Hands-on Bootloading with EZBL

Contents

Introduction

This manual introduces the EZBL Bootloader Library and the basic mechanisms required to use it. A number of step-by-step exercises are presented to help you leverage an extensive codebase of existing PIC® MCU and dsPIC® DSC bootloader capabilities and related timing, communications and flash data EEPROM emulation resources in your present or future embedded application solutions.

Exercises in this manual require the following tools:

Hardware

- PC running Microsoft **Windows®**
  Mac and Linux developers can build and program bootloader projects, but not upload application projects through the bootloader due to OS API differences for accessing USB/UART hardware. This version of EZBL does not presently have a cross-platform PC communications tool.

- **Explorer 16/32 Development Board** with the **PIC24FJ1024GB610 PIM** (DM240001-3 or DM240001-2 + MA240023).
  Exercises can be done with a number of other 16-bit PIMs for the Explorer 16/32 by changing which processor is selected and which I/O pins are initialized.

  Owners of the prior Explorer 16 Development Board can also work through these exercises, assuming they have access to an MPLAB® REAL ICE/ICD3/PICkit 3 program/debug tool and USB to RS232 interface adapter (ex: MCP2200EV-VCP) for communications.

- A **USB micro-B cable** is also required for power/ICSP programming/UART communications. The DM240001-3 kit ships with this cable.

Software

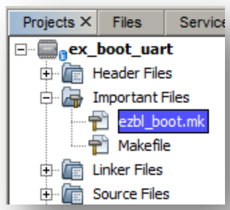
- **MPLAB® X IDE** – www.microchip.com/mplabx
  MPLAB X IDE v3.60+ is recommended
- **XC16 C Compiler** – www.microchip.com/xc16
  XC16 v1.30+ is required. Device header files for the PIC24FJ1024GB610 may be incompatible with earlier compiler releases unless a Part Support Update has been applied.
- **EZBL** – www.microchip.com/ezbl
  EZBL v2.00+ required; earlier versions lack features required for these exercises.

The EZBL source files should be unzipped on your computer in any location you want, but there must not be any spaces in the parent folder path (GNU Make does not support file paths with spaces well and will typically result in complete build failure).

Exercise 1 - Generating the bootloader project

In this section we will simply build and program a UART bootloader into the silicon. EZBL simultaneously supports nearly all 16-bit dsPIC33 DSCs and PIC24 MCUs (~454 devices considering varying flash geometry), so a minimal amount of project configuration and "bootloader uniquification" is required to get started.

1. Open the **ex_boot_uart** example bootloader project in the 'ezbl' folder in MPLAB® X IDE
    a. In MPLAB X, press the menu options: File->Open Project
    b. Navigate to the folder in which you unzipped the EZBL download
    c. Select the 'ex_boot_uart' project and press the 'Open' button
2. Change processor to **PIC24FJ1024GB610**
    a. Right click on the project within the 'Projects' tree view pane
    b. Select 'Properties'
    c. The upper right 'Device' drop down displays the target processor. If it isn't already set to 'PIC24FJ1024GB610', change it.
    d. Press the 'OK' button
3. Expand the 'Important Files' Projects tree folder branch and double click the **ezbl_boot.mk** file. This EZBL specific makefile script will open in the editor.



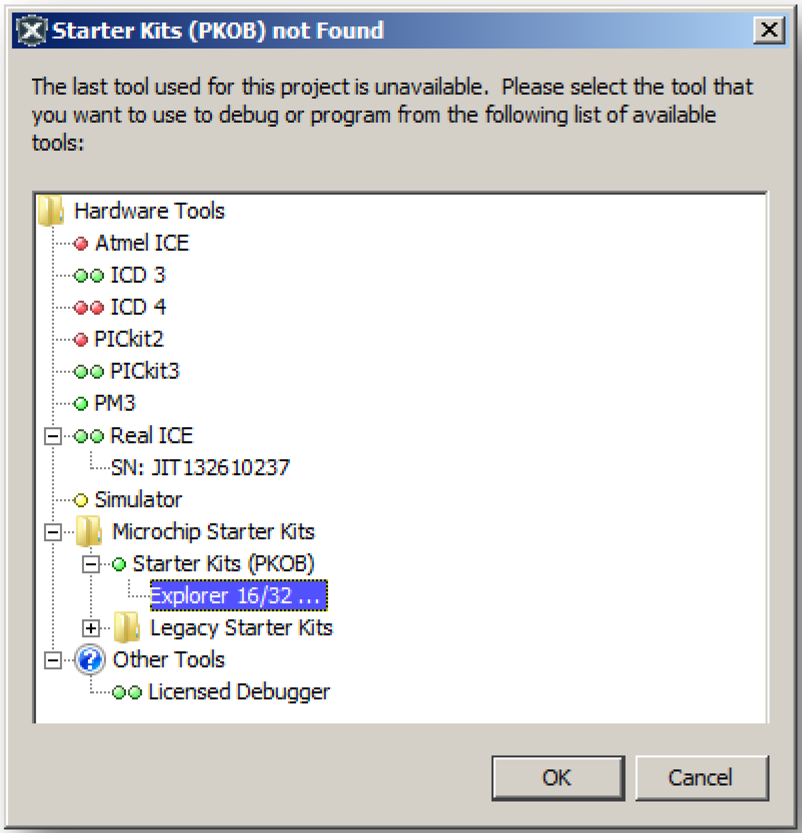
4. The top of this file contains definitions for **Vendor**, **Model**, **Name** and **Other**. Change these strings to something (anything) that uniquely represent a hypothetical end product that you are adding bootloader functionality to.

```
BOOTID_VENDOR = "Microchip Technology"
BOOTID_MODEL  = "ezbl_product"
BOOTID_NAME   = "Microchip development board"
BOOTID_OTHER  = "PIC24/dsPIC"
```

These strings are concatenated and SHA-256 hashed into a globally unique bootloader identification constant which gets stored in the bootloader. When building application image(s), the hash is carried along as project metadata so that the bootloader can reject any upload that is incompatible with the end product prior to erasing/destroying any existing application already in flash. We don't want "grandma" bricking her Tea Kettle by inadvertently sending your Coffee Pot v2 firmware to it.

The individual field names of vendor/model/etc. have no specific meaning or significance since a combined hash is the ultimate output. However, these names offer suggested input data to ensure a globally unique BOOTID_HASH will be generated.

5. Make sure you have a USB cable connected to the **PICkit on board** micro-B port near the top left of the Explorer 16/32 board. This is the ICSP debug and programming port.
6. **Build/Program** ( )
   a. If prompted for which programmer to use, make sure to select the "Starter Kits (PKOB)" option for your board.



