

## Features

- Over-the-Air (OTA) firmware update
- Shared BLE Bootloader Service to receive bootloadable images
- HID Keyboard

## General Description

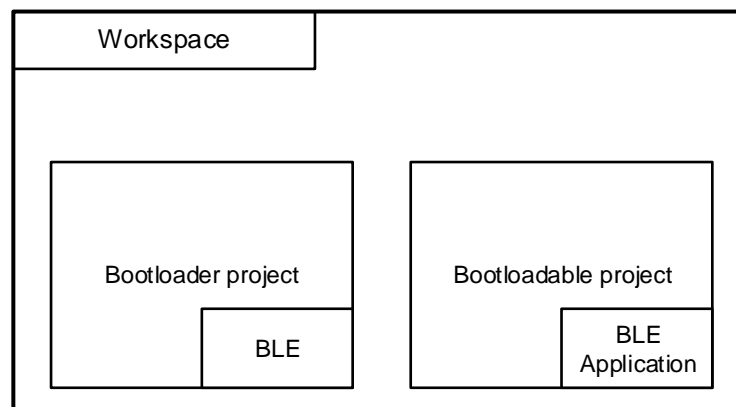
This example shows how to use the custom linker scripts to share a block of memory between the bootloader and bootloadable projects. It demonstrates how the bootloader can place the API functions so that the bootloadable can also call them. This allows creating an OTA bootloader.

The purpose of the Bootloader project is to replace a bootloadable image on the device with an image sent OTA by the Bluetooth protocol.

The bootloadable project uses BLE APIs implemented in the bootloader part of the memory (Figure 1).

**Note** Currently GCC 5.4.1, MDK, and IAR compilers are supported.

Figure 1. OTA Fixed Stack Workspace



By default, both bootloader and bootloadable projects are expected to be located in the same workspace. However, the user can save projects in any location and modify paths in the build script.

Figure 2 shows the architecture of the bootloader project; Figure 3 shows the architecture of the bootloadable project.

Figure 2. Bootloader Project Architecture

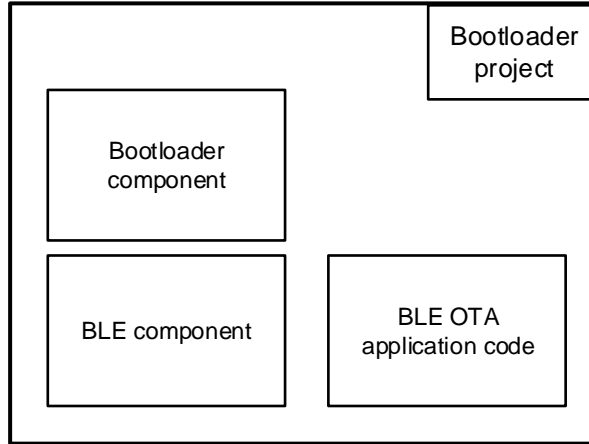
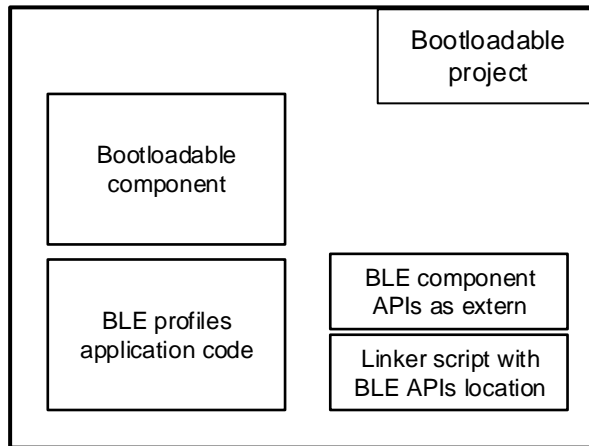


Figure 3. Bootloadable Project Architecture



The bootloadable project may use APIs from the bootloader project if:

1. These APIs are listed as "extern" in a dedicated header.
2. These APIs addresses are provided in the bootloadable project linker script by the post-build script of the bootloader.
3. Bootloader \*.hex/\*.elf has not been modified since Step 2.

## Development Kit Configuration

This example project is designed to run on the Cypress CY8CKIT-042-BLE kit. A description of the kit along with more example programs and ordering information can be found at <http://www.cypress.com/go/cy8ckit-042-ble>.

The kit must be powered by 3.3 V (J16 is set to 1 and 2). No connection on the kit board is required to use this example project. If you want to power the kit with the 5V supply you should modify the Operating Conditions in the \*.cydwr files under the **System** tab for both bootloader and bootloadable projects:

Operating Conditions	
VDDA (V)	3.3
Variable VDDA	<input checked="" type="checkbox"/>
VDDD (V)	3.3
VDDR (V)	3.3

Refer to the “Setup” and “Run Example Project” section of this document for instructions on how to use this example project.

If the kit is powered with different from 3.3 voltage – the projects settings must be updated appropriately in the System DWR settings on the **System** tab.

## Bootloader Project Configuration

The Bootloader project contains the following components:

- Bluetooth Low Energy (BLE)
- Bootloader
- Software Transmit UART
- Watchdog timer (WDT) that is used as general timer

## Linker Settings

Check the correctness of the project linker settings:

1. Right-click on the bootloader project in **Workspace Explorer**.
2. Click **Build Settings**.
3. Expand **Linker** under the selected compiler:
  - a. If **ARM GCC** compiler is selected:

Check the values of the following options in the **General** section:

Section	Option	Value
General	Custom Linker Script	.\cm0gcc.ld

b. If **ARM MDK Generic**:

Check the values of the following options in the **General** and **Command line** section:

Section	Option	Value
General	Use MicroLIB	False
Command line	Custom flags	<pre>--keep=Advertising_LED_1_SetDriveMode -- keep=Advertising_LED_2_SetDriveMode -- keep=Bootloading_LED_SetDriveMode -- keep=Bootloader_Service_Activation_Read -- keep=Bootloader_Service_Activation_SetDriveMode -- keep=Bootloader_Service_Activation_Write --keep=CyBle_StackInit -- keep=CyBle_Shutdown --keep=CyBle_SoftReset --keep=CyBle_EnterLPM -- keep=CyBle_ExitLPM --keep=CyBle_ProcessEvents -- keep=CyBle_SetDeviceAddress --keep=CyBle_GetDeviceAddress -- keep=CyBle_GetRssi --keep=CyBle_GetTxPowerLevel -- keep=CyBle_SetTxPowerLevel --keep=CyBle_GetBleClockCfgParam -- keep=CyBle_SetBleClockCfgParam --keep=CyBle_GenerateRandomNumber -- keep=CyBle_SetCeLengthParam --keep=CyBle_WriteAuthPayloadTimeout -- keep=CyBle_ReadAuthPayloadTimeout --keep=CyBle_GetStackLibraryVersion - -keep=CyBle_GetBleSsState --keep=CyBle_AesEncrypt -- keep=CyBle_AesCcmInit --keep=CyBle_AesCcmEncrypt -- keep=CyBle_AesCcmDecrypt --keep=CyBle_SetTxGainMode -- keep=CyBle_SetRxGainMode --keep=CyBle_GattGetMtuSize -- keep=CyBle_ApplCallback --keep=cyBle_deviceAddress -- keep=cyBle_sflashDeviceAddress --keep=cyBle_discoveryModelInfo -- keep=cyBle_scanRspData --keep=cyBle_discoveryData -- keep=cyBle_discoveryParam --keep=CyBle_Init --keep=CyBle_ServiceInit -- keep=CyBle_Start --keep=CyBle_Stop --keep=CyBle_StoreBondingData -- keep=CyBle_GappStartAdvertisement --keep=CyBle_GappStopAdvertisement -- keep=CyBle_ChangeAdDeviceAddress --keep=CyBle_GapSetLocalName -- keep=CyBle_GapGetLocalName --keep=CyBle_Get16ByPtr -- keep=CyBle_Set16ByPtr --keep=cyBle_bass --keep=CyBle_BasInit -- keep=CyBle_BasRegisterAttrCallback -- keep=CyBle_BassSetCharacteristicValue -- keep=CyBle_BassGetCharacteristicValue -- keep=CyBle_BassGetCharacteristicDescriptor -- keep=CyBle_BassWriteEventHandler --keep=CyBle_BassSendNotification -- keep=cyBle_diss --keep=CyBle_DisInit --keep=CyBle_DisRegisterAttrCallback -- keep=CyBle_DissSetCharacteristicValue -- keep=CyBle_DissGetCharacteristicValue --keep=cyBle_hidss -- keep=CyBle_HidsInit --keep=CyBle_HidsRegisterAttrCallback -- keep=CyBle_HidssSetCharacteristicValue -- keep=CyBle_HidssGetCharacteristicValue -- keep=CyBle_HidssGetCharacteristicDescriptor -- keep=CyBle_HidssWriteEventHandler --keep=CyBle_HidssSendNotification -- keep=CyBle_HidssOnDeviceConnected --keep=cyBle_scps -- keep=CyBle_ScpsInit --keep=CyBle_ScpsRegisterAttrCallback -- keep=CyBle_ScpsSetCharacteristicValue -- keep=CyBle_ScpsGetCharacteristicValue -- keep=CyBle_ScpsGetCharacteristicDescriptor -- keep=CyBle_ScpsWriteEventHandler --keep=CyBle_ScpsSendNotification -- keep=CyBle_GattsEnableAttribute --apcs=/pic --info unused</pre>

