

## 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. Almost 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**

- Common projects and code set supports nearly all PIC24/dsPIC33 targets without changes

#### **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 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**

- IVT entries forwarded to optimized Interrupt Goto Table in Application space
- Bootloader and Application can share any Interrupt Vector with individual run-time selection
- No AIVT hardware or complex 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
- Script-accessible Java PC command line communications executable ready for branded GUI wrapping and redistribution

## Getting Started

To get up and running with a UART bootloader on a PIC24FJ/PIC24H/PIC24E/dsPIC33F/dsPIC33E 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 should 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.

## What's In This Document

This document starts with an EZBL overview, provides usage information for the 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\\_boot\\_uart](#) – UART Bootloader
- [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 examples and Bootloaders

### 2. PC-side automation and communications projects:

- [ezbl\\_tools](#) – Java source code for all compile-time automation, code generation, data conversion, file transfer and processing
- [ezbl\\_comm](#) – Source code for OS-specific UART/I<sup>2</sup>C hardware driver access

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

- [What's Found Elsewhere](#)
- [Forward](#)
- [Typographical Conventions](#)
- [Overview](#)
  - i. [Topology](#)
  - ii. [Size](#)
  - iii. [Speed](#)
  - iv. [General Usage](#)
- [Bootloader Interrupt Handling](#)
  - i. [Dual Partition Interrupts](#)
  - ii. [Single Partition Interrupts](#)

- iii. [Default IGT Remapping](#)
- iv. [ISR Reuse](#)
- v. [Interrupt Forwarding](#)
- [Communications 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 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.

## Forward

The Microchip Easy Bootloader Library is a set of tools and building blocks to create bootloaders for all 16-bit PIC24FJ/PIC24H/PIC24E/dsPIC33F/dsPIC33E silicon products from Microchip Technology. It simultaneously provides ancillary timing, communications and flash programming APIs that applications can use – both with and without an actual EZBL bootloader implemented.

The target product lines contain hundreds of part numbers with minor differences in each. Ignoring communications protocol specifics, variations required for a one-size-fits-all bootloader to take into account are virtually endless, including: flash geometry, erase and programming sizes, ECC, configuration word locations and restricts, NVMCON<NVMOP> encodings, interrupt disable/enabling/vectoring, communications peripheral options/versions/instances, timer peripheral versions/instances, I/O pin mapping, oscillator/PLL/baud rate, security/Code Protect/Write protect modes, XC16 compilation models/optimization, flash PSV/EDS addressing, available ISA opcodes, silicon errata, hardware development boards, MPLAB® X IDE and XC16 versions, differing PC host OSes and APIs and the list continues.

Considering the sum of all permuted parameters may rival the number of atoms in the known universe, EZBL does not have an easy job internally, and initially it may not seem easy to implement in a product either. EZBL's multiple layers of abstraction, indirection and automation to try and generate parameter permutations that are valid will likely make debugging seem harder rather than easier.

However, don't be discouraged. In the end you will be successful! Engineering time required to arrive at a complete solution using EZBL should be appreciably shorter and "easier" than writing equivalent high level functionality starting with a vacuum.

## Typographical Conventions

A number of capitalization, colors and font formatting instances appear in this document. In most cases, these indicate specific entities and names.

Instance Example(s)	Meaning
<u>A</u> pplication (instead of <u>a</u> pplication) <u>B</u> ootloader (instead of <u>b</u> ootloader')	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 .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.
<code>_U2RXInterrupt()</code>	Source code or identifier
<code>ex_boot_uart</code>	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:

- UART – 9Kbytes flash, 284 bytes static RAM (96 + 32 bytes as RX/TX communications FIFOs)
- I<sup>2</sup>C Slave – 9Kbytes flash, 300 bytes static RAM (128 + 16 bytes as RX/TX communications FIFOs)
- USB Host MSD – ~33Kbytes flash, ~3.75Kbytes static RAM

FIFO sizes are configurable down to 64 bytes of RAM for RX and 8 bytes for TX. The vast majority of 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 subtracted from 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, a call to the `EZBL_SetAppReservedHole(0x004000, 0x008000)` macro is made in the `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 8192 Words (24Kbytes). The macro marks the address range as occupied to ensure no other project code or objects get linked into the same 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.

## Speed

Bootloading throughput varies greatly depending on communications baud rates, RAM RX FIFO buffer size, round-trip communications latency, processor execution clock, geometry of the device's flash, existing flash state prior to programming a new Application image, Bootloader size and performance characteristics of external hardware/communications bridges providing new Application images.

### UART

Typical, measured performance for a couple configurations are:

- 15.0Kbytes/sec (16.7 seconds total): `ex_boot_uart`, 230400 baud, MCP2221A USB to UART bridge, dsPIC33EP512MU810 @ 70 MIPS, 96 byte software RX FIFO, 250KB Application .bl2 image upload size, 241KB of flash needing to be erased and programmed, 286KB needing to be blank checked, 250KB needing verification
- 61.7Kbytes/sec (4.07 seconds total): `ex_boot_uart`, 761000 baud, MCP2200 USB to UART bridge, dsPIC33EP512MU810 @ 70 MIPS, 1024 byte software RX FIFO, 250KB Application .bl2 image upload size, 241KB of flash needing to be blank checked and programmed (no erase needed).

A baud rate in excess of ~761Kbps for 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 buffer on UART peripherals will overflow during a minimum NVM word/double word flash programming operation at higher baud rates while the CPU is blocked.

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

I<sup>2</sup>C Bootloaders will experience lower throughput (ex: < 10Kbytes/sec) when using the MCP2221A, despite a 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.

### *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 two pass installation is performed, verifying 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 kind of communications/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 adjustments to the project 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** 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()**. A few symbols are generated to initialize stack pointer and other 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 a **.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 device information provided by MPLAB X IDE. 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 erase size)
  - f) Minimum programmable block size

Additionally, 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. 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` 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` linker script. Information collected includes:
  - a) Address ranges occupied by Bootloader content in flash or marked as reserved app space
  - b) Names of all ISR functions implemented in the project
  - c) PSV/EDS page that the `.const` flash section is located on
  - d) Config word names, addresses and values defined in the projectUsing this information, plus information collected in earlier steps, the generated/derived information that gets written back to the `.gld` linker script includes:
  - a) Flash geometry occupied or reserved by the project, padded as necessary to start and end on perfect flash erase page size 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 is triggered by a hardware interrupt.
  - e) 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` and `.a` files, but with the `.gld` file modified in the prior step. 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` 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.
10. Invokes `ezbl_tools.jar --blobber` to convert the `.hex` file to a `.bl2` file

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` 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 and do implement their own interrupt handlers. Typically one Timer or MCCP/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 after the Application has begun execution.

