AN916: Blue Gecko Module WSTK BGAPI GPIO Demo Application Note

This document describes the structure and usage of the BGAPI protocol and BGLib sample implementation that comes with the Blue Gecko Software Development Kit (SDK).

The example project components are designed to be used with the Blue Gecko Module Wireless Starter Kit (with BGM111 module) and the free “Community” version of Microsoft Visual Studio, but the core implementation works with any Blue Gecko part and any ANSI C host platform. Further, since BGAPI itself is a simple binary protocol, it is possible to create or port similar libraries onto any platform capable of host-to-module communication over a UART interface.

For compiling the host application project, alternative toolchains may be used with small code modifications or Makefile creation. For client-side testing, other iOS apps or completely different BLE-capable devices may be used with similar results if you follow the same basic steps.

Note: This application note assumes as a prerequisite familiarity with concepts like BGAPI, BGLib, and the module's software architecture. Please refer to *QSG108: Blue Gecko Bluetooth® Smart Software Quick-Start Guide* to learn more about the basics.

APPLICATION NOTE REQUIREMENTS

- Blue Gecko Module Wireless Starter Kit hardware
- Microsoft Visual Studio Community Edition or similar
- iPhone 4S or newer running BLExplr app

1. Factory Default Configuration

Below are short descriptions of the default configurations in the Blue Gecko modules and development kits.

1.1 Blue Gecko *Bluetooth*® Smart Module Factory Configuration

Modules delivered from the factory will have the Bluetooth Smart software with the following configurations preinstalled:

- Software: Newest stable release of the Bluetooth Smart stack, `bgapi` example project from SDK.
- Behavior: Boot into idle mode, wait for commands from host (will not be visible in BLE scan).
- Host Interface:
  - BGAPI serial protocol over UART interface
  - UART baud rate: 115200
  - Hardware flow control: enabled
  - Data bits: 8
  - Parity: none
  - Stop bits: 1
- Firmware update interfaces:
  - DFU over UART enabled.
  - Segger J-link interface enabled.

1.2 Wireless Start Kit Factory Configuration

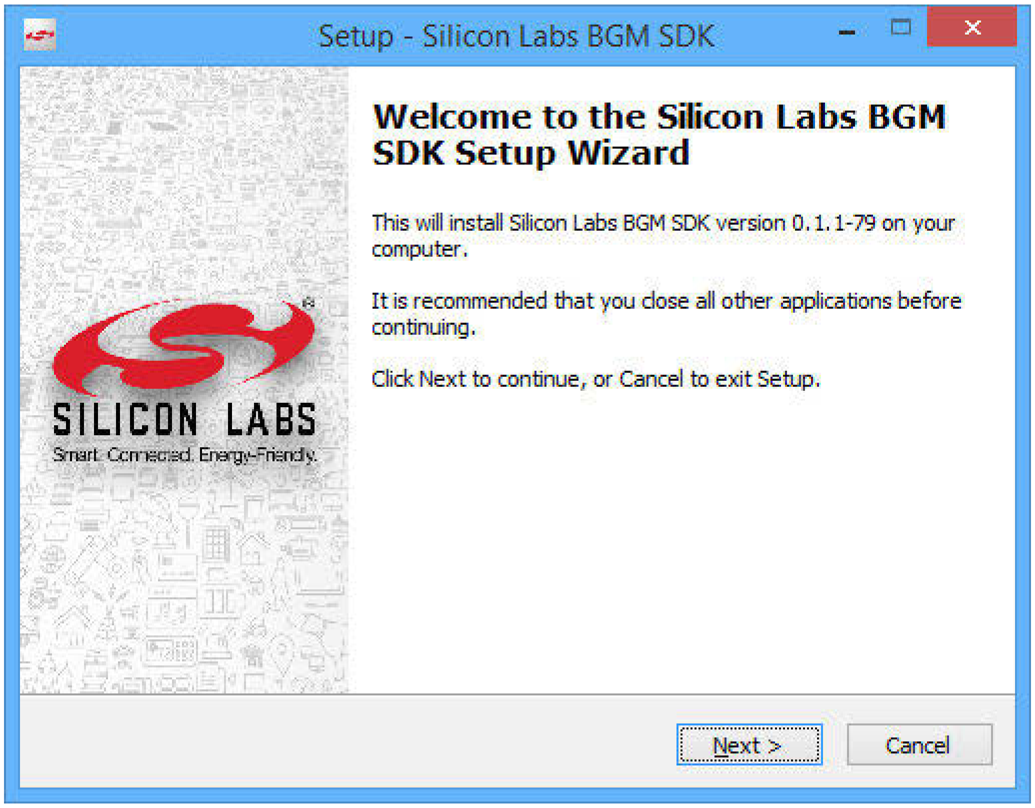
Blue Gecko Wireless Starter Kits delivered from the factory will have the following configurations preinstalled:

- Software: Newest stable release of the Bluetooth Smart stack, `bgm111demo` example project from SDK.
- Behavior: Boot into advertising mode automatically (will be visible in BLE scan).
- Host interface:
  - None.
- Firmware update interfaces:
  - Segger J-link interface enabled.

2. Getting Started with the Blue Gecko *Bluetooth*® Smart Software

2.1 Installing the *Bluetooth* Smart SDK

1. Go to the Getting Started with Bluetooth page.
2. Download the Blue Gecko Bluetooth Smart Software.
3. Run the installer executable.
4. Follow the on-screen instructions to install the SDK.


Setup - Silicon Labs BGM SDK

Welcome to the Silicon Labs BGM SDK Setup Wizard

This will install Silicon Labs BGM SDK version 0.1.1-79 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Figure 2.1. Installing the SDK

2.2 Folder Structure

The SDK creates the following folders in the installation directory:

bin:	This folder includes all the binary applications needed by the SDK.
doc:	Contains an HTML version of the BGAPI, BGScript, and BGLib API documentation.
examples:	This folder includes the BGScript demo applications, profile toolkit examples, and the Bluetooth module configuration examples, which all generate a firmware for the Bluetooth module if run through the BGBuild compiler.
host_example:	This folder contains example C source code demonstrating BGLib usage, including the project discussed in detail in this application note.
fw:	This folder contains the actual Bluetooth Smart stack firmware binaries.

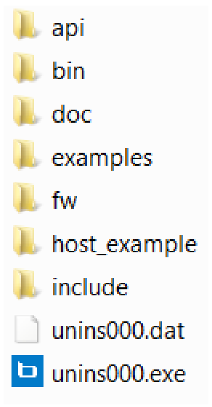


Figure 2.2. SDK Folders

2.3 Included Tools

The following tools are installed by the SDK:

BGTool:	A graphical UI tool that can be used to control the Bluetooth Smart module over UART/RS232 using the BGAPI serial protocol. This tool is useful for quickly trying the features and capabilities of the Blue Gecko Bluetooth Smart Software without writing any software first.
BGBuild:	A compiler used to build firmware images for the Bluetooth Smart modules. The BGBuild tool can also be used to compile and flash firmware onto the modules from the Windows command prompt.
eACommander:	A flash tool for installing firmware into the Blue Gecko modules over the debug interface. This tool works regardless of what firmware was previously on the module, and can be helpful in cases where previous firmware configuration has made the UART DFU interface inaccessible.

3. Walkthrough of the Blue Gecko WSTK BGAPI GPIO Project

This section is a detailed description of all the parts of a general BGAPI/BGLib project. The WSTK GPIO project is specifically the focus of this application note. The purpose of the section is to help you understand how the different parts work together, how to navigate and modify the code structure, and how to build your own BGAPI/BGLib applications with the Bluetooth Smart SDK.

The `wstk_bgapi_gpio` project discussed here implements a Bluetooth Smart peripheral device, which allows simple read/write GPIO interaction with a remote client device. The project has the following features:

- Reset and start connectable advertising during BGAPI initialization routine.
- Writable characteristic to affect LED0 state on WSTK board.
- Readable characteristic to detect PB1 button state on WSTK board.
- On-demand GPIO read accomplished with “user”-type characteristic.
- Repeating soft timer to demonstrate simple polling of button state.
- Button state changes pushed to client using GATT indications.
- Auto-resume advertising upon disconnection.

Don't worry if some of these concepts are unfamiliar to you. Each concept will be discussed in more detail in sections below. The behavior of this application is arbitrary in many ways, but is designed to provide a reference for many functions common to Bluetooth Smart peripheral applications.

3.1 BGAPI Project Components

Unlike BGScript-based projects which are built into single, standalone firmware images, projects which use BGAPI have two separate parts:

1. Module Firmware Project:
   - Hardware configuration (UART with BGAPI enabled).
   - GATT structure definition.
   - Compiled and flashed with Bluetooth Smart SDK tools.
2. Host Application Project:
   - Application logic.
   - BGAPI parser/generator (BGLib code).
   - Host-specific UART RX/TX functions.
   - Compiled with host platform toolchain (Visual Studio, gcc, etc.).
   - Flashed if necessary with host MCU flash tools.

Either half alone cannot function completely without the other in place. So, it is important to understand the function of each half and where it must be implemented. The two components for the demo project in this application note come with the Blue Gecko Bluetooth Smart SDK, and can be found in these two subfolders where the SDK is installed:

- `/example/wstk_bgapi_gpio` – Module firmware project.
- `/host_example/wstk_bgapi_gpio` – Host application project with Visual Studio solution file.

The figure below illustrates the demo application architecture.

