AN2601

时序关键型应用中的在线固件更新

作者： *Alex Dumais 和Howard Schlunder*
Microchip Technology Inc.

简介

许多应用都要求产品在发布后支持固件更新。固件更新会强制应用包含一个自举程序，以管理闪存内容和与外部主机设备共享信息的通信介质。受单片机的架构影响，闪存擦/写操作期间会停止执行指令，由于这些操作可能花费数毫秒，应用通常离线来执行固件更新。停机时间可能长达数秒，这对于某些最终应用而言可能成本过高。此类应用的一个例子是面向服务器市场的数控电源装置（Power Supply Unit，PSU）。在非冗余情况下，执行软件更新会使PSU离线，从而造成收入损失。即使在冗余环境下，也需要手动重新编程PSU，这同样会产生额外成本。

为了避免在固件升级期间发生停机，Microchip推出了几个实现双闪存分区的16位器件系列（例如dsPIC33EP64GS506、dsPIC33EP128GS808、PIC24FJ256GB412和PIC24FJ1024GB610）。当单片机架构具有两个物理上不同的闪存分区时，应用能够通过活动分区正常执行操作，而非活动分区通过活动分区的自举程序擦除、编程和验证。这些器件还实现了其他功能来支持闪存分区交换（软交换），本应用笔记将对此进行简要介绍。有关双分区实现的更多信息，请参见《dsPIC33/PIC24系列参考手册》中的**“双分区闪存程序存储器”**（DS70005156B_CN）。

本应用笔记讨论了有助于实现在线更新的编译器工具，并详细介绍了应如何设置MPLAB® X IDE项目和实现固件来支持在线更新事件。本应用笔记中创建并讨论了示例项目，以进一步帮助开发/测试在线更新项目。这些PSU示例实现了完整的固件更换，其有效停机时间小于几微秒，可避免中断丢失（这很严重）或系统输出不连续。

用于实现在线更新的编译器功能

MPLAB XC16 v1.30及更高版本中添加了多个功能来实现时间敏感型应用中的在线固件更新。在在线更新应用中，面临的最大挑战是将不同固件版本中的RAM值/变量保持在同一存储器地址中。通过保持RAM内容，可保持运行时的状态信息，而通过将相同的变量分配到同一地址，则可消除启动初始化延时。可执行且可链接的文件（`.elf`）包含保留RAM内容所需的全部相关符号（变量和函数地址）信息。在引用前一版本输出的可执行文件时，编译器/链接器能够将保留变量映射到前一版本中定义的RAM存储单元中。

为支持在线更新应用，创建了四个不同的保留数据模型。在所有模型中，均需引用前一个固件版本的`.elf`文件。后面的**“配置MPLAB® X项目”**部分将详细讨论如何引用`.elf`文件以及为保留数据模型配置项目。

根据使用的保留数据模型，编译器/链接器将：

- 仅保留具有*Preserved*属性的变量
- 保留所选C文件中的全部变量
- 保留活动项目（全部C文件）中的全部变量
- 保留整个项目（包括库归档文件）中的全部变量

保留活动/整个项目中的全部变量被视为更常用的保留数据模型。任何新变量均将通过编译器来初始化，如果新变量位于关键时序路径中，则在项目特定priority函数中初始化。函数指针和指向闪存中存储的常量数据对象（如果存在）的指针将手动重新初始化，只需最少的固件便可管理固件更新之间的更改。这种方法可在最短时间内使应用100%恢复在线状态。预计固件版本之间只进行了少量更改，因此可在这些数据模型中合理地管理新变量/指针。

有些情况下，需要或首选基于C文件或通过选择性地使用Preserved属性来保留各个变量。在这些情况下，将通过保持有限个变量使核心应用正常工作（电源调节），而使应用100%恢复在线状态所需的时间并不重要。这种方法需要通过重新初始化程序（即，自举程序和任务调度程序等）按照与上电复位（Power-on Reset，POR）相同的方式来正确配置变量/外设状态。由于应用的核心功能会持续工作，因此从用户的角度来看，重新初始化这些任务所需的时间是可以接受的。这种方法有助于更好地确保所有变量链接到项目，从而允许对项目进行大量代码更改。

决定使用哪种模型随固件更新的类型变化，具体需要由固件工程师来确定哪种模型最适合更新。在某些情况下，选择性地保留一些变量可能更好；而在另一些情况下，保留全部变量可能更好，同时选择性地选择哪些是新变量并且需要更改。

为支持这些不同的保留数据模型，编译器引入了几个全新属性。这些属性主要用于确定哪些变量需要保留或者哪些变量是新变量并且需要初始化。编译器现在还支持这样一种方法，即在初始化可定义的变量块前分配变量初始化顺序和执行时序关键型函数。以下几节将讨论这些全新属性。

Preserved属性

*Preserved*属性规定固件版本间的变量地址需保持一致，并且编译器不得在软交换事件中重新初始化该变量。具有Preserved属性的变量将始终通过常规器件复位进行初始化，除非它同时具有Persistent属性（例1）。

例1： PRESERVED属性

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

“保留所有”模型中不需要Preserved属性，因为编译器/链接器以隐性方式将所有变量视为需要保留。有关保留数据模型的详细信息，请参见**“配置MPLAB® X项目”**。

Update属性

*Update*属性规定变量为新变量或已在结构上进行修改，需要初始化。具有Update属性的变量在软交换事件以及器件复位时进行初始化，除非标记为Persistent属性（例2）。Update属性还允许链接器自由地将变量分配（或重新分配）到未被占用的RAM地址。

例2： UPDATE属性

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

只有在“保留所有”模型之一的范围内声明变量时才需要Update属性。随后，这些变量将不会继承隐性分配的保留属性。

Priority属性

*Priority*属性支持通过编译器运行时（Compiler Run Time，CRT）按一定优先级执行代码和初始化变量。使用Priority属性时，将在完成数据初始化（借助CRT）以及`main()`调用之前以一定优先级初始化变量和调用函数。Priority属性适用于软交换以及冷启动的情形。有关应用Priority属性的信息，请参见例3。

编译器支持$2^{16} - 1$个优先级，每个优先级中的函数/变量数量不受限制。优先级0为最高优先级（即，尽可能早地执行），而优先级65535将在所有低编号优先级处理完成后进行处理。具有相同优先级的函数调用的顺序是任意的；不过，变量始终在同一优先级的所有函数调用之前进行初始化。当没有明确给出优先级时，对象将在所有Priority变量/Priority函数之后进行初始化，随后，`main()`将正常调用。

Priority函数不得返回任何值，也不得带任何参数。必须注意，应避免使用未初始化的变量和调用可能使用未初始化变量的函数。如果项目在同名段（即`__attribute__((section("example")))`）内实现了多个函数，则无法应用Priority属性。同名段中各函数间的优先级不受支持。如果需要按顺序执行Priority函数，请考虑创建其他名称的段。

在在线更新应用中，可在分区交换后立即实现Priority函数，以便在调度下一个中断服务程序（Interrupt Service Routine，ISR）之前重新允许中断或设置立即使能电源传动系统控制算法所需的任何新变量。等待CRT初始化所有变量可能需要数百微秒，在此期间可能会丢失重要的ISR。应用于这些函数及其所需的全局/静态变量后，此Priority属性可提供一种抢占普通数据初始化流程的方法。

和任何应用一样，只要发生器件复位，CRT变量初始化程序就会被调用。执行引导交换指令（BOOTSWP）后（调用地址0x0），还会在在线更新的情况下执行CRT。这可确保堆栈初始化以及已升级固件的新变量初始化正确进行。不过，CRT现在将调用函数_crt_start_mode()，该函数通过检查NVMCON寄存器中的软交换位来确定是否应对所有非持久/非加载变量即未以隐性或显性方式指定Preserved属性的有限变量子集进行普通复位初始化。

例3： PRIORITY属性

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

软交换位SFTSWP（NVMCON<11>）显然是返回值的理想选择，因为该位在器件复位时由硬件清零并且通过执行BOOTSWP指令置1。如果_crt_start_mode()返回该位置1，CRT将使用.rdinit段中的数据进行初始化；否则，CRT将使用.dinit表中的普通数据。.rdinit初始化表中包含在分区交换之间需要进行初始化的所有变量以及Priority函数的地址。建议在完成所有初始化函数后清零软交换位。

CRT在内部将_crt_start_mode()定义为弱函数，因此将例4中所示的函数包含在应用源代码中将覆盖该函数，并允许用户返回软交换位以外的其他源的值（如果需要）。

例4： CRT启动模式

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

配置MPLAB® X项目

如前文所述，应用固件前一版本的.elf文件用于分配RAM变量。根据使用的保留模型，编译器/链接器将：

- 仅保留具有Preserved属性的变量
- 保留所选C文件中的全部变量
- 保留活动项目中的全部变量
- 保留整个项目（包括库归档文件）中的全部变量

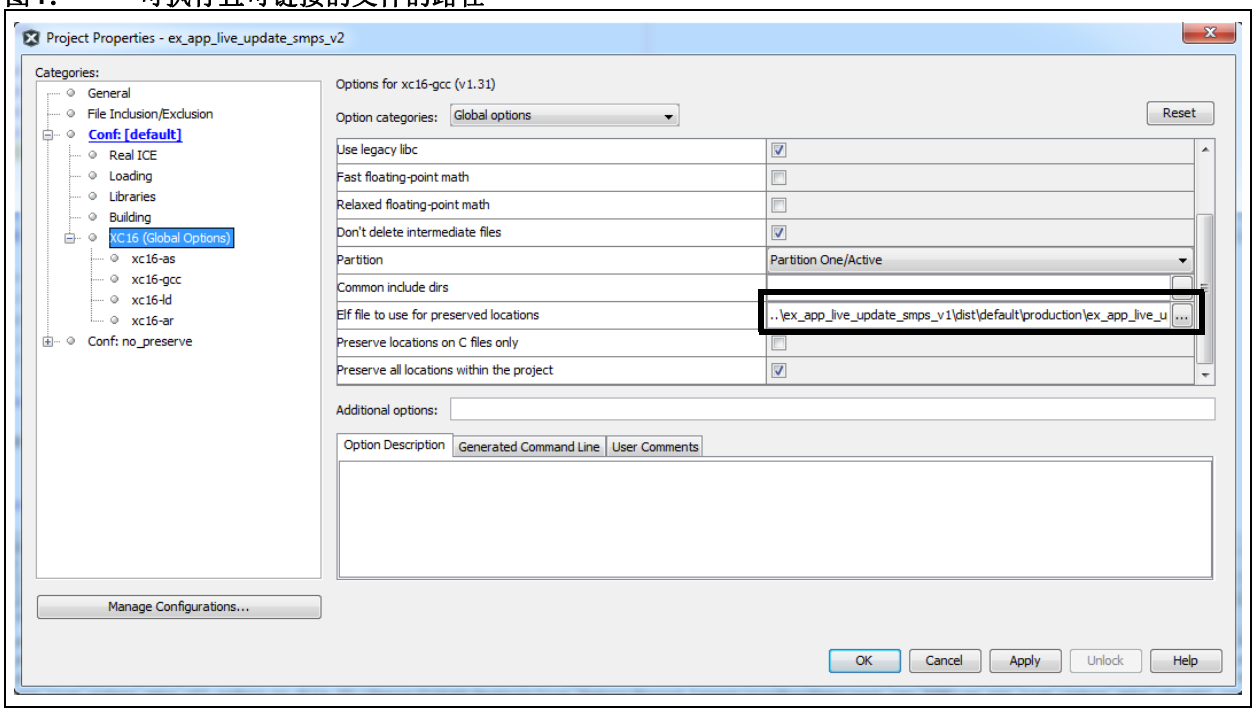
在上述全部四种情况下，链接器需要根据前一版本的信息来适当分配RAM。要在活动MPLAB X项目中为链接器设置.elf路径，请右键单击项目，然后选择**Properties**（属性）。选择**XC16 (Global Options)**（XC16（全局选项）），向下滚动，找到**Elf file to use for preserved locations**（要用于保留存储单元的elf文件）条目。在此，您可以直接浏览到.elf文件。（MPLAB X项目目录中的）.elf文件的典型存储位置如下：

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

> **注：** 对于每个固件版本，建议使用不同的目录来备份和引用.elf文件，以确保在清除或重新编译源项目时不会意外擦除或修改.elf文件。

在图1中，活动项目为版本2，链接器接收版本1的.elf文件来保留变量。

图1：　可执行且可链接的文件的路径



如果使用的保留数据模型仅依赖于标有Preserved属性的那些变量，则除了使能数据段外，MPLAB X项目不需要任何其他配置。

在项目属性的同一XC16 (Global Options)分类中的.elf文件选择下，有两个用于选择保留数据模型的复选框选项。第一个选项**Preserve locations on C files only**（仅保留C文件中的存储单元）告知编译器活动C文件中所有全局变量和文件作用域变量均需视为“保留”变量，并且要分配到.elf文件中之前使用的相同地址。C文件中的任何新变量均需要手动标记更新属性以从地址匹配中排除。如果新变量添加时未标记更新属性，则在链接编译阶段，工具链将发出这些变量无法映射到所引用.elf文件中等效变量的警告。该警告意味着CRT将使这些变量在在线更新期间处于未初始化状态，然后将其分配到任意未使用的RAM地址。