The last tool used for this project is unavailable. Please select the tool that you want to use to debug or program from the following list of available tools:

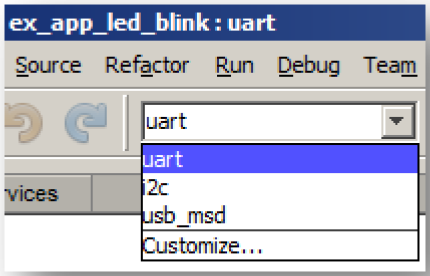
```
Hardware Tools
  Atmel ICE
  ICD 3
  ICD 4
  PICkit2
  PICkit3
  PM3
  Real ICE
    SN: JIT132610237
  Simulator
  Microchip Starter Kits
    Starter Kits (PKOB)
      Explorer 16/32 ...
    Legacy Starter Kits
  Other Tools
    Licensed Debugger
```

7. Verify **LED D3** is blinking rapidly (8 Hz). If it is, your bootloader is successfully executing and awaiting application upload.

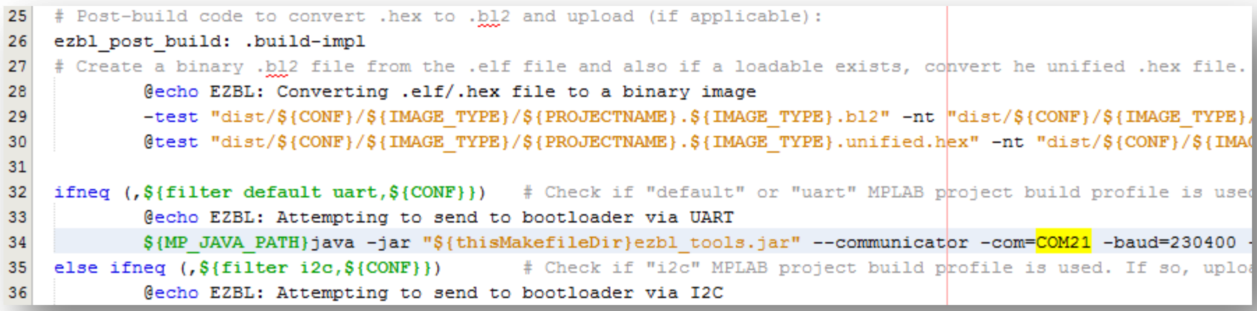
Exercise 2 – Loading an example application

This exercise will have you load an already existing demo application using the Exercise 1 bootloader. The application will blink the same D3 LED, but at a much slower 1 Hz rate to visually identify success. We will need to modify it slightly to provide system communications parameters and compile it to inherit the previously defined bootloader ID hash.

1. If you are working with only one USB cable, disconnect it from the "PICkit on board" connector and connect it to the **Serial** USB connector near the DC barrel jack.
    a. On some systems a driver installation might be required to use the MCP2221A USB to UART converter chip attached to the 'Serial' connector. This driver is part of Windows, but requires a signed .inf configuration file obtained automatically from the Microsoft Windows Update service to be used.
    b. If the system does not have Internet connectivity, necessary system policy or otherwise is unable to automatically install the needed serial driver, manually install it using the MCP2200/MCP2221 Windows Driver & Installer ('McphCdcDriverInstallationTool.exe') downloaded from the http://www.microchip.com/mcp2221a product page.
2. Close the bootloader project you had open in Exercise 1.
    a. Right click on the 'ex_boot_uart' project name in the 'Projects' tree view pane
    b. Select the 'Close' menu option
3. Open the **ex_app_led_blink** project supplied with the EZBL distribution. This is the existing demo application project already mostly configured for upload to EZBL bootloaders.
4. Change the project Build Configuration to **uart,** if not already selected:

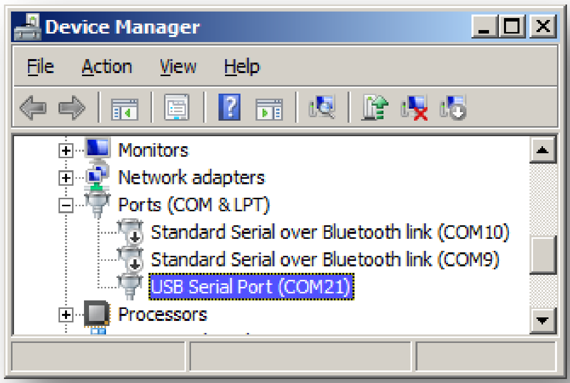


5. Change the target processor for this 'uart' build configuration to match the 'PIC24FJ1024GB610' device targeted by the bootloader just like we did in Exercise 1, Step 2.
6. Modify **ezbl_app.mk** in the "Important Files" Projects tree branch to point to the COM port assigned to your MCP2221A. (-com=COMxx parameter).



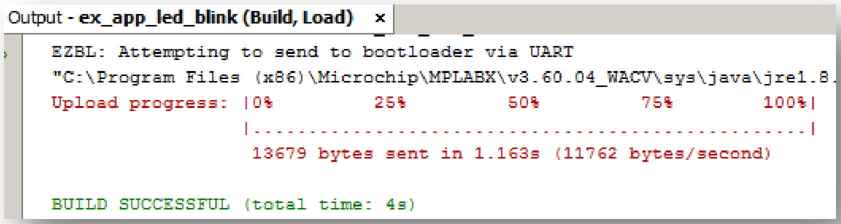
```
# Post-build code to convert .hex to .bl2 and upload (if applicable):
ezbl_post_build: .build-impl
# Create a binary .bl2 file from the .elf file and also if a loadable exists, convert he unified .hex file.
        @echo EZBL: Converting .elf/.hex file to a binary image
        -test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" -nt "dist/${CONF}/${IMAGE_TYPE}/
        @test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex" -nt "dist/${CONF}/${IMA

ifneq (,${filter default uart,${CONF}})    # Check if "default" or "uart" MPLAB project build profile is use
        @echo EZBL: Attempting to send to bootloader via UART
        ${MP_JAVA_PATH}java -jar "${thisMakefileDir}ezbl_tools.jar" --communicator -com=COM21 -baud=230400 -
else ifneq (,${filter i2c,${CONF}})        # Check if "i2c" MPLAB project build profile is used. If so, uplo
        @echo EZBL: Attempting to send to bootloader via I2C
```

If you do not know the communications port name assigned on your PC, this information can be obtained in the Windows Device Manager:



Although OS versions vary, this dialog can typically be opened by pressing WINDOWS_KEY + ‘R’ and typing **devmgmt.msc**

7. **Clean & Build** ( ) – after compilation, the ‘ezbl_post_build’ recipe in ‘ezbl_app.mk’ will automatically program the demo project using the bootloader.
    a. If successful the **D3 LED** should be blinking now at only 1 Hz.



```
EZBL: Attempting to send to bootloader via UART
"C:\Program Files (x86)\Microchip\MPLABX\v3.60.04_WACV\sys\java\jre1.8.
Upload progress: |0%          25%          50%          75%          100%|
                 |.....................................................|
                  13679 bytes sent in 1.163s (11762 bytes/second)


BUILD SUCCESSFUL (total time: 4s)
```

    b. If you toggle the ‘**MCLR**’ button on the Explorer 16/32, you will observe the LED blinking rapidly for 1 second, followed by the slower 1 Hz blinking of the new application. The initial fast blinking is caused by the bootloader executing briefly to provide a “debricking window” in case if an application is installed that disables or blocks bootloader access. After a 1 second interval with no UART activity, the application is launched for normal device operation.