**Note** MicroLib has compilation limitations which prevent the usage of code sharing. Cause of this MircoLib cannot be used neither in BLE OTA Bootloader nor in BLE OTA Bootloadable projects.

c. And if the project was exported to the IAR:



Set the following settings in the project options:

Section	Option	Value
C/C++ Compiler	Command line options	--no_fragments --no_inline --no_wrap_diagnostics --interwork --no_unaligned_access --cpu_mode=thumb
Linker	Command line options	--keep=Bootloader_Checksum --keep=Advertising_LED_1_SetDriveMode --keep=Advertising_LED_2_SetDriveMode --keep=Bootloading_LED_SetDriveMode --keep=Bootloader_Service_Activation_Read --keep=Bootloader_Service_Activation_SetDriveMode --keep=Bootloader_Service_Activation_Write --keep=CyBle_StackInit --keep=CyBle_Shutdown --keep=CyBle_SoftReset --keep=CyBle_EnterLPM --keep=CyBle_ExitLPM --keep=CyBle_ProcessEvents --keep=CyBle_SetDeviceAddress --keep=CyBle_GetDeviceAddress --keep=CyBle_GetRssi --keep=CyBle_GetTxPowerLevel --keep=CyBle_SetTxPowerLevel --keep=CyBle_GetBleClockCfgParam --keep=CyBle_SetBleClockCfgParam --keep=CyBle_GenerateRandomNumber --keep=CyBle_SetCeLengthParam --keep=CyBle_WriteAuthPayloadTimeout --keep=CyBle_ReadAuthPayloadTimeout --keep=CyBle_GetStackLibraryVersion --keep=CyBle_GetBleSsState --keep=CyBle_AesEncrypt --keep=CyBle_AesCcmInit --keep=CyBle_AesCcmEncrypt --keep=CyBle_AesCcmDecrypt --keep=CyBle_SetTxGainMode --keep=CyBle_SetRxGainMode --keep=CyBle_GattGetMtuSize --keep=CyBle_ApplCallback --keep=cyBle_deviceAddress --keep=cyBle_sflashDeviceAddress --keep=cyBle_discoveryModelInfo --keep=cyBle_scanRspData --keep=cyBle_discoveryData --keep=cyBle_discoveryParam --keep=CyBle_Init --keep=CyBle_ServiceInit --keep=CyBle_Start --keep=CyBle_Stop --keep=CyBle_StoreBondingData --keep=CyBle_GappStartAdvertisement --keep=CyBle_GappStopAdvertisement --keep=CyBle_GattsEnableAttribute --keep=CyBle_ChangeAdDeviceAddress --keep=CyBle_GapSetLocalName --keep=CyBle_GapGetLocalName --keep=CyBle_Get16ByPtr --keep=CyBle_Set16ByPtr --keep=cyBle_bass --keep=CyBle_BasInit --keep=CyBle_BasRegisterAttrCallback --keep=CyBle_BassSetCharacteristicValue

		<pre>--keep=CyBle_BassGetCharacteristicValue --keep=CyBle_BassGetCharacteristicDescriptor --keep=CyBle_BassWriteEventHandler --keep=CyBle_BassSendNotification --keep=cyBle_diss --keep=CyBle_DisInit --keep=CyBle_DisRegisterAttrCallback --keep=CyBle_DissSetCharacteristicValue --keep=CyBle_DissGetCharacteristicValue --keep=cyBle_hidss --keep=CyBle_HidsInit --keep=CyBle_HidsRegisterAttrCallback --keep=CyBle_HidssSetCharacteristicValue --keep=CyBle_HidssGetCharacteristicValue --keep=CyBle_HidssGetCharacteristicDescriptor --keep=CyBle_HidssWriteEventHandler --keep=CyBle_HidssSendNotification --keep=CyBle_HidssOnDeviceConnected --keep=cyBle_scps --keep=CyBle_ScpsInit --keep=CyBle_ScpsRegisterAttrCallback --keep=CyBle_ScpsSetCharacteristicValue --keep=CyBle_ScpsGetCharacteristicValue --keep=CyBle_ScpsGetCharacteristicDescriptor --keep=CyBle_ScpsWriteEventHandler --keep=CyBle_ScpsSendNotification --semihosting</pre>
--	--	--

## Bluetooth Low Energy (BLE)

The BLE component is used to implement the BLE Human Interface Device (HID over GATT keyboard), Battery Service (BAS), Scan Parameters (SCPS), Device Information (DIS), and a hidden Bootloader Service. The purpose of the Bootloader Service is to receive other bootloadable images. The Bootloader Service is available only in the bootloader. When a device is in the bootloadable mode (the device operation is the same as in the BLE\_HID\_Keyboard example, except for button **SW2**) in this example project, it forces the device to go to the bootloader mode.