第二个复选框选项**Preserve all locations within the project**（保留项目中的所有存储单元）在需要保留尽可能多的变量（包括预编译归档、目标文件和汇编源文件中隐藏的变量）时使用。在保留所有/整个项目选项中，如果创建了额外的变量并且需要将其放置在已占用的某些存储器中（如near空间），则编译器可能无法链接。如果出现这种情况，可能需要**Preserving variables on a C file basis**（基于C文件保留变量）和重新初始化程序。

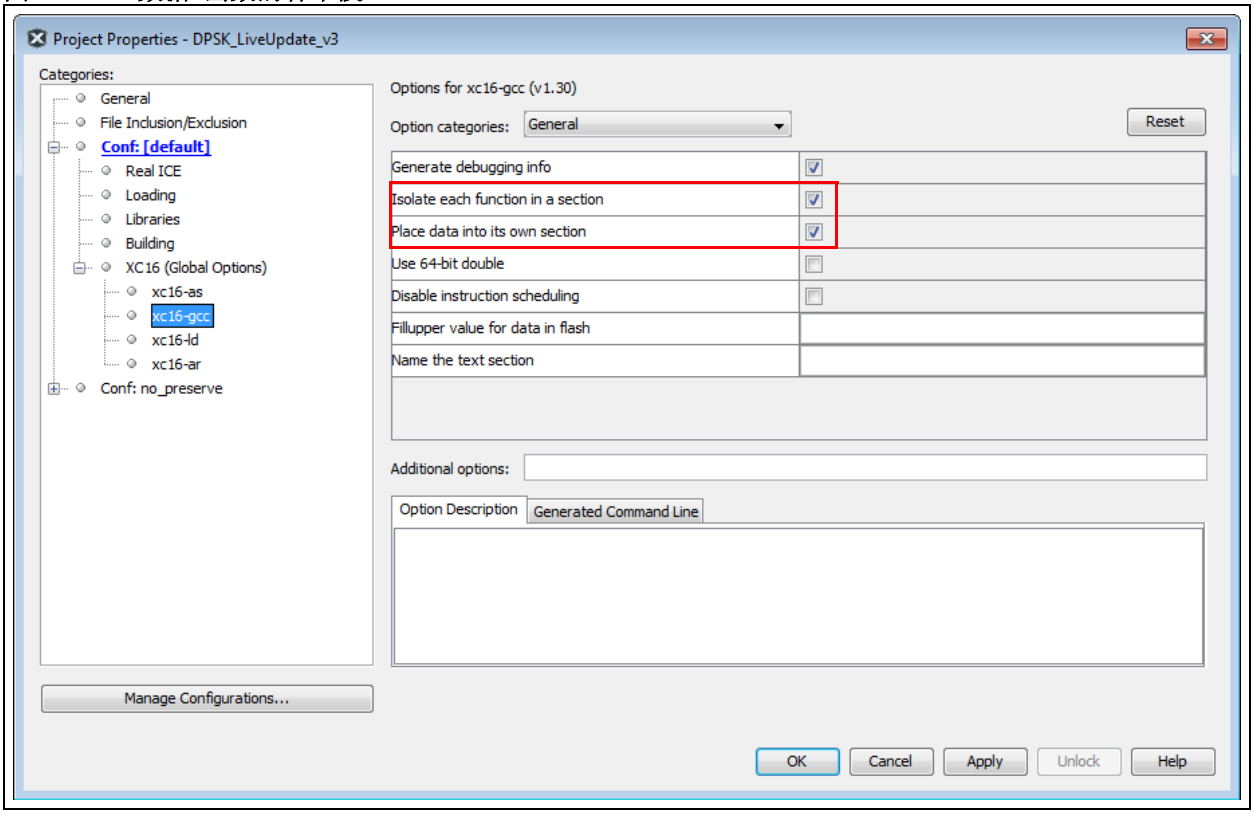
可以选择单独的C文件（而非上述全部C文件选项）来设置保留数据模型。在项目窗格中，右键单击需要保留变量的单独C文件。选择**Properties**，单击**Override build options**（覆盖编译选项）复选框。这将实现基于文件的配置。接下来，选择XC16 (Global Options)，向下滚动，选中**Preserve locations on C files**（保留C文件中的存储单元）复选框。在该配置下，不同版本中该特定C文件对应的所有文件作用域和全局变量均会保留。

此外，还建议使能编译器选项`-fdata-sections`和`-ffunction-sections`以支持全部在线更新应用。默认情况下，编译器会根据初始化设定（非零值、零或未初始化的数据）以及应用的任何属性（如near或persistent）将变量合并到通用段中。如果在分组变量的一个段中，新固件版本希望移除一些变量或包含N个元素的数组，则其历史占用的RAM存储单元通常无法恢复，因为链接器基于段分配粒度工作。此外，如果固件更新涉及扩展已合并到分组段中变量的宽度，则缺少链接器粒度会导致组中不相关的变量必须重新分配到新的RAM地址。这会增加在线更新的复制/重新初始化延时。

由于这些原因，最好将数据对象置于其自己独有的段中。工具链将能够重新使用新释放的RAM地址，并可避免不必要地重新分配与已修改变量相邻的保留变量。对于小于2个字节的变量，这会在闪存指令数量和RAM上增加一些额外的开销，但是在编译后续程序版本时，将为链接器提供最大的灵活性。使能`-ffunction-sections`对未来的在线更新限制没有特殊影响，但是当额外选中`xc16-ld linker`选项中的**Remove unused sections**（删除未使用的段）时，可以从应用中删除未引用的代码。结果是，尽管开销适中，但总代码大小通常仍会减小。

在**Project Properties**（项目属性）窗口中，选择**xc16-gcc**，然后选中**Place data into its own section**（将数据置于其自身的段中）以及**Isolate each function in a section**（将每个函数放入自己的段中）复选框。有关项目配置，请参见下面的图2。**Remove unused sections**位于**xc16-ld**子类下。

图2：　　数据/函数的各个段



软件实现

为在线更新应用创建固件涉及很多方面，包括可以管理代码版本控制和非活动分区编程/验证的智能自举程序以及无需预先进行任何异常处理即可接收代码映像并同步切换事件（以与窗口保持一致）的通信协议。

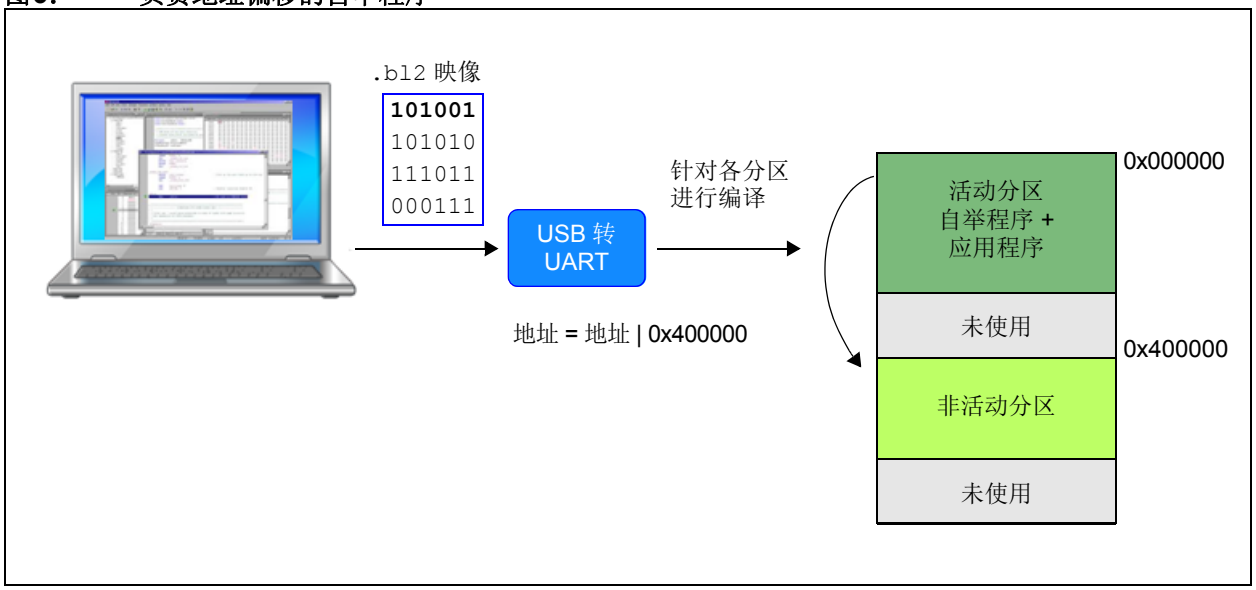
尽管自举程序固件可能特定于客户实现，但作为示例的基础，本应用笔记假定使用Microchip简易自举程序库（Easy Bootloader Library，EZBL）。欲了解EZBL的更多信息以及开发适合所需格式的二进制映像文件，请访问**Microchip简易自举程序**。

当编译完新的固件映像并通过多种通信协议（例如UART或I^2C）之一将其发送到自举程序时，自举程序的第一个步骤就是执行非活动分区擦除操作。这些双分区闪存器件具有用于擦除非活动分区的专用NVM操作码（NVMOP<3:0> (NVMCON<3:0>) = 0b100）。需要典型的NVM解锁序列来执行任何闪存擦/写程序。在此期间，应通过临时禁止中断来处理NVMKEY的连续写操作，以便处理非活动分区擦除命令。但是，在分区擦除期间，将允许中断，并且应用固件仍可完全运行。只需将中断屏蔽12个指令周期左右。

如果使用EZBL来擦除非活动分区，则只需调用EZBL_EraseInactivePartition()函数。该函数将配置并写入NVMCOM以开始擦除非活动分区。该函数为非阻塞函数（NVMKEY写入序列期间除外），将立即返回，无需等待NVM硬件完成擦除周期。

在启动非活动分区擦除之后，自举程序可继续编程和验证新的固件映像。EZBL通过实现EZBL_WriteROMEx()和EZBL_VerifyROM()函数来达成此目的。这两个函数将根据需要进行阻塞，以完成任何未完成的擦除/编程NVM操作，但后台中断处理将在此类延迟过程中正常进行。非活动分区始终驻留在绝对地址0x400000处，固件始终编译为从绝对地址0x000000（作为活动分区）开始执行。但是，为了消除以绝对分区1或分区2（具有与运行时状态相关的地址）为目标的.hex文件加载存储器地址中可能存在的差异，EZBL将强制所有编程记录处于0x400000范围内，这样只有非活动分区才可以从外部操作。有关更多信息（例如针对双引导模式配置器件的信息），请参见《dsPIC33/PIC24系列参考手册》中的**“双分区闪存程序存储器”**（DS70005156B_CN）。

图3：　负责地址偏移的自举程序



在新的程序映像的头部中，自举程序应接收与能够执行映像的系统硬件相对应的标识散列或其他惟一密钥。这消除了为不同应用编程不正确代码映像的可能性。此时，自举程序应该能够确定传入代码映像的固件版本。自举程序可以与当前活动的固件版本进行比较，决定是忽略传入代码映像的其余部分还是继续更新。

出于以下几个原因，固件版本控制对于在线更新应用至关重要：

- 确保对于已针对不兼容应用版本（RAM变量保留在不同存储单元中）编译的代码，正在运行的应用不会执行分区交换执行切换
- 当新应用的主要版本号或次要版本号指示向前或向后跳转不兼容时，使用软件复位来实现非在线更新回退执行切换
- 允许针对单个旧固件版本（而非所有历史版本）编译和测试可在线更新映像
- 当在诊断报告中提供简单的人机可读版本号时，可以提高产品的可维护性
- 可通过自举程序拒绝应用降级（可能指示用户错误）

在Microchip EZBL在线更新应用示例中，可以在ezbl_dual_partition.mk文件中找到标识字符串（以BOOTID开头）。Makefile将BOOTID字符串传送给编译时调用的ezbl_tools.jar Java实用程序，该实用程序会将字符串连接并压缩为固定宽度的SHA-256散列。然后将十六个散列字节附加到发送到目标器件并存储在闪存中的.bl2二进制映像的头部块中。APPID版本号也是头部块的一部分，存储在闪存中。该方法允许自举程序在接收实际代码映像的字节之前决定忽略还是接受传入的代码映像。这种早期识别方法通过在发生拒绝时抑制非活动分区擦除操作（可能包含有用的备用应用映像）来降低风险。此外，通过机器轻松比较二进制散列和版本号后，自举程序将被动地忽略用于其他目标器件的共享广播型通信介质上呈现的固件映像。