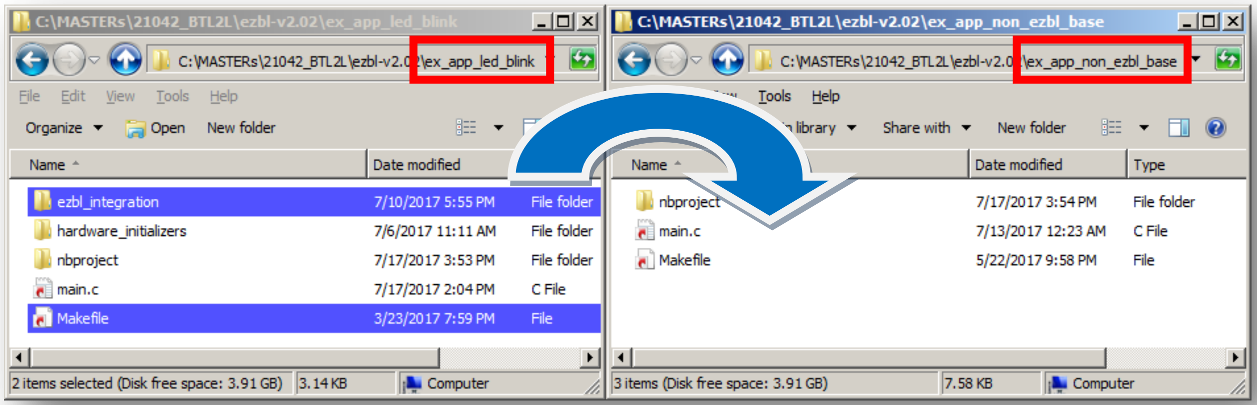
8. Open the **main.c** file in the “Source Files” Project tree branch by double clicking it. We will modify this file to change the LED blink pattern.
    a. Locate the `LEDToggle(0x01);` line (in main(), while(1) loop, at or near line 109).
    b. Modify it to `LEDToggle(0x0F);`
9. **Build** (🔨) – after compilation, the new application firmware will again be sent to the bootloader. The existing, executing application will be halted and erased as this upload will contain matching bootloader ID hash data.

Exercise 3 – Adding EZBL to an application

In this section we shall take a new or existing application project (having no association to EZBL) and perform necessary project changes to make it compatible/programmable using our EZBL bootloader.

1. Since this is a learning exercise, an existing application project without EZBL association is provided. Therefore, it is useful to first test the application project so we know what behavior to expect when we successfully pair the bootloader and application projects.
    a. Close the **ex_app_led_blink** application you had open in Exercise 2.
    b. Open the **ex_app_non_ezbl_base** project. This project requires only the XC16 compiler without other dependencies.
    c. Disconnect the USB cable from the 'Serial' connector and move it back to the **PICkit on board** connector for normal ICSP tool supported programming.
    d. **Build/Program** ( ). As before, if prompted for which programmer to use, make sure to select the "Starter Kits (PKOB)" option.
    e. Observe the LED blink pattern of **0x55** at 1 Hz. Timing for this is performed using the blocking __delay_ms() macro contained in libpic30.h, which is part of the XC16 tool chain.
2. Using your computer's file browser (i.e. – Windows Explorer), go to the **ex_app_led_blink** project folder to obtain some needed EZBL collateral, including our bootloader in binary form. Copy the following two items to the system clipboard:
    - **ezbl_integration** folder
    - **Makefile** file

   Now, navigate to the **ex_app_non_ezbl_base** project folder and paste the clipboard contents.


C:\MASTERs\21042_BTL2L\ezbl-v2.02\ex_app_led_blink
C:\MASTERs\21042_BTL2L\ezbl-v2.02\ex_app_led_blink
File
Edit
View
Tools
Help
Organize
Open
New folder
C:\MASTERs\21042_BTL2L\ezbl-v2.02\ex_app_non_ezbl_base
C:\MASTERs\21042_BTL2L\ezbl-v2.02\ex_app_non_ezbl_base
Tools
Help
library
Share with
New folder
Computer
Computer

Name	Date modified	
ezbl_integration	7/10/2017 5:55 PM	File folder
hardware_initializers	7/6/2017 11:11 AM	File folder
nbproject	7/17/2017 3:53 PM	File folder
main.c	7/17/2017 2:04 PM	C File
Makefile	3/23/2017 7:59 PM	File

2 items selected (Disk free space: 3.91 GB) 3.14 KB

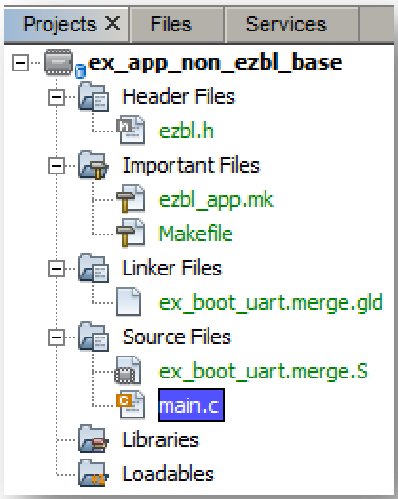
Name	Date modified	Type
nbproject	7/17/2017 3:54 PM	File folder
main.c	7/13/2017 12:23 AM	C File
Makefile	5/22/2017 9:58 PM	File

3 items (Disk free space: 3.91 GB) 7.58 KB

    a. The correct paste destination is the 'ex_app_non_ezbl_base' folder (will contain the 'nbproject' folder and 'Makefile' file created by MPLAB when starting a new project).

