

## Microchip Easy Bootloader Library

The Microchip Easy Bootloader Library, or EZBL for short, is a library of software building blocks and example projects facilitating creation of Bootloaders and compatible Applications. All 32-bit PIC32MM devices and nearly all 16-bit target PIC® MCU and dsPIC® DSC processors are supported with multiple communications protocols and bootloading topologies. Using a pre-compiled API archive and build-time automation tools, numerous manual code development scenarios and bootloader restrictions are eliminated as compared to historical bootloader App Notes and Code Examples. EZBL brings code sharing and advanced feature sets not previously feasible with manually developed bootloader projects.

### *EZBL Highlights*

#### **Targets supported**

- All 32-bit PIC32MM and 16-bit PIC24/dsPIC33 families except PIC24F[V]xxKxx devices

#### **Code reuse**

- Transparent Bootloader function, ISR and variable access from executing Applications – prevent code duplication
- Existing code can be integrated into a Bootloader or Application project without significant changes
- Push and Pull topology support with minimal Application down-time or user interaction

#### **Automatic linker script generation**

- No .gld/.ld file maintenance or understanding of GNU ld syntax

#### **Decoupled communications**

- 2-wire UART, I<sup>2</sup>C Slave, and USB Host Mass Storage Device class ("thumb drive") protocols exemplified
- Large API set to facilitate user and 3rd party protocol additions

#### **Robust self-preservation**

- Flash erase/write routines self-aware of Bootloader geometry
- Bootloader will not erase or corrupt itself when externally presented with destructive data
- Application images created for someone else's EZBL Bootloader are automatically rejected
- Will not attempt to execute a partially programmed Application
- CRC32 image integrity/communications checking with reusable APIs

#### **Interrupt vector management**

- Bootloader and Application can use any Interrupt with individual run-time vector selection
- No AIVT hardware or manual allocation required

#### **Application support functionality**

- Optimized general purpose 64-bit 'NOW' time measurement and task scheduling API
- Asynchronous software FIFO buffering for high speed serial communications
- Flash erase/write API for non-volatile data storage (emulated EEPROM in flash)

#### **Designed for performance without hardware luxuries**

- Latency adaptive software flow control tolerates Bluetooth or Internet tunneling delays without hardware RTS/CTS or sideband signaling

#### **Code secrecy compatible**

- Operation unaffected by standard ICSP Code Protect
- Bootloader does not expose program memory or RAM contents for external read back

#### **Full source with no-cost Microchip license**

- No GNU GPL code contamination or 3<sup>rd</sup> party code requiring independent licensing
- Portable command line communications executable ready for branded GUI wrapping and redistribution

## What's in This Document

This document starts with an EZBL overview, provides usage information for all embedded and host projects supplied in the EZBL distribution and proceeds to cover other reference information that may be relevant in a fully implemented communications and bootloading solution.

1. MPLAB® X Projects covered in this help:

- [ex\\_app\\_led\\_blink](#) – Application to test single partition UART, I<sup>2</sup>C and USB MSD bootloaders
- [ex\\_app\\_led\\_blink\\_pic32mm](#) – Application to test UART and USB MSD bootloaders on PIC32MM targets
- [ex\\_boot\\_uart](#) – UART Bootloader
- [ex\\_boot\\_uart\\_pic32mm](#) – UART Bootloader (PIC32MM)
- [ex\\_boot\\_i2c](#) – I<sup>2</sup>C Slave Bootloader
- [ex\\_boot\\_usb\\_msd](#) – USB Host - Mass Storage Device class Bootloader
- [ex\\_boot\\_app\\_blink\\_dual\\_partition](#) – Combined UART Bootloader + Application for Dual Partition
- [ex\\_app\\_live\\_update\\_smps\\_v1](#) – SMPS Application for Dual Partition w/Live Update
- [ex\\_app\\_live\\_update\\_smps\\_v2](#) – SMPS Application using the "Preserve All" linking model
- [ex\\_app\\_live\\_update\\_smps\\_v3](#) – SMPS Application using the "Preserve None" linking model
- [ex\\_app\\_non\\_ezbl\\_base](#) – Application not related to EZBL for EZBL Hands-on Bootloading Exercises
- [ezbl\\_lib](#) – Source code for pre-compiled APIs used in all 16-bit examples and Bootloaders
- [ezbl\\_lib32mm](#) – Source code for pre-compiled APIs used in all 32-bit examples and Bootloaders

2. PC-side automation and communications projects:

- [ezbl\\_tools](#) – Java source code for all compile-time automation, code generation and .elf/.bl2 processing
- [ezbl\\_comm](#) – C source code for transferring .bl2 firmware images to UART/I<sup>2</sup>C bootloaders using a PC or other host platform

Additionally, this help includes topics common to many projects simultaneously:

- [What's Found Elsewhere](#)
- [Typographical Conventions](#)
- [Overview](#)
  - i. [Topology](#)
  - ii. [Size](#)
  - iii. [Speed](#)
  - iv. [General Usage](#)
- [Bootloader Interrupt Handling](#)
  - i. [16-bit, PIC24/dsPIC Interrupt Handling](#)
  - ii. [32-bit, PIC32MM Interrupt Handling](#)
- [Communications Status Codes and Error Messages](#)

## What's Found Elsewhere

- Release Notes – See "[help\EZBL Release Notes.htm](#)"
- License Agreement – See "[help\EZBL License \(112114\).pdf](#)"
- UART and I<sup>2</sup>C communications protocols – See "[help\EZBL Communications Protocol.pdf](#)"
- .BL2 File Format – See "[help\EZBL BL2 File Format Specification](#)"
- ezbl\_lib.a and ezbl\_lib32mm.a API reference – See "[ezbl\\_lib\ezbl.h](#)"

Most APIs can also generate a context-sensitive help dialog when ezbl.h has been #included, the API name is typed and CTRL+SHIFT+Space are pushed within the MPLAB® X IDE code editor window.

## Getting Started

To get up and running with a UART bootloader on a PIC24FJ/PIC24H/PIC24E/dsPIC33F/dsPIC33E/dsPIC33C target, see the "[help\EZBL Hands-on Bootloading Exercises.pdf](#)" file. Exercises 1 and 2 contain a detailed step-by-step walkthrough of the typical steps needed to build and test an EZBL Bootloader + Application pair. At the end of exercise 4, you will not only be programming a new Application project using your Bootloader, but also have ancillary services, such as a task scheduler and emulated data EEPROM storage capabilities implemented.

For those interested in an I<sup>2</sup>C Slave bootloader, the same exercises can be followed, but with the nearly identical *ex\_boot\_i2c* Bootloader project used in place of *ex\_boot\_uart*.

Implementers of more exotic bootloaders, such as USB Mass Storage, Dual Partition and Live Update are encouraged to run through the exercises as well. Despite some operational differences with these varied bootloader types, most of the insights gained by testing the traditional single partition UART bootloading scenario will have applicability towards your final bootloading goals.

If interested in a PIC32MM target, most of the UART exercises will still be applicable. However, the *ex\_boot\_uart\_pic32mm* Bootloader project and *ex\_app\_led\_blink\_pic32mm* Application project would be used instead.

## Typographical Conventions

Various capitalization, color and font formatting instances appear in this document. In most cases, these indicate specific entities and names.

Instance Example(s)	Meaning
<u>Application</u> (instead of <u>application</u> ) <u>Bootloader</u> (instead of <u>bootloader</u> )	MPLAB Application project intended to be (re)programmed via a Bootloader project. Generalized concepts of bootloaders, products or PC software will typically appear as lowercase words.
<a href="#">help\EZBL Library Documentation.pdf</a> <a href="#">ezbl_tools.jar</a> Makefile .gld/.ld .elf .merge.S	Path, filename, file extension; often something distributed with EZBL or generated while building a project. May contain <b>[*]</b> wildcard elements or names inside brackets to indicate alternate variations for the file. Many filenames are abbreviated by omitting brackets and name prefixes, when the prefix is project name dependent and singular in nature or an abstract generalization covering any file with a matching file extension. A forward slash may exist to denote an alternate, often mutually exclusive case.
<code>_U2RXInterrupt()</code>	Source code or identifier
<i>ex_boot_uart</i>	MPLAB project name
<a href="#">Default IGT Remapping</a>	Hyperlink, usually to something found elsewhere in this document when applied to ordinary text, or on the Microchip.com web site when applied to a part number.

## Overview

### Topology

EZBL implements a file-oriented bootloading topology. A file-based bootloader consumes a regular file as input and internally decodes it, carrying out all erase, program, verification and integrity checking steps internal to the bootloader, taking care not to destroy itself when anything goes wrong.

This topology differs from other bootloader topologies which operate in a command-oriented mode. In command-based bootloaders, an already configured, bigger/smarter host device like a PC Application will actively generate individual "erase address x", "program address x with data y", "read back address x" commands and receive feedback from the bootloader prior to advancing to subsequent steps.

The primary advantage of file-based bootloaders is that they are very flexible. The host PC doesn't have to know anything about the target device. Indeed, the update image provider need not even have a CPU. A complete Application update can take place simply by plugging a USB thumb drive into the target device.

File oriented bootloaders encompass a class referred to as "memory bootloaders." These consume the contents of a memory located internally, onboard or external/off-board via a simple communications channel. They are well suited to Over-The-Air (OTA) bootloading, where the communications channel may have unreliable transport characteristics and require a lot of code to initialize/operate. Often these bootloaders implement a staged approach where the OTA communications stack has been offloaded to the existing Application project. The Application writes a new firmware image to an onboard memory and invokes the Bootloader only after the image is fully received and validated. The Bootloader then erases the Application and reprograms internal flash, requiring only simple communications code to read from the onboard storage medium. These systems maintain a high assurance that all Bootloading steps will complete successfully since they do not depend on the wireless medium.

When a file-oriented bootloader does source data from an off-board communications partner and writes directly to flash, the communications protocol can be simple, requiring only a flow control mechanism (to avoid receiving continued file data while the CPU is stalled for hardware flash erase/programming durations). File based bootloaders are therefore also well suited to being ported to communications standards of higher complexity, such as DFU over USB or TFTP/FTP over UDP/TCP.

EZBL's serial UART and I<sup>2</sup>C Slave bootloader examples implement an EZBL-specific method of software flow control – see [help\EZBL Communications Protocol.pdf](#). They do not require CTS/UTS signaling hardware or long SCL clock stretching periods that would block other traffic on an I<sup>2</sup>C bus.

### Size

EZBL allows Application projects to call Bootloader implemented code functions and access global variables instanced within, negating a typical need to duplicate some code in both Bootloader and Application projects. Consequently, the resource footprint of an Application project can shrink when using an EZBL Bootloader, possibly by a lot if the Bootloader has a lot in it.

The code sharing aspect makes it potentially difficult to interpret what the total memory cost of using EZBL will be by evaluating only the reserved geometry of the bootloader. Nonetheless, these are some typical sizes:

- 16-bit PIC24/dsPIC Bootloaders
  - UART – 9Kbytes flash, 284 bytes static RAM (96 + 32 bytes as RX/TX communications FIFOs)
  - I2C Slave – 9Kbytes flash, 300 bytes static RAM (128 + 16 bytes as RX/TX communications FIFOs)
  - USB Host MSD – ~33Kbytes flash, ~3.75Kbytes static RAM (512\*2 sectors + 2000 byte heap)
- 32-bit PIC32MM Bootloaders
  - UART – 8Kbytes flash, 292 bytes static RAM (96 + 32 bytes as RX/TX communications FIFOs)
  - USB Host MSD -- ~ 34Kbytes flash, ~6.25Kbytes static RAM (512\*2 sectors + 3000 byte heap)

FIFO sizes are configurable down to 64 bytes of RAM for RX and 6 bytes for TX. Most USB Mass Storage flash and RAM requirements are consumed by the USB Host software communications stack and FILEIO file system library code, both of which can be negated in Application resource footprints if the Application projects will need USB MSD media accessibility as well.

**Note:** Due to the large flash size of the USB Host MSD bootloader, which would normally fill an entire PSV window on 16-bit devices, one or more instances of the `EZBL_SetAppReservedHole()` macro is made in the XC16 flavors of `ex_boot_usb_msd` example bootloader projects. This causes MPLAB X IDE's Dashboard window Program memory size to over report the real size of the Bootloader project by up to 15,360 Words (45Kbytes). The macro marks address ranges as occupied to ensure no project code or const objects get linked into the address range, but as no code or const data is actually placed there, this reserved size must be discounted when evaluating the size of this bootloader type.

Likewise, `EZBL_SetAppReservedNearHole(512)` is exercised to ensure 'near' RAM is available for future Applications. This increases the apparent Bootloader RAM footprint by 512 bytes, when in fact, all 512 bytes are available for use in Application projects.

Finally, the Bootloader example projects contain a number of `EZBL_KeepSYM()` and `EZBL_KeepSYMIfPresent()` macro calls to force extra USB/FILEIO code and associated RAM variables to be linked into the Bootloader project, even though the Bootloader project does not call most of them. This permits the tagged APIs to be used in subsequent Application projects without duplicating any USB or FILEIO code in the Application. These macros should be removed when determining Bootloader resource requirements.

## Speed

Bootloading throughput varies greatly depending on communications baud rates, RAM RX FIFO buffer size, round-trip communications latency, processor execution clock, geometry of available flash, existing flash programmed/erased state, Bootloader size and performance characteristics of external hardware/communications bridges providing new Application images.

## UART

Typical, measured performance for a few UART configurations are:

Processor, Clock, RX FIFO Size	USB Bridge, Baud Rate, RX	.bl2 File Transfer Size	Flash needing to be: Erased, Blank Checked, Programmed	Time, Throughput
PIC24FJ1024GB610, 16 MIPS, 96 bytes	MCP2221A 230.4 kbps	1,031 KB	3 KB 1,019 KB 1,019 KB	58.3 seconds 17.7 KB/s
PIC24FJ1024GB610, 16 MIPS, 512 bytes	MCP2200 571 kbps	1,031 KB	3 KB 1019 KB 1,019 KB	25.6 seconds 40.4 KB/s
dsPIC33EP512MU810, 70 MIPS, 96 bytes	MCP2221A 230.4 kbps	537 KB	0 KB 525 KB 525 KB	28.9 seconds 18.6 KB/s

dsPIC33EP512MU810, 70 MIPS, 512 bytes	MCP2200 761 kbps	537 KB	0 KB 525 KB 525 KB	10.8 seconds 49.9 KB/s
PIC32MM0064GPL036, 24 MIPS, 96 bytes	MCP2221A 230.4 kbps	69.9KB	2 KB 56 KB 56 KB	3.7 seconds 18.7 KB/s
PIC32MM0256GPM064, 24 MIPS, 96 bytes	MCP2200 1,000 kbps	261.9KB	250 KB 0 KB 248 KB	8.3 seconds 31.4 KB/s
PIC32MM0256GPM064, 24 MIPS, 96 bytes	MCP2200 1,000 kbps	261.9KB	2 KB 248 KB 248 KB	6.2 seconds 42.5 KB/s

A baud rate in excess of ~761Kbps for 16-bit devices operating  $\geq 24$  MIPS and ~571Kbps for PIC24F devices at 16 MIPS will generally not work on normal, single partition bootloader types. Without adding additional code to support operation with a DMA, the 4-byte deep hardware RX FIFO on UART peripherals will overflow during a minimum NVM word/double word flash programming operation at higher baud rates while the CPU is blocked.

PIC32MM devices implement an 8-byte deep hardware RX FIFO on UART peripherals and thus can tolerate baud rates above 1 Mbit/s.

### I<sup>2</sup>C Slave

I<sup>2</sup>C Bootloaders will experience slower throughput when using the MCP2221A, despite a relatively fast 400kHz signaling rate. This speed reduction is largely attributable to the MCP2221A itself being optimized for UART performance (at the expense of I<sup>2</sup>C speed due to limited internal RAM and relatively large USB packet transfers). I<sup>2</sup>C Bootloaders should achieve at least twice the throughput, essentially in line with UART Bootloading performance, when communicating with a faster I<sup>2</sup>C master node.

Processor, Clock, RX FIFO Size	USB Bridge, SCK Clock	.bl2 File Transfer Size	Flash needing to be:		Time, Throughput
			Erased, Blank Checked, Programmed		
PIC24FJ1024GB610, 16 MIPS, 120 bytes	MCP2221A 400kHz	1,031 KB	0 KB 1,019 KB 1,019 KB		91.0 seconds 11.3 KB/s

### USB Host MSD

Throughput has been observed ranging from 38.2KB/s up to 47.1KB/s for PIC24F devices operating at 16 MIPS and from 44.1KB/s up to 58.3KB/s on dsPIC33E/PIC24E devices at 70 MIPS. Performance will be lower when a complete dry run is performed, verifying all aspects of the presented image before erasing an existing Application in flash.

Realized performance may also vary appreciably as sector read latency and initial USB enumeration time varies with mass storage media.

## General Usage

Starting on the PC, EZBL, as input, takes an MPLAB Project with some variation of communications and Run-Time Self Programming (RTSP) API calls in it and converts it into a Bootloader project. When the project is built, `ezbl_tools.jar`, a console-based executable, adds processing to the toolchain make/compile/link flow to generate project-specific content necessary to create a functional Bootloader.

Typically, when a project **Build** command is invoked, MPLAB® X IDE launches GNU `make`, which triggers these build steps:

1. Preprocess, compile, and assemble all `.c` files into relocatable `.o` object files
2. Preprocess (`.S`) and assemble all `.s` files into relocatable `.o` object files
3. Pass all `.o` files + `.a` archive libraries (in the project or supplied with the toolchain) + the project's `.gld` or `.ld` linker script (if any) to the toolchain linker. The linker then:
  - a) Chooses where in flash and RAM to place all functions and objects
  - b) Identifies the location of extern parameters for all opcodes referencing addresses symbolically
  - c) Generates `.dinit/.rdinit` flash section(s) to initialize all global and static variables at device reset, prior to executing `main()`. Additional symbols are generated to initialize the stack pointer and other CPU control SFRs.
  - d) (Optionally) throws away any code or RAM variables/sections which existed in the `.o/.a` input files, but which was never accessed in the project's `_reset/main()` call tree or any call tree for an ISR or 'keep' attribute function. This is dead code.
  - e) Generates an `.elf` output file, which is the final executable image with debug information
4. For Production builds, executes a tool chain converter utility to extract only the flash contents from the `.elf` file and generates a `.hex` file from it.

When a typical EZBL Bootloader project is built in MPLAB, the build flow instead follows this augmented recipe (grey text indicates a step from the normal build flow):

1. Invoke `ezbl_tools.jar` to modify IDE generated makefiles to reexecute `ezbl_tools.jar` in future steps. During this step, `ezbl_tools.jar` also collects information about the project and target device information. The following types of data are collected and saved for future steps:
  - a) Path to XC16/XC32 toolchain 'bin' folder that is going to be used for building
  - b) Target device part number
  - c) Device flash geometry
  - d) Device IVT (Interrupt Vector Table) entries and names
  - e) Minimum erasable block size (NVMCON<NVMOP> Page Erase size)
  - f) Minimum programmable block sizeAdditionally, the `BOOTID_HASH0..3` symbol values in the `ezbl_boot.mk` script are recomputed if the `BOOTID_VENDOR`, `BOOTID_MODEL`, `BOOTID_NAME` and/or `BOOTID_OTHER` strings have been modified since last build.
2. Preprocess, compile, and assemble all `.c` files into relocatable `.o` object files
3. Preprocess (`.S`) and assemble all `.s` files into relocatable `.o` object files
4. [16-bit only] Invoke `ezbl_tools.jar` to edit the project's `.gld` linker script file and updates it with toolchain supplied `.gld` linker script contents for the target device selected in the Project Properties.
5. Pass all `.o` files + `.a` archive libraries (in the project or supplied with the toolchain) + the project's `.gld` or `.ld` linker script (if any) to the toolchain linker.
6. Invoke `ezbl_tools.jar` to extract information from the `.elf` file. This project specific information is injected back into the `.gld/.ld` linker script. Information collected includes:
  - a) Address ranges occupied by Bootloader content in flash or marked as reserved Application space
  - b) Names of all ISR functions implemented in the project



- c) [16-bit only] PSV/EDS page that the .const flash section is located on
- d) Config word names, addresses and values defined in the project

Using this information, plus information collected in earlier steps, the generated/derived information that gets written back to the `.gld/.ld` linker script includes:

- a) Flash geometry occupied or reserved by the project, padded as necessary to start and end on perfect flash erase page sized boundaries.
  - b) Erase-page-size aligned address where Application code will start.
  - c) A backup of all Config word values defined in the project, saved to a new location in flash
  - d) Dispatch multiplexing code stubs to run-time select execution of a Bootloader ISR or an Application ISR whenever any of the ISR functions implemented in the Bootloader are triggered by a hardware interrupt.
  - e) [16-bit only] Branch destination for all IVT entries. For IVT entries corresponding to ISRs that the project has implemented, the branch destination is the corresponding dispatch multiplexing stub. Remaining IVT entries with hardware-implemented interrupts are pointed to Interrupt Goto Table (IGT) entries for the Application that will be generated later when the Application project is built. All reserved/unimplemented IVT entries are pointed to a coalesced, default IGT entry.
7. Reinvoke the linker a second time, passing all original project `.o` + `.a` files, but with the `.gld/.ld` file now containing project-specific data. Additionally, the `BOOTID_HASH0` . . `BOOTID_HASH3` symbol values computed in step 1 are passed to the linker on the command line.
  8. Reinvoke `ezbl_tools.jar` to extract information from the `.elf` file. This (usually final) Bootloader project specific information is checked to ensure the project's flash geometry has not overflowed over a new flash erase page boundary. Additionally, the `.elf` contents are used to generate new `merge.gld/merge.ld` and `merge.S` linker script and source files for the Application project to use in the future.
  9. Executes a tool chain converter utility to extract only the flash contents from the `.elf` file and generates a `.hex` file from it.

At this stage, you have a Bootloader that should work on your target device, assuming the project contained necessary communications and flash programming code in it. It is now necessary to create an Application project (or modify an existing Application) to make it compatible with the Bootloader. This is needed, for example, to avoid flash address overlap between the static Bootloader code and the Application project code.

The automation of EZBL makes Application compatibility changes quite effortless:

1. Add `merge.gld/merge.ld` linker script generated by Bootloader step 8 to the project's **Linker Files**.
2. Add `merge.S` Bootloader image file to the project's **Source Files**.
3. Convert `.elf` file generated when building the Application to a binary `.bl2` file. This `.bl2` file will be sent to the Bootloader rather than the `.hex` file. Scripting this post-build step is accomplished by adding a new `ezbl_app.mk` makefile script to the project (included from the MPLAB generated `Makefile`).

## Bootloader Interrupt Handling

EZBL bootloaders can implement their own interrupt handlers without precluding use of the same hardware vector from future use by Application projects. Typically, one Timer or M CCP/SCCP Timer interrupt is implemented, plus one or more communications peripheral interrupts. These facilitate fast and reliable communications buffering, communications timeout events, de-bricking interval termination for Application launch and optionally permit the Bootloader to continue listening at low priority for external bootload requests while the Application is executing.

Customized Bootloaders can seamlessly add other interrupt handlers, up to 32 total on 16-bit devices or any combination of interrupts on 32-bit devices. In all case, future Applications can receive any hardware-implemented interrupt, regardless of existing use in a Bootloader.

Ordinarily, an Alternate Interrupt Vector Table (AIVT) hardware feature is required to share the same peripheral interrupts with an Application when a Bootloader has clobbered the original Interrupt Vectors. However, not all



PIC/dsPIC products support AIVT hardware, so EZBL avoids potential problems by implementing other interrupt sharing options described in this section.

Due to quite different hardware and toolchain implementations, interrupt handling capabilities are identical between 16-bit and 32-bit EZBL implementation, but their internal implementation varies substantially. Therefore, this help section is subdivided into dedicated 16-bit and 32-bit discussion.

For 16-bit implementations, see: [16-bit, PIC24/dsPIC Interrupt Handling](#)

For 32-bit implementations, see: [32-bit, PIC32MM Interrupt Handling](#)

## 16-bit, PIC24/dsPIC Interrupt Handling

### *Dual Partition Interrupts*

For Dual Partition bootloader types, all Dual Partition capable hardware devices implement a dedicated IVT on both the Active and Inactive Partitions. The ping-ponging nature of programming new firmware on an alternate partition allows Bootloader code and Application code to come together in the same project and on the same flash pages, exercising hardware resources without any isolation boundary between code functionality. Therefore, for Dual Partition bootloaders, EZBL does nothing special to handle interrupts separately from the Application.

When implementing a Dual Partition bootloader, all sections in this chapter should be ignored with development approached as a monolithic project with full access to all hardware resources.

### *Standard (Single Partition) Interrupts*

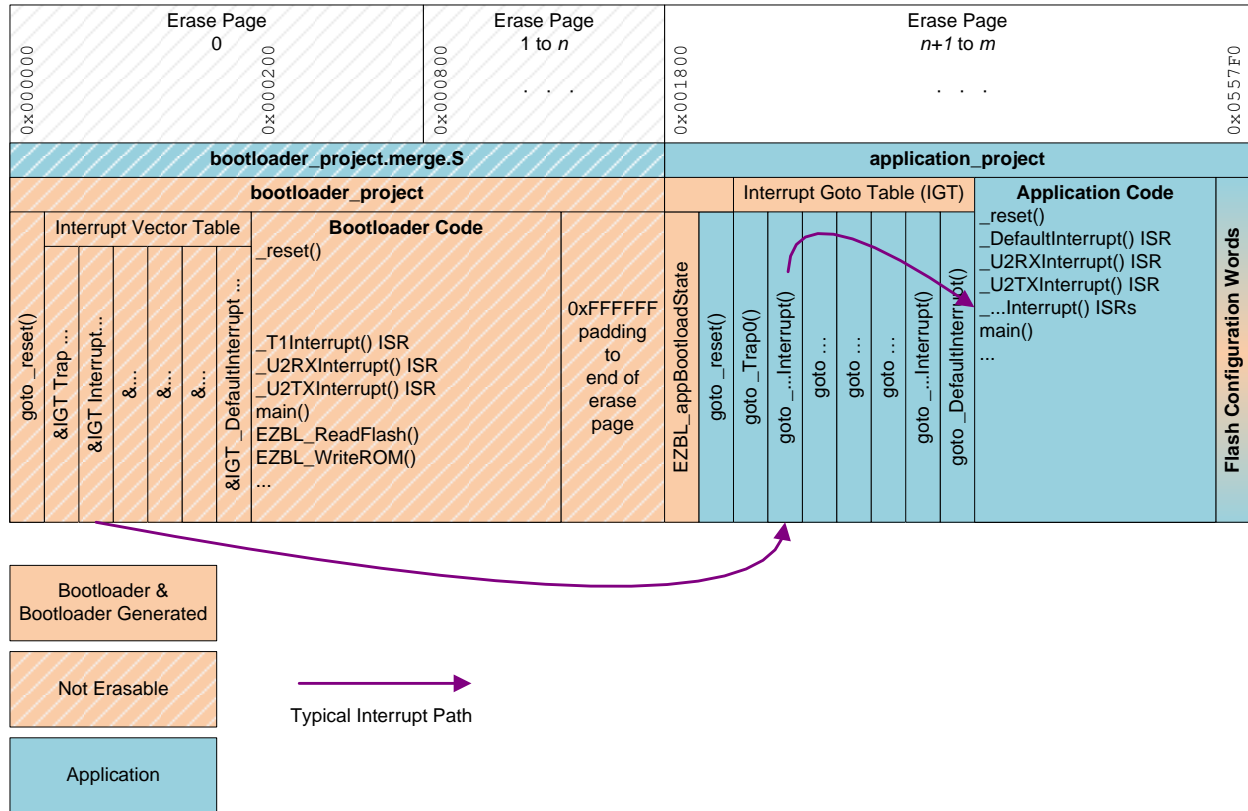
For standard, single partition, Bootloader types, EZBL implements a purely software approach to sharing interrupts that is more flexible and typically smaller than AIVT hardware based solutions. The software approach consists of three different options, which apply on a per-interrupt conditional basis:

- [Default IGT Remapping](#): any interrupt not implemented in the Bootloader project always propagates to an Application defined ISR or the Application `_DefaultInterrupt()` handler via an erasable/reprogrammable Interrupt Goto Table ("IGT").
- [ISR Reuse](#): hardware invokes the Bootloader ISR and Application code relies on higher-level Bootloader APIs without implementing a separate Application ISR to handle the same interrupt.
- [Interrupt Forwarding](#): both Bootloader and Application ISRs are implemented for the same interrupt, and a run-time flag decides which handler is executed each time the interrupt is triggered.