其他方法可能通过在一个固定的闪存地址处定义常量来进行识别和版本控制。在此使用模型中，自举程序需要将传入的代码映像编程到非活动闪存分区，并且只有在目标地址单元完成编程之后，代码才能针对预期目标进行验证，版本序列才允许在线更新。这种方法可能有缺点，因为在验证代码映像的适用性之前已编程非活动闪存分区。

进行识别和固件版本控制的另一个重要原因与闪存引导序列编号有关。应在完成所有其他闪存存储单元的编程和验证后写入非活动分区的FBTSEQ序列号。使用已知的合格映像时，如果应用在写入序列号之后、分区交换之前断电，则在器件上电时仍会选择新的代码映像。可以在分区交换后写入序列号，但这会使CPU停止几毫秒，因为闪存的最后一页需要全部复制/擦除，然后再进行编程。由于FBTSEQ配置字现在将驻留在活动分区中，双闪存分区的优势将不适用。

建议在成功编程非活动分区的闪存后，自举程序通过读取当前分区的序列号并减去1的方式来确定非活动分区的序列号。这将始终确保在器件上电时选择最新编程的应用（最低有效序列号成为活动分区）。

当非活动分区被擦除时，非活动分区的序列号被设置为无效擦除状态0xFFFFFF。如果不在项目中定义序列号，则该序列号将是无效的，直至由自举程序单独编程。如果在新固件更新事件期间断电，则会导致活动分区保持活动状态。这就是非活动序列号必须在验证完所有闪存之后单独编程的原因。只有在编程了非活动分区中的序列号后，非活动分区才会在下一次掉电再上电或复位时变为活动状态。

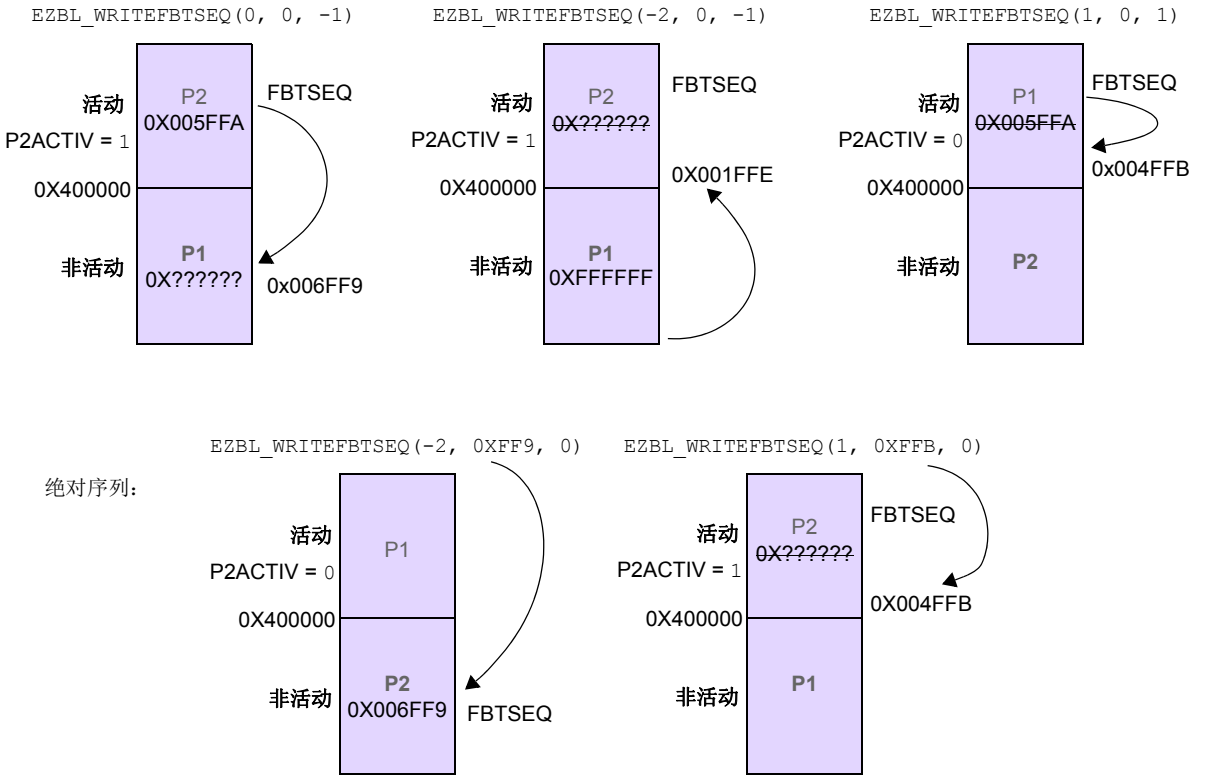
虽然建议使用以上方法来编程序列号，但EZBL_WriteFBTSEQ()函数支持写入任一分区并支持编程绝对值或相对值。该函数接受三个参数（要编程的目标分区以及要编程的绝对序列号和相对序列）并基于运算返回结果。任何小于零的返回值都表示失败。该程序引导序列函数将自动计算所提供的绝对或相对序列号的12位补码，以确保将有效的24位值写入FBTSEQ配置字。

目标分区参数选择向活动分区还是非活动分区（分别为1/0）写FBTSEQ字。由于最常见的实现将始终以非活动分区为目标，该参数将设置为0。但是，该参数也可以设置为-1或-2，分别表示绝对分区1或分区2。为使用该选项编程序列号，需解码P2ACTIV位的状态以确定分区1或2当前所在的地址。

如果需要传送序列号，则可以使用绝对序列参数。无论传入哪个12位值，都将写入指定目标分区的序列号。对于该选项，相对序列参数应为0。

相对序列号实际上是一个有符号整数，将被加到非目标分区的序列号中以确定目标分区的序列号。如果该输入参数为负数（即-1）且目标分区参数为0，则非活动分区的序列号将变为比当前活动分区的序列号小1。只要后续发生复位，就会选择当前的非活动分区来执行操作。对于该选项，绝对序列参数应为0。图4提供了EZBL_WriteFBTSEQ()函数支持的几个示例。

图4：　　闪存分区序列写入示例(1,2,3)



注 1： “0x??????”表示任何现有的FBTSEQ内容，包括无效的0xFFFFFF擦除状态。

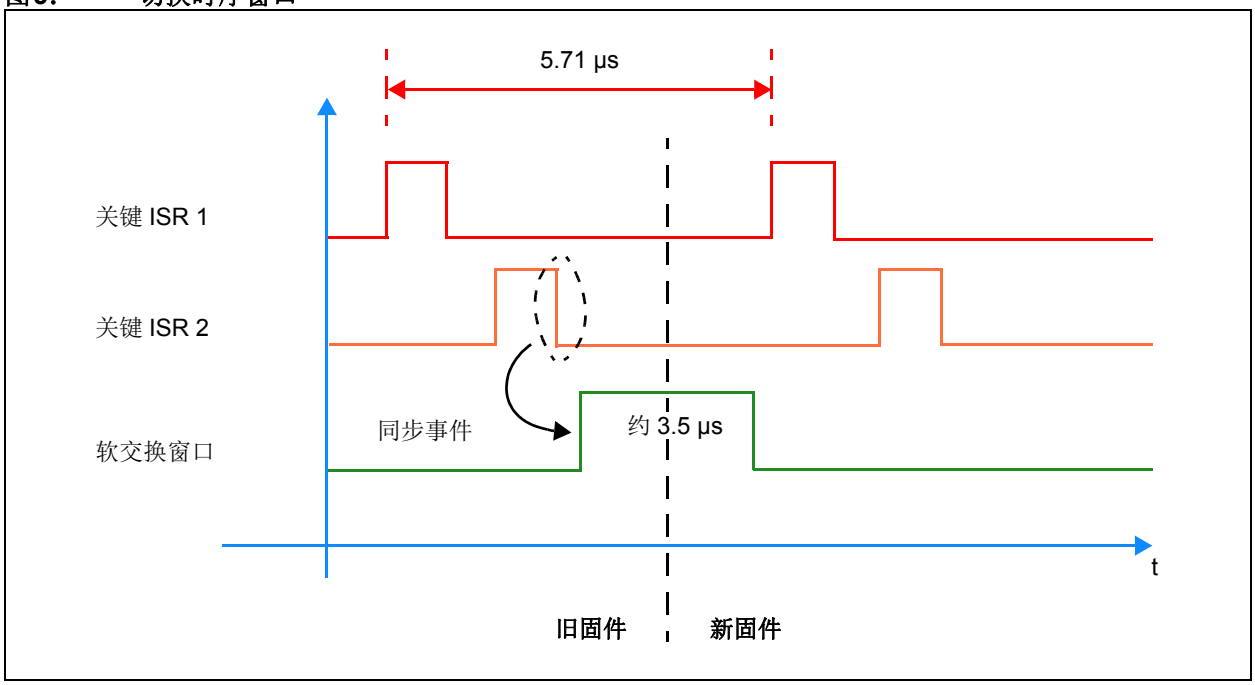
2： 如果可以预见将有超过4095次固件更新，则必须特别注意要正确地复位两个序列号。由于这似乎不太可能发生，因此EZBL_WriteFBTSEQ()不包含处理这种情况的代码开销。可通过使用绝对序列号两次调用EZBL_WriteFBTSEQ()来执行此类复位。

3： 如果在非擦除状态下写入活动分区的序列号，则需要擦除配置字所在的整个闪存页。闪存页将进行堆叠、擦除和恢复，更新的序列号除外。这需要禁止中断，并且对于在线更新应用而言不是最佳情况，因为页擦除和写入操作会使CPU停止执行操作。建议只修改非活动分区的序列号。

切换时序

在线固件更新最重要的要求是在固件版本之间实现无缝切换。这意味着在固件更新期间，电源输出或电机驱动应保持连续不中断。为在线更新应用开发固件的关键是能够同步到不执行关键ISR的时序窗口，以便启动软交换事件。较低优先级的函数/中断可能会长时间禁止，但不影响主要应用性能。有关切换时序窗口的示例，请参见图5。

图5：　切换时序窗口



在图5中，电源传动系统需要两个关键的ISR，所有其他应用中断均禁止，直至分区交换结束。每个ISR以175 kHz的频率执行，处理时间约为1 µs，留出3.5 µs的窗口来执行分区交换和优先级重新初始化函数。请注意，分区交换将与该时序窗口同步，以便在下一个关键ISR调度执行之前分配最长时间。

在本例中，应用固件负责将分区交换与关键ISR 2同步。非活动分区完成编程和验证后，固件会立即禁止所有非关键ISR并将仅在关键ISR 2中置1的软件标志清零。只有在软件标志置1后，才能执行分区交换函数。另一种方法是将软件标志清零，并从关键ISR 2中调用分区交换函数。因为这会在执行速率较高的循环内添加条件语句，因此前一个实现方法更为理想。如果没有用于切换事件的某种类型的同步程序/标志，可能会丢失关键ISR，这可能会影响应用程序的运行。

大多数关键切换时序取决于关键变量初始化和重新初始化类型函数的应用要求。

不过，有一项固定时序要求不是特定于应用的，其中包括器件初始化，如堆栈指针配置和处理器状态寄存器的一般设置。在此切换时间间隔内，通常会发生以下情况：

1. 禁止所有中断。
2. 执行NVM解锁序列和`BOOTSWP`指令。
3. 跳转到地址0x0，这是活动分区寻址空间中现有新应用程序的入口点。
4. XC16编译器运行时（CRT）将初始化堆栈、堆栈指针限制和CORCON（如果需要）。
5. 检查软交换事件并开始初始化priority变量及调用priority函数。
   a) 执行初始化关键变量并重新允许关键中断的用户定义函数

在允许时序关键型中断后，CRT将继续初始化非优先更新变量并最终调用`main()`。

例5给出了一个执行分区交换序列的简单宏。该宏应在主函数内部执行，并通过使用全局中断允许位GIE（INTCON2<15>）禁止中断。为了在分区交换后重新允许中断，新的代码映像需要适当地将GIE位置1。

例5：　　分区交换序列宏

```
#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")
```

紧跟在BOOTSWP指令之后的指令必须是驻留在当前活动分区但会跳转到新活动分区中的地址的单个24位指令。在典型应用中，地址0x0包含一条GOTO指令，该指令指向一个C运行时入口函数的地址。在这种情况下，建议在BOOTSWP后由寄存器直接调用地址0x0。该指令满足24位大小限制并且用零扩展目标地址使其变为绝对量（即，非PC相对地址）。CALL优先于GOTO，因为它可确保堆栈帧活动SR位被清零。

简易自举程序库包含EZBL_PartitionSwap()函数，该函数禁止所有中断（通过清零所有IECx寄存器）、管理BOOTSWP指令的NVM解锁并执行到地址0x0的跳转。

