AN2601

Online Firmware Updates in Timing-Critical Applications

Author: Alex Dumais and Howard Schlunder
Microchip Technology Inc.

INTRODUCTION

A requirement for many applications is to support firmware updates after the product has been released. Firmware updates force the application to include a bootloader to manage the Flash contents and a communications medium to share information with an external host device. Depending on the architecture of the microcontroller, instruction Stalls are executed during Flash erase/write operations, and because these operations can take several milliseconds, the application is typically taken offline to perform firmware updates. Downtime can last several seconds, which may be costly for certain end applications. An example of such an application would be a digitally controlled Power Supply Unit (PSU) for server markets. In a non-redundant scenario, performing a software update brings the PSU offline, resulting in revenue loss. Even in a redundant environment, manual effort is required to reprogram the PSU, which also results in additional cost.

To eliminate downtime during firmware upgrades, Microchip has introduced several 16-bit device families (such as dsPIC33EP64GS506, dsPIC33EP128GS808, PIC24FJ256GB412 and PIC24FJ1024GB610) that implement two Flash partitions. With the microcontroller architecture having two physically different Flash partitions, the application is able to execute as normal from the Active Partition, while the Inactive Partition is erased, programmed and verified by the Active Partition's bootloader. Additional device features have been implemented for supporting Flash partition swaps (soft swap) and are briefly mentioned throughout this application note. For more information on the dual partition implementation, refer to **"Dual Partition Flash Program Memory"** (DS70005156) in the *"dsPIC33/PIC24 Family Reference Manual"*.

This application note discusses the compiler tools that help facilitate LiveUpdate, as well as details regarding how to set up the MPLAB® X IDE project and implement firmware for the LiveUpdate event. Example projects are created and discussed in this application note to further aid in the development/testing of a LiveUpdate project. These PSU examples achieve a complete firmware replacement with an effective downtime of less than a few microseconds, avoiding any missed interrupts (critical) or discontinuity of the system's output.

COMPILER FEATURES FACILITATING LiveUpdate

In MPLAB XC16, Version 1.30 and later, several features have been added that enable online firmware updates in time-sensitive applications. In a LiveUpdate application, the biggest challenge is retaining RAM values/variables in the same memory address across firmware versions. By retaining RAM contents, run-time state information is maintained, and by keeping the same variables allocated to the same addresses, start-up initialization delay is negated. The executable and linkable file (`.elf`) contains all pertinent symbol (variable and function address) information needed for preserving RAM. In referencing the executable file from the previous build output, the compiler/linker will be able to map preserved variables to the RAM locations defined in the previous build.

For supporting LiveUpdate applications, four different Preserve Data models have been created. In all models, the previous firmware version's `.elf` file is referenced. Details regarding how to reference the `.elf` file and configure the project for a preserved data model will be discussed later in **"Configuring MPLAB® X Project"**.

Depending on the preserved data model used, the compiler/linker will either:

- Preserve only those variables with the *Preserved* attribute
- Preserve all variables in selected C files
- Preserve all variables in the active project (all C files)
- Preserve all variables in the entire project (including library archive files)

Preserving all variables in the active/entire project is seen as the more commonly used preserved data model. Any new variables will be initialized by the compiler, or if in the critical timing path, within a project-specific priority function. Function Pointers and pointers to constant data objects stored in Flash (if any) are manually reinitialized, and minimal firmware is required to manage changes between firmware updates. This approach brings the application 100% back online in the least amount of time. It is expected that only minor changes are made between firmware revisions, which makes managing the new variables/pointers reasonable in these data models.

There are instances where preserving individual variables on a C file basis, or selectively using the Preserved attribute, are required or preferred. In these situations, limited variables are maintained to keep the core application functional (power supply regulating) and the time to get the application 100% back online is not critical. This approach requires reinitialization routines to configure variables/peripheral states correctly (i.e., bootloader, task scheduler, etc.) as it would from a Power-on Reset (POR). Being that the application's core functionality continues uninterrupted, the time to reinitialize these tasks is acceptable from the user's point of view. This approach has a better chance of ensuring that all variables are linked to the project, permitting large code changes to the project.

Deciding what model to use can vary depending on the type of firmware update and is dependent on the firmware engineer to understand what model best fits the update. In some instances, it may be better to selectively preserve a few variables; in other instances, it might be better to preserve everything, selectively choosing what is new and needs changing.

To support these different Preserve Data models, the compiler has introduced several new attributes. Mainly these attributes determine which variables need to be preserved or which variables are new and require initialization. The compiler also now supports a way to assign order to variable initialization and execute time-critical functions before definable blocks of variables are initialized. The new attributes are discussed in the following sections.

Preserved Attribute

The *Preserved* attribute specifies that the variable's address needs to be consistent between firmware versions and that the compiler should not reinitialize this variable on a soft swap event. A variable with a Preserved attribute will always be initialized from an ordinary device Reset unless it is simultaneously attributed as Persistent (Example 1).

EXAMPLE 1: PRESERVED ATTRIBUTE

```
uint16_t __attribute__((preserved)) foo = 0;
```

The Preserved attribute is not required in the Preserve All model as the compiler/linker will implicitly treat all variables as requiring preservation. See **"Configuring MPLAB® X Project"** for more information on Preserve Data models.

Update Attribute

The *Update* attribute specifies that the variable is new or structurally modified and needs to be initialized. Variables with the Update attribute are initialized on a soft swap event, as well as a device Reset, unless marked with the Persistent attribute (Example 2). Update also gives the linker freedom to allocate (or reallocate) the variable to any unoccupied RAM address.

EXAMPLE 2: UPDATE ATTRIBUTE

```
int16_t _attribute_((update)) foo = 10;
int16_t _attribute_((update, persistent)) foobar;
```

The Update attribute is only required when the variable is declared in the scope of one of the Preserve All models. These variables will then be excluded from inheriting an implicitly assigned Preserved attribute.

Priority Attribute

The *Priority* attribute allows code execution and variable initialization by the Compiler Run Time (CRT) in a precedence manner. With the Priority attribute, variables will get initialized and functions will get called, in the order of their priority, before completion of data initialization by the CRT and the invocation of `main()`. The Priority attribute is applied in both a soft swap scenario, as well as a cold start. See Example 3 for applying the Priority attribute.

The compiler supports $2^{16} – 1$ priority classes with no limitations to the number of functions/variables in each priority class. Priority 0 is the highest priority (i.e., executes at the earliest possible opportunity) while Priority Class 65535 will be processed after all lower numbered priority classes have been processed. The order of function calls that have the same priority is arbitrary; however, variables are always initialized before any function calls in the same priority class. When no priority is explicitly given, the object will be initialized after all priority variables/priority functions and subsequently, `main()` will be called as normal.

Priority functions should not return any values or take any arguments. Care must be taken to avoid using uninitialized variables or call functions that might use uninitialized variables. If the project implements multiple functions within the same named section (i.e., `__attribute__((section("example")))`, the Priority attribute can not be applied. Priority levels between functions in the same named section are not possible. Consider creating additional named sections if execution order of priority functions is required.

In LiveUpdate applications, immediately following a partition swap, a priority function can be implemented to re-enable interrupts or set any new variables necessary to immediately enable the control algorithms for the power train before the next Interrupt Service Routine (ISR) is scheduled. Waiting for all variables to be initialized by the CRT may require, on the order of hundreds of microseconds, during which critical ISRs may otherwise be missed. This Priority attribute, applied to functions and global/static variables needed by such functions, provides a way to preempt ordinary data initialization flow.

As in any application, the CRT variable initialization routine is invoked at any device Reset. The CRT is also executed in the case of Live Updates after the boot swap instruction (BOOTSWP) is executed (call to address 0x0). This ensures proper stack initialization, as well as new variable initialization for the upgraded firmware. However, the CRT now calls the function, `_crt_start_mode()`, that checks the soft swap bit in the NVMCON register as a means to determine if ordinary reset initialization of all non-persistent/no load variables should take place, or the more limited subset of variables that are not implicitly or explicitly attributed Preserved.

EXAMPLE 3: PRIORITY ATTRIBUTE

```
int16_t _attribute_((update, priority(100)) foo = -1;
void    _attribute_((priority(100),optimize(1))) CriticalInit(void);
```

The soft swap bit, SFTSWP (NVMCON<11>), is an obvious choice for the return value as this is cleared at device Reset by hardware and set by executing the BOOTSWP instruction. If `_crt_start_mode()` returns, this bit as set. Then, data in the `.rdinit` section will be used by the CRT for initialization; otherwise, the CRT will consume the ordinary data in the `.dinit` table. The `.rdinit` initialization table contains all variables required to be initialized between partition swaps, as well as the addresses of the priority functions. It is recommended to clear the soft swap bit after completion of all initialization functions.

The CRT internally defines `_crt_start_mode()` as a weak function, so including the function shown in Example 4 in the application source code will override it and allow the user to return a value from a different source other than the Soft Swap bit, if desired.

EXAMPLE 4: CRT START MODE

```
int _attribute_((optimize(1))) _crt_start_mode(void)
{
    return _SFTSWP;   // NVMCON<SFTSWP>, bit 11
}
```

CONFIGURING MPLAB® X PROJECT

As mentioned earlier, the `.elf` file from a previous build of the application firmware is used for allocating RAM variables. Depending on the Preserved model used, the compiler/linker will either:

- Preserve only those variables with the Preserved attribute
- Preserve all variables in selected C files
- Preserve all variables in the active project
- Preserve all variables in the entire project (including library archive files)

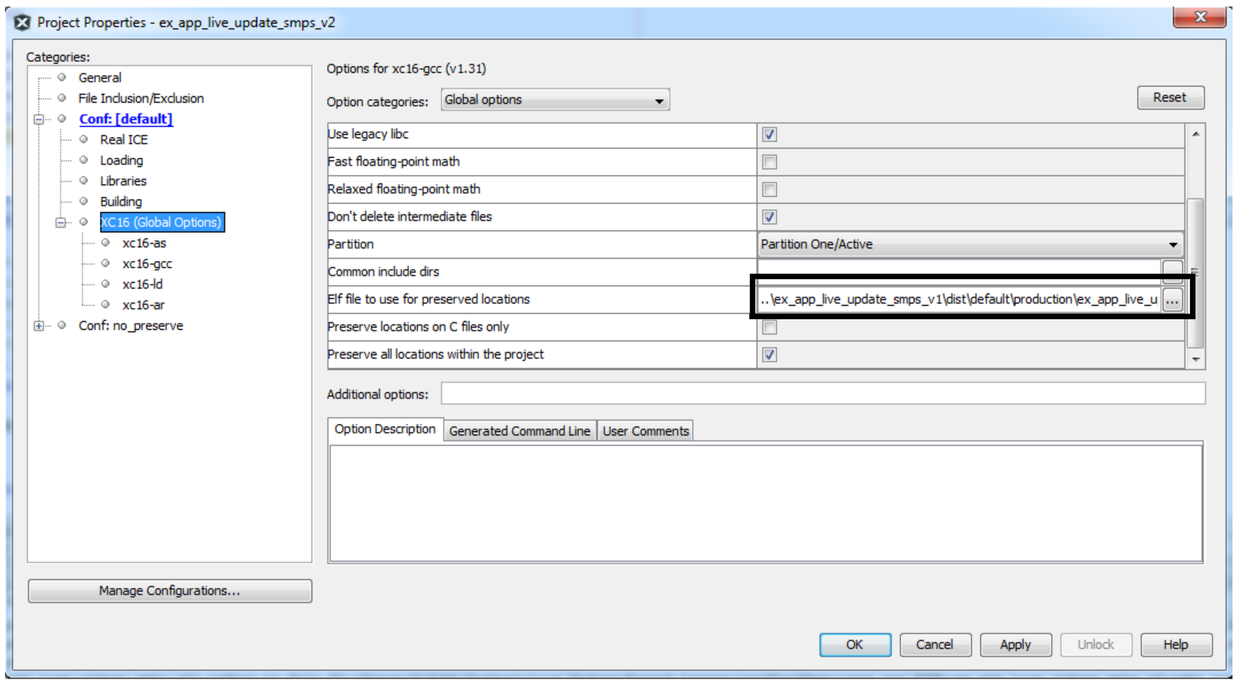
In all four cases, the linker needs information from the previous build to allocate RAM appropriately. To set the `.elf` path for the linker within the active MPLAB X project, right click the project and select **Properties**. Select **XC16 (Global Options)** and scroll down to find the **Elf file to use for preserved locations** entry. Here, you can browse directly to the `.elf` file. The typical location of the `.elf` file (within the MPLAB X project directory) is found here:

.....\<projectName>\dist\default\production\<projectName>.production.elf

Note:	For each firmware release, it is recommended to back up and reference the `.elf` file using a different directory to ensure it does not get inadvertently erased or modified if the source project is cleaned or rebuilt.

In Figure 1, the active project is on version two and the linker receives version one's `.elf` file for preserving variables.

FIGURE 1: PATH TO EXECUTABLE AND LINKABLE FILE


Project Properties - ex_app_live_update_smps_v2

Categories: General, File Inclusion/Exclusion, Conf: [default] (Real ICE, Loading, Libraries, Building, XC16 (Global Options): xc16-as, xc16-gcc, xc16-ld, xc16-ar), Conf: no_preserve

Options for xc16-gcc (v1.31)

Option categories: Global options

Use legacy libc	✓
Fast floating-point math	
Relaxed floating-point math	
Don't delete intermediate files	✓
Partition	Partition One/Active
Common include dirs	
Elf file to use for preserved locations	..\ex_app_live_update_smps_v1\dist\default\production\ex_app_live_u
Preserve locations on C files only	
Preserve all locations within the project	✓

Additional options:

If the Preserve Data model used relies only on those variables marked with the Preserved attribute, then the MPLAB X project does not require any other configurations, other than enabling data sections.

On the same XC16 (Global Options) category in the project properties, right below the `.elf` file selection, there are two check box options for selecting the Preserve Data model. The first option, **Preserve locations on C files only**, tells the compiler that all global and file scope variables in the active C files need to be treated as "Preserved" and allocated to identical addresses as previously used in the `.elf` file. Any new variable in the C files would need to be manually marked with an Update attribute to be excluded from address matching. If new variables are added without the Update attribute, during the linking build stage, the toolchain will issue a warning as these variables cannot be mapped to an equivalent variable in the referenced `.elf` file. This warning means the CRT will leave these variables uninitialized during LiveUpdate and allocate them to an arbitrary unused RAM address.

The second check box option, **Preserve all locations within the project**, is used when as many variables as possible need to be preserved, including ones hidden inside precompiled archives, object files and assembly source files. In both preserve all/entire project options, it is possible that the compiler will be unable to link if additional variables were created and required to be placed in certain memory that is already occupied (such as near space). If this situation arises, **Preserving variables on a C file basis** might be required, along with reinitialization routines.

