

# 通过 EZBL 进行自举的动手实验

---

## 内容

简介 .....	2
硬件 .....	2
软件 .....	2
练习 1——生成自举程序项目 .....	3
练习 2——加载示例应用程序 .....	5
练习 3——将 EZBL 添加到应用程序 .....	8
练习 4——使用 EZBL 库函数和宏 .....	11
访问简单的 LED 功能 .....	11
完整解决方案 .....	12
访问定时器功能 .....	12
完整解决方案 .....	14
闪存中的仿真数据 EEPROM .....	14
完整解决方案 .....	17
练习 5——更新自举程序项目 .....	18
练习 6——将未使用的符号保存在自举程序中 .....	20
EZBL_KeepSYM() .....	20
验证它能否开箱即用 .....	20
修改自举程序以保留 ButtonPeek() 函数 .....	21
验证应用程序当前是否已链接并处于工作状态 .....	21
保留所有部分 .....	21
当自举程序针对生产冻结时 .....	22
练习 7——应用程序版本控制 .....	24

## 简介

本手册介绍了 EZBL 自举程序库和使用它所需的基本机制，还提供了一些分步练习，以帮助您在现有或未来的嵌入式应用解决方案中利用现有 PIC® MCU 和 dsPIC® DSC 自举程序功能的丰富代码库以及相关的时序、通信和闪存数据 EEPROM 仿真资源。

本手册中的练习需要以下工具：

## 硬件

- 运行 Microsoft **Windows**® 的 PC  
Mac 和 Linux® 开发人员可以构建和编程自举程序项目，但是由于访问 USB/UART 硬件时存在 OS API 差异，因此不能通过自举程序上传应用程序项目。此版本的 EZBL 目前没有跨平台的 PC 通信工具。
- 带 **PIC24FJ1024GB610 PIM** 的 **Explorer 16/32** 开发板（[DM240001-3](#) 或 [DM240001-2](#) + [MA240023](#)）。  
也可使用带各种其他 PIM 的 Explorer 16/32 进行练习，方法是更改所选的处理器和初始化的 I/O 引脚。

早期的 Explorer 16 开发板也可以完成这些练习，前提是使用 MPLAB® REAL ICE/ICD3/PICKit™ 3 编程/调试工具和 USB 转 RS232 接口适配器（例如：[MCP2200EV-VCP](#)）进行通信。

- 还需要一根 **USB micro-B 电缆** 来实现供电/ICSP 编程/UART 通信。DM240001-3 套件随附此电缆。

## 软件

- **MPLAB® X IDE**——[www.microchip.com/mplabx](http://www.microchip.com/mplabx)  
建议使用 MPLAB X IDE v3.60+
- **XC16 C 编译器**——[www.microchip.com/xc16](http://www.microchip.com/xc16)  
需要 XC16 v1.30+。除非已应用“支持器件更新”，否则 PIC24FJ1024GB610 的器件头文件可能与早期编译器版本不兼容。
- **EZBL**——[www.microchip.com/ezbl](http://www.microchip.com/ezbl)  
需要使用 EZBL v2.00+；早期版本缺少这些练习所需的功能。  
EZBL 源文件原则上可在计算机上的任何位置解压缩，但父文件夹路径中不得有任何空格（GNU Make 不能完美支持具有空格的文件路径，通常会导致构建完全失败）。

---

通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

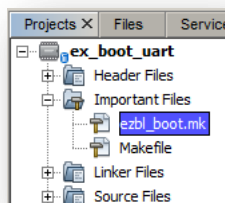
20 October 2017

第 2 页

## 练习 1——生成自举程序项目

在本节中，我们将简单地构建 UART 自举程序并将其编程到芯片中。EZBL 同时支持几乎所有 16 位 dsPIC33 DSC 和 PIC24 MCU（考虑到闪存的不同几何形状，大约有 454 个器件），因此需要最少量的项目配置和“自举程序独占”才能开始使用。

1. 在 MPLAB® X IDE 的“ezbl”文件夹中打开 **ex\_boot\_uart** 示例自举程序项目
  - a. 在 MPLAB X 中，单击菜单选项：File->Open Project（文件->打开项目）
  - b. 导航到解压缩 EZBL 下载内容的文件夹
  - c. 选择“ex\_boot\_uart”项目，然后单击“Open”（打开）按钮
2. 将处理器更改为 **PIC24FJ1024GB610**
  - a. 右键单击项目树视图窗格中的项目
  - b. 选择“Properties”（属性）
  - c. 右上角的“Device”（器件）下拉列表显示目标处理器。如果尚未设置为“PIC24FJ1024GB610”，需进行更改。
  - d. 单击“OK”（确定）按钮
3. 展开“Important Files”（重要文件）项目树文件夹分支，然后双击 **ezbl\_boot.mk** 文件。该 EZBL 特定的 makefile 脚本将在编辑器中打开。



4. 此文件的顶部包含**供应商、型号、名称和其他**的定义。将这些字符串更改为某些内容（任何内容），使其只表示您要添加自举程序功能的假定最终产品。

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

这些字符串连接在一起并通过 SHA-256 散列转换成一个全局唯一的自举程序标识常量，此常量存储在自举程序中。构建应用程序映像时，散列作为项目元数据传输，以便在擦除/损坏闪存中的已有任何应用程序之前，自举程序可以拒绝与最终产品不兼容的任何上传操作。我们不希望因不经意地将您的咖啡壶 v2 固件发送至“外婆”的“茶壶”而导致茶壶堵住。

---

通过 EZBL 进行自举动手实验


Copyright © 2017 Microchip Technology Inc.