Customized Bootloaders can seamlessly add other interrupt handlers, up to at least 32 total.

Often times, a hardware interrupt is a precious and unique hardware resource that cannot be dedicated to permanent Bootloader functionality and rendered inaccessible to Application projects. 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 both potential problems by implementing other interrupt sharing options described in this section.

### 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 comele 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.

By having all Bootloader ISR source code available, an implementation specific means of changing ISR behaviors between Bootloader and Application virtual modes can be created directly with any ISR.

### 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:

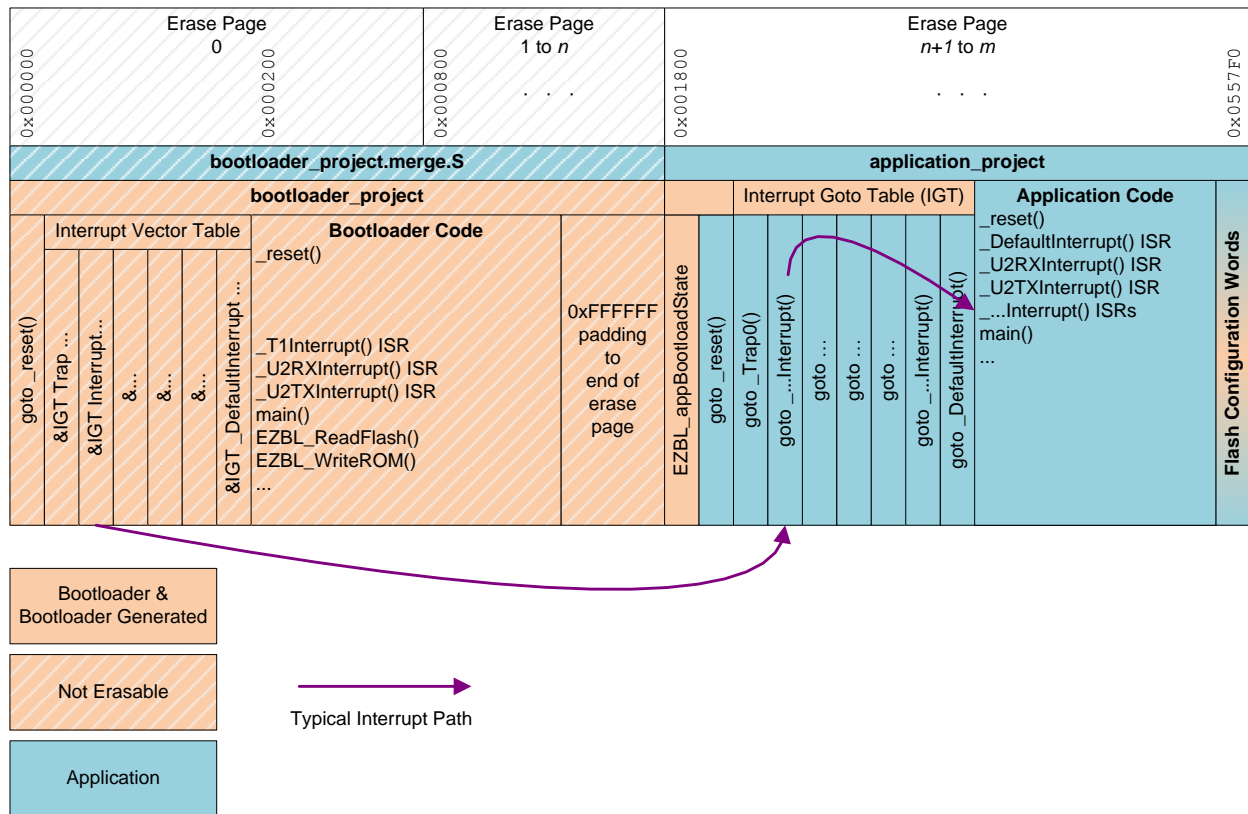
1. [Default IGT Remapping](#): any interrupt not implemented in the Bootloader project always propagates to an Application defined ISR or the Application `_DefaultInterrupt()` handler via an erasable/reprogrammable Interrupt Goto Table ("IGT").
2. [ISR Reuse](#): hardware invokes the Bootloader ISR and Application code relies on higher-level Bootloader APIs without implementing a separate Application ISR to handle the same interrupt.
3. [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 devices, or 4  $T_{CY}$  on PIC24E/dsPIC33E 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 depends 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:

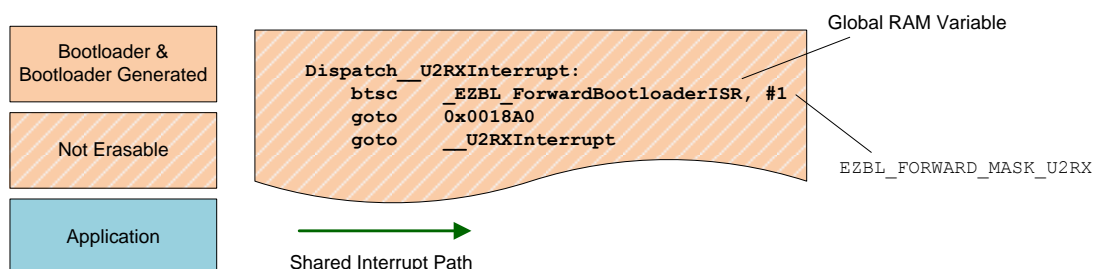
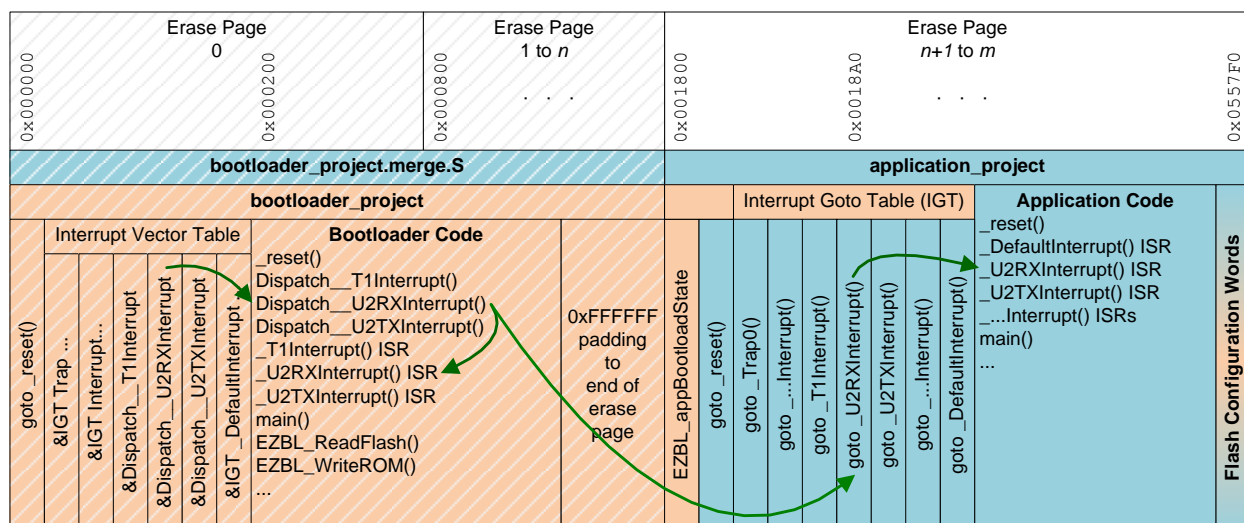
- Application writes message to `I2C1_TxFifo` by calling `EZBL_FIFOWrite()`
- I<sup>2</sup>C master issues an I<sup>2</sup>C read request to the PIC at slave address `0x60`
- `Bootloader_SI2C1Interrupt()` ISR prefixes the FIFOed outbound TX data with a 1 byte header specifying how many bytes are in the 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 similar to having hardware swap between an IVT owned by the Bootloader and an 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:

3. Disable the hardware interrupt, ex: `IEC1bits.U2RXIE = 0;`
4. Set the appropriate forwarding bit in the `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. Clear the appropriate forwarding bit in the `EZBL_ForwardBootloaderISR` global variable, ex: `EZBL_ForwardIntToBoot(U2RX);`

In this example, `U2RX` is an (undefined) C token indicating which interrupt needs forwarding. Internally, the `EZBL_ForwardIntToApp()` macro concatenates this token onto the `EZBL_FORWARD_MASK_` linker symbol name to obtain the correct bit position in `EZBL_ForwardBootloaderISR` variable for the given interrupt. The mask is auto-generated by `ezbl_tools.jar` during compilation of the Bootloader project and injected into the bootloader's `.gld` linker script, whereas the `EZBL_ForwardBootloaderISR` variable is declared in `ezbl_lib.a` and pulled into the Bootloader project automatically by code that references it.

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 and `EZBL_FORWARD_MASK_xxxy` symbol in the Bootloader's `.gld` linker script. Deleting an ISR and rebuilding will have the reverse effect of deleting an existing dispatch stub and forwarding mask symbol.

The reset state of `EZBL_ForwardBootloaderISR` is `0x00000000`, so all interrupts are, by default, are 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), you can simply clear all `IECx` registers and set `EZBL_ForwardBootloaderISR = 0xFFFFFFFF`; All '1's means to forward all interrupts, with unused bits having no meaning or effect.

### Interrupt Latency

Dispatching stubs are sometimes generated with BRANCH-always instructions instead of GOTO instructions. Therefore, the 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 ISR	Latency to Application ISR
PIC24F/PIC24H or dsPIC33F	Yes	< 128KB	+4 TCY	+5 TCY
	Yes	≥ 128KB	+5 TCY	+5 TCY
	No			+2 TCY
PIC24E/dsPIC33E	Yes	< 128KB	+6 TCY	+9 TCY
	Yes	≥ 128KB	+7 TCY	+9 TCY
	No			+4 TCY

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

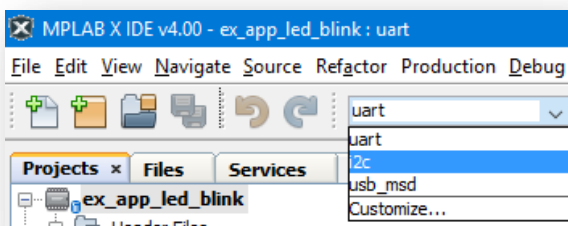
This on-demand polling behavior is primarily implemented for debugging purposes as it allows serial console writes to complete from within a trap exception handler or execution context in which the hardware interrupt is disabled or masked by a higher IPL. Such forced data transfer allows data corruption 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.

## *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

Each of these Bootloader pairings has to be selected and compiled separately. 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 actually 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:



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` being 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, dsPIC33EP512MU810 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. 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 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 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 as it
28 # Echo EBL: Converting .elf/.hex file to a binary image
29 -test "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" -nt "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).elf"
30 -test "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).unified.hex" -nt "dist/$(CONF)/$(IMAGE_TYPE)/$(PROJECTNAME).$(IMAGE_TYPE).unified.hex"
31
32 ifneq ($(filter default uart,$(CONF))) # Check if "default" or "uart" MPLAB project build profile is used
33     Echo EBL: Attempting to send to bootloader via UART
34     $(MF_JAVA_PATH)java -jar "$(thisMakefileDir)ezbl_tools.jar" --communicator=$(COM11) --baud=230400 --timeout=1100 --artifact="$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2"
35 else ifneq ($(filter i2c,$(CONF))) # Check if "i2c" MPLAB project build profile is used. If so, upload via MCP2221 I2C.
36     Echo EBL: Attempting to send to bootloader via I2C
37     $(MF_JAVA_PATH)java -jar "$(thisMakefileDir)ezbl_tools.jar" --communicator=$(I2C) --i2c_address=0x60 --baud=400000 --timeout=1100 --artifact="$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2"
38 else ifneq ($(filter usb_msd,$(CONF))) # Check if "usb_msd" MPLAB project build profile is used. Have a nice FIRMWARE.BL2 file to copy to it
39     test -e "$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" && cp "$(DISTDIR)/$(PROJECTNAME).$(IMAGE_TYPE).bl2" "$(DISTDIR)/FIRMWARE.BL2"
40 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.