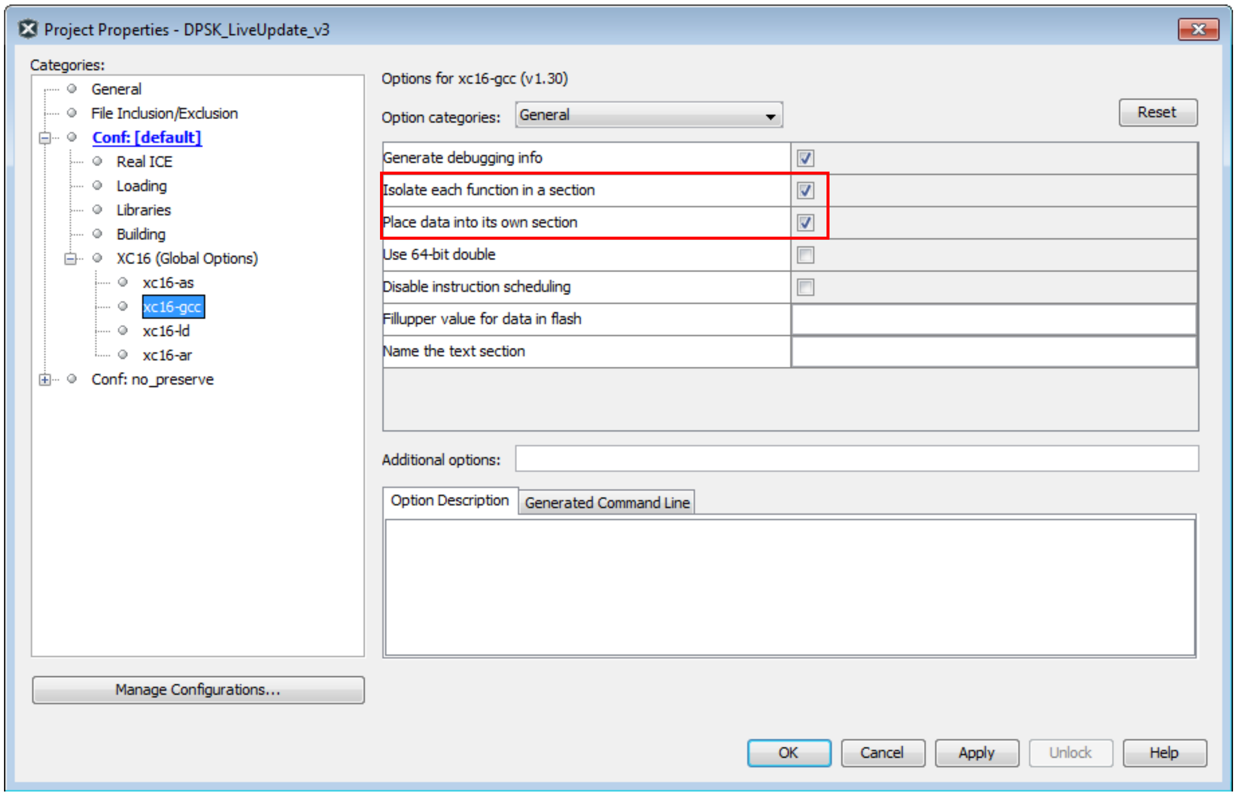
It is possible to select individual C files to have the Preserved Data model rather than the all C files option mentioned above. In the project pane, right click on the individual C file that requires preserving variables. Select **Properties** and click the **Override build options** check box. This will allow configuration on a file basis. Next, select XC16 (Global Options) and scroll down to select the **Preserve locations on C files** box. In this configuration, all file scope and global variables for that particular C file will be preserved between builds.

It is also recommended to enable `-fdata-sections` and `-ffunction-sections` compiler options for all LiveUpdate applications. By default, the compiler will coalesce variables into generalized sections based off the initializer (non-zero value, zero or no initialized data), as well as any attributes that are applied, such as near or persistent. If within one section of grouped variables, the new firmware revision wishes to remove a few variables or an array on N elements, their historically occupied RAM locations often cannot be recovered as the linker operates on section allocation granularity. Additionally, if the firmware update involves extending the width of a variable that has been coalesced in a grouped section, then lack of linker granularity can lead to unrelated variables in the group having to be reallocated to new RAM addresses. This would add copy/reinitialization delay to a LiveUpdate.

For these reasons, it is best to place data objects into their own unique sections. The toolchain will be able to reuse newly freed RAM addresses and avoid unnecessary reallocation of preserved variables that reside adjacent to modified variables. This will add a little extra overhead in the number of Flash instructions, as well as RAM, for variables less than 2 bytes, but will give the best flexibility to the linker while building subsequent program versions. Enabling `-ffunction-sections` has no particular impact on future LiveUpdate restrictions, but allows unreferenced code to be removed from the application when the **Remove unused sections**, `xc16-ld linker` option, is additionally selected. The result is that total code size will often decrease, despite the modest overhead.

In the **Project Properties** window, select **xc16-gcc**, and select both **Place data into its own section** and **Isolate each function in a section** check boxes. See Figure 2 below for project configurations. **Remove unused sections** appears on the **xc16-ld** subcategory.

FIGURE 2: DATA/FUNCTIONS INDIVIDUAL SECTIONS



Options for xc16-gcc (v1.30)

Option categories: General

Generate debugging info	✓
Isolate each function in a section	✓
Place data into its own section	✓
Use 64-bit double	☐
Disable instruction scheduling	☐
Fillupper value for data in flash	
Name the text section	

SOFTWARE IMPLEMENTATION

There are many aspects to creating firmware for LiveUpdate applications, including an intelligent bootloader that can manage code versioning and programming/verifying the Inactive Partition, communication protocol for receiving code images and synchronizing switchover events to coincide with a window without any anticipated exception processing.

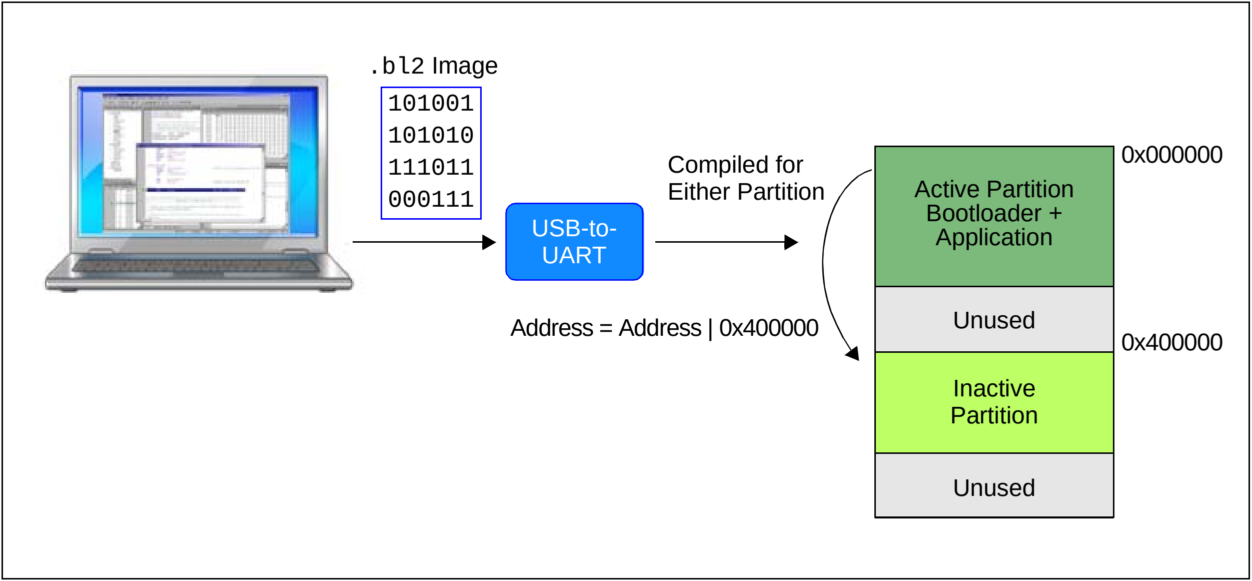
Although bootloader firmware may be specific to a customer implementation, as a starting point for examples, this application note assumes use of the Microchip Easy Bootloader Library (EZBL). For more information on EZBL and for developing a binary image file that fits the format required, visit **Microchip Easy Bootloader**.

When a new firmware image is built and sent to the bootloader over one of many communication protocols, such as UART or I^{2}C, the bootloader's first step is to perform an Inactive Partition erase. These dual partition Flash devices have a dedicated NVM opcode for erasing the Inactive Partition (NVMOP<3:0> (NVMCON<3:0>) = `0b100`). The typical NVM unlock sequence is required to perform any Flash erase/write routines. During this time, interrupts should be temporarily disabled to process the sequential writes to NVMKEY for processing the Inactive Partition erase command. However, during the partition erase time, interrupts are enabled and the application firmware is still fully operational. Interrupts need only be masked for about 12 instruction cycles.

If using EZBL to erase the Inactive Partition, simply call the `EZBL_EraseInactivePartition()` function. This function will configure and write NVMCOM to begin erasing the Inactive Partition. The function is non-blocking (except during the NVMKEY writing sequence) and will return immediately without waiting for the NVM hardware to complete the erase cycle.

After starting Inactive Partition erase, the bootloader can proceed to program and verify the new firmware image. EZBL implements the `EZBL_WriteROMEx()` and `EZBL_VerifyROM()` functions to facilitate this. These functions will block as necessary to complete any outstanding erase/program NVM operations, but background interrupt processing will proceed normally throughout such delays. The Inactive Partition always resides at absolute address 0x400000 and the firmware is always compiled to execute starting from absolute address 0x000000 as the Active Partition. However, to negate the possible difference in `.hex` file load memory addresses targeting absolute Partition 1 or Partition 2 (which have run-time state-dependent addresses), EZBL forces all programming records to the 0x400000 range, such that only the Inactive Partition can be externally manipulated. Additional information, such as information on configuring the device for Dual Boot mode, can be found in **"Dual Partition Flash Program Memory"** (DS70005156) in the *"dsPIC33/PIC24 Family Reference Manual".*

FIGURE 3: BOOTLOADER RESPONSIBLE FOR ADDRESS OFFSET


.bl2 Image
101001
101010
111011
000111
USB-to-UART
Compiled for Either Partition
Address = Address | 0x400000
Active Partition Bootloader + Application
Unused
Inactive Partition
Unused
0x000000
0x400000

Within the new program image header, the bootloader should receive an identification hash or other unique key corresponding to the system hardware that is capable of executing the image. This eliminates the possibility of programing an incorrect code image meant for a different application. Also at this time, the bootloader should be able to determine the firmware version of the incoming code image. The bootloader can make a comparison with the currently active firmware revision and decide to ignore the remainder of the incoming code image or to proceed with the update.

Firmware versioning is crucial for LiveUpdate applications for several reasons:

- Ensures the running application does not perform a partition swap execution handoff to code that was compiled against an incompatible application version with preserved RAM variables at different locations
- Facilitates non-LiveUpdate fallback execution handoff using a software Reset when the new application's major or minor version numbers indicate an incompatible jump forwards or backwards
- Allows building and testing of LiveUpdatable images against a single previous firmware revision, instead of all historical releases
- Improves product maintainability when a simple machine and human readable version number is available in diagnostic reports
- Enables bootloader rejection of application downgrades, which may be an indicator of user error

In the Microchip EZBL LiveUpdate application examples, the identification strings (starting with `BOOTID`) can be found in the `ezbl_dual_partition.mk` file. The Makefile passes the `BOOTID` strings to the build time invoked `ezbl_tools.jar` Java utility that concatenates and compresses the strings into a fixed width SHA-256 hash. Sixteen hash bytes are then appended in the header block of the `.bl2` binary image that is sent to the target device, as well as stored in Flash memory. The APPID versioning numbers are also part of the header block and stored in Flash memory. This approach allows the bootloader to decide to ignore/accept the incoming code image before receiving bytes of the actual code image. This early identification approach mitigates risk by suppressing Inactive Partition erase, which may contain a useful backup application image, whenever a rejection occurs. Additionally, with easily machine compared binary hashes and version numbers, the bootloader passively ignores firmware images presented on shared, broadcast-type communication mediums which are intended for other target devices.

A different approach could be to define constants at a fixed Flash memory address for identification and versioning. In this usage model, the bootloader will need to program the incoming code image into the Inactive Flash Partition, and only after that target address location has been programmed, can the code be validated for the intended target and that version sequencing permits LiveUpdate. This approach may have drawbacks as programming the Inactive Flash Partition occurs before verifying the code image's applicability.

Another important reason for identification and firmware versioning has to do with the Flash boot sequence numbering. Writing to the Inactive Partition's FBTSEQ sequence number should occur after completion of programming and verification of all other Flash locations. With a known good image, if application loses power after writing the sequence number but before the partition swap, the new code image will still be selected when the device powers up. Writing the sequence number can be performed after the partition swap, but this would stall the CPU for several milliseconds as the entire last page of Flash would need to be copied/erased and then programmed. As the FBTSEQ Configuration Word would now reside in the Active Partition, the benefits of the dual Flash partitions would not apply.

It is recommended that after successful programming of the Inactive Partition's Flash, the bootloader determines the Inactive Partition's sequence number by reading the current partition's sequence number and subtracting one. This will always ensure the most recently programmed application is selected at device power-up (the lowest valid sequence number becomes the Active Partition).

When the Inactive Partition is erased, the Inactive Partition's sequence number is set to the invalid erased state of 0xFFFFFF. By not defining a sequence number in the project, this sequence number will be invalid until programmed separately by the bootloader. Any loss of power during the new firmware update event would result in the Active Partition remaining active. This is why it is imperative that the inactive sequence number is programmed separately after all of Flash has been verified. Only after programming the sequence number in the Inactive Partition will the Inactive Partition become active on the next power cycle or Reset.

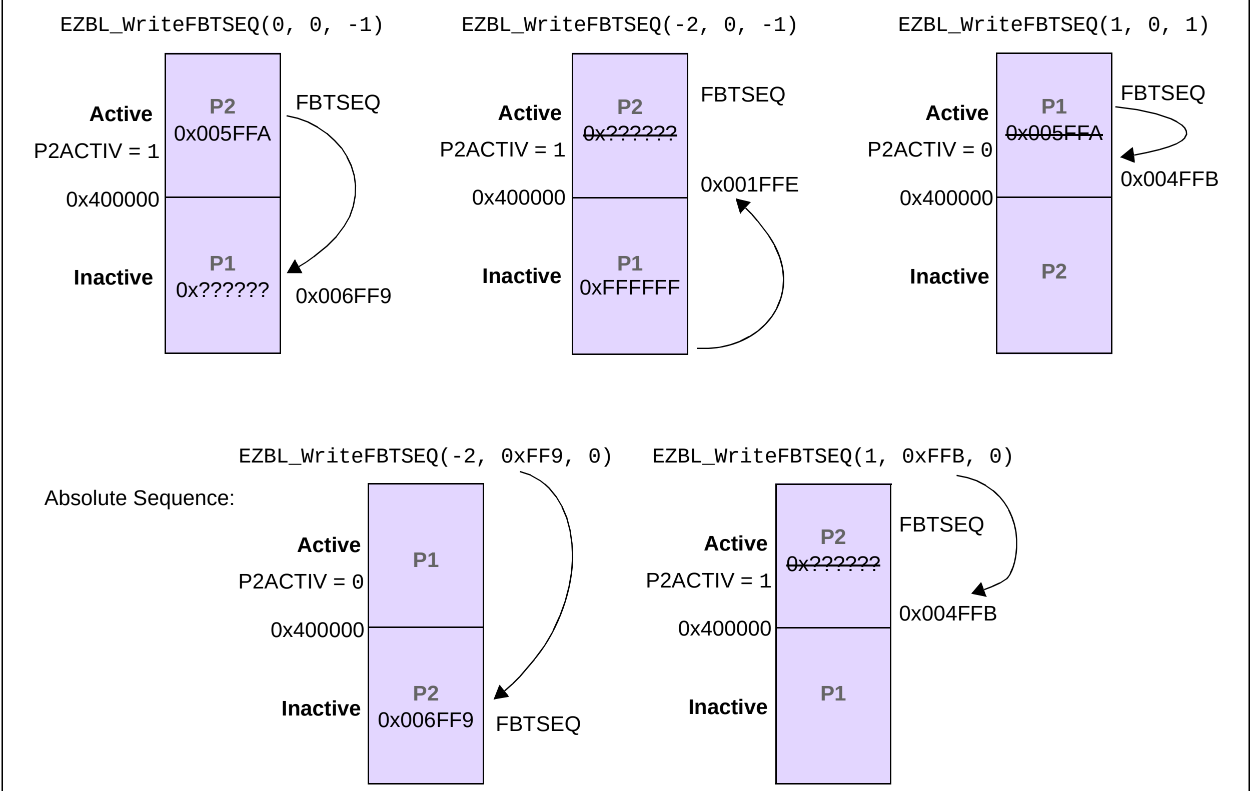
Although the method above is recommended for programming the sequence number, the `EZBL_WriteFBTSEQ()` function supports writing to either partition and supports absolute or relative values for programming. The function accepts three parameters (target partition to be programmed, absolute sequence number and relative sequence to be programmed) and returns the result based on operation. Any return value less than zero indicates a failure. This program boot sequence function will automatically calculate the 12-bit complement of the provided absolute or relative sequence number to ensure a valid 24-bit value is written to the FBTSEQ Configuration Word.

The target partition parameter chooses the FBTSEQ Word on the Active or Inactive Partition (1/0 respectively). As the most common implementation will be to always target the Inactive Partition, this parameter would be set to 0. However, the parameter could alternatively be set to -1 or -2, which indicates absolute Partition 1 or Partition 2, respectively. For programming the sequence number with this option, the state of the P2ACTIV bit is decoded to determine the address where Partition 1 or 2 currently resides.