20 October 2017

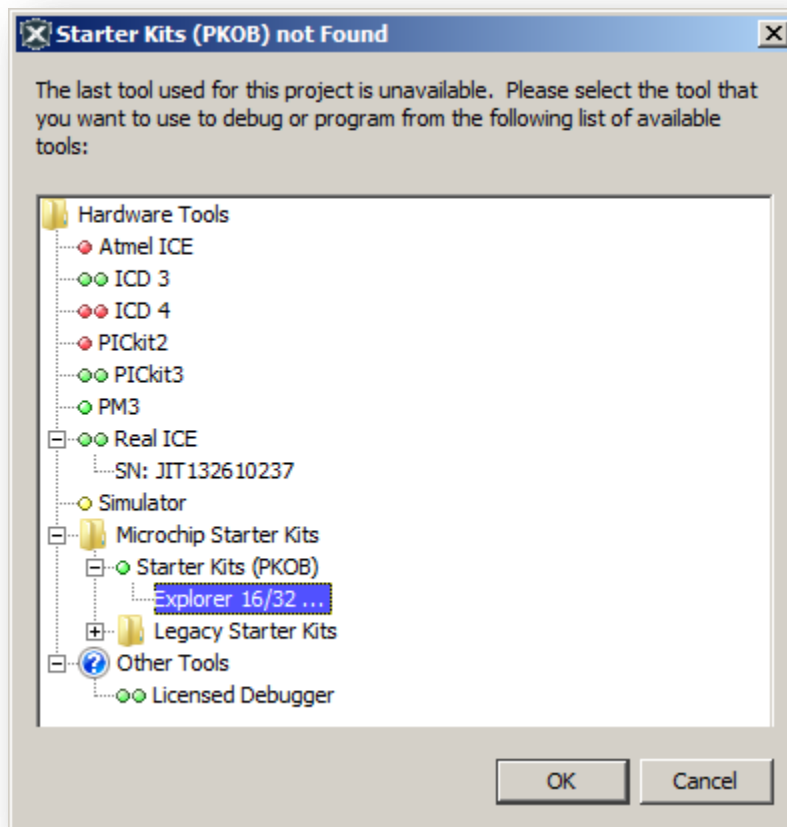
第 3 页

供应商/型号等的各个字段名称没有具体含义或意义，因为最终输出的是组合散列。但这些名称提供建议的输入数据，以确保将生成全局惟一的 `BOOTID_HASH`。

5. 确保您已将 USB 电缆连接到 Explorer 16/32 板左上角附近的 **PICkit on board** 的 micro-B 端口。这是 ICSP 调试和编程端口。

6. 编译/编程 (  )

- a. 如果提示您选择要使用的编程器，请确保为您的电路板选择 “Starter Kits (PKOB)” (入门工具包 (PKOB)) 选项。

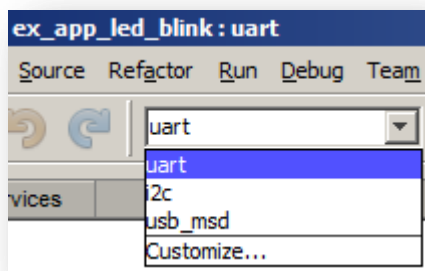


7. 验证 **LED D3** 是否正在快速闪烁 (8 Hz)。如果是，则说明您的自举程序正在成功执行并等待应用程序上传。

## 练习 2——加载示例应用程序

此练习将使用练习 1 的自举程序加载已经存在的演示应用程序。应用程序将使同一 D3 LED 闪烁，但速率要慢很多（即 1 Hz），以便目测是否成功。我们将需要对应用程序稍作修改以提供系统通信参数，然后进行编译以沿用先前定义的自举程序 ID 散列。

1. 如果您只使用一条 USB 电缆，请将其从“PICkit on board”连接器断开连接，并将其连接到 DC 桶形插孔附近的**串行 USB** 连接器。
  - a. 在某些系统上，可能需要安装驱动程序才能使用连接到“串行”连接器的 MCP2221A USB 转 UART 转换器芯片。此驱动程序是 Windows 的一部分，但需要使用从 Microsoft Windows 更新服务自动获取的已签名.inf 配置文件。
  - b. 如果系统没有互联网连接、必要的系统策略或其他方面无法自动安装所需的串行驱动程序，可从 <http://www.microchip.com/mcp2221a> 产品页面下载 MCP2200/MCP2221 Windows 驱动程序和安装程序并手动安装。
2. 关闭您在练习 1 中打开的自举程序项目。
  - a. 右键单击项目树视图窗格中的“ex\_boot\_uart”项目名称
  - b. 选择“Close”（关闭）菜单选项
3. 打开 EZBL 发行版附带的 **ex\_app\_led\_blink** 项目。这是现有的演示应用程序项目，几乎已配置完成，可上传到 EZBL 自举程序。
4. 如果尚未选择，将项目编译配置更改为 **uart**：



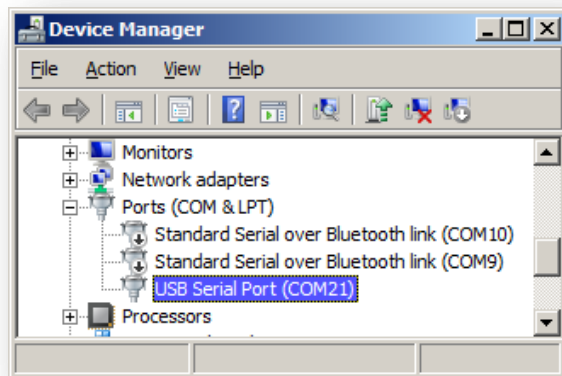
5. 按照[练习 1 的步骤 2](#)更改此“uart”编译配置的目标处理器，以匹配自举程序的目标“PIC24FJ1024GB610”器件。
6. 修改“Important Files”（重要文件）项目树分支中的 **ezbl\_app.mk** 以指向分配给 MCP2221A 的 COM 端口。（-com=COMxx 参数）。

```

25 # Post-build code to convert .hex to .bl2 and upload (if applicable):
26 ezbl_post_build: .build-impl
27 # Create a binary .bl2 file from the .elf file and also if a loadable exists, convert the unified .hex file.
28     @echo EZBL: Converting .elf/.hex file to a binary image
29     -test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" -nt "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.bl2"
30     @test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex" -nt "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex"
31
32 ifneq (,${filter default uart},${CONF})) # Check if "default" or "uart" MPLAB project build profile is used
33     @echo EZBL: Attempting to send to bootloader via UART
34     ${MP_JAVA_PATH}java -jar "${thisMakefileDir}ezbl_tools.jar" --communicator -com=COM21 -baud=230400
35 else ifneq (,${filter i2c},${CONF})) # Check if "i2c" MPLAB project build profile is used. If so, upload via I2C
36     @echo EZBL: Attempting to send to bootloader via I2C