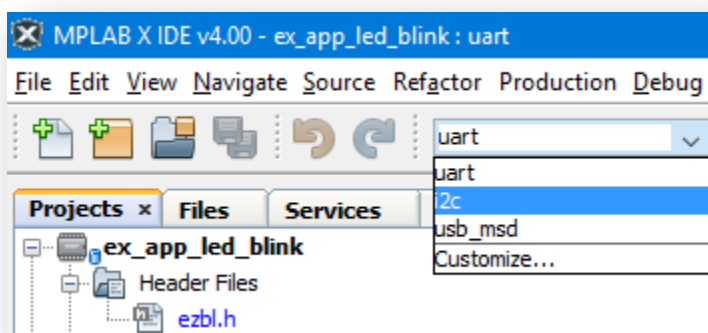
The example Bootloader projects also automatically copy their latest `ex_boot[i2c/uart/usb_msd].merge.gld` file generated out to example Application projects like `ex_app_led_blink`. This behavior will therefore overwrite any manual changes you make to the `ex_boot[i2c/uart/usb_msd].merge.gld` linker 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. All of 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[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 of the global functions and variables exported from the Bootloader project. All of the Bootloader's flash contents are encoded as absolute data at permanent addresses, so the effective contents of this file will not change when the Application project is recompiled, 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:

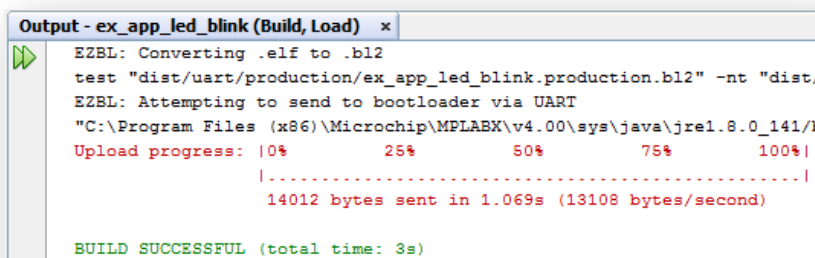


4. In the **Project Settings** for the selected build profile, select the same target PIC/dsPIC device as was specified when you built your Bootloader project.
5. [UART/I<sup>2</sup>C only] Ensure your target board is powered, has been programmed with your Bootloader using an ICSP hardware tool and has 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 "Error: no target response", refer to [Communications Error Messages](#).

- b) [USB MSD only] Using your system's file manager, copy the `dist\usb_msdc\production\FIRMWARE.BL2` file to the **root directory** on a USB thumb drive and then plug your media into your target circuit. The

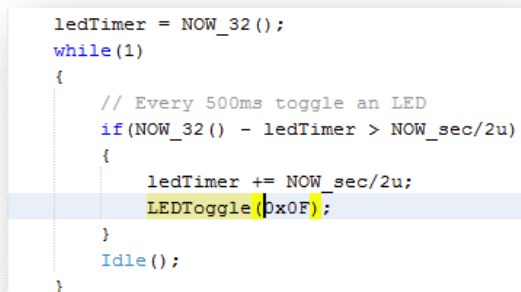
media must be FAT16/FAT32 formatted and implement a hardware sector size of 512 bytes (unless you enabled a larger FILEIO library sector size option when building your Bootloader).

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.

### 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\v1.32\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 whenever 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 a particular 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_tools.jar`
- `ezbl_comm.exe`
- `"ex_app_led_blink.production.bl2"`

Additionally, the user must have a suitable Java JRE (Runtime/Virtual Machine) installed and available in their system path. `ezbl_tools.jar` can be executed by Java 7 (SE JRE 1.7) and newer Java run-times. As this version of Java is several years old (and effectively obsolete due to security updates), it is likely that most systems will already have a suitable JRE installed. Others can be directed to <https://www.java.com/> to download the latest version of Java.

`ezbl_tools.jar` can be executed in a 32-bit or 64-bit JRE process. This is the communications front end that reads the firmware file and displays status back to the console.

`ezbl_comm.exe` is an API bridge executable for allowing `ezbl_tools.jar` to access Windows COM port drivers and the MCP2221A I<sup>2</sup>C interface. It launches automatically when `ezbl_tools.jar` requests it and closes automatically when bootloading is complete or times out (`ezbl_tools.jar` closes its handle to it). It does not contain a GUI and no console window is generated either when this executable is launched by `ezbl_tools.jar`. This is a 32-bit executable,

but is compatible with 64-bit OSES and interoperates correctly if ezbl\_tools.jar is started by a mismatched, 64-bit, JRE since both executables execute in their own system process.

"ex\_app\_led\_blink.production.bl2" is your Application firmware update file, generated automatically from your .elf compiler 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 customer confusion if multiple Application versions are released over time.

With all of these files in the same location, the procedure to transfer them to the Bootloader is:

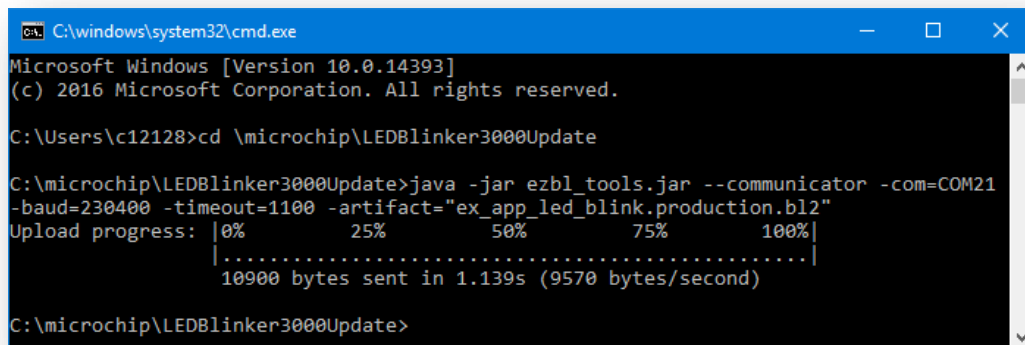
1. Open a Command Prompt or Windows PowerShell window
2. Change to the directory containing all three files
3. Determine the correct COM port that the Bootloader is attached on
4. Execute the upload command, such as:

```
java -jar ezbl_tools.jar --communicator -com=COM21 -baud=115200 -timeout=1100  
-artifact=ex_app_led_blink.production.bl2
```