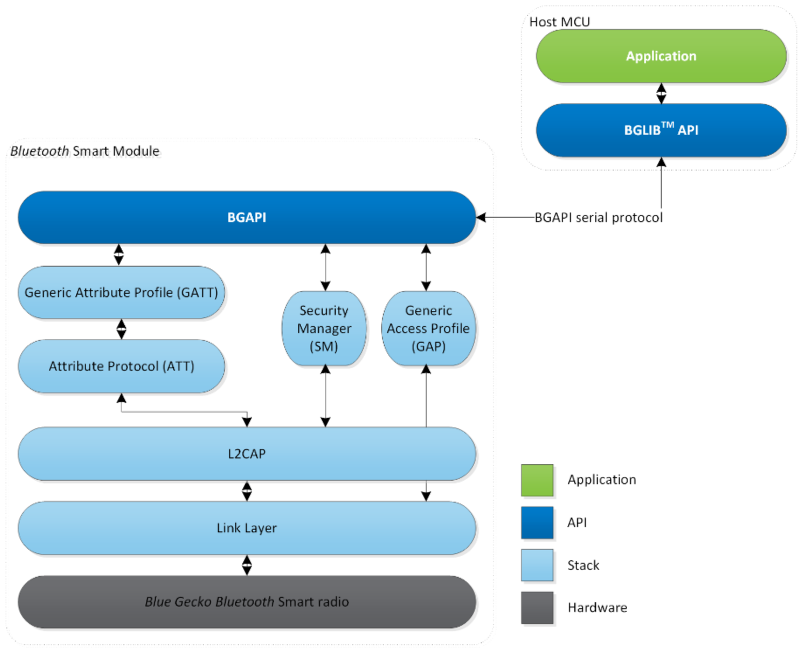


Figure 3.1. BGAPI/BGLib Project Application Architecture

3.2 Module Firmware Project

Building a Bluetooth Smart project always starts by making a project file, which is an XML file that defines the resources used in the project. In the case of this project, the entire definition is bundled into the single `wstk_bgapi_gpio.bgproj` file:

Project File

```
<?xmlversion="1.0" encoding="UTF-8" ?>
<projectdevice="bgm111">

   <hardware>
      <!-- UART enabled @115200bps -->
      <uart index="1" baud="115200" flowcontrol="false" bgapi="true"/>

      <!-- WSTK UART pins -->
      <gpio port="A" pin="5" mode="pushpull" out="1" />
      <gpio port="A" pin="3" mode="pushpull" out="0" />

      <!-- LED0 (same pin as Button0) -->
      <gpio port="F" pin="6" mode="pushpull" out="1" />

      <!-- Button1 (same pin as LED1) -->
      <gpio port="F" pin="7" mode="input" out="1" />
   </hardware>

   <gatt out="gatt_db.c" header="gatt_db.h" prefix="gattdb_">
      <service uuid="1800" id="generic_access">
         <characteristic uuid="2a00" id="device_name">
            <properties read="true" const="false" />
            <value variable_length="true" length="20" />
         </characteristic>
         <characteristic uuid="2a01">
            <properties read="true" const="true" />
            <value type="hex">4003</value>
         </characteristic>
      </service>

      <service uuid="180A" id="device_information">
         <characteristic uuid="2A29">
            <properties read="true" const="true" />
            <value>Silicon Labs</value>
         </characteristic>
      </service>

      <service uuid="ab65f91e-82c2-4b93-a535-26dcdfd2c83d" id="control" advertise="true">
         <characteristic uuid="5e2c5575-c950-4b5f-8e7a-ee3e340395e7" id="gpio_control">
            <properties read="true" write="true" indicate="true" />
            <value length="1" type="user" />
         </characteristic>
      </service>
   </gatt>

   <image out="wstk_bgapi_gpio.bin" />
</project>
```

It is common for project definitions to be split into multiple files, such as one for the GATT structure, one for the hardware configuration, and so on. However, this is not required. For simpler projects like this, combining may be convenient.

Table 3.1. Project File Explanation

`<project device=" bgm111" … />`	This tag starts the main project definition. The device attribute is used to define which Bluetooth module the project will be compiled for.
`<gatt … />`	The `<gatt>` tag is used to define the local GATT structure on this device. The out, header, and prefix attributes specify names of auto-generated files which can help with BGAPI host projects.
`<hardware … />`	The `<hardware>` tag defines the hardware configuration for interfaces like UART, SPI or I2C, and GPIO.
`<image out =" wstk_bgapi_gpio.bin " />`	The `<image>` tag defines the name of the BGBuild compiler output file. The generated .bin file contains the Bluetooth Smart stack, the GATT database, the hardware configuration and the BGScript code (optional).

Note: The full syntax of the project configuration file and more examples can be found in the *UG119: Blue Gecko Bluetooth® Smart Modules Configuration User's Guide*.

3.2.1 Hardware Configuration

The hardware configuration in the module-side component of a BGAPI/BGLib project is critical, since it must define the UART peripheral interface used for BGAPI control from the external host. Without this, it is not possible to use BGAPI externally. The hardware definition from the `wstk_bgapi_gpio` project is reproduced below:

Hardware Configuration

```
<hardware>
   <!-- UART enabled @115200bps -->
   <uart index="1" baud="115200" flowcontrol="false" bgapi="true"/>

   <!-- WSTK UART pins -->
   <gpio port="A" pin="5" mode="pushpull" out="1" />
   <gpio port="A" pin="3" mode="pushpull" out="0" />

   <!-- LED0 (same pin as Button0) -->
   <gpio port="F" pin="6" mode="pushpull" out="1" />

   <!-- Button1 (same pin as LED1) -->
   <gpio port="F" pin="7" mode="input" out="1" />
</hardware>
```

Table 3.2. Hardware Configuration Explanation

`<uart index="1" baud=" 115200" flowcontrol="false" bgapi=" true" />`	The `<uart>` tag is used to enable UART peripherals and to configure the UART port settings such as baud rate and flow control. This tag also controls whether the given UART port is to expose the BGAPI layer as a host control point This project configures the UART for 115200 baud, 8/N/1, no flow control, and BGAPI enabled. These parameters must be kept in mind for the host application design.
`<gpio port="..." pin="..." mode="..." out="..." />`	The `<gpio>` tag is used to configure the default states of GPIO pins on the module. This project configures two pins for UART behavior and two for LED/button interaction. The two UART-related pins are those wired on the WSTK board for TXD (PA5) and RTS (PA3) respectively, and the settings here simply configure "safe" idle settings—TXD de-asserted high and RTS asserted low. The `<uart>` tag takes care of the core UART peripheral assignment.

Note: The full syntax of the project configuration file and more examples can be found in the *UG119: Blue Gecko Bluetooth® Smart Modules Configuration User's Guide*.

3.2.2 GATT Services Configuration

The Bluetooth GATT services made available on the device are defined using the XML-based language shown below, compiled with the BGBuildcompiler as part of the firmware. The GATT database in this application includes two officially adopted services (Generic Access and Device Information), and one custom service for GPIO interfacing. Profile behavior and specifications are introduced later in this chapter.

More detailed information about official adopted profiles can be found on the Bluetooth SIG web page: https://developer.bluetooth.org/gatt/profiles/Pages/ProfilesHome.aspx.

First, take a look at the Generic Access service and characteristics:

Generic Access Protocol (GAP) Service

```
<serviceuuid="1800" id="generic_access">
   <characteristic uuid="2a00" id="device_name">
      <properties read="true" const="false" />
      <value variable_length="true" length="20" />
   </characteristic>
   <characteristic uuid="2a01">
      <properties read="true" const="true" />
      <value type="hex">4003</value>
   </characteristic>
</service>
```

Note: The full syntax of the GATT configuration file and more examples can be found in the *UG118: Blue Gecko Bluetooth® Smart Profile Toolkit Developer User's Guide*.

Every Bluetooth Smart peripheral implementing a GATT server contains this service, which includes the device name and appearance.

The Figure 3.7 Visual Studio Project View on page 18 shows part of the GATT database used in the demo application and how it looks in the Profile Toolkit XML format. An explanation of the syntax can be found below.

Table 3.3. GAP Service Explanation

`<service uuid=" 1800" id=" generic_access" />`	The `<service>` tag is used to define a start of a GATT service. The uuid attribute defines the 16-bit or 128-but UUID used by the service. In this case, the UUID 1800 refers to the GAP service defined by the Bluetooth SIG and details of which can be found from Bluetooth developer site here. The `id` attribute is arbitrary and included only for quick reference.

`<characteristic uuid=" 2a00">`
`   <properties read=" true" const=" true" />`
`   <value>Blue Gecko BGM111</ value>`
`</characteristic>`
	The `<characteristics>` tag starts the definition of a characteristic, so the actual data exposed by the device. Again, the characteristic UUID must be defined and every characteristic needs to have a unique 16-bit or 128-bit UUID. In this case 2a00 refers to device name characteristic, which can be found from here.
	
	The `<properties>` tag defines how the characteristics can be accessed by a remote Bluetooth device and what securities need to be in place to access the characteristic. An optional `const` parameter can also be used to define the value as constant and non-editable.
	
	In the case of constant values, the actual value of the characteristic can be defined inside the `<value>` tags. For non-const values, *all data must be initialized at runtime.*

`<characteristic uuid=" 2a01">`
`   <properties read=" true" const="true" />`
`   <value type=" hex">0768</ value>`
`</characteristic>`
	The second characteristic (appearance) is defined in the same manner and has the same properties as the device name. The only difference is that the characteristic is in hex format instead of UTF-8 as defined with `type="hex"`. If omitted, the default value type is hex.

Next, take a look at the Device Information service and single characteristic that it contains:

Device Information Service

```
<service uuid="180A" id="device_information">
   <characteristic uuid="2A29">
      <properties read="true" const="true"/>
      <value>Silicon Labs</value>
   </characteristic>
</ service >
```

This service is optional for GATT servers, and if included may contain many optional characteristics, such as manufacturer name, firmware version, serial number, and others. In this example, we include only the manufacturer name. For more information on what is available, visit the Device information service page on the Bluetooth SIG website.

Finally, look at the Control service and characteristic that it contains:

Control Service

```
<service uuid="ab65f91e-82c2-4b93-a535-26dcdfd2c83d" id="control"advertise="true">
   <characteristic uuid="5e2c5575-c950-4b5f-8e7a-ee3e340395e7" id="gpio_control">
      <properties read="true" write="true" indicate="true"/>
      <value length="1" type="user"/>
   </characteristic>
</service>
```

This is a custom service with a custom characteristic, included in this project to specifically provide a way to read and control certain GPIO states. In contrast to the above two services, notice that the UUID values used here are 128 bits long. This is required for any custom GATT elements, i.e. those not adopted and approved by the Bluetooth SIG. The UUIDs may be created using any tool like http://www.guidgenerator.com, and do not need to be confirmed or filed with the Bluetooth SIG.