```

如果您不知道 PC 上分配的通信端口名称，可以在 Windows 设备管理器中获取此信息：



尽管 OS 版本有所不同，但通常都可通过按下 **WINDOWS\_KEY + “R”** 并键入 **devmgmt.msc** 来打开此对话框

7. 清除和编译 (🧹) ——编译后，“ezbl\_app.mk”中的“ezbl\_post\_build”配方将使用自举程序自动对演示项目进行编程。
  - a. 如果编程成功，则 **D3 LED** 现在应仅以 1 Hz 的速率闪烁。

```

Output - ex_app_led_blink (Build, Load) x
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. 如果您在 Explorer 16/32 上切换“**MCLR**”按钮，您将看到 LED 快速闪烁 1 秒钟，然后新应用将以 1 Hz 的较慢速率闪烁。最初快速闪烁是由自举程序引起的，自举程


通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

20 October 2017


第 6 页

序短暂执行可提供一个“取消禁用窗口”，以免安装的应用程序会禁止或阻止自举程序访问。在经过没有任何 UART 活动的 1 秒间隔后，应用程序将启动正常器件操作。

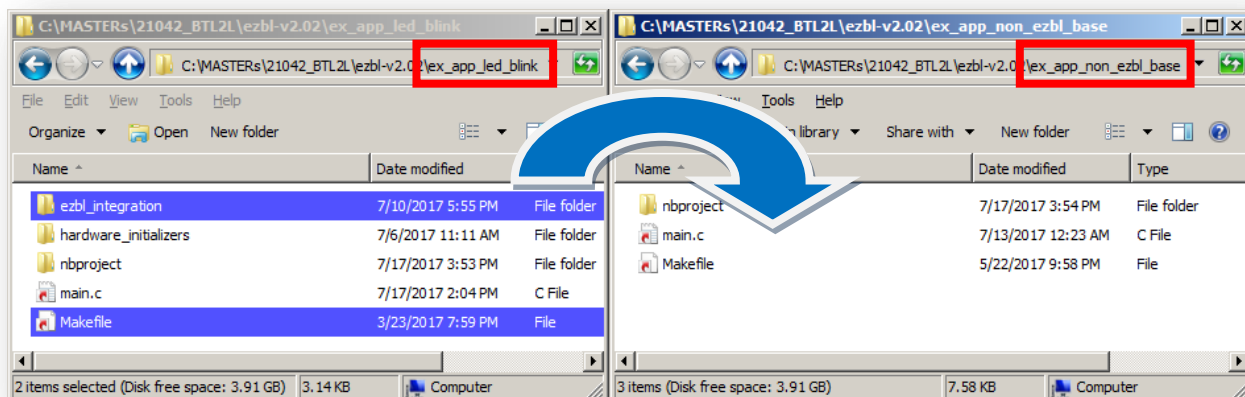
8. 双击打开“Source Files”（源文件）项目树分支中的 **main.c** 文件。我们将修改此文件以切换 LED 闪烁模式。
  - a. 找到 `LEDToggle(0x01);` 行（在 `main()` 中 `while(1)` 循环的 109 行或 109 行附近）。
  - b. 将其修改为 `LEDToggle(0x0F);`
9. **编译** () ——编译后，新的应用程序固件将再次发送到自举程序。正在执行的现有应用程序将暂停和擦除，因为此上传操作将包含匹配的自举程序 ID 散列数据。

## 练习 3——将 EZBL 添加到应用程序

在本节中，我们将采用一个新的或现有的应用程序项目（与 EZBL 无关），并对 EZBL 自举程序执行必要的项目更改，以使其兼容/可编程。

1. 由于这是一个学习练习，我们提供了与 EZBL 无关的现有应用程序项目。因此，首先测试应用程序项目十分有用，这样可了解成功配对自举程序和应用程序项目时将出现哪种行为。
  - a. 关闭您在练习 2 中打开的 **ex\_app\_led\_blink** 应用程序。
  - b. 打开 **ex\_app\_non\_ezbl\_base** 项目。该项目仅需要 XC16 编译器，而不需要其他相关性。
  - c. 断开 USB 电缆与“串行”连接器的连接，并将其移回到 **PICKit on board** 连接器，以实现 ICSP 工具支持的正常编程操作。
  - d. **编译/编程** (  )。如前所述，如果提示您选择要使用的编程器，请确保选择“Starter Kits (PKOB)”选项。
  - e. 将看到 LED 以 1 Hz 的速率按 **0x55** 模式闪烁。此操作的时序使用 `libpic30.h` 中包含的阻止 `__delay_ms()` 宏执行，该文件是 XC16 工具链的一部分。
2. 使用计算机的文件浏览器（即 Windows 资源管理器），转到 **ex\_app\_led\_blink** 项目文件夹，以获得一些必需的 EZBL 资料，包括二进制形式的自举程序。将以下两项复制到系统剪贴板中：
  - o **ezbl\_integration** 文件夹
  - o **Makefile** 文件

现在，导航到 **ex\_app\_non\_ezbl\_base** 项目文件夹并粘贴剪贴板内容。



- a. 正确的粘贴目标位置是“**ex\_app\_non\_ezbl\_base**”文件夹（启动新项目时将包含 MPLAB 创建的“**nbproject**”文件夹和“**Makefile**”文件）。文件夹路径中的多个现

---

通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

20 October 2017

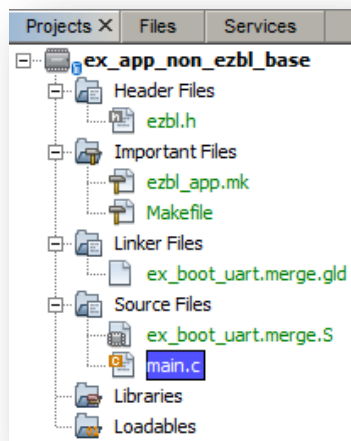
第 8 页



有项目文件夹均具有一个“.X”后缀，该后缀将是正确的目标位置，但为了最大程度缩短路径长度，EZBL 示例项目均删除了.X 后缀。

- b. 系统将提示您覆盖“Makefile”。同意此操作。
  - i. 如果您现有的项目已存在自定义 Makefile 内容，您可以将以下行粘贴到现有项目的底部，而不是覆盖 Makefile：