### *Default IGT Remapping*

In the default Interrupt Goto Table Remapping case, the hardware examines the interrupt's IVT entry, calls the address of a corresponding IGT entry and then an Application defined GOTO (or BRA) opcode is executed to branch to the Application's ISR handler. This interrupt remapping scheme is implemented for all hardware interrupts which do not get processed by a Bootloader ISR handler – do nothing and this is what you get.

This remapping scheme is represented by the following flash memory map diagram:



This software methodology is typical of many other bootloaders and EZBL implements it in a similar way. Key benefits/costs associated with this mode are:

- All Application ISR handlers can be placed anywhere in Application flash space and are automatically managed by the linker
- IGT is erasable without erasing the Bootloader's Reset Vector, which would otherwise generate an opportunity to brick the Bootloader
- ISR entrance latency is increased by exactly 1 branch delay (2  $T_{CY}$  on PIC24F/PIC24H/dsPIC33F/dsPIC33C PRAM devices, or 4  $T_{CY}$  on PIC24E/dsPIC33E/dsPIC33C flash devices). There is zero interrupt return penalty.

Unlike traditional bootloaders, EZBL uniquely assists by generating the IGT automatically when the EZBL Bootloader project is compiled and linked. This allows the IGT to support your Bootloader's target processor without any manual edits to a .gld linker script or .s assembly file. Additionally, this allows the IGT to automatically relocate itself between flash erase page boundaries, depending on the final, linked flash geometry of your Bootloader project code.

An additional EZBL benefit is that the IGT is auto-generated to contain only useful entries. The hardware IVT occupies a fixed block of flash, typically 0x000200 program addresses or 768 bytes of flash on most 16-bit products. This supports up to 253 different hardware interrupt vectors, but there is currently no silicon product that has enough peripherals on it to actually fill all 253 vector slots. This leaves unused vector slots as reserved flash locations, unsuitable for code and which will never become used in a future software update.

EZBL takes these unused entries into account and coalesces all of their IVT targets into a single `_DefaultInterrupt()` IGT entry. This typically shrinks the IGT by about half of the size it would require if every hardware IVT entry received a software IGT entry.

Although the IGT still requires a moderate amount of dedicated flash space to store on some devices, it remains much smaller than modern AIVT hardware requirements. On newer 16-bit product families, the AIVT hardware is relocatable, but requires the entire flash page of 0x000400 or 0x000800 program addresses (1536 or 3072 bytes) be reserved for it. This means the worst case IGT size for all 253 possible interrupt sources is always smaller than a modern AIVT -- usually by a 2:1 or 4:1 ratio.

### ISR Reuse

In the ISR Reuse case, the Bootloader implements an ISR, clobbering an IVT entry, and the Application gets by without implementing another ISR. Sharing works because the Bootloader exports all of its globally scoped functions and variables so that the Application can call or read/write them directly, as if they were part of the Application. In the case of UART and other serial communications mediums, a software FIFO buffering library is built around the hardware peripheral so there are convenient higher level APIs that Apps and Bootloaders alike can use without implementing separate ISRs. To use this method of Bootloader + Application Interrupt sharing:

1. `#include "ezbl.h"`
2. Add `ezbl_lib.a` to the project under **Libraries**
3. Directly call the `NOW_64()`, `EZBL_FIFORead()`, `EZBL_FIFOWrite()` or other EZBL APIs that depend on Bootloader-implemented ISRs.

The `EZBL_FIFO*()` serial communications functions allow you to read/write data and manage FIFOs in a typically non-blocking fashion (although to avoid data loss, if the TX FIFO is full and you issue another write command, the API will block until a timeout is reached or the TX ISR frees buffer space to allow complete queuing of the write data).

On receive, no direction notification occurs that data is piling up, so reading requires periodically checking the applicable `EZBL_FIFO->dataCount` variable (ex: `UART2_RxFifo.dataCount`) to see if data is waiting in the RX FIFO for your Application. Alternatively, if you know data is going to arrive, even if it is not there yet, you can call an `EZBL_FIFORead()` / `EZBL_FIFOPeek()` function ahead of time and it will block until the specified number of bytes of data are finished being received (or a timeout is reached).

The `NOW_64()` API is timing rather than communications related, but is shown as an example since it may depend on the Timer or MCCP/SCCP peripheral interrupt to implement a 64-bit software timer with single-cycle precision from only one 16-bit hardware timer resource. The Bootloader-implemented timer ISR carries out of bit 15 of the hardware timer and into the extended software portion.

### I<sup>2</sup>C Specific

Because the I<sup>2</sup>C Bootloader operates as an I<sup>2</sup>C Slave, the APIs will also behave as a slave node. Anything you place in an I<sup>2</sup>C TX FIFO will not go onto the I<sup>2</sup>C bus until a remote I<sup>2</sup>C Master issues a read request addressing the slave, and only then will the Bootloader's I<sup>2</sup>C ISR be executed. The I<sup>2</sup>C Master must conform to the framing protocol documented in the [help\EZBL Communications Protocol.pdf](#) file in order to be able to receive the same logical data that the PIC application placed in the TX FIFO. For example:

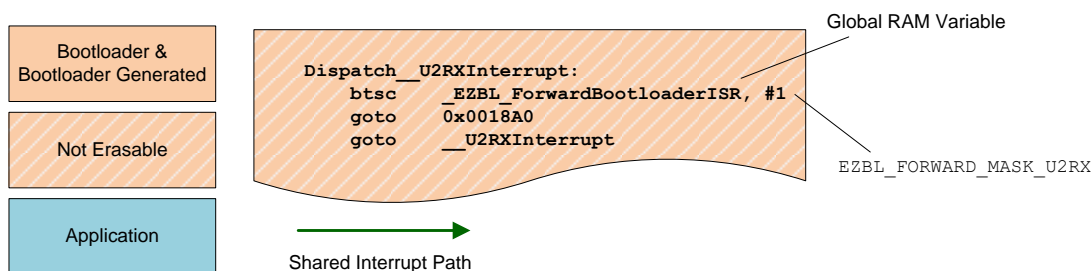
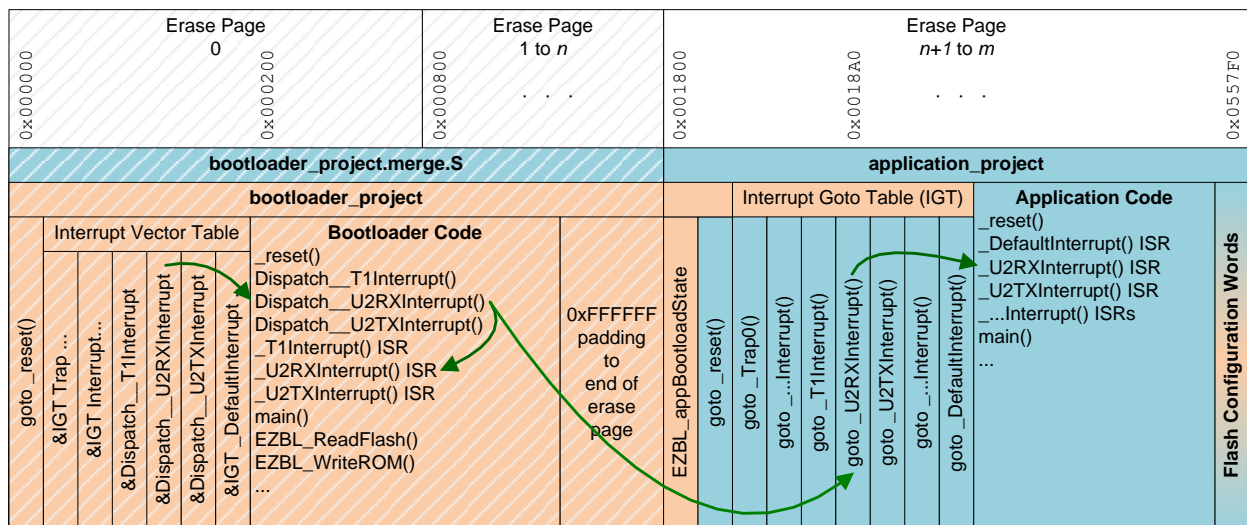
- Application writes message to `I2C1_TxFifo` by calling `EZBL_FIFOWrite()`
- I<sup>2</sup>C master issues an I<sup>2</sup>C read request to the PIC at slave address 0x60
- Bootloader `_SI2C1Interrupt()` ISR prefixes the FIFOed outbound TX data with a 1 byte header specifying how many bytes are in the FIFO, ready for transmission (`I2C1_TxFifo.dataCount`)
- I<sup>2</sup>C master strips the 1-byte framing header off and continues to clock SCL to get some of the FIFO'd bytes
- I<sup>2</sup>C master passes transferred data (without the framing header) to the reading application or software

### Interrupt Forwarding

In the Interrupt Forwarding case, the Application gains full control of the hardware interrupt and implements its own ISR to handle it. The Bootloader's ISR for the interrupt stops being called and no background Bootloader processing occurs against it. Applications previously using an ISR Reuse sharing mode can switch to this Interrupt Forwarding mode without recompiling or reprogramming their Bootloader.

This sharing mode is logically equivalent to having hardware swap between a dedicated IVT owned by the Bootloader and a dedicated AIVT owned by the Application. However, unlike a hardware IVT → AIVT swap, each interrupt vector can be individually routed to the Bootloader or Application as a run-time choice without changing routing to all other interrupts simultaneously. For example, it is possible to simultaneously have a Timer or MCCP/SCCP interrupt handled by a Bootloader ISR, while all other interrupts get handled by Application ISRs. This allows Bootloader services like the 64-bit NOW time keeping and task scheduling APIs to maintain cycle accurate counts and perform scheduled callback executions while your Application takes complete control of the communications interface.

Graphically, this Interrupt Forwarding methodology implements:



When a possible Bootloader interrupt is generated, the IVT will branch execution to a three-instruction bit-test-skip-if-clear, goto, goto dispatching stub to properly call the Application's IGT entry or the Bootloader's ISR handler for the given interrupt.

To enable Interrupt Forwarding to the Application:

1. `#include "ezbl.h"`
2. Implement the Application's ISR within the Application project, ex: `_U2RXInterrupt()`

Then, at run-time:

EZBL Help

Copyright © 2018 Microchip Technology Inc.

Updated 30 August 2018

3. Disable the hardware interrupt, ex: `IEC1bits.U2RXIE = 0;`
4. Set the appropriate forwarding bit in the EZBL generated EZBL\_ForwardBootloaderISR global variable, ex: `EZBL_ForwardIntToApp(U2RX);`
5. Reinitialize the peripheral, as appropriate, for the Application
6. Reenable the hardware interrupt, ex: `IEC1bits.U2RXIE = 1;`

This run-time sequence can be implemented in the Bootloader project, the Application project or mixed between both. If it is not necessary to reinitialize the peripheral, steps 3, 5 and 6 can be omitted. In such a case, step 4 must be executed only after the Application is prepared to immediately start receiving the interrupt.

If the Application wishes to restore processing by the Bootloader's interrupt handler:

1. Call the complementary macro `EZBL_ForwardIntToBoot()`

EZBL also contains `EZBL_GetForwardInt()` and `EZBL_GetWeakForwardInt()` macros to read the current forwarding state of a particular interrupt if needed.

To avoid a chicken-and-egg problem, the `EZBL_ForwardIntToApp()`, `EZBL_ForwardIntToBoot()`, and `EZBL_GetWeakForwardInt()` macros treat the symbol parameter reference as 'weak'. This allows the macros to devolve into a series of null operations (effectively NOPs) if the corresponding interrupt does not actually have a Bootloader defined ISR for it. This also means that no compile or linker error is emitted if a typo is made and an invalid interrupt name is passed to the macro. Therefore, double check these interrupt name parameters if run-time debugging reveals that the wrong ISR is being used to handle a given interrupt.

Correct names for a given interrupt are derived by truncating the leading underscore and trailing 'Interrupt' characters off the XC16 ISR name. For example, various interrupt names which EZBL Bootloader projects may implement include (not comprehensive):

- T1 --> `T1Interrupt()` for Timer 1 (NOW time keeping/scheduling)
- CCT1 --> `CCT1Interrupt()` for MCCP1/SCCP1 timer (alternate option for NOW time keeping/scheduling)
- U2RX --> `U2RXInterrupt()` for UART2 Receive
- U2TX --> `U2TXInterrupt()` for UART2 Transmit
- SI2C1 --> `SI2C1Interrupt()` for Slave I2C1
- USB1 --> `USB1Interrupt()` for USB1

Adding/removing ISR handlers to a Bootloader project is fully automated using EZBL. If you wish to use a new interrupt in your Bootloader, simply add an ISR handler for it to your project. Rebuilding the project will trigger generation of a `Dispatch__xxxyInterrupt` multiplexing stub routine in flash, assign a control bit for the interrupt in the `EZBL_ForwardBootloaderISR` global variable and add `EZBL_FORWARD_[MASK/POS/IRQ]_xxxy` symbols in the Bootloader's .gld linker script. Deleting an ISR and rebuilding will have the reverse effect of deleting any existing multiplexing code and symbols.

The reset state of `EZBL_ForwardBootloaderISR` is `0x00000000`, so all interrupts are, by default, handled by Bootloader implemented ISRs (or Application implement ISRs when no applicable Bootloader ISR exists due to [Default IGT Remapping](#)). If you wish to fully disable any background EZBL Bootloader processing before launching your Application (or anytime thereafter), use the `EZBL_ForwardAllIntToApp()` macro.

### Interrupt Latency

Run-time performance cost of the EZBL Interrupt Forwarding feature depends on the target architecture, flash geometry and resolved ISR target as follows:

Device Architecture	Bootloader ISR Exists	Flash Geometry	Latency to Bootloader	Latency to Application
---------------------	-----------------------	----------------	-----------------------	------------------------

			ISR	ISR
PIC24F/PIC24H/dsPIC33F or dsPIC33C (PRAM execution)	Yes	< 128KB	+4 TCY	+5 TCY
	Yes	≥ 128KB	+5 TCY	+5 TCY
	No			+2 TCY
PIC24E/dsPIC33E/dsPIC33C (flash execution)	Yes	< 128KB	+6 TCY	+9 TCY
	Yes	≥ 128KB	+7 TCY	+9 TCY
	No			+4 TCY

Dispatching stubs are generated with BRANCH-always instructions instead of GOTO instructions when the total flash geometry is small enough. These require one less FNOP instruction word to be executed when reaching Bootloader ISRs.

All figures are indicated as additional instruction cycles of ISR entrance latency as compared to a product with no Bootloader and direct use of IVT hardware.

When no Bootloader ISR exists for a given interrupt, the minimum cases of +2 or +4 instruction cycles of latency are added as a result of the IGT branch delay before reaching the Application's ISR handler.

When a Bootloader ISR does exist, a dispatching stub is added, resulting in one bit test instruction and one branch in advance of reaching the IGT for interrupts handled in the Application. When the bit test instruction skips the App IGT branch and execution heads to the Bootloader ISR, latency is added from the bit test, a branch from the dispatching stub to the Bootloader ISR, and one or two cycles extra as the App IGT branch is skipped (which is a BRA instruction on smaller devices and larger 2 instruction word GOTO opcode on devices with more flash).

### *Bootloader APIs when Interrupts are Forwarded or Disabled*

Most of the APIs in `ezbl_lib.a` are implemented to fall back to a limited, on-demand polling method when a call is made which ordinarily requires the use of a Bootloader interrupt but which isn't being processed by the Bootloader's ISR. For example, calling `EZBL_printf()` typically places data in `UART2_TxFifo` and returns immediately. However, when the UART2 TX Interrupt is forwarded to the Application's `_U2Interrupt()` ISR, the Bootloader's `_U2TXInterrupt()` ISR will not asynchronously transfer data from `UART2_TxFifo` to the `U2TXREG` SFR. `UART2_TxFifo` will fill up with continued writes to the FIFO, eventually causing `EZBL_printf()` to become a blocking call. Rather than deadlocking the system for an event that won't happen, the internal blocking loop will poll the Bootloader's `_U2TXInterrupt()` ISR to force data transfer out of the FIFO. As soon as all bytes generated by `EZBL_printf()` are written to the software FIFO (not the UART TX hardware), `EZBL_printf()` returns.

This on-demand polling behavior is primarily implemented for debugging purposes as it allows serial console writes to complete from within a trap exception handler or execution context in which the hardware interrupt is disabled or masked by a higher IPL. Such forced data transfer allows data corruption in the form of interleaved data sent from the UART and loss of data within the FIFO structure due to concurrent/reentrant data writes from two different IPLs simultaneously. Relying on this fall back on-demand polling is therefore not recommended. However, it remains an effective means of exiting API deadlock scenarios and is safe for some use cases.

### **32-bit, PIC32MM Interrupt Handling**

The MIPS microAptiv core on the PIC32MM series of devices implement a few fixed address, non-maskable exception vectors, plus an Interrupt Vector Table (IVT) that can be located on any 4KB address boundary in either RAM or flash. Unlike 16-bit devices, the IVT is treated by the CPU as executable code and interrupts cause the CPU to branch into the IVT. Code begins executing directly in the IVT, which typically contains a jump instruction to

reach the real Interrupt Service Routine's (ISR) start address, 2 cycles later. This architectural difference allows for quite flexible ISR placement and arbitrary use of vectors between Bootloader or Application projects but comes at the cost of considerable flash overhead if implementing two discrete vector tables between projects. Each hardware interrupt typically requires 8 bytes of IVT space to implement the jump operation.

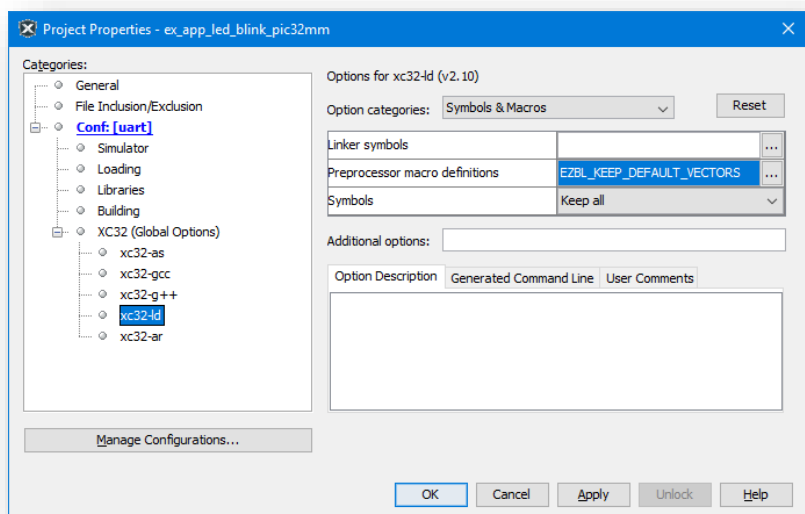
To avoid the overhead of two complete IVTs and enable per-vector remapping functionality roughly equivalent to 16-bit EZBL implementations, EZBL implements two compacted IVTs, a static one for the Bootloader project and one erasable/reprogrammable version in the Application project. Additionally, Bootloader projects implement zero or more global RAM function pointers to allow run-time remapping of vectors on a per-vector basis between projects.

### Compacted IVT

Typically, XC32 projects contain a contiguous region of flash defining the vectoring behavior for all hardware-implemented interrupts, regardless of if the project implements an ISR for a given vector or not. For both Bootloader and Application projects, EZBL does away with the complete IVT and only implements vectoring instructions for the IVT entries that have a non-default software ISR associated with it. This frees up the unused vector locations as flash available for Bootloader or Application code. Typically, the linker's best-fit allocator will be able to place small code objects in unused vector locations, even if the free space for it is fragmented.

For a typical UART Bootloader requiring two interrupts, this means the Bootloader's IVT shrinks to a negligible 16 bytes of flash, total, instead of 384 bytes (PIC32MM0064GPL036 Family) or 816 bytes (PIC32MM0256GPM064 Family) for a non-compacted IVT using 8 byte vector spacing. Application projects also benefit as their IVT is also compacted to 8 bytes/used vector.

If the default vectors are needed, they can be restored on a per-project basis by defining the `EZBL_KEEP_DEFAULT_VECTORS` xc32-ld preprocessor macro within the Project's properties:



### Individual Vector Remapping

To enable background Bootloader interrupts, such as UART RX FIFOing routines to optionally execute while an Application project is executing, EZBL automatically generates a 4 byte function pointer in RAM for each ISR implemented within the Bootloader project. Both Bootloader and Application IVTs are then populated with a RAM load (ld) instruction, followed by a jump opcode to reach an arbitrary ISR address in the Bootloader or Application project flash. This differs from the default XC32 IVT generation which executes a jump instruction whose target address is statically encoded in the flash jump instruction opcode.



Because shared vectors encode the ISR target address in a RAM variable, each ISR implemented in the Bootloader adds 4 bytes of globally allocated RAM (regardless of which project is currently active). However, vectors the Bootloader project does not implement will contribute no overhead to the Application, beyond standard XC32 vectoring resource costs. These Application-only vectors will always use the compacted IVT generated by the Application.

For fixed address MIPS hardware exception vectors, EZBL places a vector remapping stub routine at the target address that reads the Coprocessor 0, EBase register, then jumps to a fixed offset from this EBase address. As a result, both Bootloader and Application projects have independent access to these exception vectors while they are currently active, but cannot control them on an individual basis. In other words, the Bootloader will receive the fixed address exceptions while it is executing out of reset, but after the Application is launched, only the Application will be able to receive these exceptions as an all or nothing selection.

A list of all fixed address and computable vectors, their original hardware target address and their corresponding remapped targets in Bootloader/Application projects follows.

Exception	StatusBEV, StatusEXL, CauseIV	Hardware Target	EZBL Target	Effective Target for 0x2000 Bootloader Size	
				Bootloader active	Application active
Reset and NMI	x, x, x	0xBFC00001	EBase + 0x801	0x9D000801	0x9D002801
General Exception & Single Vectoring Interrupt	0, 0, 0	EBase + 0x181	EBase + 0x181	0x9D000181	0x9D002181
Multi-vectoring (IVT) Interrupt	0, 0, 1	EBase + 0x201 + (vector_spacing * IRQ)	EBase + 0x201 + (vector_spacing * IRQ)	0x9D000201 + (vector_spacing * IRQ)	0x9D002201 + (vector_spacing * IRQ)
Bootstrap General Exception	1, 0, 0	0xBFC00381	EBase + 0x141	0x9D000141	0x9D002141
Bootstrap Special Interrupt	1, 0, 1	0xBFC00401	EBase + 0x201	0x9D000201	0x9D002201
EJTAG Debug (EJTAGProbEn==0)	x, x, x	0xBFC00481	EBase + 0x161	0x9D000161	0x9D002161

All address LSbits are shown == '1' as required by hardware when branching to indicate continued instruction decoding using the microMIPS ISA (not MIPS32, which isn't supported). However, when declaring data for a vector target, the data must always start at the even-aligned address (LSbit == '0').

### Interrupt Latency

Run-time performance cost when implementing interrupts in a Bootloader project are negligible:

Device Architecture	Bootloader ISR Exists	Flash Geometry	Latency to Bootloader ISR	Latency to Application ISR
PIC32MM	Yes		+1 TCY	+1 TCY
	No			No change

The one additional cycle of latency occurs due to a need to load the currently selected target ISR address from a global RAM variable in order to branch to it. The branch itself is not included since the branch target is the ISR function itself and all native IVT entries already require one branch to reach the ISR.

### Bootloader Initialization

At reset or power up, Bootloader projects must call the `EZBL_ForwardAllIRQToBoot()` macro prior to globally enabling interrupts. This macro sets all RAM function pointers to point to matching ISRs implemented in the Bootloader project. Prior to this call, the RAM pointer will be uninitialized and contain an unknown target address.

To call this macro, as well as other macros and functions related to interrupts, some source file and project level preconditions must be met. The overall procedure to get Bootloader interrupts working is to:

1. `#include <xc.h>`
2. `#include "ezbl_integration/ezbl.h"`
3. Add `ezbl_integration/ezbl_lib32mm.a` precompiled archive library to the project (under **Libraries** in the MPLAB X IDE Projects tree view window).
4. Implement any desired ISRs as you would in a normal XC32 project. ISR functions must have the `__attribute__((interrupt(IPLx[AUTO/SOFT/SRS]), vector(IRQ)))` decorations. The vector attribute tells the compiler to generate an IVT entry to reach the ISR and is also used by EZBL to generate an `EZBL_Vector[IRQ] ISRPtr` global variable for run-time remapping it within the project's .ld linker script.
5. Call `EZBL_ForwardAllIRQToBoot();`
6. Call `__builtin_enable_interrupts();`
7. Assign the interrupt's IPCx bits to a non-zero value to set the interrupt priority. The `EZBL_WrIntPri()` function may be used to accomplish this if you wish to abstract your code by IRQ number rather than referencing a somewhat arbitrary IPCx SFR by name.
8. Set the interrupt enable bit for the interrupt in the applicable IECxSET register. The `EZBL_WrIntEn()` or `EZBLSetIntEn()` functions may be used to do this.

### Application Initialization and Forwarding Control

Prior to Bootloader execution handoff to the Application, or at Application start up, either project may optionally call the `EZBL_ForwardAllIRQToApp()` macro. This will configure the RAM function pointers to point to matching ISRs implemented in the Application project instead of the Bootloader. If the Application does not implement an ISR handler for an interrupt that the Bootloader does, then the Bootloader will remain the default handler during Application execution. Therefore, if the end goal is to effectively make the Bootloader disappear during Application execution, be sure to clear the IECx interrupt enable bits for all applicable interrupts.