Important points to note about the custom control service and its characteristic are listed below:

- The `<service>` tag has `advertise="true"` defined. This will cause the stack to automatically incorporate the service's UUID into the advertisement packet (assuming you use the default advertising mode, which we do here). Including the service UUID in the ad packet makes it easy for scanning BLE devices to identify show only advertising peripherals which match a UUID-based filter, rather than every BLE peripheral in the vicinity.
- The characteristic's `<properties>` tag enables the read, write, and indicate data transfer operations.
  - Reading allows a client to request the characteristic value on demand.
  - Writing allows a client to write a new value to the characteristic.
  - Indicating allows the server to push an updated value to the client, if the client has subscribed to updates.
- The characteristic's `<value>` tag sets a fixed length of one byte and sets the value type to `user`. Defining a user-type characteristic value changes the way that the stack handles data storage and GATT read/write management for the characteristic. Normally, the stack will automatically allocate space in RAM to store the value, and it will access or update this for you when a client reads or writes data. However, with a user-type value, none of this is done for you. Instead, the API layer will simply inform you whenever a GATT client requests a read or write operation, and it is up to your application logic to process the request and respond appropriately (with a "0" success code or a non-zero error code). This results in more work for the application logic, but it also gives you greater control over the data flow. The sections below provide further detail on how these features are used in the WSTK GPIO demo project.

3.2.3 Output Firmware Image Configuration

The project definition file ends with one final tag, used to specify the name of the binary output file resulting from the compilation process.

```
<image out="wstk_bgapi_gpio.bin" />
```

The file produced here can be flashed into a Blue Gecko module using the eACommander and debug interface, or an available DFU interface and command-line "bgbuild" or GUI-based BGTool application.

This concludes the discussion of the module firmware project definition.

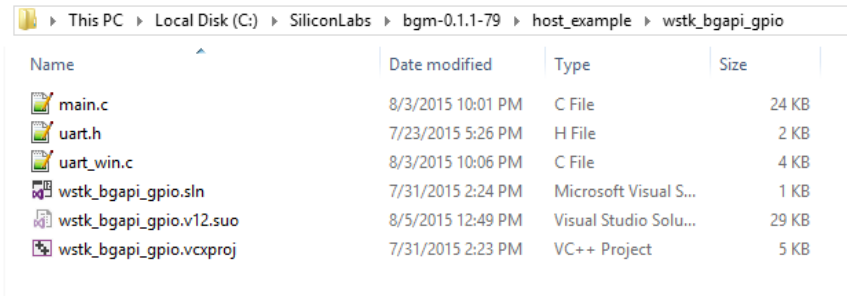
3.3 Host Application Project

The second major component to any BGAPI/BGLib project is the portion which runs on the host platform outside of the module, and which communicates with the module using the BGAPI protocol over the UART interface.

3.3.1 Project Source File Overview

The host application project that comes with the SDK is written in C and specifically prepared for the Microsoft Visual Studio Community Edition environment. The code can be ported or in some cases directly compiled with other toolchains, but this document focuses on the Visual Studio project.

The host-side source files for this project are found in the **/host_example/wstk_bgapi_gpio** subfolder where you have installed the Blue Gecko Bluetooth Smart SDK, and comprise only a few source and header files along with the Visual Studio project definition:


This PC ▸ Local Disk (C:) ▸ SiliconLabs ▸ bgm-0.1.1-79 ▸ host_example ▸ wstk_bgapi_gpio

Name	Date modified	Type	Size
main.c	8/3/2015 10:01 PM	C File	24 KB
uart.h	7/23/2015 5:26 PM	H File	2 KB
uart_win.c	8/3/2015 10:06 PM	C File	4 KB
wstk_bgapi_gpio.sln	7/31/2015 2:24 PM	Microsoft Visual S...	1 KB
wstk_bgapi_gpio.v12.suo	8/5/2015 12:49 PM	Visual Studio Solu...	29 KB
wstk_bgapi_gpio.vcxproj	7/31/2015 2:23 PM	VC++ Project	5 KB

Figure 3.2. List of Visual Studio Project Files

Below is a quick summary of the relevant files:

Table 3.4. Visual Studio Project File Explanation

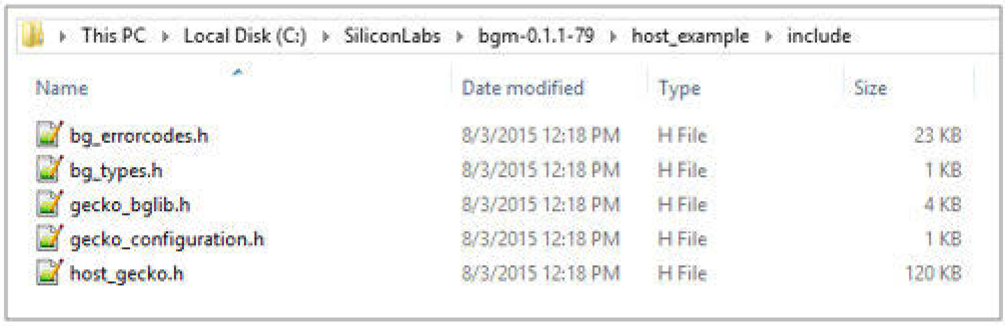
main.c	Main application logic to control the Blue Gecko module, and BGLib initialization and set-up code.
uart.h	Function declarations for simple UART port initialization and RX/TX data transfers common to most platforms and compatible with BGLib methods and callbacks.
uart_win.c	Implementations for the functions defined in **uart.h** which are appropriate for a Windows environment.
wstk_bgapi_gpio.sln **wstk_bgapi_gpio.v12.suo** **wstk_bgapi_gpio.vcxproj**	Visual Studio project and solution definition files. If you have Visual Studio 2013 or newer version installed, you should be able to open the solution (**.sln**) file directly and start working with the project.

3.3.2 BGLib Support File Overview

While the files in the previous section contain all of the application logic, the actual BGLib implementation code which contains the BGAPI parser and packet generation functions is found elsewhere, in other subfolders under the **/host_example** location from the SDK.

The SDK's specific arrangement of files is one possible way the BGAPI protocol can be used, but it is also possible to create your own library code which implements the protocol correctly with a different code architecture. The only requirement here is that the chosen implementation must be able to create BGAPI command packets correctly and send them to the module over UART, and similarly be able to receive BGAPI response and event packets over UART and process them into whatever function calls are needed to trigger the desired application behavior.

The other support files for BGLib-based projects are found in the **/host_example/include** folder:

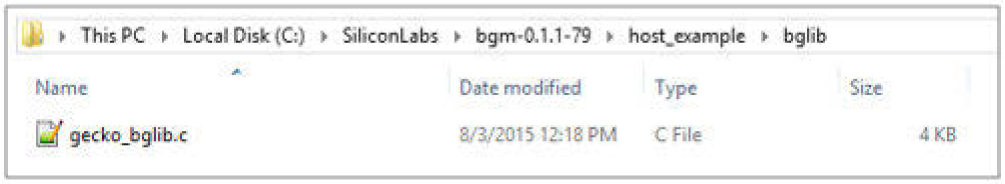


This PC › Local Disk (C:) › SiliconLabs › bgm-0.1.1-79 › host_example › include

Name	Date modified	Type	Size
bg_errorcodes.h	8/3/2015 12:18 PM	H File	23 KB
bg_types.h	8/3/2015 12:18 PM	H File	1 KB
gecko_bglib.h	8/3/2015 12:18 PM	H File	4 KB
gecko_configuration.h	8/3/2015 12:18 PM	H File	1 KB
host_gecko.h	8/3/2015 12:18 PM	H File	120 KB

Figure 3.3. List of Supporting BGLib Include Files

And in the **/host_example/bglib** folder:



This PC › Local Disk (C:) › SiliconLabs › bgm-0.1.1-79 › host_example › bglib

Name	Date modified	Type	Size
gecko_bglib.c	8/3/2015 12:18 PM	C File	4 KB

Figure 3.4. List of Supporting BGLib Source Files

The header files contain primarily **#define**'d compiler macros and named constants which correspond to all of the various API methods and enumerations that you may need to use. The gecko_bglib.h file also contains function declarations for the basic packet reception, processing, and transmission functions.The **gecko_bglib.c** file contains the implementation of the packet management functions. All functions defined here use only ANSI C code, to help ensure maximum cross-compatibility on different platforms.

The **gecko_bglib.c** file contains the implementation of the packet management functions. All functions defined here use only ANSI C code, to help ensure maximum cross-compatibility on different platforms.

Note: With structure packing, the SDK's provided BGLib implementation makes heavy use of direct mapping of packet payload structures onto contiguous blocks of memory, to avoid additional parsing and RAM usage. This is accomplished with the **PACKSTRUCT** macro used extensively in the BGLib header files. It is important to ensure than any ported version of BGLib also correctly packs structures together (no padding on multi-byte struct member variables) in order to achieve the correct operation.

Note: With byte order, the BGAPI protocol uses little-endian byte ordering for all multi-byte integer values, which means directly mapped structures will only work if the host platform also uses little-endian byte ordering. This covers most common platforms today, but some big-endian platforms exist and are actively used today (Motorola 6800 and 68k, AVR32, and others). Platforms which require native big-endian byte ordering cannot use the SDK's included BGLib implementation.

3.3.3 Functional Application Overview

Before we dig into the host-side code in **main.c**, let's look at what the application is designed to do, and how this is accomplished at a high level. Leaving aside the specifics of how BGAPI communication is handled behind the scenes, the code implements the following behavior:

1. On start-up, the application opens the serial interface and sends the "`system_reset(0)`" API command to reboot the module into a known state. The BGLib implementation translates this into the 5-byte API packet as documented in the API reference material: `[ 20 01 01 01 00 ]`. All other activity is triggered by incoming API events, so this first one is key. If the module does not reset itself as a result of this command for some reason, you can manually reset it by pressing the "RESET" button on the WSTK main board. When a reset occurs, the stack on the module will generate the `system_boot` event, which will be sent out the module's UART port for the host to receive and process.

