x02 | 0x5065626C | M->S | OK | Empty PDU | | | | 0x08E009 | -42 | OK |
| 520 | +229 =13565843 | 0x02 | 0x5065626C | S->M | OK | L2CAP-S | LLID NESN SN MD PDU-Length 2 1 1 0 9 | L2CAP-Length ChanId 0x0005 0x0004 | ATT_Error_Response Opcode ReqOpCode AttHandle ErrorCode 0x002B INSUF_ENCRYPTION(0x0F) | 0x131A4A | -39 | OK |
| 521 | +29772 =13595615 | 0x09 | 0x5065626C | M->S | OK | L2CAP-S | LLID NESN SN MD PDU-Length 2 0 1 0 11 | L2CAP-Length ChanId 0x0007 0x0006 | SM_Pairing_Req Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist 0x01 0x04 0x00 0x05 0x10 0x03 | 0x30D98B | -54 | OK |
| 522 | +317 =13595932 | 0x09 | 0x5065626C | S->M | OK | Empty PDU | | | | 0x08E6DA | -39 | OK |
| 523 | +29683 =13625615 | 0x1B | 0x5065626C | M->S | OK | Empty PDU | | | | 0x08E009 | -48 | OK |
| 524 | +229 =13625844 | 0x1B | 0x5065626C | S->M | OK | L2CAP-S | LLID NESN SN MD PDU-Length 2 1 1 0 11 | L2CAP-Length ChanId 0x0007 0x0006 | SM_Pairing_Resp Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist 0x02 0x00 0x00 0x05 0x10 0x03 | 0x8C0B8B | -41 | OK |

Figure 5-15. Sniffer Capture Example

After the client has successfully authenticated, read/write requests are forwarded to the profiles read/write callback. See the code below for a simple_gatt_profile example:

```
case SIMPLEPROFILE_CHAR5_UUID:
    *pLen = SIMPLEPROFILE_CHAR5_LEN;
    VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR5_LEN );
    break;
```

5.3.7.2 Authorization

Authorization is a layer of security provided in addition to what BLE already implements. Because applications are required to define their own requirements for authorization, the stack forwards read/write requests on these characteristics to the application layer of the profile.

For the profile to register for authorization information from the GATT server, it must define an authorization callback with the stack. The simple_gatt_profile does not do this by default, but below is an example of how it could be modified to do this.

1. Register profile level authorization callback.

```
CONST gattServiceCBs_t simpleProfileCBs =
{
    simpleProfile_ReadAttrCB, // Read callback function pointer
    simpleProfile_WriteAttrCB, // Write callback function pointer
    simpleProfile_authorizationCB // Authorization callback function pointer
};
```

2. Implement authorization callback code.

```
static bStatus_t simpleProfile_authorizationCB( uint16 connHandle,
                                                gattAttribute_t *pAttr,
                                                uint8 opcode )
{
    //This is just an example implementation, normal use cases would require
    //more complex logic to determine that the device is authorized
    if(clientIsAuthorized)
        return SUCCESS;
    else
        return ATT_ERR_INSUFFICIENT_AUTHOR;
```

The authorization callback executes in the stack context; thus, intensive processing should not be performed in this function. The implementation is left up to the developer; the above callback should be treated as a shell. The return value should be either SUCCESS if the client is authorized to access the characteristic, or ATT_ERR_INSUFFICIENT_AUTHOR if they have not yet obtained proper authorization. Authorization requires the connection to be authenticated beforehand, or ATT_ERR_INSUFFICIENT_AUTHEN will be sent as an error response.

NOTE: If a characteristic that requires authorization is registered with the GATT server, but no application level authorization callback is defined, the stack will return ATT_ERR_UNLIKELY. Because this error can be cryptic, TI recommends using an authorization callback.

5.4 GAP Bond Manager and LE Secure Connections

5.4.1 Overview

The GAP Bond Manager is a configurable module that offloads most of the security mechanisms from the application. [Table 5-1](#) lists the terminology.

Table 5-1. GAP Bond Manager Terminology

| Term | Description |
|----------------|---|
| Pairing | The process of exchanging keys |
| Encryption | Data is encrypted after pairing, or re-encryption (a subsequent connection where keys are looked up from nonvolatile memory). |
| Authentication | The pairing process completed with MITM (Man in the Middle) protection. |
| Bonding | Storing the keys in nonvolatile memory to use for the next encryption sequence. |
| Authorization | An additional application level key exchange in addition to authentication |
| OOB | Out of Band. Keys are not exchanged over the air, but rather over some other source such as serial port or NFC. This also provides MITM protection. |
| MITM | Man in the Middle protection. This prevents an attacker from listening to the keys transferred over the air to break the encryption. |
| Just Works | Pairing method where keys are transferred over the air without MITM |

The general process that the GAPBondMgr uses is as follows.

1. The pairing process exchanges keys through the following methods described in [Section 5.4.2](#).
2. Encrypt the link with keys from Step 1.
3. The bonding process stores keys in secure flash (SNV).
4. Use the keys stored in SNV to encrypt the link when reconnecting.

NOTE: Performing all of these steps is unnecessary. For example, two devices may choose to pair but not bond.

5.4.2 Selection of Pairing Mode

Version 4.2 of the Bluetooth Spec introduces a Secure Connections feature to upgrade BLE pairing. For a detailed description of the algorithms used for Secure Connections, see section 5.1 of Vol 1, Part A of the 4.2 Bluetooth Spec. The previous pairing methods used in the 4.1 and 4.0 Bluetooth Specs are still available, and are now defined as LE legacy pairing. The main difference is that Secure Connection uses Elliptic Curve Diffie-Hellman cryptography, while LE legacy pairing does not.

There are four types of pairing models, each of which are described in detail in [Section 5.4.4](#):

- just works (Secure Connections or LE Legacy)
- passkey entry (Secure Connections or LE Legacy)
- numeric comparison (Secure Connections)
- Out of Band (Secure Connections or LE Legacy)

The selection of the association model and whether or not pairing will succeed is based upon the following parameters (all tables from Section 2.3.5.1 of Vol 3, Part h of the Bluetooth 4.2 Spec). The GAPBondMgr parameters, as they map to the table parameters below are listed here. For more information on these parameters, see the GAPBondMgr API.

- GAPBOND_LOCAL_OOB_SC_ENABLED: OOB Set / Not Set
- GAPBOND_MITM_PROTECTION: MITM Set / Not Set
- GAPBOND_IO_CAPABILITIES: IO Capabilities
- GAPBOND_SECURE_CONNECTION: secure connections supported / not supported

Beyond what the spec defines, this parameter also affects whether or not pairing succeeds, as described in the GAPBondMgr API.

If both devices support secure connections, use [Figure 5-16](#) to decide upon the next step.

| | | Initiator | | | |
|-----------|--------------|-----------|-------------|---------------------|---------------------|
| | | OOB Set | OOB Not Set | MITM Set | MITM Not Set |
| Responder | OOB Set | Use OOB | Use OOB | | |
| | OOB Not Set | Use OOB | Check MITM | | |
| | MITM Set | | | Use IO Capabilities | Use IO Capabilities |
| | MITM Not Set | | | Use IO Capabilities | Use Just Works |

Figure 5-16. Parameters With Secure Connections

If at least one device does not support secure connections, use [Figure 5-17](#) to decide upon the next step.

| | | Initiator | | | |
|-----------|--------------|------------|-------------|---------------------|---------------------|
| | | OOB Set | OOB Not Set | MITM Set | MITM Not Set |
| Responder | OOB Set | Use OOB | Check MITM | | |
| | OOB Not Set | Check MITM | Check MITM | | |
| | MITM Set | | | Use IO Capabilities | Use IO Capabilities |
| | MITM Not Set | | | Use IO Capabilities | Use Just Works |

Figure 5-17. Parameters Without Secure Connections

If, based on one of the previous tables, IO capabilities are to be used to determine the association model, use [Figure 5-18](#).

| Responder | Initiator | | | | |
|------------------|--|---|---|-------------------------------|---|
| | DisplayOnly | Display YesNo | Keyboard Only | NoInput NoOutput | Keyboard Display |
| Display Only | Just Works Unauthenticated | Just Works Unauthenticated | Passkey Entry: responder displays, initiator inputs Authenticated | Just Works Unauthenticated | Passkey Entry: responder displays, initiator inputs Authenticated |
| Display YesNo | Just Works Unauthenticated | Just Works (For LE Legacy Pairing) Unauthenticated | Passkey Entry: responder displays, initiator inputs Authenticated | Just Works Unauthenticated | Passkey Entry (For LE Legacy Pairing): responder displays, initiator inputs Authenticated |
| | | Numeric Comparison (For LE Secure Connections) Authenticated | | | Numeric Comparison (For LE Secure Connections) Authenticated |
| Keyboard Only | Passkey Entry: initiator displays, responder inputs Authenticated | Passkey Entry: initiator displays, responder inputs Authenticated | Passkey Entry: initiator and responder inputs Authenticated | Just Works Unauthenticated | Passkey Entry: initiator displays, responder inputs Authenticated |
| NoInput NoOutput | Just Works Unauthenticated | Just Works Unauthenticated | Just Works Unauthenticated | Just Works Unauthenticated | Just Works Unauthenticated |
| Keyboard Display | Passkey Entry: initiator displays, responder inputs Authenticated | Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated | Passkey Entry: responder displays, initiator inputs Authenticated | Just Works Unauthenticated | Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated |
| | | Numeric Comparison (For LE Secure Connections) Authenticated | | | Numeric Comparison (For LE Secure Connections) Authenticated |

Figure 5-18. Parameters With IO Capabilities

5.4.3 Using GAPBondMgr

This section describes what the application must do to configure, start, and use the GAPBondMgr. The GAPRole handles some of the GAPBondMgr functionality. The GAPBondMgr is defined in `gapbondmgr.c` and `gapbondmgr.h`. [Appendix D](#) describes the full API including commands, configurable parameters, events, and callbacks. The project being considered here is the `security_examples_central` project available from the TI SimpleLink GitHub page in . The general steps to use the GAPBondMgr module are as follows.

1. Configure the stack to include GAPBondMgr functionality and, if desired, secure connections. Define the following in `build_config.opt` in the stack project:
 - `DGAP_BOND_MGR`
 - `DBLE_V42_FEATURES=SECURE_CONNS_CFG`
2. The stack must also be configured to use 1 or 2 SNV pages, by defining `OSAL_SNV=1` or `OSAL_SNV=2` as a preprocessor-defined symbol in the stack project.
3. If using Secure Connections, the PDU size must be ≥ 69 . This can be set by defining the following preprocessor symbol in the application project: `MAX_PDU_SIZE=69`. Also, the minimum heap size that can be used with Secure Connections is 3690.
4. Configure the GAPBondMgr by initializing its parameters as desired. See [Section D.2](#) for a complete list of parameters with functionality described. There are examples of this for the various pairing / bonding modes in [Section 5.4.4](#).
5. Register application callbacks with the GAPBondMgr, so that the application can communicate with the GAPBondMgr and be notified of events. See [Section D.3](#) for the callback definitions and an example in the SimpleBLECentral project.

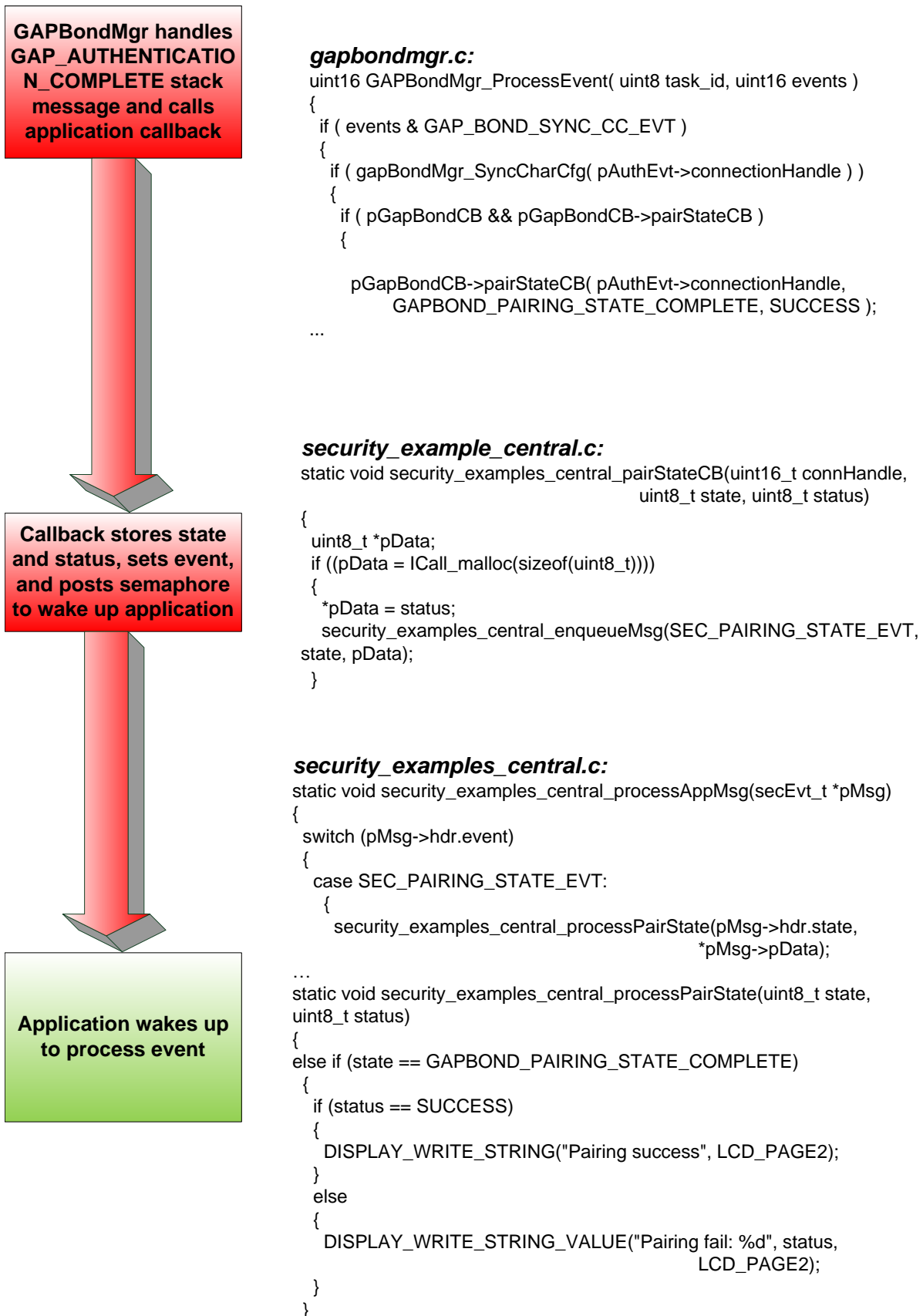
```
// Register with bond manager after starting device
GAPBondMgr_Register(&security_examples_central_bondCB);
```

This should also occur in the application initialization function after the GAPRole profile is started (that is, `GAPCentralRole_StartDevice()`).

6. Once the GAPBondMgr is configured, it operates mostly autonomously from the perspective of the application. When a connection is established, it initiates pairing and bonding, depending on the configuration parameters set during initialization, and communicates with the application as needed through the defined callbacks.

A few parameters can be set and functions called asynchronously at any time from the application. See [Appendix F](#) for more information.

Most communication between the GAPBondMgr and the application at this point occurs through the callbacks which were registered in Step 2. [Figure 5-19](#) is a flow diagram example from SimpleBLECentral of the GAPBondMgr notifying the application that pairing has completed. The same method occurs for various other events and will be expanded upon in the following section. In this diagram, red corresponds to processing in the protocol stack context and green to the application context.


Figure 5-19. Flow Diagram Example

5.4.4 GAPBondMgr Examples for Different Pairing Modes

This section provides message diagrams for the types of security that can be implemented. These modes assume acceptable I/O capabilities are available for the security mode, and that the selection of whether or not to support Secure Connections allows for the pairing mode. See the [Section 5.4.2](#) on how these parameters affect pairing. These examples only consider the pairing aspect. Bonding can be added to each type of pairing in the same manner and is shown in the next section.

NOTE: The code snippets here are not complete functioning examples, and are only intended for illustration purposes. See the `security_examples` for a complete example.

5.4.4.1 Pairing Disabled

With pairing set to `FALSE`, the BLE stack automatically rejects any attempt at pairing. Configure the `GAPBondMgr` as follows to disable pairing:

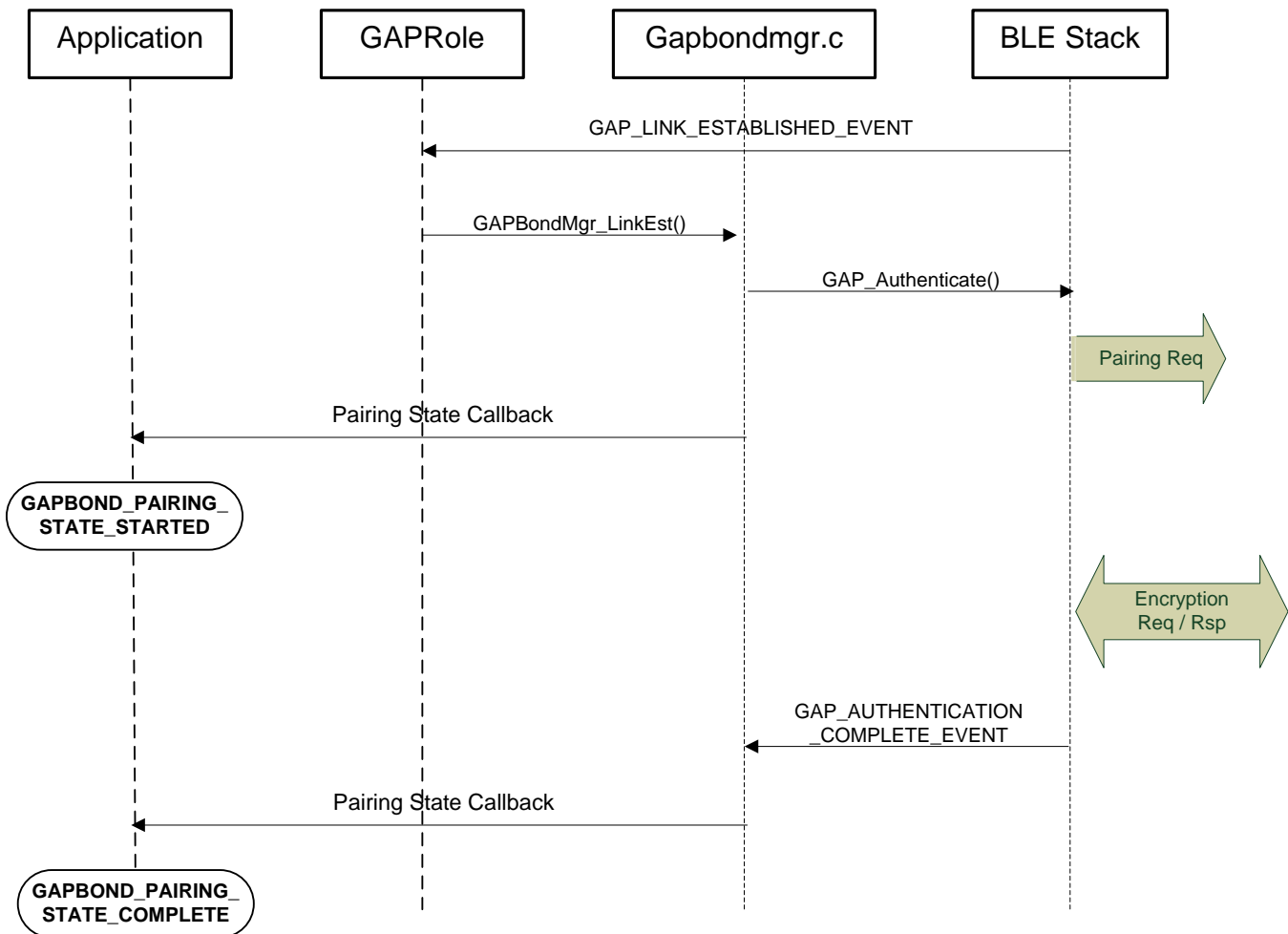
```
uint8 pairMode = GAPBOND_PAIRING_MODE_NO_PAIRING;
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &pairMode);
```

5.4.4.2 Just Works Pairing

Just Works pairing allows encryption without MITM authentication and is vulnerable to MITM attacks. Just Works pairing can be LE Legacy or Secure Connections pairing. The `GAPBondMgr` does not need any additional input from the application for just works pairing. Configure the `GAPBondMgr` for Just Works pairing as follows.

```
uint8_t pairMode = GAPBOND_PAIRING_MODE_INITIATE;
uint8_t mitm = FALSE;
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &pairMode);
GAPBondMgr_SetParameter(GAPBOND_MITM_PROTECTION, sizeof(uint8_t), &mitm);
```

[Figure 5-20](#) describes the interaction between the `GAPBondMgr` and the application for Just Works pairing. As shown, the application receives a `GAPBOND_PAIRING_STATE_STARTED` event once the pairing request has been sent, and a `GAPBOND_PAIRING_STATE_COMPLETE` event once the pairing process has completed. At this time, the link is encrypted.


Figure 5-20. Just Works Pairing

5.4.4.3 Passcode Entry

Passkey entry is a type of authenticated pairing that can prevent MITM attacks. It can be either LE Legacy pairing or Secure Connections pairing. In this pairing method, one device displays a 6-digit passcode, and the other device enters the passcode. As described in [Section 5.4.2](#), the IO capabilities decide which device performs which role. The passcode callback registered with the GAPBondMgr when it was started is used to enter or display the passcode. The following is an example of initiating Passcode Entry pairing where the passcode is displayed.

1. Define passcode callback

```

// Bond Manager Callbacks
static gapBondCBs_t security_examples_central_bondCB =
{
    (pfnPasscodeCB_t)security_examples_central_passcodeCB, // Passcode callback
    security_examples_central_pairStateCB // Pairing state callback
};

static void security_examples_central_passcodeCB(uint8_t *deviceAddr, uint16_t connHandle,
uint8_t uiInputs, uint8_t uiOutputs, uint32_t
numComparison)
{
    gapPasskeyNeededEvent_t *pData;

    // Allocate space for the passcode event.
    
```

```

if ((pData = ICall_malloc(sizeof(gapPasskeyNeededEvent_t)))
{
    memcpy(pData->deviceAddr, deviceAddr, B_ADDR_LEN);
    pData->connectionHandle = connHandle;
    pData->uiInputs = uiInputs;
    pData->uiOutputs = uiOutputs;

    // Enqueue the event.
    security_examples_central_enqueueMsg(SEC_PASSCODE_NEEDED_EVT, 0, (uint8_t *) pData);
}
}

```

2. Configure GAPBondMgr

```

uint8_t pairMode = GAPBOND_PAIRING_MODE_INITIATE;
uint8_t mitm = TRUE;
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &uint8_t pairMode =
GAPBOND_PAIRING_MODE_INITIATE;
uint8_t mitm = TRUE;
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &mitm);

```

3. Process passcode callback and send response to stack

```

static void security_examples_central_processPasscode(uint16_t connectionHandle,
gapPasskeyNeededEvent_t *pData)
{
    if (pData->uiInputs) // if we are to enter passkey
    {
        passcode = 111111;
        // Send passcode response
        GAPBondMgr_PasscodeRsp(connectionHandle, SUCCESS, passcode);
    }
    else if (pData->uiOutputs) // if we are to display passkey
    {
        passcode = 111111;
        DISPLAY_WRITE_STRING_VALUE("Passcode: %d", passcode, LCD_PAGE4);

        // Send passcode response
        GAPBondMgr_PasscodeRsp(connectionHandle, SUCCESS, passcode);
    }
}

```

Depending on what the uiInputs and uiOutputs returned from the GAPBondMgr, the passcode must either be displayed or entered. The passcode is then sent to the GAPBondMgr using GAPBondMgr_PasscodeRsp(), so that pairing can continue. In this case, the password is statically set to 111111. In a real product, the password will likely be randomly generated, and the device must expose a way for the user to enter the passcode, then send it to the GAPBondMgr using GAPBondMgr_PasscodeRsp(). There is an example of this in the security_examples projects. The complete interaction between the GAPBondMgr and the application is shown in [Figure 5-21](#).

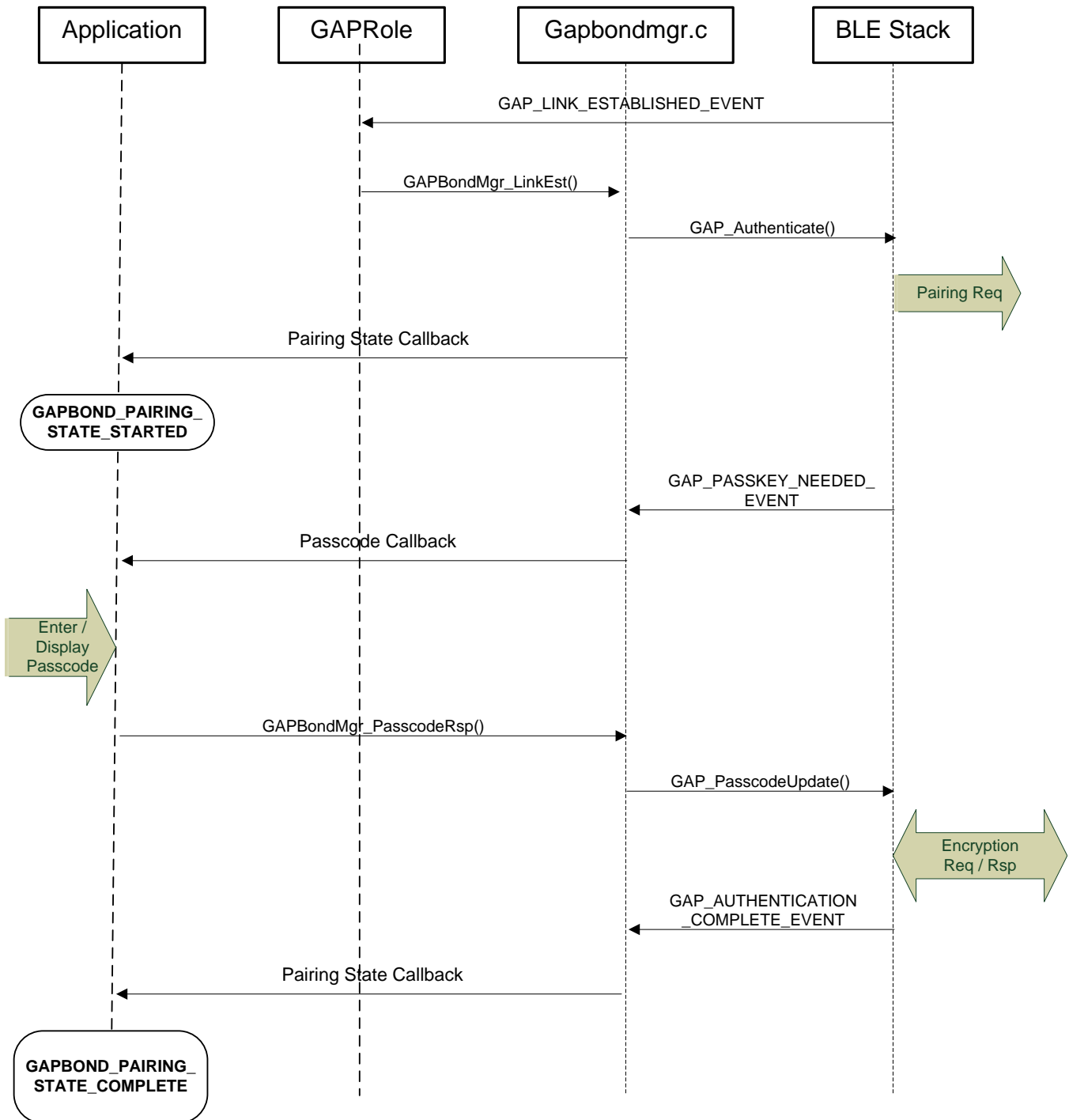


Figure 5-21. Interaction Between the GAPBondMgr and the Application

5.4.4.4 Numeric Comparison

Numeric comparison is a type of authenticated pairing that protects from MITM attacks. It is only possible as a Secure Connections pairing; not LE legacy. For numeric comparison pairing, both devices display a 6-digit code. Each device must then indicate, through a button press or some other yes-no input, whether the codes match. The passcode callback registered with the GAPBondMgr when it was started is used to display the 6-digit code. The following is an example of initiating Numeric Comparison pairing where the passcode is displayed. The IO capabilities must be set appropriately to select numeric comparison (that is, Display/Yes-No on both sides).

1. Define passcode callback to display code.

```
// Bond Manager Callbacks
static gapBondCBs_t SimpleBLECentral_bondCB =
{
    (pfnPasscodeCB_t)SimpleBLECentral_passcodeCB, // Passcode callback
    SimpleBLECentral_pairStateCB                // Pairing state callback
};

static void SimpleBLECentral_passcodeCB (uint8_t *deviceAddr, uint16_t connHandle, uint8_t
uiInputs, uint8_t uiOutputs, uint32_t numComparison)
{
    gapPasskeyNeededEvent_t *pData;

    // Allocate space for the passcode event.
    if ((pData = ICall_malloc(sizeof(gapPasskeyNeededEvent_t)))
    {
        memcpy(pData->deviceAddr, deviceAddr, B_ADDR_LEN);
        pData->connectionHandle = connHandle;
        pData->numComparison = numComparison;

        // Enqueue the event.
        security_examples_central_enqueueMsg(SEC_PASSCODE_NEEDED_EVT, 0, (uint8_t *) pData);
    }
}
```

2. Configure GAPBondMgr

```
uint8_t pairMode = GAPBOND_PAIRING_MODE_INITIATE;
uint8_t scMode = GAPBOND_SECURE_CONNECTION_ONLY;
uint8_t mitm = TRUE;
uint8_t ioCap = GAPBOND_IO_CAP_DISPLAY_YES_NO;
GAPBondMgr_SetParameter(GAPBOND_IO_CAPABILITIES, sizeof(uint8_t), &ioCap);
GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t), &pairMode);
GAPBondMgr_SetParameter(GAPBOND_MITM_PROTECTION, sizeof(uint8_t), &mitm);
GAPBondMgr_SetParameter(GAPBOND_SECURE_CONNECTION, sizeof(uint8_t), &scMode);
```

3. Process passcode callback and display code.

```
static void SimpleBLECentral_processPasscode (uint16_t connectionHandle,
gapPasskeyNeededEvent_t *pData)
{
    if (pData->numComparison) //numeric comparison
    {
        //Display passcode
        DISPLAY_WRITE_STRING_VALUE("Num Cmp: %d", pData->numComparison, LCD_PAGE4);
    }
}
```

4. Accept Yes-No input from user and send response to GAPBondMgr.

```
if (keys & KEY_RIGHT)
    GAPBondMgr_PasscodeRsp(connHandle, SUCCESS, TRUE);
DISPLAY_WRITE_STRING("Codes Match!", LCD_PAGE5);
return;
}
```

In this case, the third parameter of GAPBondMgr_PasscodeRsp, which usually accepts a passcode, is overloaded to send TRUE to the stack to indicate that the codes match and to continue with pairing.

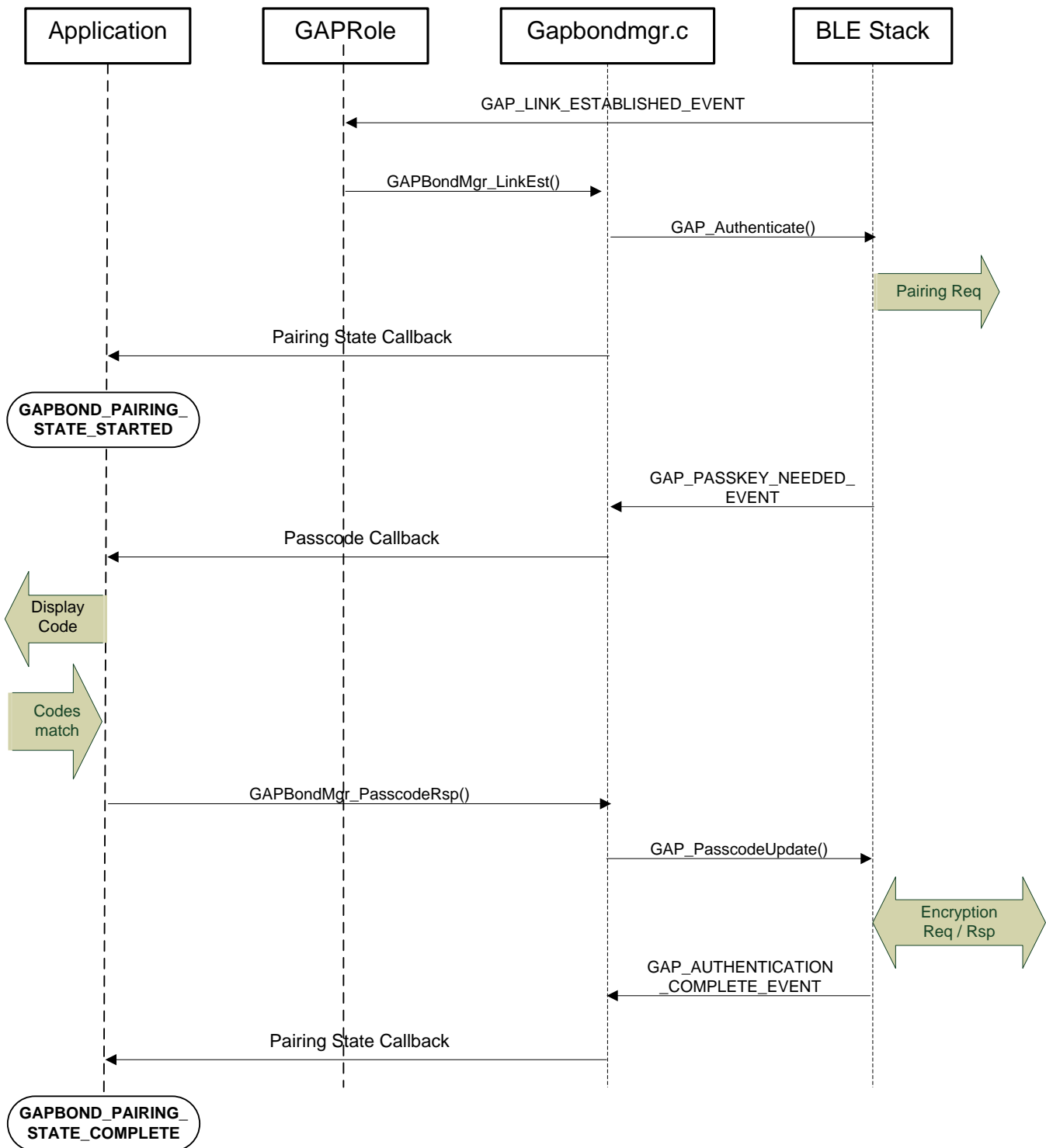


Figure 5-22. Numeric Comparison

5.4.4.5 GAPBondMgr Example With Bonding Enabled

Bonding can be enabled or disabled for any type of pairing through the `GAPBOND_BONDING_ENABLED` parameter, and occurs after the pairing process is complete. To enable bonding, configure the `GAPBondMgr` as follows:

```
uint8_t bonding = TRUE;  
GAPBondMgr_SetParameter(GAPBOND_BONDING_ENABLED, sizeof(uint8_t), &bonding);
```

With bonding enabled, the GAPBondMgr stores the long-term key transferred during the pairing process to SNV. See [Section 5.4.4.6](#) for more information. After this is completed, the application is notified through the GAPBOND_PAIRING_STATE_COMPLETE event. GAPBOND_PAIRING_STATE_COMPLETE is only passed to the application pair state callback when initially connecting, pairing, and bonding. For future connections to a bonded device, the security keys are loaded from flash, thus skipping the pairing process. In this case, only PAIRING_STATE_BONDED is passed to the application pair state callback. This is illustrated in [Figure 5-23](#).

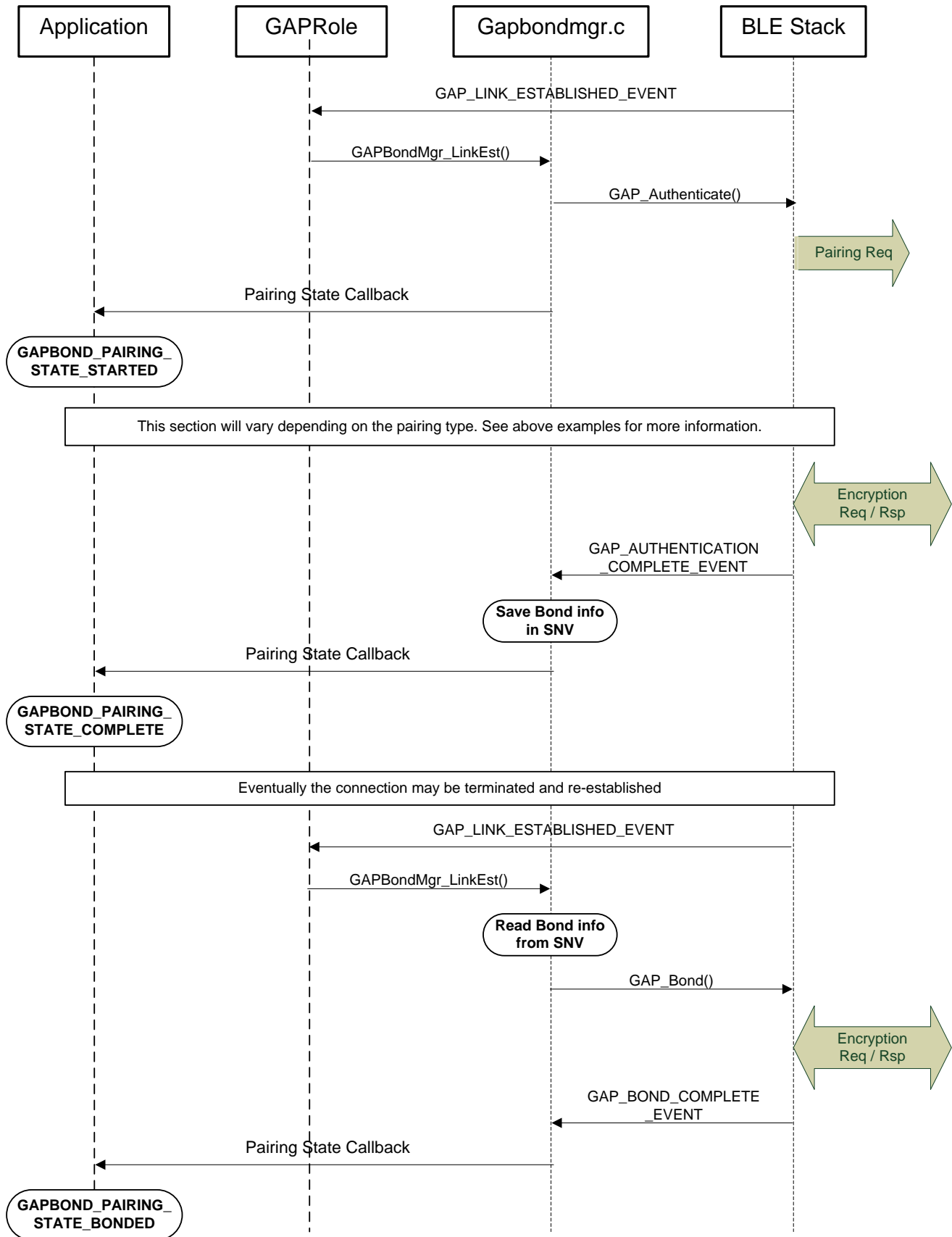


Figure 5-23. GAPBondMgr Example With Bonding Enabled

5.4.4.6 GAPBondMgr and SNV

This section describes how the GAPBondMgr uses the SNV flash area for storing bonding information. For more information on SNV itself, see [Section 3.10](#). The amount of bonds that can be stored is set by the GAP_BONDINGS_MAX definition, which is set to 10 by default in gapbondmgr.h. The functionality of the GAPBondMgr when there are no more available bonds varies based on whether the “least recently used” scheme is enabled. See [Appendix F](#) for more information on the GAPBOND_LRU_BOND_REPLACEMENT parameter. If this parameter is set to false, it is not possible to add any more bonds without manually deleting a bond. If the parameter is set to true, the least recently used bond is deleted to make room for the new bond.

The following components comprise one bonding entry:

1. Bond Record: this consists of the peer’s address, address type, privacy reconnection address, and state flags. This comprises 14 bytes and is defined as such:

```
typedef struct
{
    uint8    publicAddr[B_ADDR_LEN];    // Peer's address
    uint8    publicAddrType;           // Peer's address type
    uint8    reconnectAddr[B_ADDR_LEN]; // Privacy Reconnection Address
    uint8    stateFlags;               // State flags: @ref GAP_BONDED_STATE_FLAGS
} gapBondRec_t;
```

2. Client Characteristic Configurations (CCC): the amount of CCCs stored in each entry are set by the GAP_CHAR_CFG_MAX define. This is set to 4 by default. Each CCC is comprised of 4-bytes and is defined as follows:

```
typedef struct
{
    uint16 attrHandle; // attribute handle
    uint8  value;     // attribute value for this device
} gapBondCharCfg_t;
```

3. Local Long Term Key (LTK) info: this stores the local device’s encryption information. This comprises 28 bytes and is composed as such:

```
typedef struct
{
    uint8    LTK[KEYLEN];           // Long Term Key (LTK)
    uint16   div; //lint -e754      // LTK eDiv
    uint8    rand[B_RANDOM_NUM_SIZE]; // LTK random number
    uint8    keySize;              // LTK key size
} gapBondLTK_t;
```

4. Connected Device Long Term Key Info: this stores the connected device’s encryption information. This is also a gapBondLTK_t and comprises 28 bytes.
5. Connected Device Identity Resolving Key (IRK): this stores the IRK generated during pairing. This is a 16-byte array.
6. Connected Device Sign Resolving Key (SRK): this stores the SRK generated during pairing. This is a 16-byte array.
7. Connected Device Sign counter: this stores the sign counter generated during pairing. This is a 4-byte word.