```
include ezbl_integration/ezbl_app.mk
```
3. 返回到 MPLAB® X IDE，将各种自举程序文件添加到项目树视图中：
  - a. 右键单击 **Header Files**（头文件），然后选择“Add Existing Item...”（添加现有项目...）
    - i. 导航到 **ex\_app\_non\_ezbl\_base\ezbl\_integration**
    - ii. 添加 **ezbl.h**
  - b. 右键单击 **Important Files**（重要文件），然后选择“Add Item to Important Files...”（将项目添加到重要文件...）
    - i. 添加 **ezbl\_app.mk**
  - c. 右键单击 **Linker Files**（链接器文件），然后选择“Add Existing Item...”
    - i. 添加 **ex\_boot\_uart.merge.gld**。这是我们编译“ex\_boot\_uart”自举程序项目时由 EZBL 自动生成的链接描述文件。
  - d. 右键单击 **Source Files**，然后选择“Add Existing Item...”
    - i. 添加 **ex\_boot\_uart.merge.S**。这是绝对链接自举程序的可执行映像的副本。实际上，这是“ex\_boot\_uart.production.hex”衍生文件，先转换为二进制形式，然后转换回简单的汇编源文件，以便应用程序的.hex 输出衍生文件同时包含您的自举程序和应用程序。
  - e. 您的项目现在应该包含以下所有文件：



4. 打开 main.c（在“Sources Files”中）并删除所有 `#pragma config` 语句。
  - a. 虽然 EZBL 可以支持使用新的应用程序上传来更新配置字，但这样做要求自举程序项目中未定义配置字。由于我们的“ex\_boot\_uart”项目确实包含配置字定义，因此该应用程序项目不能同时定义它们。



#### 5. 编译/编程 ( )


- a. 忽略编译 Output（输出）窗口中显示的 EZBL 通信错误。步骤 1 需要擦除所有闪存内容并对基本应用程序进行编程，因此没有自举程序能够与“ezbl\_app.mk”后编译 make 脚本进行通信。另外，如果仅使用一根 USB 电缆，电缆将即刻连接到“PICkit on board”接口，无需使用 PC COM 端口来尝试通信。
6. 将看到电路板 LED 再次以 0x55 模式闪烁。但是，如果您切换 **MCLR** 按钮，您将看到只有 LED D3 闪烁（以 8 Hz 的速率）的一秒复位间隔。这表示我们再次获得一个自举程序。
7. 再次修改“main.c” LED 闪烁代码。将：

```
LATA ^= 0x0055;
```

修改为：

```
LATA = (LATA<<1) ^ _LATA3 ^ 1;
```

请注意，这是一个“等于”赋值语句，而不是“异或等于”语句。

8. 将 USB 电缆从“PICkit on board”连接器移回串行连接器。
9. 编译 (  ) ——这将同时编译并通过 UART 烧写自举程序。即使我们在步骤 1 中擦除“ex\_boot\_uart”自举程序项目后始终没有明确对其重新编程，项目中存在的“ex\_boot\_uart.merge.S”源文件也会导致自举程序在步骤 5 之后重新出现在闪存中。

10. 自举成功还可通过以下方式识别：观察发光的 4 位履带从最低有效 LED 延伸出来，然后在进入最高有效端的 JP2 跳线后消失。由于它们能自行发出翠绿色的光，这表示可能存在放射性，最好不要使用。

如果您只看到 4 个左旋 LED，则也说明自举成功。

## 练习 4——使用 EZBL 库函数和宏

EZBL 具有许多库 API 函数和宏，用于实现自己的时序、通信、闪存操作和硬件抽象任务。与历史自举程序设计不同，这些相同的 API 可能在应用程序项目（甚至是未实现或使用 EZBL 自举程序的项目）中使用。

此外，当应用程序调用闪存中已存在的库 API 时（由于自举程序调用相同的功能），EZBL 允许应用程序使用相同的函数，而无需在物理闪存中复制它们。

此练习将重复使用自举程序中实现的 LED 功能、定时器功能和闪存擦除/编程 API，以实现具有仿真数据 EEPROM 状态持久性的其他复杂 LED 闪烁应用程序。相同的方法可用于访问各种其他功能，包括自举程序项目中的通信协议、文件系统或任何其他全局 API。

### 访问简单的 LED 功能


在本节中，我们将在应用程序中重复使用自举程序的简单 LED 功能。

1. 从练习 3 的项目继续，右键单击项目树视图窗口中的 **Libraries**（库），然后选择 “Add Library/Object File...”（添加库/对象文件...）
  - a. 添加 **ezbl\_lib.a**。该文件位于同一个 “ex\_app\_non\_ezbl\_base\ezbl\_integration” 文件夹中，之前的练习中从该文件夹中收集过文件。
2. 如果尚未打开 **main.c**，请将其打开进行编辑。
3. 包含 **ezbl.h** 头文件：
  - a. 在文件级范围内，在 “#include <libpic30.h>” 之后插入：  

```
#include "ezbl_integration/ezbl.h"
```
4. 删除 **main()** 函数顶部的 **TRISA/LATA/ANSA** 行。“ex\_boot\_uart” 自举程序项目已初始化了这些寄存器，以便在复位取消禁用间隔期间使 D3 LED 快速闪烁。这样一来，重新初始化相同硬件便会浪费使用该自举程序的应用程序项目中的代码空间。
5. 在 **main()** 函数的 **while (1)** 循环中，将您的 **LATA** 切换代码替换为：  