2. When the `system_boot` event occurs indicating power-on or reset, the application applies a friendly device name to the appropriate local GATT characteristic, then starts connectable advertising with the "`gap_set_mode(2, 2)`" API command.

3. When the `le_connection_opened` event occurs upon a new connection from a remote BLE device, the application starts a repeating soft timer (running on the module) every 50 ms which will be used to trigger GPIO state polling.

4. When the `le_connection_closed` event occurs indicating the intentional or unintentional closure of the active BLE connection, the application stops the repeating timer and resumes connectable advertising.

5. When the `gatt_server_characteristic_status` event occurs indicating the remote GATT client has changed subscription status for "indicate" operations on the GPIO control characteristic characteristic, the application pushes the immediate PF7 pin (Button 1) logic state to the client **if** the client has just enabled indications.

6. When the `gatt_server_user_read_request` event occurs indicating the remote GATT client has tried to read a user-type value (from the GPIO control characteristic), the application reads the immediate PF7 pin logic state and sends the value back in the read response using the "`gatt_server_send_user_read_response(…)`" API command. This demonstrates an easy way to access readable GPIO logic state information remotely.

7. When the `gatt_server_user_write_request` event occurs indicating the remote GATT client has tried to write new data to the GPIO control characteristic's user-type value, the application validates the data and then sets the PF6 pin (LED0) low or high based on whether the value written is non-zero or not. This demonstrates an easy way to affect writable GPIO logic states remotely.

8. When the `hardware_soft_timer` event occurs indicating the active soft timer has "ticked" again, the application reads the immediate PF7 pin logic state. If the value has changed *and* a remote client is connected and has subscribed to indication updates on the GPIO characteristic (see #5 above), then the new value is pushed to the client using the "`gatt_server_send_characteristic_notification(…)`" API command.

   This is a polling implementation of GPIO state changes which might alternatively be accomplished using interrupts. However, to demonstrate the soft timer behavior as well, the polling method is used instead.

All of the API methods and parameters mentioned above and used in the code are described in detail in the Blue Gecko Bluetooth Smart API documentation.

3.3.4 BGLib Initialization

The **main.c** file contains two important macros which are used to set up and initialize the BGLib functionality provided in the SDK's implementation. The first one defines the function prototypes and buffers and state-tracking variables used by the parser and generator:

```
BGLIB_DEFINE();
```

The second one assigns two platform-specific functions to BGLib's internal function pointers for UART input and output communication:

```
BGLIB_INITIALIZE(on_message_send, uart_rx);
```

The "`on_message_send`" and "`uart_rx`" functions are implemented in **main.c** and **uart_win.c**, respectively, and control how data moves between the host and the module over UART. These or similar functions are necessary in every instance where BGLib is used. The parameter list for each function is simple:

```
static void  on_message_send( uint16 msg_len ,  uint8 *  msg_data ) { ... }
int  uart_rx( uint16_t data_length ,  uint8_t *  data )  { ... }
```

The first parameter is an unsigned 16-bit "length" value, and the second is a byte array pointer which should contain at least `<length>` bytes of data. For the output function, the BGLib code expects that the provided data will be sent out the UART port to the module. For the input function, the BGLib code expects that the requested number of bytes will be read from UART and stored in the provided buffer. If the receive function encounters an error, it should return `-1`.

The `BGLIB_DEFINE` and `BGLIB_INITIALIZE` macros are defined along with related macros in the **gecko_bglib.h** file. You must ensure that both of these macros are used in your application code to prepare the environment correctly. Without them, you will encounter many compile errors.

3.3.5 Understanding Basic Packet Structures

Every API packet regardless of type follows the same structure, as defined in the API reference material:

- Header (4 bytes)
  - Technology type and high-length byte
  - Low-length byte
  - Command class byte
  - Command ID byte
- Payload (0 or more bytes)
  - …depends on packet type

Within BGLib, every incoming packet (responses and events) is referenced using the `gecko_cmd_packet` struct pointer to allow access to the relevant data. This structure contains a **header** member (`uint32` mapped across all 4 header bytes), and a **data** member, which itself is a union of every possible payload structure. Your code can access data using the correct payload type structure and its members. For example, if you have an `evt` pointer which has detected the `system_boot` event, you could print the `build` parameter value this way:

```
printf("Build: %d", evt -> data.evt_system_boot.build);
```

Or, if you have an `rsp` pointer which is the returned response packet from a call to the `le_gap_set_mode` command, you could print the `result` parameter this way:

```
printf("Result: %04X", rsp -> data.rsp_le_gap_set_mode.result);
```

The basic format to remember is this:

```
obj -> data.packet_name.member_name
```

3.3.6 Event Detection and Handling

The `main()` function within **main.c** contains all of the application logic which drives this demo. If you are familiar with the way BGScript code is arranged, you might notice that each `case` statement within the `switch` block corresponds to one single API event in exactly the same way that BGScript code uses `event` handlers.

```
// ==========================================================================================
// This infinite loop is similar to the BGScript interpreter environment, and each "case"
// represents one of the event handlers that you would define using BGScript code. You can
// restructure this code to use your own function calls instead of placing all of the event
// handler logic right inside each "case" block.
// ==========================================================================================
while  (1)
{
      // blocking wait for event API packet
      evt = gecko_wait_event();

      // if a non-blocking implementation is needed, use gecko_peek_event() instead
      // (will return NULL instead of packet struct pointer if no event is ready)
      switch  ( BGLIB_MSG_ID (evt -> header))
      {
          // SYSTEM BOOT (power-on/reset)
          case gecko_evt_system_boot_id :
                ...

          // LE CONNECTION OPENED (remote device connected)
          case gecko_evt_le_connection_opened_id :
                ...

          // LE CONNECTION CLOSED (remote device disconnected)
         case gecko_evt_le_connection_closed_id :
                ...

         // GATT SERVER CHARACTERISTIC STATUS (remote GATT client changed subscription status)
         case gecko_evt_gatt_server_characteristic_status_id :
                ...
```

This kind of program flow keeps all relevant BLE module interaction code in the same place, and also makes it easy to port from on-module BGScript directly to off-module BGAPI/BGLib, in case your development process evolves this way. However, while this example also puts the full case handling code inline inside the `switch` block, it is perfectly acceptable to break that code out into user-defined function calls, or whatever is preferable to enhance code readability and organization.

3.3.7 Sending Commands Processing Responses

In addition to catching and handling API events, it is equally important to be able to send commands and receive and process responses. BGLib also makes this easy with built-in commands which each returns a pointer to the response packet structure.

Consider the code below in which the application starts advertisements inside the `system_boot` event handler case:

```
// start advertisements after boot/reset
printf( "--> Starting advertisements:\n\tgecko_cmd_le_gap_set_mode(2, 2)\n");
rsp = ( struct gecko_cmd_packet *)gecko_cmd_le_gap_set_mode(
        le_gap_general_discoverable,
        le_gap_undirected_connectable);
printf("<-- Received response:\n\tgecko_rsp_gap_set_mode(0x%04X)\n",
        rsp->data.rsp_le_gap_set_mode.result);
printf("\n--- AWAITING CONNECTION FROM BLE MASTER\n\n");
break;
```

Note specifically the `gecko_cmd_le_gap_set_mode` call and how its return value is stored in the `rsp` variable. This variable is defined earlier as a packet pointer, along with `evt` also used in the **main.c** file:

```
struct gecko_cmd_packet *evt, *rsp;
```

Every `gecko_cmd_...` function returns a pointer to the corresponding response packet. As mentioned in Section 3.3.5 Understanding Basic Packet Structures above, you can access this data by using the correct packet name and dotted structure member variable.

The application code in **main.c** uses the `evt` pointer to track the main event that triggered each case, and one `rsp` pointer reused each time a new command is sent. This arrangement is typical enough, though not strictly required.

Note: You can use as many packet pointers as you want, but since they are pointers to byte arrays elsewhere in memory, make sure you do not expect them to retain information any longer than the BGAPI protocol data flow allows. In other words, you should assume that any incoming event or response data is not permanent, and may be cleared or overwritten by new incoming data as soon as the next call to `gecko_wait_event()` or its counterparts occurs.

Note: With memory persistence, if you need to retain any specific packet data for longer periods of time, you should make sure that you copy the relevant values into other data structures.

3.4 Compiling the Module Firmware Project with BGBuild

The firmware application can be compiled and flashed into the module with the BGBuild compiler using command prompt. You can also use BGBuild only to compile the project, then use eACommander or BGTool to flash the binary firmware image in a separate step. Flashing with eACommander is shown in the next section.

To start, open a Command Prompt window and change directory to the “**wstk_bgapi_gpio**” subfolder in the /examples subfolder where the Blue Gecko SDK is installed. Then, run "**..\..\bin\bgbuild.exe wstk_bgapi_gpio.bgproj**” as shown in the screenshot below.