5.4.5 LE Privacy 1.2

5.4.5.1 Summary

This BLE–Stack SDK supports the privacy feature that reduces the ability to track an LE device over a period of time, by changing the Bluetooth device address on a frequent basis. LE Privacy 1.2 extends privacy to the controller by allowing the controller to both resolve peer and generate local resolvable private addresses (RPAs). It is used during connection mode and connection procedures. [Table 5-2](#) lists the definition of terms related to the privacy feature.

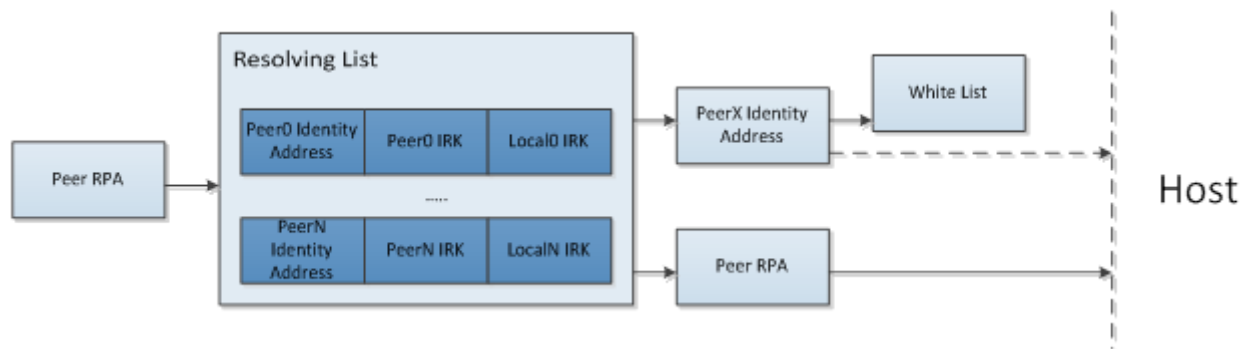
Table 5-2. Definition of Terms

| Term | Definition |
|------------------------|--|
| Resolvable address | A resolvable address is one that can potentially be resolved. Specifically, it is a device address that is a random resolvable private address (RPA). |
| Resolving list (RL) | One or more entries of local/peer IRK pairs associated with an identity address (public or random static). |
| Device address | A 48-bit value used to identify a device. A device address can be public or random. A device may use at least one, and can use both. |
| Identity (ID) address | An RPA is resolved with an identity resolving key (IRK) and is associated with a public address or a random static address, known as the identity (ID) address. |
| Non-resolvable address | A non-resolvable address is one that can never be resolved. Specifically, it is a device address that is a public address, a random static address, or a non-resolvable private address. |

5.4.5.2 Theory of Operation

For a device using the privacy feature to reconnect to known devices, the device address, referred to as the private address, must be resolvable by the other device. The private address is generated using the device's resolving identity key (IRK) exchanged during the bonding procedure.

With LE Privacy 1.2, the host is able to populate a resolving list in the controller. The resolving list consists of a set of records, where each record contains a pair of IRKs, one for local and one for peer, as well as the identity address of the peer device. A identity address of the peer device should be the public or static address of that device, which is obtained during phase 3 of pairing. The controller, which now contains all of the IRKs for previously bonded devices, is able to resolve private addresses into identity addresses of peers. These addresses are then able to be passed to the controller white list for further action, as shown in [Figure 5-24](#).


Figure 5-24. Resolving List

If the controller is unable to resolve the peer RPAs, or the white list takes no actions for the incoming address, the address is still passed to the host. If the local device or peer device wishes, it can initiate a bonding sequence to exchange IRKs as well as device identity addresses. If these are exchanged, the host can use those parameters to update the controller's resolving list, and even update the white list, so that the controller can automatically form a connection with the peer during a future connection attempt.

5.4.5.3 Enabling Privacy

Most of the privacy features are handled by the GAP bond manager in the stack. Privacy should be enabled in the stack by uncommenting the following line in `build_config.opt`:

```
-DBLE_V42_FEATURES=PRIVACY_1_2_CFG
```

5.4.5.4 New HCI Commands

The following new HCI commands are now supported in the controller:

- LE Add Device to Resolving List Command
- LE Remove Device to Resolving List Command
- LE Clear Resolving List Command
- LE Read Resolving List Size Command
- LE Read Peer Resolvable Address Command
- LE Read Local Resolvable Address Command
- LE Set Address Resolution Enable Command
- LE Set Random Private Address Timeout Command

For additional details, please see Bluetooth Core Specification, version 4.2 Vol 2, Part E, Section 7.8 for the commands, and section 7.7 for the event.

5.4.5.5 Privacy and White List

5.4.5.5.1 Enabling Auto Sync of White List

The stack can automatically add devices to the white list after bonding. Use the following code to enable this syncing of the white list.

```
uint8_t autoSyncWhiteList = TRUE;
GAPBondMgr_SetParameter(GAPBOND_AUTO_SYNC_WL, sizeof(uint8_t), &autoSyncWhiteList);

status = GAP_ConfigDeviceAddr(ADDRMODE_PRIVATE_RESOLVE, NULL);

//Set timeout value to 5 minutes
GAP_SetParamValue( TGAP_PRIVATE_ADDR_INT , 5);
```

5.4.5.5.2 Using Resolvable Private Addresses

The device also can be configured to use a random address. Use the following API to use random address:

```
status = GAP_ConfigDeviceAddr(ADDRMODE_PRIVATE_RESOLVE, NULL);
```

It can be verified with a sniffer that the address changes when advertising. The default timeout value between private (resolvable) address changes is 15 minutes. This can be modified by the `GAP_SetParamValue()` API:

```
//Set timeout value to 5 minute
GAP_SetParamValue( TGAP_PRIVATE_ADDR_INT , 5);
```

5.4.5.5.3 Testing Privacy with White List

The following steps can be made to test the privacy with white list feature:

1. Connect the iOS device to the CC2640 both supporting Privacy 1.2
2. Pair with the device with the default passcode: 000000
3. The iOS devices should be automatically added to the white list
4. Disconnect and wait for the iOS device address to change
5. Reconnect to the CC2640

5.5 Logical Link Control and Adaptation Layer Protocol (L2CAP)

The L2CAP layer sits on top of the HCI layer on the host side and transfers data between the upper layers of the host (GAP, GATT, application) and the lower layer protocol stack. This layer is responsible for protocol multiplexing capability, segmentation, and reassembly operation for data exchanged between the host and the protocol stack. L2CAP permits higher-level protocols and applications to transmit and receive upper layer data packets (L2CAP service data units, SDU) up to 64KB long. See [Figure 5-25](#) for more information.

NOTE: The actual size is limited by the amount of memory available on the specific device being implemented. L2CAP also permits per-channel flow control and retransmission.

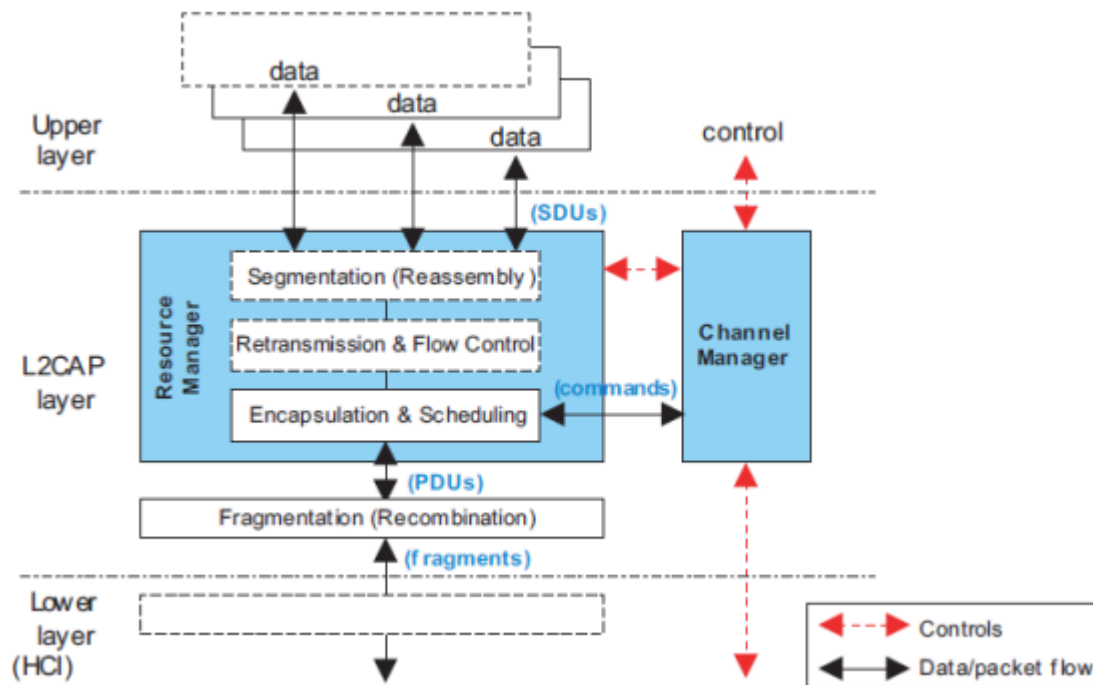


Figure 5-25. L2CAP Architectural Blocks

5.5.1 General L2CAP Terminology

| Term | Description |
|---------------------------------|--|
| L2CAP channel | The logical connection between two endpoints in peer devices, characterized by their Channel Identifiers (CIDs) |
| SDU or L2CAP SDU | Service Data Unit: a packet of data that L2CAP exchanges with the upper layer and transports transparently over an L2CAP channel using the procedures specified in this document |
| PDU or L2CAP PDU | Protocol Data Unit: a packet of data containing L2CAP protocol information fields, control information, and/or upper layer information data |
| Maximum Transmission Unit (MTU) | The maximum size of payload data, in octets, that the upper layer entity can accept (that is, the MTU corresponds to the maximum SDU size). |
| Maximum PDU Payload Size (MPS) | The maximum size of payload data in octets that the L2CAP layer entity can accept (that is, the MPS corresponds to the maximum PDU payload size). |

5.5.2 Maximum Transmission Unit (MTU)

The Bluetooth low energy stack supports fragmentation and recombination of L2CAP PDUs at the link layer. This fragmentation support allows L2CAP and higher-level protocols built on top of L2CAP, such as the attribute protocol (ATT), to use larger payload sizes, and reduce the overhead associated with larger data transactions. When fragmentation is used, larger packets are split into multiple link layer packets and reassembled by the link layer of the peer device. Figure 5-26 shows this relationship.

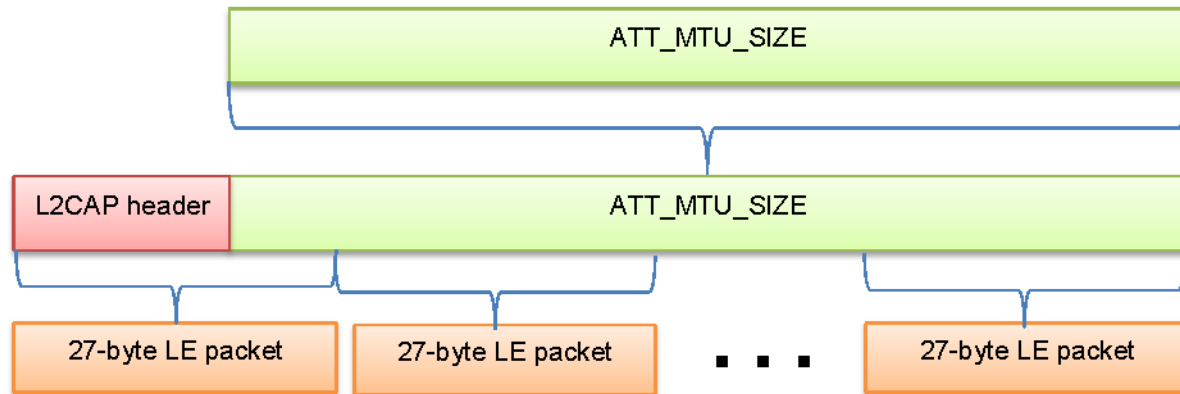


Figure 5-26. L2CAP Packet Fragmentation

The size of the L2CAP PDU also defines the size of the Attribute Protocol Maximum Transmission Unit (ATT_MTU). By default, LE devices assume the size of the L2CAP PDU is 27 bytes, which corresponds to the maximum size of the LE packet that can transmit in a single connection event packet. In this case, the L2CAP protocol header is 4 bytes, resulting in a default size for ATT_MTU of 23.

NOTE: When using the LE Data Length Extension feature, the length of the LE packet can be up to 251 bytes. See Section 5.6.

5.5.2.1 Configuring for Larger MTU Values

A client device can request a larger ATT_MTU during a connection by using the GATT_ExchangeMTU() command (see the API defined in Section D.3). During this procedure, the client (that is, Central) informs the server of its maximum supported receive MTU size and the server (that is, Peripheral) responds with its maximum supported receive MTU size. Only the client can initiate this procedure. When the messages are exchanged, the ATT_MTU for the duration of the connection is the minimum of the client MTU and server MTU values. If the client indicates it can support an MTU of 200 bytes and the server responds with a maximum size of 150 bytes, the ATT_MTU size is 150 for that connection. For more information, see the MTU Exchange section of Specification of the Bluetooth System, Covered Core Package, Version: 4.2.

Take the following steps to configure the stack to support larger MTU values.

1. Set the MAX_PDU_SIZE preprocessor symbol in the application project to the desired value (see Section 5.8) to the maximum desired size of the L2CAP PDU size. The maximum ATT_MTU size is always 4 bytes less than the value of the MAX_PDU_SIZE.
2. Call GATT_ExchangeMTU() after a connection is formed (GATT client only). The MTU parameter passed into this function must be less than or equal to the definition from step 1.
3. Receive the ATT_MTU_UPDATED_EVENT in the calling task to verify that the MTU was successfully updated. This update requires the calling task to have registered for GATT messages. See Section 5.3.6 for more information.

Though the stack can be configured to support a MAX_PDU_SIZE up to 255 bytes, each Bluetooth low energy connection initially uses the default 27 bytes (ATT_MTU = 23 bytes) value until the exchange MTU procedure results in a larger MTU size. The exchange MTU procedure must be performed on each Bluetooth low energy connection and must be initiated by the client.

Increasing the size of the ATT_MTU increases the amount of data that can be sent in a single ATT packet. The longest attribute that can be sent in a single packet is (ATT_MTU-1) bytes. Procedures, such as notifications, have additional length restrictions. If an attribute value has a length of 100 bytes, a read of this entire attribute requires a read request to obtain the first (ATT_MTU-1) bytes, followed by multiple read blob request to obtain the subsequent (ATT_MTU-1) bytes. To transfer the entire 100 bytes of payload data with the default ATT_MTU = 23 bytes, five request or response procedures are required, each returning 22 bytes. If the exchange MTU procedure was performed and an ATT_MTU was configured to 101 bytes (or greater), the entire 100 bytes could be read in a single read request or response procedure.

NOTE: Due to memory and processing limitations, not all Bluetooth low energy systems support larger MTU sizes. Know the capabilities of expected peer devices when defining the behavior of the system. If the capability of peer devices is unknown, design the system to work with the default 27-byte L2CAP PDU/23-byte ATT_MTU size. For example, sending notifications with a length greater than 20 bytes (ATT_MTU-3) bytes results in truncation of data on devices that do not support larger MTU sizes.

5.5.3 L2CAP Channels

L2CAP is based around channels. Each endpoint of an L2CAP channel is referred to by a channel identifier (CID). See Volume 3, Part A, Section 2.1 of the [Specification of the Bluetooth System, Covered Core Package, Version: 4.1](#) for more details on L2CAP Channel Identifiers. Channels can be divided into fixed and dynamic channels. For example, data exchanged over the GATT protocol uses channel 0x0004. A dynamically allocated CID is allocated to identify the logical link and the local endpoint. The local endpoint must be in the range from 0x0040 to 0xFFFF. This endpoint is used in the connection-orientated L2CAP channels described in the following section.

5.5.4 L2CAP Connection-Oriented Channel (CoC) Example

The Bluetooth low energy stack SDK provides APIs to create L2CAP CoC channels to transfer bidirectional data between two Bluetooth low energy devices supporting this feature. This feature is enabled by default in the protocol stack. [Figure 5-27](#) shows a sample connection and data exchange process between master and slave device using a L2CAP connection-oriented channel in LE Credit Based Flow Control Mode.

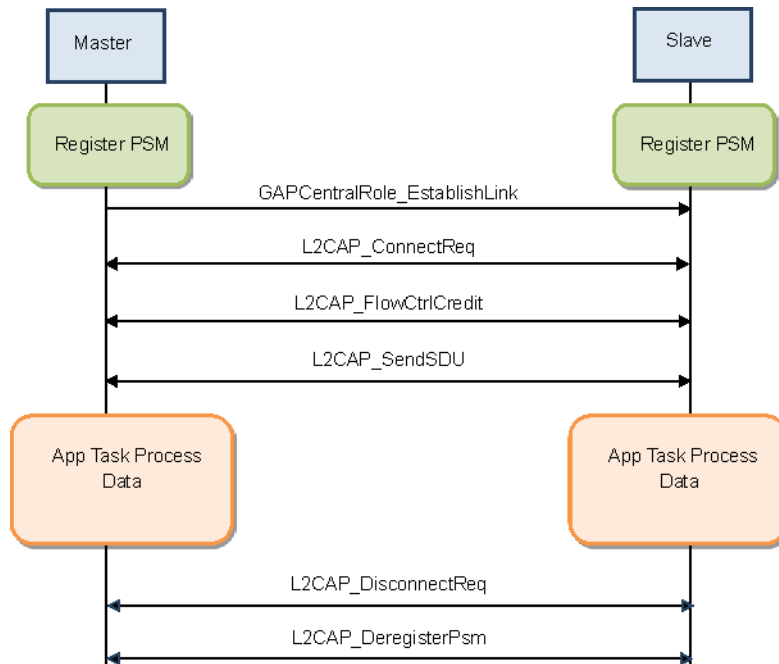


Figure 5-27. Sample Connection and Data Exchange Between a Master and Slave Device Using a L2CAP Connection-Oriented Channel in LE Credit Based Flow Control Mode

For more information on these L2CAP APIs, refer to the L2CAP API in [Appendix G](#).

5.6 LE Data Length Extension

5.6.1 Summary

The data length extension feature allows the LE controller to send data channel packet data units (PDUs) with payloads of up to 251 bytes of application data, while in the connected state. Furthermore, a new PDU size can be negotiated by either side at any time during a connection. Previously, the controller’s largest data channel payload was 27 bytes. This increases the data rate by around 2.5x, compared to Bluetooth Core Specification version 4.0 and 4.1 devices (if both devices support extended packet length and are configured properly).

5.6.2 Data Length Update Procedure

Once a connection is formed, the LE controllers of the device can use the LL_LENGTH_REQ and LL_LENGTH_RSP control PDUs to negotiate a larger payload size for data channel PDUs. A data length update may be initiated by the host or performed autonomously by the controller. Either the master or the slave can initiate the procedure.

After the data length update procedure is complete, both controllers select a new data length based on two parameters: PDU size and time. The largest size supported by both local and remote controller is selected; time is taken into account to support different data rates. These parameters are defined below:

- PDU size: The largest application data payload size supported by the controller. This size does not include packet overhead, such as access address or preamble.
- Time: The maximum number of microseconds that the device takes to transmit or receive a PDU at the PHY rate. This parameter uses units of microseconds (us).

Reference the Bluetooth Core Specification version 4.2 ([Vol 6], Part B, Section 5.1.9, Section 6.14) for more information about the data length update procedure. See [Table 5-3](#) for reference to the maximum sizes and times supported. TI’s CC2640/CC2650 supports these maximum values.

Table 5-3. Data Length Update Procedure Sizes and Times

| LE Data Packet Length Extension Feature Supported | Parameters with Names Ending in Octets | | Parameters with Names Ending in Time (µs) | |
|---|--|---------|---|---------|
| | Minimum | Maximum | Minimum | Maximum |
| No | 27 | 27 | 328 | 328 |
| Yes | 27 | 251 | 328 | 2120 |

5.6.3 Initial Values

The controller defaults to using PDU sizes compatible with 4.0 and 4.1 devices. It uses 27 bytes as its initial maximum size, and 328 us as the maximum time. The application can update the data length in two ways. First, the application can set the connection initial max octets to cause the controller to request a larger size for every connection. Second, the controller can initialize the connection with the default values of 27 octets and 328 us, then dynamically negotiate the data length at a later time in the connection.

For maximum throughput, high layer protocols such as the BLE host should also use a larger PDU size (see Section 5.5.2). Figure 5-28 illustrates various PDU sizes in the stack.

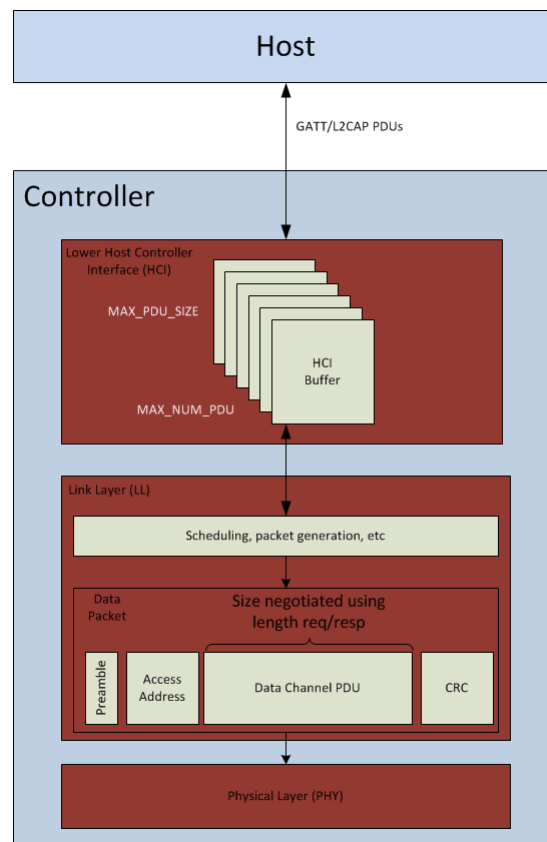


Figure 5-28. PDU Sizes

5.6.4 Data Length Extension HCI Commands and Events

The following HCI commands can be used to interact with the controller related to the data length extension feature:

- LE Read Suggested Default Data Length Command
- LE Write Suggested Default Data Length Command
- LE Read Maximum Data Length Command
- LE Set Data Length Command

The above commands may generate:

- LE Data Length Change Event

For more information about these HCI commands and their fields, see Bluetooth Core Specification version 4.2 ([Vol 2], Part E, Section 7.7-7.8). Additionally, the APIs for these new commands are documented in [Appendix H](#).

5.6.5 Enabling Extended Packet Length Feature

This section reviews how to enable and use data length extension in an application. This document uses simple_peripheral on the CC2650 LaunchPad as an example. The same principles should apply to other projects.

5.6.5.1 Enabling the Feature in the BLE-Stack

In the c2650lp_stack project, open the build_config.opt file. There is a list of BluetoothCore Specification version 4.2 features:

- /* BLE v4.2 Features */
- /* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+PRIVACY_1_2_CFG+EXT_DATA_LEN_CFG */
- /* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+PRIVACY_1_2_CFG */
- /* -DBLE_V42_FEATURES=PRIVACY_1_2_CFG+EXT_DATA_LEN_CFG */
- /* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+EXT_DATA_LEN_CFG */
- /* -DBLE_V42_FEATURES=SECURE_CONNS_CFG */
- /* -DBLE_V42_FEATURES=PRIVACY_1_2_CFG */
- -DBLE_V42_FEATURES=EXT_DATA_LEN_CFG

Uncomment a configuration that fits your needs; to use data length extension, a configuration with EXT_DATA_LEN_CFG must be selected.

5.6.5.2 Enable the Feature at Run Time

As per [Section 5.6.3](#), the LE controller initially uses packet length values compatible with 4.0 and 4.1 devices in new connections. Update these to trigger the controller to automatically attempt to negotiate a higher data length at the beginning of every new connection. To enable this feature, add the following call to the application task's initialization routine (such as simple_peripheral_init). Use valid values as shown in the table in [Section 5.6.2](#), otherwise the controller will reject this call.

```
#define APP_SUGGESTED_PDU_SIZE 251
#define APP_SUGGESTED_TX_TIME 2120
...
//This API is documented in hci.h
HCI_LE_WriteSuggestedDefaultDataLenCmd(APP_SUGGESTED_PDU_SIZE , APP_SUGGESTED_TX_TIME)
```

Once a connection is formed, the controller handles negotiating a packet size with the peer device. If both devices are set up to use data length extension, a throughput increase is observed.

5.6.5.3 Set Packet Length in a Connection

Packet length can also be changed dynamically in a connection using the below API snippet. The application can determine when this must occur based on any logic, such as sensor data or button presses.

```
uint16_t cxnHandle;
//Request max supported size
uint16_t requestedPDUSize = 251;
uint16_t requestedTxTime = 2120;
GAPRole_GetParameter(GAPROLE_CONNHANDLE, &cxnHandle);
//This API is documented in hci.h
if(SUCCESS != HCI_LE_SetDataLenCmd(cxnHandle, requestedPDUSize, requestedTxTime)
    DISPLAY_WRITE_STRING("Data length update failed", LCD_PAGE0);
```

5.7 HCI

The HCI layer is a thin layer which transports commands and events between the host and controller. In a pure network processor application (that is, the host_test project), the HCI layer is implemented through a transport protocol such as SPI or UART. In embedded wireless MCU projects or the simple_np project, the HCI layer is implemented through function calls and callbacks. All of the commands and events discussed so far, such as ATT, GAP, and so forth, pass from the given layer through the HCI layer to the controller. The controller sends data to the layers through the HCI layer.

As well as standard Bluetooth LE HCI commands, a number of HCI extension vendor-specific commands are available which extend some of the functionality of the controller for use by the application or host. See [Appendix H](#) for a description of available HCI and HCI extension commands.

5.7.1 Using HCI and HCI Vendor-Specific Commands in the Application

Follow these steps to use these commands and receive their respective events in the application:

1. Include the HCI transport layer header file.

```
#include "hci_tl.h"
```

2. Register with GAP for HCI/Host messages. This should be done in the application initialization function.

```
// Register with GAP for HCI/Host messages
GAP_RegisterForMsgs(selfEntity);
```

3. Call any HCI or HCI vendor-specific command from the application.
4. HCI events are returned as inter-task messages as a HCI_GAP_EVENT_EVENT. See the SimpleBLEPeripheral project for an example of this.

The following sections consider receiving HCI events and HCI vendor-specific events.

5.7.2 Standard LE HCI Commands

These commands are documented in Volume 0, Part A, Section 7 of the 4.2 Core Spec. The mechanism to use these commands is the same for any command in this section of the core spec, including HCI LE commands. The example below demonstrates how to use the core spec to implement an HCI command in the application. The command considered is Read RSSI Command.

5.7.2.1 Sending an HCI Command

1. Find the command in the core spec:

Read RSSI Command

| Command | OCF | Command Parameters | Return Parameters |
|---------------|--------|--------------------|-------------------------|
| HCI_Read_RSSI | 0x0005 | Handle | Status, Handle, RSSI |

Command Parameters:

Handle:

Size: 2 Octets (12 Bits meaningful)

| Value | Parameter Description |
|--------|--|
| 0xFFFF | The Handle for the connection for which the RSSI is to be read. The Handle is a Connection_Handle for a BR/EDR Controller and a Physical_Link_Handle for an AMP Controller. Range: 0x0000-0x0EFF (0x0F00 - 0x0FFF Reserved for future use) |

Figure 5-29. Read RSSI Command

2. Find mapping to BLE stack command. Using the table in [Appendix H](#), this command maps to HCI_ReadRssiCmd().
3. Using the API from Step 1, fill in the parameters and call the command from somewhere in the application. This specific command should be called after a connection is formed. There is only command parameter here: a 2-byte connection handle. In the case of this example, the connection handle is 0x0000:

```
HCI_ReadRssiCmd(0x0000);
```

5.7.2.2 Receiving HCI Events

1. Look at the core spec to see the format of the returned event:

Return Parameters:

Status: *Size: 1 Octet*

| Value | Parameter Description |
|-----------|---|
| 0x00 | Read_RSSI command succeeded. |
| 0x01-0xFF | Read_RSSI command failed. See Part D, Error Codes on page 370 for a list of error codes and descriptions. |

Handle: *Size: 2 Octets (12 Bits meaningful)*

| Value | Parameter Description |
|--------|--|
| 0xXXXX | The Handle for the connection for which the RSSI has been read. The Handle is a Connection_Handle for a BR/EDR Controller and a Physical_Link_Handle for an AMP Controller. Range: 0x0000-0x0EFF (0x0F00 - 0x0FFF Reserved for future use) |

RSSI: *Size: 1 Octet*

| Value | Parameter Description |
|-------|--|
| | BR/EDR Range: $-128 \leq N \leq 127$ (signed integer) Units: dB AMP: Range: AMP type specific (signed integer) Units: dBm LE: Range: -127 to 20, 127 (signed integer) Units: dBm |

Event(s) generated (unless masked away):

When the Read_RSSI command has completed, a Command Complete event shall be generated.

Figure 5-30. RSSI Event

2. This command returns a Command Complete event, so add this as a case in the processing of HCI_GAP_EVENT_EVENT:

```
static uint8_t SimpleBLEPeripheral_processStackMsg(ICall_Hdr *pMsg)
{
    uint8_t safeToDealloc = TRUE;

    switch (pMsg->event)
```

```

    {
        case HCI_GAP_EVENT_EVENT:
        {
            // Process HCI message
            switch(pMsg->status)
            {
                // Process HCI Command Complete Event
                case HCI_COMMAND_COMPLETE_EVENT_CODE:
                {
                    // Parse Command Complete Event for opcode and status
                    hciEvt_CmdComplete_t* command_complete = (hciEvt_CmdComplete_t*)pMsg;
                    uint8_t status = command_complete->pReturnParam[0];
                    //find which command this command complete is for
                    switch (command_complete->cmdOpcode)
                    {
                        case HCI_READ_RSSI:
                        {
                            if (status == SUCCESS)
                            {
                                uint16_t handle = BUILD_UINT16( command_complete->pReturnParam[2],
                                command_complete->pReturnParam[1]);
                                //check handle
                                if (handle == 0x00)
                                {
                                    //store RSSI
                                    uint8_t rssi = command_complete->pReturnParam[3];
                                }
                            }
                        }
                    }
                }
            }
        }
    }

```

First, the status of the stack message is checked to see what type of HCI event it is. In this case, it is an `HCI_COMMAND_COMPLETE_EVENT_CODE` (0x0E). Then the event returned from the stack as a message (`pMsg`) is cast to an (`hciEvt_CmdComplete_t*`), which is defined as:

```

// Command Complete Event
typedef struct
{
   osal_event_hdr_t hdr;
    uint8  numHciCmdPkt;
    uint16 cmdOpcode;
    uint8  *pReturnParam;
} hciEvt_CmdComplete_t;

```

Next, the `cmdOpcode` is checked and it is found that it matches `HCI_READ_RSSI` (0x1405). Then the status of the event is checked. The core spec API from above states that the first byte of the return parameters is the Status.

Then, check to see if this RSSI value is for the correct handle. The core spec API states that the second and third bytes of the return parameters are the Handle.

Finally, the RSSI value is stored. The core spec API states that the fourth byte of the return parameter is the RSSI.

5.7.3 HCI Vendor-Specific Commands

These commands are documented in the TI BLE Vendor-Specific HCI Guide included with the installer. The mechanism to use these commands is the same for all vendor-specific commands. The example below demonstrates how to use the core spec to implement an HCI command in the application. The command considered is Read RSSI Command.

5.7.3.1 Sending HCI Vendor-Specific Command

1. Find the command in the TI BLE vendor-specific guide:

HCI Extension Packet Error Rate

| Command | Opcode | Command Parameters | Return Parameters |
|----------------------------|--------|---------------------|-------------------|
| HCI_EXT_PacketErrorRateCmd | 0xFC14 | connHandle, command | Status |

Description

CC254x: ✓

CC264x: ✓

This command is used to Reset or Read the Packet Error Rate counters for a connection. When Reset, the counters are cleared; when Read, the total number of packets received, the number of packets received with a CRC error, the number of events, and the number of missed events are returned. The system default value upon hardware reset is *Reset*.

Note: The counters are only 16 bits. At the shortest connection interval, this provides a little over 8 minutes of data.

Note: This command is only valid for a valid connection handle (i.e. for an active connection). It is therefore not possible to read the packet error rate data once the connection has ended.

Command Parameters

connHandle: (2 bytes)

| Value | Parameter Description |
|------------------|--|
| 0x0000 .. 0x0EFF | Connection Handle to be used to identify a connection. Note: 0x0F00 – 0x0FFF are reserved for future use. |

command: (1 byte)

| Value | Parameter Description |
|-------|-----------------------|
| 0x00 | HCI_EXT_PER_RESET |
| 0x01 | HCI_EXT_PER_READ |

Return Parameters

Status: (1 byte)

| Value | Parameter Description |
|-------|-----------------------|
| 0x00 | HCI_SUCCESS |

Event(s) Generated

When the `HCI_EXT_PacketErrorRateCmd` has completed, a vendor specific Command Complete event shall be generated.

Figure 5-31. PER Command

- The BLE Stack function that implements this command is found under the Command column: `HCI_EXT_PacketErrorRateCmd`.
- Using the API from Step 1, fill in the parameters and call the command from somewhere in the application. The first parameter is a 2-byte `connHandle`, which is `0x00` for this example. The second parameter is a 1-byte `command`, which is `0x01`, to read the counters. Therefore, use:

```
HCI_EXT_PacketErrorRateCmd( 0, HCI_EXT_PER_READ );
```

5.7.3.2 Receiving HCI Vendor-Specific Events

- Find the corresponding event in the TI BLE Vendor-Specific HCI Guide:

HCI Extension Packet Error Rate

| Event | Opcode | Event Parameters |
|---------------------------|--------|---|
| HCI_Vendor_Specific_Event | 0x0414 | Status, cmdOpcode, cmdVal, numPkts, numCrcErr, numEvents, numMissedEvts |

Description

This event is sent to indicate the Packet Error Rate Reset or Read command has completed, or that there was an error.

Event Parameters

Status: (1 byte)

| Value | Parameter Description |
|-------|---------------------------------------|
| 0x00 | HCI_SUCCESS |
| 0x02 | HCI_ERROR_CODE_UNKNOWN_CONN_ID |
| 0x0C | HCI_ERROR_CODE_CMD_DISALLOWED |
| 0x12 | HCI_ERROR_CODE_INVALID_HCI_CMD_PARAMS |

cmdOpcode: (2 bytes)

| Value | Parameter Description |
|--------|---|
| 0xFC14 | HCI Extension Packet Error Rate Command |

cmdVal: (2 bytes)

| Value | Parameter Description |
|-------|-----------------------|
| 0x00 | HCI_EXT_PER_RESET |
| 0x01 | HCI_EXT_PER_READ |

Note: The following event parameters are for the Read command only.

numPkts: (2 bytes)

| Value | Parameter Description |
|------------------|-----------------------------------|
| 0x0000 .. 0xFFFF | Total number of received packets. |

numCrcErr: (2 bytes)

| Value | Parameter Description |
|------------------|--|
| 0x0000 .. 0xFFFF | Number of received packets with CRC error. |

numEvents: (2 bytes)

| Value | Parameter Description |
|------------------|------------------------------|
| 0x0000 .. 0xFFFF | Number of connection events. |

numMissedEvents: (2 bytes)

| Value | Parameter Description |
|------------------|-------------------------------------|
| 0x0000 .. 0xFFFF | Number of missed connection events. |

Figure 5-32. PER Event

2. As stated in the Events Generated section of the command API, this command returns a Command Complete event; thus add this as a case in the processing of HCI_GAP_EVENT_EVENT:

```
static uint8_t SimpleBLEPeripheral_processStackMsg(ICall_Hdr *pMsg)
{
    uint8_t safeToDealloc = TRUE;

    switch (pMsg->event)
    {
    case HCI_GAP_EVENT_EVENT:
        {
            // Process HCI message
            switch(pMsg->status)
            {
            // Process HCI Vendor Specific Command Complete Event
            case HCI_VE_EVENT_CODE:
                {
                    // Parse Command Complete Event for opcode and status
                    hciEvt_VSCmdComplete_t* command_complete = (hciEvt_VSCmdComplete_t*)pMsg;

                    // Find which command this command complete is for
                    switch (command_complete->cmdOpcode)
                    {
                    case HCI_EXT_PER:
                        {
                            uint8_t status = command_complete->pEventParam[2];
                            if (status == SUCCESS)
                            {
                                uint8_t cmdVal = command_complete->pEventParam[3];
                                if (cmdVal == 1) //if we were reading packet error rate
                                {
                                    uint16_t numPkts = BUILD_UINT16( command_complete->pEventParam[5],
                                                                    command_complete->pEventParam[4]);
                                    uint16_t numCrcErr = BUILD_UINT16( command_complete->pEventParam[7],
                                                                    command_complete->pEventParam[6]);
                                    uint16_t numEvents = BUILD_UINT16( command_complete->pEventParam[9],
                                                                    command_complete->pEventParam[8]);
                                    uint16_t numMissedEvents = BUILD_UINT16( command_complete->
                                                                    >pEventParam[11], command_complete->pEventParam[10]);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    ...
}
```

First, the status of the stack message is checked to see what type of HCI event it is. In this case, it is an HCI_VE_EVENT_CODE (0xFF).

Next, the event returned from the stack as a message (pMsg) is cast to an (hciEvt_VSCmdComplete_t*), which is defined as:

```
typedef struct
{
    osal_event_hdr_t  hdr;
    uint8  length;
    uint16  cmdOpcode;
    uint8  *pEventParam;
} hciEvt_VSCmdComplete_t;
```

The opcode is checked by reading command_complete->cmdOpcode, and found that it matches HCI_EXT_PER (0xFC14).

Next, the *pEventParam is parsed to extract the parameters defined in the event API. The first two bytes (shown in red in [Figure 5-33](#)) are the event opcode (0x1404). The third byte is the Status. This is the case for all vendor-specific events.

From the fourth byte of pEventParam on, the event API from the TI BLE Vendor-Specific Guide is used for parsing, starting at the third parameter. This is the case for all vendor-specific events. For this example, the fourth byte of pEventParam corresponds to the cmdVal parameter. This is shown in memory and explained further below.

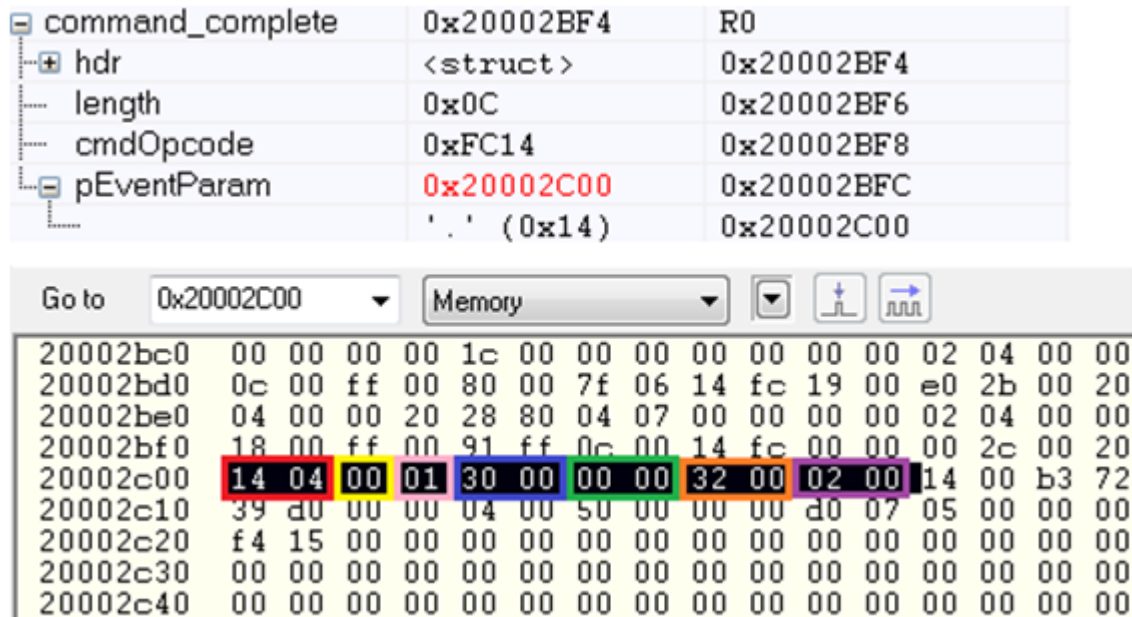


Figure 5-33. Memory Mapping

First the status is checked by reading the third byte of the event parameters (command_complete->pEventParam[2]). This is shown in yellow in [Figure 5-33](#).

Starting from the fourth byte of the event parameters (command_complete->pEventParam[3]), the event API states that the next parameter is a one-byte cmdVal. This is checked to verify that this event corresponds to a read of the PER counters. This is shown in pink.