The absolute sequence parameter can be used if it is desired to pass in the sequence number. Whatever 12-bit value is passed in will be written to the specified target partition's sequence number. For this option, the relative sequence parameter should be 0.

The relative sequence number is actually a signed integer to be added to the non-targeted partition's sequence number to determine the targeted partition's sequence number. If this input parameter is negative (i.e., -1) and the target partition parameter is 0, the Inactive Partition's sequence number will become one less than the currently Active Partition's sequence number. At any subsequent Reset, this would cause the currently Inactive Partition to be selected for execution. For this option, the absolute sequence parameter should be zero. Figure 4 provides several examples supported by the `EZBL_WriteFBTSEQ()` function.

FIGURE 4: FLASH PARTITION SEQUENCE WRITE EXAMPLES(1,2,3)



Note 1: '0x??????' represents any existing FBTSEQ contents, including the invalid 0xFFFFFF erased state.

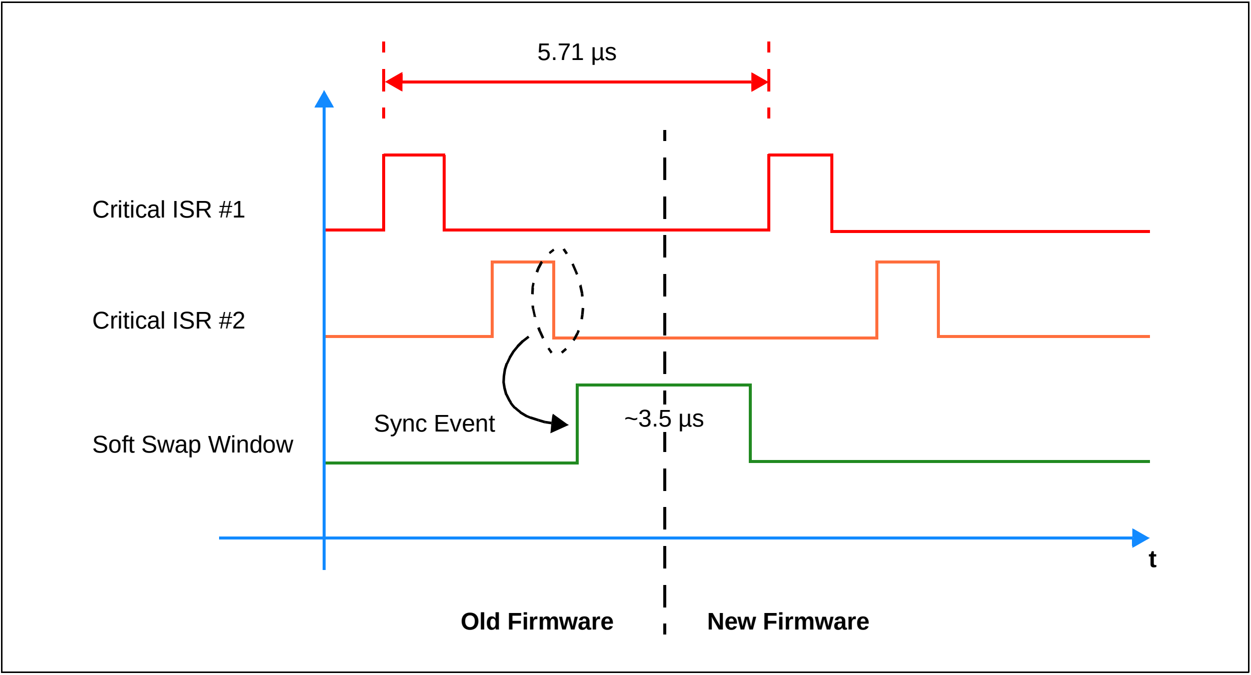
2: If it is foreseeable that there will be more than 4095 firmware updates, special care must be taken to properly reset both sequence numbers. As this seems to be an unlikely event, `EZBL_WriteFBTSEQ()` does not contain the code overhead to handle this case. Two calls to `EZBL_WriteFBTSEQ()` with absolute sequence numbers can be used to perform such a Reset.

3: Writes to the Active Partition's sequence number, when not in the erased state, will require an erase of the entire Flash page where the Configuration Words reside. The Flash page will be stacked, erased and restored, except with the updated sequence number. This requires interrupts to be disabled and is not optimal for LiveUpdate applications as the page erase and writes will stall CPU execution. It is recommended to only modify the Inactive Partition's sequence number.

SWITCHOVER TIMING

The most important requirement for online firmware updates is to have a seamless transition between firmware revisions. This means the power supply output or motor drive should be uninterrupted during the firmware update. A key piece to developing firmware for LiveUpdate applications is the ability to synchronize to a timing window, where no critical ISRs are executing, in order to initiate the soft swap event. Lower priority functions/interrupts may be disabled for a longer period of time without impacting the main application performance. See Figure 5 for an example of a switchover timing window.

FIGURE 5: SWITCHOVER TIMING WINDOW



In Figure 5, two critical ISRs are required for the power train and all other application interrupts are disabled until after the partition swap. Each ISR is executed at a frequency of 175 kHz and with a process time around 1 µs, leaving a 3.5 µs window for performing the partition swap and executing priority reinitialization functions. Note that the partition swap is synchronized to this timing window to allocate the maximum amount of time before the next critical ISR is scheduled to execute.

The application firmware is responsible for synchronizing the partition swap with critical ISR #2 in this example. Once the Inactive Partition has been programmed and verified, firmware disables all non-critical ISRs and clears a software flag that is only set within the critical ISR #2. Only after the software flag is set, the partition swap function can be executed. Another approach would be to clear the software flag, and from within the critical ISR #2, call the partition swap function. As this would add conditional statements inside a high execution rate loop, the former implementation is more desirable. Without some type of synchronization routine/flag for the switchover event, critical ISRs could be missed which could impact operation of the application.

The majority of the critical switchover timing depends on the application requirements for critical variable initialization and reinitialization type functions.

There is, however, a fixed timing requirement that is non application-specific which includes device initialization, such as Stack Pointer configuration and general setup of registers for the processor state. During this switchover time interval, the following would typically occur:

1. Disable all interrupts.
2. Execute NVM unlock sequence and `BOOTSWP` instruction.
3. Jump to address 0x0 which is the entry point to the new application now present in the Active Partition addressing space.
4. XC16 Compiler Run Time (CRT) will initialize stack, Stack Pointer limit and CORCON (if required).
5. Check soft swap event and start initializing priority variables and calling priority function.
    a) Execute user-defined functions that initialize critical variables and re-enable critical interrupts

After time-critical interrupts are enabled, the CRT will continue with initialization of non-prioritized update variables and eventually call `main()`.

Example 5 shows a simple macro that executes the partition swap sequence. This macro should be executed inside the main function and disables interrupts by using the Global Interrupt Enable bit, GIE (INTCON2<15>. In order to re-enable interrupts after the partition swap, the new code image needs to set the GIE bit appropriately.

EXAMPLE 5: PARTITION SWAP SEQUENCE MACRO

```
#define MACRO_PARTITIONSWAP() __asm__ volatile(" bclr INTCON2, #15 \n"  \
                                              " nop     \n"            \
                                              " nop     \n"            \
                                              " clr  W0 \n"            \
                                              " mov  #0x0055, W1 \n"   \
                                              " mov  W1, NVMKEY  \n"   \
                                              " mov  #0x00AA, W1 \n"   \
                                              " mov  W1, NVMKEY  \n"   \
                                              " bootswp \n"            \
                                              " call W0"  : : : "w0", w1", "memory")
```

The instruction immediately after the BOOTSWP instruction must be a single 24-bit instruction that resides on the currently Active Partition, but branches to an address in the newly Active Partition. In a typical application, address 0x0 contains a GOTO instruction that points to the address of a C run-time entry function. In this case, a register direct call to address 0x0 after BOOTSWP is recommended. This instruction satisfies the 24-bit size restriction and zero-extends the target address as an absolute quantity (i.e., not PC relative). CALL is preferred instead of GOTO as it ensures the Stack Frame Active SR bit is cleared.

The Easy Bootloader Library includes the EZBL_PartitionSwap() function that disables all interrupts (by clearing all IECx registers), manages the NVM unlock for the BOOTSWP instruction and executes the jump to address 0x0.

Although it is recommended to perform the switchover event at the main() function level, the EZBL_PartitionSwap() function implementation allows the partition swap to occur inside any level interrupt routine. This function overrides the stacks' return address to point to 0x0 and executes a Return from Interrupt instruction (RETFIE), which automatically resets the IPL state. Even though all interrupts are disabled by the EZBL_PartitionSwap() function, it is recommended to disable all non-critical interrupts prior to calling this function to ensure lower priority interrupts do not interfere with synchronization to the earliest opportunity within the soft swap window.

More information on non application-specific switchover timing is provided throughout this application note. Once the CRT initializes the core compiler environment SFRs (i.e., W15 Stack Pointer, SPLIM, DSRPAG, etc.), all priority variables and functions will be initialized/executed. From this point on, critical priority functions and variables decide the application level switchover time. This is highly dependent on what the application deems critical to maintain proper operation. Obviously, the more variables to be initialized and critical routines to be executed, the longer the complete application switchover time will grow. If the application requires major updates, which cannot fit in the available switchover window, multiple piecewise firmware updates can be performed sequentially or the bootloader can fall back to a non-LiveUpdate Reset after programming the new code image.

As all priority functions get executed in both a cold start and soft swap event, additional software, such as a conditional statement that tests the soft swap bit, may be required. Adding this conditional statement to main avoids reinitialization of the microcontroller peripherals/ clock upon main function entry and allows any remaining initialization routines for the soft swap condition to be satisfied to be executed (Example 6).

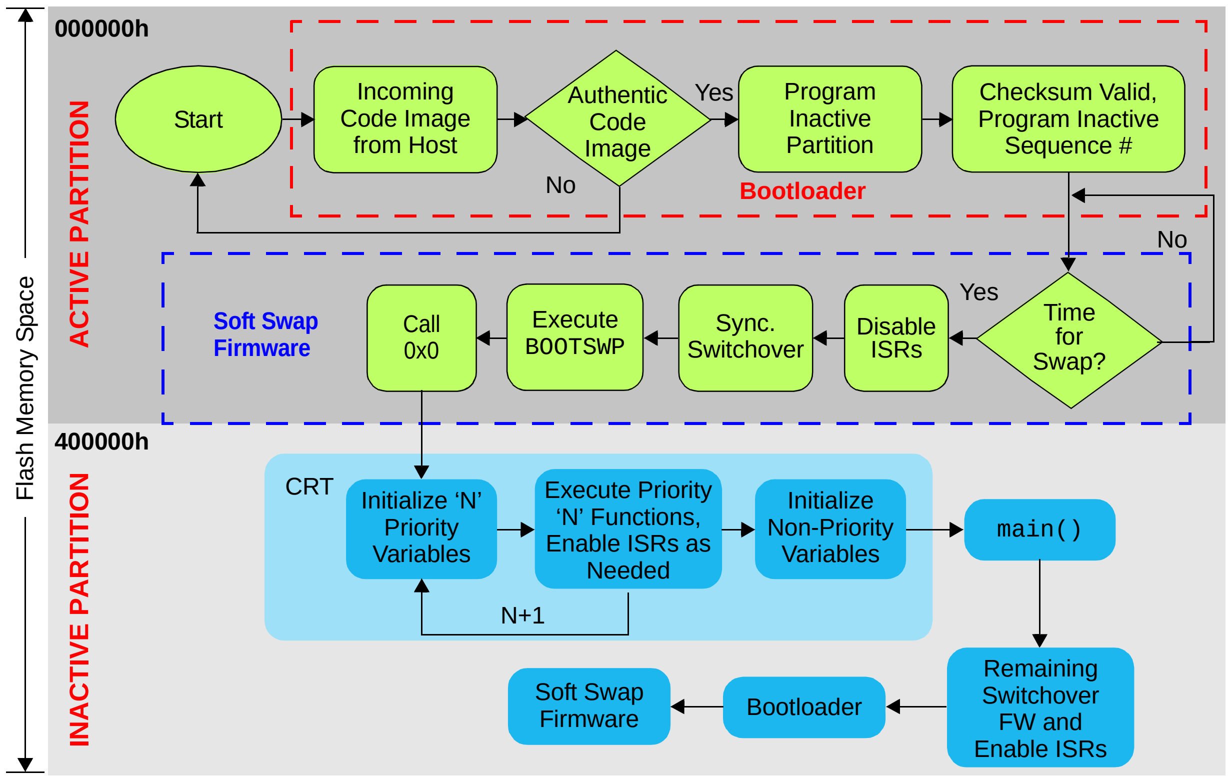
Note: It is important to remember that firmware that is executed on a soft swap may also need to be executed from a cold start the next time the application powers up (i.e., peripheral initialization).

EXAMPLE 6: PRIORITY FUNCTION IMPLEMENTATION

```
void __attribute__((priority(x)) FunctionInit(void) {
     if(!_SFTSWP) {
     // Priority X cold start initialization function
        return;
     }
     // Priority X soft swap initialization function
}
```

The block diagram shown in Figure 6 shows the overall steps discussed above with respect to executing a LiveUpdate event.

FIGURE 6: EXECUTION OF LiveUpdate EVENT(1)


Flash Memory Space
000000h
ACTIVE PARTITION
Start
Incoming Code Image from Host
Authentic Code Image
Yes
No
Program Inactive Partition
Checksum Valid, Program Inactive Sequence #
Bootloader
No
Yes
Time for Swap?
Disable ISRs
Sync. Switchover
Execute BOOTSWP
Call 0x0
Soft Swap Firmware
400000h
INACTIVE PARTITION
CRT
Initialize 'N' Priority Variables
Execute Priority 'N' Functions, Enable ISRs as Needed
N+1
Initialize Non-Priority Variables
main()
Remaining Switchover FW and Enable ISRs
Bootloader
Soft Swap Firmware

Note 1: During the CALL instruction, after execution of the BOOTSWP instruction, the two partitions swap memory locations, making the Inactive Partition the newly Active Partition.

SOFTWARE RESTRICTIONS

As a high-level language, C source files support a number of programming constructs that can lead to ambiguities when attempting to preserve variable addresses in a current project relative to a historically built project executable image. With the slightest of source content, header include, file renaming or optimization changes, it can be impossible for the toolchain to prove if one variable seen in the current project is equivalent to an older one or not. This section covers some potential problems and manual solutions necessary to build a correct, coherent LiveUpdate application.

Note:	To minimize future LiveUpdate application development effort, avoid using static local variables and ensure static file scope variables are uniquely named throughout the project.

Static Variables

Global variables are naturally required to be uniquely named as these variables are used across multiple files, so they generally can be preserved and automatically handled by XC16. Likewise, auto-storage class objects, such as a typical function's local variables, are allocated temporarily on the run-time stack, making them non-persistent and not a problem for LiveUpdate (assuming they are out-of-scope at the time of the partition swap).

However, statically allocated variables at file scope and function local scope need not have a unique name across the project or within a source file. For these variables, the compiler places them in unique, but unpredictably named sections. For example, static local variables, `cnt` or `i`, can reside locally within several different functions. The compiler will allocate each of these reoccurring variables into uniquely named sections (if not explicitly attributed) by appending a storage qualifier token and a four-digit decimal number (`.nbss.cnt.1234`). Because of the unpredictability of these section names, which may change as the code and compiler optimization change, preserving static local and file scope variables is quite difficult.

If a static local variable accidentally remained in the project, and preserving that variable is required, the linker may emit a warning or error while building the new LiveUpdate application. To solve such a problem, the variable's declaration needs to be decorated with an address attribute to force allocation to the correct address. Example 7 forces the local variable, `bootUnlockState`, to an address of 0x0000105E.