尽管建议在main()函数级别执行切换事件，但EZBL_PartitionSwap()函数实现允许在任何级别的中断程序中执行分区交换。该函数覆盖堆栈的返回地址以指向0x0并执行从中断返回指令（RETFIE），这将自动复位IPL状态。即使所有中断均被EZBL_PartitionSwap()函数禁止，仍建议在调用此函数之前禁止所有非关键中断，以确保较低优先级的中断不会过早在软交换窗口中干扰同步。

有关非应用特定切换时序的更多信息，请参见本应用笔记。CRT初始化核心编译器环境SFR（即W15堆栈指针、SPLIM和DSRPAG等）后，将立即初始化/执行所有priority变量和函数。此后，关键的priority函数和变量将决定应用程序级别的切换时间。这很大程度上取决于应用程序认为哪个因素对于维持正常工作更重要。显然，要初始化的变量和要执行的关键程序越多，完整应用的切换时间就越长。如果应用需要进行重大更新（这些更新不适合已有切换窗口），则可以按顺序执行多个分段固件更新，或者在编程新代码映像后，使自举程序回退到非在线更新复位状态。

由于所有priority函数均在冷启动和软交换事件中执行，因此可能需要其他软件（例如测试软交换位的条件语句）。将该条件语句添加到主函数中可避免在进入主函数时重新初始化单片机外设/时钟，并允许执行满足软交换条件的任何剩余初始化程序（例6）。

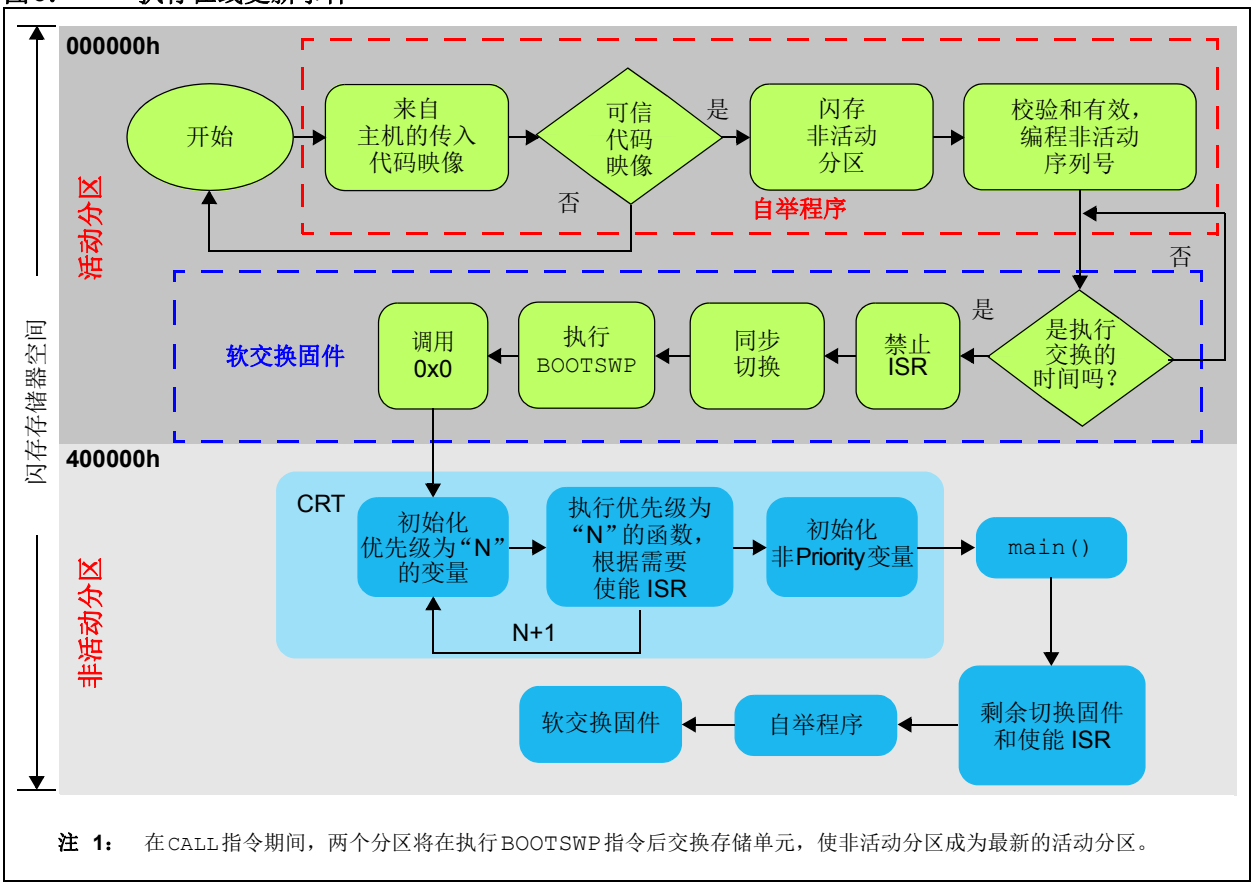
注： 务必记住，在软交换期间执行的固件还可能需要在应用下次上电时（即外设初始化）通过冷启动执行。

例6： PRIORITY函数实现

```
void __attribute__((priority(x)) FunctionInit(void) {
    if(!_SFTSWP) {
    // 优先级x冷启动初始化函数
       return;
    }
    // 优先级x软交换初始化函数
}
```

图6给出的框图说明了上述关于执行在线更新事件的整体步骤。

图6： 执行在线更新事件(1)



注 1： 在CALL指令期间，两个分区将在执行BOOTSWP指令后交换存储单元，使非活动分区成为最新的活动分区。

软件限制

作为一种高级语言，C源文件支持多种编程结构，当试图保留当前项目中与历史编译项目可执行映像相关的变量地址时，可能导致歧义。在源代码内容、包含头文件、文件重命名或优化更改最少的情况下，工具链可能无法证明在当前项目中的一个变量是否等于旧的变量。本节涵盖了编译正确、一致的在线更新应用程序可能出现的一些问题以及所需的手动解决方案。

注： 为了尽量减少未来在线更新应用程序的开发工作，请避免使用静态局部变量，并确保静态文件作用域变量在整个项目中采用惟一名称。

静态变量

由于全局变量在多个文件中使用，因此自然需要采用惟一名称，这些变量通常可以保留以及由XC16自动处理。同样，自动存储类对象（如典型函数的局部变量）在运行时堆栈中临时分配，这使其成为非持久对象，但这并不属于在线更新问题（认为它们在分区交换时超出作用域）。

但是，在文件作用域和函数局部作用域内静态分配的变量不需要在整个项目或源文件中采用惟一的名称。对于这些变量，编译器会将其置于名称惟一但不可预测的段中。例如，静态局部变量`cnt`或`i`可以本地驻留在几个不同的函数中。编译器将通过追加一个存储限定符标记和一个四位十进制数（`.nbss.cnt.1234`）将这些重复出现的变量中的每一个都分配到具有惟一名称的段（如果没有明确指定属性）。由于这些段的名称不可预测，可能会随着代码和编译器优化的变化而改变，因此保留静态局部变量和文件作用域变量相当困难。

如果静态局部变量意外地保留在项目中，并且需要保存该变量，则链接器可能在编译新的在线更新应用程序时发出警告或错误。为了解决此类问题，变量的声明需要使用address属性进行修饰，以强制分配到正确的地址。例7会将局部变量`bootUnlockState`强制到地址0x0000105E。

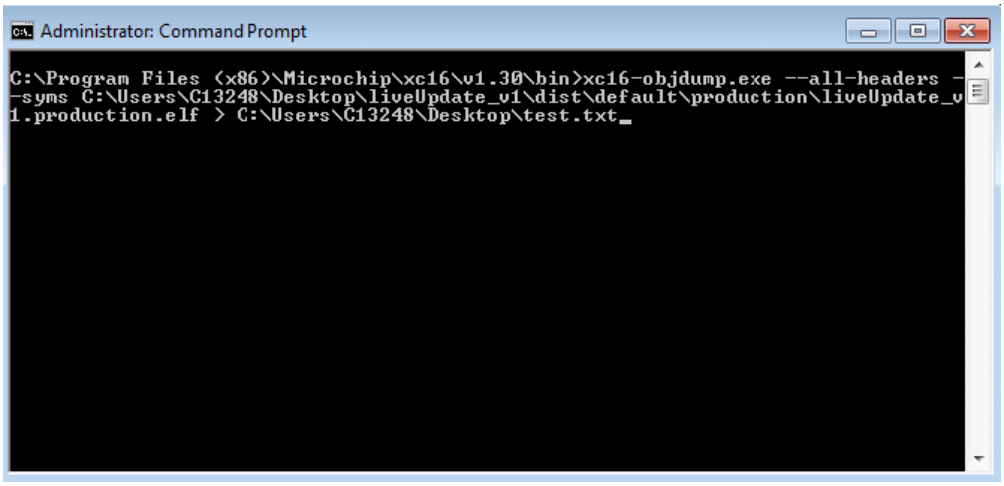
例7： 固定地址变量

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

确定等效历史变量的地址并不简单，因为这些类型变量的符号名称并不总是显示在映射文件中。XC16确实提供了一个接受`.elf`文件的目标转储可执行文件，该文件通常包含比映射文件更多的信息。

使用命令窗口，将当前目录更改为XC16安装文件夹。对于默认安装文件夹，路径将如图7所示。

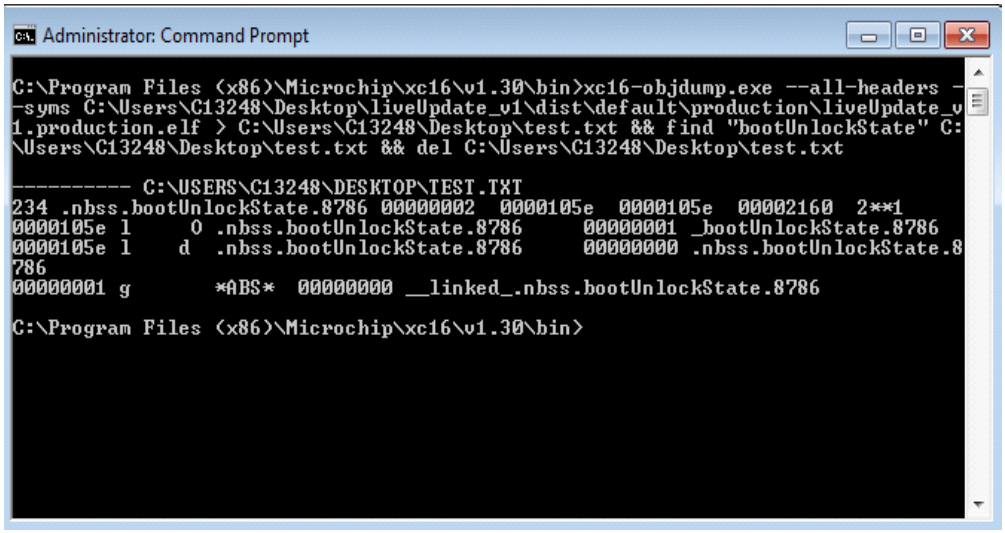
图7： 用于生成目标转储文件的命令提示符



目标转储可执行文件（xc16-objdump）位于**bin**文件夹中。指定编译器转储所有头部（**--all-headers**）和符号（**--syms**）信息通常足以找到变量信息。要了解可执行文件支持的更多选项，请在xc16-objdump可执行文件后添加**--help**。需要通过输入文件的路径来完成目标转储，这是前一个固件版本的.elf输出。最后，stdout将重定向到一个临时文本文件以查看和搜索目标信息。在以下示例中，它将保存到桌面上的test.txt中。

可以在该文本文件中搜索符号名称（如图8所示），并在命令提示符中显示地址信息，而不是打开.txt文件进行手动搜索。如果不需要保存文本文件以供将来使用，还可通过命令提示符将其删除。

图8：　通过变量名称查找地址单元的命令提示符



此删减转储的第一行是.elf头部中的一个段定义，而其余行来自符号表。第二列包含符号的单字符标志，其中“O”表示RAM或PSV const存储区中的数据对象。

第一列中符号的地址通常是尝试恢复所保留局部静态变量的地址所需的信息，该局部静态变量无法通过工具链自动映射。

指针

在线更新应用程序通常需要特别注意RAM中全局或静态分配的指针。由于闪存内容不会保持任何模型下保留的地址，因此指向闪存的任何指针（例如调用函数、const/PSV数据数组和const 字符串指针）均需要先在新固件中重新初始化，然后才能使用。 **“应用示例”** 给出了相应示例。

大多数情况下，假设保留的RAM指针将指向已保留的RAM数据对象。指针也可能指向SFR，它在本质上由硬件保存。因此，这些指针可能不需要在线更新重新初始化。如果出于任何原因，指针所指向的内容被移动，如由于数组元素的数量或每个元素的大小增加而未保留或仅被重新定位，则指针以及新的RAM内容需要在发生在线更新事件时进行初始化。