Many existing project folders have a `.X' suffix in the folder path, which would be the correct destination, however, to minimize path length, EZBL example projects all have the .X suffix removed.

   b. You will be prompted to overwrite 'Makefile'. Approve this.
      i. In the event your existing project has customized Makefile content, rather than overwrite Makefile, you can instead paste the following line at the bottom of your existing one:
      ```
      include ezbl_integration/ezbl_app.mk
      ```
3. Returning to MPLAB® X IDE, add various bootloader files to the 'Projects' tree view:
   a. Right click on **Header Files** and select "Add Existing Item…"
      i. Navigate to **ex_app_non_ezbl_base\ezbl_integration**
      ii. Add **ezbl.h**
   b. Right click on **Important Files** and select "Add Item to Important Files…"
      i. Add **ezbl_app.mk**
   c. Right click on **Linker Files** and select "Add Existing Item…"
      i. Add **ex_boot_uart.merge.gld**. This is a linker script automatically generated by EZBL when we compiled our 'ex_boot_uart' bootloader project.
   d. Right click on **Source Files** and select "Add Existing Item…"
      i. Add **ex_boot_uart.merge.S**. This is a copy of the absolutely linked bootloader's executable image. Effectively this is the 'ex_boot_uart.production.hex' artifact, converted to a binary form, and then converted back to a simple assembly source file so the application's .hex output artifact will contain both your bootloader and your application simultaneously.
   e. Your project should now contain all of these files:



4. Open main.c (in 'Sources Files') and delete all of the **#pragma config** statements.
    a. Although EZBL can support updating Config words with new application uploads, doing so requires the Config words to not be already defined in the bootloader project. Since our 'ex_boot_uart' project does contain Config word definitions, this application project cannot have them simultaneously defined.
5. **Build/Program** ( )
    a. Ignore the EZBL communications error that shows up in the build Output window. Step 1 entailed wiping all flash contents and programming the basic application, so there is no bootloader for the 'ezbl_app.mk' post-build make script to communicate with. Additionally, if using only one USB cable, the cable will presently be connected to the 'PICkit on board' interface, eliminating the PC's COM port necessary to attempt communications.
6. Observe that the board LEDs are again blinking with the 0x55 pattern. However, if you toggle the **MCLR** button, you will observe a one second reset interval in which only LED D3 blinks (at 8 Hz). This indicates we have a bootloader again.
7. Modify the 'main.c' **LED blinking code** once more. Instead of:
    ```
    LATA ^= 0x0055;
    ```
    Try:
    ```
    LATA  = (LATA<<1) ^ _LATA3 ^ 1;
    ```
    Note that this is an 'equal' assignment now instead of 'XOR equal'.
8. Move the USB cable from the 'PICkit on board' connector back to the **Serial** connector.
9. **Build** ( ) – this will both compile and result in programming via the UART bootloader. Even though we never explicitly reprogrammed the 'ex_boot_uart' bootloader project after erasing it in Step 1, the presence of the 'ex_boot_uart.merge.S' source file within the project caused the bootloader to reexist in flash after Step 5.
10. Successful bootloading will be identifiable by observing glowing 4-bit caterpillars extend out of the least significant LED and perish after walking into the JP2 jumper shunt at the most significant end. It is favorable that they perish, since their bright green self-luminescence suggests possible radioactivity.

    If you only see **4 LEDs rotating left**, this is an alternate indicator of success.

Exercise 4 – Using EZBL library functions and macros

EZBL comes with numerous library API functions and macros that it implements for its own timing, communications, flash manipulation and hardware abstraction tasks. Unlike historical bootloader designs, these same APIs may be used in application projects (or even projects that do not implement or use an EZBL bootloader).

Additionally, when an application calls library APIs which already exist in flash (due to the bootloader calling the same function), EZBL allows applications to utilize the same functions without having to duplicate them in physical flash.

This exercise will reuse an LED function, a timer function and flash erase/program APIs implemented in the bootloader in order to implement an otherwise complex LED blinking application with emulated data EEPROM state persistence. The same methodologies can be used for accessing various other functions, including communication protocols, file systems or any other global API in your bootloader project.

Accessing a simple LED function

In this section we will reuse a simple LED function from the bootloader in our application.

1. Continuing on from our Exercise 3 project, right click on **Libraries** in the Projects tree view window and select "Add Library/Object File…"
    a. Add **ezbl_lib.a**. This file is located in the same "ex_app_non_ezbl_base\ezbl_integration" folder where we collected files from in the prior exercise.
2. Open **main.c** for editing, if not already open.
3. Include the **ezbl.h** header:
    a. At file-level scope, just after the "#include <libpic30.h>" statement, insert:
    ```
    #include "ezbl_integration/ezbl.h"
    ```
4. Delete the **TRISA/LATA/ANSA** lines at the top of the main() function. The 'ex_boot_uart' bootloader project already initialized these registers in order to make the D3 LED blink rapidly during the reset debricking interval. This makes reinitialization of the same hardware a waste of code space in application projects using this bootloader.
5. Inside the main() function's while(1), loop, replace your LATA toggling code:
    ```
    LATA   = (LATA<<1) ^ _LATA3 ^ 1;
    ```
    With this hardware abstracted API call:
    ```
    LEDOn(0xF0);
    ```
6. **Build** ( ). Despite not implementing any code for LEDOn(), the project will still compile, link and upload normally.

   a. This works because the ‘ex_boot_uart’ project contains a hardware initialization file which implements LEDToggle() and which is called by the bootloader project.
   b. LEDToggle() isn’t LEDOn(), but the code still links successfully because LEDOn() is implemented within the ‘ezbl_lib.a’ precompiled archive library, which among many other APIs, includes a default LEDOn() implementation. This LEDOn() library API calls LEDToggle(0x00) in order to get the present LED state, and then calls LEDToggle() again with a derived bitmask needed to achieve the final LEDOn(0xF0) output state.
7. The LEDs should now have the **0xF0** pattern statically displayed.

If you saw the 0xF0 pattern on the LEDs, you successfully reused both LEDToggle() implemented in your bootloader (not duplicated in flash), and pulled in a new LEDOn() API from the ezbl_lib.a archive. If you like, you can go to the bootloader project’s “ex_boot_uart\hardware_initializers\pic24fj1024gb610_explorer_16.c” file to look at the `LEDToggle()` code implementation and look in the “ezbl_lib\weak_defaults\LEDOn.s” file if you wish to additionally see the `LEDOn()` implementation.