Figure 4. BLE GATT Settings

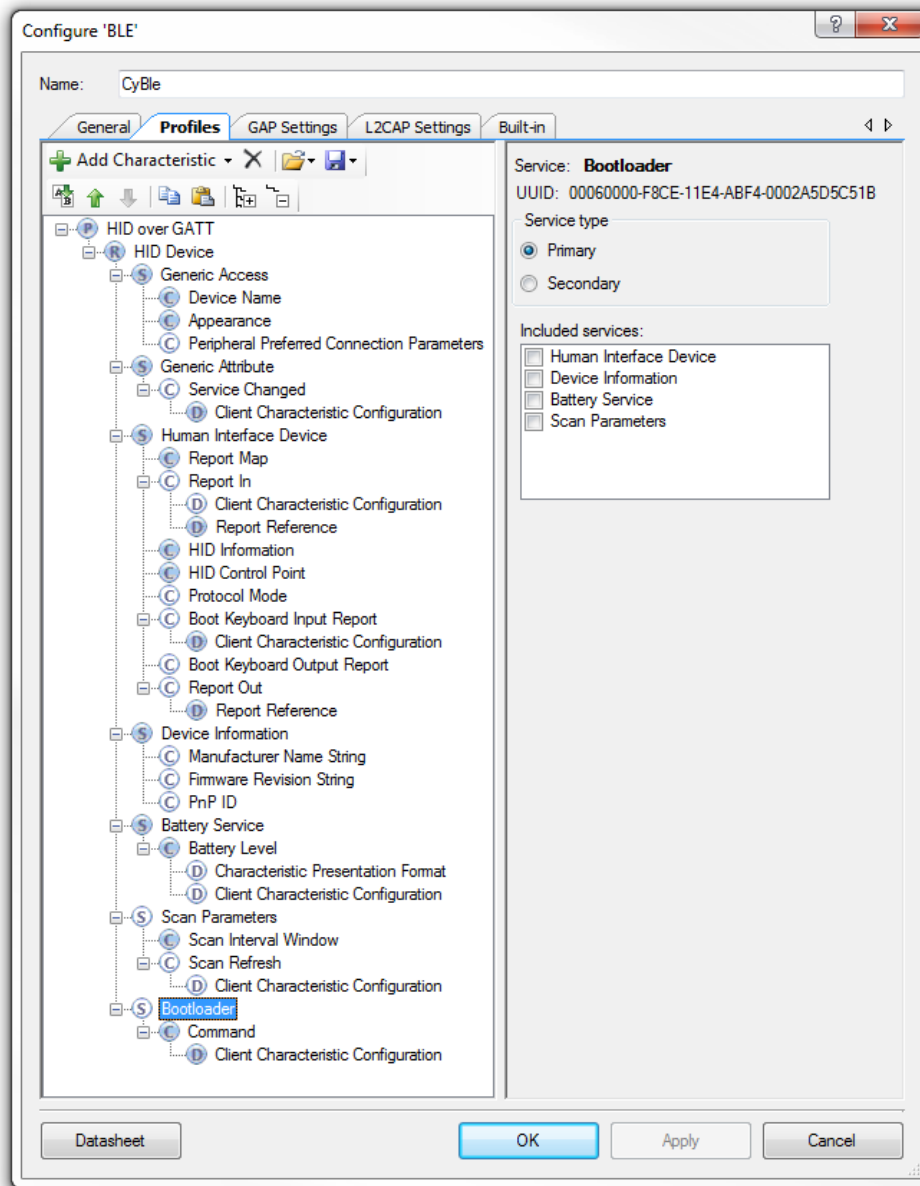


Figure 5. BLE GATT Settings

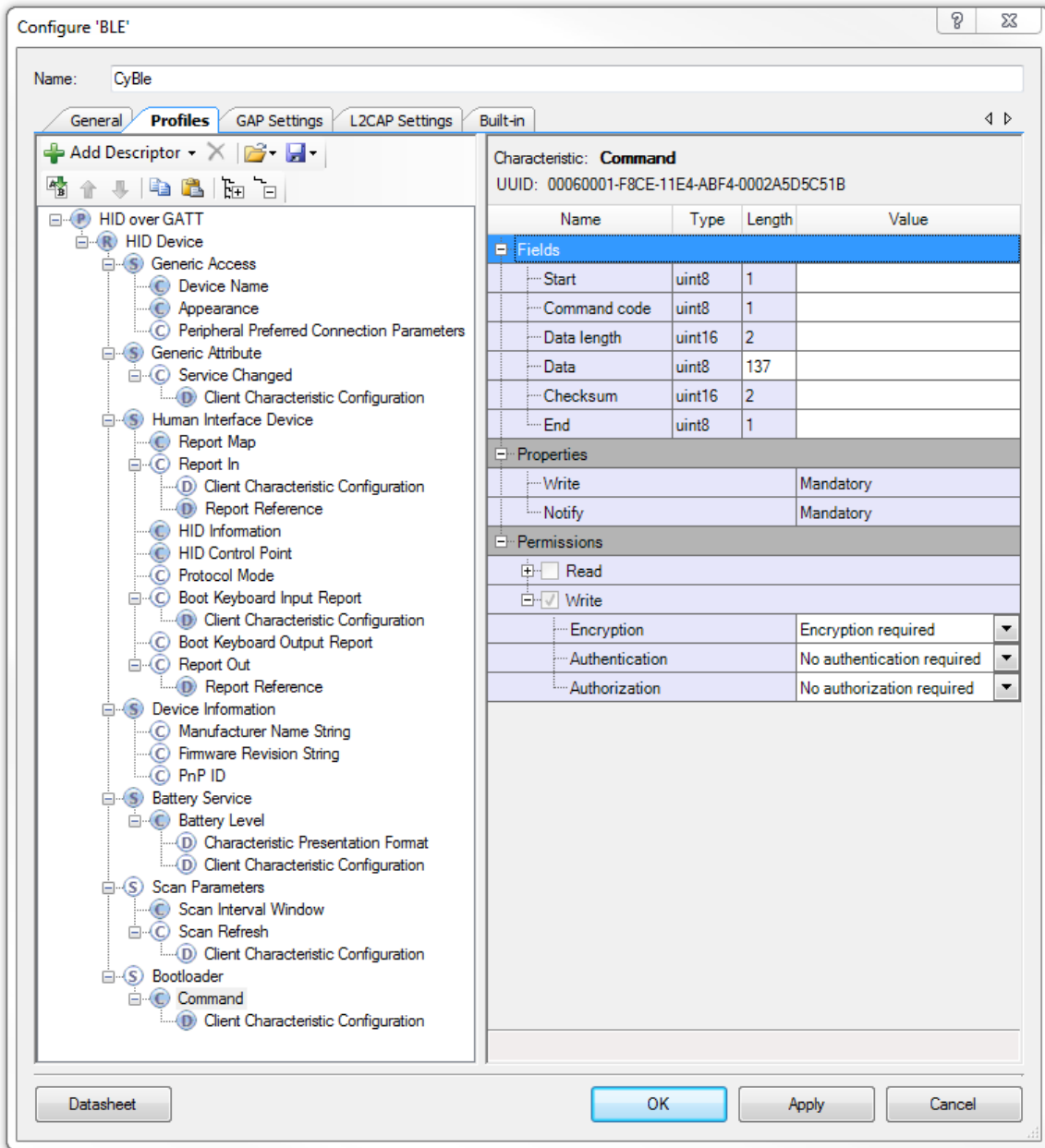
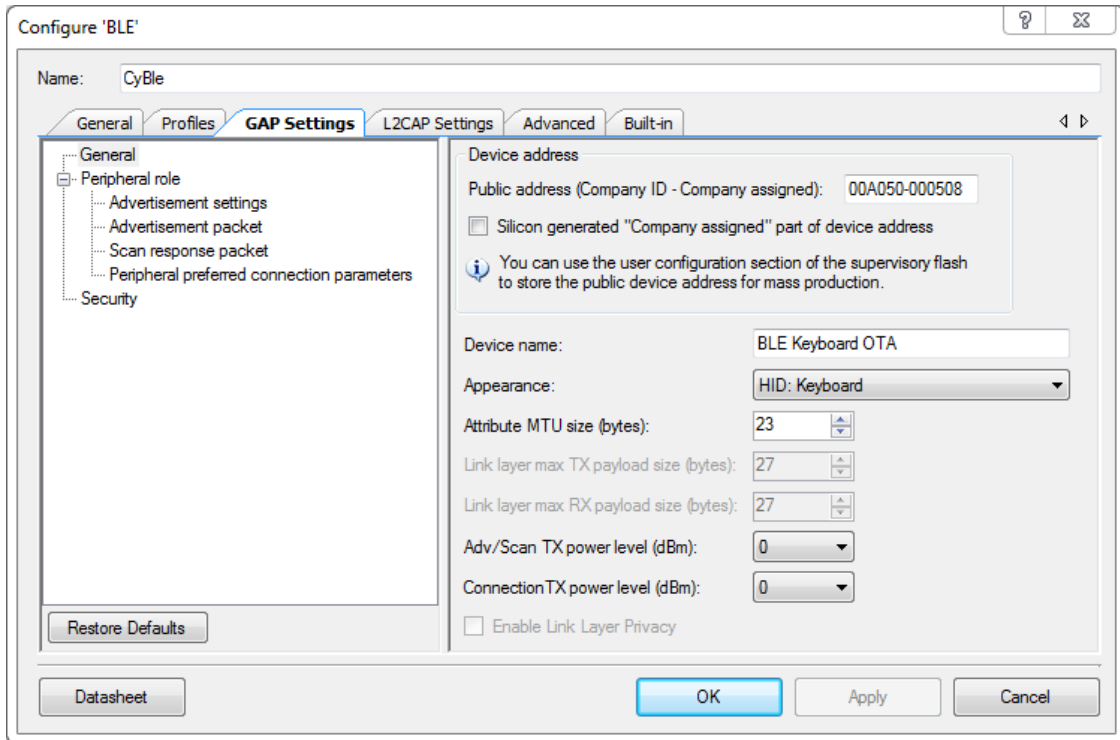




Figure 6. BLE GAP Settings



The Bootloader Service is enabled by default in the bootloader mode for communication with the Bootloader Host tool.

The BLE Bootloader Service has one characteristic that supports the “Write/Write Long Characteristic Values” procedure and notifications. The described characteristic also has a characteristic descriptor - Client Characteristic Configuration.

For communication, the Bootloader Host tool writes a command to the Command Characteristic and enables notifications in the characteristic descriptor.

The bootloader emulator reads the command from the Characteristic, processes it, and, if notifications are enabled in the characteristic descriptor, writes a response to the notification that is sent to Bootloader Host Tool.

The Bootloader Host tool receives the notification and, depending on its content (a bootloader emulator response), either sends the next command or reports an error.

The BLE component is configured to have an MTU of 23 byte and the Bootloader Service UUID, which is 00060000-F8CE-11E4-ABF4-0002A5D5C51B.

After pressing onboard **SW2**, the LED changes its indication to red. This can be applied to both “advertising” and “connected” modes.

**Note** The bootloader project is performing flash writes. Due to this, the maximum connection interval value is set to 15 ms. Changing this parameter value to a smaller value might result in disconnection during bootloading if a flash write takes longer. A flash Write takes about 10ms

(refer to the device datasheet), so this value cannot be smaller. If a device flash write takes more than 15 ms, this value should be changed.

The bootloadable project shares the all BLE component settings with the bootloader project.

## Adding Services

This section describes how to add and use another service in the BLE component of the bootloader project. The basic steps include:

1. Add a service in the BLE component's customizer of the BLE\_OTA\_FixedStack\_Bootloader project.
2. **Optional** Add code to disable the new service in the bootloader project as it is done in main.c using API CyBle\_GattsDisableAttribute(). This is done to hide the unused functionality in the bootloader mode.