Continuing parsing using the event API, the next parameter is a two-byte numPkts. This is found by building a uint16_t out of the fifth and sixth bytes of the event parameters. This is shown in blue. In a similar fashion, numCrcErr is found from the seventh and eighth bytes of the event parameters (shown in green).

Next, numEvents is found from the ninth and tenth bytes of the event parameters (shown in orange). Finally, numMissedEvents is found from the eleventh and twelfth bytes of the event parameters (shown in purple).

5.8 Run-Time Bluetooth low energy Protocol Stack Configuration

The Bluetooth low energy protocol stack can be configured with various parameters that control its runtime behavior and RF antenna layout. The available configuration parameters are described in the ble_user_config.h file in the ICallBLE IDE folder of the application. During initialization, these parameters are supplied to the Bluetooth low energy protocol stack by the user0Cfg structure, declared in main.c.

```
// BLE user defined configuration
bleUserCfg_t user0Cfg = BLE_USER_CFG;
```

Because the ble_user_config.h file is shared with projects within the SDK, TI recommends defining the configuration parameters in the preprocessor symbols of the application when using a nondefault value. For example, to change the maximum PDU size from the default 27 to 162, set the preprocessor symbol MAX_PDU_SIZE=162 in the preprocessor symbols for the application project. Increasing certain parameters may increase heap memory use by the protocol stack; adjust the HEAPMGR_SIZE as required (if not using auto sized heap). See [Section 9.2](#) lists the available configuration parameters.

Table 5-4. Bluetooth low energy Stack Configuration Parameters

| Parameter | Description |
|-----------------------|---|
| MAX_NUM_BLE_CONNS | Maximum number of simultaneous Bluetooth low energy connections. Default is 1 for Peripheral and Central roles. Maximum value is based on GAPRole. |
| MAX_NUM_PDU | Maximum number of Bluetooth low energy HCI PDUs. Default is 5. If the maximum number of connections is set to 0, then this number should also be set to 0. |
| MAX_PDU_SIZE | Maximum size in bytes of the Bluetooth low energy HCI PDU. Default is 27. Valid range is 27 to 255. The maximum ATT_MTU is MAX_PDU_SIZE - 4. See Section 5.5.2.1 . |
| L2CAP_NUM_PSM | Maximum number of L2CAP Protocol/Service Multiplexers (PSM). Default is 3. |
| L2CAP_NUM_CO_CHANNELS | Maximum number of L2CAP Connection-Oriented (CO) Channels. Default is 3. |
| PM_STARTUP_MARGIN | Defines time in microseconds (μ s) the system will wake up before the start of the connection event. Default is 300. This value is optimized for the example projects. |
| RF_FE_MODE_AND_BIAS | Defines the RF antenna front end and bias configuration. Set this value to match the actual hardware antenna layout. This value must be set directly in the ble_user_config.h file by adding a board-type preprocessor defined symbol. Default values are based on Evaluation Module (EM) boards. |

5.9 Configuring Bluetooth low energy Protocol Stack Features

The Bluetooth low energy protocol stack can be configured to include or exclude certain Bluetooth low energy features by changing the library configuration in the stack project. The available Bluetooth low energy features are defined in the build_config.opt file in the Tools folder of the stack project within the IDE. Based on the features selected in the build_config.opt file, the lib_search.exe tool selects the respective precompiled library during the build process of the stack project. [Table 5-5](#) lists a summary of configurable features. See the build_config.opt file for additional details and supported configurations.

NOTE: Selecting the correct stack configuration is essential in optimizing the amount of flash memory available to the application. To conserve memory, exclude certain Bluetooth low energy protocol stack features that may not be required.

Table 5-5. Bluetooth low energy Protocol Stack Features

| Feature | Description |
|------------------|---|
| HOST_CONFIG | This option configures the stack's host layer based on its targeted GAP role. These combo roles are supported: <ul style="list-style-type: none"> PERIPHERAL_CFG+OBSERVER_CFG CENTRAL_CFG+BROADCASTER_CFG PERIPHERAL_CFG+CENTRAL_CFG BROADCASTER_CFG+OBSERVER_CFG |
| BLE_V41_FEATURES | Setup the stack to use features from the Bluetooth Low Energy core specification v4.1. These features include: <ul style="list-style-type: none"> Ping Slave feature exchange Connection parameter update request Multirole connections |
| BLE_V42_FEATURES | Setup the stack to use features from the Bluetooth Low Energy core specification v4.2. These include: <ul style="list-style-type: none"> EXT_DATA_LEN_CFG SECURE_CONNS_CFG PRIVACY_1_2_CFG |
| L2CAP_COC_CFG | Includes support for L2CAP connection-oriented channels |

Table 5-5. Bluetooth low energy Protocol Stack Features (continued)

| Feature | Description |
|-------------|--|
| HCI_TL_xxxx | Include HCI Transport Layer (FULL, PTM or NONE). |

Peripherals and Drivers

The TI-RTOS provides a suite of CC26xx peripheral drivers that can be added to an application. The drivers provide a mechanism for the application to interface with the CC26xx onboard peripherals and communicate with external devices. These drivers make use of DriverLib to abstract register access.

There is significant documentation relating to each RTOS driver located at the RTOS installation path. Refer to the BLE-Stack release notes for the specific location. This section only provides an overview of how drivers fit into the software ecosystem. For a description of available features and driver APIs, refer to the TI-RTOS API Reference.

6.1 Adding a Driver

TI-RTOS drivers are added to the project as source files in their respective folder under the Drivers folder in the project workspace, as shown in [Figure 6-1](#).

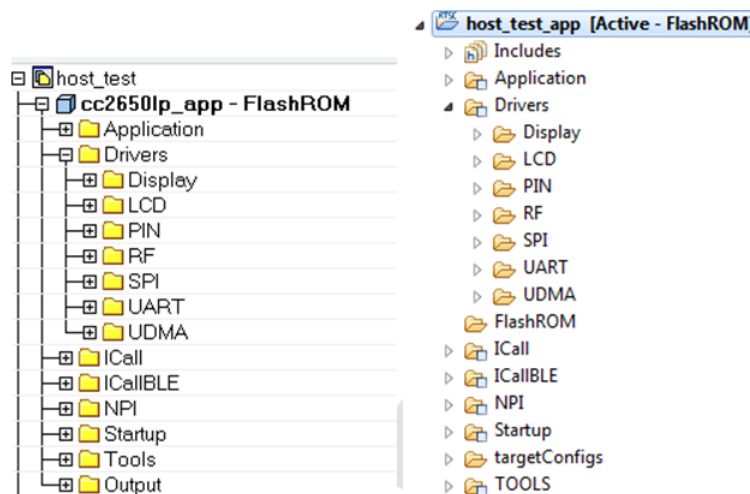


Figure 6-1. Drivers Folder

The driver source files can be found in their respective folder at `$TI_RTOS_DRIVERS_BASE$\ti\drivers`.

The `$TI_RTOS_DRIVERS_BASE$` argument variable refers to the installation location and can be viewed in IAR Tools→ Configure Custom Argument Variables menu. For CCS, the corresponding path variables are defined in the Project Options→ Resource→ Linked Resources, Path Variables tab.

To add a driver to a project, include the C and include file of respective driver in the application file (or files) where the driver APIs are referenced.

For example, to add the PIN driver for reading or controlling an output I/O pin, add the following:

```
#include <ti/drivers/pin/PINCC26XX.h>
```

Also add the following RTOS driver files to the project under the Drivers→PIN folder:

- PINCC26XX.c
- PINCC26XX.h
- PIN.h

This is described in more detail in the following sections.

6.2 Board File

The board file sets the parameters of the fixed driver configuration for a specific board configuration, such as configuring the GPIO table for the PIN driver or defining which pins are allocated to the I2C, SPI, or UART driver.

The board files for the SmartRF06 Evaluation Board are in the following path:

```
$TI_RTOS_DRIVERS_BASE$\ti\boards\SRF06EB\<Board_Type>
```

\$TI_RTOS_DRIVERS_BASE\$ is the path to the TI-RTOS driver installation and <Board_Type> is the actual Evaluation Module (EM). To view the actual path to the installed TI-RTOS version, see the following:

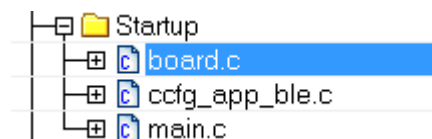
- IAR: Tools→ Configure Custom Argument Variables
- CCS: Project Options→ Resources→ Linked Resources, Path Variables tab

In the path above, the <Board_Type> is selected based on a preprocessor symbol in the application project, where the currently relevant options are:

- CC2650DK_7ID: 7x7 Evaluation Module
- CC2650DK_5XD: 5x5 Evaluation Module
- CC2650DK_4XS: 4x4 Evaluation Module
- CC2650STK: CC2650 Sensor Tag
- CC2650RC: Remote Control
- CC2650_LAUNCHXL: CC2650 LaunchPad
- BOOSTXL_CC2650MA: Booster Pack
- CC1310_LAUNCHXL: CC1310 LaunchPad
- CC1310DK_4XD: 4x4 Evaluation Module
- CC1310DK_5XD: 5x5 Evaluation Module
- CC1310DK_7XD: 7x7 Evaluation Module
- CC1350_LAUNCHXL: CC1350 LaunchPad
- CC1350 STK: CC1350 Sensor Tag

To set the board type (and thus choose a board file), define one of the above in the application preprocessor symbols.

The top-level board file (board.c) then uses this symbol to include the correct board file into the project. This top-level board file can be found at `$INSTALL$\src\components\hal\src\target\board.c`, and is located under the Startup folder in the project workspace:



The board file links in another gateway board file located at `$INSTALL$\src\components\hal\src\target\<board_type>`, which finally links in the actual board file from the RTOS install.

6.3 Board Level Drivers

There are also several board driver files which are a layer of abstraction on top of the RTOS drivers, to function for a specific board. For example, the `board_lcd.c` file is included in projects which use an evaluation module board file to use the LCD on the SmartRF06 board. `board_key.c` and `board_lcd.c` also provide similar functionality. If desired, these files can be adapted to work for a custom board.

6.4 Creating a Custom Board File

A custom board file must be created to design a project for a custom hardware board. TI recommends starting with an existing board file and modifying it as needed. The easiest way to add a custom board file to a project is to replace the top-level board file. If flexibility is desired to switch back to an included board file, the linking scheme defined in [Section 6.1](#) should be used.

At minimum, the board file must contain a PIN_Config structure that places all configured and unused pins in a default, safe state and defines the state when the pin is used. This structure is used to initialize the pins in main() as described in [Section 4.1](#). The board schematic layout must match the pin table for the custom board file. Improper pin configurations can lead to run-time exceptions.

```
PIN_init(BoardGpioInitTable);
```

See the PIN driver documentation for more information on configuring this table.

6.5 Available Drivers

This section describes each available driver and provide a basic example of adding the driver to the simple_peripheral project. For more detailed information on each driver, see the [TI-RTOS API Reference](#). Also, for a RTOS-only example (not considering the BLE stack), see the examples included with the RTOS at \$RTOS_INSTALL\$\tirtos_cc13xx_cc26xx_X_XX_XX_XX_examples.

6.5.1 PIN

The PIN driver allows control of the I/O pins for software-controlled general-purpose I/O (GPIO) or connections to hardware peripherals. The SimpleBLECentral or SensorTagExample projects use the PIN driver. As stated in [Section 6.2](#), the pins must first be initialized to a safe state (configured in the board file) in main(). After this initialization, any module can use the PIN driver to configure a set of pins for use. The following is an example of configuring the simple_peripheral task to use one pin as an interrupt and another as an output, to toggle when the interrupt occurs. IOID_x pin numbers map to DIO pin numbers as referenced in *TI CC26xx Technical Reference Manual (SWCU117)*. The following table lists pins used and their mapping on the Smart RF 06 board. These are already defined in the board file.

| Signal Name | Pin ID | SmartRF 06 Mapping: |
|--------------|---------|---------------------|
| Board_LED1 | IOID_25 | RF2.11 (LED1) |
| Board_KEY_UP | IOID_19 | RF1.10 (BTN_UP) |

The following simple_peripheral.c code modifications are required.

1. Include PIN driver files.

```
#include <ti/drivers/pin/PINCC26xx.h>
```

2. Declare the pin configuration table and pin state and handle variables to be used by the `simple_peripheral` task.

```
static PIN_Config SBP_configTable[] =
{
Board_LED1 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
  Board_KEY_UP | PIN_INPUT_EN | PIN_PULLUP | PIN_HYSTERESIS,
  PIN_TERMINATE
};

static PIN_State sbpPins;
static PIN_Handle hSbpPins;
static uint8_t LED_value = 0;
```

3. Declare the ISR to be performed in the hwi context.

```
static void buttonHwiFxn(PIN_Handle hPin, PIN_Id pinId);

static void buttonHwiFxn(PIN_Handle hPin, PIN_Id pinId)
{
  //set event in SBP task to process outside of hwi context
  events |= SBP_BTN_EVT;

  //wake up the application
  Semaphore_post(sem);
}
```

NOTE: This ISR is setting an event in the application task and waking it up to minimize processing in the hwi context.

4. Define the event and related processing (in `simple_peripheral_taskFxn()`) to handle the event from the above ISR.

```
#define SBP_BTN_EVT
if (events & SBP_BTN_EVT)
{
  events &= ~SBP_BTN_EVT; //clear event

  //toggle LED1
  if (LED_value)
  {
    PIN_setOutputValue(hSbpPins, Board_LED1 , LED_value--);
  }
  else
  {
    PIN_setOutputValue(hSbpPins, Board_LED1, LED_value++);
  }
}
```

5. Open the pins for use and configure the interrupt in `simple_peripheral_init()`.

```
// Open pin structure for use
hSbpPins = PIN_open(&sbpPins, SBP_configTable);
// Register ISR
PIN_registerIntCb(hSbpPins, buttonHwiFxn);
// Configure interrupt
PIN_setConfig(hSbpPins, PIN_BM_IRQ, Board_KEY_UP | PIN_IRQ_NEGEDGE);
// Enable wakeup
PIN_setConfig(hSbpPins, PINCC26XX_BM_WAKEUP, Board_KEY_UP|PINCC26XX_WAKEUP_NEGEDGE);
```

6. Compile
7. Download
8. Run

NOTE: Pushing the Up button on the SmartRF06 toggles LED1. No debouncing is implemented.

6.5.2 UART and SPI

There are many different methods of adding serial communication to a BLE project, and these are summarized in detail at the following wiki page:

http://processors.wiki.ti.com/index.php/CC2640_Serial_Communication.

6.5.3 Other Drivers

The other drivers included with the RTOS are: Crypto (AES), I2C, PDM, Power, RF, and UDMA. The stack makes use of the power, RF, and UDMA, so extra care must be taken if using these. As with the other drivers, these are well-documented, and examples are provided in the RTOS install [referenced here](#).

6.6 Using 32-kHz Crystal-Less Mode

BLE-Stack v2.2 adds support for operating the CC2640 (silicon Revision PG2.2 or later) in a 32-kHz crystal-less mode for peripheral and broadcaster (beacon) configurations. By using the internal low-frequency RC oscillator (RCOSC_LF), the 32-kHz crystal can be removed from the board layout. There are a few steps that must be taken to enable this feature. Refer to *Running Bluetooth Low Energy on CC2640 Without 32 kHz Crystal (SWRA499)* for additional details on this feature.

The `simple_peripheral` on the CC2650EM-7ID has a pre-built project configuration for using the RCOSC_LF. The following change is for IAR, see the app note mentioned above for changes in CCS. Follow these steps to enable this feature in the `simple_peripheral` project:

1. Select the `FlashROM_RCOSC` build configuration.
2. Exclude `Startup/ ccfg_app_ble.c` from build.
3. Include `Startup/ ccfg_app_ble_rcosc.c` in the build.
4. If using a custom board file, enable the RCOSC in the power policy. Otherwise, the board files included with TI-RTOS will:

```
PowerCC26XX_Config PowerCC26XX_config = {
    .policyInitFxn      = NULL,
    .policyFxn          = &PowerCC26XX_standbyPolicy,
    .calibrateFxn       = &PowerCC26XX_calibrate,
    .enablePolicy       = TRUE,
    .calibrateRCOSC_LF = TRUE,
    .calibrateRCOSC_HF = TRUE,
};
```

5. Constrain the temperature variation to be less than 1°C/sec. If the temperature is to change faster than 1°C/sec, then a short calibration interval must be used.

Calibration interval can be tuned in `rcosc_calibration.h`

```
// 1000 ms
#define RCOSC_CALIBRATION_PERIOD          1000
```

NOTE: Use of the internal RCOSC_LF requires a sleep clock accuracy (SCA) of 500 ppm.

Sensor Controller

The sensor controller engine (SCE) is an autonomous processor within the CC2640. The SCE can control the peripherals in the sensor controller independently of the main CPU. The main CPU does not have to wake up to (for example) execute an ADC sample or poll a digital sensor over SPI, and it saves both current and wake-up time that would otherwise be wasted. A PC tool lets you configure the sensor controller and choose which peripherals are controlled and which conditions wake up the main CPU. The sensor controller studio (SCS) is a stand-alone IDE to develop and compile microcode for execution on the SCE. Refer to [Sensor Controller Studio](#) for more details on the SCS, including documentation embedded within the SCS IDE.

Startup Sequence

For a complete description of the CC2640 reset sequence, see *TI CC26xx Technical Reference Manual (SWCU117)*.

8.1 Programming Internal Flash With the ROM Bootloader

The CC2640 internal flash memory can be programmed using the bootloader in the ROM of the device. Both UART and SPI protocols are supported. For more details on the programming protocol and requirements, see Chapter 9 of *TI CC26xx Technical Reference Manual (SWCU117)*.

NOTE: Because the ROM bootloader uses predefined DIO pins for internal flash programming, allocate these pins in the layout of your board. For details on the pins allocated to the bootloader based on the chip package type, see *TI CC26xx Technical Reference Manual (SWCU117)*.

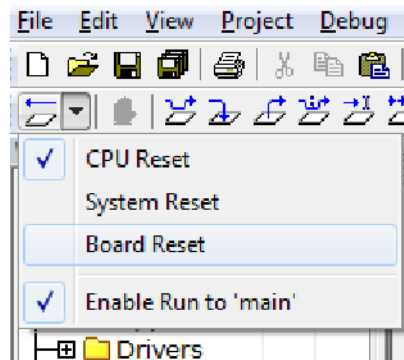
8.2 Resets

Use only hard resets to reset the device. From software, a reset can occur through one of the following.

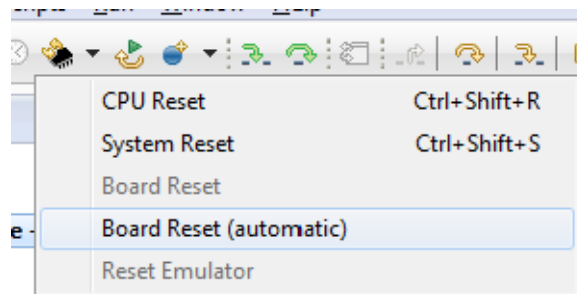
```
HCI_EXT_ResetSystemCmd(HC_EXT_RESET_SYSTEM_HARD);
```

```
HAL_SYSTEM_RESET();
```

In IAR, select the Board Reset option from the following Reset (back arrow) Debug Menu drop-down box.



In CCS, select Board Reset from the reset menu.





Development and Debugging

9.1 Debug Interfaces

The CC2640 platform supports the cJTAG (2-wire) and JTAG (4-wire) interfaces. Any debuggers that support cJTAG, like the TI XDS100v3 and XDS200, work natively. Other interfaces, like the IAR I-Jet and Segger J-Link, can only be used in JTAG mode but their drivers inject a cJTAG sequence which enables JTAG mode when connecting. The hardware resources included on the devices for debugging are listed as follows. Not all debugging functionality is available in all combinations of debugger and IDE.

- Breakpoint unit (FBP) – 6 instruction comparators, 2 literal comparators
- Data watchpoint unit (DWT) – 4 watchpoints on memory access
- Instrumentation Trace Module (ITM) – 32 × 32 bit stimulus registers
- Trace Port Interface Unit (TPIU) – serialization and time-stamping of DWT and ITM events

The SmartRF06 Board contains a XDS100v3 debug probe, and the CC2650 LaunchPad contains the XDS110 debug probe. These debuggers are used by default in the respective sample projects.

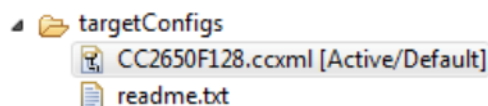
9.1.1 Connecting to the XDS Debugger

If only one debugger is attached, the IDE uses it automatically. If multiple debuggers are connected, you must choose the individual debugger. The following steps detail how to select a debugger in CCS and IAR.

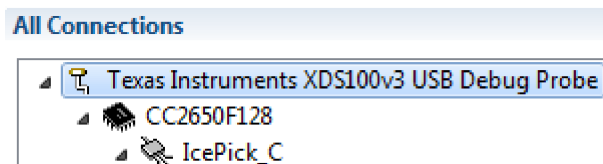
9.1.1.1 Debugging Using CCS

To debug using CCS, do as follows:

1. Open the target configuration file.
2. Open the Advanced pane.



3. Choose the top-level debugger entry.



4. Choose to select by serial number.
5. Enter the serial number.

Debug Probe Selection

-- Enter the serial number

Select by serial number

06EB12200DA5

To find the serial number for XDS100v3 debuggers, do as follows.

1. Open a command prompt.
2. Run `C:\ti\ccsv6\ccs_base\common\uscif\xds100serial.exe` to get a list of serial numbers of the connected debuggers.

9.1.1.2 Debugging Using IAR

To debug using IAR, do as follows.

1. Open the project options (Project→ Options).
2. Go to the Debugger entry.
3. Go to Extra options.
4. Add the following command line option: `--drv_communication=USB:#select`

Adding this command line option makes the IAR prompt which debugger to use for every connection.

9.2 Breakpoints

Both IAR and CCS reserve one of the instruction comparators. Five hardware breakpoints are available for debugging. This section describes setting breakpoints in IAR and CCS.

9.2.1 Breakpoints in CCS

To toggle a breakpoint, do any of the following.

- Double-click the area to the left of the line number.
- Press Ctrl+Shift+B.
- Right-click on the line.
 - Select Breakpoint→ Hardware Breakpoint.

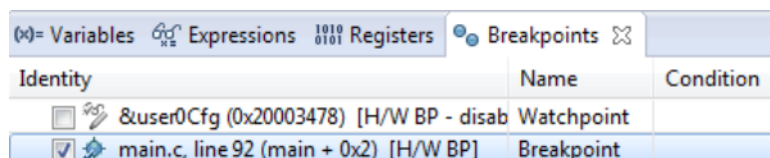
A breakpoint set on line 92 looks like the following.



```

90 Void main()
91
92 PIN_init(BoardGpioInitTable);
93

```

For an overview of the active and inactive breakpoints, click on View→ Breakpoints.



| Identity | Name | Condition |
|---|------------|-----------|
| <input type="checkbox"/>  &user0Cfg (0x20003478) [H/W BP - disab] | Watchpoint | |
| <input checked="" type="checkbox"/>  main.c, line 92 (main + 0x2) [H/W BP] | Breakpoint | |

To set a conditional break, do as follows.

1. Right-click the breakpoint in the overview.
2. Choose Properties.

When debugging, Skip Count and Condition can help skip a number of breaks or only break if a variable is a certain value.

NOTE: Conditional breaks require a debugger response and may halt the processor long enough to break. For example, a conditional break can break an active Bluetooth low energy connection if the condition is false or the skip count has yet to be reached.

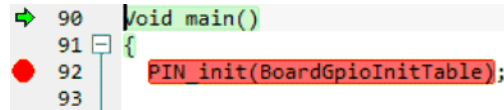
9.2.2 Breakpoints in IAR

To toggle a breakpoint, do any of the following.

- Single-click the area to the left of the line number.

- Go to the line.
 - Press F9.
- Right-click on the line.
 - Select Toggle Breakpoint (Code).

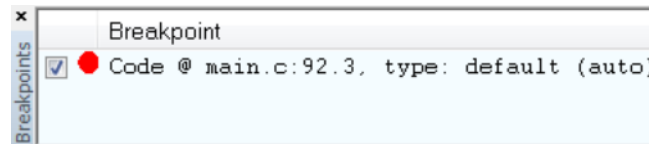
A breakpoint set on line 92 looks like the following.



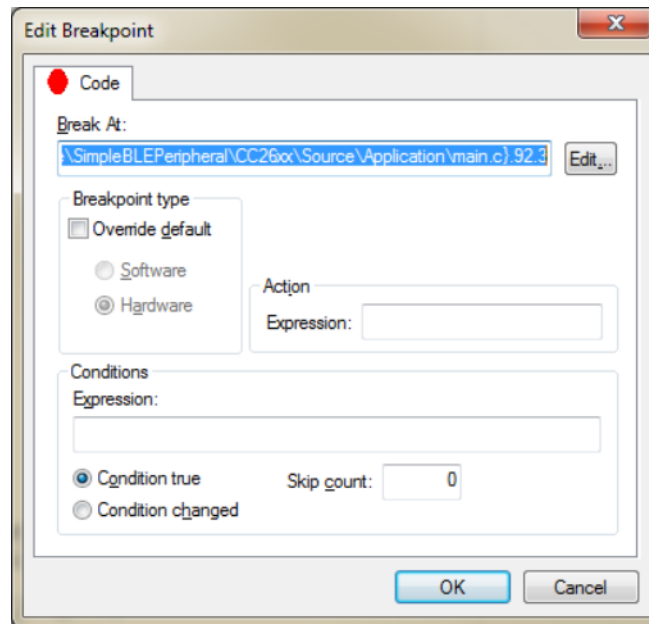
```

90 void main()
91 {
92     PIN_init(BoardGpioInitTable);
93
    
```

For an overview of the active and inactive breakpoints, click View→ Breakpoints.



To set a conditional break, do as follows.



1. Right-click the breakpoint in the overview.
2. Choose Edit....

When debugging, Skip Count and Condition can help skip a number of breaks or only break if a variable is a certain value.

NOTE: Conditional breaks require a debugger response and may halt the processor long enough to break. For example, a conditional break can break an active Bluetooth low energy connection if the condition is false or the skip count has not been reached.

9.2.3 Considerations When Using Breakpoints With an Active Bluetooth low energy

Connection

Because the Bluetooth low energy protocol is timing sensitive, any breakpoints break the execution long enough to lose network timing and break the link. Place breakpoints as close as possible to where the relevant debug information can be read or step through the relevant code segment to debug. This closeness also lets you experiment on breakpoint placements by restarting debugging and repeating the conditions that cause the code to hit the breakpoint.

9.2.4 Considerations no Breakpoints and Compiler Optimization

When the compiler is optimizing code, toggling a breakpoint on a line of C code may not result in the expected behavior. Some examples include the following.

- Code is removed or not compiled in: Toggling a breakpoint in the IDE results in a breakpoint some other unintended place and not on the selected line. Some IDEs disable breakpoints on nonexisting code.
- Code block is part of a common subexpression: For example, a breakpoint might toggle inside a function called from one other function, but can also break due to a call from another unintended function.
- An if clause is represented by a conditional branch in assembly: A breakpoint inside an if clause always breaks on the conditional statement, even if not executed.

TI recommends selecting an optimization level as low as possible when debugging. See [Section 9.4](#) for information on modifying optimization levels.

9.3 Watching Variables and Registers

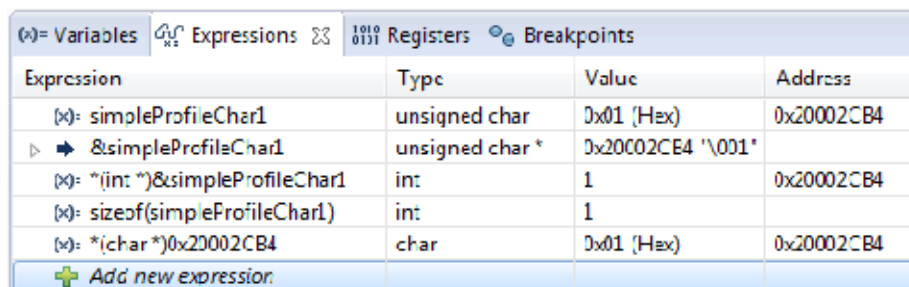
IAR and CCS provide several ways of viewing the state of a halted program. Global variables are statically placed during link-time and can end up anywhere in the RAM available to the project or potentially in flash if they are declared as a constant value. These variables can be accessed at any time through the Watch and Expression windows.

Unless removed due to optimizations, global variables are always available in these views. Local variables or variables that are only valid inside a limited scope are placed on the stack of the active task. Such variables can also be viewed with the Watch or Expression views, but can also be automatically displayed when breaking or stepping through code. To view the variables through IAR and CCS, do as follows.

9.3.1 Variables in CCS

You can view Global Variables by doing either of the following.

- Select View→ Expressions.
- Select a variable name in code.
 - Right-click and select Add Watch Expression.



| Expression | Type | Value | Address |
|----------------------------------|-----------------|-------------------|------------|
| (x): simpleProfileChar1 | unsigned char | 0x01 (Hex) | 0x20002CB4 |
| ▶ &simpleProfileChar1 | unsigned char * | 0x20002CE4 "\001" | |
| (x): *(in: *)&simpleProfileChar1 | int | 1 | 0x20002CB4 |
| (x): sizeof(simpleProfileChar1) | int | 1 | |
| (x): *(char*)0x20002CB4 | char | 0x01 (Hex) | 0x20002CB4 |
| + Add new expression. | | | |

Select View→ Variables to automatically viewed Local Variables.

| Name | Type | Value | Location |
|-------------------------|------------------|------------|------------|
| (x) charValue4 | unsigned char | . | 0x20002BBB |
| ▶ charValue5 | unsigned char[5] | 0x20002BBC | 0x20002BBC |
| (x) desiredConnTimeout | unsigned short | 1000 | 0x20002BAC |
| (x) desiredMaxInterval | unsigned short | 800 | 0x20002BA8 |
| (x) desiredMinInterval | unsigned short | 80 | 0x20002BA6 |
| (x) desiredSlaveLatency | unsigned short | 0 | 0x20002BAA |

9.3.2 Variables in IAR

To view Global Variables, do either of the following.

- Right-click on the variable.
 - Select Add to Watch: varName.
- Select View→ Watch n.
 - Enter the name of the variable.

| Expression | Value | Location | Type |
|----------------------------|--------------------|------------|--------------|
| simpleProfileChar1 | '.' (0x01) | 0x20000478 | uint8 |
| &simpleProfileChar1 | 0x20000478 ". . ." | | uint8 _data* |
| *(int*)&simpleProfileChar1 | 513 | 0x20000478 | int |
| sizeof(simpleProfileChar1) | 1 | | int |
| *(char*)0x20000478 | '.' (0x01) | 0x20000478 | char |
| <click to edit> | | | |

View→ Locals show the local variables in IAR.

```

454 {
455     uint8_t charValue1 = 1;
456     uint8_t charValue2 = 2;
457     uint8_t charValue3 = 3;
458     uint8_t charValue4 = 4;
459     uint8_t charValue5[SIMPLEPROFILE_CHARS_LEN] = { 1, 2, 3, 4, 5 };
460
461
462
463
464
465
466
467
468
469
470
471
    
```

| Variable | Value | Location | Type |
|------------|-------|------------|------------|
| charValue1 | 0x01 | 0x20001CE3 | uint8_t |
| charValue2 | 0x02 | 0x20001CE2 | uint8_t |
| charValue3 | 0x03 | 0x20001CE1 | uint8_t |
| charValue4 | 0x04 | 0x20001CE0 | uint8_t |
| charValue5 | <...> | 0x20001CF8 | uint8_t... |

9.3.3 Considerations When Viewing Variables

Local variables are often placed in CPU registers and not on the stack. These variables also have a limited lifetime even within the scope in which they are valid, depending on the optimization performed. Both CCS and IAR may struggle to show a particular variable due to its limited lifetime. The solution when debugging is as follows.

- Move the variable to global scope, so it remains accessible in RAM.
- Make the variable volatile, so the compiler fails to use a limited scope.
- Make a shadow copy of the variable that is global and volatile.

NOTE: IAR may remove the variable during optimization. If so, add the `__root` directive to volatile.

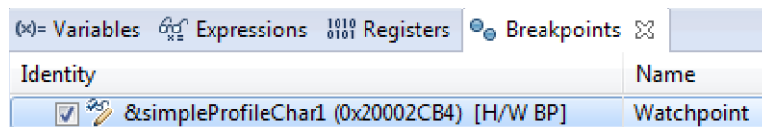
9.4 Memory Watchpoints

As mentioned in [Chapter 9](#), the DWT module contains four memory watchpoints that allow breakpoints on memory access. The hardware match functionality looks only at the address. If intended for use on a variable, the variable must be global. Using watchpoints is described for IAR and CCS as follows.

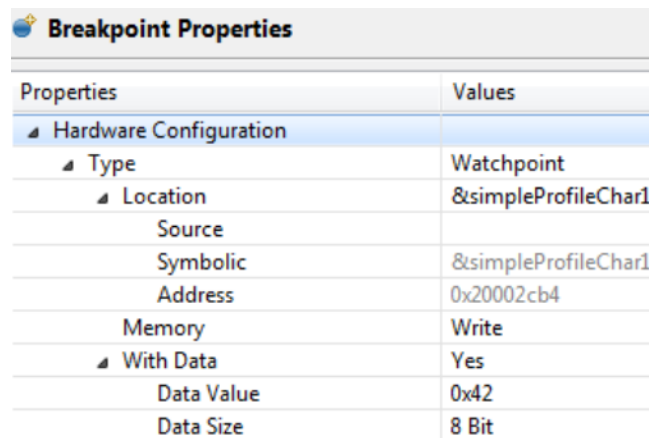
NOTE: If a data watchpoint with value match is used, two of the four watchpoints are used.

9.4.1 Watchpoints in CCS

1. Right-click on a global variable.
2. Select Breakpoint→ Hardware Watchpoint to add it to the breakpoint overview.



3. Right-click and edit the Breakpoint Properties to configure the watchpoint.

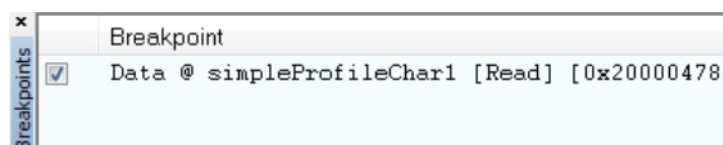


This example configuration ensures that if 0x42 is written to the memory location for Characteristic 1 in the Bluetooth low energy simple_peripheral example project. The device halts execution.

9.4.2 Watchpoints in IAR

NOTE: IAR currently does not support the watchpoint functionality with the XDS debuggers, but an IAR I-Jet can be used to accomplish this.

1. Right-click a variable.
2. Select Set Data Breakpoint for myVar to add it to the active breakpoints.



3. Right-click from the breakpoints view.
4. Choose Edit... to set up whether the watchpoint should match on read, write, or any access.

Figure 9-1 shows a break on read access when the value matches 0x42.

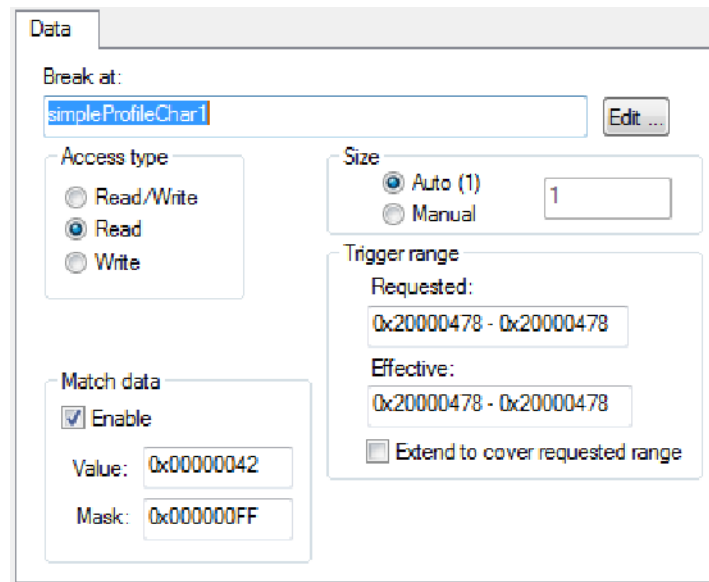


Figure 9-1. Break on Read Access

9.5 TI-RTOS Object Viewer

Both IAR and CCS include the RTOS Object Viewer (ROV) plug-in that provides insight into the current state of TI-RTOS, including task states, stacks, and so forth. Because both CCS and IAR have a similar interface, these examples discuss only CCS.

To access the ROV in IAR, do as follows.

1. Use the TI-RTOS menu on the menu bar.
2. Select a subview.

To access the ROV in CCS, do as follows.

1. Click the Tools menu.
2. Click RTOS Object View.

This section discusses some ROV views useful for debugging and profiling. More details can be found in the TI-RTOS User's Guide, including documentation on how to add log events to application code (see)

9.5.1 Scanning the BIOS for Errors

The BIOS→ Scan for errors view sweep through the available ROV modules and report any errors. This functionality can be a point to start if anything has gone unpredictably wrong. This scan only shows errors related to TI-RTOS modules and only the errors it can catch. See Figure 9-2.

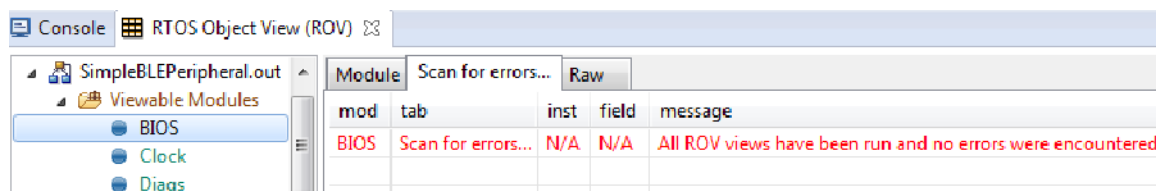


Figure 9-2. Error Scan

9.5.2 Viewing the State of Each Task

The Task→ Detailed view is useful for seeing the state of each task and its related runtime stack usage. This example shows the state the first time the user-thread is called. Figure 9-3 shows the Idle task, the GAPRole task, the simple_peripheral task, and the Bluetooth low energy stack task, represented by its ICall proxy.

| address | pri... | mode | fxn | arg0 | arg1 | stackPeak | stackSize | stackBase | blockedOn |
|------------|--------|---------|-----------------------------|--------|------------|-----------|-----------|------------|-----------------------|
| 0x20003794 | 0 | Ready | ti_sysbios_knl_idle_loop_E | 0x0 | 0x0 | 60 | 512 | 0x20002200 | |
| 0x20002968 | 3 | Blocked | gapRole_taskFxn | 0x0 | 0x0 | 608 | 816 | 0x200029b8 | Semaphore: 0x20001228 |
| 0x20002d7c | 1 | Running | SimpleBLEPeripheral_taskFxn | 0x0 | 0x0 | 640 | 896 | 0x20002dd0 | |
| 0x20000b80 | 5 | Blocked | ICall_taskEntry | 0xb001 | 0x20003a30 | 840 | 1200 | 0x20000bd0 | Semaphore: 0x200010b0 |

Figure 9-3. Viewing State of RTOS Tasks

The following list explains the column in Figure 9-3 (see Section 3.3 for more information on runtime stacks).

- address: This column shows the memory location of the Task_Struct instance for each task.
- priority: This column shows the TI-RTOS priority for the task.
- mode: This column shows the current state of the task.
- fxn: This column shows the name of the entry function of the task.
- arg0, arg1: These columns show arbitrary values that can be given to entry function of the task. In the image, the ICall_taskEntry is given 0xb001, which is the flash location of the entry function of the RF stack image and 0x20003a30 (the location of bleUserCfg_t user0Cfg, defined in main()).
- stackPeak: This column shows the maximum run-time stack memory used based on watermark in RAM, where the stacks are prefilled with 0xBE and there is a sentinel word at the end of the run-time stack.

NOTE: Function calls may push the stack pointer out of the run-time stack, but not actually write to the entire area. A stack peak near stackSize but not exceeding it may indicate stack overflow.

- stackSize: This column shows the size of the runtime stack, configured when instantiating a task.
- stackBase: This column shows the logical top of the runtime stack of the task (usage starts at stackBase + stackSize and grows down to this address).
- blockedOn: This column shows the type and address of the synchronization object; the thread is blocked on if available. For semaphores, the addresses are listed under Semaphore→ Basic.

9.5.3 Viewing the System Stack

The Hwi→ Module view allows profiling of the system stack used during boot or for main(), Hwi execution, and SWI execution. See Section 3.11.3 for more information on the system stack. For more information, see Figure 9-4 for the more details.

| address | options | activeInterrupt | pendingInterrupt | exception | hwiStackPeak | hwiStackSize | hwiStackBase |
|------------|---------|-----------------|------------------|-----------|--------------|--------------|--------------|
| 0x2000391c | ... | 0 | 18 | none | 428 | 1024 | 0x20003bfc |

Figure 9-4. Viewing the System Stack in Hwi

The hwiStackPeak, hwiStackSize, and hwiStackBase can be used to check for system stack overflow.

9.5.4 Viewing Power Manager Information

The Power→ Module view shows a value that is *logical or* of all the constraints currently enforced through the Power API. The value in the example (0x06) indicates Standby Disallow (0x4) and Shutdown Disallow (0x02); these numeric preprocessor symbols are subject to change. See [TI-RTOS Power Management for CC26xx](#) for more information.

9.6 Profiling the ICall Heap Manager (heapmgr.h)

As described in [Section 3.11.4](#), the ICall Heap Manager and its heap are used to allocate messages between the Bluetooth low energy stack task and the application task and as dynamic memory allocations in the tasks.