If an interrupt remains forwarded to the Bootloader, the Application's applicable ISR will never be called. Similarly, if an interrupt is forwarded to the Application, hardware will stop calling the Bootloader's ISR and instead branch to the Application's ISR as an exclusive choice (execution is not chained in series).

Individualized vector control is accomplished with the `EZBL_ForwardIRQ()` library function or `EZBL_ForwardIRQToApp()` / `EZBL_ForwardIRQToBoot()` macros. As an input parameter, these require the IRQ number that will be redirected. Human readable names for these are defined in the processor header, `#included via <xc.h>`. As an example, execute the following statements to transfer ISR vectoring for the UART2 RX and TX Interrupts to the Application project:

```

EZBL_ForwardIRQToApp ( _UART2_RX_VECTOR ) ;
EZBL_ForwardIRQToApp ( _UART2_TX_VECTOR ) ;

```

Any of the interrupt control functions/macros may be called at any time by either Bootloader or Application projects if/when it wants to take over or relinquish control of the hardware interrupt source(s). Internally, these APIs lookup the proper ISR function address for the targeted project and write this address to a global pointer statically allocated in RAM within the Bootloader project, but visible to both projects.

At Application startup, the XC32 Compiler Run Time (CRT) will globally disable interrupts, so the Application must call `__builtin_enable_interrupts()` to begin receiving interrupts. This requirement to enable global interrupts exists even if the Bootloader handed off execution to the Application while interrupts were already enabled. However, the global ISR pointer variables will retain their prior (Bootloader specified) state/contents when the Application starts.

### ISR Reuse

Prior discussion revolved around transferring exclusive ownership of a hardware interrupt vector between projects at run-time. An alternate option is to maintain a Bootloader ISR (or ISRs) in control of the applicable hardware and have the Application simply call existing Bootloader-defined high level APIs to reuse whatever hardware and software is attached to a particular interrupt. For example, rather than implement a second copy of software UART FIFOing routines and UART RX + TX ISRs in the Application, the Application can omit both ISRs, leave the vectors forwarded to the Bootloader ISRs, and call higher level APIs, like `EZBL_FIFOWriteStr(&UART2_TxFifo, "Hello World!\n")`;

In the ISR Reuse case, sharing is possible because the Bootloader exports all its globally scoped (non-static) functions and variables so that the Application can call or read/write them directly, as if they were part of the Application. In the case of UART and other serial communications mediums, a software FIFO buffering library is built around the hardware peripheral so there are several convenient higher level APIs that Apps and Bootloaders alike can use without implementing separate ISRs. To use this method of Bootloader + Application Interrupt sharing:

1. `#include "ezbl.h"`
2. Add `ezbl_lib32mm.a` to the project under **Libraries**
3. Add appropriate prototypes/extern declarations for the C functions and objects that you will be using. This can often be done by `#including` original headers used in the Bootloader project, or for EZBL APIs, will already be completed in step 1.
4. Directly call the `EZBL_FIFORead()`, `EZBL_FIFOWrite()`, etc. EZBL APIs that depend on Bootloader-implemented ISRs, or any other non-static function in your Bootloader project. Execution will branch to the code within the Bootloader project, then return to the Application, just as calling a function in a separate source file would.

Obviously, reusing Bootloader code and ISRs may require some planning while developing their original Bootloader implementation to make them generic enough for reuse later. However, a valuable benefit is that overall Application code size can be smaller. An additional benefit is that run-time Bootloader state doesn't get lost when the Application starts up and background Bootloader services can remain dormant but still available for immediate use, such as listening for externally initiated firmware update requests while the Application is idle. The Application could even be used to pass higher level parameters to the Bootloader which the Bootloader itself can't obtain, such as communications parameters or notification of new firmware availability on a remote server.

To achieve a code size reduction in the Application, the Application project should not contain duplicated source code for any of the APIs that exist in the Bootloader and are suitable for reuse. All Bootloader functions and variables are exported to the Application as 'weak' symbols, so having a strongly defined duplicate of an item in the Application project will override the Bootloader's copy (when referenced in the Application) and result in two instances of the same code in flash.

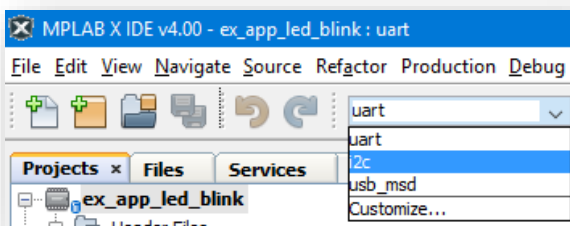
If a Bootloader implemented API requires changes for use by the Application, the Application won't be able to reuse the same code anymore, but since Bootloader symbols are exported as 'weak' in the Application project, the same mechanism of reimplementing a same named function/variable can be used to permit the code change without a global scope name collision. In this scenario, the Bootloader will continue to use its copy and the Application will use its new, amended copy (effectively hiding the one in the Bootloader).

## *ex\_app\_led\_blink MPLAB® X Project*

This project demonstrates a trivial Application project that will be reprogrammed using your EZBL Bootloader. It is a companion test project intended for use with:

- [ex\\_boot\\_uart](#) - UART Bootloader
- [ex\\_boot\\_i2c](#) - I<sup>2</sup>C Slave Bootloader
- [ex\\_boot\\_usb\\_msd](#) - USB Host - Mass Storage Device (MSD) class "thumb drive" Bootloaders

Any of these Bootloader pairings must be selected and compiled separately before use. I.e. you cannot compile *ex\_app\_led\_blink* against the UART bootloader and then upload it over I<sup>2</sup>C to a Bootloader built from the *ex\_boot\_i2c* project (technically programming may be successful, but the Application code likely won't execute correctly because any Bootloader functions it attempts to call will be located at incorrect Bootloader addresses). The Build Configuration option selects which Bootloader project *ex\_app\_led\_blink* gets built for by including/excluding a different set of `[boot_project].merge.S/[boot_project].merge.gld` files during linking:



When *ex\_app\_led\_blink* successfully executes, it will toggle an LED/GPIO pin at 1 Hz (500ms per toggle event) to indicate success.

This project can be compiled, programmed into a device via a classic ICSP method, and debugged using ordinary debuggers without separately programming the Bootloader into the device beforehand. When programmed via ICSP, the target device receives both your Bootloader project's binary image and the code implemented by the *ex\_app\_led\_blink* Application itself.

When uploaded to a preexisting Bootloader, the bootloader image encoded in the `ex_boot_[build_configuration].merge.S` file must match the Bootloader already in flash. If there is a mismatch, the Application generally fails to execute as intended. Therefore, it is critical that once a product is released to manufacturing, the `[*].merge.S` file be archived and not edited. Similarly, while still editable in some cases, the `[*].merge.gld` linker script file should be archived.

All code and referenced library code used in this project is meant to be replaceable by the Bootloader. Even shared functions or variables that are inherited for free from the Bootloader image can be replaced for purposes of Application use (the Bootloader will still use its original copy).

### When Built

1. Outputs a combined Bootloader + LED Blink Application .hex file, `ex_app_led_blink.[production/debug].hex`, for use with traditional ICSP based programmers
2. Outputs the combined Bootloader + LED Blink Application .elf file re-encoded into a compact binary .bl2 file, `ex_app_led_blink.[production/debug].bl2`. This .bl2 file is normally what you would distribute when you release a new Application firmware update. .bl2 files are similar to a .hex file, but EZBL creates them from the .elf file to obtain BOOTID\_HASH metadata and have it placed in the .bl2 file header.
3. [UART/I<sup>2</sup>C only] Attempts to transfer the generated .bl2 file to the target device and permit immediate run-time execution testing.

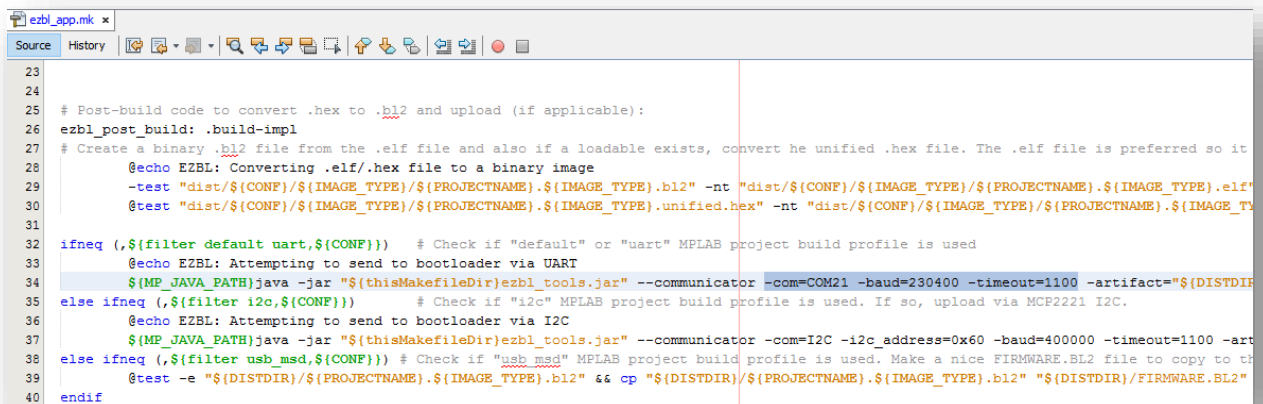
4. [USB MSD only] Copies the `ex_app_led_blink.[production/debug].bl2` file to `FIRMWARE.BL2`. The USB MSD Bootloader expects 8.3 formatted filenames with `FIRMWARE.BL2` located in the root directory of the mass storage media. This copy is created to allow easy copy/paste or drag-and-drop transfer to the media from any PC without requiring a manual file renaming step.

## Supports

1. Out-of-box demo ability on:
  - Explorer 16/32 (or Explorer 16) development boards ([DM240001-3](#) or [DM240001-2](#) + PIM)
  - MPLAB Starter Kit for Digital Power ("Digital Power Starter Kit", [DM330017-2](#)) + MCP2221 Breakout Module ([ADM00559](#))
  - PIC24FJ1024GB610, PIC24FJ256GB410, dsPIC33EP512MU810, dsPIC33CH128MP508 and many other Explorer 16/32 PIMs. See [help\EZBL Release Notes.htm](#) for a list of tested targets with already existing hardware initialization files.
2. Out-of-box demo ability on Microsoft Windows® OS (only) for UART and I<sup>2</sup>C bootloaders as the COM hardware interface code is OS and platform dependent (see [ezbl\\_comm](#) for information on compiling it to support other platforms). USB Host MSD operation does not require a PC, so Mac and Linux platforms are suitable.

## Notes

1. **Important Files** contains a customized makefile, `ezbl_app.mk`. Additionally, `Makefile` has been trivially modified at the bottom to include `ezbl_integration/ezbl_app.mk`. The added make script alters the project build behavior by executing a pre-build step to increment an `APPID_VER` version build number. On post-build, a step converts the just-built `.elf` file to a `.bl2` file and then attempts to transfer the Application to the target Bootloader.
  - a) The `ezbl_post_build`: recipe in `ezbl_app.mk` does the `.elf` to `.bl2` conversion. The additional attempt to upload the generated `.bl2` file is done using this recipe, but only for 'i2c' and 'uart' Build Configurations. Uploading is attempted solely for quicker development and debugging purposes within the IDE. Uploading is always possible outside the IDE and on systems which do not have MPLAB X IDE installed.
  - b) Using the automatic upload feature against a UART Bootloader normally requires the below communications parameters to be modified to match your development machine's COM port.



```

23
24
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. The .elf file is preferred so it
28 # can be loaded directly into the target.
29 @echo EZBL: Converting .elf/.hex file to a binary image
30 -test "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" -nt "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).elf"
31 @test "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).unified.hex" -nt "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).unified.hex"
32
33 ifneq ($(filter default uart,$(CONF))) # Check if "default" or "uart" MPLAB project build profile is used
34 @echo EZBL: Attempting to send to bootloader via UART
35 $(MP_JAVA_PATH)java -jar "$(thisMakefileDir)ezbl_tools.jar" --communicator -com=COM21 -baud=230400 -timeout=1100 -artifact="$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2"
36 else ifneq ($(filter i2c,$(CONF))) # Check if "i2c" MPLAB project build profile is used. If so, upload via MCP2221 I2C.
37 @echo EZBL: Attempting to send to bootloader via I2C
38 $(MP_JAVA_PATH)java -jar "$(thisMakefileDir)ezbl_tools.jar" --communicator -com=I2C -i2c_address=0x60 -baud=400000 -timeout=1100 -artifact="$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2"
39 else ifneq ($(filter usb_msd,$(CONF))) # Check if "usb_msd" MPLAB project build profile is used. Make a nice FIRMWARE.BL2 file to copy to the target.
40 @test -e "$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" && cp "$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" "$(DISTDIR)/FIRMWARE.BL2"
41 endif

```

If developing on a Linux or Mac OS platform, automatic UART/I<sup>2</sup>C upload will not work. Therefore, the applicable `ezbl_app.mk` lines should be commented out (lines 33 to 37 in the above example) by placing a '#' character at the very beginning of the lines.

2. **Linker Files** contains a Bootloader generated linker script, `ex_boot_[i2c/uart/usb_msd].merge.gld`. This linker script allows the Application to build without attempting to overlap any Bootloader owned flash addresses. The linker script is also used to generate and hookup an Interrupt Goto Table to allow any Application implemented ISR to be allocated however/wherever the linker chooses.
  - a) If you need to make your own manual changes to the `.gld` linker script, you can do so. However, be aware that all Bootloader related output section mappings must be left intact for correct run time operation and interaction between the Bootloader and Application projects. This means that all `EZBL_ROM_AT_*`, `EZBL_RAM_AT_*`, `EZBL_AppErasable`, `.igt`, or other explicitly mapped sections starting with an 'EZBL' prefix should not be removed or modified. These sections have absolute addresses assigned to them, so will tolerate unrelated changes around them.

The example Bootloader projects also automatically copy their latest `ex_boot_[i2c/uart/usb_msd].merge.gld` file generated out to example Application projects like `ex_app_led_blink`. This behavior will therefore overwrite any manual changes you make to the `ex_boot_[i2c/uart/usb_msd].merge.gld` linker script if you rebuild your Bootloader project. This generally should not pose a problem (since you generally will not recompile your Bootloader after you have gone to production); however, this could pose a hazard during initial development. Be sure to retain backups of any manual changes done to your Application's copy of the Bootloader generated `.merge.gld` output. In the event your manual changes are overwritten, MPLAB X IDE's "Diff To..." code editor feature may be used to graphically merge changes in a backup linker script with a freshly generated `.merge.gld` linker script.

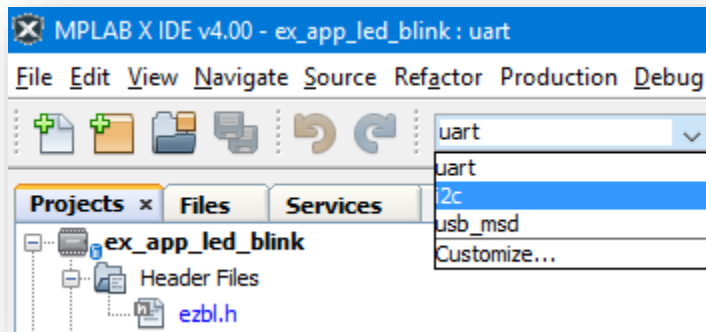
3. **Libraries** contains the `ezbl_lib.a` precompiled archive library. This file is not required, but since there are several potentially useful functions in `ezbl_lib` for Application projects besides Bootloaders, inclusion of this library in the project allows convenient access to any of the EZBL APIs. When an `ezbl_lib` API is called that was already called in the Bootloader project, the linker will preferentially use the Bootloader's copy rather than getting a new copy out of the archive. Prototypes for the library APIs are in `ezbl.h`, so this header also appears in the project under **Header Files**.
4. **Source Files** contains a Bootloader generated assembly file, `ex_boot_[i2c/uart/usb_msd].merge.S`. This source file contains a copy of the Bootloader's flash contents, Config words, and static/global reserved RAM locations. Additionally, the file contains symbol definitions and the addresses for all global functions and variables exported from the Bootloader project. All Bootloader flash contents are encoded as absolute data at pre-assigned addresses, so the effective contents of this file will not change when the Application project is rebuilt, regardless of compiler version, optimization settings, etc.
5. This project uses MPLAB defaults for all Project Properties except the "Load symbols when programming or building for production (slows process)" Loading option. This option can be useful for debugging purposes but is not required. The Project Properties need not match the Bootloader, nor does the same compiler version need to be used when building this Application project. In general, the Application is free to choose any toolchain options.
6. This project demonstrates how the Application can reuse the device initialization code and Bootloader timing/communications/device I/O abstraction APIs without having to duplicate implementations in the Application project.

### Example Usage

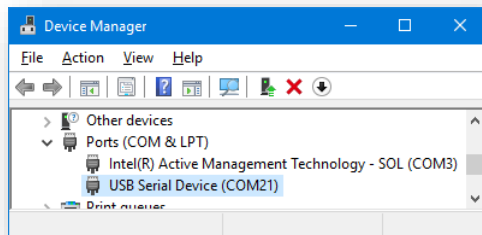
1. Build your Bootloader project for your desired device if you haven't already. See [ex\\_boot\\_uart](#), [ex\\_boot\\_i2c](#) or [ex\\_boot\\_usb\\_msd](#).
2. If your Bootloader project is still open in MPLAB, be sure that `ex_app_led_blink` is selected as the Main Project. To do this, right click on `ex_app_led_blink` in the MPLAB Projects window and choose **Set as Main Project**.



3. Select the correct Build Configuration profile to match your Bootloader:

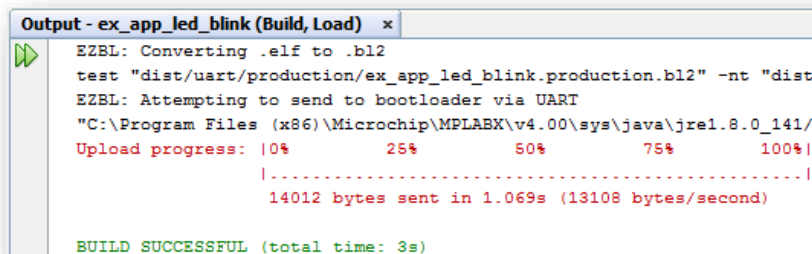


4. In the **Project Settings** for the selected build profile, select the same target PIC/dsPIC device as was specified when you built your Bootloader project.
5. [UART/I<sup>2</sup>C only] Ensure your target board is powered, has been programmed with your Bootloader using an ICSP hardware tool and has UART or MCP2221A I<sup>2</sup>C/USB cabling connected to your PC and target board.
6. [UART only] Ensure the correct system COM port is specified in the **ezbl\_app.mk** file under **Important Files**. On typical PCs running a Windows OS, a list of assigned COM ports can be found in the Device Manager (open by pressing WINDOWS\_KEY+R, then type `devmgmt.msc`):



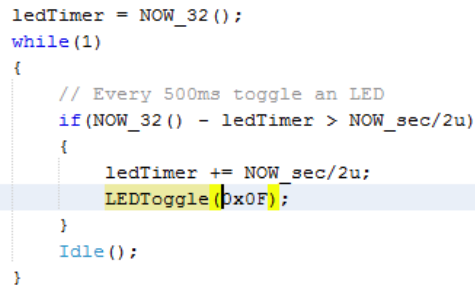
If you are using Linux or a Mac OS PC, upload will not be possible due to OS-specific driver requirements.

7. Build the **ex\_app\_led\_blink** project
  - a) [UART/I<sup>2</sup>C only] If the Bootloader was running and the PC was able to communicate with it, then you should see live bootloading progress appear in the MPLAB X build Output window:



If you instead observe something like: "Timeout reading from 'COM21'", see [OxFC13: EZBL\\_ERROR\\_LOCAL\\_COM\\_RD\\_TIMEOUT \(-1005\)](#), or more generally, the [Communications Status Codes and Error Messages](#) section.

- b) [USB MSD only] Using your system's file manager, copy the `dist\usb_msd\production\FIRMWARE.BL2` file to the **root directory** on a USB thumb drive and then plug your media into your target circuit. The media must be FAT16/FAT32 formatted and implement a hardware sector size of 512 bytes (unless you enabled a larger FILEIO library sector size option when building your Bootloader).
8. One second after bootload completion you will observe an LED change from blinking very rapidly (8Hz) to a much slower 1Hz (toggling every 500ms). This indicates that the Bootloader has exited and this example LED blinking Application is running instead.
9. To confirm that the application code is updatable, try changing the number of LEDs that toggle every 500ms. For example in `main.c`, inside the `main()` `while(1)` loop, change the line:
- ```
LEDToggle(0x01);
```
- to:
- ```
LEDToggle(0x0F);
```



```
ledTimer = NOW_32();
while(1)
{
    // Every 500ms toggle an LED
    if(NOW_32() - ledTimer > NOW_sec/2u)
    {
        ledTimer += NOW_sec/2u;
        LEDToggle(0x0F);
    }
    Idle();
}
```

10. Build the `ex_app_led_blink` project
- a) [USB MSD only] Place the new `FIRMWARE.BL2` file on the USB media by repeating step 7.b). This time, after plugging the media into your application circuit, either toggle an MCLR reset button or power cycle the board to reset back into the Bootloader. The USB MSD Bootloader only checks for new firmware when the device is reset.
11. Upon bootload completion, the number of LEDs that blink on the target board will change such that several are now toggling every 500ms instead of only one LED.
- a) Pushing the MCLR reset button or cycling power while watching the LEDs will demonstrate a visual status of the start up of Bootloader code, a one second "un-bricking" delay, following by a transition to the Application code.

For USB MSD bootloaders, the one second start up delay may actually be shorter than one second if USB media is already attached and it contains no `FIRMWARE.BL2` file on it, or the file exists, but contains the same Application as already present in flash.

12. [USB MSD only] Momentarily depress the **S4** push button. The LED will halt for a few seconds and if the USB media has an activity indicator light, it may blink instead. During this time, the Application demo code will write a `TEST.TXT` file in the root directory of the USB media.
13. [USB MSD only] Plug the USB media into a PC and confirm the presence and contents of the `TEST.TXT` file.
14. [USB MSD only] Reinsert the USB media into the application circuit and push buttons **S6** then **S3**. S6 will trigger a file read operation against `TEST.TXT` and display a status code on the LEDs. S3 will trigger a file delete operation and remove `TEST.TXT` from the media.
15. [USB MSD only] Plug the USB media into a PC and confirm successful deletion of the `TEST.TXT` file.

### Additional Implementation Notes

1. The `bl2_blob_elf_hex_content_view.bat` Windows batch file contained in your `ezbl_integration` folder can be very handy for debugging and quickly visualizing flash contents:
  - a) Using Windows Explorer, click and drag a file of type `.bl2`, `.blob`, `.elf` or `.hex` and drop it directly on top of the `bl2_blob_elf_hex_content_view.bat` file.
  - b) `.bl2` and `.blob` files will be decoded and displayed directly, whereas `.elf` files will be first converted to a `.hex` file which subsequently gets converted to a `.bl2` file for decoded display generation.
  - c) When decoding `.elf` flash contents, the XC16 compiler's `xc16-bin2hex.exe` utility must be invoked to generate a `.hex` file. Additionally, as `.elf` files still contain useful debugging meta data, `ezbl_tools.jar` will invoke `xc16-objdump.exe` to extract the `BOOTID_HASH` data stored within. These internal steps require that a valid XC16 bin folder appears in your system path. Ex: "C:\Program Files (x86)\Microchip\xc16\v1.35\bin".
2. The `upload_app.bat` batch file is useful when you wish to upload new firmware to a bootloader without using or needing MPLAB X. It is exercised in an identical manner to `bl2_blob_elf_hex_content_view.bat` by click and dragging a `.bl2` or `.elf` file directly on top of it. However, before doing so, edit the batch file and correctly set the COM port and baud rate parameters.
3. It is a good idea to open the `ezbl_lib` MPLAB X IDE project if you are debugging your Bootloader or Application project that calls `ezbl_lib` APIs. By keeping this project open, you will gain quick access to source files when additional information may be needed on an API's internal operation. Additionally, the debugger can normally open the actual `ezbl_lib` source files when you step into a function contained in the `ezbl_lib.a` archive, matching the source against the debug Program Counter.
  - a) Note, however, that `ezbl_lib.a` was compiled with `-Os` optimizations to minimize the code size. This makes most of the code very difficult to track when one-stepping.
  - b) In the event you wish to debug (or change) something in the library code, copy the applicable source file(s) out of `ezbl_lib` and place it in the **Source Files** tree in your own project. When you rebuild and program your project, the linker will select the function and variable declarations in your local project source files preferentially over the `ezbl_lib.a` copies, but will still fallback and use `ezbl_lib.a` contents for any items you do not copy over. In this manner, you will gain the ability to debug the sources for various library APIs using your global optimization level (or file override optimization level).
4. It is strongly recommended that you carry the `ezbl_lib`, `ezbl_comm`, and `ezbl_tools` folders around with your projects, checking them into source control, compressing them as needed, but always ensuring they are available and are an exact version match for the associated `ezbl_lib.a`, `ezbl_comm.exe`, and `ezbl_tools.jar` binaries that you have in your `ezbl_integration` folders. This is recommended for future maintenance and support reasons.

### Making a New or Existing Application Bootloadable

Refer to [help\EZBL Hands-on Bootloading Exercises.pdf](#), Exercise 3.

### [UART/I<sup>2</sup>C only] Bootloading Outside MPLAB® X IDE

Sending an Application's `.bl2` file to an EZBL Bootloader outside of MPLAB® X IDE will look and behave much the same as it does inside the IDE. However, instead of having the necessary commands invoked from a post-build step in a makefile, the user will need to invoke the command from a batch file or Command Prompt. The upload status will then be displayed in the console window instead of MPLAB's build Output window.

For an end user to access your Bootloader, you must minimally redistribute:

- `ezbl_comm.exe`
- `"ex_app_led_blink.production.bl2"`

`ezbl_comm.exe` is a file transfer tool, allowing the `.bl2` file to be copied to a communications port or the MCP2221A I<sup>2</sup>C interface under EZBL's software flow control requirements. It does not require installation or have other file dependencies. After being launched, it will close automatically upon bootloading completion or timeout.

It does not contain a GUI and only accepts parameters on the command line with status information printed to a console window.

"ex\_app\_led\_blink.production.bl2" is your Application firmware update file, generated automatically from your .elf build output. This .bl2 file is tied to your Bootloader by the BOOTID\_\* strings in your Bootloader project's makefile, so will be rejected by other EZBL Bootloader projects if an attempt is made to program it to an incorrect hardware device. It is suggested that you rename this file to include the APPID\_VER number (or other version identification string) to minimize user confusion if multiple Application versions are released over time.