如果应用程序具有可保留旧内容和指针以及新变量的RAM空间，则可以计算合适的偏移量并将其应用于新RAM内容，如图9所示。

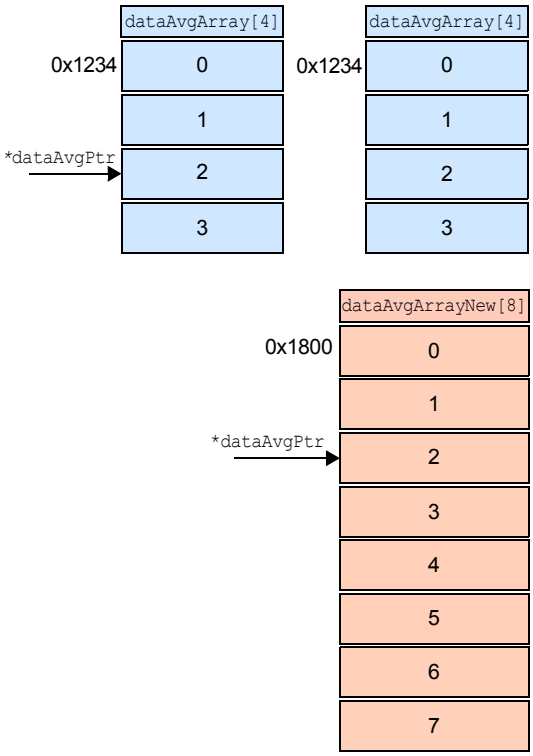
图9： 指针/RAM初始化(1,2,3)

V1 固件：

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

V2 固件（保留所有模型）：

```
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)        // 普通复位
    {
      dataAvgPtr = dataAvgArrayNew;
      return;
   }

   // 在线更新—复制现有数组数据
   for(i = 0; i < sizeof(dataAvgArray)/sizeof(dataAvgArray[0]); i++)
      dataAvgArrayNew[i] = dataAvgArray[i];

   // 确定新的指针目标
   dataAvgPtr = &dataAvgArrayNew[((unsigned int)dataAvgPtr -
(unsigned int)dataAvgArray)/sizeof(dataAvgArray[0])];
}
```



注 1： dataAvgArray[] 的声明保留在V2中，以允许对现有在线更新内容进行指定访问，并确保分配的RAM不会被无关的新（更新）变量破坏。V2到V3的在线更新项目可删除数组来恢复其RAM。

2： 指针和两个数组都具有Persistent属性，以防止使用零进行隐式CRT初始化。如果不具备持久性，则dataAvgArray[]的冷启动初始化程序将浪费代码空间，dataAvgPtr将在ArrayInit()priority函数设置正确值（仅限冷启动）后灾难性清零，dataAvgArrayNew[]（作为更新变量）将在在线更新事件期间ArrayInit()返回后灾难性清零。

3： dataAvgPtr 的值是使用数组索引和整数算术计算的，因为这样可以根据需要使用不同的数据类型重新声明目标数据类型。例如，*dataAvgPtr可更改为(uint32_t *)，dataAvgArrayNew[]可更改为8个uint32_t并且dataAvgPtr == &dataAvgArray[1]将转换为dataAvgPtr = &dataAvgArrayNew[1]，而不会导致不对齐或类型转换错误。

可以通过存储器访问范围内的本地临时计算方法对数组进行索引，也可以通过可使用并且能够始终保留的单独索引变量（偏移量）进行索引，而不是递增用于访问数组内容的静态指针。在某些情况下，可以通过写入固件来确保在执行分区交换之前，偏移始终为零（以数组为基址）。这将确保重新初始化指向新基址的指针将使指针始终指向新固件版本中的适当索引。上述内容仅仅是建议，因为如果应用程序需要更新指针和/或指针指向的RAM内容，则有许多方法来处理指针重新初始化。

在处理应用程序库时，许多相同的软件限制都适用，但现在遵循这些建议更为重要。这是因为不能总是重新编译库，尤其是库归第三方所有时。如果可能，建议在编译库项目时使能`-fdata-sections`，以确保在链接在线更新项目时保持最佳的变量操作粒度。

表1总结了使用保留所有模型时不同在线更新情形的复杂性。

表1： 在线更新情形

固件更改的级别	示例	寄存器和RAM变量	新的固件要求
简单	更改固件常量 • 补偿器系数 • 查找表 • 故障阈值限值等	没有新变量，也没有SFR更改	将新常量加载到变量存储单元（如果未存储在闪存中）
	增加/删除固件	—	—
	删除变量	孤立变量	—
	本地静态变量	—	将address属性添加到变量中以适当映射
中等偏下	新增变量（非关键，标记有Update属性）	新建变量	如果标记有persistent属性并且在priority函数中使用，则执行新变量初始化，否则进行CRT初始化
	减少数组中的元素数量	孤立变量，可能重复使用地址单元	可能需要复位索引
	修改外设	修改SFR	外设初始化函数支持在线更新和冷启动
	指向闪存中函数、const字符串和PSV常量的指针	—	重新初始化指针
中等	新增变量（时序关键型，标记有Update属性）	新建变量	通过高优先级函数初始化变量
	增加数组元素的数量或修改结构体（即对成员进行增加/删除/重新排序）——非时序关键型	新的变量大小需要附加数组/结构体，同时保留旧内容	变量重映射函数和偏移计算固件，需要更新指针
	时序关键型固件在软交换后立即执行	新变量	应用特定
复杂	需要变量重映射的时序关键型固件： • 新的补偿器算法 • 增加数组元素或修改结构	新的变量大小需要附加数组/结构体，同时保留旧内容	变量重映射函数和新变量需要在高优先级函数期间初始化
	更改数据类型但保持相同的大小	—	通过固件修正数据类型

应用示例

数字电源入门工具包示例

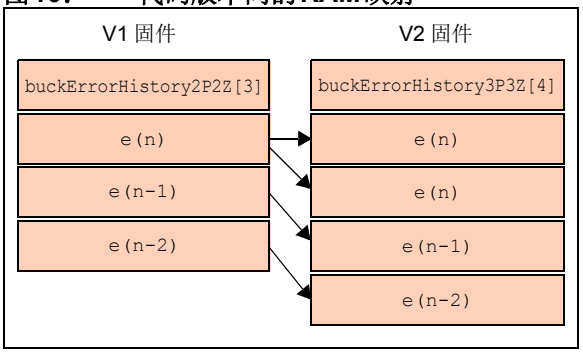
为了说明不同的保留数据模型以及如何进行在线固件更新，数字电源入门工具包（Digital Power Starter Kit，DPSK）上创建并执行了三种不同的固件版本。有关DPSK的详细信息，请访问**智能电源**网页。这些项目的源代码在EZBL下载区发布。版本1固件是最初使用的固件，其中包含降压和升压转换器的电压模式控制（控制算法在175 kHz下执行）、调用自举程序相关函数的软件任务调度程序、LCD显示函数、故障管理、通过I^2C实现的负载瞬态软件以及UART通信软件。

在本例中，降压转换器2P2Z补偿器性能被有意设计为不稳定。将示波器探头（交流耦合）连接到降压转换器的输出电压测试点（TP5）后观察瞬态响应，将呈现出非最优化设计的补偿器特性。首个在线固件更新事件将改善降压转换器补偿器性能，并增加额外的软件功能，例如计算切换时间和增加显示时序事件的LCD显示选项。

第一个在线更新事件将使用保留所有模型（“保留项目中的所有存储单元”复选框）。所有新变量都将标记有Update属性和Priority属性（如果需要）。在V2固件中，2P2Z补偿器由设计合理的3P3Z补偿器代替。3P3Z补偿器需要优先级初始化函数作为关键时序路径的一部分，因为在固件切换期间降压转换器输出必须保持稳定。2P2Z补偿器的系数已被删除，但需要控制历史记录和错误历史数组来初始化3P3Z补偿器数组。图10以图示形式说明了如何将2P2Z补偿器错误历史记录中的数据从一个RAM存储单元复制到3P3Z补偿器数组存储单元，例8则给出了在在线更新分区交换之后立即初始化新补偿器的固件。

注： 在在线更新应用程序中，关键路径中调用的一些初始化程序可能也需要通过冷启动执行。由于某些变量可能尚未初始化，因此可能需要在软件（即`main()`）的其他位置调用相同的初始化程序。

图10： 代码版本间的RAM映射



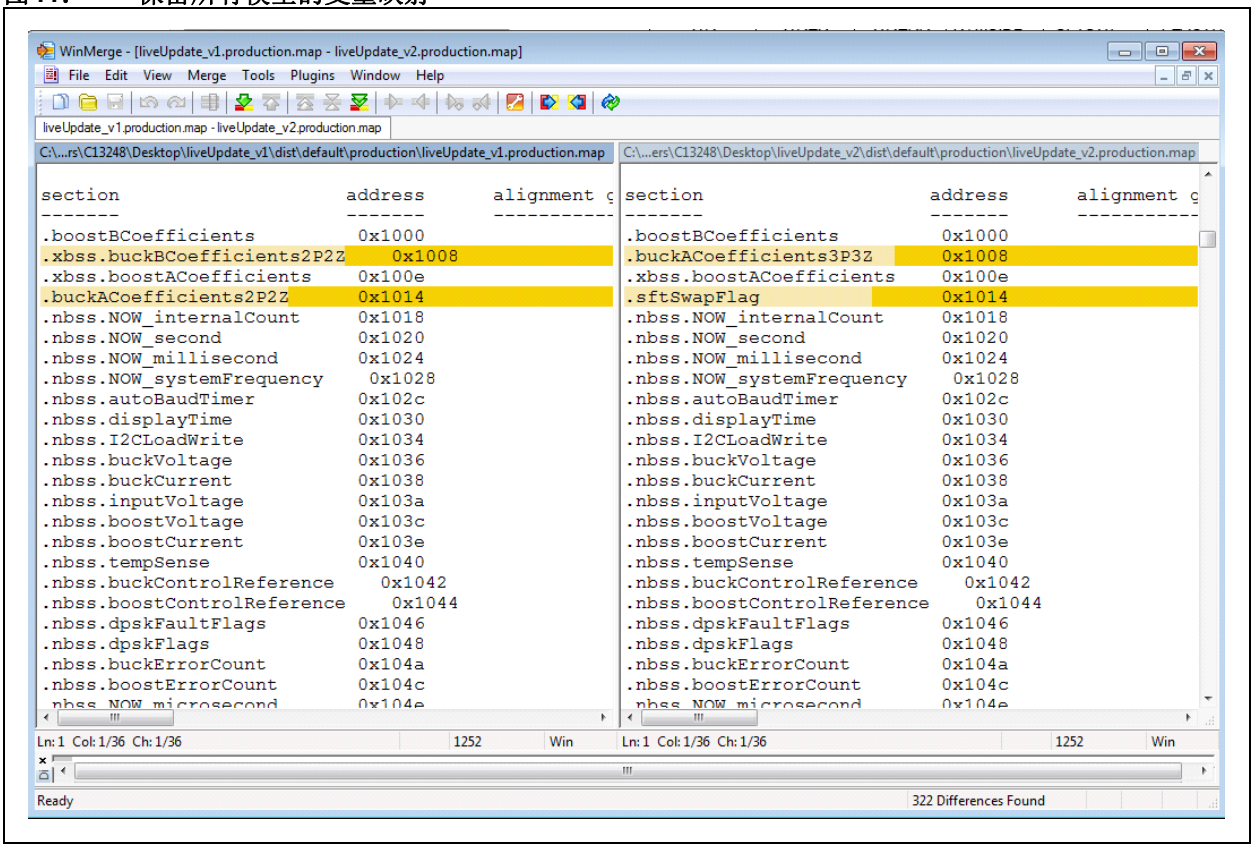
例8： 关键初始化函数(1)

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

    if(!_SFTSWP) {                  // 冷启动时返回
            sftSwapFlag = 0;
            return;
    }

    sftSwapFlag = 1;                // 标志在此优先级进行初始化而非借助CRT

    // 重新初始化任何时序关键型对象
    InitBuckComp();                 // 加载新的补偿器参数并初始化W寄存器

    // 加载控制/错误历史记录——根据需要推断值并从旧变量
    // 复制
    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];