```
LATA = (LATA<<1) ^ _LATA3 ^ 1;
```

使用以下硬件抽象 API 调用：

```
LEDOn (0xF0) ;
```
6. 编译 ()。尽管 **LEDOn()** 没有实现任何代码，项目仍将正常编译、链接和上传。
  - a. 这是因为 “ex\_boot\_uart” 项目包含实现 **LEDToggle()** 并由自举程序项目调用的硬件初始化文件。
  - b. **LEDToggle()** 不是 **LEDOn()**，但代码仍然链接成功，因为 **LEDOn()** 在 “ezbl\_lib.a” 预编译归档库中实现，除了许多其他 API 外，该库还包含默认的 **LEDOn()** 实现。

该 LEDOn() 库 API 调用 LEDToggle(0x00) 以获得当前的 LED 状态，然后使用导出的位掩码来再次调用 LEDToggle() 以获得最终的 LEDOn(0xF0) 输出状态。

#### 7. LED 现在应该静态显示 0xF0 模式。

如果您看到 LED 呈 0xF0 模式，则表示您已成功重复使用在自举程序中实现（而非在闪存中复制）的 LEDToggle()，并从 ezbl\_lib.a 归档中引入新的 LEDOn() API。需要时，您可以进入自举程序项目的“ex\_boot\_uart\hardware\_initializers\pic24fj1024gb610\_explorer\_16.c”文件来查看 LEDToggle() 代码实现，如果您还想看到 LEDOn() 实现，则可以查看“ezbl\_lib\weak\_defaults\LEDOn.s”文件。

#### 完整解决方案

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

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

#### 访问定时器功能

EZBL 具有简单的调度程序模块，用于执行各种后台自举通信和定时事件。同样地，与 EZBL 中的其他任何功能一样，此功能也可以供应用程序利用。在本节中，我们将利用“NOW”定时功能创建一个重复任务代替现有的阻塞循环结构来切换 LED。

为此，我们将使用 NOW\_CreateRepeatingTask() 宏。该宏有三个参数：

- 指向 NOW\_TASK 对象的指针，在 RAM 中静态分配
- 指向要定期调用的函数的指针
- 函数调用间隔的时长。您可以使用 NOW\_sec，NOW\_ms 和 NOW\_μs 定义来获取正确的时间。例如，3 秒表示为：

3 \* NOW\_sec

对于此练习，我们将创建一个任务函数，使所有 LED 每 500 ms 切换一次。

1. 创建一个全局 NOW\_TASK 变量（选择您喜欢的任何名称）。

```
NOW_TASK ledToggleTask;
```

如果您的环境中不需要全局变量，请将其声明为 *static*。我们只需要一个指向以下结构的指针：该结构中的 API 指定其 RAM 保持不间断分配。


2. 创建一个函数以返回 `int`、将 `void` 作为输入并切换所有 LED。将该函数添加到 `main()` 函数以外的前方：

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

3. `main()` 初始化时，在 `while(1)` 循环之前，调用包含任务变量的指针、函数的指针引用和 500 ms 时间间隔的 `NOW_CreateRepeatingTasks()` 宏。

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

注意：toggleLED 之后没有括号。带括号将导致待调用函数向 `NOW_CreateRepeatingTask()` 宏传送返回值 0，而不是我们需要函数的地址（或句柄）。

4. 由于我们正在使用此新任务写 LED，因此应该删除前一个 `LEDOn()` API 调用。此外，`main()` `while(1)` 循环中的现有 `__delay_ms()` 调用浪费资源，应替换为 `Idle()` 调用。
5. 编译（）。将看到所有 8 个 LED 均在闪烁。

## 完整解决方案

```
#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();
    }
}
```

## 闪存中的仿真数据 EEPROM

由于自举程序包含已用于自举功能的闪存擦/写 API 阵列，因此可轻松将运行时状态数据存储存储在闪存中，这只需要最少的额外代码。此练习将无符号整数存储在闪存中，并根据所存储的值切换 LED。我们将使用 Explorer 16/32 上的 S4 按钮来更改运行时 LED 切换掩码，效果是在断电再上电后一直保持不变。

本节需要比先前练习更多的代码，因此将代码复制并粘贴到应用程序的 **main.c** 文件中可以加快此任务。在 Windows 资源管理器中，“**ezbl-v2.xx\help\EZBL Hands-on Bootloading Exercises - Exercise 4 Solution.c**”文件包含本节将介绍的所有代码。您还可在本节结尾处找到所有必需的代码，但是当使用剪贴板操作时，PDF 文件不会保留空格字符。至少返回到第 6 步，观察最终结果。

1. 首先，在对齐的边界分配至少一页闪存，以便我们可以擦除和重新编程仿真的 EE 数据，而不会与自举程序或应用程序中的任何内容重叠：

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

此宏静态分配/保留闪存，因此将其放置在全局/文件级范围。

第一个参数是我们分配给此存储器的任意名称，3072 是我们将保留的字节数，0x800（程序空间地址）是 PIC24FJ1024GB610 器件的闪存页擦除大小，它定义了对齐和填充要求，-1 是一个标志，表示链接器应当选择该对象的基址（如果需要，请将其更改为绝对闪存地址）。

- 我们还会在 RAM 中声明一个结构来保存 EE 数据，以实现高效的执行引用，并允许直接变量操作：

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

由于我们至少保留有 3072 个字节的闪存，因此可以在此结构中添加更多非易失性变量。但在此练习中，我们只需要一个变量。

- 要使用此新变量，请更新 toggleLED() 函数：

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

- 在 main() 的开始，我们需要读取 emuEEData 闪存内容并将其复制到 eeVars RAM 结构中。我们还应在首次上电时（emuEEData 将不包含任何内容）实现默认值。由于闪存擦除为全“1”，因此相当于执行 0xFFFF 检查：


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

- 最后，我们需要一种方法在运行时更改 eeVars 并将其提交给闪存，以便演示结构是非易失性的。在 main() 中，这一经过更新的 while(1) 循环将通过在每次按下 S4 按钮时使 LED 切换掩码递增的方式来实现此目的：