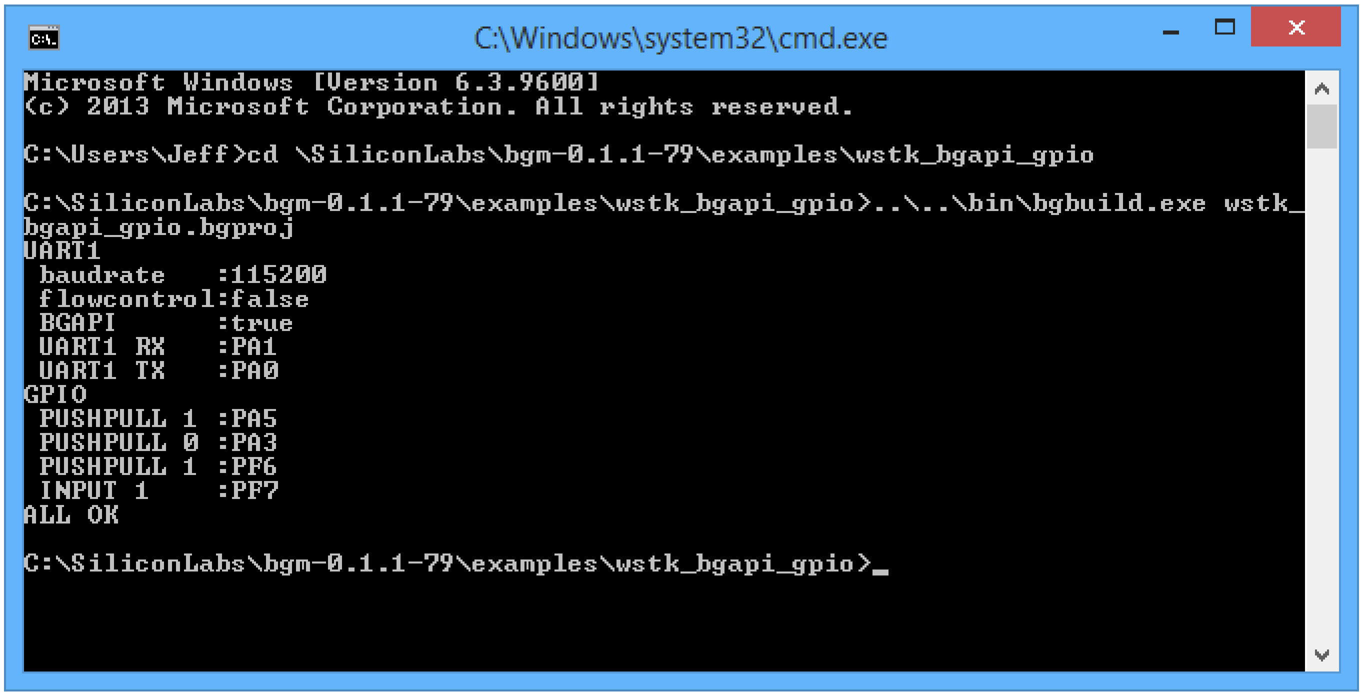
The syntax for the compiler is shown here: `bgbuild.exe [options] <input>s`

Options:

`-?, -h, --help`	Displays this help.
`-v, --version`	Displays version information.
`-g, --gattonly`	Only create GATT c-file.
`-r, --root <buildtools>`	Override default build tools location.
`-s, --scompiler <scriptcompiler>`	Path to script compiler.
`-f, --flash`	Flash resulting image to device (using Segger J-Link interface).

Arguments:

`input`	Project file to build.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Jeff>cd \SiliconLabs\bgm-0.1.1-79\examples\wstk_bgapi_gpio

C:\SiliconLabs\bgm-0.1.1-79\examples\wstk_bgapi_gpio>..\..\bin\bgbuild.exe wstk_
bgapi_gpio.bgproj
UART1
 baudrate    :115200
 flowcontrol:false
 BGAPI       :true
 UART1 RX    :PA1
 UART1 TX    :PA0
GPIO
 PUSHPULL 1  :PA5
 PUSHPULL 0  :PA3
 PUSHPULL 1  :PF6
 INPUT 1     :PF7
ALL OK

C:\SiliconLabs\bgm-0.1.1-79\examples\wstk_bgapi_gpio>_
```

Figure 3.5. Compiling the Demo Application

Note: After a firmware update, you should physically reset the Bluetooth Smart hardware by pressing the RESET button on the WSTK main board to ensure that the new firmware is running.

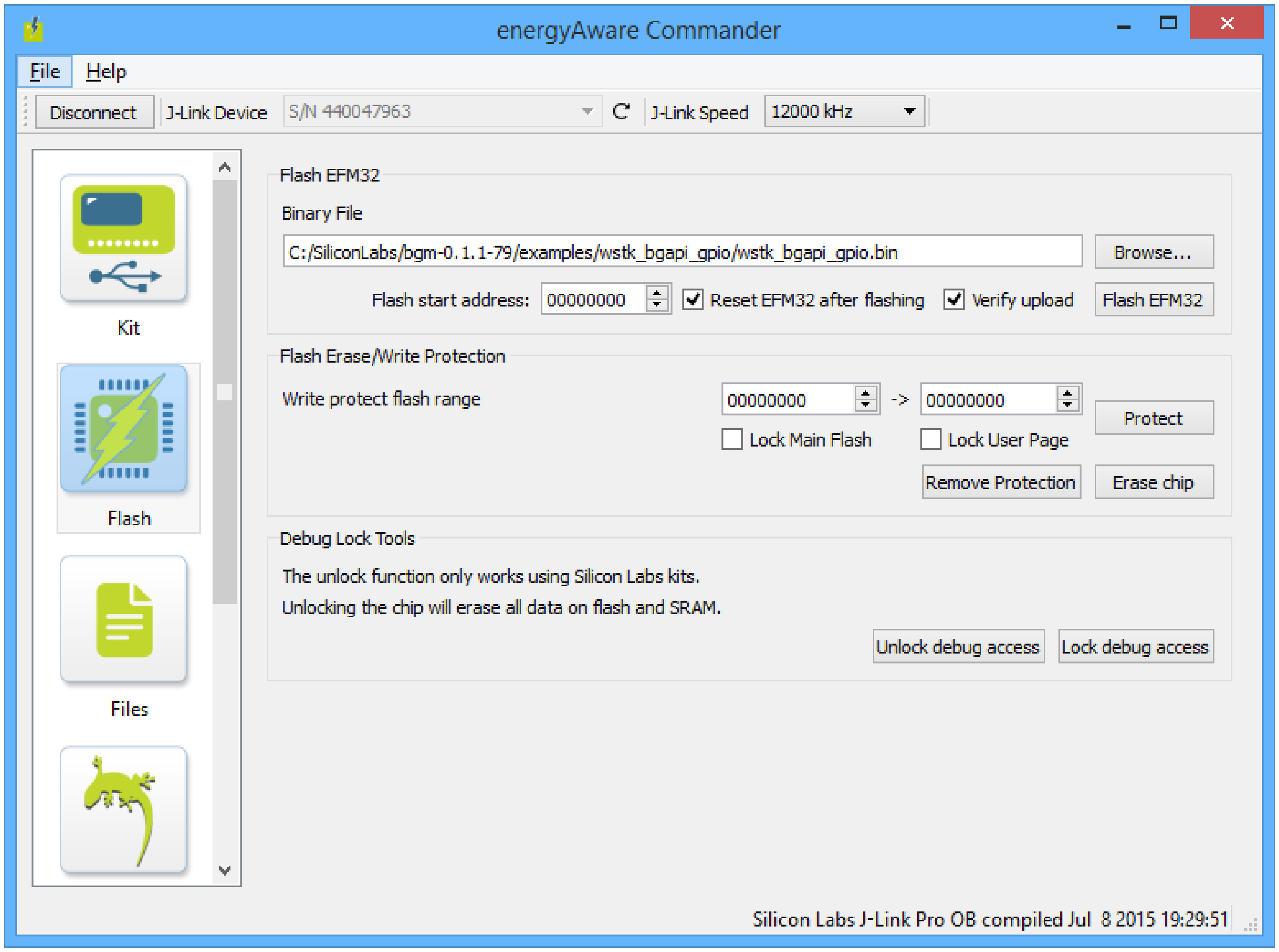
Note: Compiling the project with `bgbuild.exe -f` will also flash the resulting **.bin** file into the hardware using the Segger J-link interface. Running BGBuild without the `-f` option will only compile the project.

3.5 Flashing Module Firmware with eACommander

eACommander is a simple application which can be used to flash new firmware into the Bluetooth Smart hardware via the Segger J-link interface. eACommander is included in the Bluetooth Smart SDK and can be found in the **/bin** folder.

To flash the firmware with eACommander:

1. Connect the WSTK main board to the PC via the USB connector.
2. Start the eACommander application.
3. Make sure you see a J-link device at the top of the UI.
4. Click the **Connect** button to connect to the debugger.
5. Select the **Flash** icon on the left side.
6. Browse to the .bin file you want to flash into the device.
7. Leave all other settings at default values.
8. Press the **Flash EFM32** button and wait for the upload to finish.



J-Link Device S/N 440047963
J-Link Speed 12000 kHz

Flash EFM32
Binary File: C:/SiliconLabs/bgm-0.1.1-79/examples/wstk_bgapi_gpio/wstk_bgapi_gpio.bin
Flash start address: 00000000 ✓ Reset EFM32 after flashing ✓ Verify upload

Flash Erase/Write Protection
Write protect flash range 00000000 -> 00000000
☐ Lock Main Flash ☐ Lock User Page

Debug Lock Tools
The unlock function only works using Silicon Labs kits.
Unlocking the chip will erase all data on flash and SRAM.

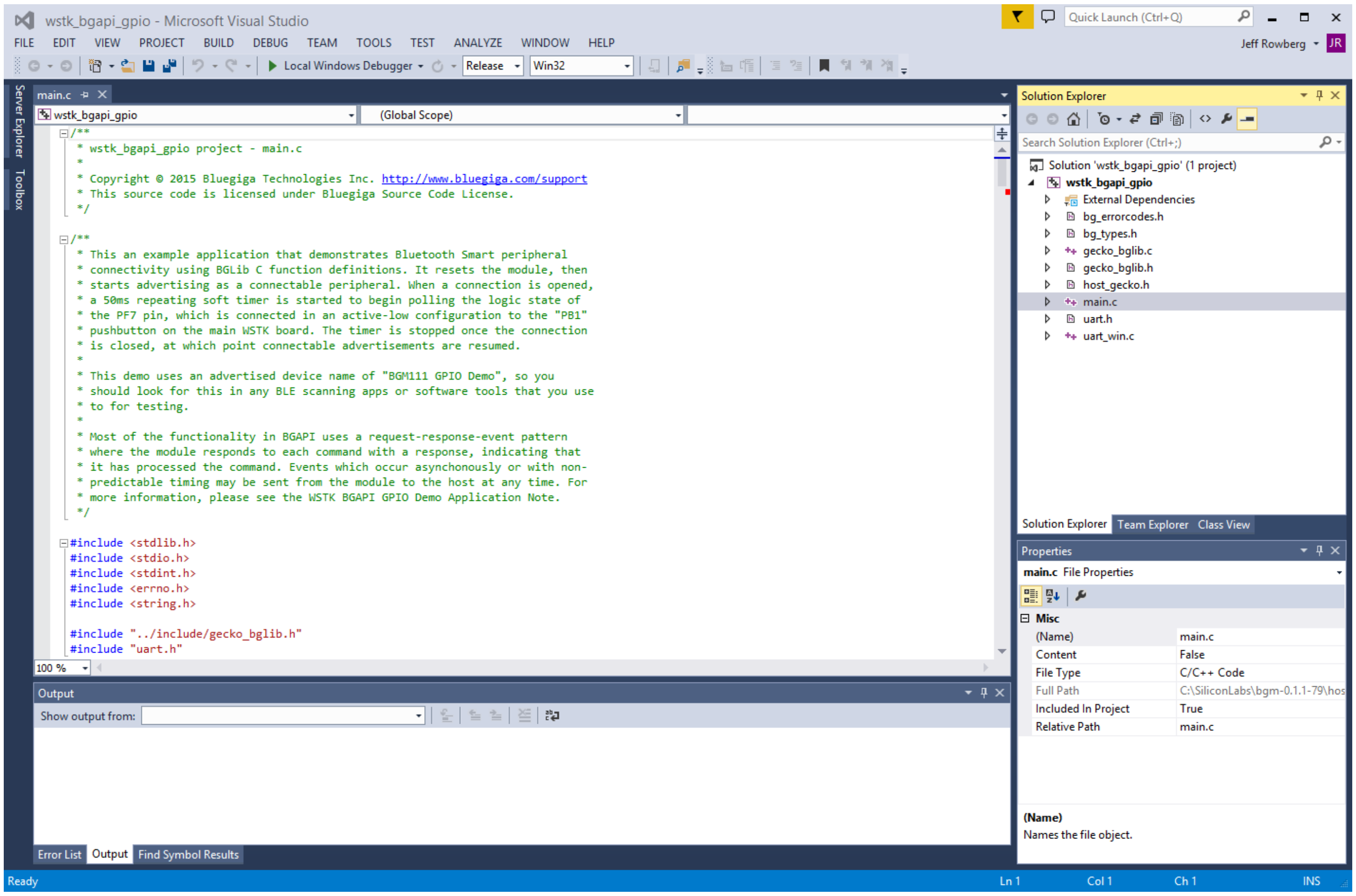
Silicon Labs J-Link Pro OB compiled Jul 8 2015 19:29:51

Figure 3.6. Flashing with eACommander

Note: After a firmware update, you should physically reset the Bluetooth Smart hardware by pressing the RESET button on the WSTK main board to ensure the new firmware is running.

3.6 Compiling the Host Application Project with Visual Studio

The host project source files for this example can be found in the **/host_example/wstk_bgapi_gpio** subfolder from the SDK. Once you have Microsoft Visual Studio installed on your PC, you can simply open the Solution (**.sln**) file from this folder, and it will open directly in the application. Here is what it looks like in Visual Studio 2013, after opening the project and then the "**main.c**" source file:



```
/**
 * wstk_bgapi_gpio project - main.c
 *
 * Copyright © 2015 Bluegiga Technologies Inc. http://www.bluegiga.com/support
 * This source code is licensed under Bluegiga Source Code License.
 */

/**
 * This an example application that demonstrates Bluetooth Smart peripheral
 * connectivity using BGLib C function definitions. It resets the module, then
 * starts advertising as a connectable peripheral. When a connection is opened,
 * a 50ms repeating soft timer is started to begin polling the logic state of
 * the PF7 pin, which is connected in an active-low configuration to the "PB1"
 * pushbutton on the main WSTK board. The timer is stopped once the connection
 * is closed, at which point connectable advertisements are resumed.
 *
 * This demo uses an advertised device name of "BGM111 GPIO Demo", so you
 * should look for this in any BLE scanning apps or software tools that you use
 * to for testing.
 *
 * Most of the functionality in BGAPI uses a request-response-event pattern
 * where the module responds to each command with a response, indicating that
 * it has processed the command. Events which occur asynchonously or with non-
 * predictable timing may be sent from the module to the host at any time. For
 * more information, please see the WSTK BGAPI GPIO Demo Application Note.
 */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>

#include "../include/gecko_bglib.h"
#include "uart.h"
```

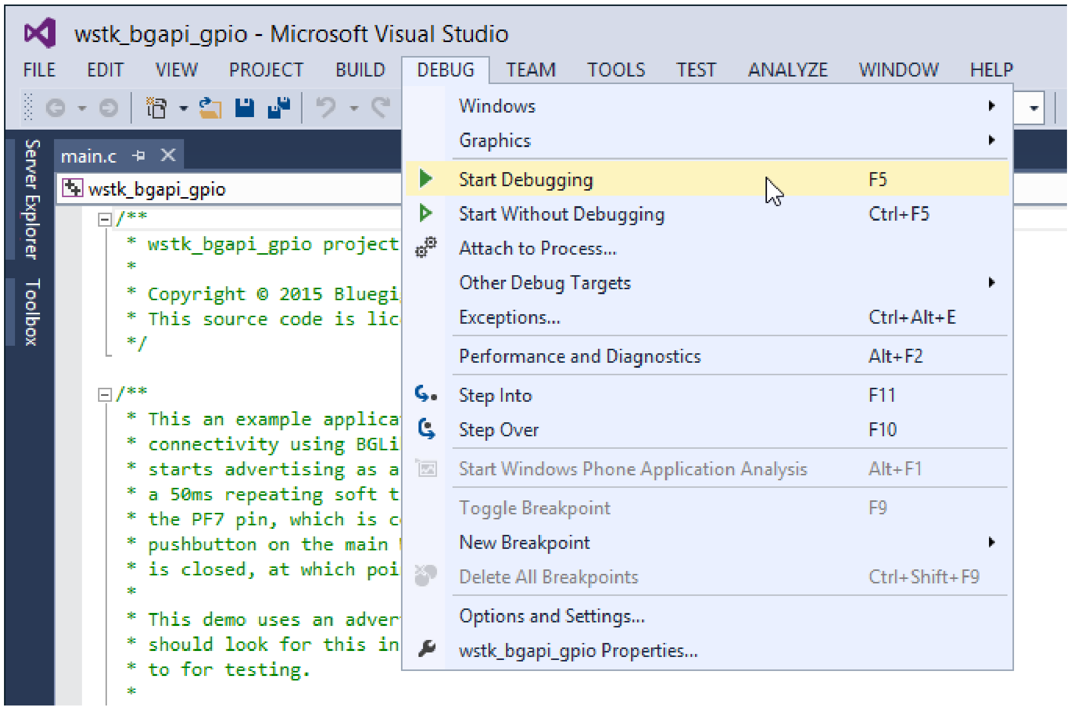
Solution 'wstk_bgapi_gpio' (1 project)
- **wstk_bgapi_gpio**
  - External Dependencies
  - bg_errorcodes.h
  - bg_types.h
  - gecko_bglib.c
  - gecko_bglib.h
  - host_gecko.h
  - main.c
  - uart.h
  - uart_win.c

Misc	
(Name)	main.c
Content	False
File Type	C/C++ Code
Full Path	C:\SiliconLabs\bgm-0.1.1-79\hos
Included In Project	True
Relative Path	main.c

(Name)
Names the file object.

Figure 3.7. Visual Studio Project View

You can compile the project using the BUILD menu and the "Build Solution item", or by pressing Ctrl+Shift+B to perform the same action. Likewise, you can compile and run the project at the same time automatically simply by using the DEBUG menu and the "Start Debugging" menu item, or by pressing the F5 key.


wstk_bgapi_gpio - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Server Explorer
Toolbox
main.c
wstk_bgapi_gpio
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Attach to Process...
Other Debug Targets
Exceptions... Ctrl+Alt+E
Performance and Diagnostics Alt+F2
Step Into F11
Step Over F10
Start Windows Phone Application Analysis Alt+F1
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Options and Settings...
wstk_bgapi_gpio Properties...

```
/**
 * wstk_bgapi_gpio project
 *
 * Copyright © 2015 Bluegi
 * This source code is lic
 */