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

```
java -jar ezbl_tools.jar -communicator -com=I2C -i2c-address=0x60 -  
baud=400000 -timeout=1100 -artifact=ex_app_led_blink.production.bl2
```

Both of these command examples need to be inputted as a single line command.



```
C:\windows\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\c12128>cd \microchip\LEDBlinker3000Update

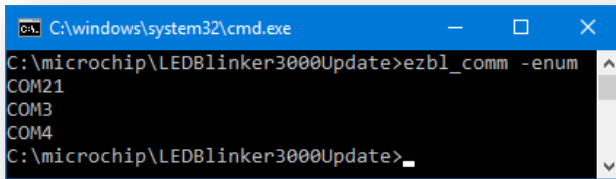
C:\microchip\LEDBlinker3000Update>java -jar ezbl_tools.jar --communicator -com=COM21
-baud=230400 -timeout=1100 -artifact="ex_app_led_blink.production.bl2"
Upload progress: |0%      25%      50%      75%      100%|
                  |.....|
                  10900 bytes sent in 1.139s (9570 bytes/second)

C:\microchip\LEDBlinker3000Update>
```

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>
```

[I2C 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 all of 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, negating a need to manually launch the 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
:: Test if java is installed, sending stdout and stderr streams to the
:: bit-bucket so nothing displays, pass or fail
java -version >>NUL 2>>NUL
if ERRORLEVEL 0 goto JavaFound
echo Java 7 or newer is required to update LED Blinker 3000. Please install
echo the latest version of Java from: https://www.java.com/
goto End

:JavaFound
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
java -jar ezbl_tools.jar --communicator ^
    -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 similar to this:

```

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 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 on board 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/I2C 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 the *ex\_boot\_uart* implementation in that the I<sup>2</sup>C master need only write the firmware ".bl2" file to the slave Bootloader's I<sup>2</sup>C bus address. 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 I2C
- 16-bit Timer or 16-bit CCP Timer (for timeout detection)

The exact I2C 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 I2C (and/or UART) peripheral is initialized, the Bootloader will accept firmware updates from multiple interfaces. The majority of hardware initialization files distributed with EZBL default to using the Timer 1 and I2C 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` less 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 their impact to allowed baud rate 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 peripheral 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 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](#). 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

This project typically implements three interrupt handlers:

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

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. The majority of hardware initialization files distributed with EZBL default to using the UART2 peripheral and Timer 1.

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**

2. Outputs a Bootloader .hex file, [ex\\_boot\\_uart.production.hex](#), for use with a traditional ICSP based programmers (PICkit, ICD, REAL ICE, etc.)
3. 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.
4. 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**

2. All PIC24FJ/PIC24H/PIC24E and dsPIC33F/dsPIC33E/dsPIC33C products (PIC24FxxK K-flash flash devices and dsPIC30F not supported)
3. Out-of-box demo ability on Explorer 16/32 development board (or Explorer 16 with external USB to RS232 converter)

4. Out-of-box demo ability on numerous PIMs. See the [hardware\\_initializers](#) project folder contents to check for immediately testable targets.
5. Out-of-box demo ability on Microsoft Windows (COM Port driver interfacing code is OS dependent)
6. Tested with MPALB® X IDE v4.01
7. Tested with MPLAB XC16 compiler v1.32

## 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. 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: EZBL_BOOT_PROJECT=uart
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" PIC 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.

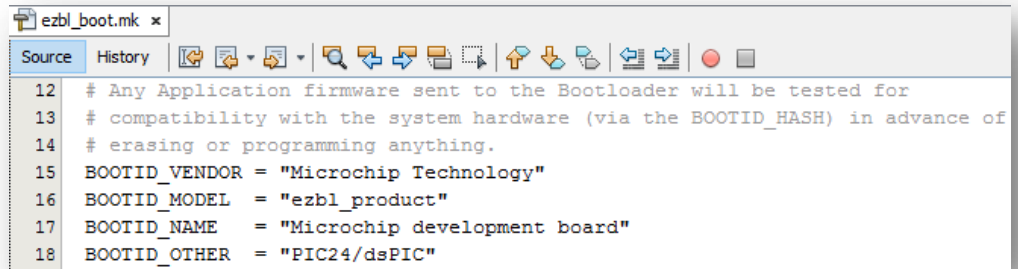
As a Bootloader project, a `_DefaultInterrupt()` ISR created by the linker is unnecessary since all unhandled interrupts are automatically mapped into the Application's Interrupt Goto Table. The default, checked, state of this option would generate dead code.

5. By default the UART operates in auto-baud mode so the host can choose the communications baud rate. If this is not desirable, such as when tunneling through a radio that requires a fixed baud rate, define the 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()`.

## 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, 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()` or implement an `unsigned int LEDToggle(unsigned int toggleMask)` function. If you don't have or want any LED(s) toggling, you may instead delete the `LEDToggle()` and `LEDOff()` function calls in `main.c`.

Other initialization code located in the provided hardware initializer example files are applicable to other EZBL projects or is unneeded. It can be removed/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 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 of 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 absolutely 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 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 collects information about the target device from a data base within MPLAB X IDE and executes using a private Java JRE copy in the MPLAB X installation path, so it is important to be thorough.



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

The *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 [ex\\_app\\_led\\_blink](#) 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* linker script, the *ezbl\_lib.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
2. Explorer 16/32 development board, an Explorer 16 + USB PICtail Plus adapter, or a number of starter kits with USB Host functionality
3. Tested with MPLAB® X IDE v4.01
4. Tested with MPLAB XC16 compiler v1.32

### Notes

1. Because memory-type bootloaders have to 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.

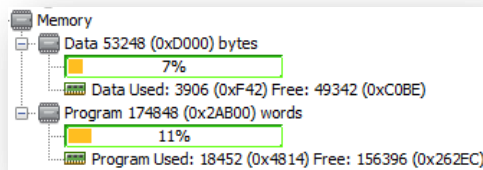
If no valid Application existed or the first pass reached successful completion, the file read pointer is seeked to file offset 0 and all data is read with real flash erase/write and verification steps executed.

2. 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.