Figure 7. Code Example of Service Disabling

```

91 | CyBle_GattsDisableAttribute (cyBle_hidss[0].serviceHandle);
92 | CyBle_GattsDisableAttribute (cyBle_diss.serviceHandle);
93 | CyBle_GattsDisableAttribute (cyBle_bass[0].serviceHandle);
94 | CyBle_GattsDisableAttribute (cyBle_scpss.serviceHandle);
95 |

```

3. Build the BLE\_OTA\_FixedStack\_Bootloader project.
4. Rerun the mk.bat script to update the bootloadable project linker scripts.
5. Add code that uses the added BLE API service in the BLE\_OTA\_FixedStack\_Bootloadable project.
6. Add all the used BLE APIs to the OTAMandatory.h file as extern declarations in the BLE\_OTA\_FixedStack\_Bootloadable project.
7. If APIs are to be used from the bootloadable project:
  - a. For the GCC compiler, add those APIs to the section with KEEP instructions in the linker script of the bootloader project.
  - b. For the MDK compiler and for the IAR, the same procedure is performed with the linker command line arguments of the bootloader project.
  - c. If the services are removed, the APIs should also be removed.
8. Add all the used BLE types and defines to the OTAMandatory.h file of the BLE\_OTA\_FixedStack\_Bootloadable project.
9. Build the BLE\_OTA\_FixedStack\_Bootloadable project.

## Bootloader

Enable the Bootloader application validation option only if the BLE profiles do not require saving bonding information or if this information is not saved during the BLE component operation. Otherwise, saving bonding information will update the Bootloader project image, which may cause the validation to fail.

## Software Transmit UART

This component is used for printing debug information. Refer to the Using UART for Debugging section for details.

## Watch Dog Timer (WDT)

The WDT is used here as a general purpose timer for LED indication.

## Bootloadable Project Configuration

The bootloadable project contains the following components:

- Bootloadable
- Software Transmit UART

## Compiler Settings

Bootloadable project uses code sharing for BLE API from Bootloader project firmware image. And it does not have a BLE component to generate the code and header files with the APIs.

So all the BLE API in Bootloadable project are declared in *ota\_mandatory.h* header file which itself references BLE API headers copied to *\${Bootloadable}/CyBle/* folder from *\${Bootloader}/Generated\_Source/PSoC4/CyBle\*.h*.

**Note** that you shall copy this files manually to Bootloadable project when changing BLE component version in Bootloader project.

This is why *\${Bootloadable}/CyBle/* folder has to be added to compiler include paths list.

It is done by default in PSoC Creator's projects for GCC and MDK compilers.

## IAR Compiler Settings

1. Open options for Bootloadable project.
2. Under **C/C++ Compiler** category choose **Preprocessor** tab.
3. Add *\$PROJ\_DIR\CyBle* to **Additional include directories** list.

## Linker Settings

Check the correctness of the project linker settings:

1. Right-click the bootloadable project in **Workspace Explorer**.
2. Click **Build Settings**.
3. Expand **Linker** under the selected compiler:
  - a. If **ARM GCC** compiler is selected:

Check the values of the following options in the **General** and **Optimization** sections:

Section	Option	Value
General	Additional Library Directories	.\LinkerScripts\
General	Custom Linker Script	.\LinkerScripts\cm0gcc.ld
Optimization	Removed Unused Function	False

- b. If **ARM MDK Generic**:

Check the values of the following options in the “Linker” section:

Section	Option	Value
General	Custom Linker Script	.\LinkerScripts\Cm0Mdk.scat
General	Use MicroLIB	False
Command line	Custom flags	.\LinkerScripts\BootloaderSymbolsMdk.txt

- c. And if the project was exported to the IAR:

Set the following settings in the project options:

Section	Option	Value
C/C++ Compiler	Command line options	--no_fragments --no_cse --no_unroll --no_inline --no_code_motion --no_tbaa --no_scheduling --debug --no_wrap_diagnostics --interwork --no_unaligned_access --cpu_mode=thumb
	Additional include directories	\$PROJ_DIR\CyBle

Linker	Command line options	--no_dynamic_rtti_elimination --no_range_reservations --no_fragments
	Linker Configuration File / Custom Linker Script	\$PROJ_DIR\$\LinkerScripts\Cm0lar.icf

## Bootloadable

The Bootloadable component is used to create an image with the bootloadable firmware that can be updated without affecting the bootloader.

## Software Transmit UART

The component is used for printing debug information. Refer to the Using UART for Debugging section for the details.

## Watch Dog Timer (WDT)

The WDT is used here as a general purpose timer for the LED indication.

## Custom Linker Scripts

The custom linker scripts are intended for keeping the Bootloader's RAM regions not re-initialized by the Bootloadable application and including the list of exported APIs.

### Custom Linker Script Description

Bootloader and Bootloadable projects use the custom linker scripts for the GCC compiler. For other supported compilers the custom linker scripts are used only for bootloadable projects.

Compared to a generated script, the bootloader's custom linker script for GCC (cm0gcc.ld), contains a few KEEP instructions to prevent the required symbols from being removed by the linker and this script is located in the Bootloader sources folder (BLE\_OTA\_FixedStack\_Bootloader.cydsn).

For the GCC compiler, the bootloadable's custom linker script (cm0gcc.ld) includes a generated file with a list of exported symbols (BootloaderSymbolsGcc.ld) and declares the ".bootloader\_data" section as NOLOAD, so that it can't be re-initialized. The Bootloadable's custom linker script is located in the Bootloadable's folder (BLE\_OTA\_FixedStack\_Bootloadable.cydsn\LinkerScripts\).

For the MDK compiler, the bootloader does not use custom linker scripts – instead all the options are sent via a command line to the linker:

```
--keep=Advertising_LED_1_SetDriveMode --keep=Advertising_LED_2_SetDriveMode --
keep=Bootloading_LED_SetDriveMode --keep=Bootloader_Service_Activation_Read --
```

```
keep=Bootloader_Service_Activation_SetDriveMode --keep=Bootloader_Service_Activation_Write --keep=CyBle_StackInit --
keep=CyBle_Shutdown --keep=CyBle_SoftReset --keep=CyBle_EnterLPM --keep=CyBle_ExitLPM --
keep=CyBle_ProcessEvents --keep=CyBle_SetDeviceAddress --keep=CyBle_GetDeviceAddress --keep=CyBle_GetRssi --
keep=CyBle_GetTxPowerLevel --keep=CyBle_SetTxPowerLevel --keep=CyBle_GetBleClockCfgParam --
keep=CyBle_SetBleClockCfgParam --keep=CyBle_GenerateRandomNumber --keep=CyBle_SetCeLengthParam --
keep=CyBle_WriteAuthPayloadTimeout --keep=CyBle_ReadAuthPayloadTimeout --keep=CyBle_GetStackLibraryVersion --
keep=CyBle_GetBleSsState --keep=CyBle_AesEncrypt --keep=CyBle_AesCcmInit --keep=CyBle_AesCcmEncrypt --
keep=CyBle_AesCcmDecrypt --keep=CyBle_SetTxGainMode --keep=CyBle_SetRxGainMode --
keep=CyBle_GattGetMtuSize --keep=CyBle_ApplCallback --keep=CyBle_deviceAddress --keep=CyBle_sflashDeviceAddress --
keep=CyBle_discoveryModelInfo --keep=CyBle_scanRspData --keep=CyBle_discoveryData --keep=CyBle_discoveryParam --
keep=CyBle_Init --keep=CyBle_ServiceInit --keep=CyBle_Start --keep=CyBle_Stop --keep=CyBle_StoreBondingData --
keep=CyBle_GappStartAdvertisement --keep=CyBle_GappStopAdvertisement --keep=CyBle_ChangeAdDeviceAddress --
keep=CyBle_GapSetLocalName --keep=CyBle_GapGetLocalName --keep=CyBle_Get16ByPtr --keep=CyBle_Set16ByPtr --
keep=CyBle_bass --keep=CyBle_BasInit --keep=CyBle_BasRegisterAttrCallback --keep=CyBle_BassSetCharacteristicValue --
--keep=CyBle_BassGetCharacteristicValue --keep=CyBle_BassGetCharacteristicDescriptor --
keep=CyBle_BassWriteEventHandler --keep=CyBle_BassSendNotification --keep=CyBle_diss --keep=CyBle_DisInit --
keep=CyBle_DisRegisterAttrCallback --keep=CyBle_DissSetCharacteristicValue --keep=CyBle_DissGetCharacteristicValue --
--keep=CyBle_hidss --keep=CyBle_HidssInit --keep=CyBle_HidssRegisterAttrCallback --
keep=CyBle_HidssSetCharacteristicValue --keep=CyBle_HidssWriteEventHandler --keep=CyBle_HidssSendNotification --
--keep=CyBle_HidssOnDeviceConnected --keep=CyBle_scps --keep=CyBle_ScpsInit --
keep=CyBle_ScpsRegisterAttrCallback --keep=CyBle_ScpsSetCharacteristicValue --
keep=CyBle_ScpsGetCharacteristicValue --keep=CyBle_ScpsGetCharacteristicDescriptor --
keep=CyBle_ScpsWriteEventHandler --keep=CyBle_ScpsSendNotification --keep=CyBle_GattsEnableAttribute --apcs=/pic
--info unused
```

For the MDK compiler, the bootloadable project uses a custom linker script that provides placement information of APIs from the bootloader project. The same approach is used for the IAR.

