

BLE Upgradable Stack Example Projects

1.20

Features

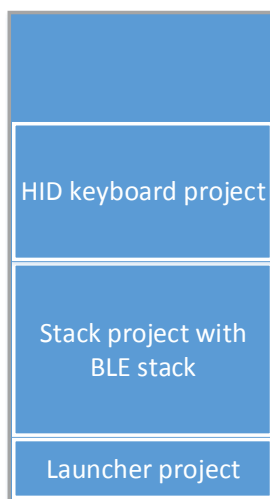
- Over-The-Air (OTA) firmware update
- Upgradable BLE Stack shared between two projects
- Independent BLE Component configuration in two projects
- Bonding data maintained across bootloading sessions
- HID keyboard

General Description

This example shows how to implement an upgradable Application project (HID keyboard) and upgradable Stack project with the BLE Stack. In addition, the Application project uses the BLE Stack from the Stack project.

The scheme of simplified memory organization is in Figure 1. The details are available in the [“Architecture”](#) section.

Figure 1. OTA Shared Memory Workspace



The three projects functions are as follows:

1. Launcher project

The purpose of the project is to:

- Start either the Stack or Application project image
- Copy the Stack project image from a temporary location to the persistent location (just after the Launcher project image).

See [“Launcher Project Configuration” for more detail.](#)

2. Stack project (contains the BLE Stack)

The purpose of the project is to:

- Host the BLE Stack
- Upgrade the Application project image
- Upgrade the Stack project image itself
- Receive updates for the Stack project image itself or for the Application project image

See [“Stack Project Configuration” for more detail.](#)

3. HID keyboard project (Application project)

This project demonstrates:

- Pressing the keyboard in the boot and protocol mode
- Handling a suspend event from the central device
- Entering the low-power mode when suspended in a form of an upgradable project

Note The Stack project is the bootloader for the HID keyboard project and the bootloadable for the Launcher project.

See [“HID Keyboard Project Configuration”](#) for more detail.

Development Kit Configuration

This example project is designed to run on the Cypress CY8CKIT-042-BLE kit with the CY8CKIT-143 PSoC 4 BLE 256KB Module. 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> and <http://www.cypress.com/documentation/development-kitsboards/cy8ckit-143-psoc-4-ble-256kb-module>.

The kit must be powered with 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.

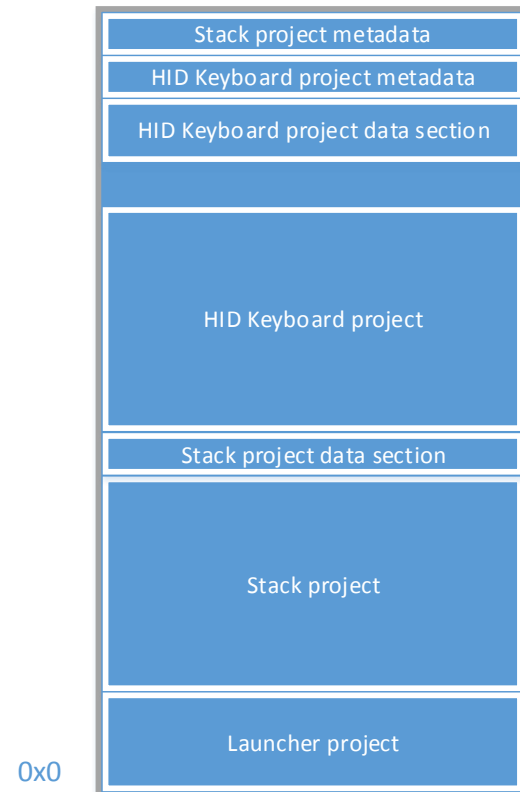
If the kit is powered with a different voltage, the project settings must be updated appropriately.

Architecture

General

Figure 2. **Flash Memory Structure** shows detailed organization of the flash memory.

Figure 2. Flash Memory Structure



This project images placement is unique for this use case. It is similar to the Dual-app bootloader/bootloadables but contains a few key differences:

1. The Application project image (HID Keyboard) is placed after the Stack project image.
2. The Launcher project image has a bootloader component but does not have communication for it.
3. The Stack project image contains both Bootloader and Bootloadable Components.
4. The Application project image references the Stack project image as its bootloader.

Code Sharing

Code sharing is a new PSoC Creator feature that allows a bootloader to share its communication component with the bootloadable project. This applies only to big communication components, such as the BLE.

Code sharing includes two parts:

1. Exporting code: the compiler and/or linker configuration that will force the compiler/linker to keep the communication component code even if its functionality is not fully used in the bootloader project itself.
2. Extracting flash symbols placement in the shared communication component and providing definitions of these symbols to the bootloadable project. The symbols here are the functions or data arrays required for a particular communication component. In this example - the BLE.