    // 使能补偿器ISR，因为降压补偿器已重新初始化
    _ADCAN1IE = _ADCAN3IE = 1;
}
```

注 1： 尽管此函数被视为项目中最为时间敏感的代码，但仍分配属性`priority(100)`而非`priority(1)`。使不同优先级之间保持一定编号差，可在未来实现一些其他更具时间关键特性的代码，而不必返回此函数并将其切换至低优先级/高优先级编号。

使用文件比较工具（如WinMerge或Beyond Compare），可以检查版本1和版本2的输出映射文件。由于所有数据均放置在单独的段中，V1固件中的一些变量已被孤立，并且这些存储单元已被V2固件重用。在图11中，输出映射文件显示了3P3Z A系数的全新6元素数组，该数组位于2P2Z B系数的6元素数组之前所在的位置。

图11：　保留所有模型的变量映射



进一步检查映射文件会发现其他新变量所在的位置，还会发现V2固件中的所有变量都已正确映射到V1固件。

执行第一个在线更新的步骤

要执行此第一个在线更新事件，首先要使用板上PICkit™（PICkit™ On-Board，PKOB）或其他编程工具（如MPLAB REAL ICE™或MPLAB ICD）对带有版本1固件的DPSK进行编程。将MCP2221 USB转UART转接板（跳至3.3V）连接至DPSK上的连接器J1。所需的三个连接引脚是TX、RX和GND。这三个引脚（1、3和6）从转接板一对一映射到DPSK板。将mini-USB电缆连接到PC，并确定USB设备枚举时使用的COM端口。

控制面板→设备管理器→端口（COM和LPT）

有关转接板的更多信息，请访问**MCP2221转接模块**页面。使用此实现执行在线更新事件时将需要端口信息。

运行时，应用应显示降压/升压输出电压和负载设置。按下开关SW1会将LCD显示内容切换为输入电压和温度读数以及活动分区。通过降压转换器输出上的示波器探头，可以观察到瞬态响应。活动分区显示基于NVMCON特殊功能寄存器中P2ACTIV位的设置。该位被读取时将区分哪个分区是活动分区。

接下来，使用MPLAB X IDE打开V2固件，并导航到`ezbl_dual_partition.mk` Makefile，它可以在重要文件下找到。打开文件并更改通信端口以匹配USB枚举时使用的端口。

--communicator -com=COMX

编译V2固件将调用ezbl_tools.jar应用程序，该应用程序将.elf衍生输出转换为二进制.bl2文件，并通过COM端口将此数据发送到dsPIC33EP64GS502目标器件。V1固件将接收UART数据并将V2固件编程到非活动分区。完成所有数据的编程和验证后，V1固件将启动BOOTSWP事件。

将一个示波器探头连接到降压转换器输出（TP5），并将另一个示波器探头连接到J3的引脚5后，可以捕捉到切换事件。图12演示了此在线更新事件，说明了在切换事件期间降压转换器输出不受影响。

图12：　从V1到V2的无缝固件切换

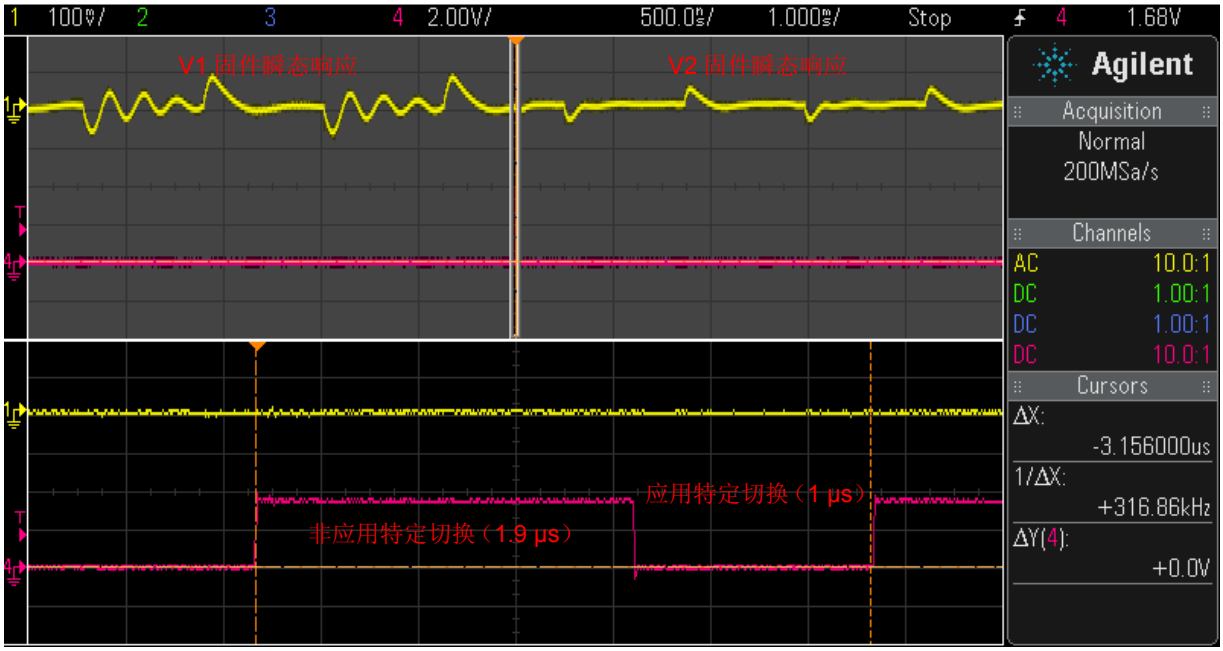


图12的上半部分显示的是每1.2 ms（1 ms/div）进行阶跃负载切换时应用的交流耦合瞬态响应。底部图是在线更新执行切换发生时的放大视图（500 ns/div）。

补偿器性能的变化非常明显。图中捕捉到从V1固件的EZBL_PartitionSwap()函数调用到V2固件的CriticalInit()函数的切换时间（通道4高电平脉冲时序）。

这是**“软件实现”**部分中讨论的非应用特定切换时间。这是所需的最小EZBL_PartitionSwap()和CRT启动时间，因为CRT在调用CriticalInit()之前不会初始化任何变量，并且项目中也没有声明任何更高优先级的初始化函数。持续时间通道4为低电平表示应用特定的时序，它构成了使用新的3P3Z补偿器重新使能电源传动系统的关键软件。

按下开关SW1将呈现一个新的LCD显示画面，用于显示关键交换时间，该时间应该在1.9 µs左右。CriticalInit()函数额外花费1 µs的时间来处理总停机时间大约3 µs的事件。对于此在线更新实例，可用的软交换窗口为3.5 µs，因此固件更新未导致降压和升压转换器丢失或延迟中断。

在使用保留所有模型的应用示例中，一些指针存储在RAM结构中并指向闪存函数，需要在执行任何引用这些指针的代码之前重新初始化。由于这些指针与UART通信和自举程序任务相关联，因此它们不会在关键路径中初始化。相反，这些指针在SecondaryInit()priority函数内更新。重新配置指针后，可以允许剩余的所有中断。例9给出了完成在线更新事件以及使应用程序完全运行所需的剩余priority代码。

注： 工具链在不同版本实例之间移动闪存的内容。存储在RAM中且指向闪存的任何指针（例如存储在RAM结构中的函数回调指针）都需要在在线更新事件中重新初始化。确定应用程序中存在哪些指针是一个手动过程，最好记录应用程序中的所有此类指针，以确保在在线更新事件中正确地重新初始化。未能将RAM指针重新初始化为闪存对象将导致应用程序出现意外行为，通常是地址错误陷阱或无效操作码复位。将address属性应用于函数可以将代码强制到其历史存储位置。但是，在较大的器件上，这并不能保证保存在RAM中的函数指针保持有效，因为XC16可能会用16位链接器生成的句柄替换函数的真实地址。句柄是跳转到超出16位寻址限制的代码所需的“GOTO trampoline”指令的地址。trampoline的存储单元不能被分配给绝对地址。

例9： 闪存指针初始化

```
void __attribute__((priority(200))) SecondaryInit(void) {
    if(!_SFTSWP) {
        return;
    }
    // 修正I2C1_OnWrite函数回调指针
    I2C_Tx.onWriteCallback = I2C1_OnWrite;
    _MI2C1IE = 1;          // I2C 1主模式动态负载切换通信代码

    // 修正UART FIFO函数回调指针
    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自举程序FIFO代码
    _U1RXIE = 1;           // UART 1 RX EZBL自举程序FIFO代码