Complete Solution

```
#include <xc.h>
#define FCY      16000000
#include <libpic30.h>
#include "ezbl_integration/ezbl.h"

int main(void)
{
    while(1)
    {
        LEDOn(0xF0);
        __delay_ms(500);
    }
}
```

Accessing a timer function

EZBL has a simple scheduler module that it uses for executing various background bootloading communications and timing events. As such, just like anything else in EZBL, this functionality can be utilized by the application. In this section we will utilize the “NOW” timing functions to create a repeating task to toggle the LEDs, replacing the existing blocking loop construct.

To do this, we are going to use the `NOW_CreateRepeatingTask()` macro. This macro takes three parameters:

- A pointer to a NOW_TASK object, statically allocated in RAM
- A pointer to a function that will be periodically called

- A length of time between function calls. You can use the NOW_sec, NOW_ms and NOW_us definitions to get the right amount of time. For example, 3 seconds is represented as:

  3 * NOW_sec

For this exercise, we will create a task function that toggles all of the LEDs every 500ms.

1. Create a global NOW_TASK variable (pick any name you like).

   ```
   NOW_TASK ledToggleTask;
   ```

   If global variables are undesirable in your environment, make this declaration *static*. We only need a pointer to the structure with the API mandating that the RAM for it stays continuously allocated.
2. Create a function that returns `int`, takes `void` as input and toggles all of the LEDs. Add this outside of and before the main() function:

   ```
   int toggleLED(void)
   {
       return LEDToggle(0xFF);
   }
   ```
3. At main() initialization, before the while(1) loop, call the NOW_CreateRepeatingTasks() macro with a pointer to the task variable, a pointer reference to the function and a 500ms time interval.

   ```
       NOW_CreateRepeatingTask(&ledToggleTask, toggleLED, 500*NOW_ms);
   ```

   Note: there are no parenthesis after `toggleLED`. Having parenthesis would cause the function to be called with the 0 return value being passed to the NOW_CreateRepeatingTask() macro instead of the function's address (or handle) that we need.
4. Since we are writing to the LEDs using this new task, we should delete the previous `LEDOn()` API call. Also, the existing `__delay_ms()` call within the main() while(1) loop is a waste of power, so we should replace it with a call to `Idle()`.
5. **Build** ( ). Observe all 8 LEDs blinking now.

Complete Solution

```
#include <xc.h>
#include "ezbl_integration/ezbl.h"


NOW_TASK ledToggleTask;

int toggleLED(void)
{
    return LEDToggle(0xFF);
}

int main(void)
{
    NOW_CreateRepeatingTask(&ledToggleTask, toggleLED, 500*NOW_ms);
    while(1)
    {
        Idle();
    }
}
```

Emulated data EEPROM in flash

Since the bootloader contains an array of flash erase/write APIs already used for bootloading functionality, it is easy to store run-time state data in flash and doing so requires minimal extra code. For this exercise, let's store an unsigned int in flash and toggle the LEDs according to the stored value. We'll use the S4 push button on the Explorer 16/32 for changing the run-time LED toggle mask, which shall persist after power cycling.

This section entails more code than prior exercises, so copying and pasting the code into your application's **main.c** file can expedite this task. In Windows Explorer, the "**ezbl-v2.xx\help\EZBL Hands-on Bootloading Exercises - Exercise 4 Solution.c**" file contains all of the code we will go over in this section. Alternatively, you will find all of the necessary code at the end of this section, but PDF files do not retain whitespace characters well when using clipboard operations. Return to at least Step 6 to observe the end results.

1. First, allocate at least a page of flash memory at an aligned boundary so we can erase and reprogram emulated EE data without overlapping anything in the bootloader or application:

   ```
   EZBL_AllocFlashHole(emuEEData, 3072, 0x800, -1);
   ```

   This macro statically allocates/reserves flash memory, so place it at global/file level scope.

The first parameter is an arbitrary name we are giving this memory, 3072 is the number of bytes we shall reserve, 0x800 (program space addresses) is the flash page erase size for the PIC24FJ1024GB610 device, which defines both alignment and padding requirements, and -1 is a flag meaning the linker should choose the base addresses for this object (change this to an absolute flash address if desired).

2. Let's also declare a structure in RAM to hold the EE data for efficient execution references and allow direct variable manipulation:

```
struct
{
    unsigned int ledsToBlink;
} eeVars;
```

   Since we have at least 3072 bytes of flash reserved, you can obviously add a lot more non-volatile variables to this structure. In this exercise, however, we only need one variable.

3. To consume this new variable, update the toggleLED() function:

```
int toggleLED(void)
{
    return LEDToggle(eeVars.ledsToBlink);
}
```

4. At the start of main(), we need to read the `emuEEData` flash contents and copy it to the `eeVars` RAM structure. We should also implement a default value for the first time we power up where `emuEEData` will contain nothing. As flash memories erase to all '1's, this equates to an 0xFFFF check:

```
    EZBL_ReadROMObj(&eeVars, EZBL_FlashHoleAddr(emuEEData));
    if(eeVars.ledsToBlink == 0xFFFFu)
    {   // On erased power up, define initial state
        eeVars.ledsToBlink = 0x0003u;
    }
```

5. Finally, we need a way to run-time change eeVars and commit it to flash in order to demonstrate the structure is non-volatile. This updated while(1) loop in main() will achieve this by incrementing the LED toggle mask each time the S4 button is pushed:

```
    while(1)
    {
        if(!_RD13)
        {// S4 pushed
            eeVars.ledsToBlink = (eeVars.ledsToBlink + 0x1) & 0x00FF;
            LEDSet(eeVars.ledsToBlink);
            EZBL_WriteROMObj(EZBL_FlashHoleAddr(emuEEData), &eeVars);
            while(!_RD13);
        }
        Idle();
    }
```

6. **Build** ( ). After the bootloader programs the application, observe that the two LSbit LEDs are blinking according to the 0x03 pattern.
7. Push the **S4** button a couple times (or as many times necessary for the appeal of pushing buttons to diminish). With each push, you will observe the LED blink pattern increment by binary 0x01.
8. Toggle the **Power** button a couple of times and/or toggle **MCLR**.

A successful implementation will demonstrate that the last displayed blink pattern remains persistently defined for each subsequent power cycle and/or reset event.

In a real application, consider placing the `EZBL_AllocFlashHole()` macro call in your bootloader project instead of this application project. By allocating the EE flash pages in your application, any EE data you have stored will get erased each time a new application is programmed by the bootloader.

By moving the flash hole to the bootloader project, the EE page(s) will survive through all application updates. The `EZBL_ReadROMObj()` and `EZBL_WriteROMObj()` APIs will be accessible from both the bootloader and application projects as a result.