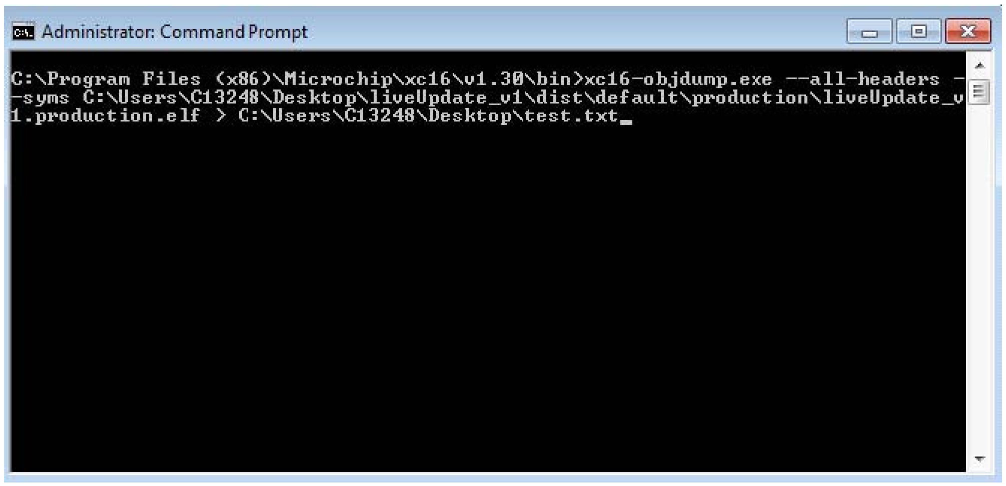
EXAMPLE 7: FIXED ADDRESS VARIABLE

```
static char __attribute__((preserved, address(0x0000105E)))
bootUnlockState = 0x00;
```

Determining the address of the equivalent historical variable is not straightforward as the symbol name for these types of variables is not always shown in the map file. XC16 does offer an object dump executable that accepts an `.elf` file, which typically contains more information than the map file.

Using the command window, change the current directory to the XC16 installation folder. For the default installation folder, the path would be as shown in Figure 7.

FIGURE 7: COMMAND PROMPT FOR GENERATING OBJECT DUMP FILE

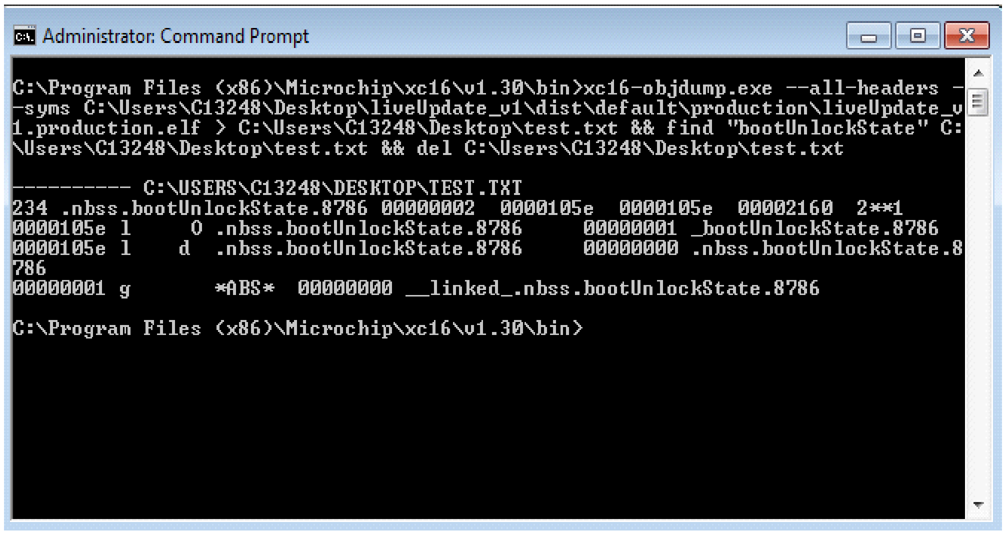


```
C:\Program Files (x86)\Microchip\xc16\v1.30\bin>xc16-objdump.exe --all-headers -
-syms C:\Users\C13248\Desktop\liveUpdate_v1\dist\default\production\liveUpdate_v
1.production.elf > C:\Users\C13248\Desktop\test.txt_
```

The object dump executable (`xc16-objdump`) is located in the **bin** folder. Specifying the compiler to dump all headers (**--all-headers**) and symbol (**--syms**) information is typically sufficient for finding variable information. For more options supported by the executable, add **--help** after the `xc16-objdump` executable. The path to the input file is needed to complete the object dump, which would be the previous firmware build's `.elf` output artifact. Lastly, stdout is redirected to a temporary text file to view and search the object information. In the example below, this is saved to `test.txt` on the Desktop.

It is possible to search the text file for the symbol name, as shown in Figure 8, and have the address information displayed in the command prompt instead of opening the `.txt` file and searching manually. If the text file isn't required to be saved for further use, it can be deleted as well from the command prompt.

FIGURE 8: COMMAND PROMPT FOR FINDING ADDRESS LOCATION BY VARIABLE NAME



```
C:\Program Files (x86)\Microchip\xc16\v1.30\bin>xc16-objdump.exe --all-headers -
-syms C:\Users\C13248\Desktop\liveUpdate_v1\dist\default\production\liveUpdate_v
1.production.elf > C:\Users\C13248\Desktop\test.txt && find "bootUnlockState" C:
\Users\C13248\Desktop\test.txt && del C:\Users\C13248\Desktop\test.txt

---------- C:\USERS\C13248\DESKTOP\TEST.TXT
234 .nbss.bootUnlockState.8786 00000002  0000105e  0000105e  00002160  2**1
0000105e l     O .nbss.bootUnlockState.8786     00000001 _bootUnlockState.8786
0000105e l    d  .nbss.bootUnlockState.8786     00000000 .nbss.bootUnlockState.8
786
00000001 g       *ABS*  00000000 __linked_.nbss.bootUnlockState.8786

C:\Program Files (x86)\Microchip\xc16\v1.30\bin>
```

The first row of this pruned dump is a section definition from the `.elf` headers, whereas the remaining lines are from the symbol table. The second column contains single character flags for the symbols, where 'O' indicates a data Object located in RAM or PSV const memory. The symbol's address, located in the first column, is generally the information needed when attempting to restore the address of a preserved local static variable that cannot be automatically mapped by the toolchain.

POINTERS

Pointers allocated globally or statically in RAM often need special consideration for LiveUpdate applications. As Flash contents do not maintain preserved addresses under any model, any pointers to Flash, such as callback functions, const/PSV data array and const string pointers will need to be reinitialized in the new firmware before use. Examples of this are shown in **"Application Examples"**.

In most instances, it is assumed that a preserved RAM Pointer will point to RAM data objects that have also been preserved. Alternatively, the pointer may point to an SFR, which is intrinsically hardware preserved. Therefore, these pointers may not require LiveUpdate reinitialization. If, for any reason, the contents to which the pointer points to are moved, as in not preserved or simply relocated due to an increase in the number of array elements or the size of each element, the pointer, along with the new RAM contents, will need to be initialized upon a LiveUpdate event.

If the application has RAM space to preserve the old contents and pointer along with the new variables, it is possible to calculate the proper offset and apply it to the new RAM contents, as shown in Figure 9.

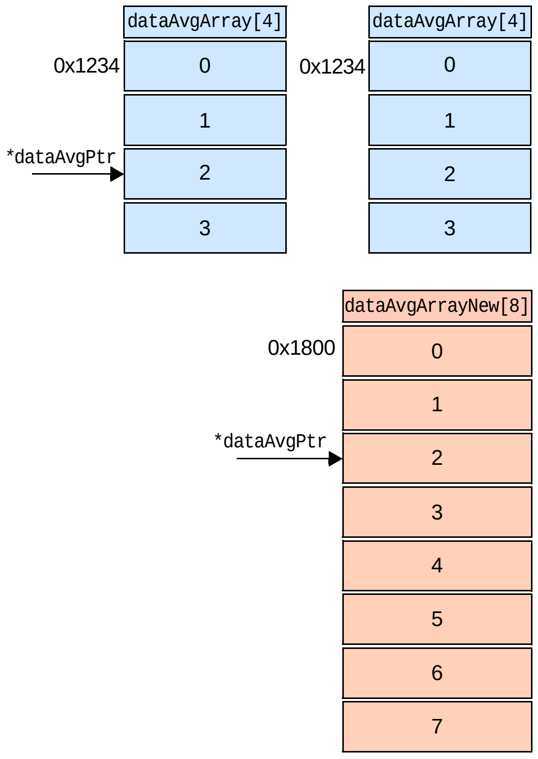
FIGURE 9: POINTER/RAM INITIALIZATION[1,2,3]

V1 Firmware:

```
uint16_t dataAvgArray[4];
uint16_t *dataAvgPtr = dataAvgArray;
```

V2 Firmware (Preserve-All model):

```
uint16_t __attribute__((persistent)) dataAvgArray[4];
uint16_t __attribute__((persistent)) *dataAvgPtr;
uint16_t __attribute__(update, persistent)) dataAvgArrayNew[8]
```



```
void __attribute__((priority(100))) ArrayInit(void)
{
   uint16_t i;

   if(!_SFTSWP)        // Ordinary reset
    {
      dataAvgPtr = dataAvgArrayNew;
      return;
   }

   // LiveUpdate - copy existing array data
   for(i = 0; i < sizeof(dataAvgArray)/sizeof(dataAvgArray[0]); i++)
      dataAvgArrayNew[i] = dataAvgArray[i];

   // Determine new pointer target
   dataAvgPtr = &dataAvgArrayNew[((unsigned int)dataAvgPtr -
(unsigned int)dataAvgArray)/sizeof(dataAvgArray[0])];
}
```

Note 1: `dataAvgArray[]`'s declaration is preserved in V2 to permit named access to the existing LiveUpdate contents and ensure the allocated RAM is not clobbered by unrelated new (update) variables. A V2 to V3 LiveUpdate project could delete the array to recover its RAM.

2: The pointer and both arrays have the Persistent attribute to prevent implicit CRT initialization with zeros. Without persistence, `dataAvgArray[]`'s cold start initializer would waste code space, `dataAvgPtr` would be catastrophically zeroed after the `ArrayInit()` priority function set the correct value (cold start-up only), and `dataAvgArrayNew[]`, being an update variable, would be catastrophically zeroed after `ArrayInit()` returned during the LiveUpdate event.

3: `dataAvgPtr`'s value is computed using array indexing and integer arithmetic as this allows the target data type to be redeclared using a different data type if needed. For example, `*dataAvgPtr` could be changed to a (`uint32_t *`), `dataAvgArrayNew[]` to 8 `uint32_t`s and `dataAvgPtr == &dataAvgArray[1]` would convert to `dataAvgPtr = &dataAvgArrayNew[1]` without causing misalignment or a type conversion error.

Instead of incrementing a static pointer to access array contents, the array can be indexed by a temporary calculation local to the scope the memory is accessed by or a separate index variable (offset) can be used that can always be preserved. In some instances, the firmware can be written to ensure the offset is always at zero, base of the array, before executing the partition swap. This would ensure that reinitializing the pointer to the new base would always point to the appropriate index in the new firmware version. These are simply just suggestions, as there are numerous ways to handle pointer reinitialization if the application requires updating pointers and/or RAM contents that the pointer points too. When working with application libraries, many of the same software restrictions apply, but now it is even more crucial to follow these recommendations. This is due to the fact that the library may not always be recompiled, which is especially true if owned by a third party. If possible, it is recommended to enable `-fdata-sections` while compiling the library project to ensure the finest variable manipulation granularity while linking LiveUpdate projects.

Table 1 summarizes the complexity for different LiveUpdate scenarios when the Preserve All model is used.

TABLE 1: LiveUpdate SCENARIOS

Level of Firmware Change	Examples	Register and RAM Variables	New Firmware Requirement
Simple	Change to Firmware Constants • Compensator coefficients • Lookup table • Fault threshold limits, etc.	No new variables or SFR changes	Loading new constants to variable locations if not stored in Flash
	Addition/deletion of firmware	—	—
	Deletion of variables	Variables orphaned	—
	Local static variable	—	Adding address attribute to variable to properly map
Moderately Low	New variables (non-critical, tagged with Update attribute)	New variables created	New variable initialization if marked persistent and used in priority function, otherwise CRT initializes
	Reduce number of elements in an array	Variables orphaned, possibly reuse location	May need to reset index
	Modification to peripherals	SFRs modified	Peripheral initialization function supports both LiveUpdate and cold start
	Pointers to functions, const strings and PSV constants in Flash	—	Reinitialization of pointers
Moderate	New variables (timing-critical, tagged with Update attribute)	New variables created	High-priority function initializing variables
	Increase number of array elements or modify structure (i.e., add/remove/reorder members) – non timing-critical	Additional array/structure required with new sizing while preserving old	Variable remapping function(s), offset calculation firmware, need to update pointers
	Timing-critical firmware to be executed once after soft swap	New variable(s)	Application-specific
Complex	Timing-critical firmware requiring variable remapping: • New compensator algorithm • Increase array elements or modify structure	Additional array/structure required with new sizing while preserving old	Variable remapping function(s) and new variables requiring initialization during high-priority functions
	Changing data type but retaining same size	—	Firmware to correct data type

Application Examples

DIGITAL POWER STARTER KIT EXAMPLE

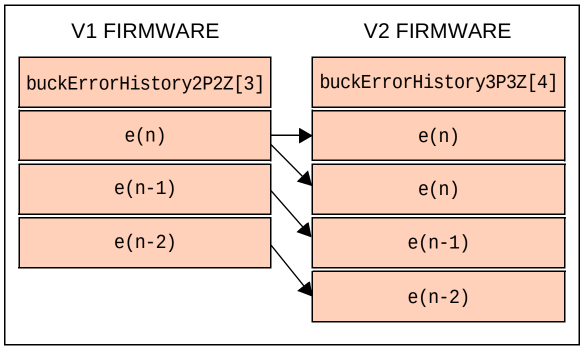
For illustrating the different Preserve Data models and how online firmware updates are possible, three different firmware versions were created and carried out on the Digital Power Starter Kit (DPSK). For details on the DPSK, please visit the **Intelligent Power** page. The source code for these projects is distributed in the EZBL download. Version 1 firmware is the initial starting point, which contains Voltage mode control of both buck and boost converters, with control algorithms executing at 175 kHz, software task scheduler that calls bootloader related functions, LCD display functions, Fault management, load transient software via I^{2}C and UART communication software.

In this example, the buck converter 2P2Z compensator performance is purposely designed to be unstable. Observing the transient response by connecting an oscilloscope probe (AC Coupled) to the buck converter's output voltage test point (TP5) will show the sub-optimally designed compensator. The first online firmware update event will be to improve the buck converter compensator performance, as well as adding additional software features, such as calculating the switchover time and additional LCD display options for showcasing timing events.

The first LiveUpdate event will use the Preserve All model (preserve all locations within the project check box). All new variables will be marked with the Update attribute and Priority attribute (if required). In V2 firmware, the 2P2Z compensator is replaced by a properly designed 3P3Z compensator. There will need to be a priority initialization function for the 3P3Z compensator that is part of the critical timing path, as the buck converter output has to remain stable during firmware transition. The coefficients for the 2P2Z compensator have been deleted, but the control history and error history arrays are needed for initializing the 3P3Z compensator arrays. Figure 10 shows how data from the 2P2Z compensator error history is copied from one RAM location to the 3P3Z compensator array location and Example 8 shows the firmware that initializes the new compensator immediately following the LiveUpdate partition swap.

Note: In a LiveUpdate application, some initialization routines called in the critical path may also need to be executed from a cold start. Since some variables may not be initialized yet, the same initialization routine may need to be called elsewhere in the software (i.e., `main()`).