    // 修正用于调用自举程序的NOW任务函数指针
    EZBL_bootloaderTask.callbackFunction = EZBL_BootloaderTaskFunc;
    _T1IE = 1;             // 定时器1 NOW节拍计时代码
}
```

依赖性更低的在线更新

为了演示不同的保留数据模型，需要第三个软件版本，但此次MPLAB X项目将删除保留所有数据模型。用于地址保留的.elf文件现在指向版本2固件的.elf输出。在这种情况下，只有少数几个关键变量手动标记为Preserved属性。这些是保持电源传动系统完全运行的变量，如降压/升压控制器要求（控制参考、控制历史记录、故障历史记录和A/B系数）以及系统状态标志。所有其他变量和外设SFR均将在分区交换时重新初始化。

在单片机上运行V2固件时，右键单击V3项目并选择版本（确保ezbl_dual_partition.mk文件选择了正确的COM端口）。这会将V3固件加载到非活动分区中，并执行分区交换（如上所述），同时使固件从V1切换到V2。

使用此全新固件版本（在线更新情形）时：

- 降压 / 升压转换器保持完全运行的稳定状态。关联的变量应用了Preserved属性。
- 固件更新中更改了升压负载。
- CriticalInit()函数删除了降压错误/控制历史记录的V1到V2 RAM副本以及降压转换器的初始化程序。
- 无需在SecondaryInit()函数中重新初始化指针。

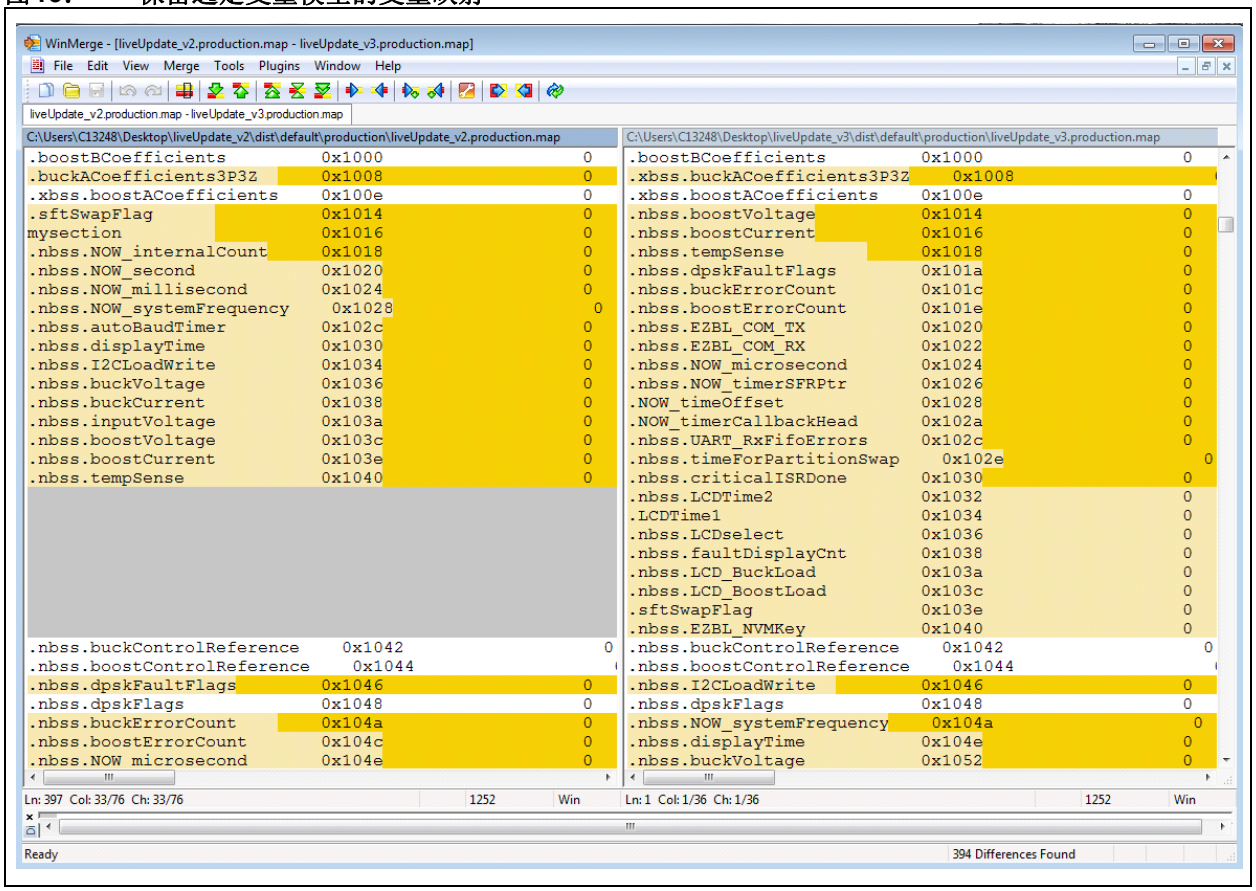
此外，在线更新到V3会在重新启动main()时调用这些初始化程序，即使未曾有意更改相关功能也是如此。需要重新初始化，因为其变量未保留，并且可以在链接期间自由移动到新地址：

- LCD屏幕显示变量已重新初始化；因此，LCD将复位为主屏幕。
- EZBL_BootloaderInit()重新初始化了UART通信、时序外设和与之相关的变量。需要可避免重新执行振荡器初始化代码的SFTSWP条件语句来防止PWM和ADC临时中断。
- I^2C外设（用于控制输出负载）和通信变量已重新初始化。

在本例中，总交换时间比从V1到V2固件的时间长得多，因为大多数应用程序变量已在引导交换事件时通过CRT重新初始化。随着结构体不断增长、结构体内的变量类型不断变化，或者因为外部软件库正在更新而无法确定其中的更改是否可以安全地重用先前库版本中实现的现有状态变量，可能需要这种特定的在线更新情形。不论原因如何，大多数变量都将重新初始化，并且需要注意的是，某些函数可能需要从软交换事件中调用，以便所有应用程序功能恢复正常运行。对于LCD重新初始化这样的事件而言，这可能造成产品经历了器件复位的错觉，但实际上发生的是选定子系统的受控复位。

图13给出了V2和V3固件之间的一个编译实例的映射文件。与预期一样，只有标记为preserved属性的变量才会保持其原始地址。

图13：　保留选定变量模型的变量映射



有效代码映像接收

ezbl_dual_partition.mk Makefile与EZBL Java应用程序一起创建包含供应商、模型、应用程序名称和任何其他标识字符串的SHA-256散列。该散列密钥被分解为多个32位块，然后以符号形式传送给链接器，并最终成为应用程序的一部分传送给器件。如**“软件实现”**部分所述，此信息用于确保接收的代码映像用于此应用程序。只有在散列验证完成后，自举程序才会执行代码映像的下载操作。

例10所示为这些应用程序的散列密钥字段。

例10： BOOTID_HASH

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

在提供的示例中，有三个用于软件版本控制的标识字段。这些字段由主要、次要和内部版本号组成。如果主要版本匹配且次要ID恰好比现有代码大1的新固件发送到自举程序，则将执行在线更新。具有不匹配主ID或次ID+1字段的任何映像将编程到非活动分区（序列号也如此），但如果验证成功，则器件将复位为开始执行新固件，而不是尝试分区交换。此行为对于编译在线更新项目非常困难的情况非常有用，它可以强制部署代码，这些代码在用于开发在线更新项目的固件版本之后（也可能是之前）更新一次或多次。

如果主要、次要和内部版本号完全匹配的映像发送到自举程序，则自举程序会拒绝映像而不修改非活动分区。这有助于避免用户不小心尝试对同一个固件进行多次编程导致的不必要编程和器件复位。

在开发过程中，针对现有代码映像和全新代码映像相差一个编译周期的同一个项目执行在线更新情形可能会有所帮助。出于这个原因，内部版本号不匹配但主要版本和次要版本匹配的映像将尝试在不复位的情况下进行在线更新。不过，必须注意哪个软件在软交换事件期间执行以及该软件是否可从自身切换到自身。在例8提供的V2固件示例中，如果关键ISR中的降压重新初始化代码已被注释掉，则此代码将接受以同一代码映像为目标的连续切换事件。未注释掉会导致降压转换器出现不可预知的行为，因为旧的2P2Z控制器代码在RAM中不存在旧的2P2Z状态时进行引用。此外，由于该项目配置为针对V1.elf输出保留变量地址，因此V2中添加的更新变量可以在与之前编译周期相关的链接期间移动。最后，需要考虑SFR，因为它们可能带有历史V1可执行文件中的复位或在线更新无法生成的状态信息。

750W AC/DC示例

为了展示在线固件更新的不同应用示例，Microchip还为750W AC/DC参考设计开发了固件示例。在数控交流/直流电源转换器中，通常有两个单片机，它们通过隔离屏障隔开，一个用于PFC前端，另一个用于直流-直流转换器。通常有一个UART通信桥接器，用于在两个单片机之间发送数据。此系统配置给在线更新应用程序带来了一个小挑战。

主要目标是在不使电源断电的情况下为两个电源转换器执行固件更新。这种配置的问题是主机（即PC）无法直接访问PFC单片机。为了避免这种情况，直流-直流级需要充当主机和PFC控制器之间的信使，这意味着需要增加直流-直流控制器的复杂性。

在本例中，直流-直流控制器充当广播中继器，自动将所有传入主机报文传递给PFC控制器。当主机提供一个新的固件映像时，直流-直流和PFC控制器都会读取标识头，如果任一控制器匹配，则该控制器会响应，而另一个则处于空闲状态（监视状态）。当PFC单片机作为目标设备时，PFC将通过将响应发送回直流-直流控制器，然后将其转发到主机来应答标识匹配。直流-直流控制器将继续广播传入的字节，但会通过本地处理丢弃它们，直到观察到文件结束（End-of-File，EOF）或通信静默时间间隔延长。一旦传输结束，内部EZBL状态机将被复位，然后需要通过自举程序唤醒字符串来再次启动自举过程。即使数据是用于其他控制器，监视字节也允许同步，并且能够消除非目标控制器对广播映像中二进制内容任何部分起作用的可能性。

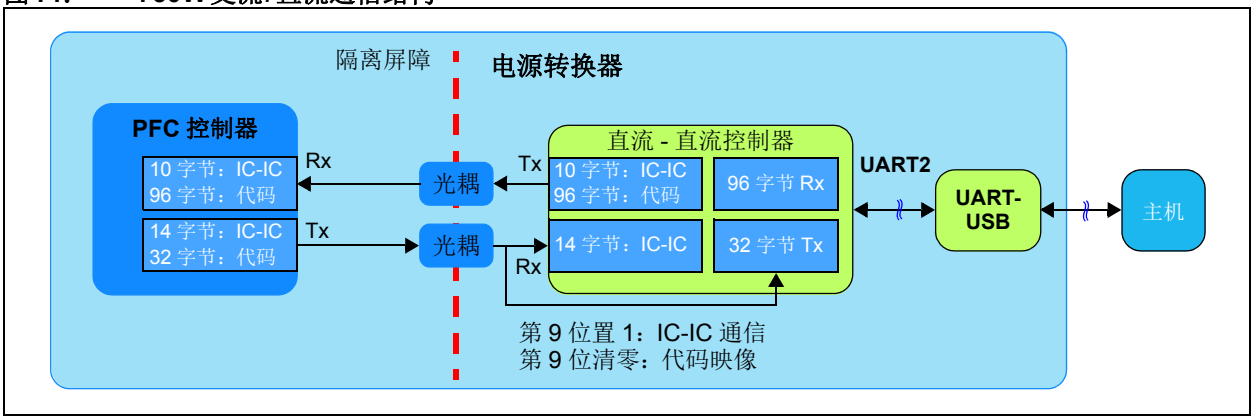