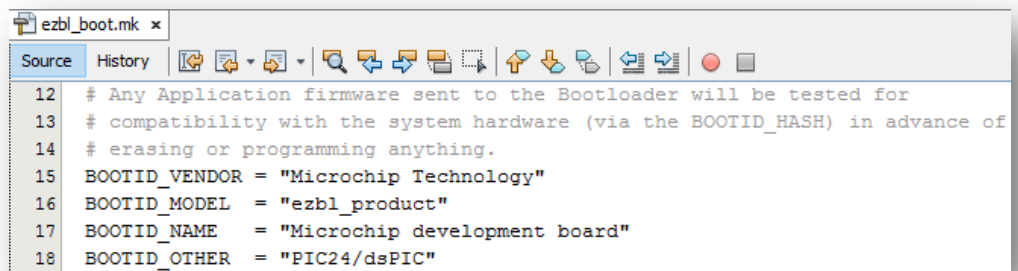
An additional, benign consequence of using the `EZBL_SetAppReservedHole()` macro is that the MPLAB® X IDE Dashboard program space utilization will indicate the reserved space as part of the Bootloader flash footprint:



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]`
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 [ex\\_app\\_led\\_blink](#) example Application project
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/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/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 not 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.

## *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 dependant). Excluding uploading, development may be done with all OSes supported by MPLAB® X IDE and the XC16 compiler.
8. Tested with MPLAB XC16 compiler v1.32
9. Tested with MPLAB X IDE v4.01

### 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:

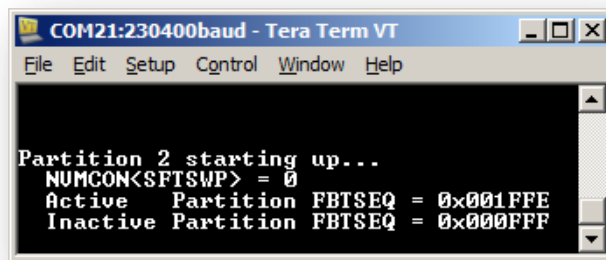
```

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)

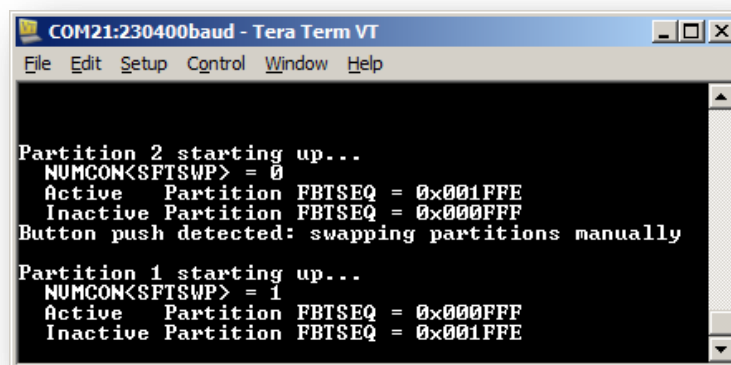
```

- 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.



- 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).



11. Toggle MCLR or Button S4 again to revert back 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

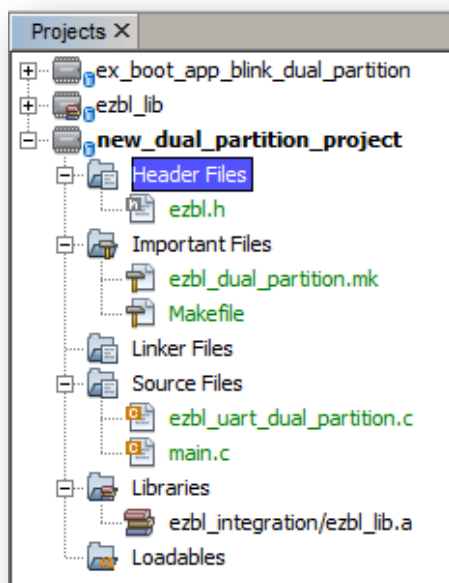
To regenerate the `ex_boot_app_blink_dual_partition` MPLAB project or add the EZBL bootloader functionally to an existing project, follow these steps:

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:
 

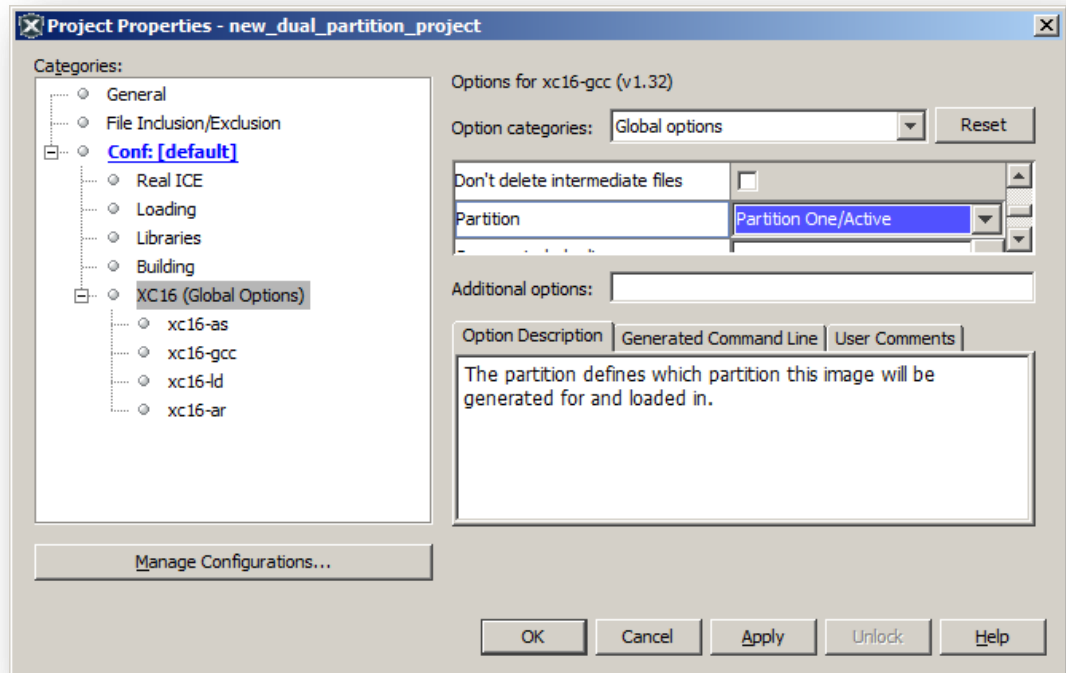
```
include ezbl_integration/ezbl_dual_partition.mk
```