Profiling functionality is provided for the ICall heap but is not enabled by default. Therefore, it must be compiled in by adding HEAPMGR_METRICS to the defined preprocessor symbols. This functionality is useful for finding potential sources for unexplained behavior and to optimize the size of the heap. When HEAPMGR_METRICS is defined, the variables and functions listed as follows become available.

Global variables:

- heapmgrBlkMax: the maximum amount of simultaneous allocated blocks
- heapmgrBlkCnt: the current amount of allocated blocks
- heapmgrBlkFree: the current amount of free blocks
- heapmgrMemAlo: the current total memory allocated in bytes
- heapmgrMemMax: the maximum amount of simultaneous allocated memory in blocks (this value must not exceed the size of the heap)
- heapmgrMemUb: the furthest memory location of an allocated block, measured as an offset from the start of the heap
- heapmgrMemFail: the amount of memory allocation failure (instances where ICall_malloc() has returned NULL)

Functions:

- void ICall_heapGetMetrics(u16 *pBlkMax, u16 *pBlkCnt, u16 *pBlkFree, u16 *pMemAlo, u16 *pMemMax, u16 *pMemUb)
 - returns the previously described variables in the pointers passed in as parameters
- int heapmgrSanityCheck(void)
 - returns 0 if the heap is ok; otherwise, returns a nonzero (that is, an array access has overwritten a header in the heap)

9.6.1 Determining the Auto Heap Size

The following procedure can be used to view the size of the ICall heap when the auto heap size feature is enabled (HEAPMGR_SIZE=0).

At runtime, view the value of the global memory symbol HEAPMGR_SIZE after ICall_init() has been executed in main(). The value of HEAPMGR_SIZE is the total size of the ICall heap. See HEAPMGR_INIT() in heapmgr.h for the source code implementation.

In IAR: View -> Watch -> Watch 1, add HEAPMGR_SIZE

In CCS Debug Session: View -> Expressions, add HEAPMGR_SIZE

NOTE: The auto heap size feature does not determine the amount of heap needed for the application. The system designer must ensure that the heap has the required space to meet the application's runtime memory requirements.

To calculate the size of the ICall heap by inspecting the application map file in IAR:

The size of the ICall heap is the difference between the address of the last item in the .bss section and the start address of the system stack (CSTACK). For example, the *simple_peripheral_cc2650lp_app.map* file, at the end of the PLACEMENT SUMMARY:

```

.bss                zero      0x20001d32    0x1  driverlib_release.o [8]
                   - 0x20001d34  0x1b2c

"A4":
CSTACK              0x20003f68    0x400 <Block>
CSTACK              uninit    0x20003f68    0x400 <Block tail>
                   - 0x20004368    0x400

```

The size of the ICall heap in this example is the address of the start of CSTACK minus the address of the last item in .bss:

$0x20003f68 - 0x20001d34 = 0x2230$ or 8752 bytes

NOTE: Due to memory placement, the actual heap size may be up to 4 bytes less.

To calculate the size of the ICall heap by inspecting the application map file in CCS:

The size of the heap is determined by the heapStart and heapEnd global symbol addresses. For example, the *simple_peripheral_cc2650lp_app.map* file:

```

20003f68  heapEnd
20001dd2  heapStart

```

The size of the ICall heap in this example is defined as:

$0x20003f68 - 0x20001dd2 = 0x2196$ or 8598 bytes

NOTE: Due to memory placement, the actual heap size may be up to 4 bytes less.

9.7 Optimizations

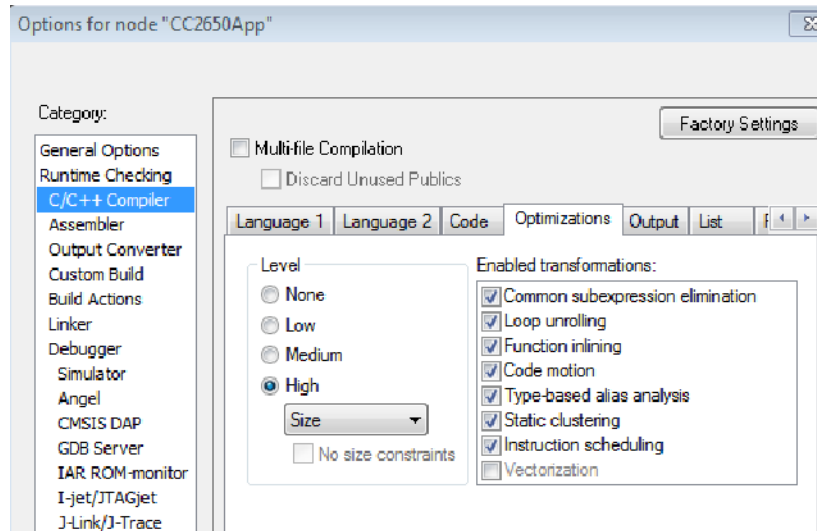
While debugging, turn off or lower optimizations to ease single-stepping through code. This optimization is possible at the following levels.

9.7.1 Project-Wide Optimizations

There may not be enough available flash to do project-wide optimizations. The following screen shots show how to set up optimization options for IAR and CCS for the entire project.

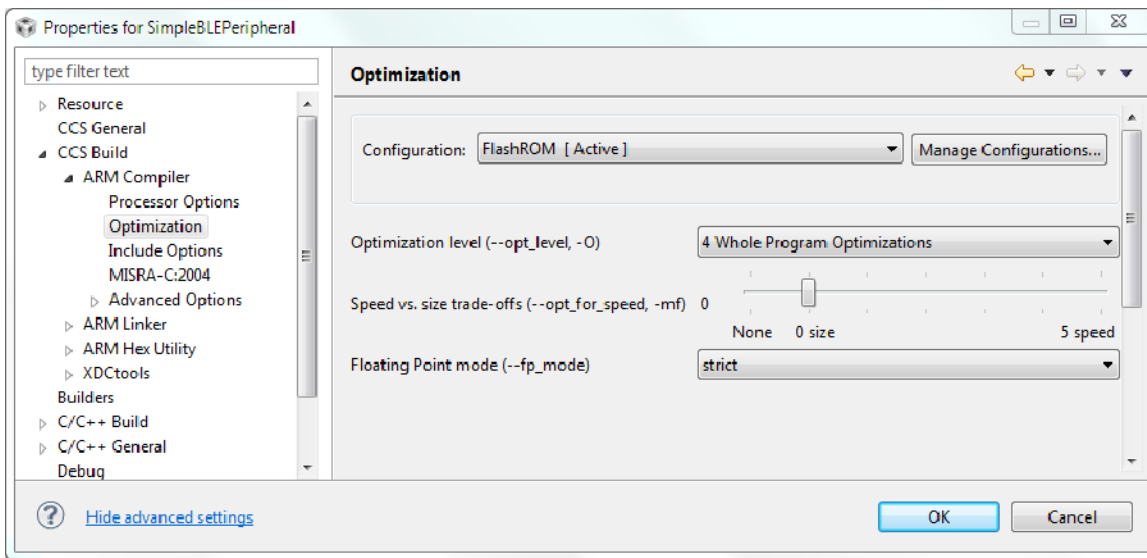
In IAR:

Project Options→ C/C++ Compiler→ Optimizations



In CCS:

Project Properties→ CCS Build→ ARM Compiler→ Optimization

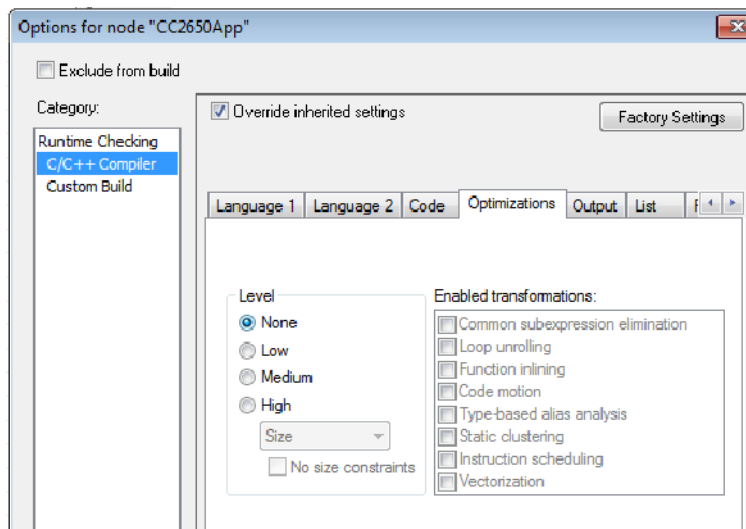


9.7.2 Single-File Optimizations

In IAR:

1. Right-click on the file in the Workspace pane.
2. Choose Options.
3. Check Override inherited Settings.
4. Choose the optimization level.

NOTE: Do single-file optimizations with care because this also overrides the project-wide preprocessor symbols.



In CCS:

1. Right-click on the file in the Workspace pane.
2. Choose Properties.
3. Change the optimization level of the file using the same menu in the CCS project-wide optimization menu.

9.7.3 Single-Function Optimizations

Using compiler directives, you can control the optimization level of a single function.

In IAR:

Use `#pragma optimize=none` before the function definition to deoptimize the entire function, that is, as follows.

```
#pragma optimize=none
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
    // Initialize application
    SimpleBLEPeripheral_init();

    // Application main loop
    for (;;)
    ...
```

In CCS:

```
#pragma FUNCTION_OPTIONS(SimpleBLEPeripheral_taskFxn, "--opt_level=0" )
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
    // Initialize application
    SimpleBLEPeripheral_init();
```

```
// Application main loop
for (;;)
...

```

9.7.4 Loading RTOS in ROM Symbols

Some RTOS code is contained in the ROM, used by the sample projects in the SDK that use the FlashROM builds. The ROM symbols can be loaded into the debugger to display further debugging information. For example, the ROM functions can be seen in the disassembly window when code is executing from ROM.

To load the ROM symbols in IAR:

1. Download

C:\TI\irtos_cc13xx_cc26xx_2_18_00_03\products\bios_6_45_02_31\packages\ti\sysbios\rom\cortexm\cc26xx\golden\CC26xx\rtos_rom_syms.xem3 in Project Options -> Debugger -> Images, with zero offset and Debug info only checked. See [Figure 9-5](#).

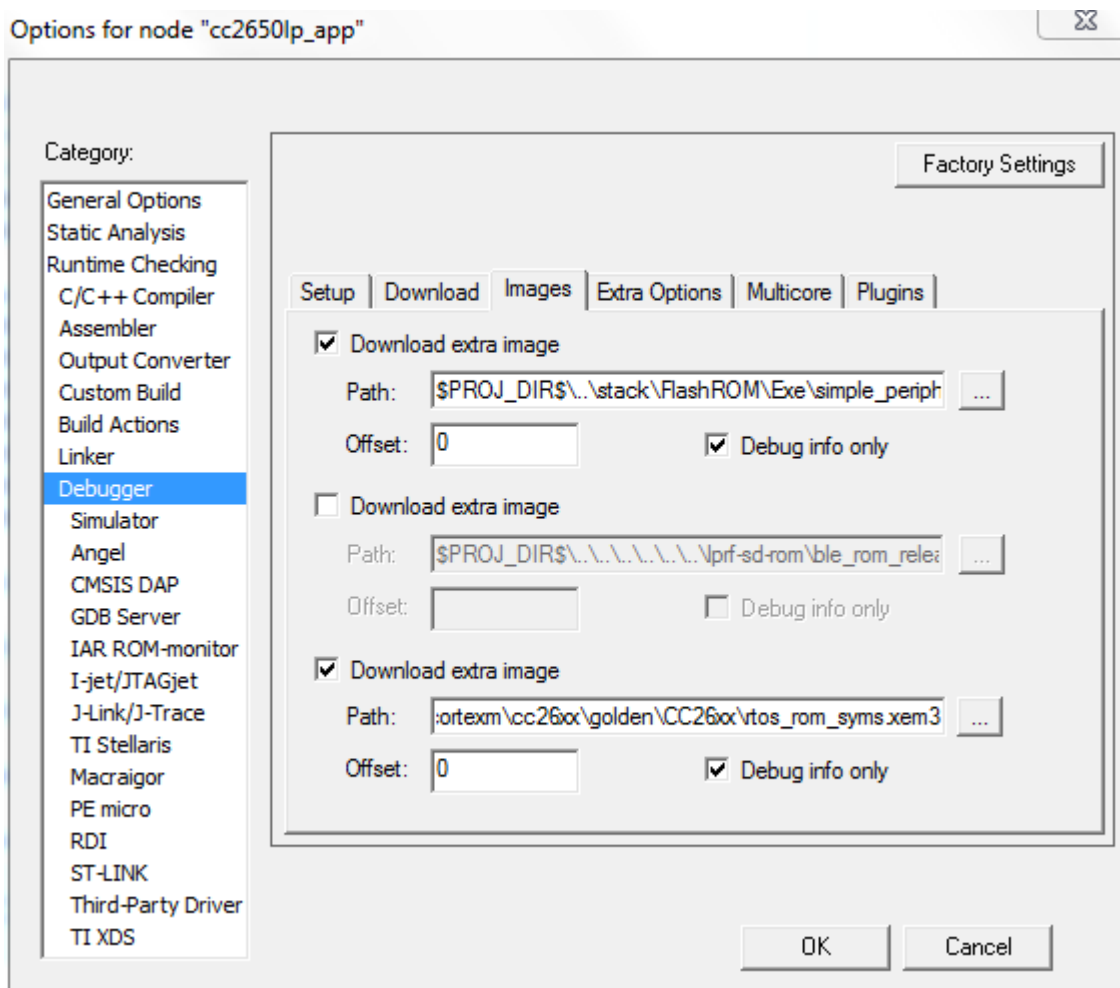


Figure 9-5. Adding RTOS ROM Symbol in IAR Project

2. Debug the project, and look at the disassembly to verify the extra ROM symbols. For example, notice the difference between two Disassembly with and without ROM symbols when an exception occurs, as in [Figure 9-6](#) and [Figure 9-7](#).

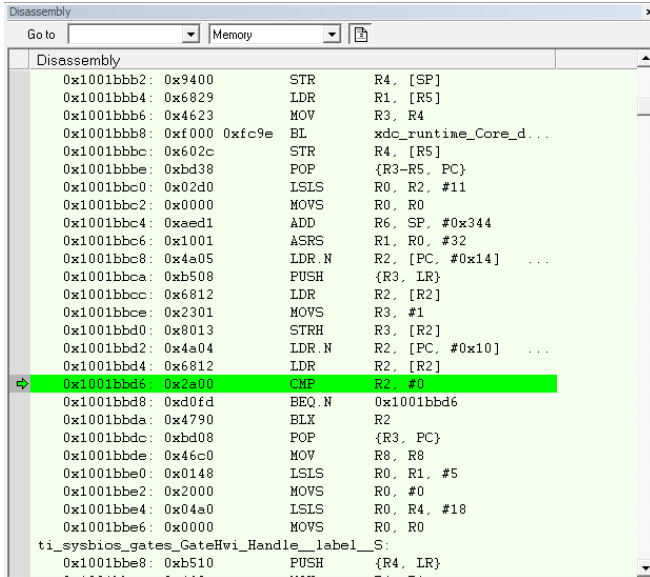


Figure 9-6. IAR Disassembly Without ROM Symbols

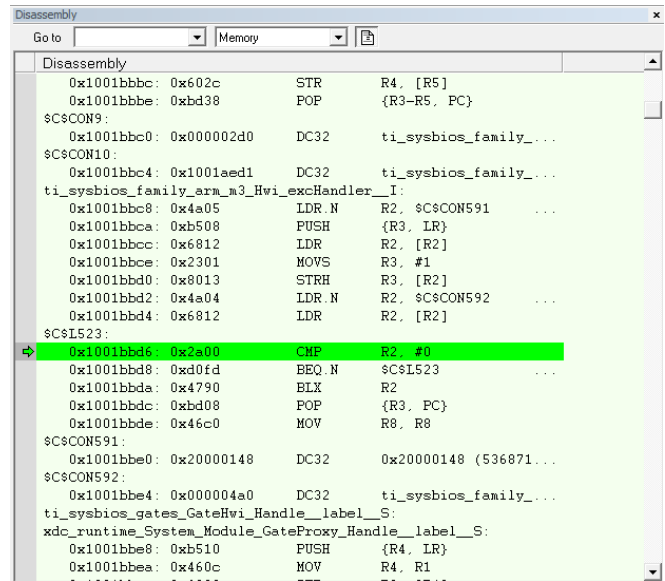


Figure 9-7. IAR Disassembly With ROM

To load the ROM symbols in CCS:

1. In CCS Debug mode, load `C:\TI\tirtos_cc13xx_cc26xx_2_18_00_03\products\bios_6_45_02_31\packages\ti\sysbios\rom\cortexm\cc26xx\golden\CC26xx\rtos_rom_syms.xem3` in Run -> Load -> Add Symbols...

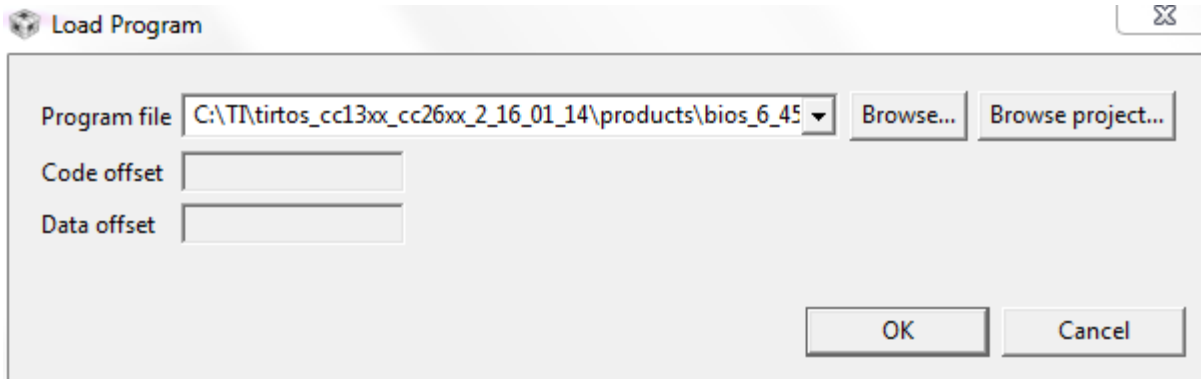
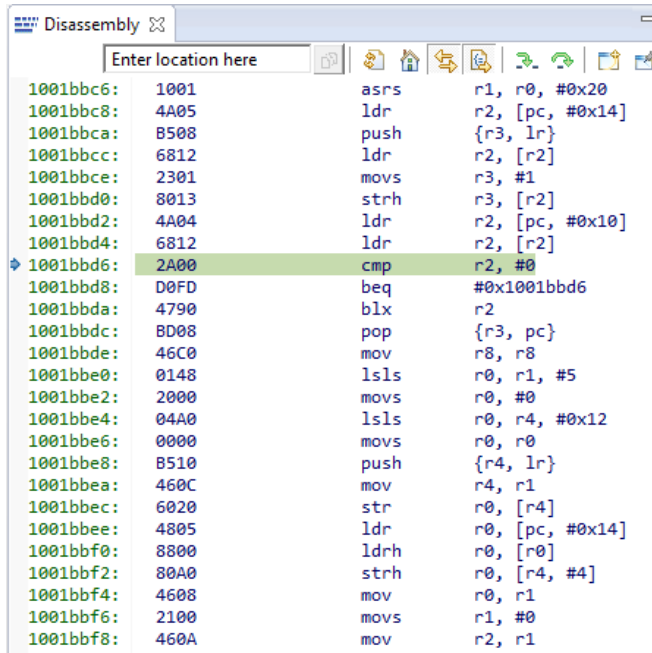


Figure 9-8. Adding RTOS ROM Symbol in CCS Project

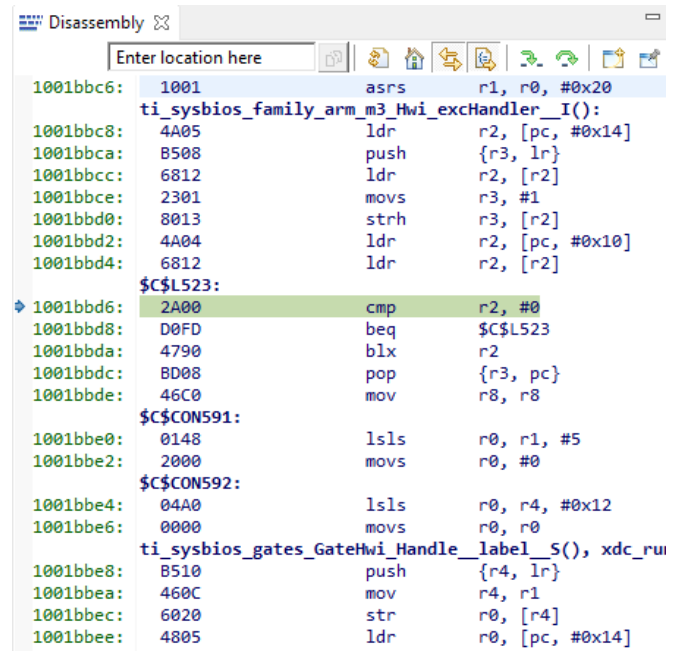
2. Observe the ROM functions in the Disassembly (View -> Disassembly); for example, when an exception occurs as shown in Figure 9-9 and Figure 9-10.



```

Disassembly
Enter location here
1001bbc6: 1001      asrs    r1, r0, #0x20
1001bbc8: 4A05      ldr     r2, [pc, #0x14]
1001bbca: B508      push   {r3, lr}
1001bbcc: 6812      ldr     r2, [r2]
1001bbce: 2301      movs   r3, #1
1001bbd0: 8013      strh   r3, [r2]
1001bbd2: 4A04      ldr     r2, [pc, #0x10]
1001bbd4: 6812      ldr     r2, [r2]
1001bbd6: 2A00      cmp     r2, #0
1001bbd8: D0FD      beq    #0x1001bbd6
1001bbda: 4790      blx    r2
1001bbdc: BD08      pop    {r3, pc}
1001bbde: 46C0      mov    r8, r8
1001bbe0: 0148      lsls  r0, r1, #5
1001bbe2: 2000      movs  r0, #0
1001bbe4: 04A0      lsls  r0, r4, #0x12
1001bbe6: 0000      movs  r0, r0
1001bbe8: B510      push  {r4, lr}
1001bbea: 460C      mov   r4, r1
1001bbec: 6020      str   r0, [r4]
1001bbee: 4805      ldr   r0, [pc, #0x14]
1001bbf0: 8800      ldrh  r0, [r0]
1001bbf2: 80A0      strh  r0, [r4, #4]
1001bbf4: 4608      mov   r0, r1
1001bbf6: 2100      movs  r1, #0
1001bbf8: 460A      mov   r2, r1
    
```

Figure 9-9. CCS Disassembly Without ROM Symbols



```

Disassembly
Enter location here
1001bbc6: 1001      asrs    r1, r0, #0x20
ti_sysbios_family_arm_m3_hwI_exHandler_I():
1001bbc8: 4A05      ldr     r2, [pc, #0x14]
1001bbca: B508      push   {r3, lr}
1001bbcc: 6812      ldr     r2, [r2]
1001bbce: 2301      movs   r3, #1
1001bbd0: 8013      strh   r3, [r2]
1001bbd2: 4A04      ldr     r2, [pc, #0x10]
1001bbd4: 6812      ldr     r2, [r2]
1001bbd6: 2A00      cmp     r2, #0
1001bbd8: D0FD      beq    $$L523
1001bbda: 4790      blx    r2
1001bbdc: BD08      pop    {r3, pc}
1001bbde: 46C0      mov    r8, r8
1001bbe0: 0148      lsls  r0, r1, #5
1001bbe2: 2000      movs  r0, #0
1001bbe4: 04A0      lsls  r0, r4, #0x12
1001bbe6: 0000      movs  r0, r0
1001bbe8: B510      push  {r4, lr}
1001bbea: 460C      mov   r4, r1
1001bbec: 6020      str   r0, [r4]
1001bbee: 4805      ldr   r0, [pc, #0x14]
1001bbf0: 8800      ldrh  r0, [r0]
1001bbf2: 80A0      strh  r0, [r4, #4]
1001bbf4: 4608      mov   r0, r1
1001bbf6: 2100      movs  r1, #0
1001bbf8: 460A      mov   r2, r1
    
```

Figure 9-10. CCS Disassembly With ROM Symbols

9.8 Deciphering CPU Exceptions

Several possible exception causes exist. If an exception is caught, an exception handler function can be called. Depending on the project settings, this handler may be a default handler in ROM, which is just an infinite loop or a custom function called from this default handler instead of a loop.

When an exception occurs, the exception may be caught and halted in debug mode immediately, depending on the debugger. If the execution halted manually later through the Break debugger, it is then stopped within the exception handler loop.

9.8.1 Exception Cause

With the default setup using TI-RTOS, the exception cause can be found in the System Control Space register group (CPU_SCS) in the register CFSR (Configurable Fault Status Register). The [ARM Cortex-M3 User Guide](#) describes this register. Most exception causes fall into the following three categories.

- Stack overflow or corruption leads to arbitrary code execution.
 - Almost any exception is possible.
- A NULL pointer has been dereferenced and written to.
 - Typically (IM)PRECISERR exceptions
- A peripheral module (like UART, Timer, and so forth) is accessed without being powered.
 - Typically (IM)PRECISERR exceptions

The CFSR is available in View→Registers in IAR and CCS.

When an access violation occurs, the exception type is IMPRECISERR because writes to flash and peripheral memory regions are mostly buffered writes.

If the CFSR:BFARVALID flag is set when the exception occurs (typical for PRECISERR), the BFAR register in CPU_SCS can be read out to find which memory address caused the exception.

If the exception is IMPRECISERR, PRECISERR can be forced by manually disabling buffered writes. Set [CPU_SCS:ACTRL:DISDEFWBUF] to 1, by either manually setting the bit in the register view in IAR/CCS or by including <inc/hw_cpu_scs.h> from Driverlib and calling the following.

```
HWREG(CPU_SCS_BASE + CPU_SCS_O_ACTLR) = CPU_SCS_ACTLR_DISDEFWBUF;
```

NOTE: This negatively affects performance.

9.8.2 Using TI-RTOS and ROV to Parse Exceptions

To enable exception decoding in the RTOS Object View without using too much memory, use the Minimal exception handler in TI-RTOS. The default choice in the BLE SDK projects is to use no exception handler.

To set this up, change the section of the TI-RTOS configuration file that relates to M3Hwi so that it looks like the code below:

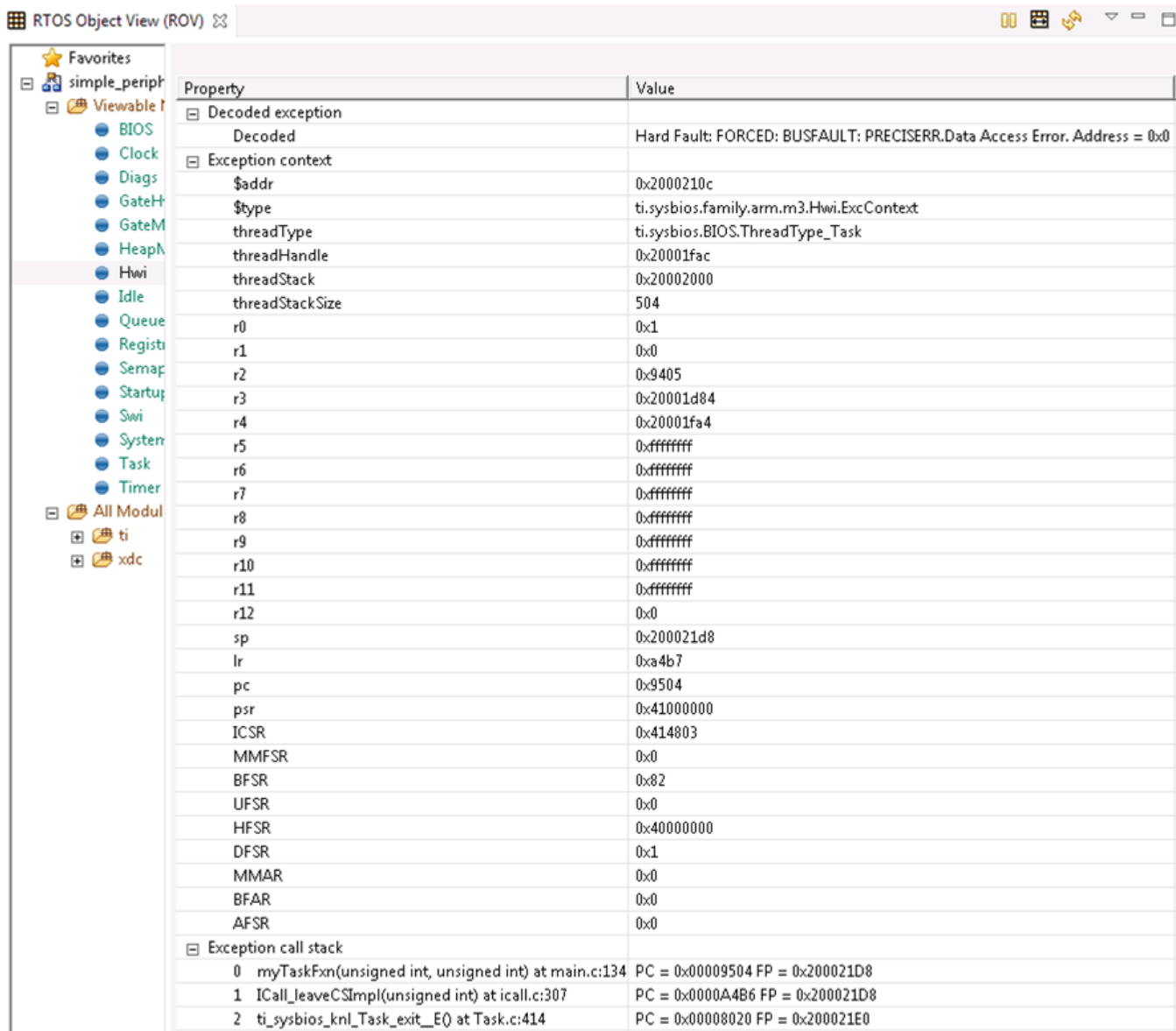
```
//m3Hwi.enableException = true;
m3Hwi.enableException = false;
//m3Hwi.excHandlerFunc = null;
m3Hwi.excHookFunc = "&execHandlerHook";
```

Then, make a function somewhere with the signature `Void (*Hwi_ExceptionHookFuncPtr)(Hwi_ExcContext*)`; such as the one below:

```
void execHandlerHook(Hwi_ExcContext *ctx)
{
    for(;;);
}
```

Setting `m3Hwi.enableException` to false enables the minimal handler, which fills out the global `Hwi_ExcContext` structure that the ROV looks at to show the decoded exception. By setting up an `excHookFunc`, the minimal exception handler will call this function and pass along a pointer to the exception context for the user to work with. This structure is defined in <ti/sysbios/family/arm/m3/Hwi.h>.

When an exception occurs, the device should end up in that infinite loop. Inspect the ROV -> Hwi -> Exception information as shown in Figure 9-11.



| Property | Value |
|----------------------|---|
| Decoded exception | Decoded |
| | Hard Fault: FORCED: BUSFAULT: PRECISERR.Data Access Error. Address = 0x0 |
| Exception context | |
| \$addr | 0x2000210c |
| \$type | ti.sysbios.family.arm.m3.Hwi.ExcContext |
| threadType | ti.sysbios.BIOS.ThreadType_Task |
| threadHandle | 0x20001fac |
| threadStack | 0x20002000 |
| threadStackSize | 504 |
| r0 | 0x1 |
| r1 | 0x0 |
| r2 | 0x9405 |
| r3 | 0x20001d84 |
| r4 | 0x20001fa4 |
| r5 | 0xffffffff |
| r6 | 0xffffffff |
| r7 | 0xffffffff |
| r8 | 0xffffffff |
| r9 | 0xffffffff |
| r10 | 0xffffffff |
| r11 | 0xffffffff |
| r12 | 0x0 |
| sp | 0x200021d8 |
| lr | 0xa4b7 |
| pc | 0x9504 |
| psr | 0x41000000 |
| ICSR | 0x414803 |
| MMFSR | 0x0 |
| BFSR | 0x82 |
| UFSR | 0x0 |
| HFSR | 0x40000000 |
| DFSR | 0x1 |
| MMAR | 0x0 |
| BFAR | 0x0 |
| AFSR | 0x0 |
| Exception call stack | |
| 0 | myTaskFxn(unsigned int, unsigned int) at main.c:134 PC = 0x00009504 FP = 0x200021D8 |
| 1 | ICall_leaveCSImpl(unsigned int) at icall.c:307 PC = 0x0000A4B6 FP = 0x200021D8 |
| 2 | ti_sysbios_knl_Task_exit_E() at Task.c:414 PC = 0x00008020 FP = 0x200021E0 |

Figure 9-11. Exception Information

In this case, a bus fault was forced in the function myTaskFxn by dereferencing address 0x0000 and trying to write to it:

```
*((uint8_t *)0) = 1;
```

This instruction was placed on line 134 of main.c, as indicated. To get a precise location, the write buffer was disabled as described earlier.

It can be instructive to look at the disassembly view for the locations specified by PC (program counter) and LR (link register). PC is the presumed exception location, and LR is normally the location the failing function should have returned to. As an example, the PC at this exception:

```

134      *((uint8_t *)0) = 1;
00009500:  2001      movs      r0, #1
00009502:  2100      movs      r1, #0
00009504:  7008      strb     r0, [r1]
  
```

Figure 9-12. PC Exception Example

As seen both from the disassembly and from the register printout above, the literal 1 was attempted to be stored at location 0, which is not allowed.

9.9 Debugging a Program Exit

The program must never exit the main() function. If this occurs, the disassembly looks like the following.

```

0x5628: 0x000051af DC32 ICall_pr
abort:
0x562c: 0x2001 MOVS R0, #1
0x562e: 0xf000 0xb801 B.W __exit
0x5632: 0x0000 MOVS R0, R0
__exit:
0x5634: 0x2120 MOVS R1, #32
0x5636: 0x0309 LSLS R1, R1,
0x5638: 0x3126 ADDS R1, R1,
0x563a: 0x2018 MOVS R0, #24
→ 0x563c: 0xbeab BKPT #0xab
0x563e: 0xe7f9 B.N __exit
?Whereas (6) for ti_ensight family arm m3 Hwi des
    
```

This code sequence can be seen in the disassembly when the ICall_abort() function is called, which can be caused by the following:

- Calling an ICall function from a stack callback
- Misconfiguring of additional ICall tasks or entities
- Incorrect ICall task registering

A breakpoint can be set in the ICall_abort function to trace from where this error is coming.

9.10 Assert Handling

Asserts can be useful when debugging, to trap undesirable states in the code. The SDK allows the application to catch asserts in the stack as well as application asserts.

9.10.1 Catching Stack Asserts in the Application

The application has an assert callback to catch asserts in the stack project. The assert callback is registered in main() function of each project.

```

/* Register Application callback to trap asserts raised in the Stack */
RegisterAssertCbback(AssertHandler);
    
```

Some generic assert causes that can be returned in the callback include HAL_ASSERT_CAUSE_TRUE, HAL_ASSERT_CAUSE_OUT_OF_MEMORY, and HAL_ASSERT_CAUSE_ICALL_ABORT. The user can decide how to handle these asserts in the callback. By default, it goes into spinlock for most of the asserts.

The assert can also define a subcause that gives a more specific reason for the assert. An example of a subcause is HAL_ASSERT_OUT_OF_HEAP, which describes the type of memory causing the assert for HAL_ASSERT_CAUSE_OUT_OF_MEMORY.

If no application callback is registered, the default assert callback is called and returns without further action unless HAL_ASSERT_SPIN is defined in the application project, which traps the application in an infinite while loop. In addition, one of the following can also be defined in the stack project if it is not caught in the application callback:

- HAL_ASSERT_RESET: Resets the device
- HAL_ASSERT_LIGHTS: Turn on the hazard lights (to be configured by user)
- HAL_ASSERT_SPIN: Spinlock in a while loop indefinitely

Enable these by ensuring that one of the above corresponding symbols are defined in the preprocessor symbols.

See hal_assert.h and hal_assert.c in the stack project for implementation details.

9.10.2 Catching App Asserts in the Application

One assert handler can trap asserts in the stack, while another handler traps asserts in the application. [Section 9.10.1](#) describes how to register a handler for stack asserts.

Follow these steps to catch app asserts in application:

1. Include `$BLE_INSTALL\src\components\hal\src\common\hal_assert.c` in the application project.
2. Define `EXT_HAL_ASSERT` in the application project.
3. Set the app assert handler after the stack has started, as in the example below:

```
static void SimpleBLEPeripheral_init(void)
{
    int dividend = 1;
    int divisor = 0;
    int quotient;
    // *****
    // NO STACK API CALLS CAN OCCUR BEFORE THIS CALL TO ICall_registerApp
    // *****
    // Register the current thread as an ICall dispatcher application
    // so that the application can send and receive messages.
    ICall_registerApp(&selfEntity, &sem);
    halAssertInit( AppAssertHandler, HAL_ASSERT_LEGACY_MODE_DISABLED );

    if (divisor == 0)
    {
        HAL_ASSERT_SET_SUBCAUSE( HAL_ASSERT_CAUSE_DIV_BY_ZERO );
        HAL_ASSERT( HAL_ASSERT_CAUSE_ARITHMETIC_ERROR );
    }
    quotient = dividend/divisor;
    ...
}
```

4. The assert handler in the application should catch the assert, as in the example below of `AppAssertHandler` defined in `simple_peripheral.c`:

```
void AppAssertHandler(uint8 assertCause, uint8 assertSubcause)
{
    Board_openDisplay(BOARD_DISPLAY_TYPE_LCD);
    Board_writeString(">>APP ASSERT", 0);
    // check the assert cause
    switch (assertCause)
    {
        case HAL_ASSERT_CAUSE_ARITHMETIC_ERROR:
            if (assertSubcause == HAL_ASSERT_CAUSE_DIV_BY_ZERO)
            {
                Board_writeString("***ERROR***", 1);
                Board_writeString(">> DIV_BY_ZERO ERROR!", 2);
            }
            else
            {
                Board_writeString("***ERROR***", 1);
                Board_writeString(">> ARITHMETIC ERROR!", 2);
            }
            break;
        default:
            Board_writeString("***ERROR***", 1);
            Board_writeString(">> DEFAULT SPINLOCK!", 2);
            HAL_ASSERT_SPINLOCK;
    }
    return;
}
```

9.11 Debugging Memory Problems

This section describes how to debug a situation where the program runs out of memory, either on the heap or on the runtime stack for the individual thread contexts. Exceeding array bounds or dynamically allocating too little memory for a structure corrupts the memory and can cause an exception like INVPC, INVSTATE, IBUSERR to appear in the CFSR register.

9.11.1 Task and System Stack Overflow

If an overflow on the runtime stack of the task or the system stack occurs (as found using the ROV plug-in as in [Section 9.5.2](#) and [Section 9.5.3](#)), perform the following steps.

1. Note the current size of the runtime stack of each task.
2. Increase by a few 100 bytes as described in [Section 3.3.1](#) and [Section 3.11.3](#).
3. Reduce the runtime stack sizes so that they are larger than their respective stackPeaks to save some memory.

9.11.2 Dynamic Allocation Errors

[Section 9.6](#) describes how to use the ICall Heap profiling functionality. To check if dynamic allocation errors occurred, do as follows:

1. Determine if memAlo or memMax approaches the preprocessor-defined HEAPMGR_SIZE.
2. Check memFail to see if allocation failures have occurred.
3. Call the sanity check function.

If the heap is sane but there are allocation errors, increase HEAPMGR_SIZE and see if the problem continues.

You can set a breakpoint in heapmgr.h in HEAPMGR_MALLOC() on the line `hdr = NULL;` to find the allocation that is failing.

9.12 Preprocessor Options

Preprocessor symbols configure system behavior, features, and resource usage at compile time. Some symbols are required as part of the Bluetooth low energy system, while others are configurable. See [Section 2.7](#) for details on accessing preprocessor symbols within the IDE. Symbols defined in a particular project are defined in all files within the project.

9.12.1 Modifying

To disable a symbol, put an `x` in front of the name. To disable power management, change `POWER_SAVING` to `xPOWER_SAVING`.

9.12.2 Options

[Table 9-1](#) lists the preprocessor symbols used by the application in the `simple_peripheral` project. Symbols that must remain unmodified are marked with an *N* in the Modify column while modifiable; configurable symbols are marked with a *Y*.