FIGURE 10: RAM MAPPING BETWEEN CODE VERSIONS



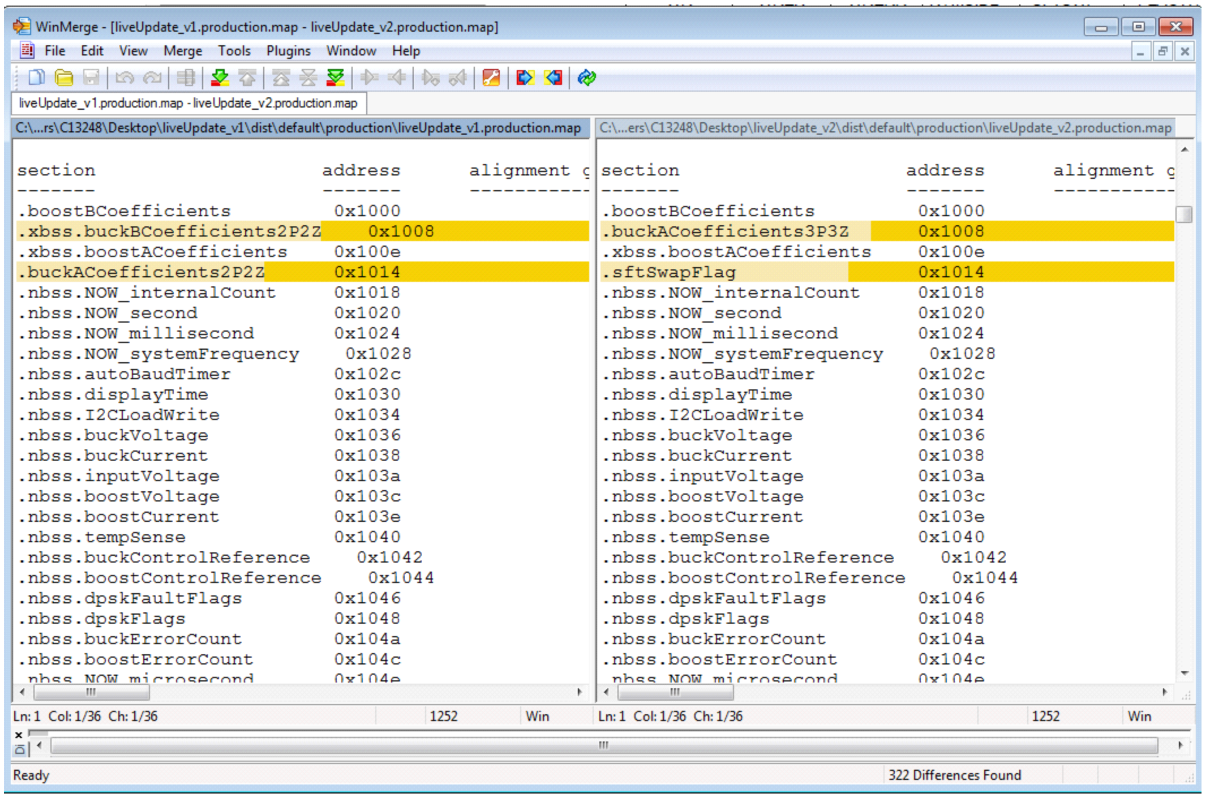
EXAMPLE 8: CRITICAL INITIALIZATION FUNCTION[1]

```
void __attribute__((priority(100), optimize(1))) CriticalInit(void) {

    if(!_SFTSWP) {                // Return if cold start
            sftSwapFlag = 0;
            return;
    }

    sftSwapFlag = 1;              // Flag initialized at this priority level not by CRT

    // Reinitialize anything time critical
    InitBuckComp();               // Load new compensator parameters and init W-Registers

    // Load control/error history - extrapolating values as needed and copying from the old
    // variable
    buckControlHistory3P3Z[0] = buckControlHistory2P2Z[0];
    buckControlHistory3P3Z[1] = buckControlHistory2P2Z[0];
    buckControlHistory3P3Z[2] = buckControlHistory2P2Z[1];

    buckErrorHistory3P3Z[0] = buckErrorHistory2P2Z[0];
    buckErrorHistory3P3Z[1] = buckErrorHistory2P2Z[0];
    buckErrorHistory3P3Z[2] = buckErrorHistory2P2Z[1];
    buckErrorHistory3P3Z[3] = buckErrorHistory2P2Z[2];

    // Enable compensator ISR(s) now that the buck compensator has been reinitialized
    _ADCAN1IE = _ADCAN3IE = 1;
}
```

Note 1: The attribute, `priority(100)`, is assigned instead of `priority(1)`, even though this function is considered the most time-sensitive code in the project. By leaving numbering gaps between priority classes, it is possible to implement some other even more time-critical code in the future without having to come back to this function and move it to a lower priority/higher number.

Using a file compare tool, such as WinMerge or Beyond Compare, the output map files for Version 1 and Version 2 can be examined. As all data is placed in individual sections, some variables from the V1 firmware have been orphaned and those locations have been reused by V2 firmware. In Figure 11, the output map files show that the new 6-element array for the 3P3Z A coefficients resides where the 6-element array of the 2P2Z B coefficients were.

FIGURE 11: VARIABLE MAPPING PRESERVE ALL MODEL


WinMerge - [liveUpdate_v1.production.map - liveUpdate_v2.production.map]
File
Edit
View
Merge
Tools
Plugins
Window
Help
liveUpdate_v1.production.map - liveUpdate_v2.production.map
C:\...rs\C13248\Desktop\liveUpdate_v1\dist\default\production\liveUpdate_v1.production.map
C:\...ers\C13248\Desktop\liveUpdate_v2\dist\default\production\liveUpdate_v2.production.map
Ln: 1 Col: 1/36 Ch: 1/36
1252
Win
Ready
322 Differences Found

```
section                        address     alignment g
-------                        -------     -----------
.boostBCoefficients              0x1000
.xbss.buckBCoefficients2P2Z         0x1008
.xbss.boostACoefficients         0x100e
.buckACoefficients2P2Z           0x1014
.nbss.NOW_internalCount          0x1018
.nbss.NOW_second                 0x1020
.nbss.NOW_millisecond            0x1024
.nbss.NOW_systemFrequency          0x1028
.nbss.autoBaudTimer              0x102c
.nbss.displayTime                0x1030
.nbss.I2CLoadWrite               0x1034
.nbss.buckVoltage                0x1036
.nbss.buckCurrent                0x1038
.nbss.inputVoltage               0x103a
.nbss.boostVoltage               0x103c
.nbss.boostCurrent               0x103e
.nbss.tempSense                  0x1040
.nbss.buckControlReference         0x1042
.nbss.boostControlReference           0x1044
.nbss.dpskFaultFlags             0x1046
.nbss.dpskFlags                  0x1048
.nbss.buckErrorCount             0x104a
.nbss.boostErrorCount            0x104c
.nbss.NOW_microsecond            0x104e
```

```
section                        address     alignment g
-------                        -------     -----------
.boostBCoefficients              0x1000
.buckACoefficients3P3Z           0x1008
.xbss.boostACoefficients         0x100e
.sftSwapFlag                     0x1014
.nbss.NOW_internalCount          0x1018
.nbss.NOW_second                 0x1020
.nbss.NOW_millisecond            0x1024
.nbss.NOW_systemFrequency          0x1028
.nbss.autoBaudTimer              0x102c
.nbss.displayTime                0x1030
.nbss.I2CLoadWrite               0x1034
.nbss.buckVoltage                0x1036
.nbss.buckCurrent                0x1038
.nbss.inputVoltage               0x103a
.nbss.boostVoltage               0x103c
.nbss.boostCurrent               0x103e
.nbss.tempSense                  0x1040
.nbss.buckControlReference         0x1042
.nbss.boostControlReference           0x1044
.nbss.dpskFaultFlags             0x1046
.nbss.dpskFlags                  0x1048
.nbss.buckErrorCount             0x104a
.nbss.boostErrorCount            0x104c
.nbss.NOW_microsecond            0x104e
```

Further examination of the map files shows locations where other new variables reside and shows that all variables in the V2 firmware were correctly mapped to V1 firmware.

STEPS TO EXECUTE FIRST LiveUpdate

To execute this first LiveUpdate event, first program the DPSK with Version 1 firmware using the PICkit™ On-Board (PKOB) or other programming tools, such as MPLAB REAL ICE™ or MPLAB ICD. Connect an MCP2221 USB-to-UART breakout board (jumpered for 3.3V) to connector J1 on the DPSK. The three connections required are TX, RX and GND. These three pins (1, 3 and 6) are mapped one-to-one from the breakout board to the DPSK board. Connect the mini-USB cable to your PC and determine the COM port the USB device enumerated with.

Control Panel → Device Manager → Ports (COM & LPT)

For more information on the breakout board, visit the **MCP2221 Breakout Module** page. The port information will be required for executing the LiveUpdate event with this implementation.

The application, when running, should display the buck/boost output voltage and load settings. Pressing switch SW1 will toggle the LCD display and show input voltage and temperature readings, as well as the Active Partition. With a scope probe on the output of the buck converter, the transient response can be observed. The Active Partition display is based on the setting of the P2ACTIV bit in the NVMCON Special Function Register. This bit, when read, will distinguish which partition is the Active Partition.

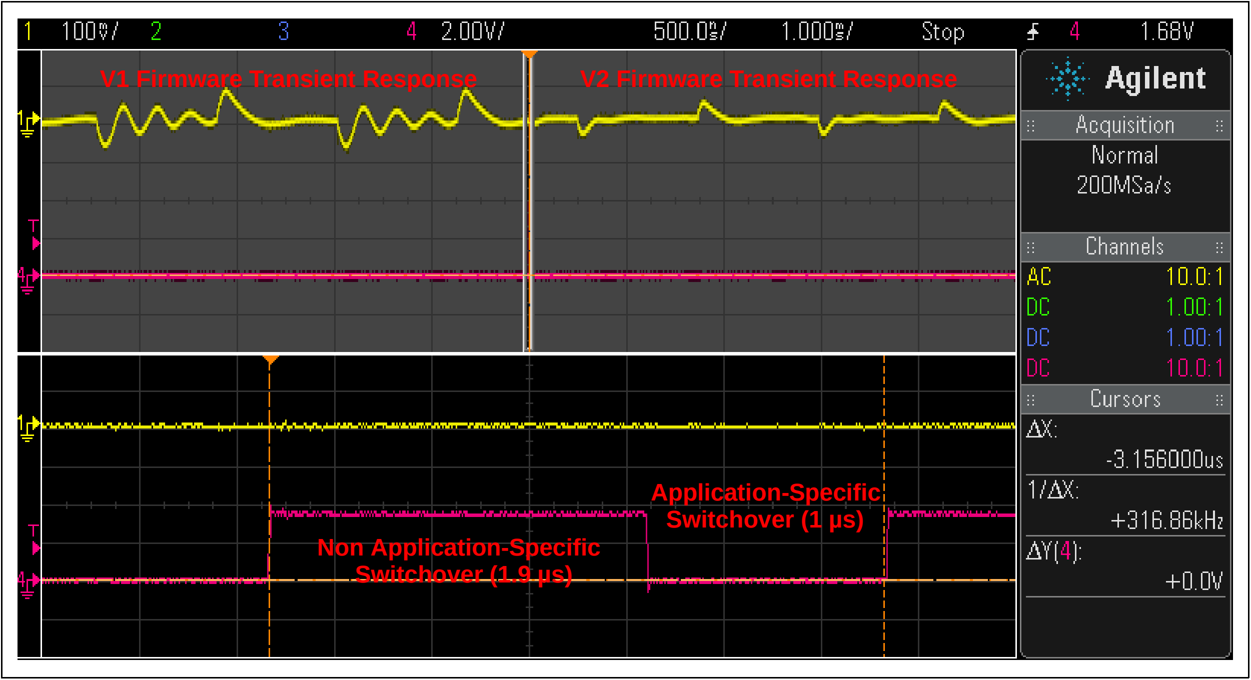
Next, open Version 2 firmware using MPLAB X IDE and navigate to the `ezbl_dual_partition.mk` Makefile, which can be found under important files. Open the file and change the communication port to match the port that the USB enumerated with.

--communicator -com=COMX

Building V2 firmware will invoke the `ezbl_tools.jar` application, which will convert the `.elf` artifact output to a binary `.bl2` file and send this data out over the COM port to the dsPIC33EP64GS502 target device. V1 firmware will receive the UART data and program V2 firmware to the Inactive Partition. After all data has been programmed and verified, V1 firmware initiates the `BOOTSWP` event.

With an oscilloscope probe connected to the buck converter output (TP5), and another scope probe connected to pin 5 of J3, the switchover event can be captured. Figure 12 demonstrates this LiveUpdate event and shows that the buck converter output was unaffected during the switchover event.

FIGURE 12: SEAMLESS FIRMWARE TRANSITION FROM V1 TO V2


1 100mV/ 2 3 4 2.00V/ 500.0ns/ 1.000ms/ Stop 4 1.68V
V1 Firmware Transient Response
V2 Firmware Transient Response
Application-Specific Switchover (1 µs)
Non Application-Specific Switchover (1.9 µs)
Agilent
Acquisition
Normal
200MSa/s
Channels
AC 10.0:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
-3.156000us
1/ΔX:
+316.86kHz
ΔY(4):
+0.0V

The upper half of Figure 12 displays the application's AC-coupled transient response for a step load toggling every 1.2 ms (1 ms/div). The bottom plot is a zoomed in view (500 ns/div) where the LiveUpdate execution handover took place.

The change in compensator performance is quite noticeable. The transition time from V1 firmware's `EZBL_PartitionSwap()` function call to V2 firmware's `CriticalInit()` function is captured in the figure (Ch4 high pulse timing).

This is the non application-specific switchover time discussed in the **"Software Implementation"** section. This is the minimum `EZBL_PartitionSwap()` and CRT start-up time required as the CRT does not initialize any variables before `CriticalInit()` is called, nor are there any higher priority initialization functions declared in the project. The duration Channel 4 is low indicates the application-specific timing that makes up the critical software for re-enabling the power train using the new 3P3Z compensator.

Pressing switch SW1 will reveal a new LCD display screen that prints out the critical swap time, which should be around 1.9 µs. The `CriticalInit()` function took an additional 1 µs for a total downtime of approximately 3 µs. For this LiveUpdate instance, the available soft swap window was 3.5 µs, so not a single interrupt for both the buck and boost converters were missed or delayed as a result of the firmware update.

In this application example with the Preserve All model, there are pointers stored in RAM structures that point to Flash functions and these will need to be reinitialized before any code referencing these pointers is executed. As these pointers are tied to UART communication and the bootloader tasks, they are not initialized in the critical path. Instead, these pointers were updated inside the `SecondaryInit()` priority function. After reconfiguring the pointers, all remaining interrupts can be enabled. Example 9 shows the remaining priority code to complete the LiveUpdate event and get the application fully operational.

Note: The toolchain moves the contents of Flash between build instances. Any pointer stored in RAM that points to Flash, such as Function Callback Pointers stored in RAM structures, need to be reinitialized in a LiveUpdate event. Determining what pointers are present in the application is a manual process and it is good practice to record all such pointers in the application to ensure proper reinitialization during LiveUpdate events. Failure to reinitialize RAM Pointers to Flash objects will result in unexpected behavior of the application, typically an address error trap or Invalid Opcode Reset. Applying the address attribute to functions can force code to their historical location. However, on larger devices, this does not ensure that a Function Pointer, preserved in RAM, will remain valid as XC16 may substitute a function's true address with a 16-bit linker generated handle. Handles are addresses to "`GOTO trampoline`" instructions required to branch to code above 16-bit addressing limits. The location of a trampoline cannot be assigned to an absolute address.

EXAMPLE 9: FLASH POINTER INITIALIZATION

```
void __attribute__((priority(200))) SecondaryInit(void) {
    if(!_SFTSWP) {
        return;
    }
    // Fix I2C1_OnWrite function callback pointer
    I2C_Tx.onWriteCallback = I2C1_OnWrite;
    _MI2C1IE = 1;          // I2C 1 Master-mode dynamic load switching communications code

    // Fix UART FIFO function callback pointers
    UART1_RxFifo.onReadCallback = UART_RX_FIFO_OnRead;
    UART1_TxFifo.onWriteCallback = UART_TX_FIFO_OnWrite;
    UART1_RxFifo.flushFunction = UART1_RX_FIFO_Flush;
    UART1_TxFifo.flushFunction = UART1_TX_FIFO_Flush;
    _U1TXIE = 1;           // UART 1 TX EZBL Bootloader FIFO code
    _U1RXIE = 1;           // UART 1 RX EZBL Bootloader FIFO code

    // Fix NOW Task function pointer for calling the Bootloader
    EZBL_bootloaderTask.callbackFunction = EZBL_BootloaderTaskFunc;
    _T1IE = 1;             // Timer 1 NOW tick timing code
}
```

LiveUpdate WITH FEWER DEPENDENCIES

To demonstrate the different Preserve Data models, a third revision of software is required, but this time the MPLAB X project will remove the Preserve All Data model. The `.elf` file to use for address preservation now points to Version 2 firmware's `.elf` output artifact. In this case, only a small number of critical variables have been manually marked with the Preserved attribute. These are the variables to keep the power train fully operational, such as buck/boost controller requirements (control reference, control history, error history and A/B coefficients), as well as the system state flags. All other variables and peripheral SFRs will be reinitialized upon partition swap.

With V2 firmware running on the microcontroller, right click the V3 project and select build (ensure the `ezbl_dual_partition.mk` file has the proper COM port selected). This will load V3 firmware into the Inactive Partition and perform the partition swap, as discussed above, with V1 to V2 firmware transition.