## Compilation Script Description

A compilation batch script file (mk.bat in the Bootloader project) generates custom link files. This script file parses the \*.elf file generated from the bootloader project and generates a file for the bootloadable project linker with shared symbols placement. The script generates files into the \BLE\_OTA\_FixedStack\_Bootloadable.cydsn\LinkerScripts\ directory. The script consists of two sections:

- Setup
- Main

The user must check the correctness of all paths in the “Setup” section. The setup options are as follows:

- **LOADER\_PRJ\_NAME** – The name of the bootloader project.
- **LOADER\_PRJ\_PATH** – The path to the bootloader project.
- **LOADABLE\_PRJ\_NAME** – The name of the bootloadable project.
- **OUTPUT\_DIR** – The path to the directory where to put generated files.

- **BUILD\_OUTPUT\_DIR** – The path to the directory with the \*.elf file. There is a list of all possible options, choose only one, uncomment it, and comment some other option.
- **LOADER\_ELF\_FILE** – The full name with a path to the \*.elf file.
- **COMPILER** – The compiler identifier.
- **COMPILE\_OPTION** – The Debug/Release option.
- **KEEP\_TMP\_FILES** – The options to keep temporary generated by this script files.
- **UTILS\_NM** – The path to the Creator directory with the arm-none-eabi-nm.exe tool. Check if this path is correct.
- The names of the input files with the section and symbols names are defined in **SECTION\_LIST** and **SEARCHING\_SYMBOLS**.
- The names of the temporary files defined in **PARSED\_ELF**, **EXTRACTED\_BLE\_INFO**, **BLE\_INFO\_ALL**, and **BLE\_INFO\_ALL\_SORTED**.
- The main section checks the input, searches specified symbols in \*.elf files and generates an output for linker scripts.

**Note** The names of the bootloader project component to be exported to the bootloadable project must be aligned with the Symbol\_list.txt file. It is not recommended to change the default values. The script file may work in a wrong way if these names are misrecognized inside other names (e.g., BLE as part of TABLE). CyBle is the recommended name for the BLE component.

### Custom linker scripts limitations:

For project(s) that use custom linker scripts heap and stack size that was set in the PSoC Creator will not be applied. The size must be specified manually in corresponding linker scripts.

## Setup and Run Example Project

### Building Example Project

The following steps are described for compiler GCC that is shipped with PSoC Creator. If the MDK compiler is used, variable COMPILER should be changed to MDK\_COMPILER and it should be set to IAR\_COMPILER for IAR. **Note** The same compiler should be used for the bootloader and bootloadable projects.

For more information on changing the compiler from a default one (GCC), refer to section “Changing projects compiler”.

1. Build the *BLE\_OTA\_FixedStack\_Bootloader* example project. Check that the proper compiler toolchain is set in **Project -> Build Settings**.
2. **Optional** Add the *BLE\_OTA\_FixedStack\_Bootloadable* example project to the workspace).
  - a. Setup the mk.bat file. Open the mk.bat file in the text editor. Set the following variables (see the chain of file generation in the table below):

Variable name	Default value	Description
LOADER_PRJ_NAME	BLE_OTA_FixedStack_Bootloader	The Bootloader project's name. <b>Note</b> The <u>Bootloader project's name is different</u> . For each example project, PSoC Creator appends a number starting from 01 to the example project's name in a selected folder
LOADABLE_PRJ_NAME	BLE_OTA_FixedStack_Bootloadable	The Bootloadable project's name. <b>Note</b> The <u>Bootloadable project's name is different</u> . For each example project PSoC Creator appends a number starting from 01 to the example project's name in a selected folder.
LOADER_PRJ_PATH	"%~dp0..\\" that means Bootloader's sources folder.	The path to the Bootloader project. By default it takes the current path, but it can be set as the absolute path.
OUTPUT_DIR	%~dp0..\..\%LOADABLE_PRJ_NAME%. cydsn\LinkerScripts\ That means Bootloadable's \LinkerScripts\ folder	The path to the Bootloadable's sources, where the "LinkerScripts" folder is located. You should put the absolute path to the Bootloadable project here. <b>Note</b> The default path is valid if the Bootloader and Bootloadable projects are in the same workspace.
CREATOR_VERSION	4.1	A version of PSoC Creator. For example, 4.0, 4.1 or any other that matches the installed PSoC Creator version. This variable must contain a version of PSoC Creator installed. <b>Note</b> PSoC Creator Service Packs, Device Packs do not matter here. You need to specify only the version number.
CREATOR_LOCATION	%PROGRAM_FILES_PATH%\Cypress\ PSoC Creator	By default, this value should not be changed. This path needs to be modified only if the PSoC Creator is installed to a different location. In this case, a full path to the PSoC Creator installation must be specified.
COMPILER	GCC_COMPILER=ARM_GCC_541 COMPILER= %GCC_COMPILER%	Keeps the compiler identifier. COMPILER=%GCC_COMPILER% or COMPILER=%MDK_COMPILER% or COMPILER=%IAR_COMPILER%
COMPILE_OPTION	COMPILE_OPTION=%COMPILE_OPTI ON_DEBUG%	Keeps either the Debug or Release option. You can choose it from the following list: COMPILE_OPTION_DEBUG=Debug COMPILE_OPTION_RELEASE=Release
KEEP_TMP_FILES	KEEP_TMP_FILES=YES	If set to "YES", it keeps temporary files generated during a batch file run.



- b. Use the following link for detailed information on what a BATCH file is and how it works:

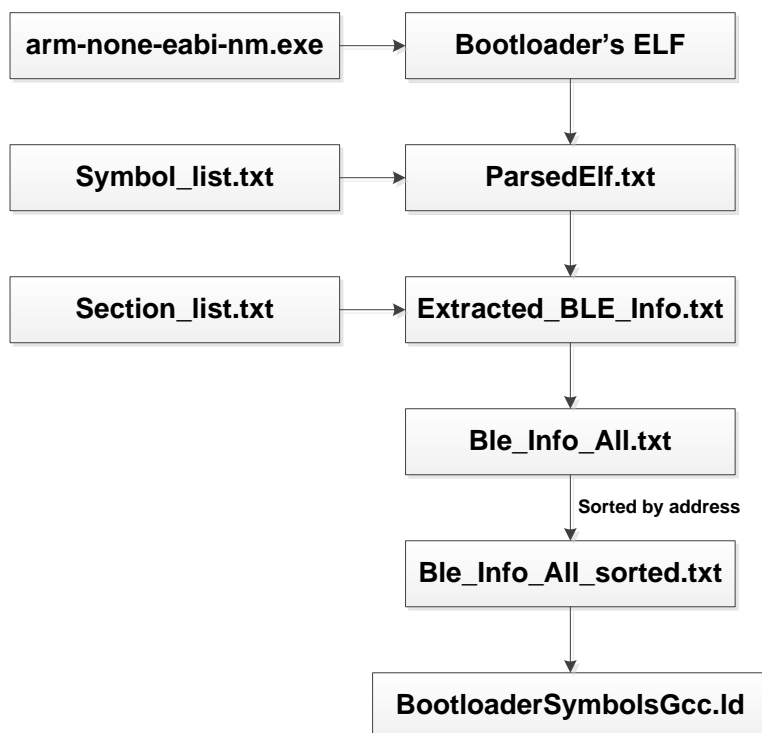
<http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/batch.msp?mfr=true>

The temporary files generated during a BATCH file run. **Note** They are available if variable **KEEP\_TMP\_FILES** is set to “YES”.

- “ParsedElf.txt” – The output file that contains extracted information from the bootloader’s ELF file in the following format:
- **<Address> <section symbol> <symbol name>**
- “Extracted\_BLE\_Info.txt” –The output file that contains filtered information about the symbols that are searched by the masks defined in “Symbol\_list.txt”. If you want some instance/component to be available in the Bootloader, you can add your own mask in that file. Those symbols names and their addresses are provided to Bootloadable project.
- “Ble\_Info\_All.txt” – The output file that contains filtered information from the “Extracted\_BLE\_Info.txt” file for certain required sections (T/t-text, B/b-bss, D/d-data, R/r-read-only, a-absolute). These sections are listed in the “Section\_list.txt” file.
- Ble\_Info\_All\_Sorted.txt – The output file that contains the final sorted by address information required for the Bootloadable project.

All symbol lists for GCC are generated from the content of this file.

Figure 8. Batch Script Workflow



- “Symbol\_list” – The input file permanently present at the \Scripts folder. It is not regenerated. It basically contains the list of symbols names masks, such as: “CyBle\_”, “LED\_”, “Bootloader\_Service\_Activation\_”, “WakeUp\_Interrupt\_” and so on.

Do not modify the compiler-specific symbols under comment “;GCC”. Also, do not delete or modify the “CyBle\_” symbol, otherwise the BLE component-related info will not be found. The search is not case-sensitive. We strongly recommend NOT modifying the BLE component name (CyBle), because it is hard to find only proper symbols with such names as BLE, because it takes also different “Table\_...” and “Enable\_...” symbols that are redundant. “CyBle” is quite unique.