Checksum Excluding

During a bootloading process, a bootloader checks for the integrity of the application code residing in the flash memory by calculating its checksum and verifying it against the originally stored checksum. However, some configurations of the BLE Component require saving some data (Bonding data) to a flash memory area, which is a part of the application. This bonding data changes whenever PSoC BLE device is paired to other devices. When bonding data changes, the stored check sum of the application image is no more valid. In this case, the bootloader will halt application loading as soon as it detects mismatch in the calculated and original checksum values.

To prevent changing the checksum of the application code, bonding data is placed in the checksum excluded area of the flash memory (named "Stack project" and "Application project" data sections in **Figure 2. Flash Memory Structure**).

Note In the Stack project configuration, saving bonding data is disabled by default.

The BLE component gives information about the checksum exclusion area size required for bonding data to the Bootloadable component that adds this value to the value configured in the component customizer. Because of that, the Bootloadable component customizer's parameter named "Checksum exclude section size (bytes)" does not include the size for bonding data.

More info about the bootloading process and checksum exclusion can be found in the Bootloader or Bootloadable component's datasheet.

RAM Sharing

In the Stack project, the BLE Stack requires a bit of the RAM memory that must be allocated statically. So, for the Application project, custom linker scripts are used providing some space reserved for the BLE Stack operation. This results in reducing the amount of the RAM memory available for the Application project. On average, the RAM size is reduced by 4K.

Updating Projects

Application project image update

The Application project image update procedure does not differ from the normal bootloadable project image update: switching to the bootloader (Stack project image) and receiving a bootloadable image. The only exception is that the “Application Project Data” section is not replaced during this update.

Stack project image update

To update the Stack project image, different architecture is used: PSoC Creator converts a file used for the update so that a new image will be placed into the temporary location: the Application project image placement is used for this purpose. When receiving an updated image is completed, the Stack project initiates the device software reset. After the software reset, the Launcher project image is launched and it detects the image placed to the temporary location and copies it to its appropriate place. After that the Stack project image is launched.

At this point, the Application project image is damaged, so it must be received as well.

Launcher Project Configuration

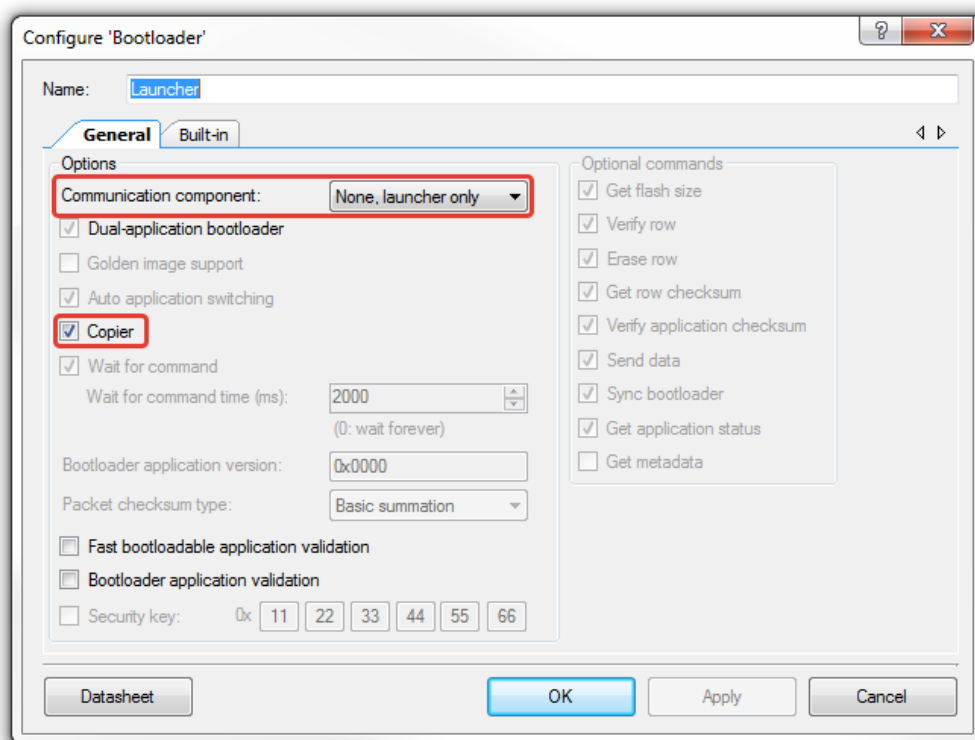
The Launcher project has simple configuration: the Bootloader component and UART for printing information to the UART console.

It is required to make Launcher project's image size as small as possible. Stack project's image size is limited and has to be optimized, so optimizing Launcher allows more space to be available for the Stack project image.

Bootloader

The Bootloader component is configured for the Launcher mode with the copier functionality enabled:

Figure 3. Bootloader Component Configuration in Launcher Project



The Copier functionality allows copying the image from the temporary location.

Software Transmit UART

This component is used for printing debug information.