With this new firmware build (LiveUpdate scenario):

- Buck/boost converter remains fully operational and stable. Associated variables had the preserve attribute applied.
- Boost load has changed in the firmware update.
- `CriticalInit()` function removed the V1 to V2 RAM copy for buck error/control history and the initialization routine for the buck converter.
- Reinitializing pointers in `SecondaryInit()` function is not required.

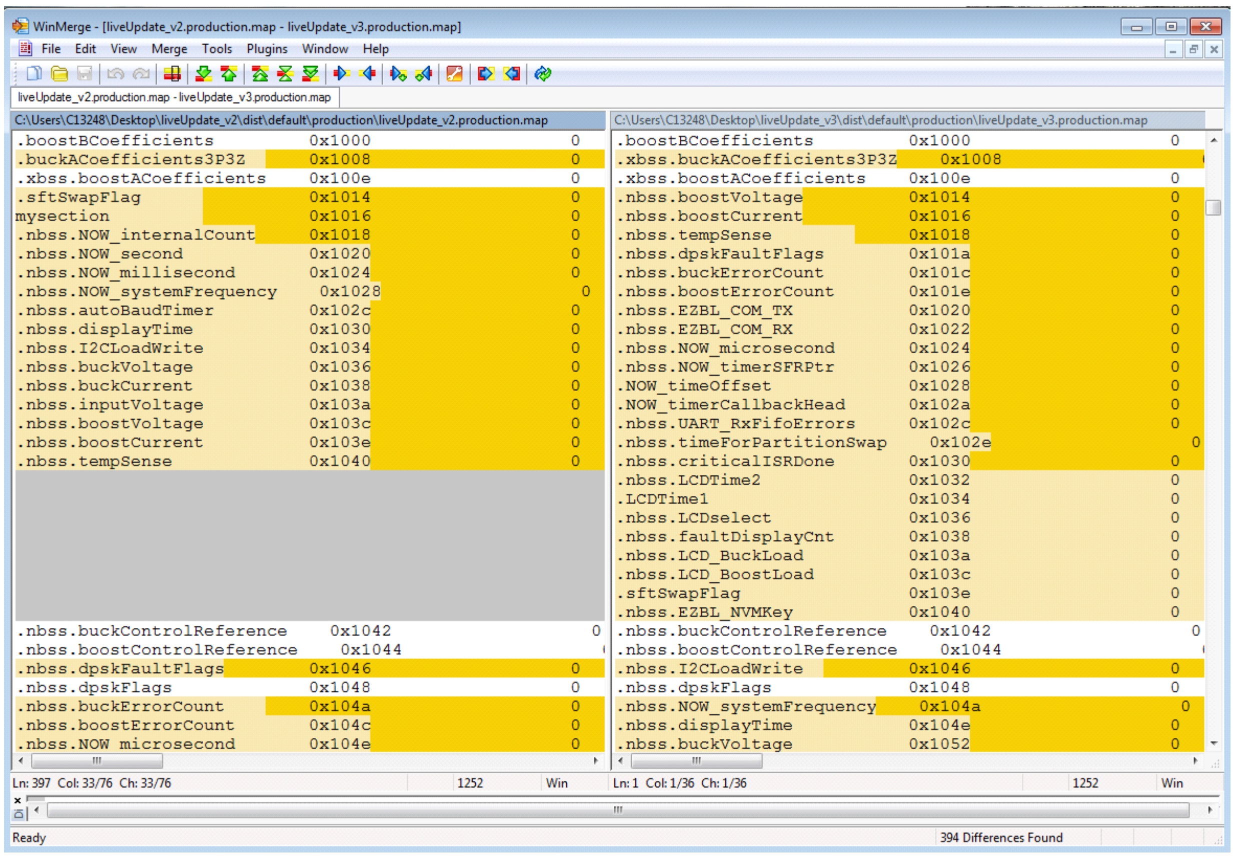
Additionally, LiveUpdate to V3 calls these initialization routines when restarting `main()`, even though no functionality has been intentionally changed relating to them. Reinitialization is required as their variables are not preserved and can freely move to new addresses during linking:

- LCD screen display variable has been reinitialized; therefore, the LCD resets to the home screen.
- `EZBL_BootloaderInit()` reinitialized UART communications, timing peripherals and variables associated with them. A conditional statement for SFTSWP to avoid re-executing of oscillator initialization code is necessary to prevent temporary interruption to the PWM and ADC.
- The I^{2}C peripheral (used to control the output loads) and communication variables are reinitialized.

In this example, the total swap time is much longer than that from V1 to V2 firmware, as the majority of the application variables were reinitialized by the CRT at the boot swap event. This particular LiveUpdate scenario may be required as structures grow, variable types change within structures, or perhaps because an outside software library is being updated with no way to identify if the changes within it can safely reuse existing state variables implemented in the previous library release. No matter the reason, the majority of variables will be reinitialized and it is important to note that certain functions may need to be called from a soft swap event for all application features to resume back to normal operation. For things like LCD reinitialization, this can yield the illusion that the product has undergone a device Reset, but in actuality, a controlled Reset of selected subsystems took place.

Figure 13 shows the map files of one build instance between V2 and V3 firmware. As expected, only those variables marked as preserved retained their original address.

FIGURE 13: VARIABLE MAPPING PRESERVE SELECTED VARIABLES MODEL


WinMerge - [liveUpdate_v2.production.map - liveUpdate_v3.production.map]
File Edit View Merge Tools Plugins Window Help
liveUpdate_v2.production.map - liveUpdate_v3.production.map

C:\Users\C13248\Desktop\liveUpdate_v2\dist\default\production\liveUpdate_v2.production.map

```
.boostBCoefficients            0x1000                 0
.buckACoefficients3P3Z         0x1008                 0
.xbss.boostACoefficients       0x100e                 0
.sftSwapFlag                   0x1014                 0
mysection                      0x1016                 0
.nbss.NOW_internalCount        0x1018                 0
.nbss.NOW_second               0x1020                 0
.nbss.NOW_millisecond          0x1024                 0
.nbss.NOW_systemFrequency       0x1028                 0
.nbss.autoBaudTimer            0x102c                 0
.nbss.displayTime              0x1030                 0
.nbss.I2CLoadWrite             0x1034                 0
.nbss.buckVoltage              0x1036                 0
.nbss.buckCurrent              0x1038                 0
.nbss.inputVoltage             0x103a                 0
.nbss.boostVoltage             0x103c                 0
.nbss.boostCurrent             0x103e                 0
.nbss.tempSense                0x1040                 0








.nbss.buckControlReference       0x1042                 0
.nbss.boostControlReference        0x1044
.nbss.dpskFaultFlags           0x1046                 0
.nbss.dpskFlags                0x1048                 0
.nbss.buckErrorCount           0x104a                 0
.nbss.boostErrorCount          0x104c                 0
.nbss.NOW_microsecond          0x104e                 0
```

C:\Users\C13248\Desktop\liveUpdate_v3\dist\default\production\liveUpdate_v3.production.map

```
.boostBCoefficients            0x1000                 0
.xbss.buckACoefficients3P3Z       0x1008
.xbss.boostACoefficients       0x100e                 0
.nbss.boostVoltage             0x1014                 0
.nbss.boostCurrent             0x1016                 0
.nbss.tempSense                0x1018                 0
.nbss.dpskFaultFlags           0x101a                 0
.nbss.buckErrorCount           0x101c                 0
.nbss.boostErrorCount          0x101e                 0
.nbss.EZBL_COM_TX              0x1020                 0
.nbss.EZBL_COM_RX              0x1022                 0
.nbss.NOW_microsecond          0x1024                 0
.nbss.NOW_timerSFRPtr          0x1026                 0
.NOW_timeOffset                0x1028                 0
.NOW_timerCallbackHead         0x102a                 0
.nbss.UART_RxFifoErrors        0x102c                 0
.nbss.timeForPartitionSwap       0x102e                 0
.nbss.criticalISRDone          0x1030                 0
.nbss.LCDTime2                 0x1032                 0
.LCDTime1                      0x1034                 0
.nbss.LCDselect                0x1036                 0
.nbss.faultDisplayCnt          0x1038                 0
.nbss.LCD_BuckLoad             0x103a                 0
.nbss.LCD_BoostLoad            0x103c                 0
.sftSwapFlag                   0x103e                 0
.nbss.EZBL_NVMKey              0x1040                 0
.nbss.buckControlReference       0x1042                 0
.nbss.boostControlReference        0x1044
.nbss.I2CLoadWrite             0x1046                 0
.nbss.dpskFlags                0x1048                 0
.nbss.NOW_systemFrequency        0x104a                 0
.nbss.displayTime              0x104e                 0
.nbss.buckVoltage              0x1052                 0
```

```
Ln: 397  Col: 33/76  Ch: 33/76        1252   Win
Ln: 1  Col: 1/36  Ch: 1/36            1252   Win
Ready                                 394 Differences Found
```

VALID CODE IMAGE RECEPTION

The `ezbl_dual_partition.mk` Makefile, in conjunction with the EZBL Java application, create the SHA-256 hash that contains vendor, model, application name and any other identification strings. This hash key is broken down into 32-bit chunks to be passed to the linker as a symbol and is eventually part of the application to get passed to the device. As described in the **"Software Implementation"** section, this information is used to ensure the code image being received is meant for this application. Only after validating the hash will the bootloader perform the download of the code image.

Example 10 shows the hash key fields for these applications.

EXAMPLE 10: `BOOTID_HASH`

```
BOOTID_VENDOR = "Microchip Technology"
BOOTID_MODEL = "ezbl_product"
BOOTID_NAME  = "Microchip Development Board"
BOOTID_OTHER = "Dual Flash Partition Device"
```

In the examples provided, there are three identification fields for software versioning. These consist of major, minor and build number. If new firmware having a major version match and a minor ID of exactly one greater than the existing code is sent to the bootloader, a LiveUpdate will be performed. Any image with a mismatched major ID or minor+1 fields will be programmed to the Inactive Partition (sequence number too), but with successful verification, the device will reset to begin executing the new firmware rather than attempt a partition swap. This behavior is useful for situations that are deemed too difficult to build a LiveUpdate project for and it enables forced deployment of code that is one or more updates behind (or potentially ahead of) the firmware release that was used to develop the LiveUpdate project.

If an image with an exact match of major, minor and build number is sent to the bootloader, the bootloader rejects the image without modifying the Inactive Partition. This avoids unnecessary programming and device Reset if a user inadvertently tries to program the same firmware more than once.

It may be useful during development to perform LiveUpdate scenarios against the same project with the existing and new code images differing by one build cycle. For this reason, an image with a mismatched build number, but matched major and minor versions, will attempt a LiveUpdate without Reset. However, it is important to note what software executes on the soft swap event and if this software can switch from itself to itself. In the V2 firmware example provided in Example 8, if the buck reinitialization code in the critical ISR is commented out, this code would then accept continuous switchover events to the same code image. Failure to comment it out would result in unpredictable behavior on the buck converter, as old 2P2Z controller code is being referenced without the old 2P2Z state existing in RAM. Additionally, since the project is configured to preserve variable addresses against the V1 `.elf` artifact, update variables added in V2 can move during linking with respect to the prior build cycle. Lastly, SFRs need to be considered as they could carry state information that would not be generated by Reset or LiveUpdate from a historical V1 executable.

750W AC/DC EXAMPLE

To show different application examples of online firmware updates, Microchip has also developed firmware examples for the 750W AC/DC reference design. In digitally controlled AC/DC power converters, it is common to have two microcontrollers separated by an isolation barrier, one for the PFC front-end and one for the DC-DC converter. There is typically a UART communication bridge for sending data between the two microcontrollers. This system configuration presents a small challenge for LiveUpdate applications.

The main objective is to perform firmware updates for both power converters without bringing the supply offline to do so. The issue with this configuration is that the host (i.e., PC) does not have direct access to the PFC microcontroller. To circumvent this, the DC-DC stage needs to act like a messenger between the host and PFC controller, which means additional sophistication is required in the DC-DC controller.

In this example, the DC-DC controller acts as a broadcast repeater by automatically passing all incoming host messages to the PFC controller. Both the DC-DC and the PFC controller will read the identification header when the host offers a new firmware image, and if there is a match with either controller, that controller will respond while the other sits Idle (monitor state). When the PFC microcontroller is the target device, the PFC will Acknowledge the identification match by sending a response back to the DC-DC controller, which is then relayed to the host. The DC-DC controller will continue to broadcast the incoming bytes, but will discard them from local processing until the End-of-File (EOF) or extended interval of communication silence is observed. Once the transfer ends, the internal EZBL state machine is reset and a bootloader wake-up string is then required to start the bootload process again. Monitoring the bytes, even though data is meant for another controller, allows for synchronization and eliminates any chance the non-targeted controller(s) could act on any part of the binary content in the broadcasted image.

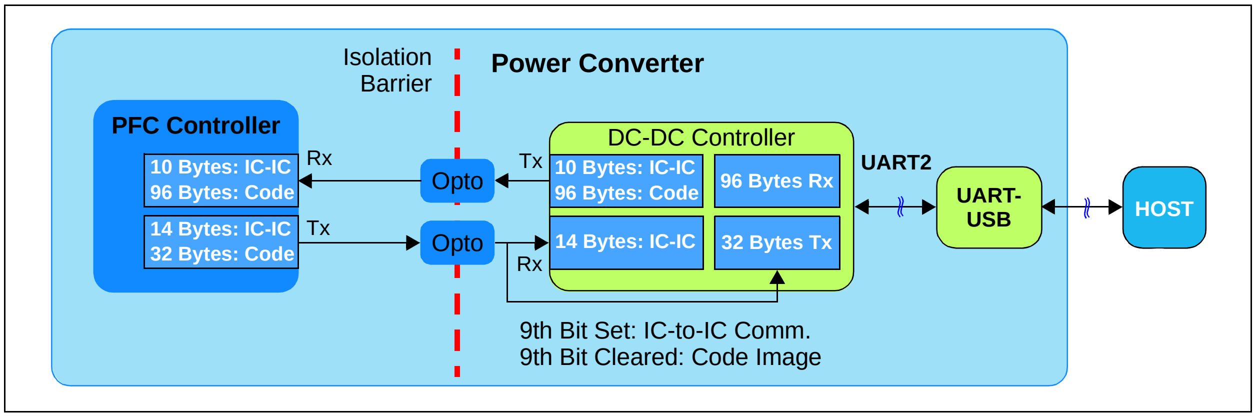
The software could be partitioned such that the DC-DC controller keeps the identification and versioning records for the PFC within its firmware. Here, the DC-DC node would be able to make the decision to pass the incoming data to the PFC controller if that was the intended target device. This approach can preserve communications bandwidth, but comes at the cost of greater overhead in the DC-DC node and requires greater maintenance should the PFC controller be treated as a substitutable block. Added challenges arise if the system is designed to scale with more nodes connecting to the bus or other types of data need to pass between the PFC node and the host PC. Because of the above issues, in Microchip's 750W AC/DC Reference Design, the same bootloader firmware is used for both converters and the DC-DC controller always rebroadcasts host data through the isolation barrier to the PFC. If the PFC generates outbound data, the DC-DC will relay such messages back to the host.

There are two challenges with getting data across the isolation barrier. The first challenge is being able to distinguish between data that is occurring regularly between power converters and that of a new code image for LiveUpdate. In this example, the PFC and DC-DC UARTs are configured for 9-Bit Data mode with the 9th bit set to indicate a board-local, or IC-IC, data byte. The 9th bit is cleared when the byte originates from or is headed to the off-board host PC. This creates two virtual communication domains while sharing the same communication hardware. The DC-DC node's UART over to the host implements traditional eight data bit formatting to maintain a normal appearance to the outside world and avoid wasting communication bandwidth on a 9th bit that would always be '`0`'. Priority is given to IC-IC traffic, which temporarily suspends any code image transfer through the isolation barrier. However, as every data byte is tagged with a target domain bit, the protocol can be readily adapted for equitable, ping-pong style sharing of the communications pipe on a per-byte basis. As 9-Bit Data mode may not be suitable for all applications, different methods for distinguishing the two paths can be developed. One option is to coalesce adjacent bytes into a frame and append a header indicating the target domain and frame's byte length prior to transmitting the frame. Although this method can scale to support almost any communication hardware, this would adversely affect message latency and jitter for periodic transmissions. It also may carry high overhead both in terms of implementation complexity and bandwidth used, especially if IC-IC messages are short and frequent.