When keeping data through application update cycles, however, it is important to keep track of which `eeVars` structure members may contain data, which ones may be uninitialized with all ‘1’s and which ones may be new to a just-bootloaded application version. This could become complicated if, for example, you regularly add new non-volatile variables and end users program v1.3 of your application when they last had v1.0 installed. This may violate your natural development progression of having v1.1 installed after v1.0, upgrading to v1.2 and finally bootloading to v1.3.

One strategy to overcome incorrect or incoherent non-volatile data is to consistently add and keep first-run initializer code when introducing new `eeVars`. Additionally, you would never delete, change the size, or reorder prior `eeVar` structure elements. Instead, you would only add new variables to the end of `eeVars` and ignore earlier members you no longer need – maintaining their allocated memory.

Complete Solution

```
#include <xc.h>
#include "ezbl_integration/ezbl.h"

EZBL_AllocFlashHole(emuEEData, 3072, 0x800, -1);
struct
{
    unsigned int ledsToBlink;
} eeVars;
NOW_TASK ledToggleTask;

int toggleLED(void)
{
    return LEDToggle(eeVars.ledsToBlink);
}

int main(void)
{
    EZBL_ReadROMObj(&eeVars, EZBL_FlashHoleAddr(emuEEData));
    if(eeVars.ledsToBlink == 0xFFFFu)
    {   // On erased power up, define initial state
        eeVars.ledsToBlink = 0x0003u;
    }
    NOW_CreateRepeatingTask(&ledToggleTask, toggleLED, 500*NOW_ms);

    while(1)
    {
        if(!_RD13)
        {
            eeVars.ledsToBlink = (eeVars.ledsToBlink + 0x1) & 0x00FF;
            LEDSet(eeVars.ledsToBlink);
            EZBL_WriteROMObj(EZBL_FlashHoleAddr(emuEEData), &eeVars);
            while(!_RD13);
        }
        Idle();
    }
}
```

Exercise 5 - Updating the bootloader project

If you are going to update the bootloader project, it is important that the 'ezbl_integration' files used by the application match between projects so that the linker keeps application flash contents off of bootloader reserved flash pages and has correct variable/function addresses for all bootloader APIs that the application references.

When you build an EZBL bootloader, it automatically generates a new linker script file ('ex_boot_uart.merge.gld') for your application, so no manual modification is required if the bootloader changes size during development. You can even make radical changes to your bootloader, like change the target processor, and it won't be necessary to manually edit your .gld linker scripts.

However, despite this automation, it is still necessary for the correct .gld linker script file to exist in your application project. In [Exercise 3](), we manually copied the 'ezbl_integration' folder in order to move bootloader files into the application project. However, this step is inconvenient and easy to forget to do while testing various bootloader revisions. In this section we are going to update the bootloader make script to copy bootloader files to our application project(s) automatically for us each time the bootloader is recompiled.

1. Open the bootloader project, **ex_boot_uart**. You may wish to close the existing application project so you don't confuse the two.
2. Expand the "Important Files" branch in the Project tree view window.
    a. Open **ezbl_boot.mk**
3. Locate the `appMergeDestFolders` variable declaration near the top of the file:

```
# List of directories where you want the .merge.gld/.merge.S EZBL output files
# to be copied to after building this Bootloader. …
appMergeDestFolders = ${thisMakefileDir}                          \
                      ../ex_app_led_blink/ezbl_integration
```

4. Modify the last line to point to the new application folder instead of the 'ex_app_led_blink' project. This will make all future merge artifacts from the bootloader propagate to the application.

```
appMergeDestFolders = ${thisMakefileDir}                          \
                      ../ex_app_non_ezbl_base/ezbl_integration
```

If you wish, you can add a new line instead, allowing multiple application projects to receive just-built bootloader artifacts. For example:

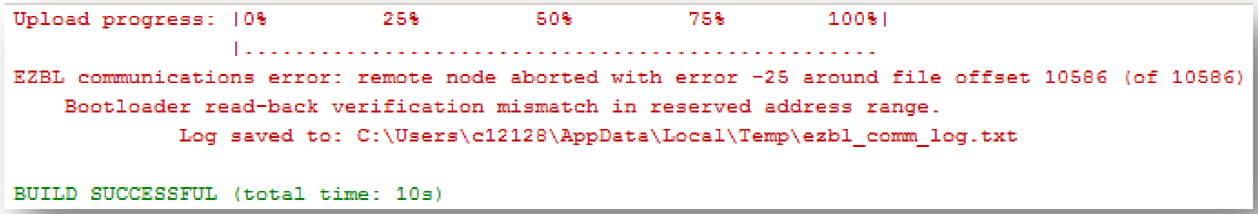
```
appMergeDestFolders = ${thisMakefileDir}                         \
                      ../ex_app_led_blink/ezbl_integration       \
                      ../ex_app_non_ezbl_base/ezbl_integration
```

5. **Build** ( ).
6. To validate success, use your OS file manager and verify that the timestamp on the "ex_app_non_ezbl_base\ezbl_integration\ex_boot_uart.merge.gld" file has been updated. The last modified time should match the system clock time when the bootloader was recompiled.

It should now be possible to make modifications to the bootloader without having to do anything to your application projects except recompile them.

Remember, however, to always program your device once via a traditional ICSP programming tool anytime you make a bootloader change. This can be accomplished by programming either the bootloader or one of the application projects since they contain the bootloader image.

If you miss the ICSP programming step and rebuild an application, having it programmed via an old, mismatched bootloader existent in flash, then you will receive a bootloader upload error message:



```
Upload progress: |0%         25%         50%         75%        100%|
                 |...................................................
EZBL communications error: remote node aborted with error -25 around file offset 10586 (of 10586)
    Bootloader read-back verification mismatch in reserved address range.
             Log saved to: C:\Users\c12128\AppData\Local\Temp\ezbl_comm_log.txt

BUILD SUCCESSFUL (total time: 10s)
```

Additionally, the application code will normally behave erratically or fail catastrophically during execution as various bootloader functions called by the application will now be attempting to branch to incorrect API addresses. This error goes away automatically once you program the correct bootloader via an ICSP programming tool.

Exercise 6 – Keeping unused symbols in a bootloader

There are various APIs in an EZBL bootloader project that might not be used by the bootloader, but which the application might like to call. For example, in a "memory" type bootloader with user insertable bulk storage, you might not use a Format() function for any bootloading operations, but you might need it for a subsequent application project. Since the function won't be called anywhere in the bootloader, XC16 optimization options within the bootloader project will cause the unreferenced Format() function to be deleted.

There are a couple ways that we can tell EZBL/XC16 to keep unused functions so that they can be used by the application. This exercise will cover an EZBL option, discuss an XC16 option and lastly discuss what to do if you've gone to production and can no longer modify your bootloader.