Table 9-1. Application Preprocessor Symbols

| Preprocessor Symbol | Description | Modify |
|-----------------------|---|--------|
| USE_ICALL | Required to use ICall Bluetooth low energy and primitive services. | N |
| POWER_SAVING | Power management is enabled when defined, and disabled when not defined. Requires same option in stack project. | Y |
| HEAPMGR_SIZE=0 | Defines the size in bytes of the ICall heap. Memory is allocated in .bss section. When zero, the heap is auto sized. See Section 3.11.4 . | Y |
| ICALL_MAX_NUM_TASKS=3 | Defines the number of ICall aware tasks. Modify only if adding a new RTOS task that uses ICall services. | Y |

Table 9-1. Application Preprocessor Symbols (continued)

| Preprocessor Symbol | Description | Modify |
|--------------------------------|--|--------|
| ICALL_MAX_NUM_ENTITIES=6 | Defines maximum number of entities that use ICall, including service entities and application entities. Modify only if adding a new RTOS task that uses ICall services. | Y |
| Display_DISABLE_ALL | All Display statements are removed and no display operations will take place. See Display.h for more details found in the Drivers virtual folder in the project. | Y |
| BOARD_DISPLAY_EXCLUDE_UART | Define this symbol to exclude the UART used in the display driver. | Y |
| BOARD_DISPLAY_EXCLUDE_LCD | Define this symbol to exclude the LCD used in the display driver. | Y |
| MAX_NUM_BLE_CONNS=1 | This is the maximum number of simultaneous Bluetooth low energy collections allowed. Adding more connections uses more RAM and may require increasing HEAPMGR_SIZE. Profile heap usage accordingly. | Y |
| CC26XX | This selects the chipset. | N |
| <Board_Type> | Selects the board type used by board.c. See board.h (\src\components\hal\src\target) for available board types. Examples: CC2650_LAUNCHXL, CC2650DK_7ID. Also see Section 6.2 for more info. | Y |
| xdc_runtime_Assert_DISABLE_ALL | Disables XDC run-time assert | N |
| xdc_runtime_Log_DISABLE_ALL | Disables XDC run-time logging | N |
| HEAPMGR_METRICS | Enables collection of ICall heap metrics. See Section 9.6 for details on how to profile heap usage. | Y |

Table 9-2 lists the only stack preprocessor options that may be modified:

Table 9-2. Stack Preprocessor Symbols

| Preprocessor Symbol | Description | Modify |
|----------------------------|---|--------|
| POWER_SAVING | Power management is enabled when defined, and disabled when not defined. Requires the same option in application project. | Y |
| GATT_NO_CLIENT | When defined, the GATT client is not included to save flash. GATT client is excluded from most peripheral projects, included in central and certain peripheral projects (for example, TimeApp). | Y |
| BLE_NO_SECURITY | Unlink security functions from the dispatcher, used in conjunction with disabling GAP bond manager and SNV to further reduce flash space. | Y |
| OSAL_SNV=1 | Select the number of NV pages to use for SNV. Each page is 4kB of flash. A minimum of one page is required when GAP_BOND_MANAGER is defined. See Section 3.10.3 | Y |
| OSAL_MAX_NUM_PROXY_TASKS=2 | Number of ICall-aware tasks the protocol task can communicate with. Default is 2. Increase this value if more RTOS tasks are added that make ICall protocol stack API calls. | Y |
| EXT_HAL_ASSERT | Extended assert enables support for application callback for asserts. | Y |

9.13 Check System Flash and RAM Usage With Map File

Both application and stack projects produce a map file which can be used to compute the combined flash and RAM system memory usage. Both projects have their own memory space and both map files must be analyzed to determine the total system memory usage. The map file is in the output folder of the respective project in IAR. To compute the total memory usage, do as follows.

1. Open the application map file (that is, `simple_peripheral_cc2650em_app.map`).

NOTE: At the end of the file, three lines contain a breakdown of memory usage for read-only code, read-only data, and read/write data.

2. Add the two values for read-only code and read-only data memory.

NOTE: This sum is the total flash memory usage for the application project. The read/write data memory is the total RAM usage by the application project.

3. Note these values.
4. Open the stack map file.
5. Compute the same flash and RAM values for the stack project.
6. Add the total flash memory value from the application with the total flash usage of the stack to determine the total system flash usage.
7. Add the total RAM usage from the application with the stack to get the total system RAM usage.

For CCS, the map file of the respective project gives a summary of flash and RAM usage. To determine the remaining available memory for each project, see [Section 3.10](#) (flash) and [Section 3.11](#) (RAM). Due to section placement and alignment requirements, some remaining memory may be unavailable. The map file memory usage is valid only if the project builds and links successfully.

Creating a Custom Bluetooth low energy Application

A system designer must have a firm grasp on the general system architecture, application, and Bluetooth low energy stack framework to implement a custom Bluetooth low energy application. This section provides indications and guidance on where and how to start implementing a custom application based on information presented in the previous sections. Decide what role and purpose the custom application should have. If an application is tied to a specific service or profile, start with that sample application. An example is the heart rate sensor project, which implements the heart rate adopted profile. Otherwise, base your project on one of the following sample applications that implement one of the fundamental GAP roles:

- simple_central
- simple_peripheral
- simple_broadcaster
- simple_observer

10.1 Adding a Board File

After selecting the reference application and preprocessor symbol, add a board file that matches the custom board layout. In many cases, just changing the board file is all that is required to move from a development kit to production hardware. The following steps provide guidance on adding a custom board file to the project.

1. Create a custom board file (TI recommends using the Evaluation Module [EM] board file as a starting reference), and modify the PIN structure to match the layout of the board. See [Section 6.2](#).
2. Add peripheral driver initialization objects according to the board design.
3. Remove the existing EM board C.
4. Include files from the folder of the start-up application.
5. Add the custom board file to the application project.
6. Update the C compiler search path of the IDE to point to the header file of the new board file.
7. Define a new board file identifier.
8. Add the RF front-end and bias configuration to the ble_user_config.h file.
9. Refer to the direction in this file for guidance on adding a new custom board RF configuration.
10. Rebuild the application project.

10.2 Configuring Parameters for Custom Hardware

1. Set parameters, such as the sleep clock accuracy of the 32.768-kHz crystal.
2. Define the CCFG parameters in ccfg_app_ble.c to enable or disable the ROM serial bootloader, JTAG access (DAP), flash protection, and so forth.

For a description of CCFG configuration parameters, see *TI CC26xx Technical Reference Manual*, ([SWCU117](#)).

10.3 Creating Additional Tasks

Many designs can benefit from the multi-threaded RTOS environment, by adding additional tasks to handle application-specific functionality. If the system design requires the addition of an additional RTOS task, see [Section 3.3.1](#) for guidance on adding a task.

When adding a new task that makes protocol stack API calls, including calls to OSAL SNV, the task must register with ICall as described in [Section 4.2.4](#). Additionally, project preprocessor settings for OSAL_MAX_NUM_PROXY_TASKS, ICALL_MAX_NUM_TASKS, and CALL_MAX_NUM_ENTITIES may need to be increased based on the number of added tasks (see [Section 9.11](#)).

10.4 Optimizing Bluetooth low energy Stack Memory Usage

Configuration of the Bluetooth low energy protocol stack is essential for maximizing the amount of RAM and flash memory available for the application. Refer to [Section 5.8](#) to configure parameters that impact runtime RAM usage, such as the maximum allowable size and number of PDUs. The TI Bluetooth low energy protocol stack is implemented to use a small RAM footprint, and allow the application to control the behavior of the stack by using the runtime ICall heap. For example, an application that only sends one GATT notification per connection event must store only one PDU in the heap, whereas as an application that must send multiple notifications must enqueue multiple PDUs in the heap.

To increase the available flash memory allocated to the application project, minimize the flash usage of the protocol stack by including only Bluetooth low energy features required to implement the defined role of the device. The available protocol stack configurable features are described in [Section 5.9](#). Adding additional features to the protocol stack has the net effect of reducing the amount of flash memory to the application.

If the peripheral device requires only GATT functionality, the default Bluetooth 4.0 configuration can be used, as the Bluetooth specification requires central devices implementing 4.1 and 4.2 core specification features to be backwards compatible with 4.0 peer devices. TI recommends only adding additional protocol stack features if the system design calls for use of such features.

10.4.1 Additional Memory Configuration Options

The following tips can be used to minimize RAM and flash usage by the protocol stack:

1. Verify that your application uses the optimize for flash size compiler optimization settings (default for TI projects).
2. Use only one page of SNV or do not use any NV pages if the GAP bond manager is not required. Set the NO_OSAL_SNV stack preprocessor option. See [Section 3.10.3](#) for a description of SNV.
3. Exclude the GATT client functionality by defining the GATT_NO_CLIENT predefined symbol in the stack project for peripheral devices. (Peripheral devices do not typically implement the GATT client role.)
4. Remove or exclude debug DISPLAY drivers from the application project (see [Section 9.11.2](#)).
5. Exclude Bluetooth 4.1-specific features from the Bluetooth low energy stack for devices that use only Bluetooth 4.0 functionality.

For example, the Bluetooth 4.1 controller and L2CAP connection-oriented channels features can be excluded by commenting out the following lines from build_config.opt.

```
/* BLE v4.1 Features */
/* -DBLE_V41_FEATURES=L2CAP_COC_CFG+V41_CTRL_CFG */
/* -DBLE_V41_FEATURES=L2CAP_COC_CFG */
/* -DBLE_V41_FEATURES=V41_CTRL_CFG */

/* BLE v4.2 Features */
/* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+PRIVACY_1_2_CFG+EXT_DATA_LEN_CFG */
/* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+PRIVACY_1_2_CFG */
/* -DBLE_V42_FEATURES=PRIVACY_1_2_CFG+EXT_DATA_LEN_CFG */
/* -DBLE_V42_FEATURES=SECURE_CONNS_CFG+EXT_DATA_LEN_CFG */
/* -DBLE_V42_FEATURES=SECURE_CONNS_CFG */
/* -DBLE_V42_FEATURES=PRIVACY_1_2_CFG */
/* -DBLE_V42_FEATURES=EXT_DATA_LEN_CFG */
```

See [Section 9.13](#) for the procedure to check the size of the configured protocol stack.

10.5 Defining Bluetooth low energy Behavior

This step involves using Bluetooth low energy protocol stack APIs to define the system behavior and adding profiles, defining the GATT database, configuring the security model, and so forth. Use the concepts explained in [Chapter 5](#) as well as the Bluetooth low energy API reference in [Appendix A](#).

Porting from CC254x to CC2640

11.1 Introduction

TI-RTOS is the new operating environment for Bluetooth low energy projects on CC26xx devices. This software is a multithreaded environment where the protocol stack, application, and its profiles exist on different threads. TI-RTOS has similar features to OSAL but different mechanisms for accomplishing them. This section covers the main differences between TI-RTOS and OSAL when developing applications on top of the Bluetooth low energy protocol stack. Although the incorporation of the RTOS is a major architecture change, Bluetooth low energy APIs and related procedures are similar to CC254x.

This section covers the following topics:

- OSAL
- Application and stack separation with ICall
- Threads, semaphores, and queues
- Peripheral Drivers
- Event Processing

Most of these differences are unique to TI-RTOS. This section covers these differences and how they relate to OSAL.

11.2 OSAL

A major change in moving to TI-RTOS is the complete removal of the application from the OSAL environment. While the stack code uses OSAL within its own thread, the application thread can only use the APIs of OSAL that are defined in ICallBleAPI.c. Many functions such as `osal_memcpy()`, `osal_memcmp()`, and `osal_mem_alloc()` are unavailable. These functions have been replaced by TI-RTOS, C run time, and ICall APIs.

11.3 Application and Stack Separation With ICall

In the CC2640 Bluetooth low energy protocol stack, the application is a separate image from the stack image unlike the OSAL method, which consists of only a single image. The benefit for this separation is detailed in the ICall (see [Section 4.2](#)). This structure allows independent upgrading of the application and stack images.

The address of the startup entry for the stack image is known by the application image at build time so the application image knows where the stack image starts. Messages between the application and stack pass through a framework developed called ICall short for indirect function calls. This functionality lets the application call the same APIs used in OSAL but is parsed by the ICall and sent to the stack for processing. Many of these stack functions are defined in ICallBleAPI.c for the application to use transparently while ICall handles the sending and receiving from the stack transparently.

11.4 Threads, Semaphores, and Queues

Unlike single-threaded operating systems such as OSAL, TI-RTOS is multithreaded with custom priorities for each thread. The TI-RTOS handles thread synchronization and APIs are provided for the application threads to use to maintain synchronization between different threads. Semaphores are the prime source of synchronization for applications. The semaphores are used to pass event messages to the event processor of the application.

Profile callbacks that run in the context of the Bluetooth low energy protocol stack thread are made re-entrant by storing event data and posting a semaphore of the application to process in the context of the application. Similarly, key press events and clock events that run in ISR context also post semaphores to pass events to the application. Unique to TI-RTOS, queues are how applications process events in the order the events were called and make callback functions from profiles and the stack re-entrant. The queues also provide a FIFO ordering for event processing. An example project may use a queue to manage internal events from an application profile or a GAP profile role (for example, Peripheral or Central). ICall uses a queue and it is accessed through the ICall API. For a description of the TI-RTOS objects used by the Bluetooth low energy stack SDK, see [Chapter 3](#).

11.5 Peripheral Drivers

Aside from switching to an RTOS-based environment, peripheral drivers represent a significant change from the CC254x architecture. Any drivers used by the CC254x software must be ported to the respective TI-RTOS driver interfaces. For details on adding and using a CC26xx peripheral driver, see [Chapter 6](#).

11.6 Event Processing

Similar to OSAL, each RTOS task has two functions that implement the fundamental tasks for an application: `simple_peripheral_init()` and `simple_peripheral_taskFxn()`.

`simple_peripheral_init()` contains ICall registration routines and initialization functions for the application profiles and the GAP and GATT roles. Function calls that are normally in the `START_DEVICE_EVT` event of the CC254x application are also made in the `simple_peripheral_init()` function. The initialization includes setting up callbacks that the application should receive from the profile and stack layers. For more details on callbacks, see [Section 4.3.3](#).

`simple_peripheral_taskFxn()` contains an infinite loop in which events are processed. After entry of the loop and having just finished initialization, the application task calls `ICall_wait()` to block on its semaphore until an event occurs. For more information on how the application processes different events, see [Section 4.3.2.2](#).

Similar to `osal_set_event()` in a CC254x application, the application task can post the semaphore of the application with a call to `Semaphore_post(sem)` after setting an event such as in `simple_peripheral_clockHandler()`. An alternative way is to enqueue a message using `simple_peripheral_enqueueMsg(,)` which preserves the order in which the events are processed. Similar to `osal_start_timerEx()` in a CC254x application, you can use a clock to set an event after a predetermined amount of time using `Util_constructClock()`. This function can also set a periodic event as shown in the `simple_peripheral` project.

Events come from within the same task, the profiles, and the stack. Events from the stack are handled first with a call to `ICall_fetchServiceMsg()` similar to `osal_msg_receive()` in a CC254x application. Internal events and messages from the profiles and the GAP role profiles received in callback functions must be treated as re-entrant are handled in the `simple_peripheral_taksFxn()` function too. In many cases such as in GAP role profile callbacks, you must place events in a queue to preserve the order in which messages arrive. For more information, see [Section 4.3](#) for general overview of application architecture.

Sample Applications

The purpose of this section is to give an overview of the sample applications included in the TI Bluetooth low energy stack software development kit. Some of these implementations are based on specifications that have been adopted by the Bluetooth Special Interest Group (Bluetooth SIG), while others are based on specifications that have not been finalized. Some applications are not based on any standardized profile being developed by the Bluetooth SIG but are custom implementations developed by TI.

All projects contain an IAR and a CCS implementation. Except for the SimpleLink Bluetooth Smart CC2650 SensorTag, TI intends most sample applications described in this section to run on the SmartRF06 Evaluation Board using a CC26xx Evaluation Module. A few select projects have been ported to the CC2650 LaunchPad.

12.1 Blood Pressure Sensor

This sample project implements the Blood Pressure profiles in a Bluetooth low energy peripheral device to provide an example blood pressure monitor (BPM) using simulated measurement data. The application implements the Sensor role of the blood pressure profile. The project is based on the adopted profile and service specifications for blood pressure. The project also includes the Device Information Service. The project is configured to run on the SmartRF06 board.

12.1.1 Interface

This application has two button inputs.

SmartRF Button Right—When disconnected, this button toggles advertising on and off. When connected, this button increases the value of various measurements.

SmartRF Button Up—This button cycles through measurement formats.

12.1.2 Operation

The following steps detail how to use the Blood Pressure Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the blood pressure sensor from a blood pressure collector peer device.

NOTE: The peer device discovers the blood pressure service and configures it to enable indication or notifications of the blood pressure measurement. The peer device may also discover the device information service for more information such as the manufacturing and serial number. When blood pressure measurements have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats in the following order:
 - MMHG | TIMESTAMP | PULSE | USER | STATUS
 - MMHG | TIMESTAMP
 - MMHG
 - KPA
 - KPA | TIMESTAMP
 - KPA |TIMESTAMP | PULSE

If the peer device initiates pairing, the blood pressure sensor requires a passcode. The default passcode is 000000. When the connection terminates, the BPM does not begin advertising until the button is pressed. The peer device may also query the blood pressure for read-only device information. The GATT_DB excel sheet for this project lists further details on the supported items (for example, model number, serial number, and so forth).

12.2 Heart Rate Sensor

This sample project implements the Heart Rate and Battery profiles in a Bluetooth low energy peripheral device to provide an example heart rate sensor using simulated measurement data. The application implements the "Sensor" role of the Heart Rate profile and the Battery Reporter role of the Battery profile. The project is based on adopted profile and service specifications for Health Rate. The project also includes the Device Information Service. The project is configured to run on the SmartRF06 board.

12.2.1 Interface

When the left button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF is connected, it cycles through different heart rate sensor data formats. When in a connection and the battery characteristic is enabled for notification, the battery level is periodically notified.

12.2.2 Operation

The following steps detail how to use the Heart Rate Sensor sample project:

1. Power up the device.
2. Press the left button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the heart rate sensor from a heart rate collector peer device.

NOTE: The peer device discovers the heart rate service and configure it to enable notifications of the heart rate measurement. The peer device may also discover and configure the battery service for battery level-state notifications. When heart rate measurement notifications have been enabled the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
 - Sensor contact not supported
 - Sensor contact not detected
 - Sensor contact and energy expended set
 - Sensor contact and R-R Interval set
 - Sensor contact, energy expended, and R-R Interval set
 - Sensor contact, energy expended, R-R Interval, and UINT16 heart rate set
 - Nothing set

If the peer device initiates pairing, the devices pair. Only just works pairing is supported by the application (pairing without a passcode). The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason, the application starts advertising at the slow interval. The advertising intervals and durations are configurable in file `heartrate.c`.

12.3 Cycling Speed and Cadence (CSC) Sensor

This sample project implements the CSC profile in a Bluetooth low energy peripheral device to provide a sample application of sensor that would be placed on a bicycle, using simulated measurement data. The application implements the Sensor role of the CSC. This profile also uses of the optional Device Info Service, which has default values that may be altered at compile or run time to help identify a specific Bluetooth low energy device. This project is configured to run on the SmartRF06 board.

12.3.1 Interface

When the right button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF, it cycles through different cycling speed and cadence sensor data formats.

Pressing the select key initiates a soft reset. A soft reset includes the following.

- Terminating all current connections
- Clearing all bond data
- Clearing white list of all peer addresses

12.3.2 Operation

The following steps detail how to use the Cycling Speed and Cadence Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the cycling sensor from a CSC collector peer device.

NOTE: The peer device receives a slave security request and initiates a bond. When bonded, the collector discovers the CSC service and configures it to enable CSC measurements. When CSC measurement notifications have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
 - Sensor at rest (no speed or cadence detected)
 - Sensor detecting speed but no cadence
 - Sensor detecting cadence but no speed
 - Sensor detecting speed and cadence

The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds. If the sensor successfully bonds to a peer device and stores the address of the device in its white list, then for the first 10 seconds of advertising the sensor only tries to connect to any device addresses stored in its white list. After 10 seconds, the sensor tries to connect to any peer device that attempts to connect. Independent of the white list, a 30-second period of slow interval advertising passes after 30 seconds of fast interval connection. The device sleeps until you press the right button before advertising again. If the device terminates connection for any other reason, the sensor advertises for 60 seconds at a slow interval and then sleeps if no connection is made. The sensor advertises only if you press the right button.

12.3.3 Neglect Timer

This device has a compile time option that lets the sensor terminate a connection if there is no input for 15 seconds. After the device has connected and notifications are disabled, the application starts a timer. This timer is restarted whenever a read or write request comes from the peer device and is disabled while notifications are enabled. If the value `USING_NEGLECT_TIMEOUT` is set to `FALSE` at compile, this timer is permanently disabled at run time.

12.4 Running Speed and Cadence (RSC) Sensor

This sample project implements the RSC profile in a Bluetooth low energy peripheral device to provide a sample application of sensor on a bicycle using simulated measurement data. The application implements the Sensor role of the RSC Profile. This profile also makes use of the optional Device Info Service in the same manner as the Cycling Sensor. This project is configured to run on the SmartRF06 Board.

12.4.1 Interface

When the right button of the SmartRF is disconnected, it toggles advertising on and off. When the up button of the SmartRF is connected, it cycles through different running speed and cadence sensor data formats.

Pressing the select key initiates a soft reset. This reset includes the following.

- Terminating all current connections
- Clearing all bond data
- Clearing the white list of all peer addresses

12.4.2 Operation

The following steps detail how to use the Running Speed and Cadence Sensor sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the cycling sensor from an RSC collector peer device.

NOTE: The peer device receives a slave security request and initiates a bond. When bonded, the collector discovers the RSC service and configures it to enable running speed and cadence measurements. When RSC measurement notifications have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows.
 - At rest: neither instantaneous stride length nor total distance is included in measurement
 - Stride: instantaneous stride length is included in measurement
 - Distance: total distance is included in measurement
 - All: both stride length and total distance are included in measurement

The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application advertises at the fast interval for 30 seconds. If the sensor successfully bonds to a peer device and stores the address of the device in its white list, the sensor tries only to connect to any device addresses stored in its white list for the first 10 seconds of advertising. After 10 seconds, the sensor tries to connect to any peer device trying to connect. After 30 seconds of fast interval connection, a 30-second period of slow interval advertising passes independent of white list use. The device sleeps and waits for you to press the right button before resuming advertising. If the device terminates connection for any other reason, the sensor advertises for 60 seconds at a slow interval and then sleeps if no connection is made. The device advertises again only if the right button is pressed.

12.4.3 Neglect Timer

This device has a compile time option that lets the sensor terminate a connection if there is no input for 15 seconds. After the device has connected and notifications are disabled, the application starts a timer. This timer is restarted whenever a read or write request comes from the peer device and is disabled while notifications are enabled. If the value `USING_NEGLECT_TIMEOUT` is set to `FALSE` at compile, this timer is permanently disabled at run time.

12.5 Glucose Collector

This sample project implements a glucose collector. The application is designed to connect to the glucose sensor sample application to demonstrate the operation of the Glucose Profile. The project is configured to run on the SmartRF06.

12.5.1 Interface

The SmartRF buttons and display provide an interface for the application. The buttons are as follows.

- Up: If not connected, start or stop device discovery. If connected to a glucose sensor, request the number of records that meet configured filter criteria.
- Left: Scroll through device discovery results. If connected to a glucose sensor, send a record access abort message.
- Select: Connect or disconnect to or from the selected device.
- Right: If connected, request records that meet configured filter criteria.
- Down: If connected, clear records that meet configured filter criteria. If not connected, erase all bonds.

The LCD display displays the following information.

- BD address of the device
- Device discovery results
- Connection state
- Pairing and bonding status
- Number of records requested
- Sequence number, glucose concentration, and Hba1c value of received glucose measurement and context notifications

12.5.2 Record Access Control Point

The Glucose Profile uses a characteristic called the record access control point to perform operations on glucose measurement records stored by the glucose sensor. The following different operations can be performed.

- Retrieve stored records.
- Delete stored records.
- Abort an operation in progress.
- Report number of stored records.

The glucose collector sends control point messages to a sensor by using write requests, while the sensor sends control point messages to the glucose collector by using indications. When records are retrieved, the glucose measurement and glucose context are sent through notifications on their respective characteristics. If an expected response is not received, the operation times out after 30 seconds and the glucose collector closes the connection.

12.6 Glucose Sensor

This sample project implements the Glucose Profile in a Bluetooth low energy peripheral device to provide an example glucose sensor using simulated measurement data. The application implements the Sensor role of the Glucose Profile. The application is compiled to run on a SmartRF06 board.

12.6.1 Interface

When the right button is disconnected, it toggles advertising on and off. When the up button is connected, it sends a glucose measurement and glucose context.

12.6.2 Operation

The following steps detail how to use the Glucose Sensor sample projects:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the glucose sensor from a glucose collector peer device.

NOTE: The peer device discovers the glucose service and configures it to enable notifications of the glucose measurement. The device may also enable notifications of the glucose measurement context. When glucose measurement notifications have been enabled a simulated measurement can be sent by pressing the up button. If the peer device has also enabled notifications of the glucose measurement context then this is sent following the glucose measurement. The peer device may also write commands to the record access control point to retrieve or erase stored glucose measurement records. The sensor has four hardcoded simulated records. If the peer device initiates pairing then the devices pair. Only just works pairing is supported by the application (pairing without a passcode).

12.7 HID-Emulated Keyboard

This sample project implements the HID-Over-GATT profile in a Bluetooth low energy peripheral device to provide an example of how a HID keyboard can be emulated with a simple four button remote control device. The project is based on adopted profile and service specifications for HID-Over-GATT and Scan Parameters. The project also includes the Device Information Service and Battery Service. This project is configured to run on the SmartRF06 board

12.7.1 Interface

When the following are connected, they send the following key presses:

- The left button sends a left arrow key.
- The right button sends a right arrow key.
- The up button sends an up arrow key.
- The down button sends a down arrow key.

A secure connection must be established before key presses is sent to the peer device.

12.7.2 Operation

The following steps detail how to use the HID-Emulated Keyboard sample project:

1. Power up the device.

NOTE: The device advertises by default.

2. Initiate a device discovery and connection procedure to discover and connect to the HID device from a HID Host peer device.

NOTE: The peer device discovers the HID service and recognizes the device as a keyboard. The peer device may also discover and configure the battery service for battery level-state notifications. By default, the HID device requires security and uses just works pairing. After a secure connection is established and the HID host configures the HID service to enable notifications, the HID device can send HID key presses to the HID host. A notification is sent when a button is pressed and when a button is released. If there is no HID activity for a period of time (20 seconds by default) the HID device disconnects. When the connection is closed, the HID device advertises again.

12.8 HostTest–Bluetooth low energy Network Processor

The HostTest project implements a pure Bluetooth low energy network processor to use with an external microcontroller or a PC software application such as BTool. Communication occurs through the HCI interface.

12.9 KeyFob

The KeyFob application demonstrates the following:

- Report battery level
- Report 3-axis accelerometer readings
- Report proximity changes
- Report key press changes

The following GATT services are used:

- Device Information
- Link Loss
- Immediate Alert (for the Reporter role of the Proximity Profile and for the Target role of the Find Me Profile)
- Tx Power (for the Report role of the Proximity Profile)
- Battery
- Accelerometer
- Simple Keys

The accelerometer and simple keys profiles are unaligned with official SIG profiles but are an example of the implementation of the profile service. The device information service and proximity-related services are based on adopted specifications.

12.9.1 Interface

This application uses two buttons for input and an LED and buzzer for output.

Right Button

When disconnected, this button toggles advertising on and off. When connected, this button registers a key press that can be enabled to notify a peer device or may be read by a peer device.

Left Button

When connected, this button registers a key press that can be enabled to notify a peer device or may be read by a peer device.

Buzzer

The buzzer activates if a Link Loss Alert is triggered.

12.9.2 Battery Operation

The Battery Profile allows for the USB dongle to read the percentage of battery remaining on the SmartRF by reading the value of <BATTERY_LEVEL_UUID>.

12.9.3 Accelerometer Operation

The SmartRF does not communicate with an accelerometer, so the accelerometer data is always set to 0. The accelerometer must be enabled <ACCEL_ENABLER_UUID> by writing a value of 01. When the accelerometer is enabled, each axis can be configured to send notifications by writing 01 00 to the characteristic configuration for each axis <GATT_CLIENT_CHAR_CFG_UUID>. The values can be read by reading <ACCEL_X_UUID>, <ACCEL_Y_UUID>, and <ACCEL_Z_UUID>.

12.9.4 Keys

The simple keys service on the SmartRF lets the device send notifications of key presses and releases to a central device. The application registers with HAL to receive a callback in case HAL detects a key change. The peer device can read the value of the keys by reading <SK_KEYPRESSED_UUID>. The peer device can enable key press notifications by writing 01 to <GATT_CLIENT_CHAR_CFG_UUID>. A value of 00 indicates that neither key is pressed. A value of 01 indicates that the left key is pressed. A value of 02 indicates that the right key is pressed. A value of 03 indicates that both keys are pressed.

12.9.5 Proximity

One of the services of the Proximity Profile is the link loss service, which lets the proximity reporter begin an alert if the connection drops. The link loss alert can be set by writing a value to <PROXIMITY_ALERT_LEVEL_UUID>.

The default alert value setting is 00, which indicates no alert. To turn on the alert, write a 1-byte value of 01 (low alert) or 02 (high alert). By default, the link does not time out until 20 seconds have passed without receiving a packet. This Supervision Timeout value can be changed in the Connection Services tab. The time-out value must be set before the connection is established. After completing the write, move the SmartRF device far enough away from the USB Dongle until disconnected. Alternatively, disconnect the USB Dongle from the PC. When the time-out expires, the alarm is triggered. If a low alert is set, the SmartRF makes a low-pitched beep. If a high alert is set, the SmartRF makes a high-pitched beep. In either case, the SmartRF beeps 10 times and then stops. To stop the beeping, either connect with the SmartRF or press the left button.

12.10 SensorTag

The SimpleLink Bluetooth low energy CC2650 SensorTag 2.0 is a Bluetooth low energy peripheral slave device that runs on the CC2650 SensorTag reference hardware platform. The SimpleLink CC2650 is a multistandard wireless MCU that supports Bluetooth low energy and other wireless protocols. Software developed with the Bluetooth low energy stack is binary compatible with the CC2650. The SensorTag 2.0 includes multiple peripheral sensors with a complete software solution for sensor drivers interfaced to a GATT server running on TI Bluetooth low energy protocol stack. The GATT server contains a primary service for each sensor for configuration and data collection. For a description of the available sensors, see <http://www.ti.com/sensortag>.

12.10.1 Operation

On start-up, the SensorTag advertises with a 100-ms interval. The connection is established by a central device and the sensors can then be configured to provide measurement data. The central device could be any Bluetooth low energy-compliant device and the main focus is on Bluetooth low energy-compliant mobile phones, running either Android™ or iOS®. The central device operates as follows:

- Scans and discovers the SensorTag (the scan response contains name SensorTag)
- Establishes connection based on user-defined connection parameters
- Performs service discovery (discovers characteristics by UUID)
- Operates as a GATT client (write to and read from Characteristic Value)

The central device initiates the connection and becomes the master. To obtain the data, first activate the corresponding sensor through a Characteristic Value write to appropriate service.

The most power-efficient way to obtain measurements for a sensor is as follows:

1. Enable notifications.
2. Enable the sensor.
3. Disable the sensor (with notification on) when notifications with data are obtained on the master side.

Alternatively, to not use notifications at all do as follows:

1. Enable the sensor.
2. Read the data and verify.
3. Disable the sensor.

For the alternative, the sensor takes a varying amount of time to measure data. Depending on the connection interval (approximately 10 to 4000 ms) set by the central device, the time to measure data varies. The individual sensors require varying delays to complete measurements. TI recommends a setting of 100 ms. For fast accelerometer and magnetometer data updates, a lower setting is required. You can stop notifications and turn the sensors on or off.

12.10.2 Sensors

The following sensors support SensorTag:

- IR Temperature, both object and ambient temperature
- Accelerometer, 3-axis
- Humidity, both relative humidity and temperature
- Magnetometer, 3-axis
- Barometer, both pressure and temperature
- Gyroscope, 3-axis

12.11 Simple BLE Central

The `simple_central` project implements a simple Bluetooth low energy central device with GATT client functionality. This project uses the SmartRF05 + CC2650EM hardware platform. This project can be run on various platforms, including the CC2650 LaunchPad. This project is configured to run on the SmartRF06 board. By default, the `simple_central` application is configured to filter and connect to peripheral devices with the TI Simple Profile UUID. To modify this behavior, set `DEFAULT_DEV_DISC_BY_SVC_UUID` to `FALSE` in `simple_central`.

12.11.1 Interface

The SmartRF buttons and display provide an interface for the application. The buttons are as follows:

- Up: If disconnected, start or stop device discovery. If connected to a `simple_peripheral`, alternate sample read and write requests.
- Left: Scroll through device discovery results.
- Select: Connect or disconnect to or from the selected device.
- Right: If connected, send a parameter update request.
- Down: If connected, start or cancel RSSI polling.

The LCD display displays the following information:

- BD address of the device
- Device discovery results
- Connection state
- Pairing and bonding status
- Attribute read or write value after parameter update

12.12 Simple BLE Peripheral

The `simple_peripheral` project implements a simple Bluetooth low energy peripheral device with GATT services and demonstrates the TI Simple Profile. This project can be a framework for developing many different peripheral-role applications. The *Software Developer's Guide* explains this project.

12.13 Simple Application Processor

The `simple_ap` project demonstrates the TI Simple Profile running on a CC2640 configured as an application processor (AP) interfacing to a CC2640, through SPI or UART, running the `simple_np` network processor application. The project provides project build configurations for the `simple_ap` running on a SensorTag and SmartRF06 + CC2650EM evaluation module. With SimpleAP project configuration, processing of GATT profile and service data is handled on the SimpleAP, while the GATT database and Bluetooth low energy stack reside on the SimpleNP network processor.

12.14 Simple Network Processor

The SimpleNP project implements the TI Simple Network Processor Bluetooth low energy device configuration. In this configuration, the CC2640 operates as a Bluetooth low energy network processor with the application and profiles executing off-chip on a host MCU. The SimpleNP network processor is for designs that require adding a Bluetooth low energy capability to an existing embedded system with a host MCU or application processor. [The Simple Network Processor API Guide](#) has further details on how to interface to the SimpleNP, including the API interface.

12.15 TimeApp

This sample project implements time and alert-related profiles in a Bluetooth low energy peripheral device to provide an example of how Bluetooth low energy profiles are used in a product like a watch. The project is based on adopted profile specifications for Time, Alert Notification, and Phone Alert Status. All profiles are implemented in the client role. The following Network Availability Profile, Network Monitor role has been implemented, based on Network Availability Draft Specification d05r04 (UCRDD). This project has been configured for the SmartRF06 board.

12.15.1 Interface

The interface for the application consists of the SmartRF06 buttons and display. The buttons are used as follows:

- Up: Starts or stops advertising.
- Left: If connected, sends a command to the Alert Notification control point.
- Center: If connected, disconnects. If held down on power-up, erases all bonds.
- Right: If connected, initiates a Reference Time update.
- Down: If connected, initiates a Ringer Control Point update.

The LCD display shows the following information:

- BD address of the device
- Connection state
- Pairing and bonding status
- Passcode display
- Time and date
- Network availability
- Battery state of peer device
- Alert notification messages
- Unread message alerts
- Ringer status

12.15.2 Operation

The following steps detail how to use the Time App sample project.

1. Power up the application.

NOTE: When the application powers up it displays Time App, the BD address of the device, and a default time and date of 00:00 Jan01 2000.

2. Press Up to start advertising.
3. Connect from a peer device.

NOTE: The connection status displays. When the application tries to discover the following services on the peer device:

- Current Time Service
 - DST Change Service
 - Reference Time Service
 - Alert Notification Service
 - Phone Alert Status Service
 - Network Availability Service
 - Battery Service
-

The discovery procedure caches handles of interest. When bonded to a peer device, the handles are saved to avoid performing the procedure again. If a service is discovered certain service characteristics are read and displayed. The network availability status and battery level is displayed and the current time updates. The application also enables notification or indication for characteristics that support these operations. The enabling of notification or indication allows the peer device to send notifications or indications updating the time, network availability, or battery status. The peer device can also send alert notification messages and unread message alerts. These updates and messages is displayed on the LCD. The peer device may initiate pairing. If a passcode is required, the application generates and displays a random passcode.

Enter this passcode on the peer device to proceed with pairing. The application advertises using either a fast interval or a slow interval. When advertising is initiated by a button press or when a connection is terminated due to link loss, the application starts advertising at the fast interval for 30 seconds followed by the slow interval. When a connection is terminated for any other reason the application starts advertising at the slow interval. The advertising intervals and durations are configurable in the `timeapp.c` file.

12.16 Thermometer

This sample project implements a Health Thermometer and Device Information Profile in a *J* low energy peripheral device to provide an example health thermometer application using simulated measurement data. The application implements the Sensor role of the Health Thermometer Profile. The project is based on the adopted profile and service specifications for Health Thermometer. The project also includes the Device Information Service. This project has been configured to run on the SmartRF06 board.

12.16.1 Interface

This application has two buttons inputs.

Right Button

When the button is disconnected and unconfigured to take measurements, it toggles advertising on and off. When connected or configured to take measurements, pressing this button increases the temperature by 1°C. After a 3°C increase in temperature, the interval is set to 30 seconds. This thermometer application also sends an indication to the peer with this interval change if configured.

Up Button

This button cycles through different measurement formats.

12.16.2 Operation

The following steps detail how to use the Thermometer sample project:

1. Power up the device.
2. Press the right button to enable advertising.
3. Initiate a device discovery and connection procedure to discover and connect to the thermometer sensor from a thermometer collector peer device.

NOTE: The peer device discovers the thermometer service and configures it to enable indication or notifications of the thermometer measurement. The peer device may also discover the device information service for more information such as manufacturing and serial number. When thermometer measurements have been enabled, the application sends data to the peer containing simulated measurement values.

4. Press the up button to cycle through different data formats as follows:
 - CELSIUS | TIMESTAMP | TYPE
 - CELSIUS | TIMESTAMP
 - CELSIUS
 - FAHRENHEIT
 - FAHRENHEIT | TIMESTAMP
 - FAHRENHEIT | TIMESTAMP | TYPE

If the peer device initiates pairing, the HT requests a passcode. The passcode is 000000.

The thermometer operates in the following states:

- Idle – In this state, the thermometer does nothing until you press the button on the right to start advertising.
- Idle Configured – The thermometer waits the interval before taking a measurement and proceeding to Idle Measurement Ready state.
- Idle Measurement Ready – The thermometer has a measurement ready and advertises to allow connection. The thermometer periodically advertises in this state.
- Connected Not Configured – The thermometer may be configured to enable measurement reports. The thermometer does not send stored measurements until the CCC is enabled. When the thermometer is connected, it sets a timer to disconnect in 20 seconds.
- Connected Configured – The thermometer does send any stored measurements if CCC is set to send measurement indications.
- Connected Bonded – The thermometer sends any stored measurements if CCC was previously set to send measurement indications.

The peer device may also query the thermometers read only device information. Examples are model number, serial number, and so forth.

GAP API

A.1 Commands

This section details the GAP commands from gap.h that the application uses. All other GAP commands are abstracted through the GAPRole or the GAPBondMgr. The return values described in this section are the return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

uint16 GAP_GetParamValue (gapParamIDs_t paramID)

Get a GAP parameter.

| | |
|------------|--|
| Parameters | parameter ID (Section A.2) |
| Returns | GAP Parameter Value if successful 0xFFFF if paramID invalid |

bStatus_t GAP_SetParamValue (gapParamIDs_t paramID, uint16 paramValue)

Set a GAP parameter.

| | |
|------------|--|
| Parameters | paramID – parameter ID (Section A.2) paramValue – new param value |
| Returns | SUCCESS (0x00) INVALIDPARAMETER (0x02): paramID is invalid |

bStatus_t GAP_ConfigDeviceAddr(uint8 addrType, uint8 *pStaticAddr);

Used to setup the device's address type and address where applicable. This must be called after the GAP layer is started, and can not be called during any BLE activity.

Parameters

addrMode: address mode

- ADDRMODE_PUBLIC: use the BD_ADDR. See the HCI_EXT_SetBDADDRCmd() API for more information.
- ADDRMODE_STATIC: static address
- ADDRMODE_PRIVATE_NONRESOLVE: non-resolvable private address
- ADDRMODE_PRIVATE_RESOLVE: resolvable private address that changes based on the TGAP_PRIVATE_ADDR_INT GAP parameter.

pStaticAddr: a pointer to a 6-byte address, only used with ADDRMODE_STATIC or ADDRMODE_PRIVATE_NONRESOLVE

Returns

SUCCESS: address type updated

bleNotReady (0x10): GAP layer has not been started

bleIncorrectMode (0x12): there is currently BLE activity, can not change address

INVALIDPARAMETER (0x02): invalid address mode passed into function

void GAP_RegisterForMsgs(uint8 taskID);

Register a task ID to receive extra HCI status, command complete, and host events that do not need to be processed by the stack. This is most commonly used to receive HCI events in the application, or to receive events that the central GAPRole does not process. See the HCI section or GAPRole section for more information.

Parameters

taskID – task ID to send events to

bStatus_t GAP_ResolvePrivateAddr(uint8 *pIRK, uint8 *pAddr)
Resolves a private address against an IRK.