With these files in the same location, the procedure to transfer the .bl2 image to the Bootloader is:

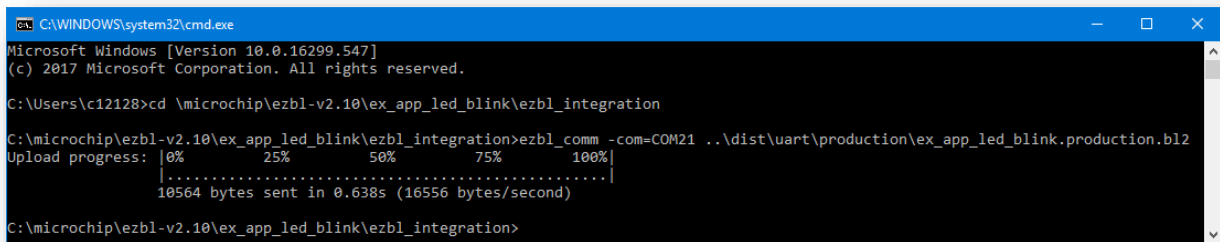
1. Open a Command Prompt
2. Change to the directory containing both files
3. Determine the correct COM port that the Bootloader is attached on
4. Execute the upload command, such as:

```
ezbl_comm -com=COM21 -baud=115200 -timeout=1100  
ex_app_led_blink.production.bl2
```

Or, for I<sup>2</sup>C bootloaders attached to an MCP2221A:

```
ezbl_comm -com=I2C -i2c-address=0x60 -baud=400000 -timeout=1100  
ex_app_led_blink.production.bl2
```

Both command examples need to be inputted as a single command on one line.

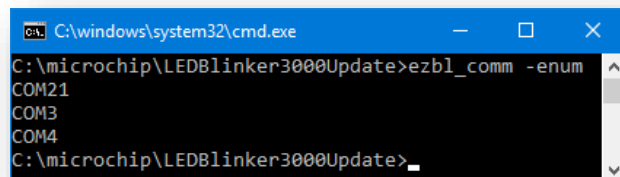


```
C:\WINDOWS\system32\cmd.exe  
Microsoft Windows [Version 10.0.16299.547]  
(c) 2017 Microsoft Corporation. All rights reserved.  
  
C:\Users\c12128>cd \microchip\ezbl-v2.10\ex_app_led_blink\ezbl_integration  
  
C:\microchip\ezbl-v2.10\ex_app_led_blink\ezbl_integration>ezbl_comm -com=COM21 ..\dist\uart\production\ex_app_led_blink.production.bl2  
Upload progress: |0%      25%      50%      75%      100%|  
                  |.....|  
                  10564 bytes sent in 0.638s (16556 bytes/second)  
  
C:\microchip\ezbl-v2.10\ex_app_led_blink\ezbl_integration>
```

Grey parameters are tool defaults and can be omitted from the command if no changes are needed. Highlighted options must be specified and generally will be different from the examples shown.

[UART only] To aid step 3, the following command can be used to display the COM ports present on the PC:

```
ezbl_comm -enum
```



```
C:\windows\system32\cmd.exe  
  
C:\microchip\LEDBlinker3000Update>ezbl_comm -enum  
COM21  
COM3  
COM4  
C:\microchip\LEDBlinker3000Update>
```

[I<sup>2</sup>C only] If the system has more than one MCP2221A attached, different instances can be referenced by adding an indexing suffix to the 'I2C' port name. For example: -com=I2C2 will specify a second MCP2221A that isn't

'I2C1'. 'I2C' is treated the same as 'I2C1'. Indexing starts from 1 and ends at the number of MCP2221A devices currently attached to the system, with no guaranteed order for the devices anytime one is removed or added.

To reduce the need to type commands, a batch file can be created and distributed to specify the parameters (except for the COM port) as they will generally be constants for your Bootloader and firmware release, as shown on the following page as "Update LED Blinker 3000.bat." Batch files can be executed by double clicking on them within Windows Explorer, negating a need to manually launch a command console.

For more advanced products, a GUI application should be developed and released for handling your firmware update releases. See [help\EZBL Communications Protocol.pdf](#).

#### Update LED Blinker 3000.bat

```
@echo off
setlocal
set USER_COM=
echo Please specify the communications port that LED Blinker 3000
echo is connected on. It will be updated to firmware v3001.
ezbl_comm.exe -enum
echo.
set /p USER_COM=Enter nothing to abort:
if "%USER_COM%"==" " goto UserAbort
ezbl_comm.exe -baud=230400 -timeout=1100 -log="update_log.txt" ^
-artifact="ex_app_led_blink.production.bl2" ^
-com=%USER_COM%
goto End

:UserAbort
echo Firmware update aborted.

:End
pause
@echo on
```

A copy of this script for modification can be found in [ex\\_app\\_led\\_blink\ezbl\\_integration\Update LED Blinker 3000.bat](#)

When executed from Windows, the script will generate a dialog and send the .bl2 firmware image to the specified the communications port:

```
C:\windows\system32\cmd.exe
Please specify the communications port that LED Blinker 3000
is connected on. It will be updated to firmware v3001.
COM21
COM3
COM4
Enter nothing to abort: com21
Upload progress: |0%      25%      50%      75%      100%|
                  |.....|
                  10900 bytes sent in 1.147s (9503 bytes/second)
Press any key to continue . . .
```



## *ex\_boot\_i2c* MPLAB® X Project

This example 16-bit PIC24/dsPIC Bootloader project implements a 2-wire I<sup>2</sup>C Slave interface for typical, single partition bootloading initiated by an I<sup>2</sup>C Master. The master may be some larger SoC onboard or a management interface from a PC through application specific communications bridging hardware. For immediate use and testing, this Bootloader is compatible with the Microchip MCP2221A USB to UART/I<sup>2</sup>C bridge chip.

The [ex\\_app\\_led\\_blink](#) example Application project targets this Bootloader when the 'i2c' Build Configuration is selected.

The communications protocol implemented on the wire is documented in [help\EZBL Communications Protocol.pdf](#). The protocol is nearly identical to UART implementations. The I<sup>2</sup>C master to Bootloader I<sup>2</sup>C slave data stream is identical and carries the firmware ".bl2" file to the slave Bootloader's I<sup>2</sup>C bus address verbatim. Communications in the reverse direction is slightly amended to permit software flow control, early abort, and final status messages to be read by the I<sup>2</sup>C master since I<sup>2</sup>C slaves cannot initiate bus transfers.

This project implements two interrupt handlers:

- Slave I<sup>2</sup>C at IPL1
- 16-bit Timer or 16-bit CCP Timer (for timeout detection) at IPL2

The exact I<sup>2</sup>C and Timer/CCP peripheral instances used by the Bootloader is board/processor specific and selectable in a hardware initialization file. Additionally, if more than one I<sup>2</sup>C (and/or UART) peripheral is initialized, the Bootloader will accept firmware updates from multiple interfaces. Most hardware initialization files distributed with EZBL default to using the Timer 1 and I<sup>2</sup>C 1 peripherals (with normal/not-alternate SDA/SCL pin assignment), but due to physical I/O pin mapping on some development boards and PIMs, this selection should not be assumed.

The Bootloader's interrupt handlers, by default, will stay enabled when the Application is launched, but the vectors are not hard-clobbered. See [Bootloader Interrupt Handling](#) if the Application wishes to control these.

### When Built

1. Same as [ex\\_boot\\_uart](#), except file names start with **ex\_boot\_i2c** instead of **ex\_boot\_uart**.

### Supports

1. Same as [ex\\_boot\\_uart](#).

### Notes

1. Same as [ex\\_boot\\_uart](#), except:
  - a. `EZBL_BOOT_PROJECT` global macro definition in the XC16 (Global Options) project properties is changed to `EZBL_BOOT_PROJECT=i2c` (instead of `EZBL_BOOT_PROJECT=uart`). This change does not have any current impact on the code, which only tests if the macro is defined or not.
  - b. The project's Build Configuration is named **i2c** instead of **uart**. This causes MPLAB X IDE to define the `XPRJ_i2c` preprocessor macro instead of `XPRJ_uart`. This, in turn results in a call to `I2C_Reset()` instead of `UART_Reset()` in the hardware initialization files after preprocessing. The hardware initialization .c files themselves are identical between projects.
  - c. `ex_boot_i2c/main.c` is a different source file, duplicating the same logic as `ex_boot_uart/main.c`, but without references to `EZBL_COMBaud` and `EZBL_FIFOSetBaud()`. The removed code is UART auto-baud specific and not applicable for I<sup>2</sup>C.

### Example Usage

1. Same as [ex\\_boot\\_uart](#), except:
  - a. `InitializeBoard()` calls `I2C_Reset()` instead of `UART_Reset()`
  - b. File names start with **ex\_boot\_i2c** instead of **ex\_boot\_uart**



- c. Discussion regarding erased Config words and their impact to available baud rates is not applicable. I<sup>2</sup>C slave communications are clocked by the I<sup>2</sup>C master with hardware clock stretching taking place if the slave needs more time to respond to a transfer.

However, a Bootloader that relies on an Alternate SDA/SCL pin assignment from the chip's default SDA/SCL pins will likely require the Config word controlling the assignment to be defined in the Bootloader project and not subsequent Application projects.

## *ex\_boot\_uart MPLAB® X Project*

This example Bootloader project for 16-bit PIC24/dsPIC devices implements a 2-wire UART interface for typical, single partition bootloading from a PC or other device capable of writing a file to a serial interface under a software flow control mechanism.

The [ex\\_app\\_led\\_blink](#) example Application project targets this Bootloader when the 'uart' Build Configuration is selected.

The communications protocol implemented on the wire is documented in [help\EZBL Communications Protocol.pdf](#) and is identical between 16-bit and 32-bit implementations of EZBL. The protocol assumes "reliable", in order transmission of stream-oriented data (i.e. no obvious start, end, or block size above the minimum of 8-data bits per atomic unit). The maximum round trip latency of the medium needs to be known at design time, but when set appropriately, the serial protocol can normally be tunneled through Bluetooth SPP (Serial Port Profile) radios and other bridges.

"Reliable" in this context means the firmware expects no bit errors or lost data in either the TX or RX directions throughout a complete firmware update event. However, any bit errors or lost data that may occur will still be detected and can be recovered by restarting the complete firmware update sequence.

Signaling is assumed to be full-duplex, point-to-point. Data transmitted to the Bootloader exactly matches the .bl2 file contents consumed by the Bootloader. Data return to the host from the Bootloader is limited to software flow control signaling and a final termination/status code. If the underlying physical layer implements carrier sense (i.e. local transmitter knows if the medium is idle and blocks when the receiver is active), then this project can also be used with a half-duplex, point-to-point communications link. A half-duplex, multi-point to multi-point broadcast bus may also work if all non-participating nodes remain silent during the firmware transfer (or a virtual point-to-point channel can be set up via a transparent sideband mechanism, like a 9<sup>th</sup> data bit flagging bootloader traffic differently from ordinary bus traffic).

As configured, this project implements three interrupt handlers:

- UART RX at IPL2
- UART TX at IPL1
- 16-bit Timer or 16-bit CCP Timer (for timeout detection) at IPL2

The exact UART and Timer/CCP peripheral instances used by the Bootloader is board/processor specific and selectable in a hardware initialization file. Additionally, if more than one UART (or I<sup>2</sup>C) peripheral is initialized, the Bootloader will accept firmware updates from multiple interfaces. Most hardware initialization files distributed with EZBL default to use UART 2 and Timer 1, but EZBL supports UART1 - UART6, I2C1 - I2C3, TMR1 - TMR6 or CCP1 - CCP8 (when present on the target device). Switching/adding interfaces or switching timers is trivialized within the [UART\\_Reset\(\)](#), [I2C\\_Reset\(\)](#) and [NOW\\_Reset\(\)](#) macros.

The Bootloader's interrupt handlers, by default, will stay enabled when the Application is launched, but the vectors are not hard-clobbered. See [Bootloader Interrupt Handling](#) if the Application wishes to control these.

### **When Built**

1. Outputs a Bootloader .hex file, [ex\\_boot\\_uart.production.hex](#), for use with a traditional ICSP based programmers (PICKit, ICD, REAL ICE, etc.)
2. Generates [ezbl\\_integration/ex\\_boot\\_uart.merge.gld](#) and [ex\\_boot\\_uart.merge.S](#) files. These files are meant to be included in any Application project that needs to be programmed through the Bootloader. These files specify all flash address and data contents for the Bootloader, as well as all public symbols. i.e. addresses of global variables and functions in the Bootloader.
3. Copies both the [ex\\_boot\\_uart.merge.gld](#) and [ex\\_boot\\_uart.merge.S](#) files to the [ex\\_app\\_led\\_blink/ezbl\\_integration](#) folder for development and testing in the [ex\\_app\\_led\\_blink](#) project.

## Supports

1. All PIC24FJ/PIC24H/PIC24E and dsPIC33F/dsPIC33E/dsPIC33C products (PIC24FxxK K-flash flash devices and dsPIC30F not supported)
2. Out-of-box demo ability on Explorer 16/32 development board (or Explorer 16 with external USB to RS232 converter)
3. Out-of-box demo ability on numerous PIMs. See the [hardware\\_initializers](#) project folder contents to check for immediately testable targets.
4. Tested with MPALB® X IDE v4.20
5. Tested with MPLAB XC16 compiler v1.35

## Notes

1. **Important Files** contains a customized makefile, `ezbl_boot.mk`. Additionally, `Makefile` has been trivially modified at the bottom to include `ezbl_integration/ezbl_boot.mk`. The added make script alters the project building behavior by executing a pre-build step launching `ezbl_tools.jar` and a post-build step that copies the `ex_boot_uart.merge.gld` and `ex_boot_uart.merge.S` build artifacts to other folders.
2. **Linker Files** contains a customized linker script, `ezbl_build_standalone.gld`. The `ezbl_tools.jar` utility modifies the linker script according the target processor selected in your project and your compiler's default linker script. Therefore, no manual edits to this file are necessary.
3. **Libraries** contains the `ezbl_lib.a` precompiled archive library. This file houses the object code for the API definitions in `ezbl.h` and is necessary to successfully link the project.
4. By default the UART operates in auto-baud mode so the host can choose the communications baud rate. This also allows the Bootloader to work with clock sources that have an unknown frequency or wide tolerance specification. If auto-baud is not desirable, such as when tunneling through a radio that requires a fixed baud rate, define a fixed baud rate using the `EZBL_COMBaud` constant at the top of your hardware initialization file. At run time, the baud rate can be changed (or auto-baud enabled/disabled) by calling `EZBL_FIFOSetBaud()`.
5. This project uses the following Project Properties which may differ from MPLAB defaults:

Category	Sub Category	Value
Loading		Load symbols when programming or building for production (slows process)
XC16 (Global Options)	Global options	Define common macros: <code>EZBL_BOOT_PROJECT=uart</code>
xc16-gcc	General	Isolate each function in a section
		Place data into its own section
	Optimizations	Optimization level=1
	Preprocessing and messages	Additional warnings
xc16-ld	General	Create Default ISR ( <i>unchecked</i> )
		Remove unused sections

With a possible exception for the `EZBL_BOOT_PROJECT=uart` common macro that enables device Config word definitions in the hardware initialization file, none of these project changes are critical to correct Bootloader compilation or operation.

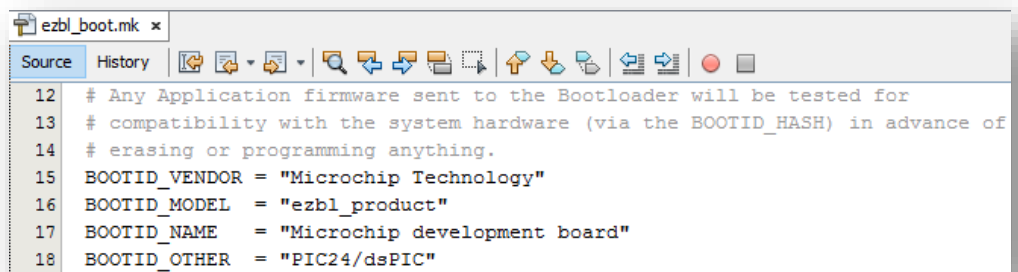
Symbol loading benefits development as the "Program Memory" Target Memory View window in MPLAB X IDE will not be populated in the absence of this option, nor would disassembly listings be available.

Isolation of functions and data into their own sections permits the "Remove unused sections" linker option to delete unreferenced code and variables, thus saving space.

As a Bootloader project, a `_DefaultInterrupt()` ISR created by the linker is unnecessary since all unhandled interrupts are automatically mapped into the Application's Interrupt Goto Table. The default, checked, state of this option would generate dead code.

### Example Usage

1. Select the desired target PIC24/dsPIC33 device in the MPLAB X **Project Properties**
  - a) Also set the desired Hardware Tool and Compiler Toolchain
2. Open the `ezbl_boot.mk` file under **Important Files** in the MPLAB Projects tree-view.
  - a) Change the `BOOTID_VENDOR`, `BOOTID_MODEL`, `BOOTID_NAME` and/or `BOOTID_OTHER` strings to be unique for your hardware product that will be running this Bootloader. The exact strings chosen are unimportant, with the field names chosen solely to suggest content that will generate a globally unique identification hash. The hash prevents your Bootloader from accepting Application firmware uploads that don't match your hardware or Bootloader version.



```

ezbl_boot.mk
Source History
12 # Any Application firmware sent to the Bootloader will be tested for
13 # compatibility with the system hardware (via the BOOTID_HASH) in advance of
14 # erasing or programming anything.
15 BOOTID_VENDOR = "Microchip Technology"
16 BOOTID_MODEL  = "ezbl_product"
17 BOOTID_NAME   = "Microchip development board"
18 BOOTID_OTHER  = "PIC24/dsPIC"
  
```

- b) Update the `appMergeDestFolders` folder list if you are targeting some other Application project(s) instead of (or in addition to) the `ex_app_led_blink` example Application. `$(thisMakefileDir)` appears in this list for the purpose of preserving the `ex_boot_uart.merge.gld` and `ex_boot_uart.merge.S` build output artifacts when a Clean or Clean and Build command is executed and serves as a fixed collection folder that is independent of production vs debug builds.
3. If you are targeting custom hardware or a Microchip development board/PIM that doesn't have a matching `hardware_initializer` file for it, copy an existing file that most closely matches your target device and edit it accordingly. The file (or some combination of files in your project) needs to:
  - Define any device Configuration words that you want in your Bootloader. These will be static and not modifiable when developing/programming new Application projects.
  - Declare the global `*EZBL_COMBootIF` pointer. This is referenced in `main.c` for handling auto-baud functionality. If you are using a fixed baud rate, you may delete this global variable along with the references to it in `main.c`.
  - Implement the `InitializeBoard()` function that sets the device clock (if needed), calls `NOW_Reset()`, initializes PPS and analog select SFRs applicable to your UART TX and RX pins and calls `UART_Reset()`. The parameters to `NOW_Reset()` and `UART_Reset()` select which peripheral instance will be initialized for the Bootloader and indirectly which ISRs your Bootloader contains.
  - Initialize GPIO SFRs for LED output(s) and call `EZBL_DefineLEDMap()`. If you don't have or want any LED(s) toggling, you may instead delete the `LEDToggle()` and `LEDOff()` calls in `main.c`.

Some initialization code located in the provided hardware initializer example files are applicable to other EZBL projects or are simply unneeded but potentially useful for generic hardware operations in Application code. Such code can be removed or left unimplemented.

Hardware initializer files that you aren't using/planning to use can be removed from the project as they are all mutually exclusive to each other and can contribute non-negligible project build duration.

4. With power applied and your ICSP programming tool connected, **Make and Program**
5. If your hardware implements an LED and was initialized appropriately, you will observe one LED blinking very rapidly at **8 Hz** (toggling every 62.5ms). This indicates that the Bootloader is executing and waiting for an Application to be uploaded. If the LED toggles much slower or not at all, check your Config words and oscillator/PLL initialization code. The frequency passed to `NOW_Reset()` should match your actual execution clock.

Slow (ex: FRC without PLL) execution normally still permits Bootloading, but requires a slow communications rate, such as 38400 baud. Additionally, the typical 1 second timeout before launching a valid Application will increase if programming an Application doesn't resolve the actual clock <--> `NOW_Reset()` frequency mismatch.

Programming an Application can resolve mismatches when oscillator or run-time clock switching enable Config word bits are defined in the Application project instead of the Bootloader. In this scenario, the Bootloader executes using the Config words in their erased state (all '1's for flash), which forces clocking from the internal FRC at 4.0 or 3.7 MIPS maximum and disables run-time clock switching.

6. You are now ready to test and use your Bootloader. Continue with the [ex\\_app\\_led\\_blink](#) Application project.

If you might be going to production with the Bootloader you just built, be sure to archive all your Bootloader project files (i.e. `ex_boot_uart` project folder, including the `dist` folder contents), the `ezbl_lib`, `ezbl_tools`, `ezbl_comm`, `help` folders, and the XC16 + MPLAB X IDE installers that you used for this Bootloader compilation. Don't forget an XC16 part support update installer if you installed one.

If you are updating an existing archive, make certain you capture the `[bootloader_proj_name].merge.gld` and `[bootloader_proj_name].merge.S` build artifacts in your backup as these are required in order to build Application projects that are compatible with your Bootloader.

All files are important to have backed up because they contributed to or are important towards using your Bootloader. If an unforeseen problem is encountered after releasing parts into the field, having the exact source files and tools used can expedite debugging, searching for possible solutions and/or creating an Application project to patch the Bootloader.

## *ex\_boot\_usb\_msd\_v201x\_xx\_xx MPLAB® X Projects*

The 16 and 32-bit *ex\_boot\_usb\_msd* Bootloader projects demonstrates a typical USB Mass Storage Device (MSD) class Bootloader (Host mode) in which new firmware images are presented to the Bootloader by plugging a USB thumb drive or other FAT16/FAT32 formatted media into the application circuit. The Bootloader contains code necessary to enumerate the USB MSD media, look for a new firmware image file placed by a user on it, and then autonomously erase and reprogram the application flash to match the file contents without any help or connectivity to a PC.

*ex\_boot\_usb\_msd* projects are testable using the 16-bit [ex\\_app\\_led\\_blink](#) or 32-bit [ex\\_app\\_led\\_blink\\_pic32mm](#) project, set to the 'usb\_msd' Build Configuration.

The bulk of the code and structure in these projects are identical to the *usb\_simple\_demo* projects distributed with the Microchip Libraries for Applications (MLA) at <http://www.microchip.com/mla>. Download and refer to the MLA documentation for USB, MSD, and FILEIO topics. The MLA release version that these bootloader projects are based around appears as a suffix to the generic "*ex\_boot\_usb\_msd*" project name.

Converting the MLA's *usb\_simple\_demo* projects to EZBL bootloaders entailed changes to the *main.c* file, device Configuration word changes to be more suitable for a Bootloader, and addition of the [\[ezbl\\_build\\_standalone.gld/ezbl\\_pic32mm.ld\]](#) linker script, the *ezbl\_lib[32mm].a* archive library, *ezb\_boot.mk* makefile with include from *Makefile*. Non-critical compiler optimization settings were also adjusted in the project settings to better accommodate a small code size.

### Supports

1. All 16-bit devices containing a USB peripheral and >= 64KB of flash or any of the 32-bit PIC32MM0256GPM064 Family devices
2. Explorer 16/32 development board, an Explorer 16 + USB PICTail Plus adapter, or a starter kit with USB Host functionality.
3. Tested with MPLAB® X IDE v4.20
4. Tested with MPLAB XC16 compiler v1.35 or XC32 compiler v2.10

### Notes

1. [32-bit only] Unlike all 16-bit USB MSD projects, the 32-bit *exp16\_pic32mm0256gpm064\_pim.x* project contains a non-shared copy of *../demo\_src/main.c*, renamed and stored in the base project folder as *main\_pic32mm.c*. The content of these two files are essentially identical, but small differences are required to properly handle Interrupt Enable SFR writes and check for abnormal causes of reset in the RNMICON SFR.
2. Because memory-type bootloaders must operate fully autonomously and generally have access to all new firmware file contents through a (presumed) reliable communications method, this Bootloader project implements a conditional two pass installation process that depends on the presence or absence of an existing Application that would be destroyed.

When a valid Application already exists in flash, two passes are used. In the first pass, the complete firmware file is read from the media and all erase/program/verification steps are executed in a simulation mode that completes all steps but blanks the actual NVM erase/program operations and read verification outcome for each step. This allows all file contents to be verified to exist (i.e. file not truncated), all record structure and address validity to be evaluated, and the full file's CRC32 integrity to be checked before deciding to erase the existing Application and proceed with bootloading.

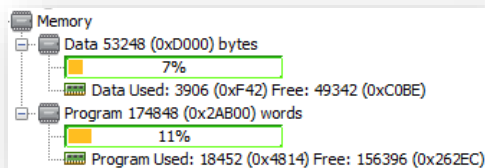
If no valid Application existed or the first dry run pass reached successful completion, the file read pointer is seeked back to file offset 0 and all data is read with real flash erase/write and verification steps executed.

3. [16-bit only] A PSV page on 16-bit devices is defined as an aligned, 0x8000 program address window mapping flash into data space. For Application code to be able to call Bootloader functions successfully, both the Bootloader and Application projects must place their constants on the same PSV page. Owing to the large total flash requirements for this Bootloader, it is possible for the linker to assign the Bootloader's .const PSV data section to a flash address that already contains Bootloader code filling most or all of the 0x8000 address PSV window. This would leave little to no ability to have PSV .const data in future Application projects.

To avoid this problem, the `EZBL_SetAppReservedHole()` macro is used in this Bootloader to force a Bootloader keep-out address range on the first PSV page and ensure a minimum amount of space is guaranteed available for future Applications. A consequence is that the Bootloader occupies discontinuous addresses in flash.

For example, instead of occupying addresses 0x000000-0x007800, the Bootloader may occupy flash addresses 0x000000-0x004000 and 0x008000-0x00B800, with the reserved hole in the middle. This ensures future Application projects will be able to use the 0x004000-0x008000 or 0x007800-0x010000 flash range for PSV constants, depending on which range matches the PSV page chosen by the linker when the Bootloader was built. The discontinuity of the address map for both the Bootloader and Applications is of no significance. `ezbl_tools.jar` will still create a valid .gld file for them and the Bootloader will still know what can and cannot be erase/programmed safely. When building Application projects, the linker's best-fit allocator will place code sections that can flow around the discontinuity, wasting few, if any bytes of flash when jumping to a new address.

An additional, benign consequence of using the `EZBL_SetAppReservedHole()` macro is that the MPLAB® X IDE Dashboard program space utilization will indicate the reserved space as part of the Bootloader flash footprint:

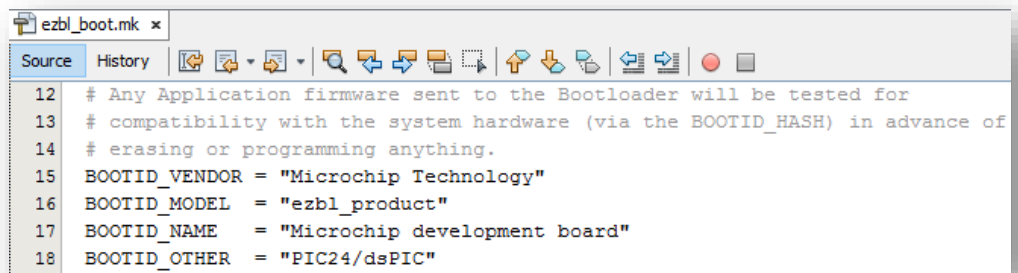


For this example, a 0x4000 program address sized hole was reserved, so 8192 or 0x2000 program words (24Kbytes) can be deducted as not actually part of the Bootloader project. In this case, the Bootloader's true size is ~31 Kbytes or 33KB after erase page padding.