```

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. 编译 (  )。自举程序对应用程序编程后，将看到两个最低有效位 LED 按照 0x03 模式闪烁。
7. 按下 **S4** 按钮几次（或要减少的所需按钮次数）。每按一次，您都将看到 LED 闪烁模式增加二进制值 0x01。
8. 切换**电源**按钮几次和/或切换 **MCLR**。

成功实现将表明，对于后续每个电源周期和/或复位事件，最后显示的闪烁模式将保持持久定义。

在实际应用中，请考虑将 EZBL\_AllocFlashHole() 宏调用置于自举程序项目中而不是此应用程序项目中。在您的应用程序中分配 EE 闪存页时，每次自举程序编程新应用程序时，都将擦除您存储的任何 EE 数据。

通过将闪存空位移动到自举程序项目，EE 页可完成所有应用程序更新。因此，可以从自举程序和应用程序项目访问 EZBL\_ReadROMObj() 和 EZBL\_WriteROMObj() API。

不过，通过应用程序更新周期保持数据时，需跟踪哪些 eeVars 结构成员可能包含数据，哪些可能未初始化为全“1”，哪些可能对于刚自举的应用程序版本而言是新成员。例如，如果您定期添加新的非易失性变量且最终用户在最后一次安装 v1.0 版应用程序时编程 v1.3 版，情况会变复杂。这可能违反了在 v1.0 之后安装 v1.1、升级到 v1.2 并最终自举到 v1.3 的自然开发进程。

避免不正确或不连贯非易失性数据的一个策略是在引入新的 eeVars 时始终添加并保留首次运行的初始化程序代码。此外，始终不要对之前的 eeVar 结构元素执行删除、更改大小或重新排序的操作。而只需在 eeVars 的末尾添加新变量，并忽略不再需要的早期成员——维护其分配的存储空间。



## 完整解决方案

```
#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();
    }
}
```

## 练习 5——更新自举程序项目

如果要更新自举程序项目，要求应用程序使用的“ezbl\_integration”文件在项目之间匹配，以便链接器将应用程序闪存内容与自举程序保留的闪存页分离，并为应用程序引用的所有自举程序 API 提供正确的变量/函数地址。

编译 EZBL 自举程序时，它会自动为应用程序生成一个新的链接描述文件

（“ex\_boot\_uart.merge.gld”），因此如果自举程序在开发过程中更改大小，无需手动修改。您甚至可以对自举程序进行彻底更改（例如更改目标处理器），而且无需手动编辑.gld 链接描述文件。

不过，尽管有这种自动机制，您的应用程序项目中仍需存在正确的.gld 链接描述文件。在[练习 3](#)中，我们手动复制“ezbl\_integration”文件夹，以便将自举程序文件移动到应用程序项目中。但是，在测试各种自举程序修订版本时，此步骤十分不便且容易忘记。在本节中，我们将在每次重新编译自举程序时，更新自举程序 make 脚本，将自举程序文件复制到我们的应用程序项目中。

1. 打开自举程序项目 **ex\_boot\_uart**。您可能希望关闭现有应用程序项目，以免混淆两者。
2. 展开项目树视图窗口中“Important Files”分支。
  - a. 打开 **ezbl\_boot.mk**
3. 在文件顶部附近找到 appMergeDestFolders 变量声明：


```
# 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. 修改最后一行以指向新的应用程序文件夹而不是“ex\_app\_led\_blink”项目。这将使来自自举程序的所有将来合并衍生内容传播到应用程序。

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

需要时，您还可以添加新行，允许多个应用程序项目接收刚刚编译的自举程序衍生内容。  
例如：

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

5. 编译（）。

---

通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

20 October 2017

第 18 页

6. 为验证是否成功，请使用您的 OS 文件管理器，并验证

“ex\_app\_non\_ezbl\_base\ezbl\_integration\ex\_boot\_uart.merge.gld”文件中的时间戳是否已更新。上次修改的时间应与自举程序重新编译时的系统时钟时间相匹配。

现在应该可以对自举程序进行修改，而无需对应用程序项目执行任何操作（重新编译除外）。

不过请记住，只要您更改自举程序，便需要通过传统的 ICSP 编程工具对器件进行一次编程。这可以通过编程自举程序或应用程序项目之一来实现，因为应用程序项目包含自举程序映像。

如果您缺少 ICSP 编程步骤并重新编译了一个应用程序，然后通过闪存中的现有不匹配旧自举程序进行编程，您将收到一条自举程序上传错误消息：

```
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)
```

此外，由于应用程序调用的各种自举程序函数现在将试图跳转到不正确的 API 地址，因此应用程序代码通常会在执行期间表现异常或发生严重故障。通过 ICSP 编程工具编程正确的自举程序后，此错误会自动消失。

## 练习 6——将未使用的符号保存在自举程序中

EZBL 自举程序项目中有多个自举程序可能不使用，但应用程序可能要调用的 API。例如，在具有用户可插入批量存储功能的“存储器”类型的自举程序中，您可能不会将 `Format()` 函数用于任何自举操作，但可能需要将它用于后续的应用程序项目。由于该函数不会在自举程序中的任何位置调用，因此自举程序项目中的 XC16 优化选项将导致未引用的 `Format()` 函数被删除。

有几种方法可以告知 EZBL/XC16 保留未使用的功能，以供应用程序使用。此练习将涵盖 EZBL 选项、讨论 XC16 选项并最终探讨如果您已经进行了生产并且不能再修改自举程序应执行哪些操作。

### EZBL\_KeepSYM()

`ezbl.h` 定义了一个名为 `EZBL_KeepSYM()` 的宏，用于指示工具链保留命名符号，即使它没有被（自举程序）项目调用或引用时也如此。


在此练习中，我们将通过更新代码来使用已定义但未在自举程序项目中调用的 `ButtonRead()` 函数。我们将指示链接器保留此自举程序的函数，供应用程序后续使用。

### 验证它不能开箱即用


1. 如果您已将（当前命名错误的）`ex_app_non_ezbl_base` 应用程序项目关闭，请重新打开。
2. 用以下代码替换 `main()` 和 `while(1)` 循环：