Parameters

pIRK: pointer to the IRK

pAddr: pointer to the resolvable private address

Returns

SUCCESS (0x00): a match was found

FAILURE (0x01): no match was found

INVALIDPARAMETER (0x02): one of the pointers was NULL

uint8 GAP_NumActiveConnections(void)
Returns the number of active connections
Returns Number of active connections

A.2 Configurable Parameters

| ParamID | Description | Default | Range |
|------------------------------|--|---------|----------|
| TGAP_GEN_DISC_ADV_MIN | Time (ms) to remain advertising in general discovery mode. Setting this to 0 turns off this timeout, advertising infinitely. | 0 | 0-65535 |
| TGAP_LIM_ADV_TIMEOUT | Time (sec) to remain advertising in limited discovery mode. | 180 | 1-65535 |
| TGAP_GEN_DISC_SCAN | Time (ms) to perform scanning for general discovery. | 10240 | 1-65535 |
| TGAP_LIM_DISC_SCAN | Time (ms) to perform scanning for limited discovery. | 10240 | 1-65535 |
| TGAP_CONN_EST_ADV_TIMEOUT | Advertising timeout (ms) when performing connection establishment. | 10240 | 1-65535 |
| TGAP_CONN_PARAM_TIMEOUT | Timeout (ms) for link layer to wait to receive connection parameter update response before giving up. | 30000 | 1-65535 |
| TGAP_LIM_DISC_ADV_INT_MIN | Minimum advertising interval in limited discovery mode (n × 0.625 ms) | 160 | 32-16384 |
| TGAP_LIM_DISC_ADV_INT_MAX | Maximum advertising interval in limited discovery mode (n × 0.625 ms) | 160 | 32-16384 |
| TGAP_GEN_DISC_ADV_INT_MIN | Minimum advertising interval in general discovery mode (n × 0.625 ms) | 160 | 32-16384 |
| TGAP_GEN_DISC_ADV_INT_MAX | Maximum advertising interval in general discovery mode (n × 0.625 ms) | 160 | 32-16384 |
| TGAP_CONN_ADV_INT_MIN | Minimum advertising interval when in connectable mode (n × 0.625 ms) | 2048 | 32-16384 |
| TGAP_CONN_ADV_INT_MAX | Maximum advertising interval when in connectable mode (n × 0.625 ms) | 2048 | 32-16384 |
| TGAP_CONN_SCAN_INT | Scan interval used during Link Layer Initiating state, when in connectable mode (n × 0.625 ms) | 480 | 4-16384 |
| TGAP_CONN_SCAN_WIND | Scan window used during Link Layer Initiating state, when in connectable mode (n × 0.625 ms) | 240 | 4-16384 |
| TGAP_CONN_HIGH_SCAN_INT | Scan interval used during Link Layer Initiating state, when in connectable mode, high duty scan cycle scan parameters (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_CONN_HIGH_SCAN_WIND | Scan window used during Link Layer Initiating state, when in connectable mode, high duty scan cycle scan parameters (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_GEN_DISC_SCAN_INT | Scan interval used during Link Layer Scanning state, when in general discovery proc (n × 0.625 ms). | 16 | 4-16384 |
| TGAP_GEN_DISC_SCAN_WIND | Scan window used during Link Layer Scanning state, when in general discovery proc (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_LIM_DISC_SCAN_INT | Scan interval used during Link Layer Scanning state, when in limited discovery proc (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_LIM_DISC_SCAN_WIND | Scan window used during Link Layer Scanning state, when in limited Discovery proc (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_CONN_EST_INT_MIN | Minimum Link Layer connection interval, when using connection establishment proc (n × 1.25 ms) | 80 | 6-3200 |
| TGAP_CONN_EST_INT_MAX | Maximum Link Layer connection interval, when using connection establishment proc (n × 1.25 ms) | 80 | 6-3200 |
| TGAP_CONN_EST_SCAN_INT | Scan interval used during Link Layer Initiating state, when using connection establishment proc (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_CONN_EST_SCAN_WIND | Scan window used during Link Layer Initiating state, when using connection establishment proc (n × 0.625 ms) | 16 | 4-16384 |
| TGAP_CONN_EST_SUPERV_TIMEOUT | Supervision timeout, when using connection establishment proc (n × 10 ms) | 2000 | 10-3200 |
| TGAP_CONN_EST_LATENCY | Slave latency, when using connection establishment proc (in number of connection events) | 0 | 0-499 |
| TGAP_CONN_EST_MIN_CE_LEN | Local informational parameter about minimum length of connection required, when using connection establishment proc (n × 0.625 ms) | 0 | < max |
| TGAP_CONN_EST_MAX_CE_LEN | Local informational parameter about maximum length of connection required, when using connection establishment proc (n × 0.625 ms). | 0 | > min |

Configurable Parameters
www.ti.com

| ParamID | Description | Default | Range |
|----------------------------|---|---------|------------|
| TGAP_PRIVATE_ADDR_INT | Minimum Time Interval between private (resolvable) address changes. In minutes (default 15 min) | 15 | 1-65535 |
| TGAP_CONN_PAUSE_CENTRAL | Central idle timer. In seconds (default 1 s) | 1 | 1-65535 |
| TGAP_CONN_PAUSE_PERIPHERAL | Minimum time upon connection establishment before the peripheral starts a connection update procedure. In seconds (default 5 seconds) | 5 | 1-65535 |
| TGAP_SM_TIMEOUT | Time (ms) to wait for security manager response before returning bleTimeout. Default is 30 s. | 30000 | 1-65535 |
| TGAP_SM_MIN_KEY_LEN | SM Minimum Key Length supported. Default 7. | 7 | 1-65535 |
| TGAP_SM_MAX_KEY_LEN | SM Maximum Key Length supported. Default 16. | 16 | 1-65535 |
| TGAP_FILTER_ADV_REPORTS | TRUE to filter duplicate advertising reports. Default TRUE. | TRUE | 0-1 |
| TGAP_SCAN_RSP_RSSI_MIN | Minimum RSSI required for scan responses to be reported to the app. Default -127. | -127 | 965535 - 0 |
| TGAP_REJECT_CONN_PARAMS | Whether or not to reject Connection Parameter Update Request received on Central device. Default FALSE. | FALSE | 0-1 |

A.3 Events

This section details the events relating to the GAP layer that can be returned to the application from the Bluetooth low energy stack. Some of these events are passed directly to the application and some are handled by the GAPRole or GAPBondMgr layers. The events are passed as a GAP_MSG_EVENT with header:

```
typedef struct
{
    osal_event_hdr_t hdr;        //!< GAP_MSG_EVENT and status
    uint8 opcode;               //!< GAP type of command. Ref: @ref GAP_MSG_EVENT_DEFINES
} gapEventHdr_t;
```

The following is a list of the possible hdr and the associated events. See gap.h for all other definitions used in these events.

- **GAP_DEVICE_INIT_DONE_EVENT:** Sent when the Device Initialization is complete

```
typedef struct
{
    osal_event_hdr_t hdr;        //!< GAP_MSG_EVENT and status
    uint8 opcode;               //!< GAP_DEVICE_INIT_DONE_EVENT
    uint8 devAddr[B_ADDR_LEN];  //!< Device's BD ADDR
    uint16 dataPktLen;          //!< HC_LE_Data_Packet_Length
    uint8 numDataPkts;          //!< HC_Total_Num_LE_Data_Packets
} gapDeviceInitDoneEvent_t;
```

- **GAP_DEVICE_DISCOVERY_EVENT:** Sent when the Device Discovery Process is complete

```
typedef struct
{
    osal_event_hdr_t hdr;        //!< GAP_MSG_EVENT and status
    uint8 opcode;               //!< GAP_DEVICE_DISCOVERY_EVENT
    uint8 numDevs;              //!< Number of devices found during scan
    gapDevRec_t *pDevList;      //!< array of device records
} gapDevDiscEvent_t;
```

- **GAP_ADV_DATA_UPDATE_DONE_EVENT:** Sent when the Advertising Data or SCAN_RSP Data has been updated

```
typedef struct
{
    osal_event_hdr_t hdr;        //!< GAP_MSG_EVENT and status
    uint8 opcode;               //!< GAP_ADV_DATA_UPDATE_DONE_EVENT
    uint8 adType;               //!< TRUE if advertising data, FALSE if SCAN_RSP
} gapAdvDataUpdateEvent_t;
```

- **GAP_MAKE_DISCOVERABLE_DONE_EVENT:** Sent when the Make Discoverable Request is complete

```
typedef struct
{
    osal_event_hdr_t hdr;        //!< GAP_MSG_EVENT and status
    uint8 opcode;               //!< GAP_MAKE_DISCOVERABLE_DONE_EVENT
} gapMakeDiscoverableRspEvent_t;
```

- **GAP_END_DISCOVERABLE_DONE_EVENT**: Sent when the Advertising has ended

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                   //!< GAP_END_DISCOVERABLE_DONE_EVENT
} gapEndDiscoverableRspEvent_t;
```

- **GAP_LINK_ESTABLISHED_EVENT**: Sent when the Establish Link Request is complete

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                   //!< GAP_LINK_ESTABLISHED_EVENT
    uint8 devAddrType;              //!< Device address type: @ref ADDRTYPE_DEFINES
    uint8 devAddr[B_ADDR_LEN];     //!< Device address of link
    uint16 connectionHandle;       //!< Connection Handle from controller used to ref the device
    uint8 connRole;                 //!< Connection formed as Master or Slave
    uint16 connInterval;           //!< Connection Interval
    uint16 connLatency;            //!< Connection Latency
    uint16 connTimeout;            //!< Connection Timeout
    uint8 clockAccuracy;           //!< Clock Accuracy
} gapEstLinkReqEvent_t;
```

- **GAP_LINK_TERMINATED_EVENT**: Sent when a connection was terminated

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                   //!< GAP_LINK_TERMINATED_EVENT
    uint16 connectionHandle;       //!< connection Handle
    uint8 reason;                   //!< termination reason from LL
} gapTerminateLinkEvent_t;
```

Where reason can be (not all reasons are included here):

- **LL_STATUS_ERROR_UNKNOWN_CONN_HANDLE (0x02)**: master cancelled connection establishment
 - **LL_STATUS_ERROR_PIN_OR_KEY_MISSING (0x06)**: pin or key missing for encryption
 - **LL_STATUS_ERROR_OUT_OF_CONN_RESOURCES (0x07)**: memory capacity exceeded
 - **LL_STATUS_ERROR_CONNECTION_TIMEOUT (0x08)**: supervision timeout occurred
 - **LL_STATUS_ERROR_ILLEGAL_PARAM_COMBINATION (0x12)**: illegal combination of connection interval, slave latency, supervision timeout in connection establishment request
 - **LL_STATUS_ERROR_HOST_TERM (0x16)**: command terminated by local host
 - **LL_STATUS_ERROR_UNSUPPORTED_REMOTE_FEATURE (0x1A)**: Reject Indication Extended is not supported and reject indication can not be used
 - **LL_STATUS_ERROR_LL_TIMEOUT (0x22)**: peer or host procedure timeout
 - **LL_STATUS_ERROR_INSTANT_PASSED_TERM (0x28)**: instant passed when performing connection parameter update or channel map update procedure
 - **LL_STATUS_ERROR_UNACCEPTABLE_CONN_PARAMETERS (0x3B)**: connection formed with bad parameters
 - **LL_STATUS_ERROR_DIRECTED_ADV_TIMEOUT (0x3C)**: directed advertising finished without establishing a connection
 - **LL_STATUS_ERROR_CONN_TERM_DUE_TO_MIC_FAILURE (0x3D)**: MIC failure occurred
 - **LL_STATUS_ERROR_CONN_FAILED_TO_BE_ESTABLISHED (0x3E)**: error in establishing the connection
- **GAP_LINK_PARAM_UPDATE_EVENT**: Sent when an Update Parameters Event is received

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                   //!< GAP_LINK_PARAM_UPDATE_EVENT
    uint8 status;                   //!< bStatus_t
    uint16 connectionHandle;       //!< Connection handle of the update
}
```

```

uint16 connInterval;          //!< Requested connection interval
uint16 connLatency;          //!< Requested connection latency
uint16 connTimeout;          //!< Requested connection timeout
} gapLinkUpdateEvent_t;
    
```

Where status can be:

- LL_STATUS_ERROR_INACTIVE_CONNECTION (0x02): the connection is not active
 - LL_STATUS_ERROR_COMMAND_DISALLOWED (0x0C): not configured correctly to send this command
 - LL_STATUS_ERROR_ILLEGAL_PARAM_COMBINATION (0x12): connection interval, slave latency, timeout combination is invalid
 - LL_STATUS_ERROR_UNSUPPORTED_REMOTE_FEATURE (0x1A): the peer device does not support this request
 - LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A): there is already a parameter update in process
 - LL_STATUS_ERROR_UNACCEPTABLE_CONN_INTERVAL (0x3B): connection interval, slave latency, timeout combination is invalid
- GAP_RANDOM_ADDR_CHANGED_EVENT: Sent when a random address is changed

```

typedef struct
{
    osal_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_RANDOM_ADDR_CHANGED_EVENT
    uint8 addrType;                //!< Address type: #ref GAP_ADDR_TYPE_DEFINES
    uint8 newRandomAddr[B_ADDR_LEN]; //!< the new calculated private addr
} gapRandomAddrEvent_t;
    
```

- GAP_SIGNATURE_UPDATED_EVENT: Sent when the device's signature counter is updated

```

typedef struct
{
    osal_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_SIGNATURE_UPDATED_EVENT
    uint8 addrType;                //!< Device's address type for devAddr
    uint8 devAddr[B_ADDR_LEN];     //!< Device's BD_ADDR, could be own address
    uint32 signCounter;            //!< new Signed Counter
} gapSignUpdateEvent_t;
    
```

- GAP_AUTHENTICATION_COMPLETE_EVENT: Sent when the Authentication (pairing) process is complete

```

typedef struct
{
    osal_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_AUTHENTICATION_COMPLETE_EVENT
    uint16 connectionHandle;       //!< Connection Handle from controller used to ref the device
    uint8 authState;               //!< TRUE if the pairing was authenticated (MITM)
    smSecurityInfo_t *pSecurityInfo; //!< BOUND - security information from this device
    smSigningInfo_t *pSigningInfo;  //!< Signing information
    smSecurityInfo_t *pDevSecInfo;  //!< BOUND - security information from connected device
    smIdentityInfo_t *pIdentityInfo; //!< BOUND - identity information
} gapAuthCompleteEvent_t;
    
```

- GAP_PASSKEY_NEEDED_EVENT: Sent when a Passkey is required (part of the pairing process)

```

typedef struct
{
    osal_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_PASSKEY_NEEDED_EVENT
    uint8 deviceAddr[b_ADDR_LEN];  //!< address of device to pair with, and could be either
public or random
    uint16 connectionHandle;       //!< Connection handle
    uint8 uiInputs;                 //!< Pairing User Interface Inputs -
    Ask user to Input passcode
    uint8 uiOutputs;                //!< Pairing User Interface Outputs - Display passcode
    uint32 numComparison;           //!< Numeric Comparison value to be displayed
} gapPasskeyNeededEvent_t;
    
```

- **GAP_SLAVE_REQUESTED_SECURITY_EVENT**: Sent when a Slave Security Request is received

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_SLAVE_REQUESTED_SECURITY_EVENT
    uint16 connectionHandle;       //!< Connection handle
    uint8 deviceAddr[B_ADDR_LEN];  //!< address of device requesting security
    uint8 authReq;                 //!< Authentication Requirements: Bit 2: MITM, Bits 0-
1: bonding (0 - no bonding, 1 - bonding)
} gapSlaveSecurityReqEvent_t;
```

- **GAP_DEVICE_INFO_EVENT**: Sent during the Device Discovery Process when a device is discovered

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_DEVICE_INFO_EVENT
    uint8 eventType;              //!< Advertisement Type:@ ref
GAP_ADVERTISEMENT_REPORT_TYPE_DEFINES
    uint8 addrType;               //!< address type: @ref GAP_ADDR_TYPE_DEFINES
    uint8 addr[B_ADDR_LEN];       //!< Address of the advertisement or SCAN_RSP
    int8 rssi;                    //!< Advertisement or SCAN_RSP_RSSI
    uint8 dataLen;                //!< Length (in bytes) of the data field (evtData)
    uint8 *pEvtData;             //!< Data field of advertisement or SCAN_RSP
} gapDeviceInfoEvent_t;
```

- **GAP_BOND_COMPLETE_EVENT**: Sent when the bonding (bound) process is complete

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_BOND_COMPLETE_EVENT
    uint16 connectionHandle;       //!< connection Handle
} gapBondCompleteEvent_t;
```

- **GAP_PAIRING_REQ_EVENT**: Sent when an unexpected Pairing Request is received

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP_PAIRING_REQ_EVENT
    uint16 connectionHandle;       //!< connection Handle
    gapPairingReq_t pairReq;       //!< The Pairing Request fields received.
} gapPairingReqEvent_t;
```

- **GAP_AUTHENTICATION_FAILURE_EVT**: Sent when pairing fails due to a connection dropping when other pairings are queued.

```
typedef struct
{
    osal_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
    uint8 opcode;                  //!< GAP type of command. Ref: @ref GAP_MSG_EVENT_DEFINES
} gapEventHdr_t;
```

GAPRole Peripheral Role API

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

B.1 Commands

bStatus_t GAPRole_SetParameter(uint16_t param, uint8_t len, void *pValue)

Set a GAP Role parameter.

Parameters: param – Profile parameter ID (see [Section B.2](#))
 len – length of data to write
 pValue – pointer to value to set parameter. This pointer is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)
 INVALIDPARAMETER (0x02): param was invalid
 bleInvalidRange (0x18): len is invalid for the given param
 blePending (0x16): previous param update has not been completed
 bleIncorrectMode (0x12): can not start connectable advertising because nonconnectable advertising is enabled

bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID ([Section B.2](#))
 pValue – pointer to location to get parameter. This pointer is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)
 INVALIDPARAMETER (0x02): param was in valid

bStatus_t GAPRole_StartDevice(gapRolesCBs_t *pAppCallbacks)

Initializes the device as a peripheral and configures the application callback function.

Parameters pAppCallbacks – pointer to application callbacks ([Section B.3](#))

Returns SUCCESS (0x00)
 bleAlreadyInRequestedMode (0x11): device was already initialized

bStatus_t GAPRole_TerminateConnection(void)

Terminates an existing connection.

Returns

SUCCESS (0x00): connection termination process has started

bleIncorrectMode (0x12): there is no active connection

bleInvalidTaskID (0x03): application did not register correctly with ICall

LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A): disconnect is already in process

bStatus_t GAPRole_SendUpdateParam(uint16_t minConnInterval, uint16_t maxConnInterval, uint16_t latency, uint16_t connTimeout, uint8_t handleFailure)

Update the parameters of an existing connection. See [Section 5.1](#) for more details.

Parameters

connInterval – the new connection interval

latency – the new slave latency

connTimeout – the new timeout value

handle failure– what to do if the update does not occur. Available actions:

- GAPROLE_NO_ACTION 0 // Take no action upon unsuccessful parameter updates
- GAPROLE_RESEND_PARAM_UPDATE 1 // Continue to resend request until successful update
- GAPROLE_TERMINATE_LINK 2 // Terminate link upon unsuccessful parameter updates

Returns

SUCCESS (0x00): parameter update process has started

bleNotConnected (0x14): there is no connection so can not update parameters

void GAPRole_RegisterAppCBs(gapRolesParamUpdateCB_t *pParamUpdateCB) Register application param update callback with peripheral GAPRole

Parameters pParamUpdateCB: pointer to param update callback. See [Section B.3](#) for more information.

B.2 Configurable Parameters

| ParamID | R/W | Size | Description |
|-------------------------|-----|------------|--|
| GAPROLE_PROFILEROLE | R | uint8 | GAP profile role (peripheral) Possible values: GAP_PROFILE_PERIPHERAL: when using the peripheral GAPRole, this is always the case. |
| GAPROLE_IRK | R/W | uint8[16] | Identity resolving key returned from GAP_DEVICE_INIT_DONE_EVENT or read from SNV Possible values: 0x00000000000000000000000000000000 – 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Default value: 0x00000000000000000000000000000000 |
| GAPROLE_SRK | R/W | uint8[16] | Signature resolving key returned from GAP_DEVICE_INIT_DONE_EVENT or read from SNV Possible values: 0x00000000000000000000000000000000 – 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Default value: 0x00000000000000000000000000000000 |
| GAPROLE_SIGNCOUNTER | R/W | uint32 | Sign counter returned from GAP_EVENT_SIGN_COUNTER_CHANGED Possible values: 0x0000 – 0xFFFF Default value: 0x0000 |
| GAPROLE_BD_ADDR | R | uint8[6] | Device address read from controller. This can be set with the HCI_EXT_SetBDADDRCmd(). Possible values: 0x000000000000 – 0xFFFFFFFFFE Default value - BDADDR read from the info page, secondary address from flash, or set through software in descending orders of priority |
| GAPROLE_ADVERT_ENABLED | R/W | uint8 | Enable or disable advertising. Possible values: <ul style="list-style-type: none"> 0x00: connectable advertising is disabled 0x01: connectable advertising is enabled. GAPROLE_ADV_NONCONN_ENABLED must be set to 0x00 Default value: 0x01 |
| GAPROLE_ADVERT_OFF_TIME | R/W | uint16 | How long to remain off (in seconds) after advertising stops before starting again. If set to 0, advertising will not start again. Possible values: 0 - 65535 Default value: 30 |
| GAPROLE_ADVERT_DATA | R/W | <uint8[32] | Advertisement data. This third byte sets limited / general advertising as defined in Vol 3, Part C, section 11.1.3 of the BT 4.2 Core Spec. Possible values: A 1 – 31 byte array, formatted as defined in Vol 3, Part C, section 11.1.3 of the BT 4.2 Core Spec. Default value: 02:01:01 (general advertising) |
| GAPROLE_SCAN_RSP_DATA | R/W | <uint8[32] | Scan Response data. This should be formatted as defined in Vol 3, Part C, section 11.1.3 of the BT 4.2 Core Spec. Possible values: A 1-31 byte array. Default value: All zeroes |

| ParamID | R/W | Size | Description |
|-----------------------------|-----|----------|--|
| GAPROLE_ADV_EVENT_TYPE | R/W | uint8 | Advertisement type. Possible values: <ul style="list-style-type: none"> GAP_ADTYPE_ADV_IND (0x01) : Connectable undirected advertisement GAP_ADTYPE_ADV_HDC_DIRECT_IND (0x02) : Connectable high duty cycle directed advertisement GAP_ADTYPE_ADV_SCAN_IND (0x03) : Scannable undirected advertisement GAP_ADTYPE_ADV_NONCONN_IND (0x04) : Non-Connectable undirected advertisement GAP_ADTYPE_ADV_LDC_DIRECT_IND (0x05) : Connectable low duty cycle directed advertisement Default value: GAP_ADTYPE_ADV_IND |
| GAPROLE_ADV_DIRECT_TYPE | R/W | uint8 | Direct advertisement type. |
| GAPROLE_ADV_DIRECT_ADDRESS | R/W | uint8[6] | Direct advertisement address. Possible values: <ul style="list-style-type: none"> ADDRMODE_PUBLIC (0x00) : Use the BD_ADDR ADDRMODE_STATIC (0x01) : Use provided static address ADDRMODE_PRIVATE_NONRESOLVE (0x02) : Generate and use non-resolvable private address ADDRMODE_PRIVATE_RESOLVE (0x03) : Generate and use resolvable private address Default value: ADDRMODE_PUBLIC |
| GAPROLE_ADV_CHANNEL_MAP | R/W | uint8 | Which channels to advertise on. Multiple channels can be selected by ORing the bit values below. Possible values: <ul style="list-style-type: none"> GAP_ADVCHAN_37 (0x01): Channel 37 GAP_ADVCHAN_38 (0x02): Channel 38 GAP_ADVCHAN_39 (0x04): Channel 39 GAP_ADVCHAN_ALL (0x07): Channel 40 Default value: GAP_ADVCHAN_ALL |
| GAPROLE_ADV_FILTER_POLICY | R/W | uint8 | Policy for filtering advertisements. Ignored in direct advertising. Possible values: <ul style="list-style-type: none"> GAP_FILTER_POLICY_ALL (0x00): Allow scan request from any, allow connect request from any GAP_FILTER_POLICY_WHITE_SCAN (0x01): Allow scan request from white list only, allow connect from any GAP_FILTER_POLICY_WHITE_CON (0x02): Allow scan request from any, connect from white list only GAP_FILTER_POLICY_WHITE (0x03): Allow scan request and connect from white list only Default value: GAP_FILTER_POLICY_ALL |
| GAPROLE_CONNHANDLE | R | uint16 | Handle of current connection. Possible values: 0x0000 – 0xFFFFD Default value: 0x000 |
| GAPROLE_PARAM_UPDATE_ENABLE | R/W | uint8 | Whether to request a connection parameter update upon connection. Possible values: <ul style="list-style-type: none"> 0x00: Do not request parameter update 0x01: Request parameter update Default value: 0x01 |
| GAPROLE_MIN_CONN_INTERVAL | R/W | uint16 | Minimum connection interval to use when performing param update through GAPROLE_PARAM_UPDATE_ENABLE or GAPROLE_PARAM_UPDATE_REQ (n x 1.25 ms) Possible values: 6 – (GAPROLE_MAX_CONN_INTERVAL) Default value: 6 |

| ParamID | R/W | Size | Description |
|-----------------------------|-----|----------|---|
| GAPROLE_MAX_CONN_INTERVAL | R/W | uint16 | Maximum connection interval to use when performing param update through GAPROLE_PARAM_UPDATE_ENABLE or GAPROLE_PARAM_UPDATE_REQ (n x 1.25 ms) Possible values: GAPROLE_MIN_CONN_INTERVAL - 3200 Default value: 3200 |
| GAPROLE_SLAVE_LATENCY | R/W | uint16 | Slave latency to use when performing param update through GAPROLE_PARAM_UPDATE_ENABLE or GAPROLE_PARAM_UPDATE_REQ Possible values: 0-499 Default value: 0 |
| GAPROLE_TIMEOUT_MULTIPPLIER | R/W | uint16 | Supervision timeout to use when performing param update through GAPROLE_PARAM_UPDATE_ENABLE or GAPROLE_PARAM_UPDATE_REQ (n x 10 ms) Possible values: 10-3200 Default value: 1000 |
| GAPROLE_CONN_BD_ADDR | R | uint8[6] | Address of connected device. Possible values: 0x000000000000 – 0xFFFFFFFFFFD Default value: 0x000000000000 |
| GAPROLE_CONN_INTERVAL | R | uint16 | Current connection interval (n x 1.25 ms) Possible values: 6 - 3200 Default value: 0 |
| GAPROLE_CONN_LATENCY | R | uint16 | Current slave latency Possible values: 0 - 499 Default value: 0 |
| GAPROLE_CONN_TIMEOUT | R | uint16 | Current supervision timeout (n x 10 ms) Possible values: 10 – 3200 Default value: 0 |
| GAPROLE_PARAM_UPDATE_REQ | W | uint8 | Used to send an asynchronous parameter update request. Possible values: <ul style="list-style-type: none"> 0x00: Doesn't do anything 0x01: Sends parameter update request |
| GAPROLE_STATE | R | uint8 | Current peripheral GAPRole state. Possible values: See Section B.3.1 Default value: GAPROLE_INIT |
| GAPROLE_ADV_NONCONN_ENABLED | R/W | uint8 | Enable or disable non-connectable advertising. Possible values: <ul style="list-style-type: none"> 0x00: Disable non-connectable advertising 0x01: Enable non-connectable advertising. GAPROLE_ADVERT_ENABLE must be set to 0x00. Default value: 0x00 |
| GAPROLE_BD_ADDR_TYPE | R | uint8 | Address type of connected device Possible values: <ul style="list-style-type: none"> ADDRTYPE_PUBLIC (0x00): Public Device Address ADDRTYPE_RANDOM (0x01): Random Device Address ADDRTYPE_PUBLIC_ID (0x02): Public Identity Address (corresponds to peer's RPA) ADDRTYPE_RANDOM_ID (0x03): Random (static) Identity Address (corresponds to peer's RPA) Default value: ADDRTYPE_PUBLIC |

| ParamID | R/W | Size | Description |
|--------------------------|-----|-------|--|
| GAPROLE_CONN_TERM_REASON | R | uint8 | Reason of the last connection terminated event Possible values: <ul style="list-style-type: none"> LL_STATUS_ERROR_UNKNOWN_CONN_HANDLE (0x02): master cancelled connection establishment LL_STATUS_ERROR_PIN_OR_KEY_MISSING (0x06): pin or key missing for encryption LL_STATUS_ERROR_OUT_OF_CONN_RESOURCES (0x07): memory capacity exceeded LL_STATUS_ERROR_CONNECTION_TIMEOUT (0x08): supervision timeout occurred LL_STATUS_ERROR_HOST_TERM (0x16): command terminated by local host LL_STATUS_ERROR_UNSUPPORTED_REMOTE_FEATURE (0x1A): Reject Indication Extended is not supported and reject indication can't be used LL_STATUS_ERROR_LL_TIMEOUT (0x22): peer or host procedure timeout LL_STATUS_ERROR_INSTANT_PASSED_TERM (0x28): instant passed when performing connection parameter update or channel map update procedure LL_STATUS_ERROR_UNACCEPTABLE_CONN_PARAMETERS (0x3B): connection formed with bad parameters LL_STATUS_ERROR_DIRECTED_ADV_TIMEOUT (0x3C): directed advertising finished without establishing a connection LL_STATUS_ERROR_CONN_TERM_DUE_TO_MIC_FAILURE (0x3D): MIC failure occurred LL_STATUS_ERROR_CONN_FAILED_TO_BE_ESTABLISHED (0x3E): error in establishing the connection Default value: 0 |

B.3 Callbacks

Callback are functions whose pointers are passed from the application to the GAPRole so that the GAPRole can return events to the application. They are passed as the following:

```
/**
 * Callback structure - must be setup by the application and used when
 *                       GAPRole_StartDevice() is called.
 */
typedef struct
{
    gapRoleStateNotify_t pfnStateChange;    ///< Whenever the device changes state
} gapRolesCBs_t;

/**
 * Callback when the device has been started.  Callback event to
 * the Notify of a state change.
 */
typedef void (*gapRolesStateNotify_t)(gaprole_States_t newState);
```

See the simple_peripheral application for an example.

B.3.1 State Change Callback (pfnStateChange)

This callback passes the current GAPRole state to the application whenever the state changes. This function is of the following type:

```
typedef void (*gapRolesStateNotify_t)(gaprole_States_t newState);
```

The various GAPRole states (newState) are as follows:

- GAPROLE_INIT ///< Waiting to be started

- GAPROLE_STARTED //!< Started but not advertising
- GAPROLE_ADVERTISING //!< Currently advertising
- GAPROLE_ADVERTISING_NONCONN //!< Currently using non-connectable advertising
- GAPROLE_WAITING //!< Device is started and in a waiting period before advertising again
- GAPROLE_WAITING_AFTER_TIMEOUT //!< Device timed out from a connection but is not yet advertising, the device is in waiting period before advertising again.
- GAPROLE_CONNECTED //!< In a connection
- GAPROLE_CONNECTED_ADV //!< In a connection + advertising
- GAPROLE_ERROR //!< Error occurred – invalid state



GAPRole Central Role API

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command never gets processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

C.1 Commands

bStatus_t GAPCentralRole_StartDevice(gapCentralRoleCB_t *pAppCallbacks)

Start the device in Central role. This function is typically called once during system startup.

Parameters pAppCallbacks – pointer to application callbacks

Returns SUCCESS (0x00)
bleAlreadyInRequestedMode (0x11): Device already started

bStatus_t GAPCentralRole_SetParameter(uint16_t param, uint8_t len, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID ([Section C.2](#))
len – length of data to write
pValue – pointer to value to set parameter. This is dependent on the parameter ID and is cast to the appropriate data type

Returns SUCCESS (0x00)
INVALIDPARAMETER (0x02): param was not valid
bleInvalidRange (0x18): len is invalid for the given param

bStatus_t GAPCentralRole_GetParameter (uint16_t param, void *pValue)

Set a GAP Role parameter.

Parameters param – Profile parameter ID ([Section C.2](#))
pValue – pointer to buffer to contain the read data

Returns

SUCCESS (0x00)

INVALIDPARAMETER (0x02): param was not valid

bStatus_t GAPCentralRole_TerminateLink (uint16_t connHandle);
Terminates an existing connection.

Parameters connHandle – connection handle of link to terminate or...
 0xFFFFE – cancel the current link establishment request or...
 0xFFFFF – terminate all links

Returns SUCCESS (0x00) – termination has started
 bleIncorrectMode (0x12) – there is no active connection
 bleInvalidTaskID (0x03) – application did not register correctly with ICall
 LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A) – terminate procedure
 already started

**bStatus_t GAPCentralRole_EstablishLink(uint8_t highDutyCycle, uint8_t whiteList, uint8_t
 addrTypePeer, uint8_t *peerAddr)**
Establish a link to a peer device.

Parameters highDutyCycle – TRUE to high duty cycle scan, FALSE if not
 whiteList – determines use of the white list
 addrTypePeer – address type of the peer device
 peerAddr – peer device address

Returns SUCCESS (0x00): link establishment has started
 bleIncorrectMode (0x12): invalid profile role
 bleNotReady (0x10): a scan is in progress
 bleAlreadyInRequestedMode (0x11): unable to process at this time
 bleNoResources (0x15): too many links

bStatus_t GAPCentralRole_UpdateLink(uint16_t connHandle, uint16_t connIntervalMin, uint16_t connIntervalMax, uint16_t connLatency, uint16_t connTimeout)
Update the link connection parameters.

Parameters

connHandle – connection handle
 connIntervalMin – minimum connection interval in 1.25 ms
 connIntervalMax – maximum connection interval in 1.25 ms
 connLatency – number of LL latency connection events
 connTimeout – connection timeout in 10 ms

Returns

SUCCESS (0x00): parameter update has started
 bleNotConnected (0x14): no connection to update
 INVALIDPARAMETER (0x02): connection parameters are invalid
 LL_STATUS_ERROR_ILLEGAL_PARAM_COMBINATION (0x12): connection parameters do not meet *Bluetooth* low energy specification requirements: $LSTO > (1 + \text{Slave Latency}) \times (\text{Connection Interval} \times 2)$
 LL_STATUS_ERROR_INACTIVE_CONNECTION (0x02): connHandle is inactive
 LL_STATUS_ERROR_CTRL_PROC_ALREADY_ACTIVE (0x3A): there is already a param update in process
 LL_STATUS_ERROR_UNACCEPTABLE_CONN_INTERVAL (0x3B): connection interval does not work because it is not a multiple or divisor of intervals of the other simultaneous connection or the interval of the connection is not less than the allowed maximum connection interval as determined by the maximum number of connections times the number of slots per connection

bStatus_t GAPCentralRole_StartDiscovery(uint8_t mode, uint8_t activeScan, uint8_t whiteList)
Start a device discovery scan.

Parameters

mode – discovery mode
 activeScan – TRUE to perform active scan
 whiteList – TRUE to only scan for devices in the white list

Returns

SUCCESS (0x00): device discovery has started
 bleAlreadyInRequestedMode (0x11): Device discovery already started
 bleMemAllocError (0x13): not enough memory to allocate device discovery structure
 LL_STATUS_ERROR_BAD_PARAMETER (0x12): bad parameter

bStatus_t GAPCentralRole_CancelDiscovery(void)
Cancel a device discovery scan.

Parameters

None

Returns

SUCCESS (0x00): cancelling of device discovery has started
 bleInvalidTaskID (0x03): Application has not registered correctly with ICall or this is not the same task that started the discovery.
 bleIncorrectMode (0x12): Not in discovery mode

C.2 Configurable Parameters

| ParamID | R/W | Size | Description |
|---------------------------------|-----|-----------|---|
| GAPCENTRALROLE_IRK | R/W | uint8[16] | Identity resolving key. Default is all 0, which means the IRK is randomly generated. |
| GAPCENTRALROLE_SRK | R/W | uint8[16] | Signature resolving key. Default is all 0, which means the SRK is randomly generated. |
| GAPCENTRALROLE_SIGNCOUNTER | R/W | uint32 | Sign counter. |
| GAPCENTRALROLE_BD_ADDR | R | uint8[6] | Device address read from controller. This can be set with the <code>HCI_EXT_SetBDADDRCmd()</code> . |
| GAPCENTRALROLE_MAX_SCAN_RESULTS | R/W | uint8 | Maximum number of discover scan results to receive. Default is 8, 0 is unlimited. |

C.3 Callbacks

Callbacks are functions whose pointers are passed from the application to the GAPRole so that the GAPRole can return events to the application. They are passed as follows.

```
typedef struct
{
    pfnGapCentralRoleEventCB_t eventCB;    //!< Event callback.
} gapCentralRolecb_t;
```

See the SimpleBLECentral application for an example.

C.3.1 Central Event Callback (eventCB)

This callback passes GAP state change events from the GAPRole to the application. This callback is of the following type.

```
typedef uint8_t (*pfnGapCentralRoleEventCB_t)
(
    gapCentralRoleEvent_t *pEvent    //!< Pointer to event structure.
);

static uint8_t SimpleBLECentral_eventCB(gapCentralRoleEvent_t *pEvent)
{
    // Forward the role event to the application
    if (SimpleBLECentral_enqueueMsg(SBC_STATE_CHANGE_EVT, SUCCESS, (uint8_t *)pEvent)
        {
            // App will process and free the event
            return FALSE;
        }

    // Caller should free the event
    return TRUE;
}
```

If the message is successfully queued to the application for later processing, FALSE is returned because the application deallocates it later. Consider the state change event as an example of this:

```
static void SimpleBLECentral_processAppMsg(sbcEvt_t *pMsg)
{
    switch (pMsg->hdr.event)
    {
        case SBC_STATE_CHANGE_EVT:
            SimpleBLECentral_processStackMsg((ICall_Hdr *)pMsg->pData);

            // Free the stack message
            ICall_freeMsg(pMsg(pMsg->pData);
            break;
    }
    ...
}
```

If the message is not successfully queued to the application, TRUE is returned so that the GAPRole can deallocate the message. If the heap has enough room, the message must always be successfully enqueued. The possible GAP events that can be forward through this callback to the application are defined in [Section A.3](#).

GATT and ATT API

This section describes the API of the GATT and ATT layers. The two sections are combined because the general procedure is to send GATT commands and receive ATT events as described in [Section 5.3.3.1](#). The return values for the commands referenced in this section are described in [Section D.4](#).

The possible return values are similar for all of these commands so they are described in [Section D.4](#). The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command never gets processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

D.1 General Commands

void GATT_RegisterForMsgs(uint8 taskID);

Register a task ID to receive GATT local events and ATT response messages pending for transmission. When the GATT Server fails to respond to an incoming ATT Request due to lack of HCI Tx buffers, the response is forwarded the app for retransmission.

Parameters taskID – task ID to send events to

D.2 Server Commands

bStatus_t GATT_Indication(uint16 connHandle, attHandleValueInd_t *pInd, uint8 authenticated, uint8 taskId);

Indicates a characteristic value to a client and expect an acknowledgment.

Parameters

connHandle: connection to use

pInd: pointer to indication to be sent

authenticated: whether an authenticated link is required

- 0x01: LE Legacy Authenticated
- 0x02: Secure Connections Authenticated

taskId: task to be notified of acknowledgment

NOTE: The payload must be dynamically allocated as described in [Section 5.3.5](#).

Corresponding Events If the return status is SUCCESS, the calling application task receives a GATT_MSG_EVENT message with type ATT_HANDLE_VALUE_CFM upon an acknowledgment. Only at this point, this subprocedure is complete.

bStatus_t GATT_Notification(uint16 connHandle, attHandleValueNoti_t *pNoti, uint8 authenticated)
Indicates a characteristic value to a client and expect an acknowledgment.

Parameters

connHandle: connection to use
 pNoti: pointer to notification to be sent
 authenticated: whether an authenticated link is required

- 0x01: LE Legacy Authenticated
- 0x02: Secure Connections Authenticated

NOTE: The payload must be dynamically allocated as described in [Section 5.3.5](#).

D.3 Client Commands

bStatus_t GATT_InitClient(void)
Initialize the GATT client in the Bluetooth low energy stack.

NOTE: GATT clients must call this from the application init function.

bStatus_t GATT_RegisterForInd (uint8 taskId)
Register to receive incoming ATT Indications or Notifications of attribute values.

Parameters

taskId: task to forward indications or notifications to

NOTE: GATT clients must call this from the application initialization function.

bStatus_t GATT_ExchangeMTU(uint16 connHandle, attExchangeMTUReq_t *pReq, uint8 taskId);
Used by a client to set the ATT_MTU to the maximum possible that can be supported by both devices when the client supports a value greater than the default ATT_MTU.

Parameters

taskId: task to forward indications or notifications to

NOTE: This function can only be called once during a connection. For more information on the MTU, see [Section 5.5.2](#).

Corresponding Events If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message. The type of the message is either ATT_EXCHANGE_MTU_RSP (with SUCCESS or bleTimeout status) indicating a SUCCESS or ATT_ERROR_RSP (with status SUCCESS) if an error occurred on the server.

bStatus_t GATT_DiscAllPrimaryServices(uint16 connHandle, uint8 taskId)

Used by a client to discover all primary services on a server. The ATT Read By Group Type Request is used with the Attribute Type parameter set to the UUID for "Primary Service". The Starting Handle is set to 0x0001, and the Ending Handle is set to 0xFFFF.

Parameters

connHandle: connection to use
taskId: task to be notified of response

Corresponding Events: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_GRP_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_GRP_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscPrimaryServiceByUUID(uint16 connHandle, uint8 *pValue, uint8 len, uint8 taskId)

Used by a client to discover a specific primary service on a server when only the service UUID is known. The ATT Find By Type Value Request is used with the Attribute Type parameter set to the UUID for "Primary Service" and the Attribute Value set to the 16-bit Bluetooth UUID or 128-bit UUID for the specific primary service. The Starting Handle shall be set to 0x0001 and the Ending Handle shall be set to 0xFFFF.

Parameters

connHandle: connection to use
pValue: pointer to value (UUID) for which to look
len: length of value
taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_FIND_BY_TYPE_VALUE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_FIND_BY_TYPE_VALUE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_FindIncludedServices(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find included services with a primary service definition on a server. The ATT Read By Type Request is used with the AttributeType parameter set to the UUID for "Included Service". The Starting Handle is set to starting handle of the specified service, and the Ending Handle is set to the ending handle of the specified service.

Parameters

connHandle: connection to use
startHandle: start handle of primary service in which to search
endHandle: end handle of primary service in which to search
taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout

status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscAllChars(uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find all the characteristic declarations within a service when the handle range of the service is known.