Another option would be to toggle between protocol domains by means of an out-of-band signal. For example, the transmitting node could send a Break to intentionally trigger a framing error on the receiver, followed by a 1-byte indicator signaling the new target communications domain. This method would contribute almost no bandwidth overhead since LiveUpdate traffic is very rare and both nodes would normally stay in IC-IC mode. When communicating with the host, the bootloader data passed back includes status or available buffer sizes in order to implement software flow control. This tells the host how much free space is available in the receive data buffer and forces the host to wait if the bootloader is busy erasing or programming existing data with nowhere to queue more data. As a result, the bandwidth lost by 9-bit signaling and temporary delay induced by IC-IC traffic prioritization is immaterial.

The second challenge is in handling mismatch between data rates. The communication medium with the host is UART in this example, but could just as easily have been I^{2}C. The communication link between the host and DC-DC converter supports auto-baud and can sustain 460 Kbytes per second (460800 baud), which is the maximum permitted by the MCP2221A USB to UART converters on the reference board. The communication between the two microcontrollers is typically a fixed baud rate with a slower throughput (115200 baud; limited by the isolator's performance). To handle the difference in communication rates, the DC-DC buffer holding host data to be sent to the PFC microcontroller needs to be as large as the receive data buffer in the PFC. This ensures the PFC will not request more data from the host than the DC-DC has room to buffer in case IC-IC traffic fully blocks the isolation barrier for an extended interval. Although of lessor significance, it is suggested that the DC-DC transmit buffer towards the host be allocated as big as the PFC's transmit buffer. This ensures PFC status and flow control signaling to the host will not be corrupted in the event the host node chooses a baud rate appreciably slower than the communications rate though the isolation barrier. See Figure 14 for implementation details.

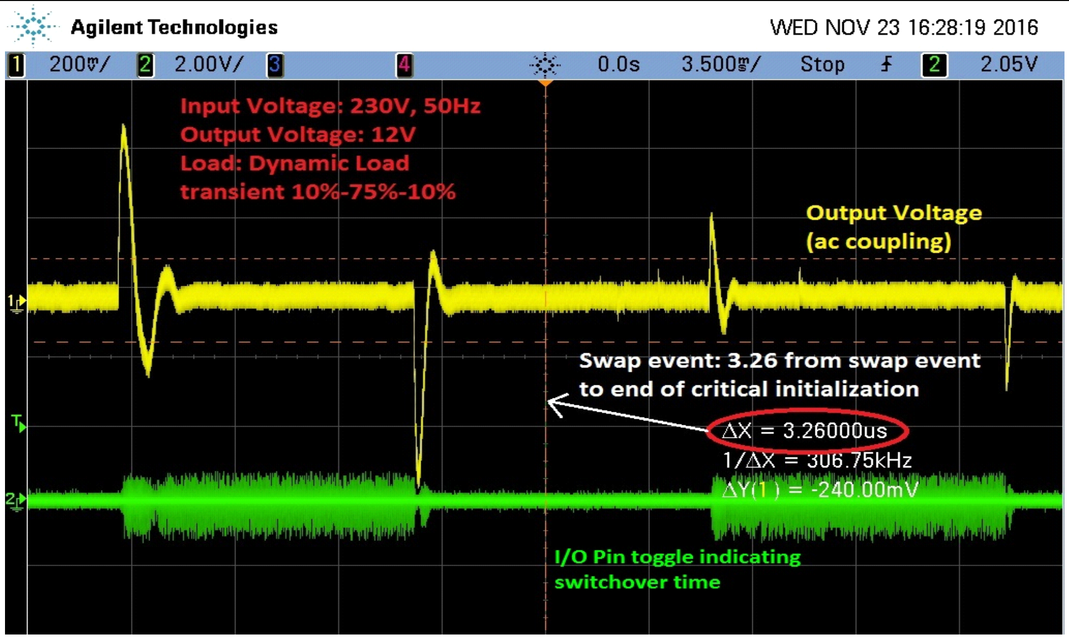
FIGURE 14: 750W AC/DC COMMUNICATION STRUCTURE



This implementation shares the host transmit buffer as only one converter should be writing to it at a time.

There are multiple LiveUpdate examples associated with this reference design. On the DC-DC side, the main example modifies the I/Q format for the controller coefficients to obtain a better loop gain response by increasing the controller bandwidth. This is observable when looking at the load transient response. Figure 15 captures the switchover instant and the new controller's transient behavior.

FIGURE 15: 750W DC-DC CONVERTER TRANSIENT RESPONSE IMPROVEMENT

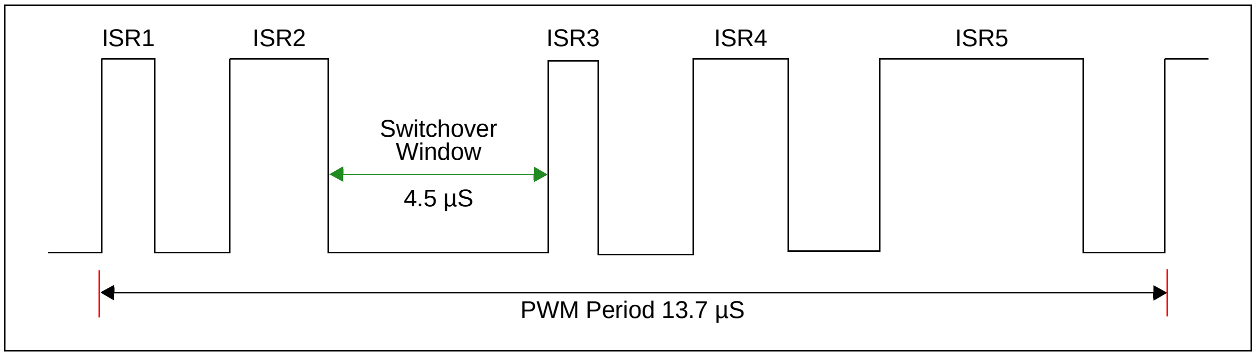


This is a similar example to the DPSK, but instead of bringing in a new controller type and having to initialize that controller, additional firmware was developed to ensure the controller modifications occur with small steady-state error. The new coefficients are then loaded and new software flags are initialized, which will adjust the accumulated control history appropriately for the new controller coefficients. The controller update happens well after the critical timing path.

Setting up the appropriate switchover timing window posed a few more challenges relative to the DPSK example, as there are five critical interrupt events for executing the control system (see Figure 16).

One interrupt executes the voltage compensator (ISR5), two ISRs control the Fault input source for proper Peak Current mode control with a single current sense input (ISR1/3) and the other two events are the slope compensation calculations (ISR2/4). In Figure 16, it can be seen that the switchover event is synchronized to ISR2 completion.

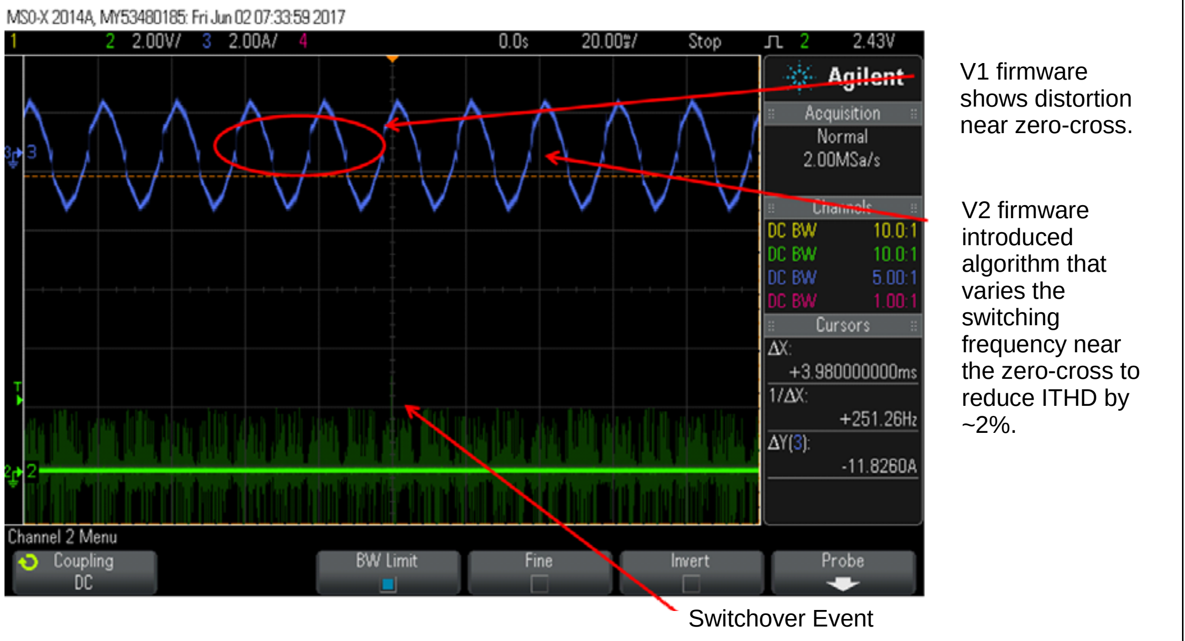
FIGURE 16: 750W DC-DC CONVERTER SWITCHOVER SYNCHRONIZATION



Since ISR2 and ISR4 are the same interrupt routine, just called twice, knowledge of what ISR event occurred previously is required to align to the window where the voltage compensator isn't being called. In this design, a maximum switchover window of 4.5 µs is obtained. Removing the 1.9 µs for non application-specific switchover timing, our critical path timing is 2.6 µs, which is short, but remains useful for many different update cases. For more details on the DC-DC converter design, see AN2388, *"Peak Current Controlled ZVS Full-Bridge Converter with Digital Slope Compensation"* (DS00002388) application note.

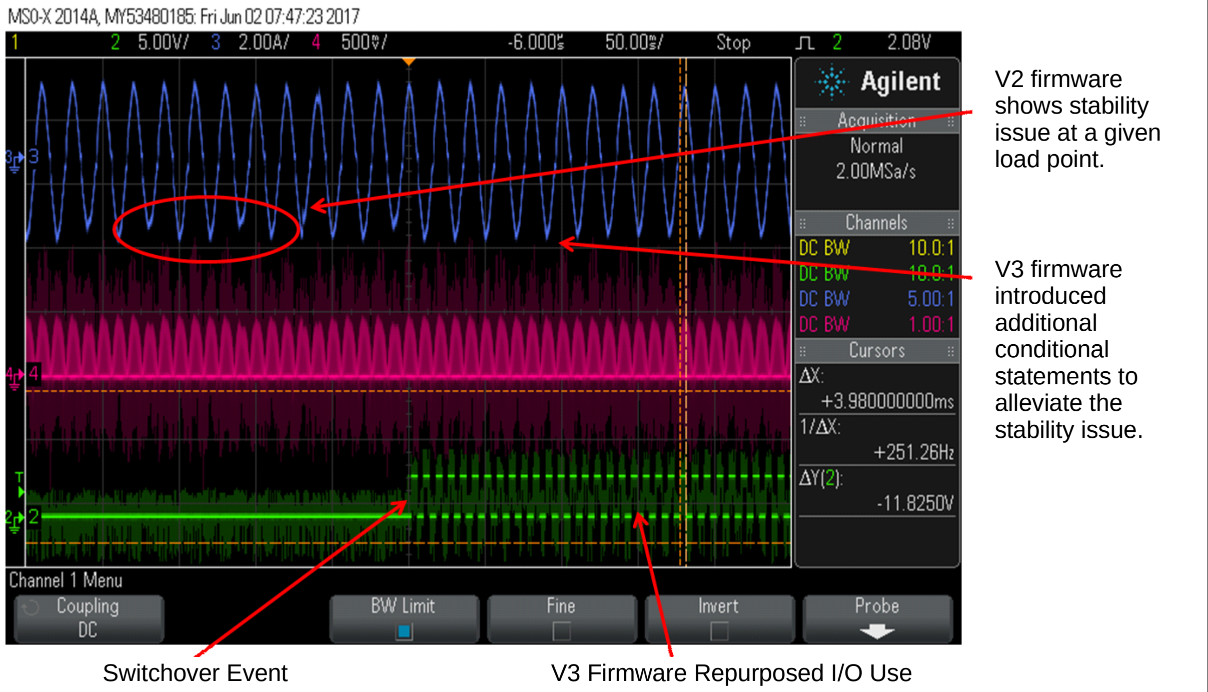
For the PFC converter, two different LiveUpdate examples were created to show the flexibility of online firmware updates. The first example introduces a new algorithm that is executed at 50 kHz to improve ITHD of the power converter. This new algorithm improves ITHD by reducing the switching frequency near the AC zero-cross events. This update required an additional bit field to be added to an existing bit field structure. This is a fairly simple update as there is nothing timing-critical and no variable remapping is required. The firmware does require pointers to Flash to be reinitialized, but otherwise is mostly a change to executable data, preserving all of RAM. Figure 17 captures the input AC current before and after the switchover event. The AC cycles following the switchover event show less distortion near the zero-crossings.

FIGURE 17: 750W PFC DISTORTION CORRECTION THROUGH LiveUpdate


MSO-X 2014A, MY53480185: Fri Jun 02 07:33:59 2017
1 2 2.00V/ 3 2.00A/ 4 0.0s 20.00ms/ Stop 2 2.43V
Agilent
Acquisition
Normal
2.00MSa/s
Channels
DC BW 10.0:1
DC BW 10.0:1
DC BW 5.00:1
DC BW 1.00:1
Cursors
ΔX:
+3.980000000ms
1/ΔX:
+251.26Hz
ΔY(3):
-11.8260A
Channel 2 Menu
Coupling
DC
BW Limit
Fine
Invert
Probe
V1 firmware shows distortion near zero-cross.
V2 firmware introduced algorithm that varies the switching frequency near the zero-cross to reduce ITHD by ~2%.
Switchover Event

The second online firmware update is similar to the first example with respect to adding a new function and a single member to a structure. In this example, it was observed at a given load point, the system would oscillate due to the adaptive algorithms for the current reference calculation. Additional firmware was developed which required algorithms to be enabled/disabled near the zero-cross event. Adding in the additional conditional statements fixed the instability issue, as seen in Figure 18. Before the switchover event, it can be seen that the input current amplitude oscillates between cycles. After the switchover event, this issue is no longer seen. Also in this example, the I/O indicator for the switchover event in the new firmware was repurposed and is toggling at a different rate to indicate new firmware is executing.

FIGURE 18: 750W PFC INSTABILITY CORRECTION THROUGH LiveUpdate

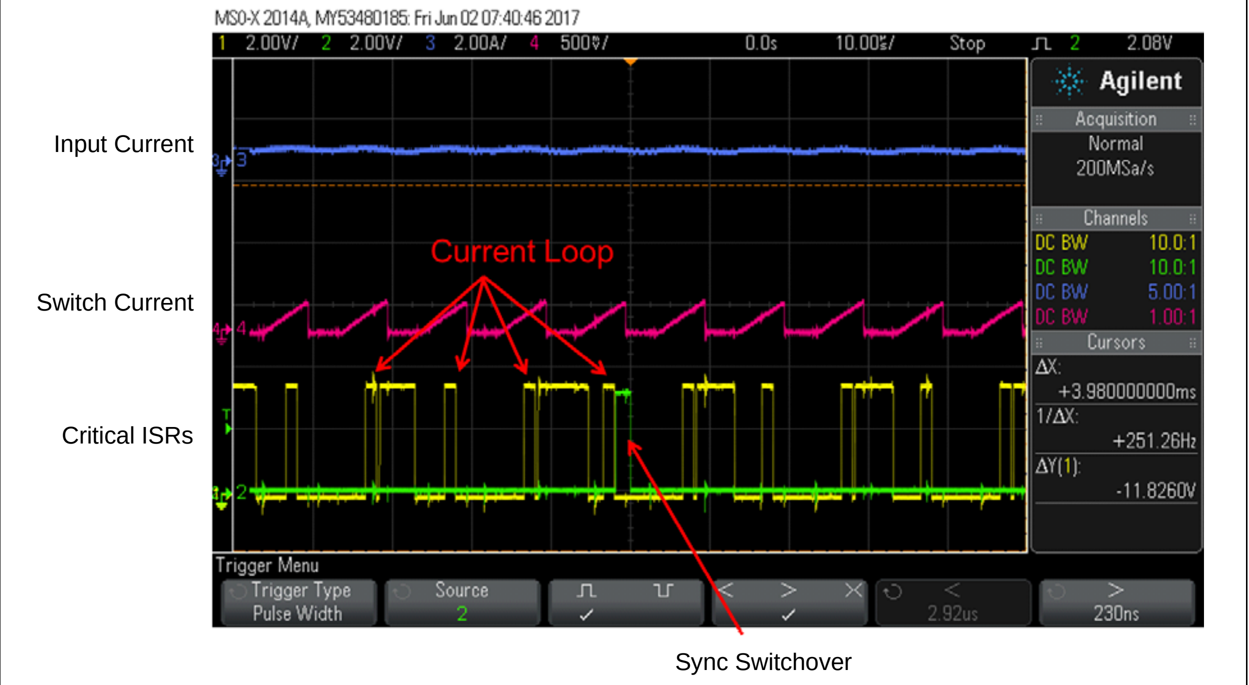


The PFC converter presented its own challenges for determining the switchover event. In this application we have the current loop, which is the highest executed loop running at 100 kHz, critical input signal conditioning functions, several adaptive algorithms executing at 50 kHz and many functions executing at 12.5 kHz, including the voltage compensator. In between every other current interrupt event, there are a lot of functions that get executed. This means the switchover event needs to synchronize to the end of the current loop ISR, but in the window that alternates with the other priority functions.