Compiler Settings

If **ARM GCC** compiler is used, then check that Optimization Level is set to **Size** (-Os) in both Debug and Release builds:

1. Right-click the Stack project in **Workspace Explorer**.
2. Click **Build Settings**.
3. Select **Configuration - Debug**.
4. Expand **Compiler** under the selected compiler:

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

Section	Option	Value
Optimization	Optimization Level	Size

Export to IAR

In IAR IDE open Project options, *C/C++ Compiler Category, Optimizations* tab and set Level to either *Medium*, or *High/Size*.

Stack Project Configuration

The Stack project combines both the Bootloader and Bootloadable Components. In addition, it has the BLE Component with the **Stack Only** option enabled.

Stack project image size is limited and BLE component code takes most part of it, so this project has to be optimized even in debug builds. With the new BLE component versions the optimizations have to be even more aggressive, as this component adds new features which may cause code size growth if not optimized out.

Compiler Settings

The same compiler optimization settings as for Launcher project are required for this project too, see subsection [Compiler Settings](#) of the section [Launcher Project Configuration](#).

Linker Settings

If **ARM MDK Generic** compiler is used, then check the correctness of the project linker settings:

1. Right-click the Stack project in **Workspace Explorer**.
2. Click **Build Settings**.
3. Expand **Linker** under the selected compiler:

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

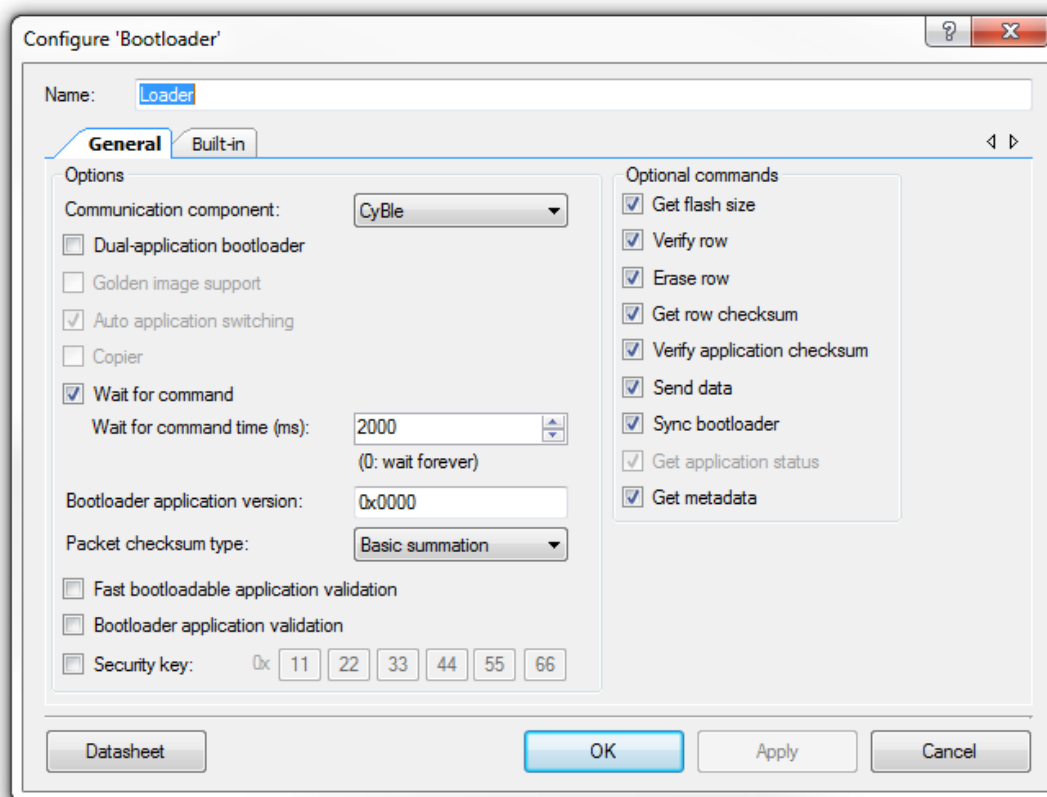
Section	Option	Value
General	Use MicroLIB	False

Note MicroLib has compilation limitations preventing the code sharing usage. So, MicroLib cannot be used neither in the BLE OTA Stack nor in BLE OTA Application projects, but can be used in the Launcher project as it does not use BLE OTA.

Bootloader Component

The Bootloader Component in this project is not involved in the “boot” process, it only for loading data via a communication component:

Figure 4. Bootloader Component Configuration in Stack Project



The BLE component is selected as a communication component.