Parameters

connHandle: connection to use

startHandle: start handle of service in which to search

endHandle: end handle of service in which to search

taskId: task to be notified of response

Corresponding Events: If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscCharsByUUID(uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

Used by a client to discover service characteristics on a server when the service handle range and characteristic UUID is known.

Parameters

connHandle: connection to use

pReq: pointer to request to be sent, including start and end handles of service and UUID of characteristic value for which to search

taskId: task to be notified of response

Corresponding Events If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_DiscAllCharDescs (uint16 connHandle, uint16 startHandle, uint16 endHandle, uint8 taskId)

Used by a client to find all the attribute handles and attribute types of the characteristic descriptor within a characteristic definition when only the characteristic handle range is known.

Parameters

connHandle: connection to use
startHandle: start handle
endHandle: end handle
taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_FIND_INFO_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_FIND_INFO_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadCharValue (uint16 connHandle, attReadReq_t *pReq, uint8 taskId)

Used to read a characteristic value from a server when the client knows the characteristic value Handle. The Read Response only contains a Characteristic Value that is less than or equal to (ATT_MTU – 1) octets in length. If the Characteristic Value is greater than (ATT_MTU – 1) octets in length, the Read Long Characteristic Value procedure may be used if the rest of the Characteristic Value is required.

Parameters

connHandle: connection to use
pReq: pointer to request to be sent
taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadUsingCharUUID (uint16 connHandle, attReadByTypeReq_t *pReq, uint8 taskId)

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 ATT Read By Type Request is used to perform the sub-procedure. The Attribute Type is set to the known characteristic UUID and the Starting Handle and Ending Handle parameters shall be set to the range over which this read is to be performed. This is typically the handle range for the service in which the characteristic belongs.

Parameters

connHandle: connection to use
pReq: pointer to request to be sent
taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_BY_TYPE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BY_TYPE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadLongCharValue (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

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 ATT Read Blob Request is used in this sub-procedure.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BLOB_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadMultiCharValues (uint16 connHandle, attReadMultiReq_t *pReq, uint8 taskId)

Used to read multiple characteristic values from a server when the client knows the characteristic value handles. The Attribute Protocol Read Multiple Requests is used with the Set Of Handles parameter set to the Characteristic Value Handles. The Read Multiple Response returns the Characteristic Values in the Set Of Values parameter. The ATT Read Multiple Request is used in this sub-procedure.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

Corresponding Events

If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_MULTI_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_MULTI_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_WriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

This 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 acknowledgment that the write was successfully performed. This sub-procedure only writes the first (ATT_MTU – 3) octets of a Characteristic Value. This sub-procedure can not be used to write a long characteristic; instead, the Write Long Characteristic Values sub-procedure should be used. The ATT Write Command is used for this sub-procedure. The Attribute Handle parameter shall be set to the Characteristic Value Handle. The Attribute Value parameter shall be set to the new Characteristic Value.

Parameters

connHandle: connection to use
 pReq: pointer to command to be sent

Corresponding Events

No response are sent to the calling application task for this subprocedure. If the Characteristic Value write request is the wrong size, or has an invalid value as defined by the profile, then the write does not succeed and no error is generated by the server. The payload must be dynamically allocated as described in [Section 5.3.5](#).

bStatus_t GATT_SignedWriteNoRsp (uint16 connHandle, attWriteReq_t *pReq)

Used to write a characteristic value to a server when the client knows the characteristic value handle and the ATT bearer is not encrypted. This subprocedure must only be used if the characteristic properties authenticated bit is enabled and the client and server device share a bond as defined in the GAP. This sub-procedure only writes the first (ATT_MTU – 15) octets of an Attribute Value. This sub-procedure cannot be used to write a long Attribute. The ATT Write Command is used for this sub-procedure. The Attribute Handle parameter shall be set to the Characteristic Value Handle. The Attribute Value parameter shall be set to the new Characteristic Value authenticated by signing the value, as defined in the Security Manager.

Parameters

connHandle: connection to use
pReq: pointer to command to be sent

Corresponding Events

No response is sent to the calling application task for this subprocedure. If the authenticated Characteristic Value that is written is the wrong size, or has an invalid value as defined by the profile, or the signed value does not authenticate the client, then the write does not succeed and no error is generated by the server. The payload must be dynamically allocated as described in [Section 5.3.5](#).

bStatus_t GATT_WriteCharValue (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

Used to write a characteristic value to a server when the client knows the characteristic value handle. This sub-procedure only writes the first (ATT_MTU-3) octets of a characteristic value. This sub-procedure can not be used to write a long attribute; instead the Write Long Characteristic Values sub-procedure should be used. The ATT Write Request is used in this sub-procedure. The Attribute Handle parameter shall be set to the Characteristic Value Handle. The Attribute Value parameter shall be set to the new characteristic.

Parameters

connHandle: connection to use
pReq: pointer to request to be sent
taskId: task to be notified of response

Corresponding Events

If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in [Section 5.3.5](#).

bStatus_t GATT_WriteLongCharValue(uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

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 ATT Prepare Write Request and Execute Write Request are used to perform this sub-procedure.

Parameters

connHandle: connection to use
pReq: pointer to request to be sent
taskId: task to be notified of response

Corresponding Events

If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the

server). This subprocedure completes when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in [Section 5.3.5](#).

bStatus_t GATT_ReliableWrites (uint16 connHandle, attPrepareWriteReq_t *pReq, uint8 numReqs, uint8 flags, uint8 taskId)

Used to write a characteristic value to a 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. The sub-procedure has two phases: the first phase prepares the characteristic values to be written. Once this is complete, the second phase performs the execution of all of the prepared characteristic value writes on the server from this client. In the first phase, the ATT Prepare Write Request is used. In the second phase, the attribute protocol Execute Write Request is used.

Parameters

connHandle: connection to use
 pReq: pointer to requests to be sent (must be allocated)
 numReqs – number of requests in pReq
 flags – execute write request flags
 taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in [Section 5.3.5](#).

bStatus_t GATT_ReadCharDesc (uint16 connHandle, attReadReq_t *pReq, uint8 taskId)

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration. The ATT Read Request is used for this sub-procedure. The Read Request is used with the Attribute Handle parameter set to the characteristic descriptor handle. The Read Response returns the characteristic descriptor value in the Attribute Value parameter.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

Corresponding Events

If the return status from this function is SUCCESS, the calling application task receives an OSAL GATT_MSG_EVENT message with type ATT_READ_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure completes when either ATT_READ_RSP (with SUCCESS or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_ReadLongCharDesc (uint16 connHandle, attReadBlobReq_t *pReq, uint8 taskId)

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 descriptor declaration is longer than can be sent in a single read response attribute protocol message. The ATT Read Blob Request is used to perform this sub-procedure. The Attribute Handle parameter shall be set to the characteristic descriptor handle. The Value Offset parameter shall be the offset within the characteristic descriptor to be read.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_READ_BLOB_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_READ_BLOB_RSP (with bleProcedureComplete or bleTimeout status) or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task.

bStatus_t GATT_WriteCharDesc (uint16 connHandle, attWriteReq_t *pReq, uint8 taskId)

Used to read a characteristic descriptor from a server when the client knows the attribute handle of the characteristic descriptor declaration. The ATT Write Request is used for this sub-procedure. The Attribute Handle parameter shall be set to the characteristic descriptor handle. The Attribute Value parameter shall be set to the new characteristic descriptor value.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

bStatus_t GATT_WriteLongCharDesc (uint16 connHandle, gattPrepareWriteReq_t *pReq, uint8 taskId)

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 ATT Prepare Write Request and Execute Write Request are used to perform this sub-procedure.

Parameters

connHandle: connection to use
 pReq: pointer to request to be sent
 taskId: task to be notified of response

Corresponding Events

If the return status is SUCCESS, the calling application task receives multiple GATT_MSG_EVENT messages with type ATT_PREPARE_WRITE_RSP, ATT_EXECUTE_WRITE_RSP or ATT_ERROR_RSP (if an error occurred on the server). This subprocedure is complete when either ATT_PREPARE_WRITE_RSP (with bleTimeout status), ATT_EXECUTE_WRITE_RSP (with SUCCESS or bleTimeout status), or ATT_ERROR_RSP (with SUCCESS status) is received by the calling application task. The payload must be dynamically allocated as described in [Section 5.3.5](#).

D.4 Return Values

- SUCCESS (0x00): Command was executed as expected (See the individual command API for corresponding events to expect.)
- INVALIDPARAMETER (0x02): Invalid connection handle or request field
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): attribute requires authentication
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): key size used for encrypting is insufficient
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): attribute requires encryption
- MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available (Retry later.)
- bleNotConnected (0x14): the device is not connected
- blePending (0x17):
 - When returned to a client function: a response is pending with the server or the GATT subprocedure is in progress
 - When returned to server function: confirmation from a client is pending
- bleTimeout (0x16): the previous transaction timed out (No more ATT and GATT messages can be sent until the connection is reestablished.)
- bleMemAllocError (0x13): memory allocation error occurred
- bleLinkEncrypted (0x19): link is already encrypted (An attribute PDU that includes an authentication signature that must not be sent on an encrypted link.)

D.5 Events

Events are received from the Bluetooth low energy stack in the application as a GATT_MSG_EVENT stack message sent through ICall. Events are received as the following structure where the method signifies the ATT event and the message is a combination of all the various ATT events.

```
typedef struct
{
   osal_event_hdr_t hdr;    //!< GATT_MSG_EVENT and status
    uint16 connHandle;     //!< Connection message was received on
    uint8 method;         //!< Type of message
    gattMsg_t msg;        //!< Attribute protocol/profile message
} gattMsgEvent_t;
```

This section lists the various ATT events by their method and displays their structure that is used in the message payload. These events are listed in the att.h file.

- ATT_ERROR_RSP (0x01)

```
typedef struct
{
    uint8 reqOpcode;    //!< Request that generated this error response
    uint16 handle;     //!< Attribute handle that generated error response
    uint8 errCode;     //!< Reason why the request has generated error response
} attErrorRsp_t;
attErrorRsp_t
```

- ATT_FIND_INFO_RSP (0x03)

```
typedef struct
{
    uint16 numInfo;    //!< Number of attribute handle-UUID pairs found
    uint8 format;     //!< Format of information data
    uint8 *pInfo;     //!< Information data whose format is determined by format field (4 to
ATT_MTU_SIZE-2)
} attFindInfoRsp_t;
```

- ATT_FIND_BY_TYPE_VALUE_RSP (0x07)

```
typedef struct
{
    uint16 numInfo;    //!< Number of handles information found
    uint8 *pHandlesInfo; //!< List of 1 or more handles information (4 to ATT_MTU_SIZE-1)
} attFindByTypeValueRsp_t;
```

- ATT_READ_BY_TYPE_RSP (0x09)

```
typedef struct
{
    uint16 numPairs;  //!< Number of attribute handle-UUID pairs found
    uint16 len;      //!< Size of each attribute handle-value pair
    uint8 *pDataList; //!< List of 1 or more attribute handle-value pairs (2 to ATT_MTU_SIZE-2)
    uint16 dataLen;  //!< Length of data written into pDataList. Not part of actual ATT Response
} attReadByTypeRsp_t;
```

- ATT_READ_RSP (0x0B)

```
typedef struct
{
    uint16 len;    //!< Length of value
    uint8 *pValue; //!< Value of the attribute with the handle given (0 to ATT_MTU_SIZE-1)
} attReadRsp_t;
```

- ATT_READ_BLOB_RSP (0x0D)

```
typedef struct
{
    uint16 len;    //!< Length of value
    uint8 *pValue; //!< Part of the value of the attribute with the handle given (0 to
                    ATT_MTU_SIZE-1)
} attReadBlobRsp_t;
```

- ATT_READ_MULTI_RSP (0x0F)

```
typedef struct
{
    uint16 len;    //!< Length of values
    uint8 *pValue; //!< Set of two or more values (0 to ATT_MTU_SIZE-1)
} attReadMultiRsp_t;
```

- ATT_READ_BY_GRP_TYPE_RSP (0x11)

```
typedef struct
{
    uint16 numGrps;  //!< Number of attribute handle, end group handle and value sets found
    uint16 len;      //!< Length of each attribute handle, end group handle and value set
    uint8 *pDataList; //!< List of 1 or more attribute handle, end group handle and value (4 to
ATT_MTU_SIZE-2)
} attReadByGrpTypeRsp_t;
```

- ATT_WRITE_RSP (0x13)

No data members

- ATT_PREPARE_WRITE_RSP (0x17)

```
typedef struct
{
    uint16 handle; //!< Handle of the attribute that has been read
    uint16 offset; //!< Offset of the first octet to be written
    uint16 len;    //!< Length of value
    uint8 *pValue; //!< Part of the value of the attribute to be written (0 to ATT_MTU_SIZE-5)
} attPrepareWriteRsp_t;
```

- ATT_EXECUTE_WRITE_RSP (0x19)

No data members

- ATT_HANDLE_VALUE_NOTI (0x1B)

```
typedef struct
{
    uint16 handle; //!< Handle of the attribute that has been changed (must be first field)
    uint16 len;    //!< Length of value
    uint8 *pValue; //!< New value of the attribute (0 to ATT_MTU_SIZE-3)
} attHandleValueNoti_t;
```

- ATT_HANDLE_VALUE_IND (0x1D)

```
typedef struct
{
    uint16 handle; //!< Handle of the attribute that has been changed (must be first field)
    uint16 len;    //!< Length of value
    uint8 *pValue; //!< New value of the attribute (0 to ATT_MTU_SIZE-3)
} attHandleValueInd_t;
```

- ATT_HANDLE_VALUE_CFM (0x1E)

– o Empty msg field

- ATT_FLOW_CTRL_VIOLATED_EVENT (0x7E)

```
typedef struct
{
    uint8 opcode;          //!< opcode of message that caused flow control violation
    uint8 pendingOpcode;  //!< opcode of pending message
} attFlowCtrlViolatedEvt_t;
```

- ATT_MTU_UPDATED_EVENT (0x7F)

```
typedef struct
{
    uint16 MTU; //!< new MTU size
} attMtuUpdatedEvt_t;
```

D.6 GATT Commands and Corresponding ATT Events

The following table lists the possible commands that may have caused an event.

| ATT Response Events | GATT API Calls |
|----------------------------|---|
| ATT_EXCHANGE_MTU_RSP | GATT_ExchangeMTU |
| ATT_FIND_INFO_RSP | GATT_DiscAllCharDescs, GATT_DiscAllCharDescs |
| ATT_FIND_BY_TYPE_VALUE_RSP | GATT_DiscPrimaryServiceByUUID |
| ATT_READ_BY_TYPE_RSP | GATT_PrepareWriteReq, GATT_ExecuteWriteReq, GATT_FindIncludedServices, GATT_DiscAllChars, GATT_DiscCharsByUUID, GATT_ReadUsingCharUUID, |
| ATT_READ_RSP | GATT_ReadCharValue, GATT_ReadCharDesc |
| ATT_READ_BLOB_RSP | GATT_ReadLongCharValue, GATT_ReadLongCharDesc |
| ATT_READ_MULTI_RSP | GATT_ReadMultiCharValues |
| ATT_READ_BY_GRP_TYPE_RSP | GATT_DiscAllPrimaryServices |
| ATT_WRITE_RSP | GATT_WriteCharValue, GATT_WriteCharDesc |
| ATT_PREPARE_WRITE_RSP | GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc |
| ATT_EXECUTE_WRITE_RSP | GATT_WriteLongCharValue, GATT_ReliableWrites, GATT_WriteLongCharDesc |

D.7 ATT_ERROR_RSP errCodes

This section lists the possible error codes in the ATT_ERROR_RSP event and their possible causes.

- ATT_ERR_INVALID_HANDLE (0x01): The attribute handle value given is not valid on this attribute server.
- ATT_ERR_READ_NOT_PERMITTED (0x02): The attribute is unable to be read.
- ATT_ERR_WRITE_NOT_PERMITTED (0x03): The attribute is unable to be written.
- ATT_ERR_INVALID_PDU (0x04): The PDU attribute is invalid.
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): The attribute requires authentication before it can be read or written.
- ATT_ERR_UNSUPPORTED_REQ (0x06): The attribute server does not support the request received from the attribute client.
- ATT_ERR_INVALID_OFFSET (0x07): The offset specified is past the end of the attribute.
- ATT_ERR_INSUFFICIENT_AUTHOR (0x08): The attribute requires an authorization before it can be read or written.
- ATT_ERR_PREPARE_QUEUE_FULL (0x09): Too many prepare writes have been queued.
- ATT_ERR_ATTR_NOT_FOUND (0x0A): No attribute exists within the attribute handle range.
- ATT_ERR_ATTR_NOT_LONG (0x0B): The attribute is unable to be read or written using the read blob request or prepare write request.
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The encryption key size for encrypting this link is insufficient.
- ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is invalid for the operation.
- ATT_ERR_UNLIKELY (0x0E): The attribute request requested has encountered an error that is unlikely and could not be completed as requested.
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): The attribute requires encryption before it can be read or written.
- ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10): The attribute type is an unsupported grouping attribute as defined by a higher layer specification.
- ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient resources exist to complete the request.



GATTServApp API

This section details the API of the GATTServApp defined in gattservapp_util.c. These API are only the public commands that must be called by the profile and/or application.

The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never get processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) are returned.

E.1 Commands

void GATTServApp_InitCharCfg(uint16 connHandle, gattCharCfg_t *charCfgTbl)

Initialize the client characteristic configuration table for a given connection. This API must be used when a service is added to the application ([Section 5.3.4.2.2](#)).

Parameters

connHandle – connection handle (0xFFFF for all connections)

charCfgTbl – client characteristic configuration table where this characteristic resides

bStatus_t GATTServApp_ProcessCharCfg(gattCharCfg_t *charCfgTbl, uint8 *pValue, uint8 authenticated, gattAttribute_t *attrTbl, uint16 numAttrs, uint8 taskId, pfnGATTReadAttrCB_t pfnReadAttrCB)

Process client characteristic configuration change

Parameters

charCfgTbl – profile characteristic configuration table

pValue – pointer to attribute value

authenticated – whether an authenticated link is required

attrTbl – attribute table

numAttrs – number of attributes in attribute table

taskId – task to be notified of confirmation

pfnReadAttrCB – read callback function pointer

Returns

SUCCESS (0x00): parameter was set

INVALIDPARAMETER (0x02): one of the parameters was a null pointer

ATT_ERR_INSUFFICIENT_AUTHOR (0x08): permissions require authorization

bleTimeout (0x17): ATT timeout occurred

blePending (0x16): another ATT request is pending

LINKDB_ERR_INSUFFICIENT_AUTHEN (0x05): authentication is required but link is not authenticated

bleMemAllocError (0x13): memory allocation failure occurred when allocating buffer

gattAttribute_t *GATTServApp_FindAttr(gattAttribute_t *pAttrTbl, uint16 numAttrs, uint8 *pValue)

Find the attribute record within a service attribute table for a given attribute value pointer

Parameters

pAttrTbl – pointer to attribute table
 numAttrs – number of attributes in attribute table
 pValue – pointer to attribute value

Returns

Pointer to attribute record if found
 NULL, if not found

bStatus_t GATTServApp_ProcessCCCWriteReq(uint16 connHandle, gattAttribute_t *pAttr, uint8 *pValue, uint16 len, uint16 offset, uint16 validCfg)

Process the client characteristic configuration write request for a given client

Parameters

connHandle – connection message was received on
 pAttr – pointer to attribute value
 pValue – pointer to data to be written
 len – length of data
 offset – offset of the first octet to be written
 validCfg – valid configuration

Returns

SUCCESS (0x00): CCC was written correctly
 ATT_ERR_INVALID_VALUE (0x80): an invalid value for a CCC
 ATT_ERR_INVALID_VALUE_SIZE (0x0D): an invalid size for a CCC
 ATT_ERR_ATTR_NOT_LONG (0x0B): offset needs to be 0
 ATT_ERR_INSUFFICIENT_RESOURCES (0x11): CCC not found

GAPBondMgr API

This section details the API of the GAPBondMgr defined in gapbondmgr.c. Many of these commands are called by the GAPRole or the Bluetooth low energy stack and do not need to be called from the application. The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

F.1 Commands

bStatus_t GAPBondMgr_SetParameter(uint16_t param, void *pValue)

Set a GAP bond manager parameter

Parameters param – profile parameter ID (see [Section D.2](#))
 len – length of data to write
 pValue – pointer to value to set parameter (This pointer depends on the parameter ID and will be cast to the appropriate data type.)

Returns SUCCESS (0x00): parameter was set
 INVALIDPARAMETER (0x02): param was invalid
 bleInvalidRange (0x18): len is invalid for the given param

bStatus_t GAPBondMgr_GetParameter(uint16_t param, void *pValue)

Get a GAP bond manager parameter

Parameters param – profile parameter ID (see [Section D.2](#))
 pValue – pointer to a location to get the value (This pointer is dependent on the param ID and will be cast to the appropriate data type.)

Returns SUCCESS (0x00): param was successfully placed in pValue
 INVALIDPARAMETER (0x02): param was not valid

bStatus_t GAPBondMgr_LinkEst(uint8 addrType, uint8 *pDevAddr, uint16 connHandle, uint8 role)

Notify the bond manager that a connection has been made

Parameters addrType – address type of the peer device
 peerAddr – peer device address
 connHandle – connection handle
 role – master or slave role

Returns SUCCESS (0x00): GAPBondMgr was notified of link establishment

void GAPBondMgr_LinkTerm(uint16_t connHandle)

Notify the bond manager that a connection has been terminated

Parameters connHandle – connection handle

void GAPBondMgr_SlaveReqSecurity(uint16_t connHandle)

Notify the bond manager that a slave security request is received

Parameters connHandle – connection handle
 authReq – slave device's authentication requirements

uint8 GAPBondMgr_ResolveAddr(uint8 addrType, uint8 *pDevAddr, uint8 *pResolvedAddr)

Resolve an address from bonding information

Parameters addrType – address type of the peer device
 peerAddr – peer device address
 pResolvedAddr – pointer to buffer to put the resolved address

Returns Bonding index (0 – (GAP_BONDINGS_MAX-1): if address was found...
 GAP_BONDINGS_MAX: if address was not found

bStatus_t GAPBondMgr_ServiceChangeInd(uint16_t connectionHandle, uint8 setParam)

Set and clear the service change indication in a bond record

Parameters connHandle – connection handle of the connected device or 0xFFFF for all devices in database
 setParam – TRUE to set the service change indication, FALSE to clear it

Returns SUCCESS (0x00) – bond record found and changed
 bleNoResources (0x15) – no bond records found (for 0xFFFF connHandle)
 bleNotConnected (0x14) – connection with connHandle is invalid

bStatus_t GAPBondMgr_UpdateCharCfg(uint16 connectionHandle, uint16 attrHandle, uint16 value)

Update the characteristic configuration in a bond record

Parameters connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database
 attrHandle – attribute handle
 value – characteristic configuration value

Returns SUCCESS (0x00) – bond record found and changed
 bleNoResources (0x15) – no bond records found (for 0xFFFF connectionHandle)
 bleNotConnected (0x14) – connection with connectionHandle is invalid

void GAPBondMgr_Register(gapBondCBs_t *pCB)

Register callback functions with the bond manager

Parameters pCB – pointer to callback function structure (see [Section D.3](#))

bStatus_t GAPBondMgr_PasscodeRsp(uint16 connectionHandle, uint8 status, uint32 passcode)

Respond to a passcode request and update the passcode if possible

| | |
|-------------------|---|
| Parameters | connectionHandle – connection handle of the connected device or 0xFFFF for all devices in database status – SUCCESS if passcode is available, otherwise see SMP_PAIRING_FAILED_DEFINES in gapbondmgr.h passcode – integer value containing the passcode |
| Returns | SUCCESS (0x00): connection found and passcode was changed bleIncorrectMode (0x12): connectionHandle connection not found or pairing has not started INVALIDPARAMETER (0x02): passcode is out of range bleMemAllocError (0x13): heap is out of memory |

uint8 GAPBondMgr_ProcessGAPMsg(gapEventHdr_t *pMsg)

A bypass mechanism to allow the bond manager to process GAP messages.

| | |
|-------------------|--|
| Parameters | pMsg – GAP event message |
| Returns | TRUE: safe to deallocate incoming GAP message, FALSE: otherwise |

NOTE: This is an advanced feature and must not be called unless the normal GAP Bond Manager task ID registration is overridden.

uint8 GAPBondMgr_CheckNVLen(uint8 id, uint8 len)

This function checks the length of a bond manager NV Item.

| | |
|-------------------|---|
| Parameters | id – NV ID len – lengths in bytes of item |
| Returns | SUCCESS (0x00): NV item is the correct length FAILURE (0x01): NV item is an incorrect length |

bStatus_t GAPBondMgr_ReadCentAddrResChar(uint16 connectionHandle) *Send a Read by Type Request to get value attribute of Central Address resolution characteristic to determine if the peer device supports Enhanced Privacy. If applicable, a bond record is automatically updated based on the peer's response.*

Parameters

connHandle – connection handle of the connected device

Returns

SUCCESS (0x00): Request was sent successfully

INVALIDPARAMETER (0x02): Invalid connection handle or request field

bleNotConnected (0x14): Connection is down

blePending (0x16): A response is pending with this server

bleMemAllocError (0x13): Memory allocation error occurred

bleTimeout (0x17): Previous transaction timed out

uint8 GAPBondMgr_SupportsEnhancedPriv(uint8 *pPeerIdAddr) *Determine if the peer device supports enhanced privacy by checking the Enhanced Privacy state flag of the bond record that corresponds to the peer's identity address.*

Parameters pPeerIdAddr – pointer to peer identity address

Returns TRUE (0x00) – peer supports enhanced privacy
FALSE (0x01) – peer does not support enhanced privacy

bStatus_t GAPBondMgr_syncResolvingList(void) *Add all device address and IRK pairs from bond records to the controller.*

Returns

SUCCESS (0x00)

LL_STATUS_ERROR_BAD_PARAMETER (0x12) – invalid parameter

F.2 Configurable Parameters

| ParamID | R/W | Size | Description |
|-------------------------|-----|-----------|---|
| GAPBOND_PAIRING_MODE | R/W | uint8 | Whether to allow pairing, and if so, whether to initiate pairing. Possible values: <ul style="list-style-type: none"> GAPBOND_PAIRING_MODE_NO_PAIRING (0x00): pairing requests will be rejected GAPBOND_PAIRING_MODE_WAIT_FOR_REQ (0x01): GAPBondMgr waits to receive a pairing request GAPBOND_PAIRING_MODE_INITIATE (0x02): GAPBondMgr will send a pairing request after connection Default is GAPBOND_PAIRING_MODE_WAIT_FOR_REQ |
| GAPBOND_INITIATE_WAIT | R/W | uint16 | The amount of time to wait for a pairing request before sending the slave initiate request. Possible values: 0 to 65535 ms. Default is 1000 ms. |
| GAPBOND_MITM_PROTECTION | R/W | uint8 | Whether to turn on authenticated pairing. Possible Values: <ul style="list-style-type: none"> TRUE (0x00): use authenticated pairing FALSE (0x01): do not use authenticated pairing Default value: TRUE |
| GAPBOND_IO_CAPABILITIES | R/W | uint8 | The I/O capabilities of the local device. Possible values: <ul style="list-style-type: none"> GAPBOND_IO_CAP_DISPLAY_ONLY (0x00): display only device GAPBOND_IO_CAP_DISPLAY_YES_NO (0x01): Display and yes / no capable device GAPBOND_IO_CAP_DISPLAY_YES_NO (0x01): Display and yes / no capable device GAPBOND_IO_CAP_NO_INPUT_NO_OUTPUT (0x03): No display or input device GAPBOND_IO_CAP_KEYBOARD_DISPLAY (0x04): Both keyboard and display capable Default is GAPBOND_IO_CAP_DISPLAY_ONLY |
| GAPBOND_OOB_ENABLED | R/W | uint8 | Whether to use OOB for pairing. Possible values: <ul style="list-style-type: none"> 0x00: OOB is disabled, do not use OOB data 0x01: OOB is enabled, use OOB data Default is 0x00 |
| GAPBOND_OOB_DATA | R/W | uint8[16] | OOB data to use for pairing. Possible values: 0x00000000000000000000000000000000 – 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Default is 0x000000000000000000000000 |
| GAPBOND_BONDING_ENABLED | R/W | uint8 | Request bonding during the pairing process. Possible values: <ul style="list-style-type: none"> 0x00: Bonding is disabled, do not request bonding 0x01: Bonding is enabled, do request bonding Default is 0x00 |

| ParamID | R/W | Size | Description |
|---------------------------|-----|----------|---|
| GAPBOND_KEY_DIST_LIST | R/W | uint8 | <p>The key distribution list for bonding.</p> <p>Possible values: a bitwise OR of the following:</p> <ul style="list-style-type: none"> GAPBOND_KEYDIST_SENCKEY (0x01): Slave Encryption Key GAPBOND_KEYDIST_SIDKEY (0x02): Slave IRK and ID information GAPBOND_KEYDIST_SSIGN (0x04): Slave CSRK GAPBOND_KEYDIST_SLINK (0x08): Slave Link Key GAPBOND_KEYDIST_MENCKEY (0x10): Master Encryption Key GAPBOND_KEYDIST_MIDKEY (0x20): Master IRK and ID information GAPBOND_KEYDIST_MSIGN (0x40): Master CSRK GAPBOND_KEYDIST_MLINK (0x80): Master Link Key <p>Default value: a bitwise OR of the following:</p> <ul style="list-style-type: none"> GAPBOND_KEYDIST_SENCKEY GAPBOND_KEYDIST_SIDKEY GAPBOND_KEYDIST_MIDKEY GAPBOND_KEYDIST_MSIGN |
| GAPBOND_DEFAULT_PASS_CODE | R/W | uint32 | <p>The default passcode to use for passcode pairing.</p> <p>Possible values: 0 to 999999</p> <p>Default is 0.</p> |
| GAPBOND_ERASE_ALLBONDS | W | None | Erase all bonds from SNV and remove all bonded devices. |
| GAPBOND_KEYSIZE | R/W | uint8 | <p>Key Size used in pairing.</p> <p>Possible values: TGAP_SM_MIN_KEY_LEN - TGAP_SM_MAX_KEY_LEN. See the GAP API.</p> <p>Default is 16.</p> |
| GAPBOND_AUTO_SYNC_WL | R/W | uint8 | <p>First clears the whitelist. Then, each unique address stored by bonds in SNV will be synched with the whitelist.</p> <p>Possible values:</p> <ul style="list-style-type: none"> TRUE (0x00): synch whitelist FALSE (0x01): do not synch whitelist <p>Default is FALSE.</p> |
| GAPBOND_BOND_COUNT | R | uint8 | <p>Gets the total number of bonds stored in NV.</p> <p>Possible values: 0 to 256</p> <p>Default is 0.</p> |
| GAPBOND_BOND_FAIL_ACTION | W | uint8 | <p>Sets the action that the device takes after an unsuccessful bonding attempt.</p> <p>Possible values:</p> <ul style="list-style-type: none"> GAPBOND_FAIL_NO_ACTION (0x00): Take no action GAPBOND_FAIL_INITIATE_PAIRING (0x01): Reinitiate pairing GAPBOND_FAIL_TERMINATE_LINK (0x02): Terminate link GAPBOND_FAIL_TERMINATE_ERASE_BONDS (0x03): Terminate link and erase all existing bonds <p>Default value: GAPBOND_FAIL_TERMINATE_LINK</p> |
| GAPBOND_ERASE_SINGLEBOND | W | uint8[9] | <p>Erase a single bonded device.</p> <p>Possible values: a nine-byte array where the first byte is the address type and next 8 are the device address.</p> <p>First byte:</p> <ul style="list-style-type: none"> ADDRTYPE_PUBLIC (0x00): public device address ADDRTYPE_RANDOM (0x01): random device address <p>Default value: GAPBOND_SECURE_CONNECTION_ALLOW</p> |

| ParamID | R/W | Size | Description |
|-------------------------------|-----|------------------|--|
| GAPBOND_ECCKEY_REGEN_POLICY | W | uint8 | <p>Define reuse of the private and public ECC keys for multiple pairings. The default is to always regenerate the keys upon each new pairing. This parameter has no effect when the application specifies the keys using the GAPBOND_ECC_KEYS parameter. The behavior is that upon each pairing the number of recycles remaining is decremented by 1, but if the pairing fails the count is decremented by 3. The specification recommends that this value be set to no higher than 10 to avoid an attacker from learning too much about a private key before it is regenerated. Only applicable for Secure Connections.</p> <p>Possible values - 0-256 Default Value - 2</p> |
| GAPBOND_ECC_KEYS | R/W | gapBondEccKeys_t | <p>Allows the application to specify the private and public keys to use with pairing. When this is set, the keys are used indefinitely even if a regeneration policy was set with GAPBOND_DHKEY_REGEN_POLICY. To make the Bond Manager stop using these keys, pass a 1 byte value of 0x00. These keys are stored in RAM and are not retained in non-volatile memory. These keys can be defined by the application, or the application can request them using the SM_GetEccKeys command. Only applicable for Secure Connections.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • A valid gapBondEccKeys_t structure: these keys will be used • 0x00: previously passed keys will no longer be used. <p>Default values - By default, the keys are generated using GAPBOND_ECCKEY_REGEN_POLICY</p> |
| GAPBOND_REMOTE_OOB_SC_ENABLED | R/W | uint8 | <p>Indicate to the Bond Manager that any Secure Connections OOB data that has been received from a remote device, which has been supplied to the Bond Manager by the GAPBOND_REMOTE_OOB_SC_DATA parameter, is valid. Only applicable for Secure Connections.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • 0x00: The remote OOB data is not valid • 0x01: The remote OOB data is valid <p>Default value - 0x00</p> |
| GAPBOND_REMOTE_OOB_SC_DATA | R/W | gapBondOobSC_t | <p>Used to pass OOB Secure Connections data to the bond manager that has been received from a remote device. This data is not only the 16 bytes of OOB data, but also the 32-byte ECC Public Key X-Coordinate of the remote device (that it must use when pairing) and a 16-byte confirmation value computed using the said public key, OOB data is used as input to the SM_F4 function. This data can be invalidated by writing 0x00 to the GAPBOND_REMOTE_OOB_SC_ENABLED parameter. Only applicable for Secure Connections.</p> <p>Possible values: A valid gapBondOobSC_t structure: this is used for pairing Default Value - All zeroes</p> |
| GAPBOND_LOCAL_OOB_SC_ENABLED | R/W | uint8 | <p>Indicates to the Bond Manager that the local device has valid OOB data it has attempted to send to the remote device, and that the local OOB data supplied to the Bond Manager by the GAPBOND_LOCAL_OOB_SC_DATA parameter is valid. This is needed to determine if the OOB protocol is expected to be used for Secure Connections. Only one device needs to have received OOB data for OOB pairing to be used in Secure Connections. Only applicable for Secure Connections.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • 0x00: The Local OOB data is not valid • 0x01: The Local OOB data is valid <p>Default value - 0x00</p> |
| GAPBOND_LOCAL_OOB_SC_DATA | R/W | uint8[16] | <p>Passes Secure Connections data to the bond manager that was sent to the remote device. This is only the 16 bytes of byte OOB data, which is needed to complete pairing. Only applicable for Secure Connections.</p> <p>Possible values: 0x00000000000000000000000000000000 – 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF Default value: 0x00000000000000000000000000000000</p> |

| ParamID | R/W | Size | Description |
|------------------------------|-----|-------|--|
| GAPBOND_LRU_BOND_REPLACEMENT | R/W | uint8 | <p>Whether to enable the least recently used bond scheme so that, when a newly bonded device is added and all the entries are full, the least recently used device's bond is deleted to make room.</p> <p>Possible values:</p> <ul style="list-style-type: none"> 0x00: Disabled, do not use LRU scheme 0x01: Enabled, do use LRU scheme <p>Default value - 0x00</p> |

F.3 Callbacks

These callbacks are functions whose pointers are passed from the application to the GAPBondMgr so that it can return events to the application as required. They are passed as the following structure.

```
typedef struct
{
    pfnPasscodeCB_t    passcodeCB;    //!< Passcode callback
    pfnPairStateCB_t   pairStateCB;   //!< Pairing state callback
} gapBondCBs_t;
```

F.3.1 Passcode Callback (passcodeCB)

This callback returns to the application the peer device information when a passcode is requested during the pairing process or when numeric comparison is used. This function is defined as follows.

```
typedef void (*pfnPasscodeCB_t)
(
    uint8 *deviceAddr,
    uint16 connectionHandle,
    uint8 uiInputs,
    uint8 uiOutputs,
    uint32 numComparison
);
```

The parameters are described in more detail here:

deviceAddr: Pointer to 6-byte device address which the current pairing process relates to.

connectionHandle: Connection handle of the current pairing process

uiInputs / uiOutputs: These dictate what role the local device should play in the passcode pairing process. If uiInputs is TRUE, the local device should accept a passcode input. If uiOutputs is TRUE, the local device should display the passcode. uiInputs and uiOutputs are never both true.

numComparison: If this is a nonzero value, then it is the code that should be displayed for numeric comparison pairing. If this is zero, then passcode pairing is occurring.

F.3.2 Pairing State Callback (pairStateCB)

This callback returns the current pairing state to the application whenever the state changes and as the current status of the pairing or bonding process associated with the current state. This function is defined as follows.

```
typedef void (*pfnPairStateCB_t)
(
    uint16 connectionHandle,
    uint8 state,
    uint8 status
);
```

The parameters are described in more detail here:

connectionHandle: Connection handle of the current pairing and bonding process

state / status: The pairing states are listed here with possible statuses:

- GAPBOND_PAIRING_STATE_STARTED

- The following status are possible for this state:
 - SUCCESS (0x00): pairing has been initiated. A pairing request has been either sent or received.
- GAPBOND_PAIRING_STATE_COMPLETE
 - The following statuses are possible for this state:
 - SUCCESS (0x00): pairing is complete (Session keys have been exchanged.)
 - SMP_PAIRING_FAILED_PASSKEY_ENTRY_FAILED (0x01): user input failed
 - SMP_PAIRING_FAILED_OOB_NOT_AVAIL (0x02): Out-of-band data not available
 - SMP_PAIRING_FAILED_AUTH_REQ (0x03): Input and output capabilities of devices do not allow for authentication
 - SMP_PAIRING_FAILED_CONFIRM_VALUE (0x04): the confirm value does not match the calculated compare value
 - SMP_PAIRING_FAILED_NOT_SUPPORTED (0x05): pairing is unsupported
 - SMP_PAIRING_FAILED_ENC_KEY_SIZE (0x06): encryption key size is insufficient
 - SMP_PAIRING_FAILED_CMD_NOT_SUPPORTED (0x07): The SMP command received is unsupported on this device
 - SMP_PAIRING_FAILED_UNSPECIFIED (0x08): encryption failed to start
 - bleTimeout (0x17): pairing failed to complete before timeout
 - bleGAPBondRejected (0x32): keys did not match
- GAPBOND_PAIRING_STATE_BONDED
 - The following statuses are possible for this state:
 - SUCCESS: bonding is complete
 - LL_ENC_KEY_REQ_REJECTED (0x06): encryption key is missing
 - LL_ENC_KEY_REQ_UNSUPPORTED_FEATURE (0x1A): feature is unsupported by the remote device
 - LL_CTRL_PKT_TIMEOUT_TERM (0x22): Timeout waiting for response
 - bleGAPBondRejected (0x32): this is received due to one of the previous three errors

L2CAP API

G.1 Commands

This section describes the API related to setting up bidirectional communication between two Bluetooth low energy devices using L2CAP connection orientated channels. The return values described in this section are only the possible return values from processing the command. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

bStatus_t L2CAP_RegisterPsm(l2capPsm_t *pPsm)

Register a protocol or service multiplexer with L2CAP

| | |
|-------------------|--|
| Parameters | pPsm: pointer to PSM structure |
| Returns | <p>SUCCESS (0x00): Registration was successful.</p> <p>INVALIDPARAMETER (0x02): maximum number of channels is greater than total supported</p> <p>bleInvalidRange (0x18): PSM value is out of range</p> <p>bleInvalidMtuSize (0x1B): MTU size is out of range</p> <p>bleNoResources (0x15): out of resources</p> <p>bleAlreadyInRequestedMode (0x11): PSM already registered</p> |

bStatus_t L2CAP_DeregisterPsm(uint8 taskId, uint16 psm)

Deregister a protocol or service multiplexer with L2CAP

| | |
|-------------------|---|
| Parameters | <p>taskId – the task to which PSM belongs</p> <p>psm – PSM to deregister</p> |
| Returns | <p>SUCCESS (0x00): Registration was successful.</p> <p>INVALIDPARAMETER (0x02): PSM or task ID is invalid.</p> <p>bleIncorrectMode (0x12): PSM is in use.</p> |

bStatus_t L2CAP_PsmInfo(uint16 psm, l2capPsmInfo_t *pInfo)

Get information about a given registered PSM

| | |
|-------------------|---|
| Parameters | <p>pPsm: PSM ID</p> <p>pInfo – structure into which to copy PSM information</p> |
| Returns | <p>SUCCESS (0x00): Operation was successful.</p> <p>INVALIDPARAMETER (0x02): PSM is not registered.</p> |

bStatus_t L2CAP_PsmChannels(uint16 psm, uint8 numCIDs, uint16 *pCIDs)
Get all active channels for a given registered PSM

Parameters pPsm: PSM ID
 numCIDs – number of CIDs can be copied
 pCIDs – structure into which to copy CIDs

Returns SUCCESS (0x00): Operation was successful.
 INVALIDPARAMETER (0x02): PSM is not registered.

bStatus_t L2CAP_ChannelInfo(uint16 CID, l2capChannelInfo_t *pInfo)
Get information about an active connection-oriented channel

Parameters CID – local channel ID
 pInfo – structure into which to copy channel information

Returns SUCCESS (0x00): Registration was successful.
 INVALIDPARAMETER (0x02): No such channel

bStatus_t L2CAP_ConnectReq(uint16 connHandle, uint16 psm, uint16 peerPsm)
Send connection request

Parameters connHandle – connection handle
 id – identifier received in connection request
 result – outcome of connection request

Returns SUCCESS (0x00): Request was sent successfully.
 INVALIDPARAMETER (0x02): PSM is not registered.
 MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available
 bleIncorrectMode (0x12): PSM not registered
 bleNotConnected (0x14): Connection is down.
 bleNoResources (0x15): No available resource.
 bleMemAllocError (0x13): Memory allocation error occurred.

bStatus_t L2CAP_ConnectRsp(uint16 connHandle, uint8 id, uint16 result)
Send connection response.