```
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();
}
```


此代码已删除 (`!_RD13`) 条件代码并用 `ButtonPeek()` 硬件抽象 API 替换。

3. 编译 (  )。您应当收到一条编译错误，指示“**undefined reference to `\_ButtonPeek'**”（未定义对“`_ButtonPeek`”的引用）。不要担心，这是意料之中的结果。在自举程序中，始终未调用或引用 `ButtonPeek()` 函数，因此 XC16 通过去除 `ButtonPeek()` 的代码来优化闪存占用空间。

### 修改自举程序以保留 ButtonPeek() 函数

4. 打开您的 **ex\_boot\_uart** 自举程序项目。
5. 在 **main.c** 文件的顶部，在 `#included ezbl.h` 后，添加此宏语句：  
`EZBL_KeepSYM(ButtonPeek);`
6. 如果仅使用一根 USB 电缆，则将其移回 **PICkit on board** 连接器。
7. 编译/编程 (  )。

### 验证应用程序是否已链接并处于工作状态

8. 返回 **ex\_app\_non\_ezbl\_base** 应用程序项目。
9. 确认到串行连接器的 USB 连接。
10. 编译 (  )。应用程序现在应链接成功，并通过自举程序自行编程。如果仍然收到未定义的引用错误，则练习 5 可能未成功完成，应检查  
“`ex_app_non_ezbl_base\ezbl_integration\ex_boot_uart.merge.gld`”（和  
“`ex_boot_uart.merge.S`”）文件的时间戳。
11. 现在调用 `ButtonPeak()`，只要按下板上的任何按钮（**S3、S6 或 S4**），LED 切换模式就会增加。之前，只有 **S4** 触发这种行为。

注：按下 **S5** 不会执行任何操作。**S5** 与 **D10 LED** 共用相同的硬件 I/O 引脚。因此，在此代码中，**RA7 GPIO** 输出功能会阻止使用输入功能。

### 保留所有部分

`EZBL_KeepSYM()` 练习演示了如何保留一个特定的函数。这将自动保留所保留函数引用的任何函数和变量，因此您不会丢失任何相关性。但是，如果您希望保留很多相互关联的代码，您可能会无意间忘记保留一些函数，只有在您生产一段时间后才会发现它们丢失，然后开始处理更新的应用程序修订版。在本节中，我们将讨论如何指示 **XC16** 工具链来避免同时删除未使用的函数和变量。

在大型堆栈应用程序中，保留所有未使用的代码可能特别有用，因为在这些应用中，很难预测您哪天将在哪些应用程序项目（例如文件系统函数）中使用哪些符号。这种方法的缺点是，无论是否需要，都会保留很多函数，这将导致自举程序长度增加，有时甚至会很明显。

要将所有未使用的函数保留在自举程序项目中，您需要更改项目链接器选项。转到自举程序的项目属性，然后选择 **xc16-ld** 部分。取消选中 “**Remove unused sections**”（删除未使用的部分）框。如果选中此框（EZBL 自举程序项目中的默认设置），所有未引用的部分都会被视为死代码并被删除。

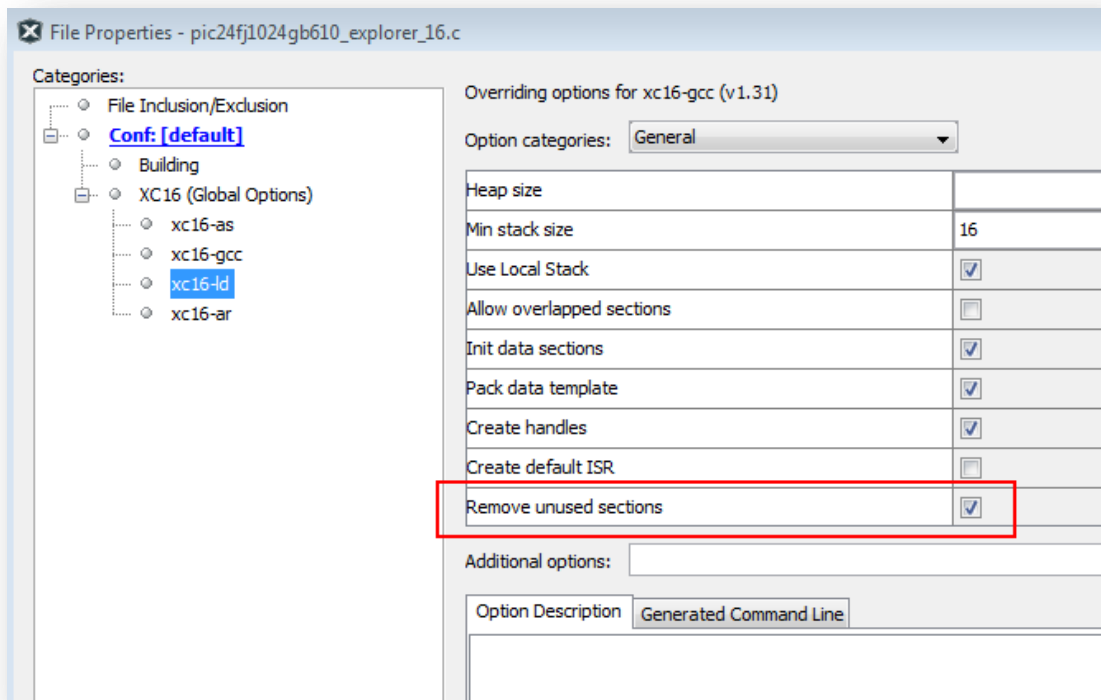
---

通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

20 October 2017

第 21 页



## 当针对生产冻结自举程序时

当您在开发自举程序时，之前的选项仍将适用。但是，一旦进入生产阶段，就无法再通过重新编译自举程序来保留在后续应用程序更新中可能最终需要的各种 API。

在这种情况下，解决方案是在您的应用程序项目中实现所需的 API。自举程序项目中的所有函数和全局变量导出为“弱”属性的函数/变量，以供在应用程序项目中使用。这意味着如果应用程序项目中存在相同名称的定义，链接器将优先忽略自举程序实现。应用程序将按照链接器优先级的降序生成对以下内容的引用：

1. 最高优先级：在同一应用程序项目中定义的全局或静态函数或变量
2. 自举程序项目中的“弱”属性全局函数或变量（函数/变量地址来自“ex\_boot\_uart.merge.S”源文件）。
3. 0（空指针），如果函数原型或外部变量声明为“弱”属性。
4. 查看函数或变量的所有预编译的.a 归档库，当发现匹配时停止。其中包括 MPLAB X 项目中包含的“ezbl\_lib.a”归档以及由 XC16 工具链隐含的归档，如 libpic30-elf.a 和 liblega-c-elf.a 等。归档搜索顺序不易控制。

---

通过 EZBL 进行自举动手实验

Copyright © 2017 Microchip Technology Inc.

20 October 2017

第 22 页




5. 最低优先级：以“undefined reference to ‘\_xxx’”（对“\_xxx”的引用未定义）链接器错误消息中止。