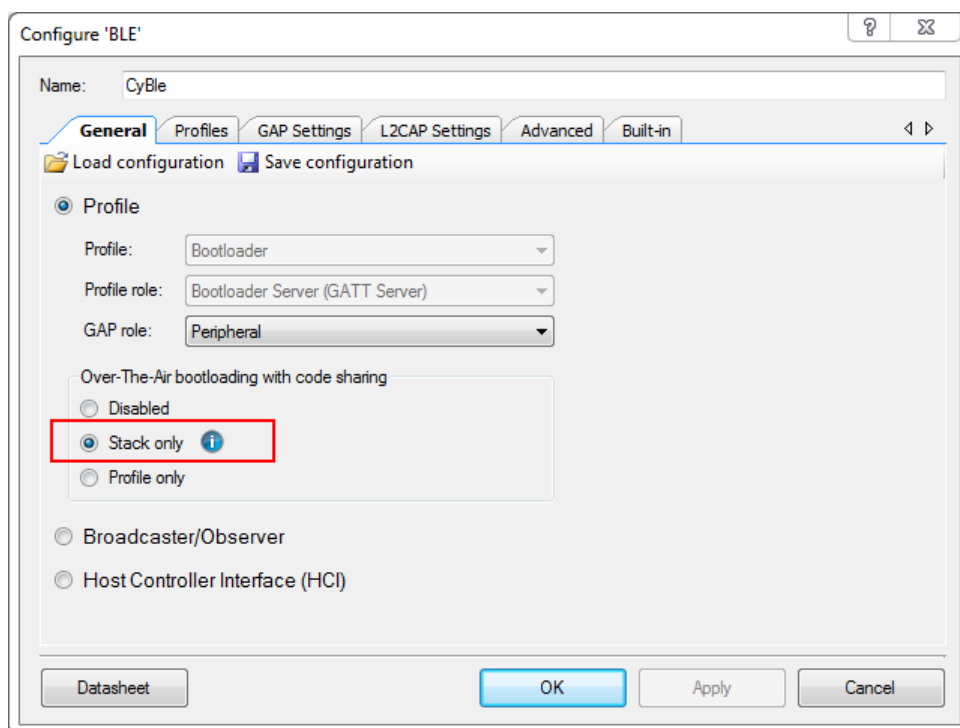
Bootloadable Component

The Bootloadable component has the default setting, except for the **Dependencies** tab to be configured to point to the Launcher project.

BLE Component

The BLE Component is configured to have the Stack-only mode:

Figure 5. BLE Component Configuration in Stack Project



Note The Stack-only mode requires additional heap memory. For the component configuration in this example project, it is about 4.5K to be added to the system heap size. This addition causes high system heap memory in the Stack project settings:

Figure 6. Stack Project Heap Settings

Configuration		
Device Configuration Mode	Compressed	▼
Unused Bonded IO	Allow but warn	▼
Heap Size (bytes)	0x2500	
Stack Size (bytes)	0x0800	
Include CMSIS Core Peripheral Library Files	<input checked="" type="checkbox"/>	

Do not reduce this value unless you know what you are doing.

The Stack-only mode of the BLE component provides only the bootloader service. To maximize the transfer speed, an attribute MTU value is set to 300 bytes.

Note The Stack project image performs 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 lasts longer. A flash write takes about 10 ms

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

Software Transmit UART

This component is used for printing debug information.

HID Keyboard Project Configuration

The Application project is a bootloadable project that has the BLE Component and Bootloadable Component.

Linker Settings

If the **ARM MDK Generic** compiler is used, then check the correctness of the project linker settings:

4. Right-click on the HID Keyboard project in **Workspace Explorer**.
5. Click **Build Settings**.
6. Expand **Linker** under the selected compiler:

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

Section	Option	Value
General	Use MicroLIB	False

Note MicroLib has compilation limitations preventing the code sharing usage. So, MircoLib cannot be used neither in BLE OTA Stack nor in BLE OTA Application projects, but can be used in the Launcher project as it does not use BLE OTA.

Bootloadable Component

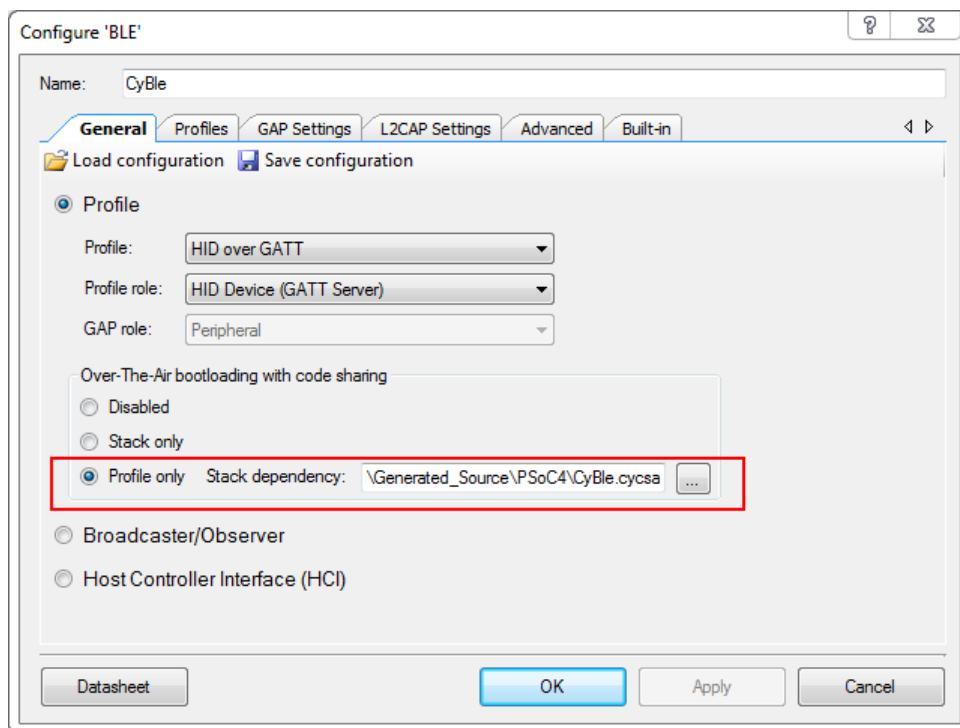
The Bootloadable Component has the default configuration. The bootloader dependency has to be configured to point to the Stack project image – it is a bootloader for the Application project image.