Parameters connHandle – connection onto which to create channel
 psm – local PSM
 peerPsm – peer PSM

Returns SUCCESS: (0x00) Request was sent successfully.
 INVALIDPARAMETER (0x02): PSM is not registered or Channel is closed.
 MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.
 bleNotConnected (0x14): Connection is down.
 bleMemAllocError (0x13): Memory allocation error occurred.

L2CAP_DisconnectReq(uint16 CID)
Send disconnection request.

Parameters CID – local CID to disconnect

Returns SUCCESS (0x00): Request was sent successfully.
 INVALIDPARAMETER (0x02): Channel ID is invalid.
 MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.
 bleNotConnected (0x14): Connection is down.
 bleNoResources (0x15): No available resource
 bleMemAllocError (0x13): Memory allocation error occurred.

bStatus_t L2CAP_FlowCtrlCredit(uint16 CID, uint16 peerCredits)

Send flow control credit.

Parameters

CID – local CID

peerCredits – number of credits to give to peer device

Returns

SUCCESS (0x00): Request was sent successfully.

INVALIDPARAMETER (0x02): Channel is not open.

MSG_BUFFER_NOT_AVAIL (0x04): No HCI buffer is available.

bleNotConnected (0x14): Connection is down.

bleInvalidRange (0x18): Credits is out of range.

bleMemAllocError (0x13): Memory allocation error occurred.

bStatus_t L2CAP_SendSDU(l2capPacket_t *pPkt)

Send data packet over an L2CAP connection-oriented channel established over a physical connection.

Parameters

pPkt – pointer to packet to be sent

Returns

SUCCESS (0x00): Data was sent successfully.

INVALIDPARAMETER (0x02): SDU payload is null.

bleInvalidRange (0x18): PSM value is out of range.

bleNotConnected (0x14): Connection or Channel is down.

bleMemAllocError (0x13): Memory allocation error occurred.

blePending (0x16): Another transmit in progress.

bleInvalidMtuSize (0x1B): SDU size is larger than peer MTU.

HCI API

This section describes the vendor specific HCI Extension API, the HCI LE API, and the HCI Support API. An example is provided when more detail is required. The return values for these commands is always SUCCESS unless otherwise specified. This return value does not indicate successful completion of the command. These commands result in corresponding events that must be checked by the calling application. If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

H.1 HCI Commands

This section lists the mapping between stack APIs and function names, as described in the BT core spec. See [Section 5.7.2](#) for an example of how to implement these commands using the BT core spec. The App column in [Table H-1](#) indicates if the command can be called by the application where Y means yes and N means no.

Table H-1. API Function Map

| Stack API | BT Function | App |
|--------------------------------------|--|-----|
| HCI_DisconnectCmd | Disconnect Command | N |
| HCI_ReadRemoteVersionInfoCmd | Read Remote Version Information Command | Y |
| HCI_SetEventMaskCmd | Set Event Mask Command | Y |
| HCI_SetEventMaskPage2Cmd | Set Event Mask Page 2 Command | Y |
| HCI_ResetCmd | Reset Command | Y |
| HCI_ReadTransmitPowerLevelCmd | Read Transmit Power Level Command | Y |
| HCI_SetControllerToHostFlowCtrlCmd | Set Controller To Host Flow Control Command | N |
| HCI_HostBufferSizeCmd | Host Buffer Size Command | N |
| HCI_HostNumCompletedPktCmd | Host Number Of Completed Packets Command | N |
| HCI_ReadAuthPayloadTimeoutCmd | Read Authenticated Payload Timeout Command | N |
| HCI_WriteAuthPayloadTimeoutCmd | Write Authenticated Payload Timeout Command | N |
| HCI_ReadLocalSupportedCommandsCmd | Read Local Supported Commands Command | Y |
| HCI_ReadLocalSupportedFeaturesCmd | Read Local Supported Features Command | Y |
| HCI_ReadLocalVersionInfoCmd | Read Local Version Information Command | Y |
| HCI_ReadBDADDRCmd | Read BD_ADDR Command | Y |
| HCI_ReadRssiCmd | Read RSSI Command | Y |
| HCI_LE_SetEventMaskCmd | LE Set Event Mask Command | Y |
| HCI_LE_ReadBufSizeCmd | LE Read Buffer Size Command | N |
| HCI_LE_ReadLocalSupportedFeaturesCmd | LE Read Local Supported Features Command | Y |
| HCI_LE_SetRandomAddressCmd | LE Set Random Address Command | N |
| HCI_LE_SetAdvParamCmd | LE Set Advertising Parameters Command | N |
| HCI_LE_SetAdvDataCmd | LE Set Advertising Data Command | N |
| HCI_LE_SetScanRspDataCmd | LE Set Scan Response Data Command | N |
| HCI_LE_SetAdvEnableCmd | LE Set Advertise Enable Command | N |
| HCI_LE_ReadAdvChanTxPowerCmd | LE Read Advertising Channel Tx Power Command | Y |
| HCI_LE_SetScanParamCmd | LE Set Scan Parameters Command | N |
| HCI_LE_SetScanEnableCmd | LE Set Scan Enable Command | N |

Table H-1. API Function Map (continued)

| Stack API | BT Function | App |
|--|---|-----|
| HCI_LE_CreateConnCmd | LE Create Connection Command | N |
| HCI_LE_CreateConnCancelCmd | LE Create Connection Cancel Command | N |
| HCI_LE_ReadWhiteListSizeCmd | LE Read White List Size Command | Y |
| HCI_LE_ClearWhiteListCmd | LE Clear White List Command | Y |
| HCI_LE_AddWhiteListCmd | LE Add Device To White List Command | Y |
| HCI_LE_RemoveWhiteListCmd | LE Remove Device From White List Command | Y |
| HCI_LE_SetHostChanClassificationCmd | LE Set Host Channel Classification Command | Y |
| HCI_LE_ReadChannelMapCmd | LE Read Channel Map Command | Y |
| HCI_LE_ReadRemoteUsedFeaturesCmd | LE Read Remote Used Features Command | Y |
| HCI_LE_EncryptCmd | LE Encrypt Command | Y |
| HCI_LE_RandCmd | LE Rand Command | N |
| HCI_LE_StartEncryptCmd | LE Start Encryption Command | N |
| HCI_LE_LtkReqReplyCmd | LE Long Term Key Request Reply Command | N |
| HCI_LE_LtkReqNegReplyCmd | LE Long Term Key Request Negative Reply Command | N |
| HCI_LE_ReadSupportedStatesCmd | LE Read Supported States Command | Y |
| HCI_LE_ReceiverTestCmd | LE Receiver Test Command | Y |
| HCI_LE_TransmitterTestCmd | LE Transmitter Test Command | Y |
| HCI_LE_TestEndCmd | LE Test End Command | Y |
| HCI_LE_RemoteConnParamReqReplyCmd | LE Remote Connection Parameter Request Reply Command | N |
| HCI_LE_RemoteConnParamReqNegReplyCmd | LE Remote Connection Parameter Request Negative Reply Command | N |
| HCI_LE_SetDataLenCmd | LE Set Data Length Command | Y |
| HCI_LE_ReadSuggestedDefaultDataLenCmd | LE Read Suggested Default Data Length Command | Y |
| HCI_LE_WriteSuggestedDefaultDataLenCmd | LE Write Suggested Default Data Length Command | Y |
| HCI_LE_ReadMaxDataLenCmd | LE Read Maximum Data Length Command | Y |
| HCI_LE_AddDeviceToResolvingListCmd | LE Add Device to Resolving List Command | N |
| HCI_LE_RemoveDeviceFromResolvingListCmd | LE Remove Device From Resolving List Command | N |
| HCI_LE_ClearResolvingListCmd | LE Clear Resolving List Command | N |
| HCI_LE_ReadResolvingListSizeCmd | LE Read Resolving List Size Command | N |
| HCI_LE_ReadPeerResolvableAddressCmd | LE Read Peer Resolvable Address Command | N |
| HCI_LE_ReadLocalResolvableAddressCmd | LE Read Local Resolvable Address Command | N |
| HCI_LE_SetAddressResolutionEnableCmd | LE Set Address Resolution Enable Command | N |
| HCI_LE_SetResolvablePrivateAddressTimeoutCmd | LE Set Resolvable Private Address Timeout Command | N |
| HCI_LE_ReadLocalP256PublicKeyCmd | LE Read Local P-256 Public Key Command | N |
| HCI_LE_GenerateDHKeyCmd | LE Generate DHKey Command | N |

H.2 Vendor-Specific HCI Commands

This section describes the vendor-specific HCI Extension API. Examples are provided when more detail is required. The return values for these commands are always SUCCESS unless otherwise specified. This return value does not indicate successful completion of the command; it only indicates the command has been sent to the protocol stack. These commands result in corresponding events that must be checked by the calling application. See [Section 5.7.3](#) for an example of this. The events are defined in the TI HCI Vendor Specific Guide.

If ICall is incorrectly configured or does not have enough memory to allocate a message, the command is never processed by the Bluetooth low energy stack. In this case, one of the ICall return values from [Appendix I](#) is returned.

Table H-2 maps the stack APIs with the function names from the TI HCI Vendor Specific API Guide.

Table H-2. API Function Map

| Stack API | TI HCI Vendor-Specific API Guide Function Name |
|--------------------------------------|--|
| HCI_EXT_SetRxGainCmd | HCI Extension Set Receiver Gain |
| HCI_EXT_SetTxPowerCmd | HCI Extension Set Transmitter Power |
| HCI_EXT_OnePktPerEvtCmd | HCI Extension One Packet Per Event |
| HCI_EXT_DecryptCmd | HCI Extension Decrypt |
| HCI_EXT_SetLocalSupportedFeaturesCmd | HCI Extension Set Local Supported Features |
| HCI_EXT_SetFastTxResponseTimeCmd | HCI Extension Set Fast Transmit Response Time |
| HCI_EXT_SetSlaveLatencyOverrideCmd | HCI Extension Set Slave Latency Override |
| HCI_EXT_ModemTestTxCmd | HCI Extension Modem Test Transmit |
| HCI_EXT_ModemHopTestTxCmd | HCI Extension Modem Hop Test Transmit |
| HCI_EXT_ModemTestRxCmd | HCI Extension Modem Test Receive |
| HCI_EXT_EndModemTestCmd | HCI Extension End Modem Test |
| HCI_EXT_SetBDADDRCmd | HCI Extension Set BDADDR |
| HCI_EXT_SetSCACmd | HCI Extension Set SCA |
| HCI_EXT_EnablePTMCmd | HCI Extension Enable PTM |
| HCI_EXT_SetMaxDtmTxPowerCmd | HCI Extension Set Max DTM Transmitter Power |
| HCI_EXT_DisconnectImmedCmd | HCI Extension Disconnect Immediate |
| HCI_EXT_PacketErrorRateCmd | HCI Extension Packet Error Rate |
| HCI_EXT_PERbyChanCmd | HCI Extension Packet Error Rate By Channel |
| HCI_EXT_AdvEventNoticeCmd | HCI Extension Advertiser Event Notice |
| HCI_EXT_ConnEventNoticeCmd | HCI Extension Connection Event Notice |
| HCI_EXT_BuildRevisionCmd | HCI Extension Build Revision |
| HCI_EXT_DelaySleepCmd | HCI Extension Delay Sleep |
| HCI_EXT_ResetSystemCmd | HCI Extension Reset System |
| HCI_EXT_NumCompIPktsLimitCmd | HCI Extension Number Completed Packets Limit |
| HCI_EXT_GetConnInfoCmd | HCI Extension Get Connection Information |

hciStatus_t HCI_EXT_AdvEventNoticeCmd (uint8 taskID, uint16 taskEvent)

This command configures the device to set an event in the user task after each advertisement event completes. A nonzero taskEvent value is enable, while a zero valued taskEvent is disable.

Parameters

taskID – task ID of the user

taskEvent – task event of the user (This event must be a single bit value.)

Returns

SUCCESS: event configured correctly

LL_STATUS_ERROR_BAD_PARAMETER: more than one bit set exists

NOTE: This command does not return any events but has a meaningful return status and requires additional checks in the task function as described in [Section 4.3.2.1](#).

Example (code additions to simple_peripheral.c):

1. Define the event in the application.

```
// BLE Stack Events
#define SBP_ADV_CB_EVT    0x0001
```

2. Configure the Bluetooth low energy protocol stack to return the event (in simple_peripheral_init()).

```

HCI_EXT_AdvEventNoticeCmd( selfEntity, SBP_ADV_CB_EVT);
3. Check for and receive these events in the application (simple_peripheral_taskFxn()).
if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
{
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
        ICall_Event *pEvt = (ICall_Event *)pMsg;

        // Check for BLE stack events first
        if (pEvt->signature == 0xffff)
        {
            if (pEvt->event_flag & SBP_ADV_CB_EVT)
            {
                // Advertisement ended. Process as desired.
            }
        }
    }
}
...

```

hciStatus_t HCI_EXT_BuildRevision(uint8 mode, uint16 userRevNum)

*This command allows the embedded user code to set their own 16-bit revision number or read the build revision number of the Bluetooth low energy stack library software. The default value of the revision number is zero. When you update a Bluetooth low energy project by adding their own code, use this API to set your own revision number. When called with mode set to **HCI_EXT_SET_APP_REVISION**, the stack saves this value. No event is returned from this API when used this way. TI intended the event to be called from within the target. That the event is intended to be called from within the target does not preclude this command from being received through the HCI. No event is returned.*

Parameters
 Mode – HCI_EXT_SET_APP_REVISION, HCI_EXT_READ_BUILD_REVISION
 userRevNum – Any 16-bit value

Returns (only when mode == HCI_EXT_SET_USER_REVISION)
 SUCCESS: build revision set successfully
 LL_STATUS_ERROR_BAD_PARAMETER: an invalid mode

Corresponding Events (only when mode == HCI_EXT_SET_USER_REVISION)
 HCI_VendorSpecificCommandCompleteEvent

hciStatus_t HCI_EXT_ConnEventNoticeCmd (uint16 connHandle, uint8 taskID, uint16 taskEvent)

This command configures the device to set an event in the user task after each connection event completes. A non-zero taskEvent value is enable, while a zero valued taskEvent is disable.

Parameters
 connHandle – connection ID for connection event notice
 taskID – task ID of the user
 taskEvent – task event of the user

Returns
 SUCCESS or FAILURE
 LL_STATUS_ERROR_BAD_PARAMETER: more than one bit set exists

NOTE: This command does not return any events but it has a meaningful return status and requires additional checks in the task function as described in [Section 4.3.2.1](#).

Example (code additions to simple_peripheral.c):

1. Define the event in the application.

```
// BLE Stack Events
#define SBP_CON_CB_EVT 0x0001
```

2. Configure the Bluetooth low energy protocol stack to return the event (in simple_peripheral_processStateChangeEvt()) after the connection is established.

```
case GAPROLE_CONNECTED:
{
    HCI_EXT_ConnEventNoticeCmd ( connHandle, selfEntity, SBP_CON_CB_EVT );
```

3. Check for and receive these events in the application (simple_peripheral_taskFxn()).

```
if (ICall_fetchServiceMsg(&src, &dest, (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
{
    if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
    {
        ICall_Event *pEvt = (ICall_Event *)pMsg;

        // Check for BLE stack events first
        if (pEvt->signature == 0xffff)
        {
            if (pEvt->event_flag & SBP_CON_CB_EVT)
            {
                // Connection Event ended. Process as desired.
            }
        }
    }
    ...
}
```

hciStatus_t HCI_EXT_DecryptCmd (uint8 *key, uint8 * encText)

This command decrypts encrypted data using the AES128.

| | |
|-----------------------------|--|
| Parameters | key – Pointer to 16-byte encryption key encText – Pointer to 16-byte encrypted data |
| Corresponding Events | HCI_VendorSpecificCommandCompleteEvent |

hciStatus_t HCI_EXT_DisconnectImmedCmd (uint16 connHandle)

This command disconnects a connection immediately. This command is useful when a connection must be ended without the latency associated with the normal Bluetooth low energy controller terminate control procedure. The host issuing the command receives the HCI disconnection complete event with a reason status of 0x16 (that is, Connection Terminated by Local Host), followed by an HCI vendor-specific event.

| | |
|-----------------------------|--|
| Parameters | connHandle – The handle of the connection |
| Corresponding Events | HCI_Disconnection_Complete HCI_VendorSpecificCommandCompleteEvent |

hciStatus_t HCI_EXT_EnablePTMCmd (void)

This command enables production test mode (PTM). This mode is used by the customer during assembly of their product to allow limited access to the Bluetooth low energy controller for testing and configuration. This mode remains enabled until the device is reset. See the related application note for additional details.

Return Values HCI_SUCCESS: Successfully entered PTM

NOTE: This command causes a reset of the controller. To reenter the application, reset the device. This command does not return any events.

hciStatus_t HCI_EXT_EndModemTestCmd (void)

This command shuts down a modem test. A complete link layer reset occurs.

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

hciStatus_t HCI_EXT_GetConnInfoCmd(uint8 *numAllocConns, uint8 *numActiveConns, hciConnInfo_t *activeConnInfo)

This command acquires connection related information: number of allocated connections, the number of active connections, connection ID, connection role, peer address, and address type. The number of allocated connections can be modified with the MAX_NUM_BLE_CONNS define in bleUserConfig.h (see [Section 5.8](#))

Parameters
 numAllocConns – pointer to number of build time connections allowed
 numActiveConns – pointer to number of active Bluetooth low energy connections
 activeConnInfo – pointer for active connection information

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: If all the parameters are NULL, the command is assumed to have originated from the transport layer. Otherwise, the command is assumed to have originated from a direct call by the application and any non-NULL pointer is used.

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

This API is used to start a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37-byte packet of pseudo-random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0 to 39) every 625 μ s. Use the HCI_EXT_EndModemTest command to end the test.

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs. The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

hciStatus_t HCI_EXT_ModemTestRxCmd(uint8 rxFreq)

This API starts a continuous receiver modem test using a modulated carrier wave tone, at the frequency that corresponds to the specific RF channel. Any received data is discarded. Receiver gain can be adjusted using the HCI_EXT_SetRxGain command. RSSI may be read during this test by using the HCI_ReadRssi command. Use HCI_EXT_EndModemTest command to end the test.

Parameters rxFreq – selects which channel [0 to 39] on which to receive

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: The RF channel is specified, not the Bluetooth low energy frequency. The RF channel can be obtained from the Bluetooth low energy frequency as follows: RF channel = (Bluetooth low energy frequency – 2402) ÷ 2.

When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

hciStatus_t HCI_EXT_ModemTestTxCmd(uint8 cwMode, uint8 txFreq)

This API starts a continuous transmitter modem test, using either a modulated or unmodulated carrier wave tone, at the frequency that corresponds to the specified RF channel. Use the HCI_EXT_EndModemTest command to end the test.

Parameters cwMode – HCI_EXT_TX_MODULATED_CARRIER, HCI_EXT_TX_UNMODULATED_CARRIER

txFreq – Transmit RF channel k = 0 to 39, where Bluetooth low energy frequency = 2402 + (k × 2 MHz)

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: The RF channel, not the Bluetooth low energy frequency, is specified by txFreq. The RF channel can be obtained from the Bluetooth low energy frequency as follows: RF channel = (Bluetooth low energy frequency – 2402) ÷ 2.

When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

hciStatus_t HCI_EXT_NumComplPktsLimitCmd (uint8 limit, uint8 flushOnEvt)

This command sets the limit on the minimum number of complete packets before a number of completed packets event is returned by the controller. If the limit is not reached by the end of a connection event, then the number of completed packets event is returned (if non-zero) based on the flushOnEvt flag. The limit can be set from one to the maximum number of HCI buffers (see the LE Read Buffer Size command in the Bluetooth Core specification). The default limit is one; the default flushOnEvt flag is FALSE.

Parameters

limit – from 1 to HCI_MAX_NUM_DATA_BUFFERS (returned by HCI_LE_ReadBufSizeCmd)

flushOnEvt

- HCI_EXT_DISABLE_NUM_COMPL_PKTS_ON_EVENT: return only a number of completed packets event when the number of completed packets is greater than or equal to the limit
- HCI_EXT_ENABLE_NUM_COMPL_PKTS_ON_EVENT: return the number of completed packets event at the end of every connection event

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: This command minimizes the overhead of sending multiple number of completed packet events, maximizing the processing available to increase over-the-air throughput.

hciStatus_t HCI_EXT_OnePktPerEvtCmd (uint8 control)

This command configures the link layer to allow only one packet per connection event. The default system value for this feature is disabled. This command can be used to tradeoff throughput and power consumption during a connection. When enabled, power can be conserved during a connection by limiting the number of packets per connection event to one, at the expense of more limited throughput. When disabled, the number of packets transferred during a connection event is not limited, at the expense of higher power consumption per connection event.

Parameters

control – HCI_EXT_DISABLE_ONE_PKT_PER_EVT,
HCI_EXT_ENABLE_ONE_PKT_PER_EVT

Corresponding Events HCI_VendorSpecificCommandCompleteEvent: this event is returned only if the setting is changing from enable to disable or from disable to enable

NOTE: A thorough power analysis of the system requirements to be performed before it is certain that this command saves power. Transferring multiple packets per connection event may be more power efficient.

hciStatus_t HCI_EXT_PacketErrorRateCmd (uint16 connHandle, uint8 command)

This command is used to reset or read the packet error rate counters for a connection. When reset, the counters are cleared; when read, the total number of packets received, the number of packets received with a CRC error, the number of events, and the number of missed events are returned.

Parameters
connHandle – the connection ID on which to perform the command
command – HCI_EXT_PER_RESET, HCI_EXT_PER_READ

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: The counters are 16 bits. At the shortest connection interval, 16-bit counters provide a little over 8 minutes of data.

hciStatus_t HCI_EXT_PERbyChanCmd (uint16 connHandle, perByChan_t *perByChan) *This command starts or ends the packet error rate by channel counter accumulation for a connection, and can be used by an application to make coexistence assessments. Based on the results, an application can perform an update channel classification command to limit channel interference from other wireless standards. If *perByChan is NULL, counter accumulation is discontinued. If *perByChan is not NULL, this location for the PER data has sufficient memory, based on the following type definition perByChan_t located in ll.h:*

```
#define LL_MAX_NUM_DATA_CHAN 37
// Packet Error Rate Information By Channel
typedef struct
{
    uint16 numPkts[ LL_MAX_NUM_DATA_CHAN ];
    uint16 numCrcErr[ LL_MAX_NUM_DATA_CHAN ];
} perByChan_t;
```

NOTE: Ensure there is sufficient memory allocated in the perByChan structure and maintain the counters, clearing them if required before starting accumulation.

The counters are 16 bits. At the shortest connection interval, 16-bit counters provide a bit more than 8 minutes of data.

This command can be used in combination with HCI_EXT_PacketErrorRateCmd.

Parameters
connHandle – The connection ID on which to accumulate the data.
perByChan – Pointer to PER by channel data, or NULL

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

hciStatus_t HCI_EXT_ModemHopTestTxCmd(void)

This API starts a continuous transmitter direct test mode test using a modulated carrier wave and transmitting a 37-byte packet of pseudo-random 9-bit data. A packet is transmitted on a different frequency (linearly stepping through all RF channels 0 to 39) every 625 μ s. Use the HCI_EXT_EndModemTest command to end the test.

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: When the HCI_EXT_EndModemTest is issued to stop this test, a controller reset occurs.

The device transmits at the default output power (0 dBm) unless changed by HCI_EXT_SetTxPowerCmd.

hciStatus_t HCI_EXT_ResetSystemCmd (uint8 mode)

This command issues a hard or soft system reset. A hard reset is caused by a watchdog timer time-out, while a soft reset is caused by jumping to the reset ISR.

Parameters mode – HCI_EXT_RESET_SYSTEM_HARD

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: The reset occurs after a 100-ms delay to allow the correspond event to be returned to the application.

Only a hard reset is allowed. A soft reset causes the command to fail. See [Section 8.2](#).

hciStatus_t HCI_EXT_SetBDADDRCmd(uint8 *bdAddr)

This command sets the Bluetooth low energy address (BDADDR) of the device. This address overrides the address of the device determined when the device is reset). To restore the initialized address of the device stored in flash, issue this command with an invalid address (0xFFFFFFFFFFFF).

Parameters bdAddr – A pointer to a buffer to hold the address of this device. An invalid address (that is, all FFs) restores the address of this device to the address set at initialization.

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

hciStatus_t HCI_EXT_SetFastTxResponseTimeCmd (uint8 control)

This command configures the link layer fast transmit response time feature. The default system value for this feature is enabled.

Parameters control – HCI_EXT_ENABLE_FAST_TX_RESP_TIME,
HCI_EXT_DISABLE_FAST_TX_RESP_TIME

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: This command is only valid for a slave controller. When the host transmits data, the controller (by default) ensures the packet is sent over the link layer connection with minimal delay, even when the connection is configured to use slave latency. That is, the transmit response time is at or less than the connection interval (instead of waiting for the next effective connection interval due to slave latency). This transmit time results in lower power savings because the link layer may wake to transmit during connection events that would have been skipped due to slave latency. If saving power is more critical than fast transmit response time, you can disable this feature using this command. When disabled, the transmit response time is at or less than the effective connection interval (slave latency + 1× the connection interval).

hciStatus_t HCI_EXT_SetLocalSupportedFeaturesCmd (uint8 * localFeatures)

This command sets the local supported features of the controller.

| | |
|-----------------------------|--|
| Parameters | <p>localFeatures – A pointer to the feature set where each bit where each bit corresponds to a feature</p> <p>0: Feature is not used</p> <p>1: Feature can be used</p> |
| Corresponding Events | HCI_VendorSpecificCommandCompleteEvent |

NOTE: This command can be issued either before or after one or more connections are formed. The local features set in this manner are only effective if performed before a feature exchange procedure has been initiated by the master. When this control procedure has been completed for a particular connection, only the exchanged feature set for that connection is used. Because the link layer may initiate the feature exchange procedure autonomously, use this command before the connection is formed. The features are initialized by the controller upon start-up. You are unlikely to require this command. The defined symbols for the feature values are in ll.h.

hciStatus_t HCI_EXT_SetMaxDtmTxPowerCmd (uint8 txPower)

This command overrides the RF transmitter output power used by the direct test mode (DTM). The maximum transmitter output power setting used by DTM is typically the maximum transmitter output power setting for the device (that is, 5 dBm). This command changes the value used by DTM.

Parameters

txPower – one of the following:

- HCI_EXT_TX_POWER_MINUS_21_DBM
- HCI_EXT_TX_POWER_MINUS_18_DBM
- HCI_EXT_TX_POWER_MINUS_15_DBM
- HCI_EXT_TX_POWER_MINUS_12_DBM
- HCI_EXT_TX_POWER_MINUS_9_DBM
- HCI_EXT_TX_POWER_MINUS_6_DBM
- HCI_EXT_TX_POWER_MINUS_3_DBM
- HCI_EXT_TX_POWER_0_DBM
- HCI_EXT_TX_POWER_1_DBM
- HCI_EXT_TX_POWER_2_DBM
- HCI_EXT_TX_POWER_3_DBM
- HCI_EXT_TX_POWER_4_DBM
- HCI_EXT_TX_POWER_5_DBM

Corresponding Events HCI_VendorSpecificCommandCompleteEvent

NOTE: When DTM is ended by a call to HCI_LE_TestEndCmd or a HCI_Reset is used, the transmitter output power setting is restored to the default value of 0 dBm.

hciStatus_t HCI_EXT_SetSCACmd (uint16 scalnPPM)

This command sets the sleep clock accuracy (SCA) value of this device, in parts per million (PPM), from 0 to 500. For a master device, the value is converted to one of eight ordinal values representing a SCA range per [Specification of the Bluetooth System, Covered Core Package, Version: 4.1](#), which is used when a connection is created. For a slave device, the value is directly used. The system default value for a master and slave device is 50 ppm and 40 ppm, respectively.

Parameters

scalnPPM – This SCA of the device in PPM from 0 to 500.

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: This command is only allowed when the device is disconnected.
The SCA value of the device remains unaffected by an HCI Reset.

hciStatus_t HCI_EXT_SetSlaveLatencyOverrideCmd (uint8 control)

This command enables or disables the Slave Latency Override, letting you temporarily suspend Slave Latency even though it is active for the connection. When enabled, the device wakes up for every connection until Slave Latency Override is disabled again. The default value is Disable.

Parameters control – HCI_EXT_ENABLE_SL_OVERRIDE, HCI_EXT_DISABLE_SL_OVERRIDE

Corresponding Event HCI_VendorSpecificCommandCompleteEvent

NOTE: The function applies only to devices acting in the slave role. The function can be helpful when the slave application receives something that must be handled without delay. The function does not change the slave latency connection parameter; the device wakes up for each connection event.

hciStatus_t HCI_EXT_SetTxPowerCmd(uint8 txPower)

This command sets the RF transmitter output power. The default system value for this feature is 0 dBm.

Parameters txPower – transmit power of the device, one of the following corresponding events

Corresponding Events HCI_VendorSpecificCommandCompleteEvent:

- HCI_EXT_TX_POWER_MINUS_21_DBM
- HCI_EXT_TX_POWER_MINUS_18_DBM
- HCI_EXT_TX_POWER_MINUS_15_DBM
- HCI_EXT_TX_POWER_MINUS_12_DBM
- HCI_EXT_TX_POWER_MINUS_9_DBM
- HCI_EXT_TX_POWER_MINUS_6_DBM
- HCI_EXT_TX_POWER_MINUS_3_DBM
- HCI_EXT_TX_POWER_0_DBM
- HCI_EXT_TX_POWER_1_DBM
- HCI_EXT_TX_POWER_2_DBM
- HCI_EXT_TX_POWER_3_DBM
- HCI_EXT_TX_POWER_4_DBM
- HCI_EXT_TX_POWER_5_DBM

H.3 Host Error Codes

This section lists the various possible error codes generated by the host. If an HCI extension command that sent a command status with the SUCCESS error code before processing may find an error during execution then the error is reported in the normal completion command for the original command. The error code 0x00 means SUCCESS. The possible range of failure error codes is 0x01-0xFF. The following table lists an error code description for each failure error code.

| Value | Parameter Description |
|-------|---------------------------|
| 0x00 | SUCCESS |
| 0x01 | FAILURE |
| 0x02 | INVALIDPARAMETER |
| 0x03 | INVALID_TASK |
| 0x04 | MSG_BUFFER_NOT_AVAIL |
| 0x05 | INVALID_MSG_POINTER |
| 0x06 | INVALID_EVENT_ID |
| 0x07 | INVALID_INTERRUPT_ID |
| 0x08 | NO_TIMER_AVAIL |
| 0x09 | NV_ITEM_UNINIT |
| 0x0A | NV_OPER_FAILED |
| 0x0B | INVALID_MEM_SIZE |
| 0x0C | NV_BAD_ITEM_LEN |
| 0x10 | bleNotReady |
| 0x11 | bleAlreadyInRequestedMode |
| 0x12 | bleIncorrectMode |
| 0x13 | bleMemAllocError |
| 0x14 | bleNotConnected |
| 0x15 | bleNoResources |
| 0x16 | blePending |
| 0x17 | bleTimeout |
| 0x18 | bleInvalidRange |
| 0x19 | bleLinkEncrypted |
| 0x1A | bleProcedureComplete |
| 0x1B | bleInvalidMtuSize |
| 0x21 | bleUnexpectedRole |
| 0x30 | bleGAPUserCanceled |
| 0x31 | bleGAPConnNotAcceptable |
| 0x32 | bleGAPBondRejected |
| 0x40 | bleInvalidPDU |
| 0x41 | bleInsufficientAuthen |
| 0x42 | bleInsufficientEncrypt |
| 0x43 | bleInsufficientKeySize |
| 0xFF | INVALID_TASK_ID |

ICall API

I.1 Commands

The ICall commands that are useful from the application task are defined in [Section 4.2](#)

I.2 Error Codes

This section lists the error codes associated with ICall failures. The following values can be returned from any function defined in ICallBleAPI.c.

| Value | Error Name | Description |
|-------|-------------------------------|---|
| 0x04 | MSG_BUFFER_NOT_AVAIL | Allocation of ICall Message Failed |
| 0xFF | ICALL_ERRNO_INVALID_SERVICE | The service corresponding to a passed service id is not registered |
| 0xFE | ICALL_ERRNO_INVALID_FUNCTION | The function id is unknown to the registered handler of the service |
| 0xFD | ICALL_ERRNO_INVALID_PARAMETER | Invalid Parameter Value |
| 0xFC | ICALL_ERRNO_NO_RESOURCE | Not available entities, tasks, or other ICall resources |
| 0xFB | ICALL_ERRNO_UNKNOWN_THREAD | The task is not a registered task of the entity id is not a registered entity |
| 0xFA | ICALL_ERRNO_CORRUPT_MSG | Corrupt message error |
| 0xF9 | ICALL_ERRNO_OVERFLOW | Counter Overflow |
| 0xF8 | ICALL_ERRNO_UNDERFLOW | Counter Underflow |

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

| Changes from B Revision (January 2016) to C Revision | Page |
|---|-------------|
| • Updated to version 4.2 of Specification of the Bluetooth System. | 11 |
| • Updated Getting Started with Bluetooth LE Development section. | 13 |
| • Updated Project Zero on CCS Cloud image. | 14 |
| • Added Hardware and Software Architecture Overview section. | 20 |
| • Updated Single-Device and Simple Network Processor Configuration image. | 21 |
| • Updated Directory Structure section. | 22 |
| • Updated Projects section to Sample Applications section. | 24 |
| • Updated Bluetooth low energy SDK version from 2.1.0 to 2.2.0. | 25 |
| • Updated IAR EW ARM IDE version from 7.40.2 to 7.50.3. | 25 |
| • Updated Code Composer Studio IDE version from 6.1.0 to 6.1.2. | 25 |
| • Updated TI-RTOS version from 2_13_00_06 to 2_18_00_03. | 25 |
| • Updated XDC Tools version from 3_32_00_06_core to 3_32_00_06_core. | 25 |
| • Updated Sensor Controller Studio version from 1.0.1 to 1.2.1. | 25 |
| • Updated BTool PC Application version from 1.41.05 to 1.41.09. | 25 |
| • Updated SmartRF Flash Programmer 2 version from 1.6.2 to 1.7.3. | 25 |
| • Updated SmartRF Studio 7 version from 2.1.0 to 2.3.1. | 25 |
| • Changed IAR EW ARM version from 7.40.2 to 7.50.3. | 25 |
| • Updated Full Verbosity image. | 26 |
| • Updated Custom Argument Variables image. | 27 |
| • Updated IAR Workspace Pane image. | 28 |
| • Updated Compile and Download section. | 29 |
| • Updated Configure CCS section. | 29 |
| • Updated Installation Details images. | 30 |
| • Added Installing a Specific TI ARM Compiler section. | 31 |
| • Updated Import CSS Projects image. | 32 |
| • Updated Project Explorer Structure image. | 33 |
| • Updated Working With Hex Files section and example. | 34 |
| • Updated Defined Symbols Box image. | 35 |
| • Updated CCS Properties image. | 36 |
| • Updated Software Interrupts section. | 48 |
| • Updated Flash section. | 49 |
| • Updated Application and Stack Flash Boundary section. | 51 |
| • Removed Manually Modifying Flash Boundary section. | 51 |
| • Updated Using Simple NV for Flash Storage section. | 51 |
| • Updated Customer Configuration Area section. | 52 |
| • Updated linker files in Memory Management (RAM) section. | 53 |
| • Updated RAM Memory Map section. | 53 |
| • Updated Application and Stack RAM Boundary section. | 54 |
| • Removed Manually Modifying the RAM Boundary section. | 54 |
| • Updated Dynamic Memory Allocation section. | 54 |
| • Removed Configuration of RAM and Flash Boundary Using the Boundary Tool section. | 55 |
| • Added Configuration of RAM and Flash Boundary Using the Frontier Tool section. | 56 |
| • Updated Disabling Frontier Tool from Stack Project in IAR image. | 57 |
| • Updated Disabling Frontier Tool from Stack Project in CCS image. | 58 |
| • Updated code locations for simple_peripheral task. | 59 |
| • Added Events Signaled Using TI-RTOS Events Module section. | 69 |
| • Updated Peripheral Role section. | 76 |
| • Updated Central Role section. | 79 |

| | |
|--|-----|
| • Updated GATT Client and Server image. | 82 |
| • Updated GATT Services and Profile section. | 83 |
| • Added GAP GATT Service section. | 84 |
| • Added Generic Attribute Profile Service section. | 85 |
| • Added GATT Security section. | 102 |
| • Updated GAP Bond Manager and LE Secure Connections section. | 104 |
| • Added LE Privacy 1.2 section. | 117 |
| • Updated code in Enabling Auto Sync of White List section. | 119 |
| • Updated HCI section. | 126 |
| • Changed Maximum Default Number of Bluetooth low energy HCI PDUs from 27 to 5. | 133 |
| • Updated Configuring Bluetooth low energy Protocol Stack Features section. | 133 |
| • Updated Bluetooth low energy Protocol Stack Features table. | 133 |
| • Updated Adding a Driver section. | 135 |
| • Updated Board File section. | 136 |
| • Updated Available Drivers section. | 137 |
| • Added Using 32-kHz Crystal-Less Mode section. | 139 |
| • Added Determining the Auto Heap Size section. | 150 |
| • Added Loading RTOS in ROM Symbols section. | 154 |
| • Removed Using a Custom Exception Handler section. | 157 |
| • Added Using TI-RTOS and ROV to Parse Exceptions section. | 157 |
| • Removed Parsing the Exception Frame section. | 159 |
| • Added Assert Handling section. | 160 |
| • Added Board Type Symbol to Application Preprocessor Symbols table. | 162 |
| • Added BLE_NO_SECURITY Preprocessor Symbol to Stack Preprocessor Symbols table. | 164 |
| • Changed OSAL_SNV Preprocessor Symbol to OSAL_SNV=1. | 164 |
| • Added OSAL_MAX_NUM_PROXY_TASKS=2 Preprocessor Symbol to Stack Preprocessor Symbols table. | 164 |
| • Added EXT_HAL_ASSERT Preprocessor Symbol to Stack Preprocessor Symbols table. | 164 |
| • Updated Creating a Custom <i>Bluetooth</i> low energy Application section. | 165 |
| • Updated Optimizing Bluetooth low energy Stack Memory Usage section. | 166 |
| • Updated Additional Memory Configuration Options section. | 166 |
| • Added bStatus_t GAP_ConfigDeviceAddr(uint8 addrMode, uint8 *pStaticAddr) API command. | 183 |
| • Added bStatus_t GAP_ResolvePrivateAddr(uint8 *pIRK, uint8 *pAddr) API Command. | 184 |
| • Added uint8 GAP_NumActiveConnections(void) API Command. | 185 |
| • Updated Configurable Parameters table. | 185 |
| • Added void GAPRole_RegisterAppCBs(gapRolesParamUpdateCB_t *pParamUpdateCB) API Command. | 193 |
| • Updated Configurable Parameters table. | 193 |
| • Removed bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue) API Command. | 206 |
| • Added bStatus_t GAPBondMgr_ReadCentAddrResChar(uint16 connectionHandle) API Command. | 221 |
| • Added uint8 GAPBondMgr_SupportsEnhancedPriv(uint8 *pPeerIdAddr) API Command. | 222 |
| • Added bStatus_t GAPBondMgr_syncResolvingList(void) API Command. | 223 |
| • Updated Configurable Parameters table. | 224 |
| • Added API Function Map table. | 233 |
| • Added Vendor-Specific HCI Commands section. | 234 |
| • Updated Host Error Codes table. | 246 |

Revision History

| Changes from A Revision (November 2015) to B Revision | Page |
|--|-------------|
| • Edited and updated documents to TI standards. | 11 |

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

| | |
|------------------------------|--|
| Audio | www.ti.com/audio |
| Amplifiers | amplifier.ti.com |
| Data Converters | dataconverter.ti.com |
| DLP® Products | www.dlp.com |
| DSP | dsp.ti.com |
| Clocks and Timers | www.ti.com/clocks |
| Interface | interface.ti.com |
| Logic | logic.ti.com |
| Power Mgmt | power.ti.com |
| Microcontrollers | microcontroller.ti.com |
| RFID | www.ti-rfid.com |
| OMAP Applications Processors | www.ti.com/omap |
| Wireless Connectivity | www.ti.com/wirelessconnectivity |

Applications

| | |
|-------------------------------|--|
| Automotive and Transportation | www.ti.com/automotive |
| Communications and Telecom | www.ti.com/communications |
| Computers and Peripherals | www.ti.com/computers |
| Consumer Electronics | www.ti.com/consumer-apps |
| Energy and Lighting | www.ti.com/energy |
| Industrial | www.ti.com/industrial |
| Medical | www.ti.com/medical |
| Security | www.ti.com/security |
| Space, Avionics and Defense | www.ti.com/space-avionics-defense |
| Video and Imaging | www.ti.com/video |

TI E2E Community

e2e.ti.com