### Example Usage

1. Open the MPLAB X example Bootloader project applicable or most similar to your hardware under `ex_boot_usb_msd_v201x_xx_xx/apps/usb/host/ex_boot_usb_msd/firmware/[processor_or_board_name].X]`
  - a. For PIC32MM targets, the `exp16_pic32mm0256gpm064_pim.x` project must be used
2. Open the `ezbl_boot.mk` file under **Important Files** in the MPLAB Projects tree-view.
  - a. Change the `BOOTID_VENDOR`, `BOOTID_MODEL`, `BOOTID_NAME` and/or `BOOTID_OTHER` strings to be unique for your hardware product that will be running this Bootloader. The exact strings chosen are unimportant, with the field names chosen solely to suggest content that will generate a globally unique identification hash. The hash prevents your Bootloader from accepting Application firmware uploads that don't match your hardware or Bootloader version.





```

ezbl_boot.mk
Source History
12 # Any Application firmware sent to the Bootloader will be tested for
13 # compatibility with the system hardware (via the BOOTID_HASH) in advance of
14 # erasing or programming anything.
15 BOOTID_VENDOR = "Microchip Technology"
16 BOOTID_MODEL  = "ezbl_product"
17 BOOTID_NAME   = "Microchip development board"
18 BOOTID_OTHER  = "PIC24/dsPIC"

```

3. Compile the Bootloader project and program it to the target board using an ICSP programming tool
4. Open the 16-bit [ex\\_app\\_led\\_blink](#) or 32-bit [ex\\_app\\_led\\_blink\\_pic32mm](#) example Application project, as appropriate for your target device
5. Select the 'usb\_msd' Build Configuration
6. Open the Project Properties and select the target processor that matches the one selected when compiling your Bootloader project
7. Compile the Application project
8. Copy the [ex\\_app\\_led\\_blink\[\\_pic32mm\]/dist/usb\\_msd/production/FIRMWARE.BL2](#) file to a FAT16/FAT32 formatted USB thumb drive's root folder. NOTE: exFAT is patent encumbered and not supported in the MLA.
9. Plug the USB media into the application circuit
10. The Bootloader will locate the [/FIRMWARE.BL2](#) file, check to see that it contains a valid `BOOTID_HASH` that matches the product, confirm that the `APPID_VER` data is not the same as the already installed Application, and if different or missing, erase and reprogram the Application's flash region from the [FIRMWARE.BL2](#) file
11. The Application will begin executing and you should observe one LED toggling every 500ms (1 Hz)
  - a. If a failure occurs, the lower 8-bits of the `EZBL_InstallFILEIO2Flash()` return code may be displayed on the board's LEDs, assuming 8 LEDs exist and have been initialized
12. To confirm that the application code is updatable, try changing the number of LEDs that toggle every 500ms. For example in the [ex\\_app\\_led\\_blink](#) project's [main.c](#) file, inside the `main()` `while(1)` loop, change the line:
 

```
LEDToggle(0x01);
```

 to:
 

```
LEDToggle(0x03);
```
13. Build the project and copy the [ex\\_app\\_led\\_blink\[\\_pic32mm\]/dist/usb\\_msd/production/FIRMWARE.BL2](#) file onto the USB media again, overwriting the original version
14. Plug the USB media into the application circuit
15. Power cycle the board or issue an MCLR reset to restart the Bootloader
16. Bootloader will detect the new firmware image, program it and launch the new Application
17. Confirm that two LEDs are now blinking at 1 Hz
18. Power cycle the board or issue an MCLR reset to restart the Bootloader
19. Observe that the Bootloader checks the USB media, decides the [FIRMWARE.BL2](#) contents already matches the existing Application and then immediately launches the Application. The Bootloader does not waste time and flash endurance erasing and reprogramming an unchanged [FIRMWARE.BL2](#) file.
20. The [ex\\_app\\_led\\_blink\[\\_pic32mm\]](#) projects contain some `FILEIO` calls to test reading/writing/deleting from the media for the Application, but using Bootloader APIs. To test these, see the last few, [USB MSD only] steps in the [ex\\_app\\_led\\_blink\[\\_pic32mm\] Example Usage](#) documentation.

## *ex\_boot\_app\_blink\_dual\_partition MPLAB® X Project*

This is a specialty project that combines both the Bootloader and Application functionality into the same project. It requires a target device with dual partition hardware capabilities (ex: PIC24FJ1024GA610/GB610 family, PIC24FJ256GB412/GA412 family, dsPIC33EP128GS808 family and dsPIC33EP64GS50x devices).

Bootloading functionality is accommodated via UART using the same protocol implemented in the *ex\_boot\_uart* project and using the same .bl2 binary firmware image files. However, unlike *ex\_boot\_uart*, no build time processing is done, no Interrupt Forwarding is performed, no linker script is required (uses compiler default) and in general the bootloading functionality is simpler as the run-time-self-programming target is always assumed to be the Inactive Partition.

By occupying isolated partitions, the Bootloader and Application code can fully intermingle with each other on the same flash erase pages, so in fact, there is no distinction as to what is the "Bootloader" and what is the "Application". The result is that everything is the Application or an extension to the Application.

This hardware paradigm is not without its risks. If ever the uploaded Application lacks or has broken bootloading functionality in it, and the FBTSEQ config word is programmed to make the last programmed partition always Active on reset, then the device effectively loses all ability to be reprogrammed. It is therefore imperative that proper testing of new firmware images be done before release to ensure the new image contains usable bootloader functionality in it.

Additionally, if critical changes are made to the bootloader code through the course of Application updates, then it becomes possible for devices in the field to have different and incompatible bootloader code in them, depending upon the latest firmware that was programmed. This could, in turn, require careful version control to ensure a valid reprogramming sequence is used to update a very old device to the latest firmware through sequential programming of intermediate versions.

### Supports

5. PIC24FJ1024GA610/GB610 family, PIC24FJ256GB412/GA412 family, dsPIC33EP128GS808 family, dsPIC33EP64GS50x devices and potentially other future dual partition devices.
6. Explorer 16/32 development board, or an Explorer 16 + USB to RS232 serial converter is required for out-of-box use.
7. Out-of-box demo ability on Microsoft Windows® OS only (due to UART hardware interface code being OS and platform dependent). Excluding uploading, development may be done with all OSes supported by MPLAB® X IDE and the XC16 compiler.
8. Tested with MPLAB XC16 compiler v1.35
9. Tested with MPLAB X IDE v4.20

### Example Usage

- 1) Open the *ezbl\_dual\_partition.mk* file under **Important Files** in the MPLAB Projects tree-view.
  - a) Change the `BOOTID_VENDOR`, `BOOTID_MODEL`, `BOOTID_NAME` and/or `BOOTID_OTHER` strings to be unique for your hardware product that will be running this Bootloader. The exact strings chosen are unimportant, with the field names chosen solely to suggest content that will generate a globally unique identification hash. The hash prevents your Bootloader from accepting Application firmware

uploads that don't match your hardware or Bootloader version.

```

ezbl_dual_partition.mk
Source History
13 # Any Application firmware sent to the Bootloader will be tested for
14 # compatibility with the system hardware (via the BOOTID_HASH) in advance of
15 # erasing or programming anything.
16 BOOTID_VENDOR = "Microchip Technology"
17 BOOTID_MODEL = "ezbl_product"
18 BOOTID_NAME = "Microchip development board"
19 BOOTID_OTHER = "dual flash partition device"

```

- b) A few lines down, under the `ezbl_post_build:` recipe, adjust the `"-com=COMx"` parameter to `ezbl_tools.jar` to match your PC's serial port. The baud rate can be any hardware supportable value as auto-baud is implemented by default. However, this project prints various status messages to the same UART used for bootloading, so when connecting a PC serial terminal application to view these messages, a fixed baud rate of 230400 is used by default. Turning auto-baud off or choosing a different baud rate for status messages can be accomplished by changing the `EZBL_COMBaud` constant near the top of `ezbl_uart_dual_partition.c`.

```

38 ezbl_post_build: .build-impl
39 # Create a binary .bl2 file from the .elf file and also if a loadable exists, convert the unified .hex file. The .elf
40 # file is converted to a binary image
41 @echo EZBL: Converting .elf/.hex file to a binary image
42 -test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" -nt "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.bl2"
43 @test "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex" -nt "dist/${CONF}/${IMAGE_TYPE}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex"
44 ifneq (,$(filter default uart,$(CONF))) # Check if "default" or "uart" MPLAB project build profile is used
45 @echo EZBL: Attempting to send to bootloader via UART
46 $(MP_JAVA_PATH)java -jar "$(thisMakefileDir)ezbl_tools.jar" --communicator -com=COM21 -baud=230400 -timeout

```

2. Compile and program a PIC24FJ1024GB610 or other dual partition PIM on an Explorer 16/32 or Explorer 16 via ICSP.
  - a) Ignore any EZBL communications errors in the build output window during this initial step
3. Observe LED D3 (right-most indicator) on the Explorer board blinking at 1 Hz. This is the example Application code, found in the `main.c` file, `main()` function's `while(1)` loop.
4. Change the `LEDToggle(0x01);` statement within the `main()` `while(1)` loop to `LEDToggle(0x0F);`
5. Compile the project, this time without programming it via ICSP.
6. Observe the build output window updates as the new application image is uploaded:

```

test "dist/default/production/ex_boot_app_blink_dual_partition.production.bl2" -nt "dist/default/production/ex_boot_app_blink_dual_partition.production.bl2"
BL2 content SHA-256 hash + CRC32 (stored at offset 0x000030C7) is: 218c3f86ba90f4a0fde7c76cd200112b9fae575179d4157e33569849c16e96
Successfully wrote 12523 bytes to dist\default\production\ex_boot_app_blink_dual_partition.production.bl2
EZBL: Attempting to send to bootloader via UART
"C:\Program Files (x86)\Microchip\MPLABX\v3.60\sys\java\jre1.8.0_121\bin\java -jar "ezbl_integration\ezbl_tools.jar" --communicator
Upload progress: |0%      25%      50%      75%      100%|
                  |.....|
                  12523 bytes sent in 1.110s (11282 bytes/second)

BUILD SUCCESSFUL (total time: 12s)

```

- a) During the image transfer, the existing Application will continue to execute without halting the CPU or disabling interrupts for any significant length of time (interrupts are disabled a number of times, but only for a handful of clock cycles to execute `NVMCON/NVMKEY` unlocking sequences, which require exact cycle timing and do not support speculative/polling methods).

- b) With a larger firmware image upload, it is even possible to observe that the 1 Hz blinking of LED D3 continues to execute throughout the bootload sequence, even though this is performed at IPL 0 in the main() while(1) loop. A larger firmware upload can be generated for testing by adding a static or global variable declaration with this syntax to a source file:
- ```
__prog__ char __attribute__((space(prog), keep)) dummy[0x7FFE];
```
- c) Flash erase/programming and general bootloader processing is written to occur at IPL0, alongside the lowest priority main() while(1) execution. However, Timer 1 at IPL 4, U2RX at IPL 2 and U1TX at IPL 1 all have interrupts implemented. These ISRs (contained in `ezbl_lib.a:weak_defaults/_T1Interrupt.s` and `uart2_fifo.c`) do only minimal processing to move data to/from RAM FIFOs and trigger tasks for processing at IPL 0. This implementation is minimally invasive to Applications with a strong base of existing APIs for the Application to share these same hardware resources as needed. The Timer and UART peripheral instances can be trivially adjusted in the hardware initialization file, and the ISRs are not sensitive to assignment under different priority levels.
- Observe that all 4 LSbits of the LED array are now blinking at 1Hz, as defined at step 4. The default implementation will decrement FBTSEQ after verifying successful firmware installation to the Inactive Partition and then perform a BOOTSWP run-time sequence to start executing the new firmware without resetting peripherals.
  - Open a Serial Terminal application on the PC and connect it to the same COM port and baud rate used for the EZBL upload (default 230400 baud, 8N1, no hardware or XON/XOFF flow control).
  - Toggle MCLR. Observe that Partition 2 persistently starts out of reset now and LED blinking matches the step 7 observation.

```
COM21:230400baud - Tera Term VT
File Edit Setup Control Window Help

Partition 2 starting up...
NUMCON<SFTSWP> = 0
Active Partition FBTSEQ = 0x001FFE
Inactive Partition FBTSEQ = 0x000FFF
```

- Toggle Button S4 (right-most button) on the Explorer board to execute a BOOTSWP instruction via the EZBL\_PartitionSwap() API. This will temporarily revert back to executing the original firmware programmed in step 2.
  - After clearing all interrupt enable bits, execution will continue starting at address 0x000000 on the newly activated partition (Partition 1).

```
COM21:230400baud - Tera Term VT
File Edit Setup Control Window Help

Partition 2 starting up...
NUMCON<SFTSWP> = 0
Active Partition FBTSEQ = 0x001FFE
Inactive Partition FBTSEQ = 0x000FFF
Button push detected: swapping partitions manually

Partition 1 starting up...
NUMCON<SFTSWP> = 1
Active Partition FBTSEQ = 0x000FFF
Inactive Partition FBTSEQ = 0x001FFE
```

11. Toggle MCLR or Button S4 again to revert to the step 6 (Partition 2) updated Application. The LED blinking will return to having 4 LSB LEDs blinking.
12. Hold down Button S3 (left-most button) and simultaneously toggle Button S4 (right-most button). This will write FBTSEQ on the Inactive Partition to be "-1" relative to the currently Active Partition's FBTSEQ and then issue a BOOTSWP to begin execution again with the step 2 Application. The partition with the lowest FBTSEQ value is made active at reset, so this effectively makes the partition you are about to swap into the future persistent one.

```

COM21:230400baud - Tera Term VT
File Edit Setup Control Window Help

Partition 2 starting up...
NUMCON<SFTSWP> = 0
Active Partition FBTSEQ = 0x001FFE
Inactive Partition FBTSEQ = 0x000FFF
Button push detected: swapping partitions manually
Also second button held:
Decrementing FBTSEQ on Inactive Partition so it is reset active...success

Partition 1 starting up...
NUMCON<SFTSWP> = 1
Active Partition FBTSEQ = 0x002FFD
Inactive Partition FBTSEQ = 0x001FFE

```

13. Toggle MCLR
  - a) Observe that Partition 1 containing the original 1 LED blink code has now been made persistent with the updated Application only accessible through an EZBL\_PartitionSwap() command.
14. Disconnect the Serial Terminal from the COM port or terminate the application process before attempting to recompile your project and upload new code to the Bootloader. Windows COM drivers do not permit simultaneous access to the same hardware resource from two processes.
15. Read through the source code. A number of interesting features may pop out, as will configuration options that may be useful in certain products. For example, by declaring a bigger RX FIFO, using a higher baud rate of 571000, and a faster USB to UART bridge, like the MCP2200 breakout board, programming throughput can be raised to at least 45KB/sec on the PIC24FJ1024B610/GA610 target and over 61KB/sec on dsPIC targets (at 761900 baud)
  - a) The code to declare the bigger RX FIFO is shown (commented out) near the top of `ezbl_uart_dual_partition.c`. See `UART2_RxFifoBuffer[]`.

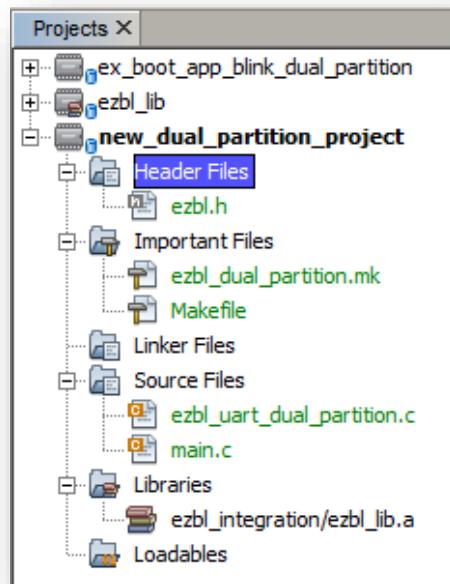
### Example Regeneration

To regenerate the `ex_boot_app_blink_dual_partition` MPLAB project or add the EZBL bootloader functionally to an existing project, follow these steps:

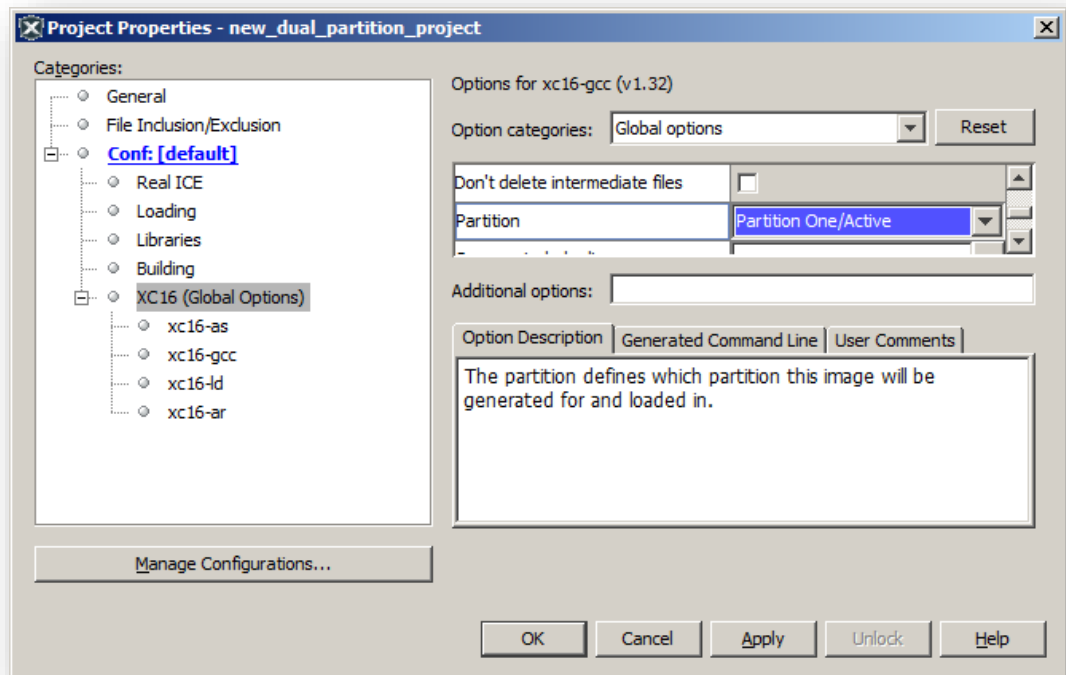
1. Start a new MPLAB project or open an existing one
2. Copy various files and folders from the `ex_boot_app_blink_dual_partition` example project to your base project folder using your OS's file manager. Your base project folder is the one containing a file called `Makefile` and the `nbproject` subfolder.
  - a) `ex_boot_app_blink_dual_partition/ezbl_integration` folder
  - b) `ex_boot_app_blink_dual_partition/ezbl_uart_dual_partition.c` file
  - c) `ex_boot_app_blink_dual_partition/Makefile`, overwriting the existing MPLAB generated copy of `Makefile`
    - (1) Alternatively, edit the existing `Makefile`, scroll to the bottom and add:
 

```
include ezbl_integration/ezbl_dual_partition.mk
```
  - d) If you started a new, blank project, copy the `ex_boot_app_blink_dual_partition/main.c` file as well

3. Within MPLAB X IDE, add the various files to their correct locations in the Projects tree-view pane:
  - `ezbl_integration/ezbl.h` → **Header Files**
  - `ezbl_integration/ezbl_dual_partition.mk` → **Important Files**
  - `ezbl_integration/ezbl_lib.a` → **Libraries** (use the "Add Library/Object File..." option on the right click menu and not "Add Library Project")
  - `ezbl_uart_dual_partition.c` (and `main.c`, if applicable) → **Source Files**

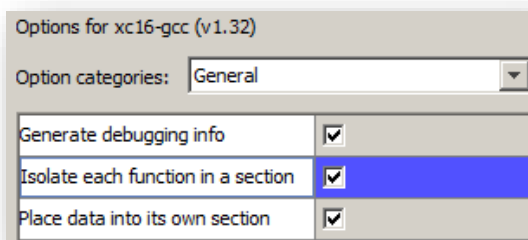


4. Open the Project Properties dialog
  - a) XC16 (Global Options)
    - (1) Set "Partition" to "Partition One/Active". Note: this option may not exist unless your project is first targeting a Dual Partition capable device. Be sure and set the device part number, click "OK", and reopen the Project properties dialog to see the "Partition" option.



## b) xc16-gcc

- (1) None of the remaining compiler options are critical to operation of EZBL code, however, turning on "Isolate each function in a section" and "Place data into its own section" can normally result in smaller flash and RAM requirements when combined with the "Remove unused sections" xc16-ld linker option.

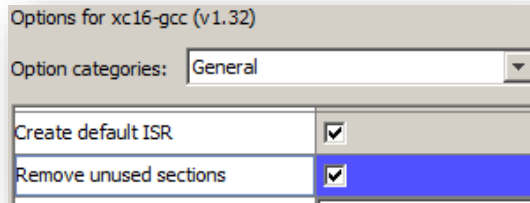


By compiling functions and variables into individualized sections, some compiler optimizations become less effective. However, this is typically offset by the beneficial ability to throw away more chunks of dead code/variables. In various source code libraries, such as MLA reference projects, numerous features may exist but will not actually get called or accessed in your project. The ability to throw away uncalled functions and variables automatically can cause appreciable size savings in such cases.

## c) xc16-ld

- (1) Turn on the "Remove unused sections" option if you've selected the section isolation options.





**Note:** discarded sections can cause the debugger and disassembly listings to show dead/discarded source code in places where the debugger does not have a source file with matching PC location to correctly display (such as libc or other library code you have stepped into).

Temporarily unselecting the "Remove unused sections" option may be needed. However, heightened awareness of what is reasonable can also help – anytime the debugger shows code you are not using anywhere in your project, you can ignore what is displayed as the effect of remnant DWARF debugging data that couldn't be simultaneously discarded when the linker discarded unused/uncalled functions.

5. Add the EZBL header to your main.c file to gain access to the many EZBL APIs:  

```
#include "ezbl_integration/ezbl.h"
```
6. Implement suitable hardware configuration and initialization code like the existing files in [ex\\_boot\\_app\\_blink\\_dual\\_partition/hardware\\_initializers](#). Especially important aspects are:
  - a) Device Config words need to be suitable with FBOOT set to enable the Dual Partition memory layout in hardware. Be sure to enable the BOOTSWP instruction in the FICD Config word. In addition to a valid clock, it is highly recommended that run-time clock switching be enabled (FOSC: CSECMD or CSECME).
  - b) `InitializeBoard()` needs to return a valid `FCY` frequency for correct `NOW_*()` API timing abilities. On dsPICs, the `FCY` definition is normally a control parameter that configures the PLL to achieve the given `FCY` target, but on PIC24F targets, `FCY` is generally serves an opposite role of getting accurate clock information from your Config word and physical hardware information into the software.
  - c) The `_U2RXR` and `_RPxR` register bitfields must be set for the I/O pins used for U2RX and U2TX functions.

Set `_U2RXR` equal to the RP or RPI number of the pin you are using for U2RX. If the same pin has analog functions, clear the `_ANSxy` bit to enable digital input functionality.

For the TX line, `_RPxR` should be set to `_RPOUT_U2TX` (or the literal 5), where 'x' is the RP number for the U2TX pin.

If PWMxH/PWMxL I/O pin functions exist on the communications pins, these I/O functions must be disabled as they take a higher priority on the pin function multiplexing order and reset to enabled.

7. Near the top of [ezbl\\_uart\\_dual\\_partition.c](#), set the `EZBL_ADDRESSES_PER_SECTOR` symbol value to match your device's flash page erase size, in program space addresses. The PIC24FJ1024GA610/GB610 family has a `0x800` page size (3072 bytes or 1024 instruction words) whereas many of the other dual partition device families implement a `0x400` page size (1536 bytes or 512 instruction words).
8. Update the `BOOTID_*` strings, `-com=COMx` and `-baud=` options in [ezbl\\_dual\\_partition.mk](#) and compile the project, as discussed in [Step 1 of the Example Usage](#) section.
9. Test the Bootloader by programming it with an ICSP method and then rebuild the project to invoke the post-build upload step.

## *ex\_app\_live\_update\_smmps\_v1, v2, v3 MPLAB® X Projects*

These three example projects demonstrate bootloading on a Dual Partition capable processor target while maintaining always-on, time-critical Application execution all throughout the erase, programming, partition execution hand-over events. With appropriate planning, task synchronization and testing, they permit a "Live Update" of all of the code in the project effectively as an instantaneous event without requiring a processor reset or any down time. These examples target Applications implementing high frequency control loops, such as Switch Mode Power Supplies where going offline for even a few 10s of microseconds would have prohibitive downstream consequences.

**Note:** Performing a successful Live Update requires appreciably more engineering effort than a typical Application firmware update. If your product can tolerate down time on the order of milliseconds to permit a full processor reset and state reinitialization, it is strongly suggested that you use the [ex\\_boot\\_app\\_blink\\_dual\\_partition](#) project instead of these Live Update projects to minimize development effort.

All three projects implement a combined UART Bootloader and SMPS Application intended for execution on the MPLAB® Starter Kit for Digital Power (dsPIC33EP64GS502 version), [DM330017-2](#). An MCP2221 Breakout Module ([ADM00559](#)) is also needed to supply logic level UART communications back to a host PC via USB to supply updated firmware images.

The three projects represent a chronological set, where *ex\_app\_live\_update\_smmps\_v1* implements a baseline reference SMPS Buck + Boost power supply and *ex\_app\_live\_update\_smmps\_v2/ex\_app\_live\_update\_smmps\_v3* are follow on upgrades to the baseline firmware that need to be installed in the middle of device operation without interrupting the SMPS outputs.

*ex\_app\_live\_update\_smmps\_v2* can only Live Update against v1. It follows a "Preserve All" linking model where most code and run-time state does not change in the code update. The small number of changes that take place, while critical to control loop operation, require a few RAM variables to be (re)initialized after the v1 to v2 execution handover. This permits complete handover in under 3μs, not missing any time-critical periodic interrupts.

*ex\_app\_live\_update\_smmps\_v3* can only Live Update against v2. It follows a "Preserve None" linking model where most code and run-time state is discarded and only variables deemed critical for continuous operation of the power supply survive the handover. The control loop ISRs are enabled before normal compiler variable initialization takes place or the main() function gets called. Critical handover completes in under 2μs with all other reset initialization code then executing normally without strict timing limits. This model affords greater flexibility when larger sections of code need to be updated which do not participate in timing critical operations.