Therefore, symbols with names that contain any of those masks are filtered from the “ParsedElf.txt” file in “Extracted\_BLE\_Info.txt”. If you want some instance/component to be available in the Bootloader, you can add your own mask in that file. Those symbols names and their addresses will be provided to the Bootloadable project.

- “Section\_list.txt” – The list of sections required for filtering (T/t-text, B/b-bss, D/d-data, R/r-read-only, a-absolute). Do not modify this file, unless you know what you are doing.

### 3. Run mk.bat file:

- a. Open the mk.bat directory in the **Explorer** or other file manager. ([Project path]\Scripts\)
- b. Double click the mk.bat file.

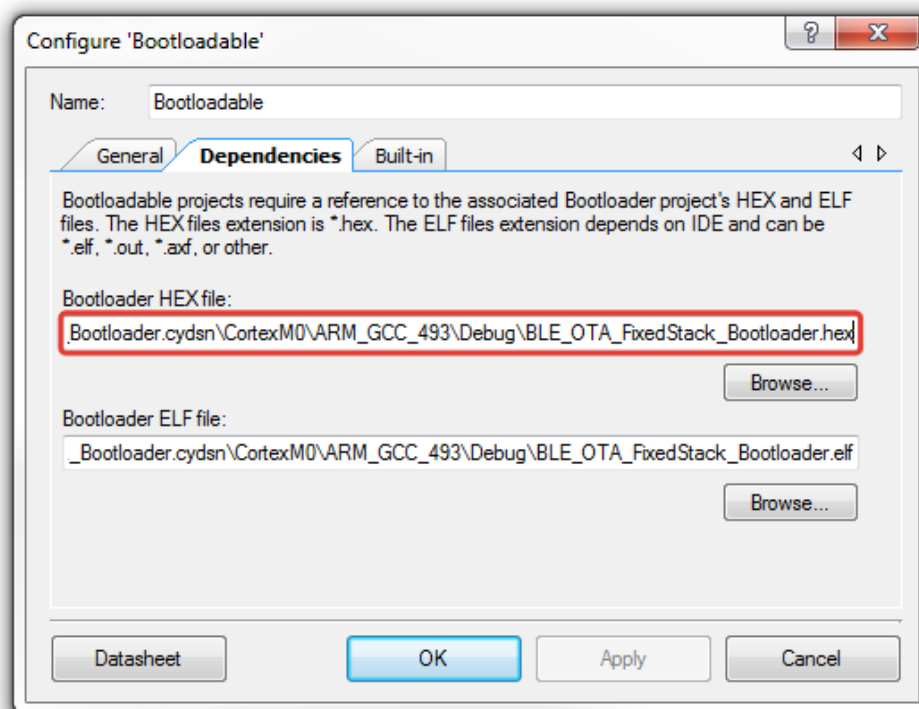
If everything is correct, it reports the following:

- Parsing the ELF file...
- Filtering exported symbols...
- Filtering by the section list...
- Sorting by the address...
- Generating output file(s) ...
- Press any key to continue ...

No error should be allowed on the output and a compiler-specific file to be created (**Note** Such a file is regenerated each time the mk.bat file runs):

- BootloaderSymbolsGcc.ld – The file with a list of the PROVIDE() directive for all exported symbols required for the GCC compiler.
  - BootloaderSymbolsMdk.txt – The exported symbol definition for the MDK compiler.
  - BootloaderSymbolsIar.icf – The exported symbol definitions for the IAR compiler.
4. Open the top design schematic of the *BLE\_OTA\_FixedStack\_Bootloadable* project. Specify the path to the bootloader project HEX and ELF files by double-clicking on the Bootloadable component and going to the **Dependencies** tab and link Bootloadable to the BLE\_OTA\_FixedStack\_Bootloader.hex file, as Figure 9 shows.

Figure 9. Bootloadable Component Configuration



#### 5. Program the BLE\_OTA\_FixedStack\_Bootloader project.

At this point, the CY8CKIT-042-BLE contains firmware that can receive new updates over-the-air. The LED3 flashes red.

## Upgrading Projects images

To perform any application update, a Bootloader project image should be running. If the current running project image is a Bootloadable project image, rebooting to the Bootloader project image can be done by a SW2 button press. While the device is running the Bootloader project image code, and is ready to receive Application project image updates (\*.cyacd files), the red LDE is blinking.

There is one software tool provided by Cypress that can send Stack or Application project image updates to the device - CySmart. It is external software to be downloaded and installed separately.

There are two hardware Dongles that communicate with the BLE devices and can be used to update the firmware:

- CY5670, with 128KB Flash chip, supports BLE 4.1.

- CY5677, with 256KB Flash chip, supports BLE 4.2. It supports higher transfer rates, up to 1Mbps.

Refer to the documentation that came with the dongle to figure out its type. Also, it can be received from CySmart.

### Upgrade Project images with CySmart

More detailed information about the CySmart usage can be found in CySmart User Guide (section 2.7 Updating Peripheral Device Firmware). You can download the CySmart tool and its user guide from here <http://cypress.com/cysmart>.

To upgrade the device firmware OTA follow:

1. Make sure the OTA device is running a Bootloader project image, and is ready to receive updates.
2. Connect BLE Dongle.
3. Open CySmart software, and select BLE Dongle.
4. Press the **Start Scan** button to start scanning for the peripheral device.
5. When the device is listed in the Discovered Device list, stop scanning by clicking the **Stop Scan** button.
6. Select the device from the Discovered Device list.
7. Click the **Update Firmware** button.
8. Select the firmware update type on the OTA firmware update window.

Firmware Update Type	Description
Application and Stack (Combined) update	The Application and Stack firmware co-exist in the same memory location. This option updates complete firmware.
Application only update	The Application and Stack firmware exist in independent memory locations. This option updates only the application firmware.
Application and Stack update	Application and Stack firmware exist in independent memory locations. Select this option to update first the stack firmware and then the Application firmware.

9. Click **Next**. In the next page, browse and select the new firmware image (\*.cyacd) file. It is located in the project folder (*[project folder]\CortexM0\[compiler name]*).
10. Click on the **Update** button.
11. Wait for the device firmware to be updated.

**Note** If the update fails at authentication, then check if the security settings in CySmart matches with the settings used in the BLE component of the Stack project. To change the security

settings in CySmart, click on the **Configure Master Settings** button in the tool and go to the **Security parameters** page.

## Expected Results

### LED Behavior Description

#### Bootloader project

Color	State	Description
Red	Blinking	Advertising
Red	Static	Bootloading
White (Red+Green+Blue)	Static	Low power mode

At a reset, when the Bootloader runs, the red LED is blinking, which means that the device is advertising. Once connected and an application image is sent from Bootloader Host Tool to the device to be updated, the red LED stops blinking and remains static. This means the device is bootloading.

If the LED color changes to white (all 3 LEDs are on – green, blue, and red), the device goes into the low power mode.

#### Bootloadable project

Color	State	Description
Green	Blinking	Advertising
Blue	Static	Disconnected
Cyan (Green+Blue)	Blinking	Disconnected + Advertising

At a reset, when the Application runs, the green LED is blinking, which means that the device is advertising. Once connected, it stops blinking. The blue LED ON indicates that the device is disconnected. The Cyan (2 LED – Green+Blue simultaneously) color means that the device is disconnected and advertising.

After the firmware update process is completed, the device should operate as the BLE\_HID\_Keyboard example. Refer to the BLE\_HID\_Keyboard example project datasheet for details.

If there are more changes to the *BLE\_OTA\_FixedStack\_Bootloadable* project, repeat all the steps described for OTA firmware update in the **Setup** and run the “Example Project” section.

## Using UART for Debugging

In this example projects, the UART is used to print various debug information (enabled by default).

File *options.h* contains define “DEBUG\_UART\_ENABLED”. It is set to “YES” and it provides extra debugging information in each of the bootloader or bootloadable projects. This slightly decreases the projects performance, but it provides extra debugging output to UART. If it is not needed – set it to NO.

A HyperTerminal program is required in the PC to receive debugging information. If you don't have a HyperTerminal program installed, download and install any serial port communication program. Freeware such as Bray's Terminal, Putty etc. are available on the web.

1. Connect the PC and kit with a USB cable.
2. Open the device manager program in your PC, find the COM port to which the kit is connected, and note the port number.
3. Open the HyperTerminal program and select the COM port to which the kit is connected.
4. Configure the Baud rate, Parity, Stop bits and Flow control information in the HyperTerminal configuration window. By default, the settings are: Baud rate – 57600, Parity – None, Stop bits – 1 and Flow control – XON/XOFF. These settings must match the configuration of the PSoC Creator UART component in the project.
5. Start communicating with the device as explained in the project description.