EZBL_KeepSYM()

ezbl.h defines a macro called `EZBL_KeepSYM()` that instructs the tool chain to keep the named symbol even if it isn't called or otherwise referenced by the (bootloader) project.

In this exercise we will update the code to use a `ButtonRead()` function that is defined, but not called in the bootloader project. We will instruct the linker to keep this bootloader function for later use by the application.

Verify it doesn't work out of the box

1. If you've closed it, reopen your (now misnamed) **ex_app_non_ezbl_base** application project.
2. Replace the main(), while(1) loop with this code:

```
    while(1)
    {
        if(ButtonPeek())   // S3, S6 or S4 pushed
        {
            eeVars.ledsToBlink = (eeVars.ledsToBlink + 0x1) & 0x00FF;
            LEDSet(eeVars.ledsToBlink);
            EZBL_WriteROMObj(EZBL_FlashHoleAddr(emuEEData), &eeVars);
            while(ButtonPeek());
        }
        Idle();
    }
```

   This code has had the `(!_RD13)` conditional code removed and replaced with the `ButtonPeek()` hardware abstracted API.

3. **Build** ( ). You should get a build error indicating an "undefined reference to `_ButtonPeek'". Don't worry, this is expected. The `ButtonPeek()` function was never called or referenced in the bootloader so XC16 optimized your flash footprint by throwing `ButtonPeek()`'s code away.

Modify the bootloader to keep the ButtonPeek() function

4. Open your **ex_boot_uart** bootloader project.
5. At the top of the **main.c** file, after ezbl.h was #included, add this macro statement: `EZBL_KeepSYM(ButtonPeek);`
6. If using only one USB cable, move it back to the **PICkit on board** connector.
7. **Build/Program** ( ).

Verify the application now links and works

8. Go back to the **ex_app_non_ezbl_base** application project.
9. Confirm USB connectivity to the **Serial** connector.
10. **Build** ( ). The application should now link successfully and program itself via the bootloader. If you still get an undefined reference error, Exercise 5 may not have been completed successfully and the timestamp on the “ex_app_non_ezbl_base\ezbl_integration\ex_boot_uart.merge.gld“ (and “ex_boot_uart.merge.S”) files should be checked.
11. With `ButtonPeak()` now being called, anytime you depress any of the **S3, S6 or S4** buttons on the board, the LED toggle pattern will increment. Previously, only S4 triggered this behavior.

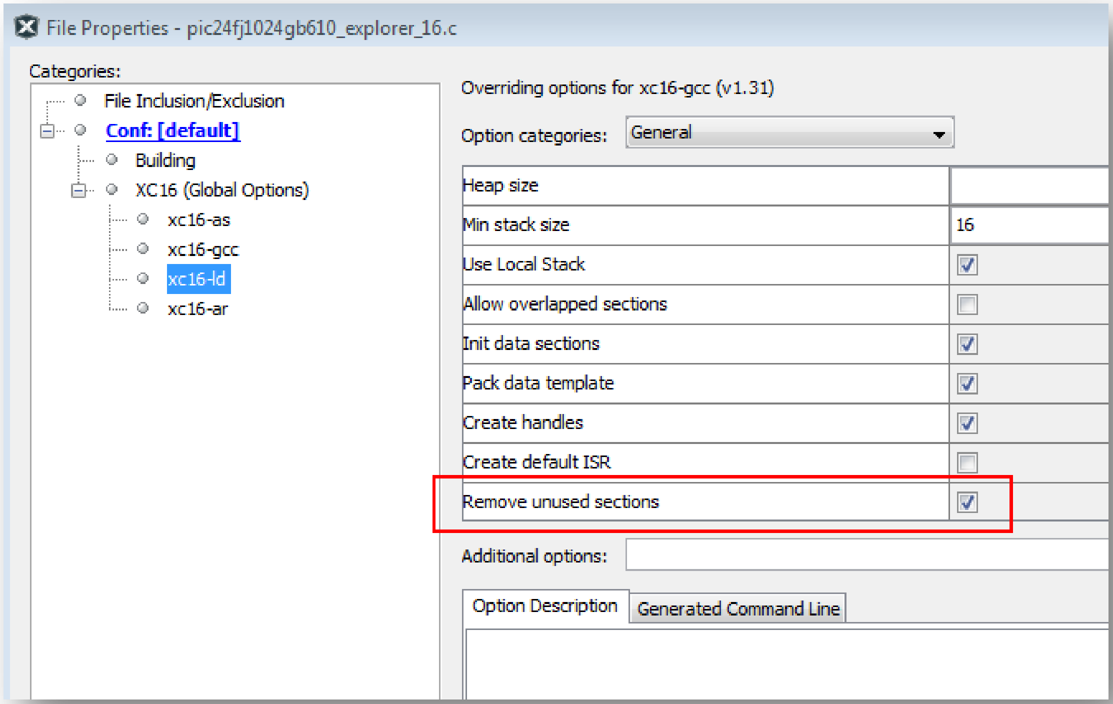
    Note: Pushing S5 doesn’t do anything. S5 shares the same hardware I/O pin as the D10 LED. Therefore, in this code, the RA7 GPIO output function blocks use of the input function.

Keep All Sections

The EZBL_KeepSYM() exercise demonstrated how to keep one specific function. This automatically keeps any functions and variables that the kept function references, so you don’t miss any related dependencies. However, if you have a lot of interrelated code that you wish to keep, you may inadvertently forget to keep some functions, only to discover that they are missing after you’ve been in production for a while and begin work on an updated application revision. In this section we will discuss how to instruct the XC16 tool chain to avoid pruning unused functions and variables altogether.

Keeping all unused code could be especially useful in large stack applications where it is difficult to predict which symbols you will someday want to use in application projects (file system functions for example). The downside to this approach is that a lot of functions will be kept regardless of if they are needed or not and this will cause the size of the bootloader to grow, sometimes substantially so.

To keep all unused functions in your bootloader project, you will need to change a project linker option. Go to the bootloader’s project properties and select the **xc16-ld** section. Uncheck the “**Remove unused sections**” box. If this box is checked, as it is by default in EZBL bootloader projects, all unreferenced sections are considered dead code and removed.


File Properties - pic24fj1024gb610_explorer_16.c

Categories:
- File Inclusion/Exclusion
- Conf: [default]
  - Building
  - XC16 (Global Options)
    - xc16-as
    - xc16-gcc
    - xc16-ld
    - xc16-ar

Overriding options for xc16-gcc (v1.31)

Option categories: General

Heap size	
Min stack size	16
Use Local Stack	✓
Allow overlapped sections	☐
Init data sections	✓
Pack data template	✓
Create handles	✓
Create default ISR	☐
Remove unused sections	✓

Additional options:

When the Bootloader is Frozen for Production

The prior options are applicable when you are still developing your bootloader. However, once you enter production, it will no longer be possible to rebuild your bootloader to keep various APIs that you may eventually need in subsequent application updates.