If an attempt to upload firmware out of order occurs, the Bootloader will detect this and rather than attempt a Live Update partition swap that would corrupt run-time state, the LCD prints a message indicating the out-of-order upload and then resets the device to execute the new code.

For more information on using these projects, see Microchip AN2601 "Online Firmware Updates in Timing-Critical Applications" (DS00002601), downloadable on the Microchip.com web site.

### ***ex\_app\_non\_ezbl\_base MPLAB® X Project***

This is a trivial application project that does not depend on or use any EZBL bootloader or library code. It has default project settings for working through various getting started/training exercises.

See [help\EZBL Hands-on Bootloading Exercises.pdf](#)

## *ex\_app\_led\_blink\_pic32mm MPLAB® X Project*

This project demonstrates a trivial Application project that will be reprogrammed using your EZBL Bootloader. It is a companion test project intended for use with:

- [ex\\_boot\\_uart\\_pic32mm](#) - UART Bootloader for all PIC32MM devices
- [ex\\_boot\\_usb\\_msd/exp16\\_pic32mm0256gpm064\\_pim.x](#) - USB Host - Mass Storage Device (MSD) class "thumb drive" Bootloaders for PIC32MM0256GPM064 Family devices

When *ex\_app\_led\_blink\_pic32mm* successfully executes, it will toggle an LED/GPIO pin at 1 Hz (500ms per edge) to indicate success. When paired with the *ex\_boot\_usb\_msd* bootloader, three push button/GPIO pins will be also be monitored and when pushed, FILEIO API calls will be made to demonstrate reuse of the USB Host stack, MSD class and FILEIO library code contained in the Bootloader. These push buttons will trigger writing, reading, or deleting of a file on the USB media.

This project can be compiled, programmed into a device via a classic ICSP method, and debugged using ordinary debuggers without separately programming the Bootloader into the device beforehand. When programmed via ICSP, the target device receives both your Bootloader project's binary image and the code implemented by the *ex\_app\_led\_blink\_pic32mm* Application itself.

When uploaded or programmed via a preexisting Bootloader executing from flash, the bootloader image encoded in the *ex\_boot\_[uart\_pic32mm/usb\_msd].merge.S* file must match the Bootloader already in flash. If there is a mismatch, the Application generally fails to execute as intended. Therefore, it is critical that once a product is released to manufacturing, the *ex\_boot\_[uart\_pic32mm/usb\_msd].merge.S* file be archived and not edited. Similarly, while still editable in some cases, the *ex\_boot\_[uart\_pic32mm/usb\_msd].merge.ld* linker script file should be archived.

All code and referenced library code used in this project is meant to be replaceable by the Bootloader. Even shared functions or variables that are inherited from the Bootloader image can be replaced for purposes of Application use (the Bootloader will still use its original copy).

### When Built

1. Outputs a combined Bootloader + LED Blink Application .hex file, *ex\_app\_led\_blink\_pic32mm.[production/debug].hex*, for use with traditional ICSP based programmers
2. Outputs the combined Bootloader + LED Blink Application .elf file re-encoded into a compact binary .bl2 file, *ex\_app\_led\_blink\_pic32mm.[production/debug].bl2*. This .bl2 file is normally what you would distribute when you release a new Application firmware update. .bl2 files are similar to a .hex file, but EZBL creates them from the .elf file to obtain BOOTID\_HASH metadata and have it placed in the .bl2 file header.
3. [uart Build Configuration] Attempts to transfer the generated .bl2 file to the target device and permit immediate run-time execution testing.
4. [usb\_msd Build Configuration] Renames the .bl2 output image file in the dist folder to **FIRMWARE.BL2**. This enables drag-and-drop copying to a USB MSD media.

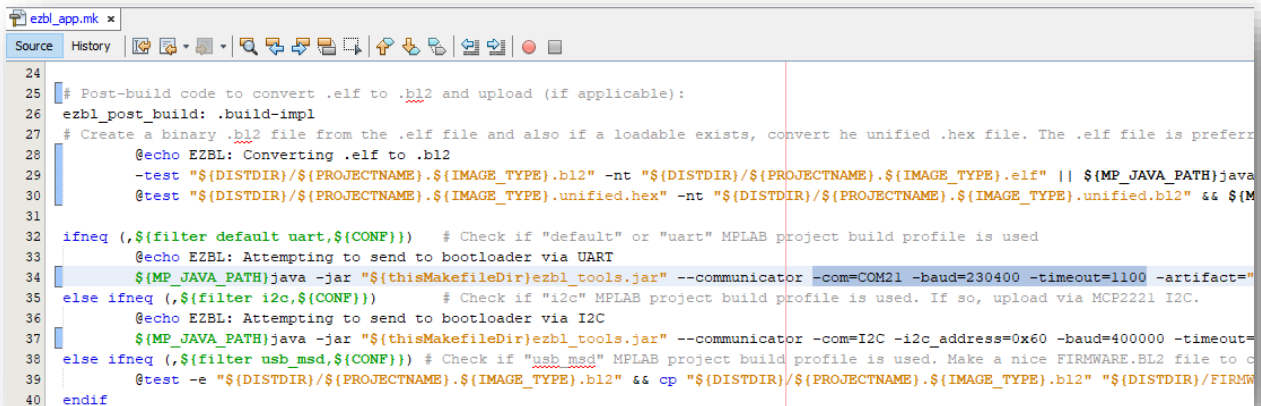
### Supports

1. Out-of-box demo ability on:
  - Explorer 16/32 (or Explorer 16) development boards ([DM240001-3](#) or [DM240001-2](#))
  - PIC32MM0064GPL036 ([MA320020](#)) and PIC32MM0256GPM064 ([MA320023](#)) Explorer 16/32 PIMs. All other PIC32MM devices are also supported, but will require appropriate pinout/PPS, UART selection, and/or device Config word changes.
2. [UART only] Attempts to transfer the generated .bl2 file to the target device and permit immediate run-time execution testing.
3. [USB MSD only] Copies the *ex\_app\_led\_blink\_pic32mm.[production/debug].bl2* file to **FIRMWARE.BL2**. The USB MSD Bootloader expects 8.3 formatted filenames with FIRMWARE.BL2 located in the root

directory of the mass storage media. This copy is created to allow easy copy/paste or drag-and-drop transfer to the media from any PC without requiring a manual file renaming step.

## Notes

1. **Important Files** contains a customized makefile, `ezbl_app.mk`. Additionally, **Makefile** has been trivially modified at the bottom to include `ezbl_integration/ezbl_app.mk`. The added make script alters the project building behavior by executing a pre-build step to increment an APPID\_VER version build number and a post-build step that converts a just-built .elf file to a .bl2 file before attempting to transfer the Application to the target Bootloader.
  - a) The `ezbl_post_build`: recipe in `ezbl_app.mk` does the .elf to .bl2 conversion. The recipe also attempts to upload the generated .bl2 file to the Bootloader, if applicable to the build configuration. Uploading is attempted solely for quicker development and debugging purposes within the IDE. Uploading is always possible outside the IDE and on systems which do not have MPLAB X IDE installed.
  - b) Using the automatic upload feature against a UART Bootloader normally requires the below communications parameters to be modified to match your development machine's COM port.



```

24
25 # Post-build code to convert .elf 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. The .elf file is preferred
28 @echo EZBL: Converting .elf to .bl2
29 -test "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" -nt "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.elf" || ${MP_JAVA_PATH}java
30 @test "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.unified.hex" -nt "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.unified.bl2" && ${MP_JAVA_PATH}java
31
32 ifneq (,$(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 -timeout=1100 -artifact="
35 else ifneq (,$(filter i2c,$(CONF))) # Check if "i2c" MPLAB project build profile is used. If so, upload via MCP2221 I2C.
36 @echo EZBL: Attempting to send to bootloader via I2C
37 ${MP_JAVA_PATH}java -jar "${thisMakefileDir}ezbl_tools.jar" --communicator -com=I2C -i2c_address=0x60 -baud=400000 -timeout=
38 else ifneq (,$(filter usb_msd,$(CONF))) # Check if "usb_msd" MPLAB project build profile is used. Make a nice FIRMWARE.BL2 file to c
39 @test -e "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" && cp "${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bl2" "${DISTDIR}/FIRMWARE
40 endif
  
```

If developing on a Linux or Mac OS platform, automatic firmware transfer will not work. Therefore, the applicable `ezbl_app.mk` lines should be commented out (lines 32 to 40 in the above example) by placing a '#' character at the very beginning of the lines.

The 'i2c' build configuration lines are not used in this project with EZBL v2.11 and can be deleted.

2. **Linker Files** contains a Bootloader derived linker script, `ex_boot_uart_pic32mm.merge.ld` or `ex_boot_usb_msd.merge.ld`, depending on selected Build Configuration. This linker script allows the Application to build without attempting to overlap any Bootloader owned flash addresses. The linker script is also used to save appropriate IVT data and allow run-time selection between Application implemented ISRs and Bootloader implemented ISRs whenever both projects implement the same interrupt.
  - a) If you wish to make manual changes to the `ex_boot_[uart_pic32mm/usb_msd].merge.ld` linker script, be aware that this file is auto-generated and will be overwritten if you rebuild your Bootloader project. To avoid being impacted by this, it is advisable to make changes to the Bootloader's `ezbl_pic32mm.ld` linker script instead and, if only applicable to the Application project, use an `#if !defined(EZBL_BOOT_PROJECT)` preprocessor conditional to have the Application-only modifications get excluded during Bootloader linking. When the Bootloader build process creates the `ex_boot_[uart_pic32mm/usb_msd].merge.ld`

output file, it starts by copying its own `ezbl_pic32mm.ld` linker script and then updates it with section mapping directives to ensure Bootloader memory contents are mapped to their required addresses. The result is that the two files are grossly identical with changes in the Bootloader's version propagating automatically into the Application's copy whenever the Bootloader is rebuilt.

3. **Libraries** contains the `ezbl_lib32mm.a` precompiled archive library. This file is not required, but since there are several potentially useful functions in `ezbl_lib32mm.a` for Application projects besides Bootloaders, inclusion of this library in the project allows convenient access to any of the EZBL APIs. When an `ezbl_lib32mm` API is called that was already called in the Bootloader project, the linker will preferentially use the Bootloader's copy rather than getting a new copy out of the archive. All the prototypes for the library are in `ezbl.h`, so this header also appears in the project under **Header Files**.
4. **Source Files** contains a Bootloader generated assembly file, `ex_boot_[uart_pic32mm/usb_msdc].merge.S`. This source file contains a copy of the Bootloader's flash contents, Config words and static/global reserved RAM locations. Additionally, the file contains symbol definitions and addresses for all global functions/variables exported from the Bootloader project. All Bootloader flash contents are encoded as absolute data at fixed addresses, so the effective contents of this file will not change when the Application project is built, regardless of toolchain version, optimization settings, etc.

This file contains a couple `#if !defined(EZBL_HIDE_BOOT_SYMBOLS)` preprocessor conditional statements. Defining this macro for both the xc32-as assembler and the xc32-ld linker has the effect of putting the Bootloader in a black box with no interfaces, isolating it from the Application. All Bootloader functions, variables, RAM use and ISR vectoring will no longer be visible/accessible to the Application at link time or run-time. This negates run-time resource use and the option to monitor for background firmware update requests in the Bootloader. The only parts of the Bootloader that will still exist and impact the Application afterwards would be:

- a) Existing SFR states at Bootloader execution handoff to the Application
- b) Flash footprint, including Config words. The Bootloader will still be in flash, execute at reset and flash overlap will still cause an Application linking conflict.

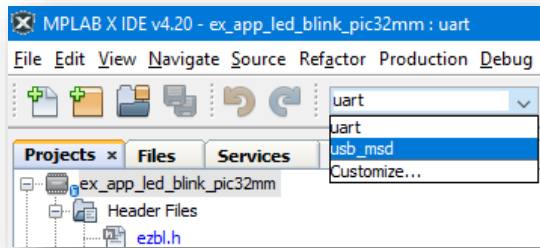
This option can be used if functional safety or other project requirements dictate project separation but is generally not recommended since the Application doesn't gain anything from it.

5. This project uses MPLAB defaults for all Project Properties except the "Load symbols when programming or building for production (slows process)" Loading option and "Display memory usage" xc32-ld Diagnostic option. These options can be useful for debugging purposes but are not required. The Project Properties need not match the Bootloader, nor does the same toolchain version need to be used when building the Application project. In general, the Application is free to choose any setting.
6. This project demonstrates how the Application can reuse the device initialization code and Bootloader timing/communications/device I/O abstraction APIs without having to duplicate implementations in the Application project.

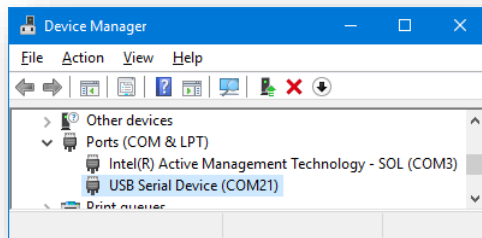
### Example Usage

1. Build your Bootloader project for your desired device if you haven't already. See [ex\\_boot\\_uart\\_pic32mm](#) or [ex\\_boot\\_usb\\_msdc](#).
2. If your Bootloader project is still open in MPLAB, be sure that `ex_app_led_blink_pic32mm` is selected as the Main Project. To do this, right click on `ex_app_led_blink_pic32mm` in the MPLAB Projects window and choose **Set as Main Project**.

3. Select the correct Build Configuration profile to match your Bootloader:

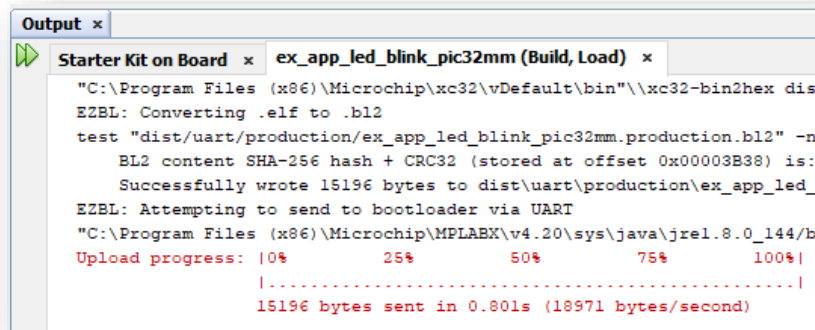


4. In the **Project Settings** for the selected build profile, select the same target PIC32MM device as was specified when you built your Bootloader project.
5. Ensure your target board is powered, has been programmed with your Bootloader using an ICSP hardware tool and, if using a UART bootloader, has UART or MCP2221A USB cabling connected to your PC and target board.
6. [UART only] Ensure the correct system COM port is specified in the **ezbl\_app.mk** file under **Important Files**. On typical PCs running a Windows OS, a list of assigned COM ports can be found in the Device Manager (open by pressing WINDOWS\_KEY+R, then type `devmgmt.msc`):



If you are using Linux or a Mac OS PC, upload will not be possible as the `ezbl_comm.exe` transfer utility has not been compiled on those OS's (see [ezbl\\_comm](#) for information on compiling it to support other platforms).

7. Build the `ex_app_led_blink_pic32mm` project
  - a) [UART only] If the Bootloader was running and the PC was able to communicate with it, then you should see live bootloading progress appear in the MPLAB X build Output window:





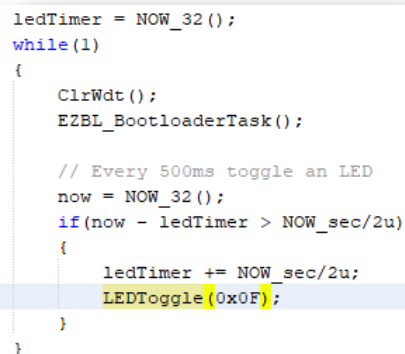
If you instead observe "Timeout reading from 'COMx'", see [0xFC13: EZBL ERROR LOCAL COM RD TIMEOUT \(-1005\)](#), or more generally, the [Communications Status Codes and Error Messages](#) section.

- b) [USB MSD only] Using your system's file manager, copy the `dist\usb_msd\production\FIRMWARE.BL2` file to the **root directory** on a USB thumb drive and then plug your media into your target circuit. The media must be FAT16/FAT32 formatted and implement a hardware sector size of 512 bytes (unless you enabled a larger FILEIO library sector size option when building your Bootloader).
8. One second after bootload completion you will observe an LED change from blinking very rapidly (8Hz) to a much slower 1Hz (toggling every 500ms). This indicates that the Bootloader has exited and this example LED blinking Application is running instead.
9. To confirm that the application code is updatable, try changing the number of LEDs that toggle every 500ms. For example in `main.c`, inside the `main()` while(1) loop, change the line:

`LEDToggle(0x01);`

to:

`LEDToggle(0x0F);`



```

ledTimer = NOW_32();
while(1)
{
    ClrWdt();
    EZBL_BootloaderTask();

    // Every 500ms toggle an LED
    now = NOW_32();
    if(now - ledTimer > NOW_sec/2u)
    {
        ledTimer += NOW_sec/2u;
        LEDToggle(0x0F);
    }
}

```

10. Rebuild the `ex_app_led_blink_pic32mm` project
  - a) [USB MSD only] The USB MSD Bootloader checks for new firmware only at device reset. Therefore, push the MCLR reset button once you've updated the `FIRMWARE.BL2` file on the USB media and reinserted it into the target board.
11. Upon bootload completion, the number of LEDs that blink on the target board will change such that several are now toggling simultaneously instead of only one. Note: physical location of the LEDs that now blink may not match the right-justified hex value in code. This happens on boards/PIMs which do not have all LEDs connected or which have other, prioritized hardware multiplexed on the same GPIO pin.
  - a) Pushing the MCLR reset button or cycling power while watching the LEDs will demonstrate a visual status of the startup of Bootloader code, a one second "un-bricking" delay, following by a transition to the Application code.

For USB MSD bootloaders, the one second start up delay may be shorter than one second if USB media is already attached and it contains no `FIRMWARE.BL2` file on it, or the file exists, but contains the same Application as already present in flash.

12. [USB MSD only] Momentarily depress the **S4** push button. The LED will halt for a few seconds and if the USB media has an activity indicator light, it may blink instead. During this time, the Application demo code will write a `TEST.TXT` file in the root directory of the USB media.
13. [USB MSD only] Plug the USB media into a PC and confirm the presence and contents of the `TEST.TXT` file.
14. [USB MSD only] Reinsert the USB media into the application circuit and push buttons **S6** then **S3**. S6 will trigger a file read operation against TEST.TXT and display a status code on the LEDs. S3 will trigger a file delete operation and remove TEST.TXT from the media.



15. [USB MSD only] Plug the USB media into a PC and confirm successful deletion of the **TEST.TXT** file.

### Additional Implementation Notes

1. The **bl2\_blob\_elf\_hex\_content\_view.bat** Windows batch file contained in the **ezbl\_integration** folder can be very handy for debugging and quickly visualizing flash contents:
  - a) Using Windows Explorer, click and drag a file of type **.bl2**, **.blob**, **.elf** or **.hex** and drop it directly on top of the **bl2\_blob\_elf\_hex\_content\_view.bat** file.
  - b) **.bl2** and **.blob** files will be decoded and displayed directly, whereas **.elf** files will be first converted to a **.hex** file which subsequently gets converted to a **.bl2** file for decoded display generation.
  - c) When decoding **.elf** flash contents, the XC32 toolchain's **xc32-bin2hex.exe** utility must be invoked to generate a **.hex** file. Additionally, as **.elf** files still contain useful debugging meta data, **ezbl\_tools.jar** will invoke **xc32-objdump.exe** to extract the **BOOTID\_HASH** data stored within. These internal steps require that a valid XC32 bin folder appears in your system path. Ex: "C:\Program Files (x86)\Microchip\xc32\v2.10\bin".
2. The **upload\_app.bat** batch file is useful when you wish to upload new firmware to a serial bootloader without using or needing MPLAB X. It is exercised in an identical manner to **bl2\_blob\_elf\_hex\_content\_view.bat** by click and dragging a **.bl2** or **.elf** file directly on top of it. However, before doing so, edit the batch file and correctly set the COM port and baud rate parameters.
3. It is a good idea to open the **ezbl\_lib32mm** MPLAB X IDE project whenever you are debugging your Bootloader or Application project that calls **ezbl\_lib32mm** APIs. By having it open, you will gain quick access to source files when additional information is needed on an API. Additionally, the debugger can normally open the **ezbl\_lib32mm** source files when you step into a function contained in the **ezbl\_lib32mm.a** archive.
  - a) Note: **ezbl\_lib32mm.a** was compiled with `-Os` optimizations to minimize the code size. The code may be difficult to track when one-stepping.
  - b) To debug at lower optimization (or change) something in the library, copy the applicable source file(s) out of **ezbl\_lib32mm** and place it in the **Source Files** tree in your own project. When rebuilding your project, the linker will select function and variable definitions in your local project preferentially over the **ezbl\_lib32mm.a** copies but will still fallback and use the archived contents for any items you haven't copied.
4. It is strongly recommended that you carry the **ezbl\_lib32mm**, **ezbl\_comm**, and **ezbl\_tools** folders around with your projects, checking them into source control, compressing them as needed, but always ensuring they are available and are an exact version match for the associated **ezbl\_lib32mm.a**, **ezbl\_comm.exe**, and **ezbl\_tools.jar** binaries that you have in your **ezbl\_integration** folders. This is recommended for future maintenance and support reasons.

### Making a New or Existing Application Bootloadable

Refer to [help\EZBL Hands-on Bootloading Exercises.pdf](#), Exercise 3. Information in this document was developed against the 16-bit EZBL implementation, so some substitutions or extrapolations will be needed. Notably, "ex\_boot\_uart" should be treated as "ex\_boot\_uart\_pic32mm", and the ".gld" linker script will have a ".ld" extension instead.

### [UART only] Bootloading Outside MPLAB® X IDE

Sending an Application's **.bl2** file to an EZBL Bootloader outside of MPLAB® X IDE will look and behave much the same as it does inside the IDE. However, instead of having the necessary commands invoked from a post-build step in a makefile, the user will need to invoke the command from a batch file or Command Prompt. The upload status will then be displayed in the console window instead of MPLAB's build Output window.

For an end user to access your Bootloader, you must minimally redistribute:

- **ezbl\_comm.exe**
- **"ex\_app\_led\_blink\_pic32mm.production.bl2"**

`ezbl_comm.exe` is a file transfer tool, allowing the .bl2 file to be copied to a communications port or the MCP2221A I<sup>2</sup>C interface under EZBL's software flow control requirements. It does not require installation or have other file dependencies. After being launched, it will close automatically upon bootloading completion or timeout. It does not contain a GUI and only accepts parameters on the command line with status information printed to a console window.

"`ex_app_led_blink_pic32mm.production.bl2`" is your Application firmware update file, generated automatically from your .elf build output. This .bl2 file is tied to your Bootloader by the `BOOTID_*` strings in your Bootloader project's makefile, so will be rejected by other EZBL Bootloader projects if an attempt is made to program it to an incorrect hardware device. It is suggested that you rename this file to include the `APPID_VER` number (or other version identification string) to minimize user confusion if multiple Application versions are released over time.

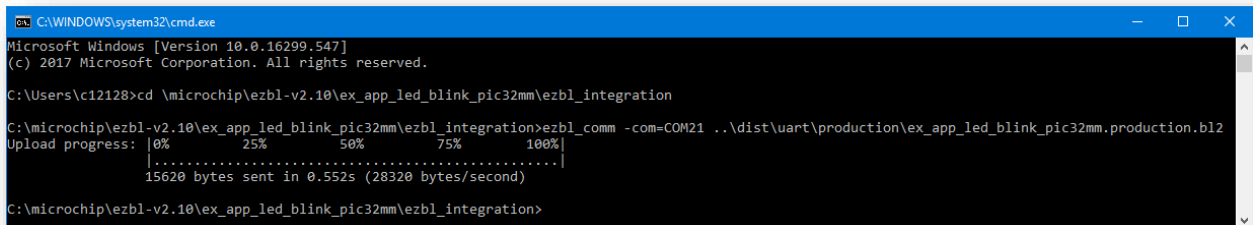
With these files in the same location, the procedure to transfer the .bl2 image to the Bootloader is:

1. Open a Command Prompt
2. Change to the directory containing both files
3. Determine the correct COM port that the Bootloader is attached on
4. Execute the upload command, such as:

```
ezbl_comm -com=COM21 -baud=115200 -timeout=1100  
ex_app_led_blink_pic32mm.production.bl2
```

This command needs to be inputted on a single line.

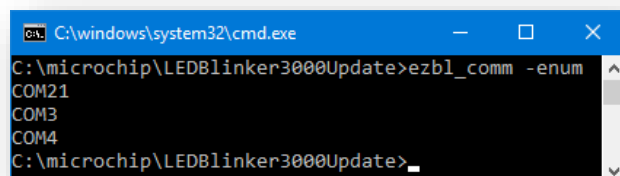
Grey parameters are tool defaults and can be omitted if no changes are needed. Highlighted options must be specified and generally will be different from the examples shown.



```
C:\WINDOWS\system32\cmd.exe  
Microsoft Windows [Version 10.0.16299.547]  
(c) 2017 Microsoft Corporation. All rights reserved.  
C:\Users\c12128>cd \microchip\ezbl-v2.10\ex_app_led_blink_pic32mm\ezbl_integration  
C:\microchip\ezbl-v2.10\ex_app_led_blink_pic32mm\ezbl_integration>ezbl_comm -com=COM21 ..\dist\uart\production\ex_app_led_blink_pic32mm.production.bl2  
Upload progress: |0% 25% 50% 75% 100%|  
|.....|  
15620 bytes sent in 0.552s (28320 bytes/second)  
C:\microchip\ezbl-v2.10\ex_app_led_blink_pic32mm\ezbl_integration>
```

To aid step 3, the following command can be used to display the COM ports present on the PC:

```
ezbl_comm -enum
```



```
C:\windows\system32\cmd.exe  
C:\microchip\LEDBlinker3000Update>ezbl_comm -enum  
COM21  
COM3  
COM4  
C:\microchip\LEDBlinker3000Update>
```

To reduce the need to type commands, a batch file can be created and distributed to specify the parameters (except for the COM port) as they will generally be constants for your Bootloader and firmware release, as shown on the following page. Batch files can be executed by double clicking on them within Windows Explorer.

For more advanced products, a GUI application should be developed and released for handling your firmware update releases. See [help\EZBL Communications Protocol.pdf](#).