File *debug.h* contains macros used to print various types of data:

- `DBG_PRINT_TEXT(a)` - Prints a text string.
- `DBG_PRINT_DEC(a)` – Prints a decimal number.
- `DBG_PRINT_HEX(a)` – Prints a hexadecimal number.
- `DBG_PRINT_ARRAY(a,b)` – Prints **b** first elements of array **a**.
- `DBG_PRINTF(...)` – Prints the function macro.

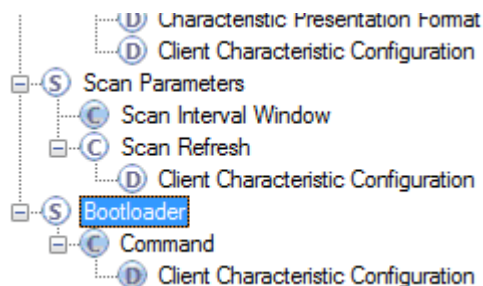
These macros print information to UART only if `DEBUG_UART_ENABLED` define is set to “YES”.

## Adding OTA Bootloader Support to another BLE Project using shared code

This section describes how to add the OTA shared bootloader capability to your own project with a BLE component.

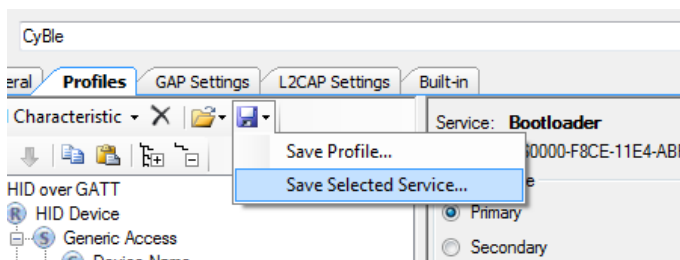
1. Open the BLE\_OTA\_FixedStack\_Bootloader example project.
2. Navigate to the project **TopDesign**, open the BLE component customizer by double clicking on the CyBle component and select the Bootloader Service Top node:

Figure 12. Bootloader Service Top Node



3. Now you can export the Bootloader Service into a file:

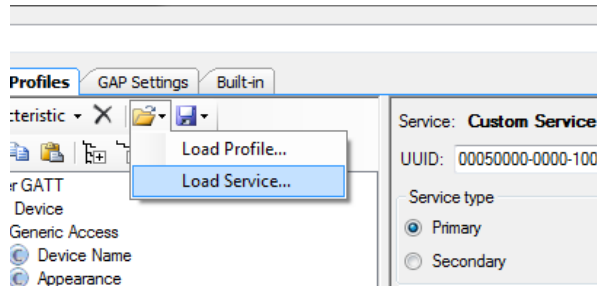
Figure 13. Exporting BLE Service Configuration



4. Once the BLE Bootloader Service configuration is saved, it can be imported to any project with a BLE component:

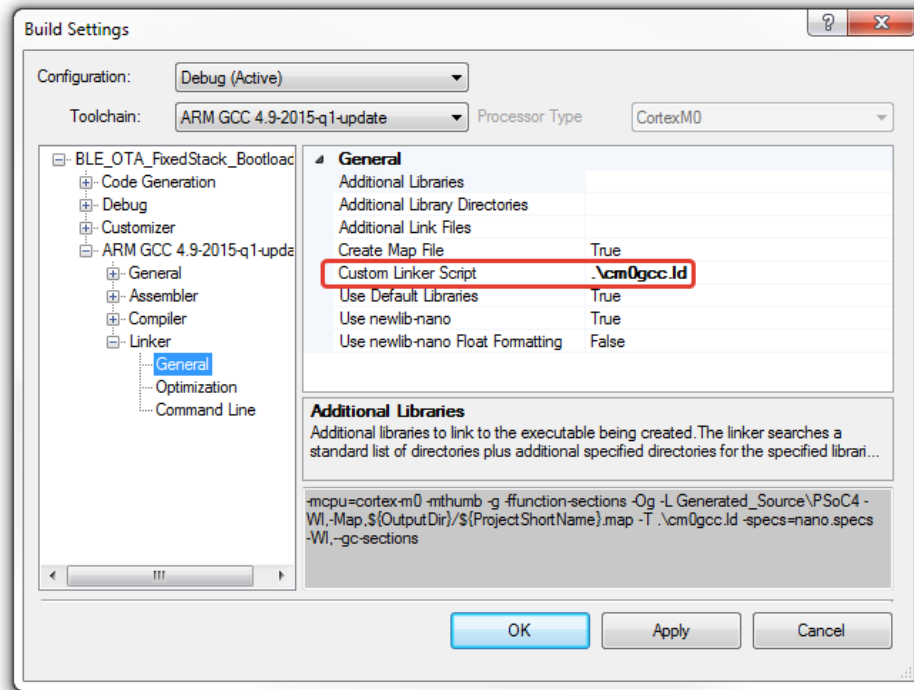


Figure 14. Loading Service Configuration



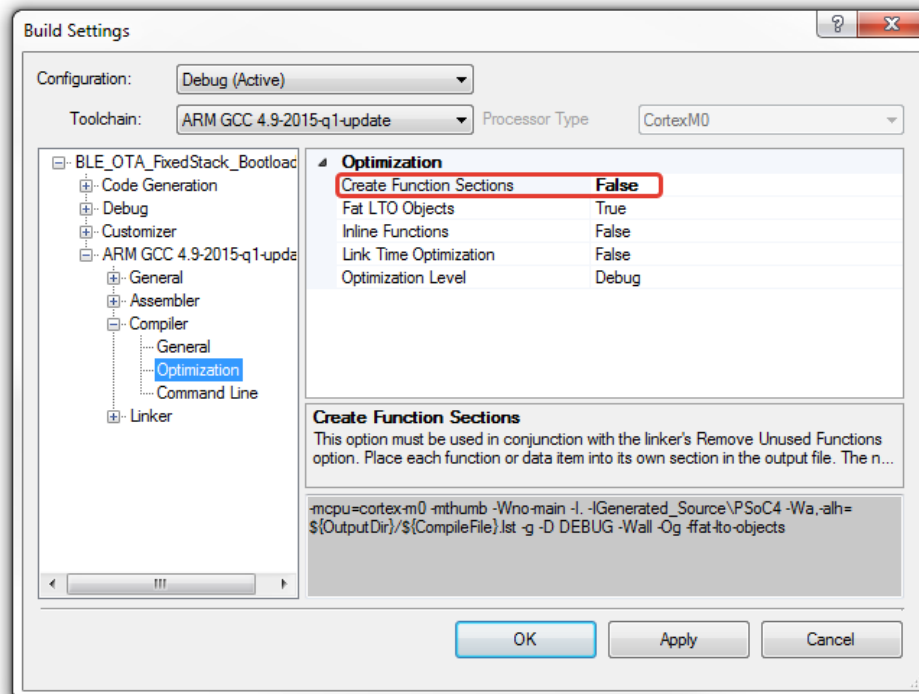
5. To enable the Bootloader Service to receive long packets, the MTU size of the BLE component does not matter.
6. You can copy a bootloader component symbol from the BLE Shared Memory Bootloader example project to your project. Change the project type to bootloader. This can be done by selecting the project, and selecting **Project > Build Settings** and changing the Application type to Bootloader.
7. You need to copy all the Application layer code from the project that you are converting to a new project and configure it to be bootloadable.
8. Copy all the linker files and scripts from example projects (BLE\_OTA\_FixedStack\_Bootloader/ BLE\_OTA\_FixedStack\_Bootloadable) to your bootloader/bootloadable projects respectively maintaining their placement in the project folders and change the projects names and paths in the mk.bat script.
9. Enable option **Custom Linker Script** in the bootloader project by selecting the copied linker script in the build settings of the bootloader project:

Figure 15. Custom Linker Script Configuration



10. Change the option of the compiler to generate functions sections:

Figure 16. Compilation settings:



11. Now the bootloader project can be compiled.
12. After the bootloader project compilation, batch script mk.bat should be executed. It analyzes the symbols placement in the bootloader image and provide them to the bootloadable project linker script.
13. You can compile your bootloadable project.
14. If there are some compilation errors related to the missing APIs/symbols, you should declare these APIs/symbols as external ones, as it is done in the files OTAMandatory.c/OTAMandatory.h. This is also true for typedefs/structures or enums (OTAMandatory.h).
15. Now all the changes made to the bootloadable project can be updated OTA. **Note** changes that are made to the bootloader project can be updated only via reprogramming the device.

## Functional description

The projects operation is slightly different for the GCC, MDK, and IAR compilers with the same approach. The Bootloader project works this way in any generic case. On the other hand, the Bootloadable project uses APIs from the bootloader project and, therefore initializes global and static variables for them by creating two separate segments in RAM – one for bootloader APIs and one for the bootloadable project code. Afterwards, bootloadable project calls either the function generated by the compiler or the custom initialization function to initialize the bootloader APIs RAM.