BLE Component

The BLE component is configured to use the code import. This mode is called “Profile Only” and implies that the bootloader project referenced for this particular bootloadable has the BLE

Component in the Stack-only mode and the path to the *.cycsa file that was generated in the bootloader project is referenced in the Component customizer:

Figure 7. BLE Component Configuration in Application Project



Other BLE component settings match the BLE HID keyboard example project.

Software Transmit UART

This component is used for printing debug information.

Custom Linker Script

The application project uses the custom linker scripts. For **ARM GCC Compiler** – file `.\LinkerScripts\cm0gcc.ld`, for **ARM MDK Compiler** – file `.\LinkerScripts\Cm0Mdk.scats` and for **IAR ARM Compiler** – file `.\LinkerScripts\Cm0Iar.icf`.

In these files, the checksum exclude section size (CY_CHECKSUM_EXCLUDE_SIZE) is defined as a constant with some value. This value can be changed when a BLE component version or configuration is changed, so the custom linker script must be updated when the BLE component is updated.

To update the custom linker script:

1. Find the generated linker script file with the same name in location:
.\Generated_Source\PSoC4\ , e.g. for ARM GCC Compiler this file is
.\Generated_Soucre\PSoC4\cm0gcc.ld .
2. Find the line with CY_CHECKSUM_EXCLUDE_SIZE defined in the generated linker script file.
3. Copy that line to your custom linker script file, where CY_CHECKSUM_EXCLUDE_SIZE is defined.

GCC compiler linker script

The GCC linker script has the only change compared to the generated script. At the start of the RAM, there is a separate segment for the BLE stack static data:

Figure 8. GCC Linker Script Modifications

218	218	
	219	BOOTLOADER_RAM_SIZE = (Bootloader__bss_end__ - 0x20000000);
	220	.bootloader_data (NOLOAD) : ALIGN(8)
	221	{
	222	. += (BOOTLOADER_RAM_SIZE);
	223	}
	224	
219	225	.data : ALIGN(8)

Note Make sure the lines where constant CY_CHECKSUM_EXCLUDE_SIZE is defined are equal in both generated and custom linker scripts.

MDK compiler linker script

The MDK compiler linker script in this example project has the same modifications as the GCC linker script, but in the MDK compiler, the start of RAM available for this project is moved further from the physical start of RAM. This is done for the project compiled with the MDK compiler to have a separate portion of RAM for the Stack static data.

Figure 9. MDK Linker Script Modifications

32	32	;*****
33		#include "cyfitter.h"
34		#include "cycodeshareimport.h"
	33	#include "../Generated_Source\PSoC4\cyfitter.h"
	34	#include "../Generated_Source\PSoC4\cycodeshareimport.h"
35	35	
113	113	
114		NOINIT_DATA +0 UNINIT
	114	NOINIT_DATA Bootloader_Image\$\$DATA\$\$ZI\$\$Limit UNINIT
115	115	{

Note Make sure the lines where constant CY_CHECKSUM_EXCLUDE_SIZE is defined are equal in both generated and custom linker scripts.

IAR compiler linker script

The IAR linker script has the same modifications. The difference is in line #55: you have to manually set the size of RAM occupied by the Stack project without heap and stack.