## Update LED Blinker 3000.bat

```

@echo off

setlocal
set USER_COM=
echo Please specify the communications port that LED Blinker 3000
echo is connected on. It will be updated to firmware v3001.
ezbl_comm.exe -enum
echo.
set /p USER_COM=Enter nothing to abort:
if "%USER_COM%"==" " goto UserAbort
ezbl_comm.exe -com=%USER_COM% -baud=230400 -timeout=1100 ^
-log="update_log.txt"
-artifact="ex_app_led_blink.production.bl2"

goto End

:UserAbort
echo Firmware update aborted.

:End
pause
@echo on

```

A copy of this script for modification can be found in `ex_app_led_blink_pic32mm\ezbl_integration\Update LED Blinker 3000.bat`

When executed from Windows, the script will generate a dialog and send the .bl2 firmware image to the specified the communications port:

```

C:\windows\system32\cmd.exe
Please specify the communications port that LED Blinker 3000
is connected on. It will be updated to firmware v3001.
COM21
COM3
COM4
Enter nothing to abort: com21
Upload progress: |0% 25% 50% 75% 100|
|.....|
10900 bytes sent in 1.147s (9503 bytes/second)
Press any key to continue . . .

```

## *ex\_boot\_uart\_pic32mm MPLAB® X Project*

This example Bootloader project for 32-bit PIC32MM devices implements a 2-wire UART interface for typical, single partition bootloading from a PC or other device capable of writing a file to a serial interface under a software flow control mechanism.

The [ex\\_app\\_led\\_blink\\_pic32mm](#) example Application project targets this Bootloader when the 'uart' Build Configuration is selected.

The communications protocol implemented on the wire is documented in [help\EZBL Communications Protocol.pdf](#). It is identical between 16 and 32-bit implementations. The protocol assumes "reliable", in order transmission of stream-oriented data (i.e. no obvious start, end, or block size above the minimum of 8-data bits per atomic unit). The maximum round trip latency of the medium needs to be known at design time, but when set appropriately, the serial protocol can normally be tunneled through Bluetooth SPP (Serial Port Profile) radios and other bridges.

"Reliable" in this context means the firmware expects no bit errors or lost data in either the TX or RX directions throughout a complete firmware update event. However, any bit errors or lost data that may occur will still be detected and can be recovered by restarting the complete firmware update sequence.

Signaling is assumed to be full-duplex, point-to-point. Data transmitted to the Bootloader exactly matches the .bl2 file contents consumed by the Bootloader. Data return to the host from the Bootloader is limited to software flow control signaling and a final termination/status code. If the underlying physical layer implements carrier sense (i.e. local transmitter knows if the medium is idle and blocks when the receiver is active), then this project can also be used with a half-duplex, point-to-point communications link. A half-duplex, multi-point to multi-point broadcast bus may also work if all non-participating nodes remain silent during the firmware transfer (or a virtual point-to-point channel can be set up via a transparent sideband mechanism, like a 9<sup>th</sup> data bit flagging bootloader traffic differently from ordinary bus traffic).

This project implements two interrupt handlers:

- UART RX at IPL2
- UART TX at IPL1

The exact UART instance used by the Bootloader is board/processor specific and selectable in a hardware initialization file. Default hardware initializer files for PIC32MM0064GPL036 Family devices will implement UART2, while PIC32MM0256GPM064 Family devices will implement UART1. Additionally, if more than one UART peripheral is initialized, the Bootloader will accept firmware updates from multiple interfaces.

Timing operations are performed using the 32-bit MIPS Core Timer in a polling-only mode. ezbl\_lib32mm APIs software extend this timer to 64-bits for use in other projects.

The Bootloader's interrupt handlers, by default, will stay enabled when the Application is launched, but the vectors are not hard-clobbered. See [32-bit, PIC32MM Interrupt Handling](#) if the Application wishes to control these.

### **When Built**

1. Outputs a Bootloader .hex file, [ex\\_boot\\_uart\\_pic32mm.production.hex](#), for use with a traditional ICSP based programmers (PICKit, ICD, REAL ICE, etc.)
2. Generates [ezbl\\_integration/ex\\_boot\\_uart\\_pic32mm.merge.ld](#) and [ex\\_boot\\_uart\\_pic32mm.merge.S](#) files. These files are meant to be included in any Application project that needs to be programmed through the Bootloader. These files specify all flash address and data contents for the Bootloader, as well as all public symbols. i.e. addresses of global variables and functions in the Bootloader.
3. Copies both the [ex\\_boot\\_uart\\_pic32mm.merge.ld](#) and [ex\\_boot\\_uart\\_pic32mm.merge.S](#) files to the [ex\\_app\\_led\\_blink\\_pic32mm/ezbl\\_integration](#) folder for development and testing in the [ex\\_app\\_led\\_blink\\_pic32mm](#) project.

## Supports

1. All PIC32MM products
2. Out-of-box demo ability on Explorer 16/32 development board (or Explorer 16 with external USB to RS232 converter)
3. Out-of-box demo ability on the PIC32MM0064GPL036 and PIC32MM0256GPM064 PIMs.
4. Out-of-box demo ability on Microsoft Windows
5. Tested with MPALB® X IDE v4.20 and v5.00
6. Tested with MPLAB XC32 compiler v2.10

## Notes

1. **Important Files** contains a customized makefile, `ezbl_boot.mk`. Additionally, `Makefile` has been trivially modified at the bottom to include `ezbl_integration/ezbl_boot.mk`. The added make script alters the project building behavior by executing a pre-build step launching `ezbl_tools.jar` and a post-build step that copies the `ex_boot_uart_pic32mm.merge.ld` and `ex_boot_uart_pic32mm.merge.S` build artifacts to other folders.
2. **Linker Files** contains a customized linker script, `ezbl_pic32mm.ld`. The `ezbl_tools.jar` utility modifies the linker script according to the content compiled or linked into your project. No manual edits to this file should be required, regardless of which PIC32MM target device you are using.
3. **Libraries** contains the `ezbl_lib32mm.a` precompiled archive library. This file houses the object code for the API definitions in `ezbl.h` and is necessary to successfully link the project.
4. By default the UART operates in auto-baud mode so the host can choose the communications baud rate. This also allows the Bootloader to work with clock sources that have an unknown frequency or wide tolerance specification. If auto-baud is not desirable, such as when tunneling through a radio that requires a fixed baud rate, define a fixed baud rate using the `EZBL_COMBaud` constant at the top of your hardware initialization file. At run time, the baud rate can be changed (or auto-baud enabled/disabled) by calling `EZBL_FIFOSetBaud()`.
5. This project uses the following Project Properties which may differ from MPLAB defaults:

| Category              | Sub Category               | Value                                                                    |
|-----------------------|----------------------------|--------------------------------------------------------------------------|
| Loading               |                            | Load symbols when programming or building for production (slows process) |
| XC32 (Global Options) | Global options             | Use GP relative addressing threshold = 8192                              |
| xc32-gcc              | General                    | Isolate each function in a section                                       |
|                       |                            | Additional options: -mbranch-cost=1                                      |
|                       | Optimizations              | Optimization level = 1                                                   |
|                       | Preprocessing and messages | Additional warnings                                                      |
| xc32-ld               | General                    | Minimum stack size (bytes) = 3072                                        |
|                       |                            | Remove unused sections                                                   |
|                       | Libraries                  | Optimization level of Standard Libraries = 1                             |
|                       | Diagnostics                | Display memory usage                                                     |

None of these project changes are critical to correct Bootloader compilation or operation. They may be changed or removed.

Symbol loading benefits development as the "Execution Memory" Target Memory View window in MPLAB X IDE will not be populated in the absence of this option, nor would disassembly listings be available.

Assigning a large value to the GP (Global Pointer) relative addressing threshold is done to ensure efficient access to RAM variables without reloading their address unnecessarily. Since existing PIC32MM products implement 32KB or less of RAM, the Global Pointer's +/-32KB immediate offset range can always access any RAM address regardless of the linker's chosen assignment for the Global Pointer. This option is unlikely to have any effect whatsoever in the example project, but its presence acts as a hint to the toolchain that any new variables added to the project can be reached via the GP register.

Isolation of functions into their own sections permits the "Remove unused sections" linker option to delete unreferenced/dead code, typically saving space. In rare circumstances, this might increase code size, however, as fewer inlining optimizations can be performed.

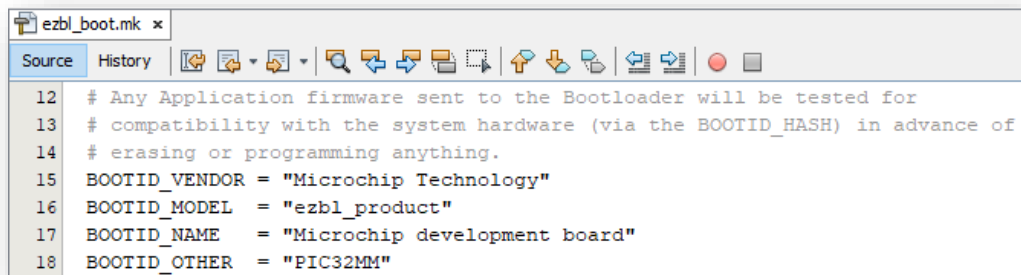
-mbranch-cost=1 is an optimization hint aimed at helping the compiler generate faster and/or smaller code with low risk of adversely affecting the other metric. PIC32MM devices have 0 wait-state flash access, so with only a 1 cycle execution penalty of using a branch, the compiler should aggressively branch places and reuse existing instructions rather than duplicate short instruction sequences in separate code flows.

A 3KB minimum stack size ensures the project fails to link if too many variables or overly fragmented variables are statically allocated in the Bootloader to maintain at least 3KB free for the stack. Since the Bootloader may need to have a 2048 byte flash page temporarily on the stack during an erase-modify-write operation, a relatively large minimum was selected.

Remove unused sections deletes unused code and variables so they don't waste flash and RAM. This setting can interfere with debugging when the PC points to addresses that don't have source code available in the project. The IDE may instead present some other (untrue) piece of code, so this option should be conditionally deselected during debug, but is recommended to ensure minimum bootloader size.

### Example Usage

1. Select the desired target PIC32MM device in the MPLAB X **Project Properties**
  - a) Also set the desired Hardware Tool and Compiler Toolchain
2. Open the **ezbl\_boot.mk** file under **Important Files** in the MPLAB Projects tree-view.
  - a) Change the `BOOTID_VENDOR`, `BOOTID_MODEL`, `BOOTID_NAME` and/or `BOOTID_OTHER` strings to be unique for your hardware product that will be running this Bootloader. The exact strings chosen are unimportant, with the field names chosen solely to suggest content that will generate a globally unique identification hash. The hash prevents your Bootloader from accepting Application firmware uploads that don't match your hardware or Bootloader version.



```

12 # Any Application firmware sent to the Bootloader will be tested for
13 # compatibility with the system hardware (via the BOOTID_HASH) in advance of
14 # erasing or programming anything.
15 BOOTID_VENDOR = "Microchip Technology"
16 BOOTID_MODEL  = "ezbl_product"
17 BOOTID_NAME   = "Microchip development board"
18 BOOTID_OTHER  = "PIC32MM"
  
```

- b) Update the `appMergeDestFolders` folder list if you are targeting some other Application project(s) instead of (or in addition to) the `ex_app_led_blink_pic32mm` example Application. `$(thisMakefileDir)` appears in this list for the purpose of preserving the

`ex_boot_uart_pic32mm.merge.ld` and `ex_boot_uart_pic32mm.merge.S` build output artifacts when a Clean or Clean and Build command is executed and serves as a fixed collection folder that is independent of production vs debug builds.

3. If you are targeting custom hardware or a Microchip development board/PIM that doesn't have a matching `hardware_initializer` file for it, copy an existing file that most closely matches your target device and edit it accordingly. The file (or some combination of files in your project) needs to:
  - Define any device Configuration words that you want in your Bootloader. These will be static and not modifiable when developing/programming new Application projects.
  - Declare the global `*EZBL_COMBootIF` pointer. This is referenced in `main.c` for handling auto-baud and communications idle/Application launch functionality. If you are using a fixed baud rate, you may delete this global variable along with the references to it in `main.c` and amend the code determining when it is appropriate to launch the Application.
  - Implement the `InitializeBoard()` function that sets the device clock (if needed), calls `NOW_Reset()`, initializes PPS and analog select SFRs applicable to your UART TX and RX pins calls `UART_Reset()`, and sets the global interrupt enable. The parameters to `NOW_Reset()` and `UART_Reset()` select which peripheral instance will be initialized for the Bootloader and indirectly which ISRs your Bootloader contains.
  - Initialize GPIO SFRs for LED output(s) and call `EZBL_DefineLEDMap()`. If you don't have or want any LED(s) toggling, you may instead delete the `LEDToggle()` and `LEDOff()` calls in `main.c`.

Other initialization code in the provided hardware initializer example files is not needed. Such code can be removed but retaining it might be useful for debug/development purposes.

Hardware initializer files that you aren't using/planning to use can be removed from the project as they are mutually exclusive to each other and contribute to project build duration.

4. With power applied and your ICSP programming tool connected, **Make and Program**
5. If your hardware implements an LED and was initialized appropriately, you will observe one LED blinking very rapidly at **8 Hz** (toggling every 62.5ms). This indicates that the Bootloader is executing and waiting for an Application to be uploaded. If the LED toggles much slower or not at all, check your Config words and any oscillator/PLL initialization code. The frequency passed to `NOW_Reset()` should match your actual execution clock.

Slow (ex: FRC without PLL) execution normally still permits Bootloading but may limit the baud rates available. Additionally, the typical 1 second timeout before launching a valid Application will increase if programming an Application doesn't resolve the actual clock <--> `NOW_Reset()` frequency mismatch.

6. You are now ready to test and use your Bootloader. Continue with the [ex\\_app\\_led\\_blink\\_pic32mm](#) Application project.

If you might be going to production with the Bootloader you just built, be sure to archive all of your Bootloader project files (i.e. `ex_boot_uart_pic32mm` project folder, including the `dist` folder contents), the `ezbl_lib32mm`, `ezbl_tools`, `ezbl_comm`, `help` folders, and the XC32 + MPLAB X IDE installers that you used for this Bootloader compilation. Don't forget an XC32 part support update installer if you installed one. If you are updating an existing archive, make absolutely certain you capture the `[bootloader_proj_name].merge.ld` and `[bootloader_proj_name].merge.S` build artifacts in your backup as these are required in order to build Application projects that are compatible with your Bootloader.

All of these files are important to have backed up because they contributed to or are important towards using your Bootloader. If an unforeseen problem is encountered after releasing parts into the field, having the exact source files and tools used can expedite debugging, searching for possible solutions and/or creating an Application project to patch the Bootloader.





## *ezbl\_tools* NetBeans Java Project

*ezbl\_tools* is a PC Java application that handles everything needed to build a valid Bootloader for an arbitrary target device and then convert Application build artifacts to a “.bl2” format suitable for serial transmission and Bootloader programming. This java application, through invocation via a project’s **Makefile** (or more specifically, **ezbl\_boot.mk/ezbl\_app.mk**), triggers several build-time actions, including:

- Modifies other MPLAB® X IDE generated makefiles to reinvoke **ezbl\_tools.jar** at the appropriate times during project build
- Interrogates an internal device characteristics data base for flash geometry, minimum erasable page/programmable block sizes, Config word addresses, available hardware interrupt vectors and other information specific to the target processor
- Updates Bootloader **.gld/.ld** linker script with information from the compiler's default **.gld/.ld** linker script for the target processor
- Launches **xc16-objdump/xc32-objdump** and extracts symbols/section/ISR information of just compiled Bootloader **.elf** files
- Creates per-vector interrupt multiplexing code for run-time selecting Bootloader/Application ISR execution on hardware interrupts which the Bootloader implements an ISR for
- Creates flash data tables to make the Bootloader "self-aware." These data tables include:
  - Application and Bootloader flash geometry to suppress erase/programming of Bootloader occupied flash locations
  - Location of flash addresses with special hardware behaviors and requiring special erase or verification restraints
  - Backup/erase-restore values for Bootloader defined Config word contents
- Creates Application Interrupt Goto Table (IGT) for remapping interrupts to linker assigned Application ISR addresses (16-bit devices), or modified IVT entries that load the ISR target address from a RAM pointer and branch to the pointer’s target (32-bit devices)
- Creates **[bootloader\_project\_name].merge.S** assembly source file and **[bootloader\_project\_name].merge.gld/ld** linker script to encapsulate a pre-compiled/binary Bootloader image and API addresses within Application projects.
- Converts **.elf** toolchain outputs to a **.bl2** file for upload to the Bootloader. Data record reordering/coalescing, address alignment and padding, SHA-256 hashing and CRC32 generation for **.bl2** contents are implemented to minimize the run-time work in Bootloader projects and improve overall robustness as compared to direct use of **.hex** files.
- [Legacy] Acts as a communications bridge to launch **ezbl\_comm.exe** in order to transfer Application **.bl2** files to an EZBL bootloader immediately upon Application build completion.

This executable does not present a GUI or implement any graphical elements. It is intended to be executed from makefile and batch/shell scripts, directly via a console command line or as a sub process within a different, company branded GUI or user front end.

### When Built

1. Generates **ezbl\_tools.jar** executable

### Supports

1. NetBeans IDE 8.2
2. Java JDK v1.7 (i.e. Java SE 7) and newer

### Notes

1. Changes to this project generally should be avoided as it is maintained by Microchip and subject to change in future EZBL distributions. Altering the tool significantly will make future upgrades more difficult or impractical.

2. This project source and structure files are fully included for reference and debugging purposes.

## *ezbl\_lib* MPLAB® X Project

This is a 16-bit PIC24/dsPIC source library project designed for portable, reusable, pre-compiled functions that a Bootloader or Application alike will regularly need. For use on 32-bit/PIC32MM/XC32 projects, see [ezbl\\_lib32mm](#) instead. Both library projects implement flash Run Time Self Programming APIs, blank checking, CRC32 calculation, software communications FIFOs and UART/I<sup>2</sup>C peripheral drivers, Timer/CCP Timer peripheral drivers, and other functions that are suitable for abstraction and use on arbitrary device targets without changes.

The files in this project are compiled at the -Os optimization level, omitting frame pointers (when possible) and with large memory models for data, code and scalars. This combination trades off debug ability in favor of smallest code size and affords greatest compatibility across differing target RAM/flash geometries and compiler optimization settings applicable in Bootloader and Application projects that link against *ezbl\_lib.a*.

Archive source files written in assembly language conform to XC16 C calling conventions and similarly assume large data/code models to maintain maximum portability across the entire Microchip 16-bit product portfolio of PIC24 MCU and dsPIC DSCs.

### When Built

1. Outputs *ezbl\_lib.a* archive library, suitable for inclusion in both Bootloader and Application projects. To see and use the archived APIs, `#include "ezbl.h"` first.

### Supports

1. All 16-bit products. However, flash erase/programming, communications, and timer APIs are not supported on dsPIC30F and PIC24FxxKxxx devices. PIC24FJ, PIC24H, PIC24E, dsPIC33F, dsPIC33E, and dsPIC33C product lines can link to and use any of the exported APIs.
2. The EZBL v2.11 version of this archive was compiled using XC16 v1.35 for all build configurations. Any XC16 compiler version can be used when linking against *ezbl\_lib.a* APIs.

### Notes

1. Changes to this project should be avoided as it is maintained by Microchip and subject to change in future EZBL distributions. Independent changes could make future versions require greater effort to migrate to.
  - a) If it is desirable to change code within *ezbl\_lib*, instead copy the applicable source files from the *ezbl\_lib/sectioned\_functions* or *ezbl\_lib/weak\_defaults* folder and place them as local files within your Bootloader and/or Application project(s). These local copies can then be edited or used for debugging under lesser compiler optimizations without affecting the archive. The linker will resolve references to the local API code preferentially over searching for the archived base versions while still permitting linkage to other components of *ezbl\_lib.a* which you have not chosen to copy locally.
  - b) Generally, it is a good idea to include *ezbl.h* and *ezbl\_lib.a* in all Bootloader and Application projects as the API set provided can be useful in both places but contributes no flash or RAM usage to the overall project if nothing in the archive is actually referenced.
2. Current versions of MPLAB X IDE do not support building the same source file simultaneously for multiple different generic processor types implemented by XC16. To generate a single archive containing binary implementations for each of the generic processor types/ISA variants, this project implements a highly customized Makefile that can iterate over the project's various build configurations and place the generated objects into a single archive. If it is necessary to rebuild the entire archive, special procedures must be followed:
  - a) Select each Build Configuration in sequence. MPLAB will generate/update the configuration-specific makefiles for your local toolchain paths and IDE settings when each configuration is selected. It is not necessary to Build these configurations after selection
  - b) Select the Build All configuration
  - c) Issue a Clean and Build command. This will delete the existing *ezbl\_lib.a* archive and iterate over each of the generic processor configurations, adding all versions to a new *ezbl\_lib.a* archive.

- d) If any changes to the build configuration properties or file presence/absence in the project changes, it will be necessary to repeat steps a), b) and c) to correctly regenerate the archive. Repeating these steps is particularly important when files are added or removed from the project as the archive contents is only updated each time the Build All configuration is built, with no obsolete file removal or addition of new files for anything but the currently selected build Configuration.

## ***ezbl\_lib32mm* MPLAB® X Project**

This project builds the source files and creates the ***ezbl\_lib32mm.a*** precompiled archive library. This library has the same intended uses as the ***ezbl\_lib*** project, but this variant supports all 32-bit PIC32MM devices using the XC32 toolchain instead of PIC24/dsPIC devices using the XC16 toolchain. The two projects are segregated from each other to avoid potential errors during linking that would otherwise occur if one archive contained objects of the same name, but with incompatible architecture/ISA signatures.

Many of the APIs present in ***ezbl\_lib*** also exist in ***ezbl\_lib32mm*** and sometimes share an exact copy of the source code implementation. However, the two libraries require different source code in other cases due to differences between the XC16 and XC32 toolchains, as well as differences in the underlying CPU hardware, instruction set, and operational requirements.

### **When Built**

1. Outputs ***ezbl\_lib32mm.a*** archive library, suitable for inclusion in both Bootloader and Application projects. To see and use the archived APIs, `#include "ezbl.h"` first.

### **Supports**

1. All 32-bit PIC32MM products, including the PIC32MM0064GPL036 Family and PIC32MM0256GPM064 Family of devices.
2. The EZBL v2.11 version of this archive was compiled using XC32 v2.10. Any XC32 toolchain version can be used when linking against ***ezbl\_lib32mm.a*** APIs.

### **Notes**

3. Changes to this project should be avoided as it is maintained by Microchip and subject to change in future EZBL distributions. Independent changes could make future versions require greater effort to migrate to.
  - a) If it is desirable to change code within ***ezbl\_lib32mm***, instead copy the applicable source files from the ***ezbl\_lib32mm/functions*** or ***ezbl\_lib32mm/weak\_defaults*** folder and place them as local files within your Bootloader and/or Application project(s). These local copies can then be edited or used for debugging under lesser compiler optimizations without affecting the archive. The linker will resolve references to the local API code preferentially over searching for the archived base versions while still permitting linkage to other components of ***ezbl\_lib32mm.a*** which you have not chosen to copy locally.

Generally, it is a good idea to include ***ezbl\_lib32mm.a*** and ***ezbl.h*** in all Bootloader and Application projects as the API set provided can be useful in both places but contributes no flash or RAM usage to the overall project if nothing in the archive is referenced. When something is referenced, only the item and its dependencies, if any, contribute to the project resource footprint.

## ***ezbl\_comm C Project (gcc or Visual C++ 2008 Express Edition)***

This is a console executable intended for transferring Application .bl2 firmware images on a PC to an EZBL Bootloader project via UART or I<sup>2</sup>C (using the MCP2221A HID I<sup>2</sup>C Master interface).

Communications via serial ports on a PC uses OS-specific APIs to open a handle to the COM port and then C stdio calls to read and write to the file handle. OS-specific code exists for both Windows and Linux style OSes, with the Windows version being pre-compiled and supplied in the EZBL distribution.

Communications via the MCP2221A I<sup>2</sup>C interface use MCP2221A driver APIs statically linked inside **ezbl\_comm.exe**. This interface type can only be used on the Windows OS.

The code in this project implements the EZBL communications protocol, documented by [help\EZBL Communications Protocol.pdf](#).

### **Building on Windows**

To rebuild this project, you will need:

- Microsoft Visual C++ compiler (Visual C++ 2008 Express Edition IDE was used to create the pre-compiled **ezbl\_comm.exe** copy)
- Microsoft Windows SDK

The Visual Studio project is configured to generate an x86 (32-bit) executable and perform static linking for the Visual Studio C Run-Time library and MCP2221A APIs, so the resulting **ezbl\_comm.exe** executable has no external .dll/library dependencies and does not need any type of installation on the machine that will execute it. I.e. it is a single file, “portable” executable and should execute directly on a USB thumb or read-only file system.

### **Building on other OSes**

This project has not been tested on OSes other than Windows. However, code to operate on Linux-like OSes has been included and tested under Cygwin64. To rebuild this project, you will need:

- gcc

Building the executable entails simply executing ‘make’ in the folder containing **Makefile**.

## Communications Status Codes and Error Messages

Several things could go wrong when bootloading. Although EZBL is unlikely to brick itself, there are multiple messages that you may see when using the `ezbl_comm` host application to upload firmware to a Bootloader. The most authoritative source of information regarding a message will always be the source code itself; however, this section enumerates the error messages you may see and their likely cause/solution.

For all but the `EZBL_ERROR_SUCCESS` case, the hex error number in the title and the same error number in parenthesis as a decimal number, is the value `#defined` in the Bootloader to match the error string. This 16-bit value is also the bootload termination status code observable on the communications medium, and, sign extended to an (int), the return code when the `ezbl_comm` process terminates itself.

All error codes implemented in EZBL are 16-bit signed integers (regardless of the target's native `int` width) and transmitted least significant byte first. Positive error codes generally indicate something favorable occurred while negative valued codes indicate something went wrong.

```
0x0001: EZBL_ERROR_SUCCESS (1)
    "Operation completed successfully"
0x0002: EZBL_ERROR_SUCCESS_VER_GAP (2)
    "Operation completed successfully, but the programmed image did not have an
    APPID_VER_MAJOR.APPID_VER_MINOR field that was +1 (minor code) from the existing application."
0x0003: EZBL_ERROR_ALREADY_INSTALLED (3)
    "Offered firmware image already matches the existing target firmware"
0xFFEC: EZBL_ERROR_COM_READ_TIMEOUT (-20)
    "Bootloader signaled communications timeout while waiting for image data"
0xFFEB: EZBL_ERROR_IMAGE_MALFORMED (-21)
    "Bootloader rejected firmware as malformed or of unsupported type. Possible communications error."
0xFFEA: EZBL_ERROR_BOOTID_HASH_MISMATCH (-22)
    "Bootloader rejected firmware as incompatible"
0xFFE9: EZBL_ERROR_APPID_VER_MISMATCH (-23)
    "Bootloader rejected firmware as out of the required programming order"
0xFFE8: EZBL_ERROR_HARD_VERIFY_ERROR (-24)
    "Bootloader read-back verification failure"
0xFFE7: EZBL_ERROR_SOFT_VERIFY_ERROR (-25)
    "Bootloader read-back verification mismatch in reserved address range"
0xFFE6: EZBL_ERROR_IMAGE_CRC (-26)
    "Bootloader computed CRC mismatch with CRC contained in firmware image. Probable communications
    error."
0xFFE5: EZBL_ERROR_IMAGE_REJECTED (-27)
    "Bootloader or running application rejected the offered image"
0xFFE4: EZBL_ERROR_CUSTOM_MESSAGE (-28)
    Custom bootloader response
0xFC13: EZBL_ERROR_LOCAL_COM_RD_TIMEOUT (-1005)
    "Timeout reading from 'com_port_path'"

```

### 0x0001: EZBL\_ERROR\_SUCCESS (1)

**"Operation completed successfully"**

This is not actually an error, but rather, a status code indicating that all bootloading operations completed normally and verified successfully. The new firmware will normally begin executing immediately, or after the `BOOTLOADER_TIMEOUT` delay following a device reset.