- 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**

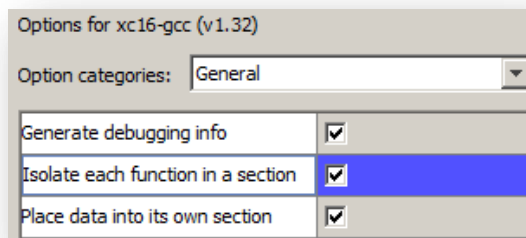


4. Open the Project Properties dialog
- a) XC16 (Global Options)
- (1) Set "Partition" to "Partition One/Active". Note: this option may not exist unless your project is first targeting a Dual Partition capable device. Be sure and set the device part number, click "OK", and reopen the Project properties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



b) xc16-gcc

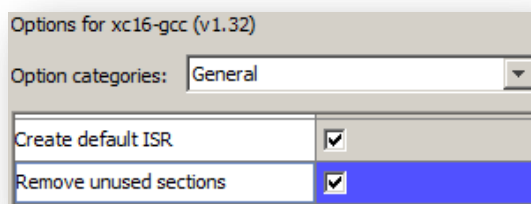
- (1) None of the remaining compiler options are critical to operation of EZBL code, however, turning on "Isolate each function in a section" and "Place data into its own section" can normally result in smaller flash and RAM requirements when combined with the "Remove unused sections" xc16-ld linker 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 particular 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  $F_{CY}$  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\\_smps\\_v1,](#)*

*[ex\\_app\\_live\\_update\\_smps\\_v2,](#)*

*[ex\\_app\\_live\\_update\\_smps\\_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](#)) or MCP2200 Breakout Module ([ADM00393](#)) 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\_smps\_v1* implements a baseline reference SMPS Buck + Boost power supply and *ex\_app\_live\_update\_smps\_v2/ex\_app\_live\_update\_smps\_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\_smps\_v2* can only Live Update against v1. It follows a "Preserve All" linking model where the majority of 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, need 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\_smps\_v3* can only Live Update against v2. It follows a "Preserve None" linking model where the majority of 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.

Documentation for these projects will be expanded in a future EZBL release and be augmented by a Microchip Application Note on Live Updating Dual Partition devices.



### ***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](#)

## *ezbl\_tools* NetBeans Java Project

*ezbl\_tools* is a PC Java application that handles essentially everything needed to build a valid Bootloader for an arbitrary target device, convert Application build artifacts to a format suitable for serial transmission/Bootloader programming and communicate with the target. This java application, through invocation via a project's **Makefile**, triggers several build-time actions, including:

- Modifies other MPLAB® X IDE generated makefiles to reinvoke *ezbl\_tools* at the appropriate times during project build
- Interrogates MPLAB X Crownking edc (Essential 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** linker script with information from the compiler's default **.gld** 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." Data table contents include:
  - Bootloader geometry to suppress erase/programming of Bootloader occupied flash locations
  - 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
- Creates **.merge.S** assembly source files to encapsulate a Bootloader image and **.merge.gld** linker scripts for Application projects to use flash addresses that are compatible with the Bootloader
- Converts **.elf/.hex** program space contents to **.bl2** file for upload to the Bootloader. Data record reordering and coalescing, address alignment and padding, SHA-256 hashing and CRC32 generation for **.bl2** contents are implemented.
- Communicates via multi-threaded pipe read/writes to **ezbl\_comm.exe** process to facilitate UART or MCP2221A I<sup>2</sup>C with an EZBL Bootloader to upload new Application images

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**
2. Copies **ezbl\_tools.jar** to applicable **ezbl\_integration** folders in the Application and Bootloader example projects.

### 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.
  - a) If making significant functionality customizations to this tool, becoming familiar with what is included in the *MPLAB X SDK* is recommended. This SDK contains libraries and documentation for various

MPLAB X classes and functionality that may be relevant to your customization. Changes to the communications protocol or .bl2 container format implemented by EZBL, however, can be done without installing or studying the MPLAB X SDK as these elements were written to be isolated from and executed without having any of the MPLAB X IDE/IPE tools installed.

- b) The MPLAB X SDK is available from <http://www.opensource4pic.org/content/content/mplab-x-sdk-mplab-x-ide>
- c) If the MPLAB X SDK is installed, API documentation can show up automatically within the NetBeans IDE as you type Class names or reference their member functions/variables/objects for certain classes. Getting context-sensitive code pop-ups for MPLAB X SDK components requires you to modify the NetBeans Project Properties and include the proper path in the Compile Time Libraries for Javadocs. For example, *ezbl\_tools* depends on and calls into *crownking.jar* (or *crownking.common.jar*) + *crownking.edc.jar*, which is distributed with the MPLAB X IDE/IPE and also has documentation available in the MPLAB X SDK. Getting context sensitive help means you would use Add JAR/Folder to reference *crownking.jar/crownking.common.jar* and then Edit it to include Javadocs from a path like *C:\mplab-x-sdk-v4.00\crownking\html\javadoc\*.



## ***ezbl\_lib* MPLAB® X Project**

This is a source library project designed for portable, reusable, pre-compiled functions that a Bootloader or Application alike will regularly need. Features such as 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 PIC24/dsPIC targets without changes are located here.

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.

### **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 generally use any of the exported APIs.
2. Most files compiled with MPLAB® XC16 compiler v1.32. Object code for PIC24E, dsPIC33E and dsPIC33C processor targets may have been compiled with XC16 v1.31. Any compiler version can normally 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*, *ezbl\_lib/sectioned\_functions* or *ezbl\_lib/weak\_defaults* folder and place them as local files within your Bootloader and/or Application projects. 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) Include the *ezbl\_lib.a* archive in any project that wishes to use these library functions, not the *ezbl\_lib* project itself. Generally, it is a good idea to include *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.