Figure 10. IAR Linker Script Modifications

```

33 33
34 include "..\Generated_Source\PSoC4\cybootloader.icf";
34 include "cybootloader.icf";
35 35 if (!CY_APPL_LOADABLE) {
36 36     define symbol CYDEV_BTLDLR_SIZE = 0;
37 37
38 38     /* The first 0x100 Flash bytes become unavailable right after remapping of the vector table to RAM.
39 39      * This space should be used for .romvectors section.
40 40      */
41 41     define block ROMVEC with size = 0x100 {readonly section .romvectors};
42 42
43 43     define block APPL with fixed order {block ROMVEC, section .psocinit, readonly};
44 44 } else {
45 45     define block APPL with fixed order {readonly section .romvectors, section .psocinit, readonly};
46 46 }
47 47
48 48 define memory mem with size = 4G;
49 49 define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__];
50 50 define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__];
51 51
52 52 define block CSTACK with alignment = 8, size = __ICFEDIT_size_cstack__ { };
53 53 define block HEAP with alignment = 8, size = __ICFEDIT_size_heap__ { };
54 54 define block HSTACK {block HEAP, last block CSTACK};
55 55 define block BTLDLRAM with alignment = 8, size = 0x2000 { readwrite section .bootloaderram };
56 55
57 57 define block RAMVEC with fixed order {readwrite section .ramvectors, readwrite section .bootloaderruntype, block BTLDLRAM };
56 define block RAMVEC with fixed order {readwrite section .ramvectors, readwrite section .bootloaderruntype};
58 57

```

...

```

104 103
105 105 keep { block BTLDLRAM,
106 106     section .cybootloader,
104 104 keep { section .cybootloader,
107 105     section .cyloadermeta,
108 106     section .cyloadablemeta,
109 107     section .cy_checksum_exclude,
110 108     section .cyflashprotect,
111 109     section .cymeta,
112 110     section .cychipprotect };

```

In most cases, the value set in the example projects is sufficient. Modify this value if the Stack project has changed.

Note The value in **Figure 10** might not match the one in the linker script.

Note Make sure the lines where constant CY_CHECKSUM_EXCLUDE_SIZE is defined are equal in both generated and custom linker scripts.

Custom linker scripts limitations

The biggest limitation is that changes to the system stack and heap sizes will have no effect on the custom linker script. If the size of the system heap or stack is changed, the custom linker scripts must be adjusted:

- GCC linker script

To change the system heap size, change line#265 of the GCC linker script:

Figure 11. GCC Linker Script System Heap Size

```
.heap (NOLOAD) :
{
    . = _end;
    . += 0x400;
    __cy_heap_limit = .;
} >ram
```

And the system stack size is configured in lines 69, 265, 272:

Figure 12. GCC Linker Script System Stack Size

```
PROVIDE(__cy_heap_end = __cy_stack - 0x0800);
```

```
.stack (__cy_stack - 0x0800) (NOLOAD) :
{
    __cy_stack_limit = .;
    . += 0x0800;
} >ram
```

The linker script is located in the LinkerScripts folder of the Application project.

- MDK linker script

In the MDK compiler, the size of the system heap is set in line124 of the custom linker script:

Figure 13. MDK Compiler System Heap Size

```
ARM_LIB_HEAP (0x20000000 + 32768 - 0x400 - 0x0800) EMPTY 0x400
```

The system stack size is configured in line 128 of the custom linker script:

Figure 14. MDK Compiler System Stack Size.\

```
ARM_LIB_STACK (0x20000000 + 32768) EMPTY -0x0800
```

- IAR linker script

For IAR system, the heap and/or stack size can be configured using IAR project settings.

Setup and Run Example Project

By default, the Launcher, Stack, and Application projects are expected to be located in the same workspace. However, the user can save the projects in any location. This document describes the case when all three projects are located in the same workspace.

The following steps are described for the GCC compiler shipped with PSoC Creator. These steps are applicable for the ARM MDK compiler as well.

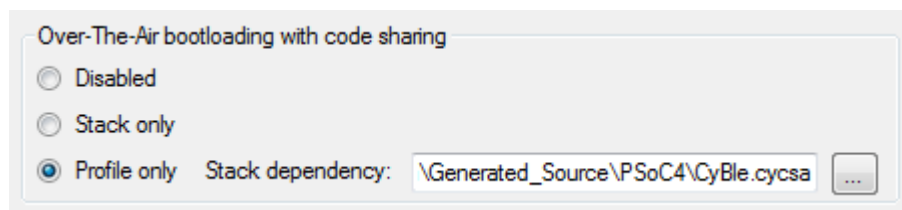
For more information on changing the compiler from a default one (GCC), refer to section [“Changing Projects Compiler”](#).

1. Build the Launcher project.
2. Add the BLE Upgradable stack example project to the workspace.
3. Change the path to the bootloader *.hex and *.elf files in the Bootloadable Component customizer to point to the files generated when the Launcher project was compiled. This file is located in the Launcher project folder: `CortexM0\%compiler name%\%compilation mode%` (for example, in the GCC compiler: `CortexM0\ARM_GCC_541\Debug\`).
4. Build the Stack project.

Note At this point, when the Stack project is compiled, the resulting *.hex file contains the Launcher project and the BLE Stack with the bootloader functionality. This means that if this file is programmed, the device will be in the update mode waiting for the Application project uploading.