/**
 * This an example applica
 * connectivity using BGLi
 * starts advertising as a
 * a 50ms repeating soft t
 * the PF7 pin, which is c
 * pushbutton on the main
 * is closed, at which poi
 *
 * This demo uses an adver
 * should look for this in
 * to for testing.
 *
```

Figure 3.8. Compiling and Running in Visual Studio

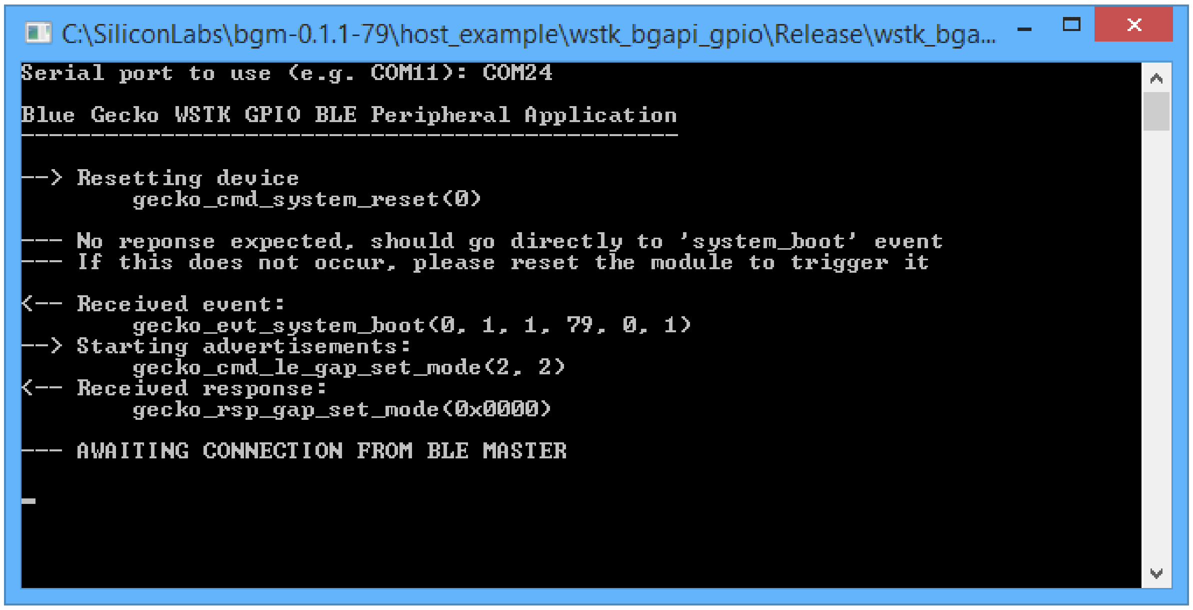
Note: The source code takes advantage of the `_DEBUG` compiler macro exposed by the Visual Studio compiler depending on whether you have selected the `Debug` or `Release` project configuration. While both will work, the `Debug` configuration will generate much more console output during testing. The steps below show the output when using the `Release` configuration.

3.7 Running the Host Application on Windows

Once you have compiled the host application in Visual Studio, there will be a new executable file called **wstk_gpio_demo.exe** found in the **/host_example/wstk_gpio_demo/Release** folder (or **/Debug** if you chose the Debug configuration instead). To run the host application on a PC, do the following:

- Ensure the WSTK board has been flashed with the corresponding module firmware project.
- Connect the WSTK main board to the PC via a mini USB cable.
- Make sure the power switch on the main board is set to the AEM position.
- Run the **wstk_gpio_demo.exe** file that is in the relevant **/Release** or **/Debug** subfolder.
- Enter the correct COM port to match what is provided by the AEM debug port (mini USB connector on WSTK).
- Follow the on-screen instructions (only the reset button may be required, everything else is automatic).

When you first start the application, the reset API command is sent, followed by a command to start advertising once the expected boot event occurs. Incoming response and events and outgoing commands used by the application are shown with their parameters:



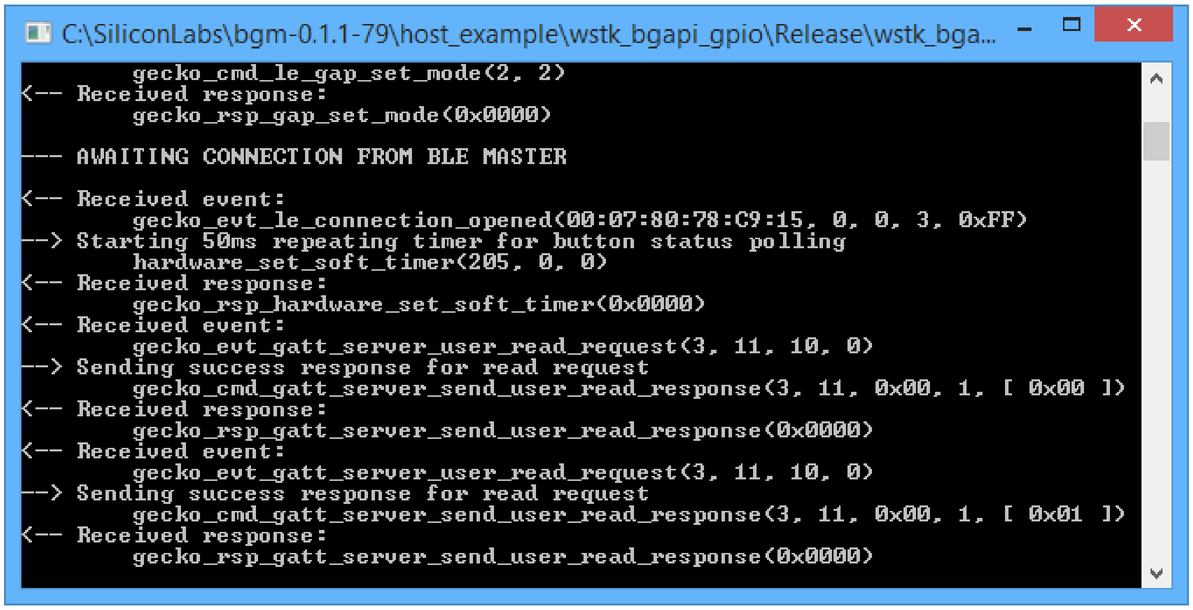
C:\SiliconLabs\bgm-0.1.1-79\host_example\wstk_bgapi_gpio\Release\wstk_bga...

```
Serial port to use (e.g. COM11): COM24

Blue Gecko WSTK GPIO BLE Peripheral Application
-----------------------------------------------

--> Resetting device
        gecko_cmd_system_reset(0)

--- No reponse expected, should go directly to 'system_boot' event
--- If this does not occur, please reset the module to trigger it

<-- Received event:
        gecko_evt_system_boot(0, 1, 1, 79, 0, 1)
--> Starting advertisements:
        gecko_cmd_le_gap_set_mode(2, 2)
<-- Received response:
        gecko_rsp_gap_set_mode(0x0000)

--- AWAITING CONNECTION FROM BLE MASTER

_
```

Figure 3.9. GPIO Demo Console Application

Further steps which require some interaction from a remote device are briefly described in the console output, such as the "`AWAITING CONNECTION FROM BLE MASTER`" note shown in the screenshot above. You can follow these prompts and the instructions in Section 3.3.3 Functional Application Overview above to observe the GPIO interaction this demo provides. Refer to the detailed instructions for client-side testing from a mobile device in the next section.

The screenshot below shows sample output after a connection and a few GPIO read requests. Notice the parameters listed in each incoming and outgoing packet. These parameters are documented in the API reference material in more detail:


C:\SiliconLabs\bgm-0.1.1-79\host_example\wstk_bgapi_gpio\Release\wstk_bga...

```
        gecko_cmd_le_gap_set_mode(2, 2)
<-- Received response:
        gecko_rsp_gap_set_mode(0x0000)