### GCC Compiler

For the GCC compiler, the bootloadable project uses the custom function to initialize the bootloader APIs RAM by passing the bootloader segments addresses to the bootloadable application and copying data from ROM to RAM.

### MDK compiler

For the MDK compiler, a similar approach is used, but instead of writing a function to initialize RAM, the function already existing in the bootloader project is used. To be able to return to the bootloadable project, the return point address is passed through the not initialized segment of RAM.

### IAR compiler

The IAR compiler uses roughly the same approach, but RAM separation should be done manually. After the bootloader project is compiled, the amount of used RAM should be compared

to one, specified in row 48 of the IAR linker script in the bootloadable project. If the size is bigger, increase the number specified in the mentioned line.

## Changing projects compiler

Due to the specific implementation of the projects, the compiler should be the same for the bootloader and bootloadable projects. By default, the projects are configured to work with the GCC compiler but this can be changed.

### MDK compiler

To use the example project with the MDK compiler:

1. Change the compiler to MDK Generic in the project build settings.
2. Open file mk.bat with Notepad or any other text editor. It is stored in the **Scripts** subfolder of bootloader project.
3. Find the line that contains setting COMPILER variable and change it to MDK\_COMPILER:

```

33  ::
34  ::  Compiler choice. Currently only GCC 4.9.3, MDK and IAR compilers are supported.
35  ::
36  set GCC_COMPILER=ARM_GCC_493
37  set IAR_COMPILER=ARM_IAR_Generic
38  set MDK_COMPILER=ARM_MDK_Generic
39
40  set COMPILER=%GCC_COMPILER%
```

**Note** The symbol “%” should be present on both sides of the value.

4. Build the bootloder project.
5. Verify the path to the bootloader \*.elf file as it is described in section “Setup and Run Example Project”.
6. Run the mk.bat script.
7. Compile the bootloadable project and program it.

### IAR Compiler

To use the IAR compiler, export both bootloader and bootloadable projects to IAR IDE as described in PSoC® Creator™ User Guide with the two exceptions:

- The linker script for the bootloadable project. You should use the one that is provided in the **LinkerScripts** folder of the bootloadable project.
- Include path for the bootloadable project. See [IAR Compiler Settings](#).

After the projects are exported and configured, verify that the RAM segment for the bootloader is big enough by getting the amount of RAM used in the bootloader project and comparing it to the size of the BTLDRRAM block defined in the IAR linker script:

```
48  define block BTLDRRAM    with alignment = 8, size = 0x2AFF { readwrite section .bootloaderram };
```

If the amount of RAM used by the bootloader is bigger, the size of the section should be increased, so that the RAM left in the bootloadable was big enough to contain the heap and stack.

Also, the mk.bat file should be modified:

Replace the LOADER\_ELF\_FILE variable value with the path to the \*.out file that is generated by IAR after building the bootloader project:

```
58  :: Bootloader's *.elf file path.
59  set LOADER_ELF_FILE=%LOADER_PRJ_PATH%BUILD_OUTPUT_DIR%LOADER_PRJ_NAME%.elf
```

to something like:

```
58  :: Bootloader's *.elf file path.
59  set LOADER_ELF_FILE=C:\Users\User\Desktop\BLE_Shared_Memory_Bootloader01\BLE_Shared_Memory_Bootloadable01.
    cydsn\Debug\Exe\BLE_Shared_Memory_Bootloadable01.out
```

Also, in the batch file that is located in subfolder **Export** of the bootloadable project (**prebuild.bat**) you should replace path to the \*.elf file that is generated in PSoC Creator with the path to the \*.out file of the bootloader project. The elf tool requires including the correct bootloader image into the final \*.hex image of bootloadable. It is required only if you plan to program the bootloadable \*.hex file into the device and it is not required for generation of the correct \*.cyacd file.

## Configuring projects for other Cypress BLE devices

BLE OTA Fixed Stack example projects use custom linker scripts so PSoC Creator will not be able to configure linker scripts to provide correct linking and placement. This section describes the steps to be done if you are not using CY8C4247LQI-BL483 or CYBL10563-56LQXI devices or devices that have different sizes of the RAM or ROM memory.

The following sections describe linker scripts modification after you have changed the device in PSoC Creator.

### Steps for GCC Compiler

For the GCC compiler changes are required for both bootloader and bootloadable linker scripts:

## BLE OTA Fixed Stack Bootloader and Bootloadable

```
21 MEMORY
22 {
23     rom (rx) : ORIGIN = 0x0, LENGTH = 131072
24     ram (rwx) : ORIGIN = 0x20000000, LENGTH = 16384
25 }
26
27
28 CY_APPL_ORIGIN      = 0;
29 CY_FLASH_ROW_SIZE  = 128;
30 CY_APPL_NUM        = 1;
31 CY_METADATA_SIZE   = 64;
32
```

1. Change the device flash memory size.
2. Change the device RAM size.
3. Change the device row size.

Linker scripts are located in **.\*cm0gcc.ld*** for the bootloader project and in **.\*LinkerScripts\cm0gcc.ld*** for the bootloadable project.

If you are not sure in values that are to be entered for your device, refer to the values that are in linker script generated by PSoC Creator even if it is not used. It is located in the folder: **%PROJECT\_DIR%\Generated\_Source\PSoC4\cy\_boot** under the name ***cm0gcc.ld***.

## Steps for MDK compiler

For the MDK compiler, only the linker script of the bootloadable project is changed:

```

33 | #include "BootloaderSymbols.h"
34 |
35 | #define CY_FLASH_SIZE 131072
36 | #define CY_FLASH_ROW_SIZE 128
37 | #define CY_METADATA_SIZE 64

```

1. Change the device flash memory size.
2. Change the device row size.

And a bit further:

```

91 |     ARM_LIB_HEAP (0x20000000 + 16384 - 0x400 - 0x1000) EMPTY 0x400
92 |     {
93 |     }
94 |
95 |     ARM_LIB_STACK (0x20000000 + 16384) EMPTY -0x1000
96 |     {
97 |     }

```

3. Change the device RAM size (both values).

The linker script is located in folder **.LinkerScripts\Cm0Mdk.scats**

If you are not sure in values to be entered for your device, refer to the values that are in the linker script generated by PSoC Creator even if it is not used after project rebuild. It is located in the folder: **%PROJECT\_DIR%\Generated\_Source\PSoc4\cy\_boot** under the name **Cm0Mdk.scats**.

## Steps for IAR Compiler

If you use the IAR, the linker script of the bootloader project is safe to use one that is generated by PSoC Creator after the device was changed in PSoC Creator. But the bootloadable project linker script must be modified:

```

6  define symbol __ICFEDIT_intvec_start__ = 0x00000000;
7  /*-Memory Regions-*/
8  define symbol __ICFEDIT_region_ROM_start__ = 0x0;
9  define symbol __ICFEDIT_region_ROM_end__ = 131072 - 1;
10 define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
11 define symbol __ICFEDIT_region_RAM_end__ = 0x20000000 + 16384 - 1;
12 /*-Sizes-*/
13 define symbol __ICFEDIT_size_cstack__ = 0x0800;
14 define symbol __ICFEDIT_size_heap__ = 0x400;
15 /**** End of ICF editor section. ###ICF###*/
16
17
18 /***** Definitions *****/
19 define symbol CY_FLASH_SIZE = 131072;
20 define symbol CY_APPL_ORIGIN = 0;
21 define symbol CY_FLASH_ROW_SIZE = 128;
22 define symbol CY_APPL_LOADABLE = 1;
23 define symbol CY_APPL_LOADER = 0;
24 define symbol CY_APPL_NUM = 1;
25 define symbol CY_METADATA_SIZE = 64;

```

1. Change the device flash memory size.
2. Change the device RAM size.
3. Change the device row size.

The linker script is located in folder **.\LinkerScripts\Cm0lar.icf**

If you are not sure in values to be entered for your device, refer to the values that are in the linker script generated by PSoC Creator even if it is not used after project rebuild. It is located in the folder: **%PROJECT\_DIR%\Generated\_Source\PSoC4\cy\_boot** under the name **Cm0lar.icf**.

## Updating BLE component to other version

The following steps are required to update the bootloader component version of the BLE component:

1. Ensure you're using the desired version of the BLE component in the bootloader project.
2. Rebuild the bootloader project.
3. Locate the CYBLE\_EVT\_T enum declaration in the bootloadable project (for the BLE\_OTA\_FixedStack\_Bootloadable example project this is **.\OTAMandatory.h** starting from line 1539) and replace it with the CYBLE\_EVT\_T enum declaration generated in the bootloader project (BLE\_OTA\_FixedStack\_Bootloader example project this is **.\Generated\_Source\PSoC4\BLE\_eventHandler.h** starting from line 146).
4. Run the mk.bat script located in the bootloader project in the Scripts subfolder.



5. After updating the BLE component in the bootloader project, program the new bootloader image to the device;

© Cypress Semiconductor Corporation, 2016. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.