5. Add the BLE Upgradable Stack HID Keyboard example project to the workspace.
6. Open the Bootloadable Component customizer and on the **Dependency** tab change the dependency to the bootloader *.hex/*.elf by the path to the *.hex and *.elf file that was generated after step #4.
7. Apply the changes and open the BLE component customizer. Make sure that the path to the *.cydsa file is valid:

Figure 15. BLE Component Path to Code Sharing File



This file was generated during step #4 and is located in the Stack project folder: `Generated_Source\\PSoC4\\`

Note The compilers used for the Stack project compilation must match the compiler used for the Application project.

8. Build the BLE Upgradable Stack HID Keyboard example project.
9. Program the BLE Upgradable Stack HID Keyboard example project and optionally, connect to debug UART.

Changing Projects Compiler

Both The Stack project and Application project must be compiled with the same compiler. This includes changes in the bootloadable component customizer of the Application project – an appropriate bootloader must be selected.

Note Only GCC MDK and IAR compilers are supported.

Exporting projects to IAR Embedded Workbench

If you want to export projects to the IAR, there is a slight difference from the general project export procedure:

1. Add the **BLE_OTA_UpgradableStackExample_Launcher** project to the workspace and compile it.
2. (Optional) Export it to the IAR and compile it there. Configure the project properly, see [Compiler Settings](#).
3. Add to workspace **BLE_OTA_UpgradableStackExample_Stack** and configure **Bootloader** Component paths to the bootloader: it should be either paths to images compiler compiled by PSoC Creator or compiled by IAR depending on #2: if it was done or not.
4. Export the Stack project to the IAR and compile it there. Configure the project properly, see [Compiler Settings](#).
5. Add to workspace **BLE_OTA_UpgradableStack_HID_Keyboard**.
6. Configure the Bootloadable component to use images that were compiled with the IAR in Step #4.
7. Add a reference to the *.cydsa file from the Stack project.
8. Export the project to the IAR.
9. Change the linker script in the IAR to the one located in the **BLE_OTA_UpgradableStack_HID_Keyboard** folder: **\LinkerScripts\Cm0Iar.icf**.
10. Compile the **BLE_OTA_UpgradableStack_HID_Keyboard** project in the IAR.
11. Program the *.hex file.

- Note**
1. It is necessary to have both Stack and Application projects compiled with the IAR.
 2. If the Stack project was changed or if BLE component was updated, the RAM section size highlighted on **Figure 10** might need to be changed.
 3. By default, IAR produces the Lp023 warning when the **BLE_OTA_UpgradableStack_HID_Keyboard** project is imported. This is caused by fact that its bootloader (**BLE_OTA_UpgradableStackExample_Stack** for this project) overlaps with symbols that are imported to it from the bootloader project. Given that this is intended implementation and it is not an issue here, the Lp023 warning is suppressed.

Exporting Projects to uVision

The steps to export projects to uVision:

Step 1. Export the **Launcher** project to the uVision:

- a. Add the **BLE_OTA_UpgradableStackExample_Launcher** project to the PSoC Creator workspace.
- b. Open “Build Settings” of the **Launcher** project and change “Toolchain” to “ARM MDK Generic”
- c. Generate the **Launcher** application in the PSoC Creator (click “Build” -> “Generate Application”)
- d. Export the **Launcher** project to uVision (click “Project” -> “Export to IDE”).
- e. Build the exported **Launcher** project in the uVision.

Step 2. Export the **Stack** project to uVision:

- a. Add the **BLE_OTA_UpgradableStackExample_Stack** project to the PSoC Creator workspace.
- b. Open “Build Settings” of the **Stack** project and change “Toolchain” to “ARM MDK Generic”.
- c. Configure the **Bootloadable** component to use images that were compiled with uVision in Step 1.e.
- d. Generate the **Stack** application in the PSoC Creator (click “Build” -> “Generate Application”).
- e. Export the **Stack** project to uVision (click “Project” -> “Export to IDE”).
- f. Add argument “*--symdefs=exported_symbols.obj*” to the Linker options in uVision:
 - open BLE_OTA_UpgradableStackExample_Stack01.uvproj in uVision
 - open the "Options for Target" window and select the **Linker** tab
 - in the **Misc controls** field append "*--symdefs=exported_symbols.obj*" to the end of the arguments list.
- g. Build the exported **Stack** project in the uVision.

Step 3. Export the HID_Keyboard project to uVision:

- Add the **BLE_OTA_UpgradableStack_HID_Keyboard** project to the PSoC Creator workspace.
- Open “Build Settings” of the **HID_Keyboard** project and change “Toolchain” to “ARM MDK Generic”
- Configure the **Bootloadable** component to use images that were compiled with uVision in Step 2.g.
- Configure the **BLE** component. Make sure that the path to the *.cycsa file is refer to *.cycsa file which was generated during step #2.g and is located in the **Stack** project folder: Generated_Source\PSoC4\
- Generate the **HID_Keyboard** application in the PSoC Creator (click “Build” -> “Generate Application”).
- Export the **HID_Keyboard** project to uVision (click “Project” -> “Export to IDE”).