Figure 19 shows a screenshot of the critical ISRs running in the PFC application.

From the figure, it can be seen that the switchover event occurs in the proper cycle and right after the current loop. This will maximize the critical timing path for the PFC, which in this example, will be close to 8 µs. This scope plot was captured during the first online update event.

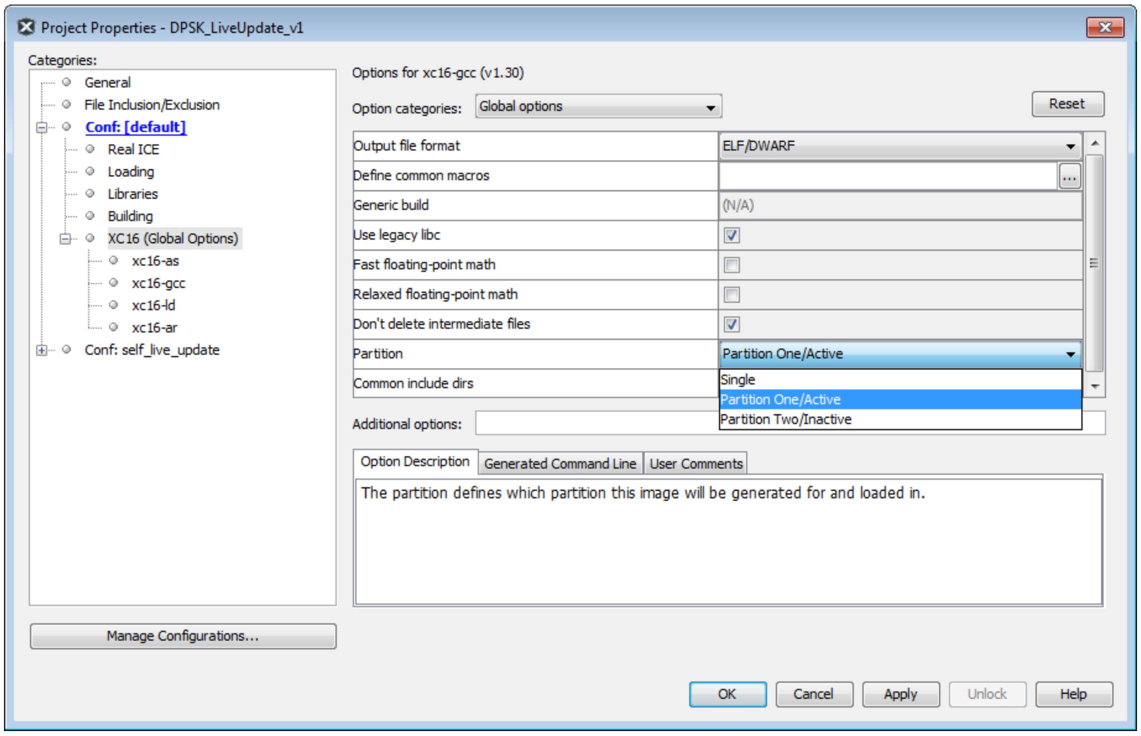
FIGURE 19: 750W PFC SWITCHOVER TIMING SYNCHRONIZATION



DEBUGGING LiveUpdate APPLICATIONS

Debugging online firmware updates is possible as MPLAB X allows selecting the target partition for where code will be physically programmed. In a dual partition device, the Active Partition is the only one which the CPU can execute from and it is always located starting at address 0x000000. The Inactive Partition always resides at 0x400000, and is available for non-blocking erase and programming commands, but cannot be used for execution until it swaps places with and becomes the Active Partition. The physical concept of Partition 1 versus Partition 2 has no significance to software outside the context of determining which partition will be made active at device Reset when the hardware compares FBTSEQ Configuration Word values. By selecting either Partition One/Active or Two/Inactive in the XC16 global options within the **Project Properties** dialog, the code will be compiled and linked to execute starting from address 0x000000, but the data records in the final `.hex` file will have a 0x000000 or 0x400000 offset applied. See Figure 20 for the location of partition selection.

FIGURE 20: PARTITION SELECTION FOR DEBUGGING APPLICATION



Project Properties - DPSK_LiveUpdate_v1

Categories:
- General
- File Inclusion/Exclusion
- Conf: [default]
  - Real ICE
  - Loading
  - Libraries
  - Building
  - XC16 (Global Options)
    - xc16-as
    - xc16-gcc
    - xc16-ld
    - xc16-ar
- Conf: self_live_update

Options for xc16-gcc (v1.30)

Option categories: Global options

Output file format	ELF/DWARF
Define common macros	
Generic build	(N/A)
Use legacy libc	✓
Fast floating-point math	☐
Relaxed floating-point math	☐
Don't delete intermediate files	✓
Partition	Partition One/Active
Common include dirs	

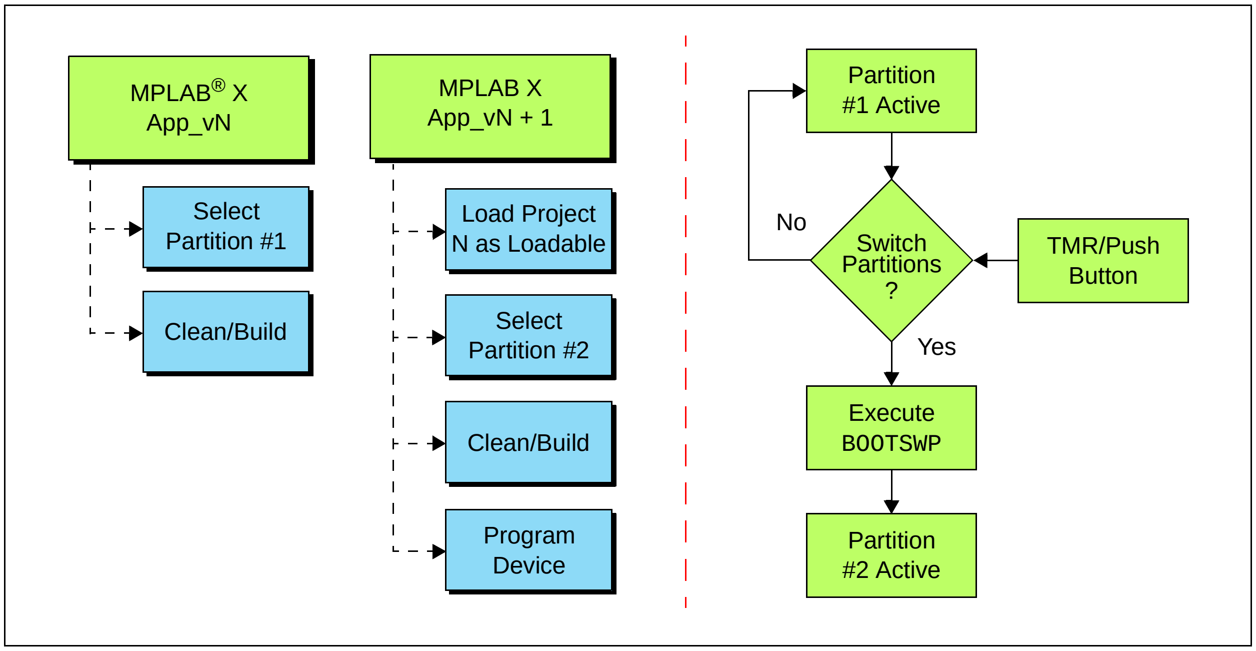
Partition options: Single / Partition One/Active / Partition Two/Inactive

Additional options:

Option Description: The partition defines which partition this image will be generated for and loaded in.

In order to debug the switchover event, the previous project will need to be added as a Loadable into the newly active project and the proper partition will need to be selected. As the examples used in this application note, use default sequence numbers and always target Partition 1; the active project for debugging will be configured for Partition 2. This aligns with the natural order for performing LiveUpdate events. To add the previous project as a loadable, right click the **Loadables** folder under the **Projects** tab in MPLAB X and select **Add Loadable Project**. Browse to the previous project and select **Add**. Ensure the previous project was built with the correct partition selected (Figure 21).

FIGURE 21: DEBUGGING SWITCHOVER EVENT



At the moment, with current development tools, it is not possible to add hardware breakpoints to the Inactive Partition. Hardware breakpoints will always map to the Active Partition. To halt CPU execution after the switchover event, software breakpoints can be added to the new firmware. Once halted in the newly Active Partition, hardware breakpoints may be used. It may be important to ensure certain peripherals (such as PWM) continue executing after halting the microcontroller.

As both firmware images are being programmed, debugging a LiveUpdate switchover event does not require bootloader firmware. However, additional software will need to be developed to initiate the partition swap, such as an I/O port change (push button) or even a time-out period using a timer. This can greatly simplify testing partition swap events and testing critical initialization variables/functions in an application.

TIPS AND TRICKS

Swap Time Optimization

The CRT initialization function and the `EZBL_PartitionSwap()` function are provided for reference. These functions can be modified on an application basis to reduce the non application-specific switchover timing. In the partition swap function, all interrupts are disabled by clearing all IECx registers. Although this routine is generic for use with multiple processors, it is optimized and ensures all interrupts get disabled. However, disabling all interrupts may be redundant. All low-priority interrupts would have been disabled earlier, before initiating the switchover, as to create the switchover window. Only critical interrupts are required to be disabled at this time. Reducing this routine down to just critical interrupts disabled could reduce the switchover time by ~200 ns. The Global Interrupt Enable bit, GIE (INTCON2<15>), could be used to disable critical ISRs and reduce switchover time further. Just ensure that all critical initialization code tied to the critical ISRs is executed before enabling interrupts.

Similar to the code examples discussed in this Application note, the CRT function can also be rewritten to help reduce the switchover time. There are several branch conditions that could be replaced with bit test, skip type instructions that could save a few instruction cycles in the critical path.

The Compiler Run-Time start-up function can be found in the compiler directory at the following path:

<xc16 install directory>\src\libpic30.zip

As CRT data initialization is implemented by decoding a table of packed records located in Flash, initializing priority variables using the CRT could take many instruction cycles. For this reason, variables in the critical timing path should have the Persistent attribute and be initialized using code placed in a priority function that explicitly assigns values, and not by the CRT. Although literals and derived constants encoded in the instruction stream consume more Flash space, this will save significant time in the critical path and allow these variables to be appropriately initialized for both cold start and soft swap cases. See priority initialization in Example 8.

Preserved Address; Not Preserved Correctness

It is worth noting that there are a few variable declaration instances where the toolchain is able to satisfy a variable's preservation address, but which results in incorrect run-time behavior without a build warning. For example, as a structure's member order changes, or maybe even a structure and/or array size shrinks, there is no build indication to the user. Although this is to be expected, in a LiveUpdate scenario this could be problematic if improperly handled. Consider the example where a preserved structure's member order changes. The linker sees a structure as one conglomerated unit and the Preserve attribute will cause the base address of the structure to remain unchanged. However, as a structure contains member variables, each with a sequentially increasing address offset relative to the base of the structure, any reordering of internal members or a change to a member's type width will cause all subsequent member variables to be accessed using the wrong relative offset, and thus, the wrong system RAM address. To preserve a structure with these internal changes, an initialization function will need to swap the RAM values for those members that were affected, paying close attention to effects of hidden alignment padding bytes that may be added or removed as a result of the new member ordering. Also note that any function that uses a stored pointer or offset to internal structure members will also need to be updated in the new software version. If changes to the structure are complicated, a less fragile approach may be to implement the desired structure as a new variable, preserving the old one, and writing an initialization routine to marshal the preserved data out of the original structure and into the new one.

A modified array can produce essentially identical problems. The linker can only preserve the base address of the array and complain if the array grew such that it overlaps some other preserved or absolutely positioned variable. Changes to the base data type, indexable geometries, attributes or reduction of elements can materially affect how the data is stored or accessed and may need special transition routines.

For the two scenarios where preserved structure or array lengths change (shorten), with `-fdata-sections` enabled, these newly open RAM locations can be reused by the linker. This means that the newly added (Update attributed) variables may get initialized by the CRT and clobber the existing tail data in the preserved variable. If the preserved tail was intended to be saved and reused in a separate, dedicated variable, then this tail needs to by copied in a priority function before the CRT overwrites it. Additionally, the user is responsible for ensuring that the new firmware revision has properly changed any indexing component if being accessed indirectly with pointers. If this is not taken care of, incorrect data may be read and/or an address error trap could be generated.

Additional Debugging Technique

Sometimes, despite best efforts to maintain variable addresses and define handover routines to maintain a coherent operating state after swapping partitions, you may still run into a trap exception or spontaneous device Reset shortly following a LiveUpdate. These are almost certainly caused by run-time data that needs special handling, but which got missed when decorating variables with attributes and setting up initialization routines. Depending on the application, reproducing and then debugging the problem can pose a serious challenge.

To help in these scenarios, it is valuable to have hardware provisions for a fast UART debugging console, preferably electrically isolated, such that you can transfer comprehensive RAM, SFR and Flash dumps for leisurely analysis in a text file. By saving a log before swapping partitions and another identically structured log at the earliest opportunity following a Fault, it is possible to compare the two files and look for unexpected data in unexpected places.

Several EZBL APIs can be used to help generate such logs. Of potential interest is the `EZBL_TrapHandler()` function, located in `ezbl_lib.a` (source code in EZBL distribution at `ezbl_lib\weak_defaults\EZBL_TrapHandler.c`). This function implements a generic trap exception handler that prints some SFRs, RAM contents and Flash contents commonly needed for debugging, and can be easily modified to print a comprehensive report of all device states. That data appears as human readable text, so any serial console application on the PC may be used to view and/or save it.

To try this trap handler:

1. Ensure `ezbl_lib.a` is added to the project under Libraries.
2. `#include “ezbl.h”`
3. At file level scope in any `.c` file, insert:

   `EZBL_KeepSYM(EZBL_TrapHandler);`
4. Comment out any other trap handler functions that you have in your project, such as `_DefaultInterrupt()` and `_AddressErrorTrap()`. `EZBL_TrapHandler()` will only be called when a project function isn’t defined for a particular trap.
5. Ensure that the system clock, the NOW timing APIs and a UART TX pin are configured to transmit the debug output prior to any trap. The code necessary to configure the UART may take the form:

EXAMPLE 11:

```
NOW_Reset(TMR1, 7370000/2);            // Configure NOW timing APIs for Timer 1 with system clock assumed
                                          to be FRC/2 Hz
IOCON2bits.PENH = 0;                   // Disable PWM2H function on desired U2TX pin
_RP45R = _RPOUT_U2TX;                  // Assign U2TX function to RP45 pin
UART_Reset(2, NOW_Fcy, 230400, 1);     // Initialize UART2 @ 230400 baud and set as target of stdout messages
```

6. The handler can be tested by calling it or creating a divide-by-zero math error:

   `EZBL_CallISR(EZBL_TrapHandler);`

   or

   `volatile int i = i/0;`

NOTES:

ISBN: 978-1-5224-2730-8

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra’anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-7289-7561

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

10/25/17