请注意，尽管可以向自举程序中一直不存在的应用程序添加功能，并实现防止应用程序重用自举程序弱属性函数的新功能，但需要注意确保在处理复杂堆栈函数时存在调用树一致性。如果所有函数并非完全同意变量的用途，则不需要新应用程序定义的函数来修改在其他旧自举程序定义函数中使用的全局变量。在最坏的情况下，可能需要包含复杂代码或堆栈的全新副本，以忽略应用程序中的所有现有自举程序函数和变量。

## 练习 7——应用程序版本控制

EZBL 自动进行版本编号。每次编译应用程序项目时，版本号都将自动递增。自举程序还执行版本号检查，因此，需要时，自举程序可以实现禁止向后修订的策略。对于存储器类型的自举程序（例如：U 盘），提供给自举程序的应用程序固件映像还可以持续存在。在这种情况下，如果提供的版本与闪存中已存在的应用程序相匹配，则将其忽略较为有利。



在此练习中，我们将操作版本号并观察自举程序如何处理提供的应用程序映像。

1. 打开 **ex\_app\_non\_ezbl\_base** 应用程序项目。
2. 打开“Important Files”部分中的 **ezbl\_app.mk** 文件。
3. 您将看到 **APPID\_VER** 的一些定义。提供主要版本、次要版本和修订版本字段。主要版本和次要版本通过在此处键入相应内容来手动控制。版本号则在每次编译项目时，通过 EZBL 编译工具自动递增。
4. 按下 **编译** () 按钮。将看到版本号递增 1（需要时，可以再多试几次）。
5. 将 **APPID\_VER\_BUILD\_INC** 变量更改为 0。
6. 按下 **编译** () 按钮。将看到版本号保持不变。该零增量值在预发行项目开发过程中十分有用，因为“ezbl\_app.mk”文件不会不断变化且被标记为“脏文件”，并准备好检查源版本控制资源库。
7. 将 **APPID\_VER\_BUILD\_INC** 变量更改为-5。
8. 按下 **编译** () 按钮。注意结果。

为避免意外的版本降级，我们需要在自举程序项目中添加链接器标志，以指示其检查应用程序的版本号。

9. 打开 **自举程序项目**，并将其设置为 **MPLAB 主项目**
10. 打开 **main.c**
11. 将以下行添加到 **main.c** 文件的顶部（或者取消注释示例中已存在的行）：

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

12. 确认到 **PICkit on board** 连接器的 USB 电缆连接。
13. 编译并 **烧写** 新的自举程序 ()。
14. 确认到用于自举的 **串行连接器的 USB** 电缆连接。
15. 返回 **ex\_app\_non\_ezbl\_base** 应用程序项目，将其设置为 **MPLAB 主项目**。
16. 按下 **编译** () 按钮。这会将一个初始应用程序置于闪存中，作为版本参考。

---

通过 EZBL 进行自举动手实验


Copyright © 2017 Microchip Technology Inc.

20 October 2017

第 24 页



17. 如果将“ezbl\_app.mk”中的 **APPID\_VER\_BUILD\_INC** 设置为-5，则下一次编译/上传操作将采用降级版本控制顺序。您可以通过手动递减 **APP\_ID\_VER\_BUILD**、**APPID\_VER\_MINOR** 和/或 **APPID\_VER\_MAJOR** 来获得相同效果。

18. 清除并编译 ()。注：与充分编译的先前步骤不同，使用 **Clean**（清除）来确保.elf/.hex/.bl2 文件重新生成（或者，在 main.c 中输入任意字符、删除字符，然后发出编译命令。使 main.c 更新其时间戳并触发一个新的链接事件。）。

通常，make 将跳过重新链接相同项目的过程，因为在步骤 8 之后没有源输入文件发生更改。APPID\_VER 编号自身从“ezbl\_app.mk”文件传递给编译输出，方法是它们作为调用链接器的命令行上的定义传递。因此，更改的值在 make 系统中不可作为“干净”值或“脏”值跟踪。

19. 当编译输出发送到自举程序时，自举程序将拒绝应用程序更新尝试，而不是终止并擦除现有应用程序。在步骤 16 中编程的现有应用程序将继续执行而不会中断。

```
Upload progress: 10%      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\ci12128\AppData\Local\Temp\ezbl_comm_log.txt
```

如果将“ezbl\_app.mk”中的 **APPID\_VER\_BUILD\_INC** 更改回 1，并手动将修订版本、次要版本或主要版本增加到 >= 步骤 16 的版本，则可在下一次清除并编译或通过链接事件编译时重新编程。

**注：** EZBL\_NO\_APP\_DOWNGRADE 功能不是安全功能，其作用是防止可能干扰“实时更新”数据保存或仿真数据 EEPROM 内容连续性的意外降级。

如果用户启动自举程序重新编程序列，然后在更新完成之前人为对器件进行断电再上电，则闪存的内容将在没有安装有效应用程序的情况下保留。此时，自举程序将接受发送到其中的任何有效应用程序映像，而与其 APPID\_VER 修订版本无关。只有继承的 BOOTID\_HASH 数据必须在此无应用程序状态下匹配。