For BLE component version 3.30 (or older), you must add an empty declaration of the CyBle_BleSSInterruptHandler function in main.c to prevent a build linker error “Error: L6218E: Undefined symbol CyBle_BleSSInterruptHandler (referred from cyble_hal_int.o)” in the uVision:

```
void CyBle_BleSSInterruptHandler(void)
{
}
```

- Build the exported **HID_Keyboard** project in uVision.

Upgrading Project images

To perform any application update, the Stack project image should be launched. If the current running project image is the HID keyboard, rebooting to Stack project image can be done by a long (>500ms) **SW2** button press. While the device is running the Stack project image code, and is ready to receive Stack or Application project image updates (*.cyacd files), the red LED has to be lighting.

Note After any of project images are updated, the bootloader metadata rows are changed: the information about project image’s bounding data is saved to the unused bytes of the metadata row. This will result in the failure of the verification after the project is launched for the first time.

There is one software tool provided by Cypress that can send Stack or Application project image updates to the device - CySmart. 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 the 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:

1. Make sure the OTA device is running a Stack project image, and is ready to receive updates.
2. Connect the BLE Dongle.
3. Open the CySmart software, and select the 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	The 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 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 match the settings used in the BLE component of the Stack project. To change the security settings in CySmart, click the **Configure Master Settings** button in the tool and go to the **Security parameters** page.

HID keyboard Project image upgrade

Upgrade can be done by CySmart. The *.cyacd file that was generated for the HID keyboard project (BLE Upgradable Stack Example HID keyboard) is used.

The bonding data might be maintained only if the HID keyboard project image has been updated and if the CCCD data size is not changed.

Stack Project image upgrade

The Stack project image update procedure includes erasing the Application project image, thus the new Application project image (*BLE_Upgradable_Stack_Example_HID_Keyboard* in this example project) must be uploaded after the Stack project image update procedure.

The Stack project image similar to the Application project image can be updated using CySmart application. The *.cyacd file that was generated during the Stack project compilation should be used.

Expected Results

LED Behavior Description

Launcher project

Color	State	Description
Blue	Static	Project is active

The Launcher project image is intended to be active for a short period of time. If it is active for 5 seconds or more, it means that there are no valid applications on the device. The only exit from this state is reprogramming the device.

Stack project

Color	State	Description
Red	Static	Waiting for an update or the update is in process.
None	None	The low-power mode.

The Stack project implements only the Bootloading functionality. Thus, when the device is active, it is waiting for bootloading connections. If no connection event occurs for ~40 seconds, the Stack project image automatically switches to the Application project image if it is present and valid.

Application project

Color	State	Description
Green	Blinking	The device is in an active state.
Cyan (Green+Blue)	Static	The low-power mode.
Yellow (Red+Green)	Static	Connected.

The Application project implements the BLE HID keyboard example with a key press and battery level notifications simulation.

Bonding Data Retention

The Application project image supports saving bonding information after a new image has been received. This means that after updating the Application project image, it is not required to repeat a pairing procedure.

Bonding data is removed if:

1. The Stack project has been updated.
2. The number of CCCD descriptors has changed.

Using UART for Debugging

In this example projects, 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 and increases the code size, but it provides an extra debugging output to the UART. If it is not required – set it to “NO”.

Also, file *options.h* contains define “DEBUG_UART_USE_PRINTF_FORMAT”. It is set to “NO” to prevent using the `printf` function if it is prohibited, thus `DBG_PRINTF(...)` becomes unavailable. This greatly reduces the code size (around 4 KB for GCC) but still allows printing debug information.

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 Baud rate, Parity, Stop bits and Flow control information in the HyperTerminal configuration window. By default, the settings are the following: 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 of type `uint32`. Prints starting with nonzero digit, e.g. 100 not 0000100
- `DBG_PRINT_HEX(a)` – Prints a hexadecimal number of type `uint32`. Prints starting with nonzero digit, e.g. 1a0 not 000001a0
- `DBG_PRINT_ARRAY(a,b)` – Prints ***b*** first elements of array ***a***.
- `DBG_PRINTF(...)` – Prints the function macro. Note that it is present only if “`DEBUG_UART_USE_PRINTF_FORMAT`” is set to “YES”.
- `DBG_PRINT_HEX_BYTE(a)` – Prints only two least significant hexadecimal digits of the `uint32` number.
- `DBG_PRINT_HEX_TEXT(uint32 hex, char *text)` – Prints hexadecimal number `hex` with the `DBG_PRINT_HEX` macro, then prints string `text` with the `DBG_PRINT_TEXT` macro. A useful replacement for `DBG_PRINTF(...)` when it is unavailable.
- `DBG_PRINT_DEC_TEXT(uint32 hex, char *text)` – Prints decimal number `hex` with the `DBG_PRINT_DEC` macro, then prints string `text` with the `DBG_PRINT_TEXT` macro. A useful replacement for `DBG_PRINTF(...)` when it is unavailable.

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

© Cypress Semiconductor Corporation, 2017. 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.