In this scenario, the solution is to implement the needed APIs in your application project. All functions and global variables in bootloader projects are exported for use in application projects as 'weak' attributed functions/variables. This means the linker will preferentially ignore a bootloader implementation if a same-named definition exists in your application project. In descending order of linker preference, your application will generate references to:

1. Highest priority: global or static function or variable defined in the same application project
2. 'weak' attributed global function or variable in the bootloader project (function/variable addresses come from the 'ex_boot_uart.merge.S' source file).
3. 0 (null pointer), if the function prototype or extern variable declaration is attributed as 'weak'.
4. Look inside all precompiled .a archive libraries for the function or variable, stopping when a match is found. This includes the 'ezbl_lib.a' archive included in the MPLAB X project, as well as

implicitly included archives by the XC16 tool chain, such as libpic30-elf.a, liblega-c-elf.a, etc. The archive search order is not readily controllable.

5. Lowest priority: abort with “undefined reference to ‘_xxx’” linker error message.

Note that while it is possible to add functions to the application that never existed in the bootloader and implement new functions which prevent weak bootloader functions from being reused by the application, care needs to be taken to ensure call tree coherency exists when dealing with complicated stack functions. You wouldn’t want a new application defined function to modify a global variable that is used in other older bootloader defined functions if all of the functions don’t completely agree on what the variable is intended to be used for. In the worst case, it may be necessary to include a whole new copy of the complicated code or stack in order to ignore all the existing bootloader functions and variables within your application.

Exercise 7 – Versioning your application

EZBL automatically does version numbering. Each time the application project is built, the version/build number is automatically incremented. The bootloader also does version number checking, so if desired, the bootloader can implement a policy that disallows backwards revisions. For memory type bootloaders (ex: USB thumb drive), there can also be a persistent presence of an application firmware image being offered to the bootloader. In this case it is beneficial to ignore a version being offered if it matches the application already existing in flash.

In this exercise we will manipulate the version numbers and observe bootloader handling of the offered application image.

1. Open the **ex_app_non_ezbl_base** application project.
2. Open the **ezbl_app.mk** file in the "Important Files" section.
3. You will see a few definitions for **APPID_VER**. There is a major, minor, and build field. The major and minor revisions are manually controlled by typing them here. The build number is automatically incremented by the EZBL build tools every time the project is compiled.
4. Press **Build** ( ). Observe that the build number incremented by +1 (try again a few more times if you like).
5. Change the **APPID_VER_BUILD_INC** variable to **0**.
6. Press **Build** ( ). Observe that the build number is left unchanged. This zero-increment value can be useful during pre-release project development as the 'ezbl_app.mk' file won't be constantly changing and marked as dirty, ready for checking into a source version control repository.
7. Change the **APPID_VER_BUILD_INC** variable to **-5**.
8. Press **Build** ( ). Note the result.

To suppress accidental version downgrading, we need to add a linker flag to the bootloader project to instruct it to check the application version number.

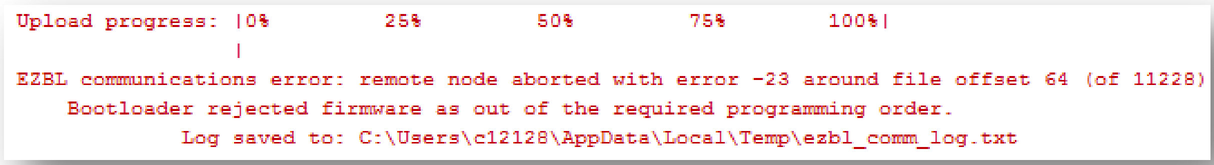
9. Open the **bootloader project** and set it as the MPLAB Main Project
10. Open **main.c**
11. Add the following line to the top of the main.c file (or uncomment the one that exists in the example already):

```
EZBL_SetSYM(EZBL_NO_APP_DOWNGRADE, 1);
```

12. Confirm USB cable connectivity to the **PICkit on board** connector.

13. Build and **Program** the new bootloader ( ).
14. Confirm USB cable connectivity to the **Serial** connector for bootloading.
15. Return to the **ex_app_non_ezbl_base** application project, setting it as the MPLAB Main Project.
16. Press **Build** ( ). This will place an initial application into the flash which will serve as the version reference.
17. If you left **APPID_VER_BUILD_INC** in 'ezbl_app.mk' set to **-5**, then the next build/upload will be in downgraded versioning order. You can decrement APP_ID_VER_BUILD, APPID_VER_MINOR and/or APPID_VER_MAJOR manually for the same effect.
18. **Clean & Build** ( ). NOTE: Unlike prior steps where Build was sufficient, use Clean to ensure .elf/.hex/.bl2 files are regenerated (alternatively, type any character into main.c, delete the character, and then issue a Build. Touching main.c updates its timestamp and triggers a fresh linking event.).

    Ordinarily, make will skip relinking the same project since no source input file changed after Step 8. The APPID_VER numbers themselves are passed from the 'ezbl_app.mk' file to the build output by passing them as definitions on the command line that invokes the linker. The changed values are therefore not trackable as clean or dirty within the make system.
19. When the build output is sent to the bootloader, rather than terminating the existing application and erasing it, the bootloader will reject the application update attempt. The existing application programmed in Step 16 will continue to execute without interruption.



```
Upload progress: |0%         25%         50%         75%        100%|
                 |
EZBL communications error: remote node aborted with error -23 around file offset 64 (of 11228)
    Bootloader rejected firmware as out of the required programming order.
             Log saved to: C:\Users\c12128\AppData\Local\Temp\ezbl_comm_log.txt
```

Changing **APPID_VER_BUILD_INC** in 'ezbl_app.mk' back to **1** and manually incrementing the build, minor or major version numbers to a version >= the Step 16 version will permit reprogramming after the next Clean & Build or Build with-linking event.

NOTE: The EZBL_NO_APP_DOWNGRADE feature is not a security feature. It exists to prevent accidental downgrades which could interfere with "Live Update" data preservation or continuity of emulated data EEPROM contents.

If a user initiates a bootloader reprogramming sequence and then intentionally power cycles the device before the update is complete, the flash contents will be left without a valid application installed. At this point the bootloader will accept any valid application image sent to the bootloader, regardless of its APPID_VER revision. Only the inherited BOOTID_HASH data must match in this app-less state.