NOTE: 0x0001 is the code sent by the Bootloader, but `ezbl_comm` will translate this value to 0 when terminating and using it as a process return code. This translation is done to conform to the typical `main()` return on PC applications indicating success.

#### 0x0002: EZBL\_ERROR\_SUCCESS\_VER\_GAP (2)

**“Operation completed successfully, but the programmed image did not have an APPID\_VER\_MAJOR.APPID\_VER\_MINOR field that was +1 (minor code) from the existing application.”**

This error is only emitted for Dual Partition, Live Update projects, such as *ex\_app\_live\_update\_smps\_v1*, *ex\_app\_live\_update\_smps\_v2* and *ex\_app\_live\_update\_smps\_v3*. It indicates the firmware was programmed to the Inactive Partition and verified successfully, but that the Application version does not follow a natural progression of +1 minor version code relative to the currently executing Application’s version. For example, if the existing Application has APPID\_VER set to 1.0.xxxx, the Bootloader will be expecting the new firmware to have APPID\_VER set to 1.1.xxxx. Any other value received, such as 1.4.xxxx will trigger this error code instead of EZBL\_ERROR\_SUCCESS.

EZBL\_ERROR\_SUCCESS\_VER\_GAP is used to notify the human operator that the device will begin executing the new firmware at device reset rather than attempting to perform a live partition swap into the new Application without issuing a reset. The Dual Partition, Live Update project examples differentiate between the sequential progression of version numbers because successfully changing the Application code during operation requires all preserved system state to be allocated to matching RAM memory addresses and have predictable values that the firmware developer can assume as existing. Jumping backwards or more than one version ahead will quite trivially devolve to system states that violate the developer’s assumptions, leading to erroneous execution, thus necessitating a full system restart in order to safely begin execution of the new Application.

#### 0x0003: EZBL\_ERROR\_ALREADY\_INSTALLED (3)

**“Offered firmware image already matches the existing target firmware”**

This error occurs immediately upon attempting to send the same Application firmware image to a Bootloader that already has a valid Application programmed, which has an exactly matching APPID\_VER code (including build number), and which doesn’t wish to permit unnecessary erase and reprogramming of the same code.

Presently, only the *ezbl\_uart\_dual\_partition.c* and *EZBL\_InstallFILEIO2Flash.c* source files generate this error code, so the *ex\_app\_live\_update\_smps\_vx* and *ex\_boot\_usb\_msd* Bootloader example projects can generate this error while other Bootloaders like *ex\_boot\_uart* and *ex\_boot\_i2c* will follow out the erase/programming sequence as normal without triggering this error.

Live Updating a project to a copy of itself is disallowed as conditional reinitialization of ‘update’ attributed and priority code will execute immediately after partition swap, generally leading to corrupt device state if allowed to occur.

USB mass storage bootloaders generate this error response since the FIRMWARE.BL2 file stays on the mass storage media after bootloading and can remain inserted at subsequent device reset events. Erasing and reprogramming the same firmware wouldn’t typically harm any system state, but would lead to unnecessary flash endurance being wasted. It would also cause excessively long power up or reset start up sequences. This Bootloader type therefore pre-reads the FIRMWARE.BL2 header contents at reset and issues an early abort if the firmware available is the same firmware already in flash and ready for execution.

#### 0xFFEC: EZBL\_ERROR\_COM\_READ\_TIMEOUT (-20)

**“Bootloader signaled communications timeout while waiting for image data”**

This error indicates that the Bootloader has accepted the image, blank checked/erased the existing application flash space, started programming zero or more data records to flash, but then subsequently stopped receiving any data from the communications medium for an extended period. This error is generated when the Bootloader thinks there is more data required to reach the End of File (EOF) stream position, but with the host communications partner typically having already reached EOF. In this state, the



Bootloader knows the Application has been truncated or corrupt, so it will prohibit execution of the partial Application written prior to the timeout. Recovery will require restarting the whole bootloading procedure.

This error has several possible causes:

1. Communications baud rate is too high. Many modern USB to UART converters support UART baud rates of 921600 bits/sec or higher, and while the PIC/dsPIC UART may be able to successfully match the baud rate, attempting to use such high data rates can fail due to UART hardware RX FIFO overflow.

For example, PIC24F/PIC24E and dsPIC33F/E devices all implement a UART hardware RX FIFO size of 4 characters + 1 one still being sampled from the wire. At 921600 baud, each 8-bit data character received requires only 10.85us. Even with software flow control, the hardware will still overflow and lose data if a minimal flash programming cycle blocks CPU execution and all interrupt processing for more than about 40us. When data is lost in this manner, the host node providing the .bl2 image file will reach EOF and stop transmitting while the EZBL bootloader's state machine will still be looking for more data that got discarded in hardware earlier in the file transfer processes.

2. Communications medium unreliable. For the same reason an excessive baud rate can trigger this error, a momentary EMI/ESD glitch or dropped Bluetooth packet corrupting the data stream can desynchronize the host and bootloader nodes.
3. BOOTLOADER\_TIMEOUT set too short or the CPU clock is running faster than the NOW\_Reset() call specified.
4. Bug in the Bootloader project. As the bootloader implements RX interrupts to move incoming data into a larger software RX FIFO, any code in the Bootloader project which disables interrupts for an extended period of time will subject the hardware to the same RX overflow condition that an excessive baud rate will. Likewise, any code in the Bootloader project which triggers extended interrupt handling at a higher priority than the RX interrupt is configured for can block the movement of data out of the overflowable hardware FIFO and into the flow-controlled software FIFO. Finally, a bug that corrupts RAM state (ex: writing to pointer or array indexes beyond the intended boundaries) can overwrite the Bootloader's software FIFO and state tracking information with garbage. None of the EZBL library code or example projects are known to have any such bugs in them, but because bootloading operations can occur in the background while still processing the Bootloader's main() while(1) loop and interrupts, it is possible to combine conflicting code with a concurrent bootloading session.
5. (Unlikely) .bl2 image data is corrupt, such as by unintentional file truncation, or user attempt to modify its contents without updating the appropriate header fields. `ezbl_comm` does not attempt to decode or generate anything within the .bl2 image, so relies on the host OS's filesystem level EOF indicator to determine when it should stop transmitting data. The bootloader instead must rely on the file length stored within the .bl2 file header to know where EOF is expected in the stream.
6. [USB Host Mass Storage bootloaders only] A FILEIO\_Seek() or FILEIO\_Read() call returned an error. The storage media or filesystem structure may be damaged, someone may have unplugged the media, the media may have consumed too much power from your circuit and caused a brownout for itself, or some other problem occurred in the USB stack or FILEIO library.

**0xFFEB: EZBL\_ERROR\_IMAGE\_MALFORMED (-21)**

**"Bootloader rejected firmware as malformed or of unsupported type. Possible communications error."**

This message is typically generated after the system has decided to accept the image, completed the erase phase and is starting or has been programming data for a while. If this is true, the error is triggered by parsing of a record header containing a record length above 0x00FFFFFF, or for 16-bit bootloaders only, alternatively a destination address above 0x00FFFFFF. Such large records or addresses are not reasonable for any of the devices EZBL Bootloaders can be executed on, so finding them in the data stream indicates that RX communications data has been corrupted.

This message is also generated if the first 64 bytes of the .bl2 file have been received, has a valid BOOTID\_HASH value, the APPID\_VER field is acceptable, but the reserved 16-byte HMAC field in the .bl2 header has an unexpected value that isn't all zeros. When this happens, the existing flash contents won't be erased. Since the current version of EZBL generates .bl2 files with this field set to all zeros and requires a match in the Bootloader, this error also corresponds to RX communications data getting corrupted.

The conditions that can cause RX data corruption match those specified in the EZBL\_ERROR\_COM\_READ\_TIMEOUT response code, so refer to its documentation for possible things to investigate. This response code differs from a timeout only in that it can detect failure and try to signal the host to abort transferring new data earlier in the overall communications exchange, including potentially before any existing Application has to be erased.

#### 0xFFEA: EZBL\_ERROR\_BOOTID\_HASH\_MISMATCH (-22)

##### "Bootloader rejected firmware as incompatible"

This error code means the bootloader has been awoken by reception of a BOOTLOADER\_WAKE\_KEY string located in the RX interrupt or initial FIRMWARE.BL2 file read parsing, but the value of the BOOTID\_HASH field in the .bl2 file header mismatches the BOOTID\_HASH compiled into the Bootloader project. In other words, a valid EZBL image is being offered, but the image is intended for some other product and not our bootloader. This state results in the data being silently thrown away from the RX FIFO up to EOF or until a passive idle timeout occurs to return the bootloader state machine back to a sleeping state.

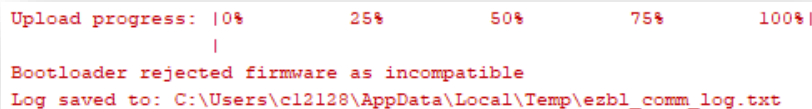
This error is not transmitted automatically back to the `ezbl_comm` host process. However, it is still generated internally and passed back to the code that calls `EZBL_Install2Flash()/EZBL_Install2IP()` via the `EZBL_INSTALL_CTX.bootCode` parameter, or directly as the return code from `EZBL_InstallFILEIO2Flash()`.

The example projects in the EZBL distribution do not transmit this response to `ezbl_comm` in order to play nicely on broadcast mediums where multiple bootloadable targets may share the same communications bus.

If your hardware enjoys a dedicated, point to point connection with the host, you may propagate this error message back to `ezbl_comm` for user display and early host termination prior to letting it timeout. Do this by adding (or uncommenting) code to poll `bootCode` after `EZBL_Install2Flash()/EZBL_Install2IP()` returns. For example:

```
EZBL_Install2Flash(&EZBL_bootCtx);
if(EZBL_bootCtx.bootCode == EZBL_ERROR_BOOTID_HASH_MISMATCH)
{
    // Propagate BOOTID_HASH mismatches to ezbl_comm
    EZBL_FIFOwrite32(EZBL_COM_TX,
        (((unsigned long)EZBL_bootCtx.bootCode)<<16) | 0x0000u);
    EZBL_bootCtx.bootCode = 0; // Don't repeat status transmission
}
```

`ezbl_comm` will correspondingly print the rejection and terminate the bootload attempt.



```
Upload progress: |0%      25%      50%      75%      100%|
                  |
Bootloader rejected firmware as incompatible
Log saved to: C:\Users\cl2128\AppData\Local\Temp\ezbl_comm_log.txt
```

#### 0xFFE9: EZBL\_ERROR\_APPID\_VER\_MISMATCH (-23)

##### **“Bootloader rejected firmware as out of the required programming order”**

This error occurs when you enable the `EZBL_NO_APP_DOWNGRADE` build time option in your Bootloader, then attempt to upload an Application .bl2 file with a numerically smaller APPID\_VER code in it than the Application that already exists in flash. Normally version numbers increment though time, so this condition indicates the user is trying to downgrade their firmware to an older version.

NOTE: The test to generate this error code is strictly a less-than test, not less-than-or-equal. It is therefore possible to use the Bootloader to reprogram the same Application image (or potentially the same chronologically generated version, but with a different payload, like a series of order independent production hardware testing projects).

NOTE 2: Do not rely on the `EZBL_NO_APP_DOWNGRADE` feature for any type of versioning enforcement. Anyone can start uploading the same or newer firmware to the device and purposely unplug the power or communications cable to leave the device without a valid Application. In this state, the Bootloader will accept any valid Application image, since there will be no existing version number to compare against.

#### 0xFFE8: EZBL\_ERROR\_HARD\_VERIFY\_ERROR (-24)

##### **“Bootloader read-back verification failure”**

This error occurs when your Application project contains flash data defined at an address that the Bootloader has not reserved for itself, which isn't recognized as having special hardware behaviors and verification requirements and which read back after being programmed as mismatched with the data defined in your Application image.

Unless you've executed a tight code loop that performed an exceedingly large number of page erase and programming cycles against the same flash address, it is very unlikely that this error will be caused by a flash cell that has reached its endurance limit. More likely cases are:

1. Communications unreliable or baud rate excessive. If you push the data rate very high, such as  $\geq 921600$  baud, the 4-deep hardware RX FIFO on a UART peripheral can overflow while the CPU is blocked for a programming operation. In this case, initial image acceptance and the first data record in the .bl2 image will begin programming normally, but then as RX bytes get dropped by hardware overflow, the next record header will be read from an incorrect location in the data stream, potentially resulting in an attempt to program an illegal memory address. Around 800kbaud should be treated as the upper limit for blocking, single partition bootloaders.
2. Hardware flash write protect has been enabled. EZBL bootloaders are designed to operate without any form of hardware write protection as it monitors all erase/programming requests in software and negates ones targeting Bootloader-reserved addresses.
3. Invalid combination of Code Protect/Code Guard settings, such as having a Boot Segment defined and assigning the highest level of protection to the General Segment.
4. Application project contains data defined at an address that the Bootloader cannot erase and/or reprogram, such as OTP addresses or elsewhere at addresses  $\geq 0x800000$ . Most such high

addresses do not have programmable memory located at them, but which will not cause an Address Error Trap (and Bootloader reset) when read-back is performed.

To debug this failure, it is recommended that you copy the `ezbl_lib\sectioned_functions\EZBL_Install2Flash.c` or `ezbl_lib32mm\functions\EZBL_Install2Flash.c` file to your local Bootloader project and execute the Bootloader in debug mode. A breakpoint set after the `EZBL_VerifyROM()` API call and failure detection can help discover which address(es) are not reading back with matching program data and what the differences were.

#### 0xFFE7: EZBL\_ERROR\_SOFT\_VERIFY\_ERROR (-25)

##### **“Bootloader read-back verification mismatch in reserved address range”**

This error occurs when the .bl2 image contains data at a Bootloader address that does not match what already exists in the physical device at the same Bootloader address. EZBL bootloaders block access to erase or program addresses reserved for itself, but they are still presented as part of the Application image and read-back verification is still performed. Applications can call and use Bootloader APIs at run time, so the contents of Bootloader flash regions must match what the Application thinks should exist in the Bootloader.

Receiving this error means:

1. You’ve modified and recompiled your Bootloader project, but forgot to program the Bootloader via an ICSP programming tool. Your board still has an old Bootloader in it when you subsequently try to use the old Bootloader to program an Application that you just recently built containing the [boot\_project].merge.[S/gld/ld] files from the new Bootloader.
2. You’ve updated your Bootloader and successfully have it programmed to your board, but are attempting to use the Bootloader to program an Application .bl2 image built some time in the past with an earlier Bootloader in it.
3. You’ve changed your Bootloader and/or Application project names, but still have old [boot\_project].merge.[S/gld/ld] files present in your Application project and corresponding to a the previous Bootloader artifacts.
4. [Rare] You’ve modified something in your [boot\_project].merge.[S/gld/ld] files by hand and now are building your Application project with no matching Bootloader.

Resolving this error usually entails correcting one of the above oversights. However, if you don’t care which Bootloader your Application has been linked against and just want to quickly test something related to your Application, one way to guarantee consistency between the projects is to use an ICSP programming tool to reprogram your board with the Application’s .hex file. This file always contains a binary copy of the Bootloader it was compiled against, so programming via ICSP will resolve the mismatch. Assuming whatever copy of the Bootloader in your Application is functional, you’ll be able to continue your work on the Application using the Bootloader to reprogram subsequent Application iterations rather than continuing to use the ICSP tool and you will no longer receive this error message.

#### 0xFFE6: EZBL\_ERROR\_IMAGE\_CRC (-26)

##### **“Bootloader computed CRC mismatch with CRC contained in firmware image. Probable communications error.”**

All .bl2 firmware image files are created with a CRC-32 stored as the last 4 bytes of the file and calculated over all other bytes of the file (starting at byte offset 16). During reception and programming by the Bootloader, the same bytes of the incoming data stream are used to recompute a CRC-32. Failure with this error code means the two checksums came out different.

Using less aggressive communications parameters, such as lowering the baud rate will likely resolve this error.

**0xFFE5: EZBL\_ERROR\_IMAGE\_REJECTED (-27)****“Bootloader or running application rejected the offered image”**

This message is displayed when you return `<= 0` in your `EZBL_PreInstall()` Bootloader callback function, or when implemented, the `EZBL_BootloadRequest()` Application callback function.

To always accept firmware offers, delete the `EZBL_PreInstall()` function or modify it to return 1 instead.

**0xFFE4: EZBL\_ERROR\_CUSTOM\_MESSAGE (-28)*****Custom bootloader response***

This error code does not have a fixed meaning or error string associated with it. It is intended to allow the Bootloader project to display an arbitrary error message of its own choosing by transmitting a null terminated string back to the host via the communications channel.

EZBL Bootloading functions and example projects do not generate this error internally. However, your Bootloader project can create it in the `EZBL_PreInstall()` callback if you’ve decided to reject the firmware offer/bootload request. Likewise, an existing Application could generate it if you are forwarding the callback to the Application project. Here is an example showing how to create this error via an executing Application:

```
int EZBL_BootloadRequest(EZBL_FIFO *rxFromRemote, EZBL_FIFO *txToRemote,
                        unsigned long fileSize, EZBL_APPID_VER *appIDVer)
{
    if(some_state_inappropriate_for_bootloading)
    {
        EZBL_FIFOWrite32(txToRemote,
                        (((unsigned long)EZBL_ERROR_CUSTOM_MESSAGE)<<16) | 0x0000u);
        EZBL_FIFOWriteStr(txToRemote,
                        "I'm busy. Try reprogramming me some other time.");
        return 0;
    }

    return 1;    // 1 = Allow erase/programming and this App dies.
}
```

If you want to get fancy, you could instead use something like `EZBL_FIFOprintf(txToRemote, "formatStr", ...)` in place of the `EZBL_FIFOWriteStr()` call.

**0xFC13: EZBL\_ERROR\_LOCAL\_COM\_RD\_TIMEOUT (-1005)****“Timeout reading from 'com\_port\_path'”**

This error indicates the host `ezbl_comm` communication process was expecting a flow control advertisement or status message, but which never received it. This is an error generated by the host process and not the Bootloader itself.

Seeing this error indicates that `ezbl_comm` did successfully open the specified COM/I2C port on the PC, and also successfully wrote at least the first 64 bytes of the .bl2 file to this communications port, but what became of the data after this, is unknown.

Unless the cause is already known, the first diagnostic step should be to reset the target device (by power cycling it, or better, issuing an MCLR reset) and then retry bootloading. Many potential failure mechanisms exist for a read timeout error, with a large percentage being permanent procedural, design or hardware errors while another large percentage are transiently generated unsupported/unsupportable device state with transient failure characteristics. Reattempting bootloading soon after device reset can potentially rule out causes in the permanent failure class.

Several causes of permanent failure (i.e. bootloading always fails) are:

1. Target not powered or communications cable not plugged in
2. Target does not have a Bootloader programmed in flash or the Bootloader contains Config words invalid for execution (ex: invalid clock)
3. TX/RX or SDA/SCL I/O pins are not configured correctly.
  - a. Check signaling with an oscilloscope or logic analyzer to confirm that TX/RX or SDA/SCL lines are both idling to the logic high state and see activity when a bootload attempt is issued from the PC.
  - b. Ensure that Peripheral Pin Select registers, when applicable, have been initialized correctly.
  - c. Ensure ANSEL/PCFG bits on the Bootloader target communications lines are set to place the proper pins in a digital I/O mode.
  - d. Ensure that no higher priority pin functions are enabled, such as PWM[H/L]x functions, PWM fault, analog comparator, OSC/SOSC/TCK/TDO/TDI/TMS with JTAG Enable Config bit == '1'/etc.
  - e. For I<sup>2</sup>C, ensure the Config words do not have the desired I2C peripheral instance assigned to an alternate SCL/SDA pin function location.
4. Target is connected to a different COM/I2C port from the one opened on the PC for bootloading
5. PC COM port not functioning correctly. Occasionally with USB to UART converters, rapid unplug/plugin cycling, manual assignment of COM number that is used by a different USB to UART converter or other driver installation problem may result in a COM port that shows as valid in the system Device Manager, but which is not actually functional. In such cases, it may be necessary to unplug the converter from the USB and target circuit, wait a couple seconds, then plug it back into the PC on a different USB port. This may cause assignment of a new COM port number during USB enumeration and otherwise resolve driver initialization problems.
6. Bootloader didn't want the .bl2 image and passively ignored it. This occurs when the BOOTID\_HASH contained in the .bl2 file does not match the BOOTID\_HASH used when the Bootloader project was compiled. To permit bootloading of multiple different targets on a broadcast oriented communications medium, mismatched BOOTID\_HASH values cause the Bootloader to silently ignore the image rather than send some kind of more specific rejection notice back to the host node or blindly accept and erase/program its flash. This means you must maintain the same BOOTID\_HASH criteria between the time you initially programmed your target chip with the Bootloader code via ICSP and when you compiled your Application project.

To minimize the chance of this mismatch, be very mindful that you must use an ICSP programming tool and reprogram the Bootloader on any boards you are working with after rebuilding your Bootloader project.
7. Application was compiled with the wrong [boot\_project].merge.S and/or [boot\_project].merge.[gld/ld] files in it. The BOOTID\_HASH encoded in the .bl2 file header is obtained from metadata present in the [boot\_project].merge.S file, so sending an Application .bl2 file to a target device running a different Bootloader will result in a passive ignoring state. The root cause of this failure is identical to case 1 above, but deserves special notice since it is caused by misconfiguration of the Application project rather than something wrong with the contents of the Bootloader installed via ICSP.

NOTE: Mismatched `.merge.[S/gld/ld]` files can readily occur if you select the wrong Build Configuration in the `ex_led_blink_app_pic32mm` Application project (or potentially your

derivatives of it) since each Build Configuration includes one pair of files while excluding several others.

NOTE 2: Mismatched `.merge.[S/gld/ld]` files can also readily occur if you copy the wrong files from your customized Bootloader project or add the wrong files to your customized Application project. This error opportunity is especially likely if you've renamed the Bootloader project as part of your customizations. To avoid this, always delete all existing `[boot_project].merge.[S/gld/ld]` files from both your Bootloader and Application project's `ezbl_integration` folders whenever you rename the Bootloader project.

8. Application .bl2 file generated from the wrong build artifact. .bl2 files must be generated from the compiler's .elf output file after building the Application, not from a .hex file. Hex files do not contain BOOTID\_HASH meta data at a fixed location, so ezbl\_tools.jar will not be able to create a correct .bl2 file header suitable for Bootloader accept/reject testing when a .hex file is passed to ezbl\_tools.jar to generate a .bl2 file.
9. Obsolete copy of a .bl2 file selected for upload or some other file altogether was chosen. `ezbl_comm.exe` does not decode the file contents or know anything about its structure, so will happily transmit a .txt file or something else if you ask it to.
10. Application .bl2 image contains illegal data in it. For example, if the .bl2 file contains a flash data record at address 0x123400, meanwhile the physical target device implements only 256KB of flash, then the Bootloader will attempt to program an unimplemented program space address. This programming operation will be ignored by hardware, but upon read-back verification, an (unhandled) Address Error Trap exception will occur and reset the device. The Bootloader will now be unaware that a prior bootload session was underway and stop sending any expected flow control advertisements back to the `ezbl_comm.exe` process.

Unlike other permanent failure mechanisms, this type of error will typically occur someplace in the middle of what appears to be successful bootloader erase and programming operations, (normally) not at the very beginning of the .bl2 file upload attempt. However, this error will still take place with 100% failure repeatability and fail each time at the same point in the file transfer.

Several causes of transient/state related failure are:

11. Target UART baud rate mismatch. Default project configurations implement auto-baud, allowing the PC to select the desired communications signaling rate. However, auto-baud is only enabled while the Bootloader is currently executing, or an existing Application is executing, but with the Application not having reconfigured the UART or received any characters.

Additionally, not all PC selectable baud rates are valid for the Bootloader, even when auto-baud is active. If no Config words are currently programmed, the Bootloader must execute from the internal FRC clock without any hardware option of run-time clock switching to use a PLL derived clock. As the internal FRC by itself yields a maximum of 4.0 or 3.7 MIPS, the PC host may need to communicate at a slow baud rate, such as 38400 bps to ensure auto-baud hardware on the target device is able to match the PC's baud rate without excessive mismatch.

This type of condition can also sometimes generate the `EZBL_ERROR_COM_READ_TIMEOUT (-20 or 0xFFEC)` "Bootloader signaled communications timeout while waiting for image data" error condition since it ultimately results in a corrupt data stream being transiently consumed by the Bootloader.

12. An existing Application is executing that has Bootloader RX/TX/NOW Timer interrupts disabled. I.e. these interrupts may be disabled, masked by execution of other ISRs for an extended



duration, the EZBL\_ForwardBootloaderISR bits are set to forward needed interrupts to Application ISRs, or the Bootloader NOW\_TASK callback has been run-time disabled.

13. Application eating RX characters from the software communications FIFO before Bootloader's NOW\_TASK callback (i.e. EZBL\_BootloaderTask()) executes to process them.

Hardware RX characters are written to the software RX FIFO in an ISR, with only an 8-byte wake up sequence match decoded in the ISR that decides if the Bootloader NOW\_TASK callback needs to be invoked for more extensive data testing. As this callback executes at the main(), IPL 0 context only sporadically, an Application can read from the FIFO and steal part of the .bl2 file needed for the Bootloader to decide if the incoming data is a valid bootload attempt.

To avoid this problem, the Application should only read or write to shared communications FIFOs when bootloading is disallowed or when `EZBL_COM_RX == (void*)0`. I.e. App code should suppress all communications processing after the ISR successfully decodes the 8-byte wake up sequence and sets the global EZBL\_COM\_RX pointer to the applicable FIFO address. If extended communications processing by the Bootloader reveals a non-applicable .bl2 file is being offered, or Bootloading is rejected, the Bootloader sets EZBL\_COM\_RX back to a null value to resume Application handling of the communications data streams.

14. Target is sleeping or otherwise operating without a clock that maintains the communications peripheral.
15. PC communications `-timeout=x` option set too short. This parameter is specified in milliseconds and must allow enough time to permit round-trip communications with the Bootloader and may require several 10's of ms up to 10's of seconds of delay that may occur locally on the PC. For example, when transmitting to a Bluetooth based UART, several seconds may be required before the Bluetooth recipient wakes up, establishes an active connection with the PC and begins receiving data that the PC has queued for transmission to the bootloader.

This error can occur transiently since Bluetooth and other communications hardware may keep the medium alive for some duration following bootload completion, avoiding the delay associated with exiting sleep modes and detecting that data is waiting for transfer from the host node.