--- AWAITING CONNECTION FROM BLE MASTER

<-- Received event:
        gecko_evt_le_connection_opened(00:07:80:78:C9:15, 0, 0, 3, 0xFF)
--> Starting 50ms repeating timer for button status polling
        hardware_set_soft_timer(205, 0, 0)
<-- Received response:
        gecko_rsp_hardware_set_soft_timer(0x0000)
<-- Received event:
        gecko_evt_gatt_server_user_read_request(3, 11, 10, 0)
--> Sending success response for read request
        gecko_cmd_gatt_server_send_user_read_response(3, 11, 0x00, 1, [ 0x00 ])
<-- Received response:
        gecko_rsp_gatt_server_send_user_read_response(0x0000)
<-- Received event:
        gecko_evt_gatt_server_user_read_request(3, 11, 10, 0)
--> Sending success response for read request
        gecko_cmd_gatt_server_send_user_read_response(3, 11, 0x00, 1, [ 0x01 ])
<-- Received response:
        gecko_rsp_gatt_server_send_user_read_response(0x0000)
```

Figure 3.10. Connection and Read Requests

3.8 Testing Client Connectivity with an iPhone and BLExplr

The module firmware and host application project described above implement only the BLE peripheral / GATT server side of the application. In order to use the functionality provided by the peripheral, a central/client device is needed. This section describes how to do this using an iPhone with Bluetooth Smart capabilities and the BLExplr app.

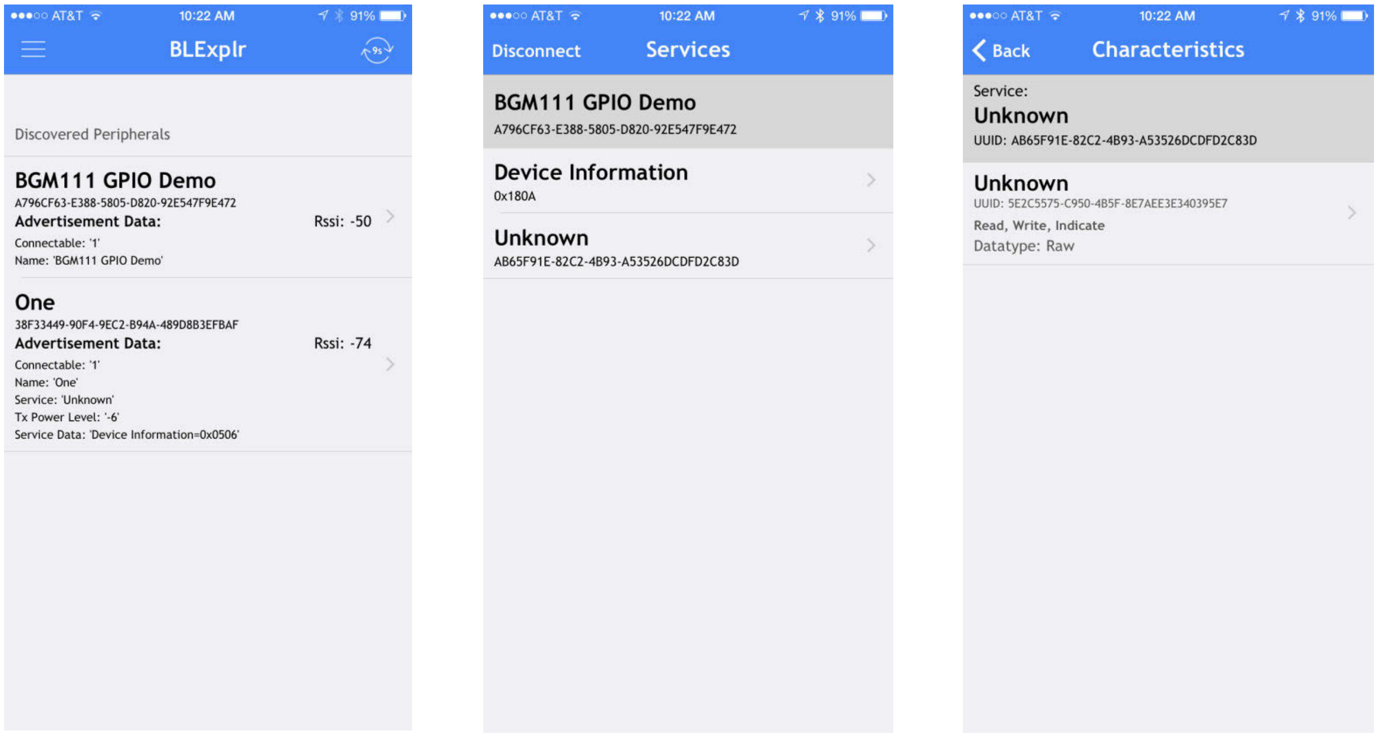
Note: It is possible to perform the steps outlined below using any standards-compliant Bluetooth Smart client device. This choice of hardware and software is chosen since it is fairly common and works reliably. Some alternatives include:

- Any BLE-capable iOS device (iPhone 4S or newer, iPad 3 or newer) and BLE scanning app.
- Any BLE-capable Android device (Android 4.3 or newer) and BLE scanning app.
- Bluegiga BLEGUI software and BLED112 dongle.

3.8.1 Scanning and Connecting to the BGM111 GPIO Demo Peripheral

To begin, first ensure that all firmware compilation/flashing is completed as described in previous sections, and that the host application is running on the PC that the WSTK board is connected to. The console output should indicate that the device is waiting for connections from a BLE master device. Once this is prepared:

1. Start the BLExplr app on your iPhone and scan for devices using the icon in the top right corner. Once you see the “BGM111 GPIO Demo” peripheral appear, tap on it to connect.
2. Select the service in the list with UUID beginning “`AB65F91E...`”. This is our custom GPIO control service. The app displays “Unknown” for the service name because it is not an official, adopted service from the Bluetooth SIG.
3. Select the characteristic in the list with UUID beginning “`5E2C5575...`”. This is our custom GPIO control characteristic. Similar to the service, this shows “Unknown” since it is not an official characteristic. Notice that the supported GATT data transfer methods are shown: **Read, Write, Indicate**.



Discovered Peripherals

BGM111 GPIO Demo
A796CF63-E388-5805-D820-92E547F9E472
Advertisement Data: Rssi: -50
Connectable: '1'
Name: 'BGM111 GPIO Demo'

One
38F33449-90F4-9EC2-B94A-489D8B3EFBAF
Advertisement Data: Rssi: -74
Connectable: '1'
Name: 'One'
Service: 'Unknown'
Tx Power Level: '-6'
Service Data: 'Device Information=0x0506'

BGM111 GPIO Demo
A796CF63-E388-5805-D820-92E547F9E472

Device Information
0x180A

Unknown
AB65F91E-82C2-4B93-A53526DCDFD2C83D

Service:
Unknown
UUID: AB65F91E-82C2-4B93-A53526DCDFD2C83D

Unknown
UUID: 5E2C5575-C950-4B5F-8E7AEE3E340395E7
Read, Write, Indicate
Datatype: Raw

Figure 3.11. BLExplr Scanning, Connection, and GATT

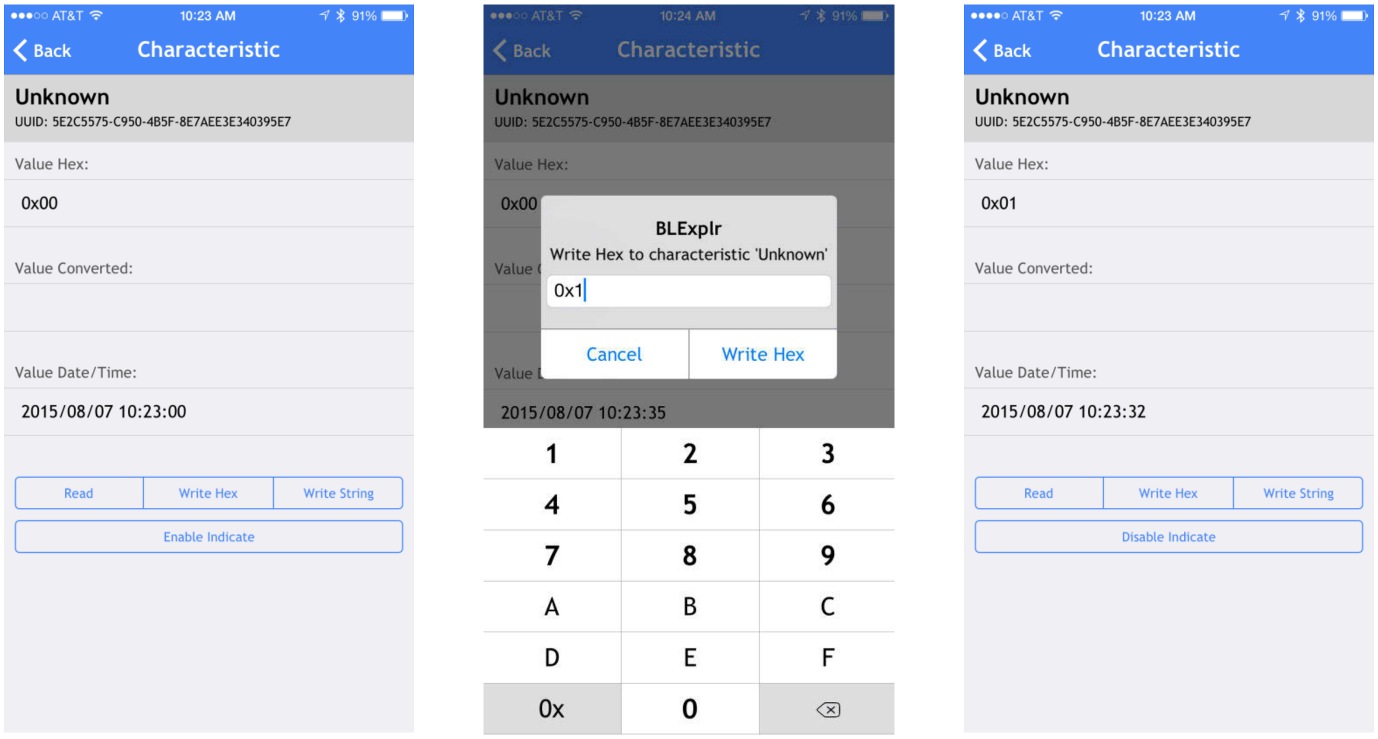
3.8.2 Reading and Writing GPIO Logic States

The module firmware and host application logic provide a method to set the logic state of PF6 (WSTK Button 0 / LED 0), and to read the logic state of the PF7 (WSTK Button 1 / LED 1). These two operations are accomplished by using standard GATT write and read operations from a remote GATT client, such as BLExplr on an iPhone as shown here.

Once you have selected the GPIO control characteristic in the previous step, BLExplr will read the initial value of the characteristic (since it is readable), and then give you control over additional reads, writes, and the “indicate” subscription status that allows the peripheral to push updates to the phone.

The value read or indicated from the GPIO control characteristic is only a single byte, and will be either 0x00 if Button1 is not pressed, or 0x01 if Button1 is pressed. If you have subscribed to indications with the “Enable Indicate” button in BLExplr, then changes will be pushed automatically, and the timestamp will be updated accordingly each time the value changes. Otherwise, you will need to tap the “Read” button in order to get the current logic state. When you read the value or if the button status changes when indications are enabled, you should be able to observe additional console output from the host application which shows the API methods being used.

To change the PF7 logic state and set LED1 on or off, write either a single zero byte (off) or any non-zero byte (on) to the same characteristic using the “Write Hex” button in BLExplr. The change will take effect immediately, and you should see LED1 light up or go off.


Unknown
UUID: 5E2C5575-C950-4B5F-8E7AEE3E340395E7
Value Hex: 0x00
Value Converted:
Value Date/Time: 2015/08/07 10:23:00

Unknown
UUID: 5E2C5575-C950-4B5F-8E7AEE3E340395E7
Value Hex: 0x00
Value Date/Time: 2015/08/07 10:23:35
BLExplr
Write Hex to characteristic 'Unknown'
0x1

Unknown
UUID: 5E2C5575-C950-4B5F-8E7AEE3E340395E7
Value Hex: 0x01
Value Converted:
Value Date/Time: 2015/08/07 10:23:32

Figure 3.12. BLExplr Read, Write, and Indicate

Once you have finished testing, you can close the connection by tapping “Back” until you disconnect from the peripheral. The console output on the PC should show that connectable advertising resumes automatically.

4. Summary and BGAPI Quick-Reference Guide

This document shows the steps required to implement a BGLib/BGAPI-based design using a Windows PC host for the host application. If you are implementing a project on a different platform, keep the following checklist handy for reference—remembering that module firmware and host application are separate projects that interact with each other over UART:

- Module firmware project
  - Firmware requires UART enabled with BGAPI control configured.
  - Flow control between host as modules is recommended if possible.
  - BGScript is not used for Bluetooth Smart projects, which are built on BGAPI/BGLib.
  - Named characteristic IDs are generated as ***#define***'d constants in a **gatt_db.h** file generated as a result of the compile process, which you can copy or include in your host application.
  - The module will boot into an idle state and requires commands sent from the host before it will exhibit any other behavior. Usually, this means starting advertisements or something similar.
- Host application project
  - Include all of the required BGLib source and support files in your main project.
  - Define platform-specific UART RX/TX routines with length/data arguments.
  - Use **BGLIB_DEFINE** and **BGLIB_INITIALIZE** macros to set up the environment.
  - All packets have one 4-byte header, followed by zero or more payload bytes.
  - Create an event-catching loop and switch/case statement (or "if" block) for all events needed.
  - Use packet pointers to keep track of events and responses.
  - Use the data structure pointers and members to access event and response parameters:

    ```
    obj -> data.packet_name.member_name
    ```

  - The SDK's provided BGLib implementation may be substituted with a custom implementation as long as it can correctly parse and create command, response, and event packets.

Simplicity Studio

One-click access to MCU tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

www.silabs.com/simplicity

MCU Portfolio
www.silabs.com/mcu

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com