## *ezbl\_comm* Visual C++ 2008 Express Edition Project

This is a multi-threaded, headless executable project intended to bridge bidirectional COM Port data between the Windows API and an interface that is suitable for native access in the Java environment, i.e. [ezbl\\_tools](#). Bridging is achieved by implementing two dedicated threads that copy COM RX data from the Windows API into the `\\.\pipe\ezbl_pipe_in_from_com` local communications pipe and a separate thread that copies Java TX data from the `\\.\pipe\ezbl_pipe_out_to_com` communications pipe out to the Windows API for the COM port.

Java natively has file I/O library classes that can read and write to named pipes much as if they were physical files on a hard disk. However, Java does not have a native library to access serial COM Ports, nor can easily call native Windows OS APIs, thus necessitating a secondary [ezbl\\_comm.exe](#) helper executable for [ezbl\\_tools.jar](#) to create a software communications link with the target hardware.

For I<sup>2</sup>C master mode communications, *ezbl\_comm* is exercised as well. Communications threads write to the MCP2221A driver object code statically linked into the [ezbl\\_comm.exe](#) executable image. The pipe names remain the same as they exist for UART use, but to create an identical data stream for the [ezbl\\_tools.jar](#) Java process, the 1-byte framing bytes received from the Bootloader during an I<sup>2</sup>C Read command are stripped off within *ezbl\_comm*. *ezbl\_comm* also periodically generates I<sup>2</sup>C Read commands so Bootloader data can be returned to the PC.

**Porting Note:** The code in this project is very complex for what it does. It is not recommended that this code be modified or ported to other platforms. The internal complexity stems from being implemented with multiple threads, including debug TX/RX logging with timestamps and Windows error codes, including MCP2221A I<sup>2</sup>C communications and from managing local named pipe accesses. The project uses multiple threads to operate as fast and scalable as possible without regards to the underlying communications protocol that gets tunneled through the bridge.

If your goal is to create a communications application to support bootloading from Linux, Mac OS, a mobile device, or other host hardware platform, then it would be much simpler to create a standalone executable from scratch to support your product. In other words, it is suggested that you ignore the generic pipe tunneling concept with [ezbl\\_tools.jar](#) and just implement a program that copies a .bl2 firmware image to the platform's hardware COM driver under software flow control indicated by the Bootloader response traffic. This will avoid the need to redistribute [ezbl\\_tools.jar](#) and simultaneously make the task simpler as all pipe communications, multi-threading, synchronization and use of unfamiliar tools can be negated.

The information necessary to implement the EZBL communications protocol is located in [help\EZBL Communications Protocol.pdf](#).

### When Built

1. Outputs [ezbl\\_comm.exe](#). This file is copied to [ezbl\\_integration](#) folders containing [ezbl\\_tools.jar](#) which is normally the executable that launches [ezbl\\_comm.exe](#).

### Supports

1. Microsoft Windows environments only. [ezbl\\_comm.exe](#) is a 32-bit executable, but due to inter-process communications via named pipes provided by the OS, the companion [ezbl\\_tools.jar](#) executable can be launched in either a 32-bit or 64-bit Java VM.

### Notes

1. Besides the Visual C++ compiler, this project requires a Microsoft Windows SDK to be installed with proper IDE path settings to be able to rebuild this executable.

2. The input and output pipe names are constant and therefore it is not possible to open two or more COM ports at a time.
  - a) This generally is of no significance since it is impractical to start two bootloading tasks simultaneously when they individually don't take long to complete and self-close.

## Communications Error Messages

A number of things could go wrong when bootloading. Although EZBL is unlikely to brick itself, there are multiple messages that you may see when using the Communicator class in `ezbl_tools.jar` to upload firmware to a Bootloader. The most authoritative source of information regarding any particular message will always be the source code itself; however, this section enumerates the most common messages and their likely cause/solution.

### Error: no target response

This error indicates that the PC's COM port is valid and the first few bytes of the .bl2 file were (believe to be) transmitted onto the wire without receiving any positive or negative acknowledgement back from the Bootloader target.

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 "no target response" 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:

- Target not powered or communications cable not plugged in
- Target does not have a Bootloader programmed in flash or the Bootloader contains Config words invalid for execution (ex: invalid clock)
- TX/RX or SDA/SCL I/O pins are not configured correctly.
  - 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.
  - Ensure that Peripheral Pin Select registers, when applicable, have been initialized correctly.
  - Ensure ANSEL/PCFG bits on the Bootloader target communications lines are set to place the proper pins in a digital I/O mode.
  - 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.
  - 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.
- Target is executing the wrong Bootloader. Application images are tied to the Bootloader they were compiled against (via the .merge.S/.merge.gld files and BOOTID\_HASH values defined in the Bootloader project). To facilitate easy migration to shared/broadcast based communications mediums with multiple bootloadable nodes on it, EZBL Bootloader projects are designed to passively ignore .bl2 file offerings for Bootloader targets that are not itself, resulting in no Bootloader response back to the PC for .bl2 files that are not applicable. If in doubt, program the Application .hex file using an ICSP programming tool and retest bootloading. The Application .hex image contains the Bootloader, so ICSP programming once ensures the Application is matched to the Bootloader contained within it.
- Application .bl2 file created 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 generate 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.
- Target is connected to a different COM port from the one opened on the PC for bootloading

- Anti-Virus software blocking execution of the ezbl\_comm.exe driver bridge process from being launched by the Java ezbl\_tools.jar communications application.
- 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 few 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.

Several causes of transient/state related failure are:

- 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.

- 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.
- Application eating RX characters from the software communications FIFO before Bootloader NOW\_TASK callback 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 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.

- Target is sleeping or otherwise operating without a clock that maintains the communications peripheral.
- PC communications `-timeout=x` option to `ezbl_tools.jar --communicator` set too short. This parameter is specified in milliseconds and must allow enough time to permit round-trip communications with the Bootloader, plus several 100's of ms of delay that may occur locally on the PC. For example, launching a bootload attempt can indirectly trigger a virus scanner that blocks further ezbl\_tools.jar execution, pending a clean result for the ezbl\_comm.exe helper communications executable and the .bl2 file being read from disk for transmission to the Bootloader.

**Error: Cannot run program "ezbl\_integration\ezbl\_comm.exe": CreateProcess error=2, The system cannot find the file specified**

**ezbl\_comm.exe** is a required file that manages OS-level communications calls originating in the Java **ezbl\_tools.jar** application. You must have a matching **ezbl\_comm.exe** file in the same folder that **ezbl\_tools.jar** is located and have file permissions set to allow execution.