可以对软件进行分区，以使直流-直流控制器能够将PFC的标识和版本记录保存在其固件中。这里，如果直流-直流节点是预期的目标设备，则它将能够决定是否将传入数据传送给PFC控制器。这种方法可以保留通信带宽，但代价是增加直流-直流节点的开销，并且如果将PFC控制器视为可替代模块，则还需要更多维护操作。如果系统设计为可以通过更多连接到总线的节点或需要在PFC节点和主机PC之间传送的其他类型数据进行扩展，则会增加挑战。由于上述问题，在Microchip的750W交流/直流参考设计中，两个转换器都使用相同的自举程序固件，并且直流-直流控制器始终通过隔离屏障将主机数据重新广播到PFC。如果PFC生成出站数据，则直流-直流将把这些消息转发回主机。

通过隔离屏障获取数据面临两个挑战。第一个挑战是能够区分电源转换器之间定期出现的数据以及在线更新的全新代码映像的数据。在本例中，PFC和直流-直流UART配置为9位数据模式，第9位设置为表示板-本地或IC-IC数据字节。当字节源自板外主机PC或发送到板外主机PC时，第9位将清零。这创建了两个虚拟通信域（共用相同的通信硬件）。将直流-直流节点的UART连接到主机可实现传统的八位数据位格式，从而能够继续正常应对外界，并避免浪费第9位的通信带宽（始终为0）。IC-IC通信优先，它能够通过隔离屏障暂时中止任何代码映像传输。但是，由于每个数据字节都标记有目标域位，因此该协议可以轻松适应以每字节为基础的通信管道的乒乓式均衡共享。由于9位数据模式可能不适用于所有应用，因此可以开发用于区分两条路径的不同方法。一种选择是在发送帧之前将相邻字节合并到一个帧中并附加一个报头来指示目标域和帧的字节长度。尽管此方法经调整可支持几乎所有通信硬件，但这会对定期传输操作的报文延迟和抖动产生不利影响。在实现复杂度和所用带宽的过程中，可能会带来很高的开销，特别是在IC-IC报文短小而频繁时。

另一种选择是通过带外信号在协议域之间切换。例如，发送节点可以发送中止（Break）信号来有意地触发接收器上的帧错误，随后发送指示全新目标通信域的1字节指示符。这种方法几乎不会带来带宽开销，因为在线更新通信非常少见，两个节点通常都会保持IC-IC模式。与主机通信时，传回的自举程序数据包含状态或可用的缓冲区大小，以实现软件流量控制。这些信息将向主机告知接收数据缓冲区中有多少可用空间，并强制主机在自举程序忙于擦除或编程现有数据且没有空间可以排入更多数据时进行等待。因此，9位信号传输所损失的带宽和IC-IC通信优化引起的短暂延时无关紧要。

第二个挑战是处理数据速率之间的不匹配。在本例中，与主机之间的通信媒介是UART，但是也可以很容易地通过I^2C进行通信。主机和直流-直流转换器之间的通信链路支持自动波特率，并且可以保持每秒460 KB（460800波特）的速率，这是参考板上MCP2221A USB转UART转换器允许的最大值。两个单片机之间的通信通常采用固定波特率，吞吐量较低（115200波特；受隔离电路性能的限制）。为了处理通信速率的差异，保存要发送到PFC单片机的主机数据的直流-直流缓冲区需要与PFC中的接收数据缓冲区一样大。这可以确保当IC-IC通信长时间完全阻塞隔离屏障时，PFC从主机请求的数据不会超过直流-直流缓存区空间。尽管意义不大，但仍建议将面向主机的直流-直流发送缓冲区分配为与PFC的发送缓冲区一样大。这将确保当主机节点选择的波特率明显低于通过隔离屏障的通信速率时，传输到主机的PFC状态和流控制信号不会被破坏。有关实现的详细信息，请参见图14。

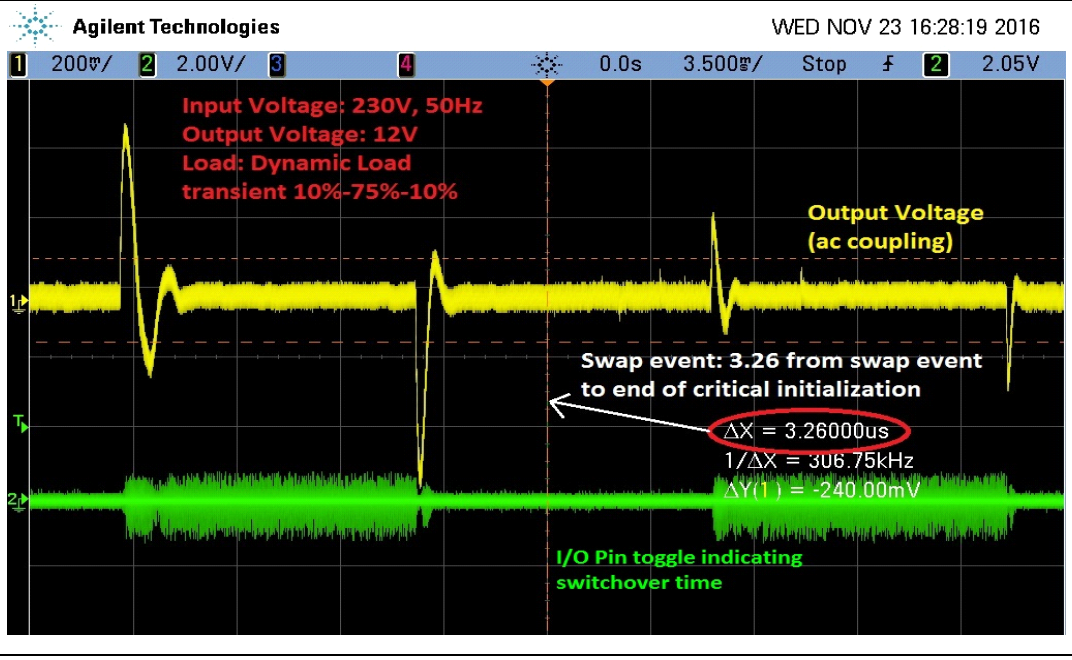
图14：　750W交流/直流通信结构



该实现将共享主机发送缓冲区，因为一次只应有一个转换器向其写入数据。

有多个与此参考设计相关的在线更新示例。在直流-直流端，主要示例通过增加控制器带宽来修改控制器系数的I/Q格式，以获得更好的环路增益响应。这可以在查看负载瞬态响应时观察到。图15捕捉的是切换瞬间和新控制器的瞬态行为。

图15：　750W直流-直流转换器瞬态响应改进

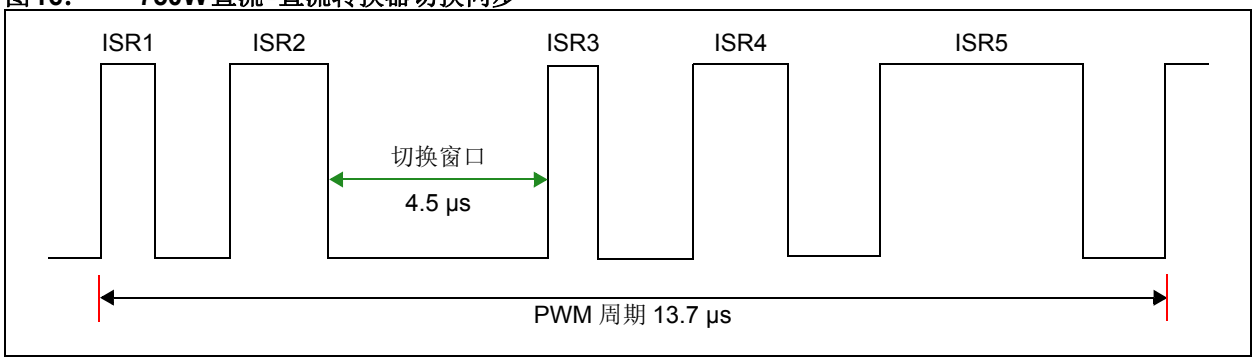


这是与DPSK类似的示例，但并未引入新的控制器类型，也并非必须初始化该控制器，而是开发了额外的固件来确保控制器修改时的稳态误差较小。随后将加载新的系数并初始化新的软件标志，这将针对新的控制器系数适当地调整累积的控制历史。控制器更新紧接在关键时序路径后进行。

设置适当的切换时序窗口给DPSK示例带来了更多挑战，因为有5个关键中断事件用来执行控制系统（见图16）。

一个中断执行电压补偿器（ISR5），两个ISR通过单个电流检测输入（ISR1/3）控制故障输入源以实现适当的峰值电流模式控制，另外两个事件是斜率补偿计算（ISR2/4）。从图16中可以看出，切换事件与ISR2完成同步。

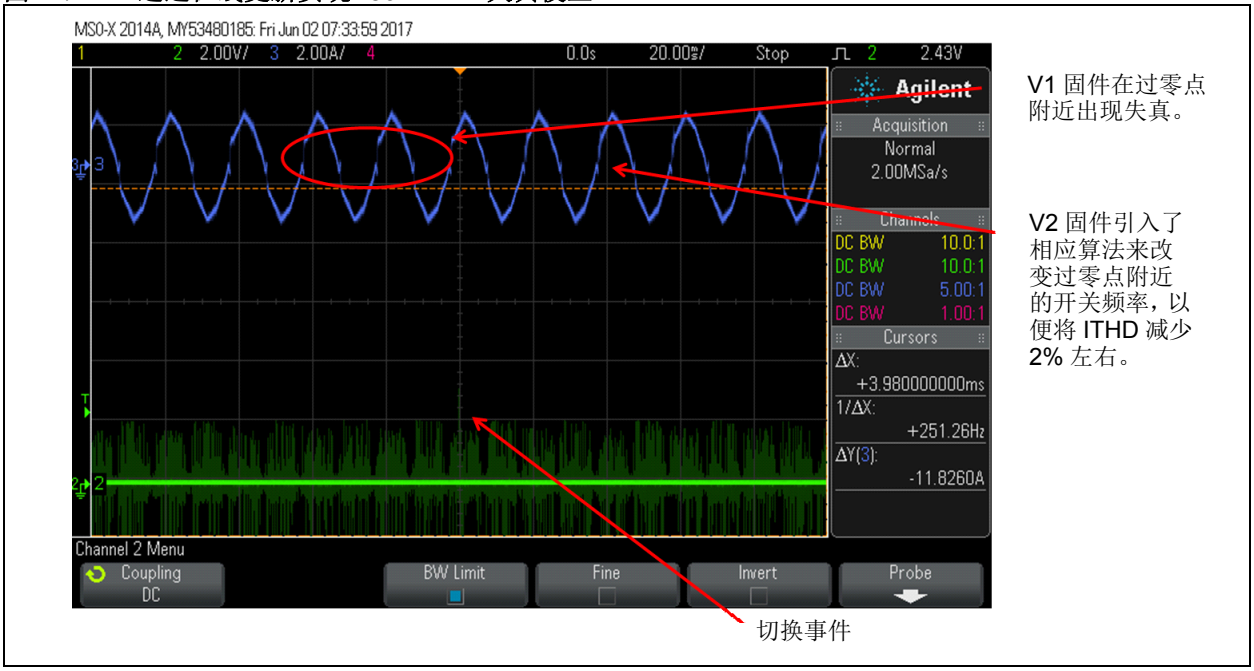
图16： 750W直流-直流转换器切换同步



由于ISR2和ISR4是同一个中断程序（调用两次），因此需要知道之前发生了哪个ISR事件，以便与没有调用电压补偿器的窗口对齐。此设计中实现了4.5 µs的最大切换窗口。非应用特定的切换时序要去掉1.9 µs，而我们的关键路径时序为2.6 µs，虽然这很短，但对于许多不同的更新情况仍然有用。有关直流-直流转换器设计的更多详细信息，请参见AN2388《带数字斜率补偿的峰值电流控制ZVS全桥转换器》（DS00002388A_CN）应用笔记。

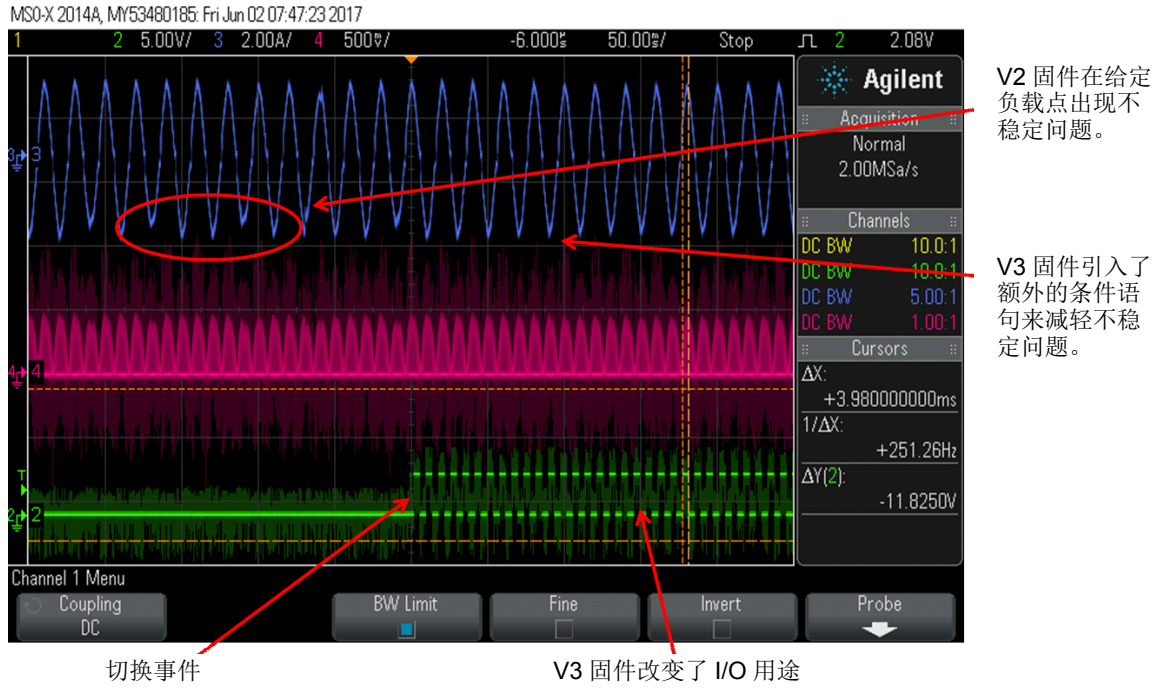
对于PFC转换器，创建了两个不同的在线更新示例以展示在线固件更新的灵活性。第一个示例介绍了一种以50 kHz执行以改善电源转换器ITHD的新算法。这种新算法通过降低交流过零事件附近的开关频率来改善ITHD。此更新需要将附加位域添加到现有的位域结构体中。这是一个相当简单的更新，因为没有任何时序关键型对象，并且不需要变量重映射。固件确实需要指向闪存的指针才能重新初始化，但除此之外主要更改了可执行数据，同时保留RAM的所有内容。图17捕捉切换事件前后的输入交流电流。切换事件之后的交流周期在过零点附近的失真减少。

图17：　通过在线更新实现750W PFC失真校正



第二个在线固件更新与涉及向结构体中添加新功能和单个成员的第一个示例类似。在本示例中，可以从给定的负载点上观察到，由于电流参考计算的自适应算法，系统会发生振荡。此外还开发了额外的固件，要求在过零事件附近使能/禁止算法。添加额外的条件语句修正了不稳定问题，如图18所示。在切换事件之前，可见输入电流幅值会在周期之间振荡。切换事件后，此问题不再出现。同样在此示例中，新固件中切换事件的I/O指示被改变用途，并以不同的速率翻转以指示新的固件正在执行。

图18： 通过在线更新实现750W PFC不稳定性修正

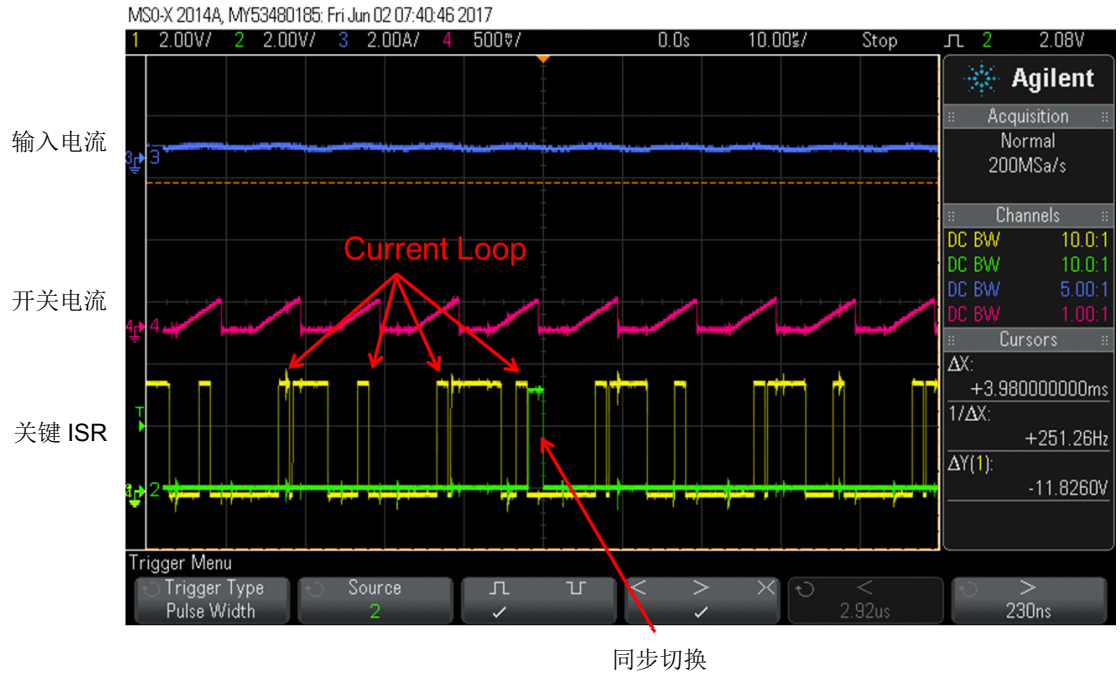


PFC转换器在确定切换事件时面临着挑战。此应用程序中有电流环（以100 kHz最高执行速度运行的环路）、关键输入信号调理函数，几个以50 kHz执行的自适应算法和许多以12.5 kHz执行的函数，包括电压补偿器。在电流中断事件之间，执行很多函数。这意味着切换事件需要同步到电流环ISR的结尾，但在两次中断之间的窗口中，交替执行其他priority函数。

图19给出了在PFC应用程序中运行的关键ISR的屏幕截图。

从图中可以看出，切换事件发生在正确的周期，即恰好在电流环之后。这将最大程度延长PFC的关键时序路径（本例中将接近8 µs）。此示波器图是在第一个在线更新事件期间捕捉的。

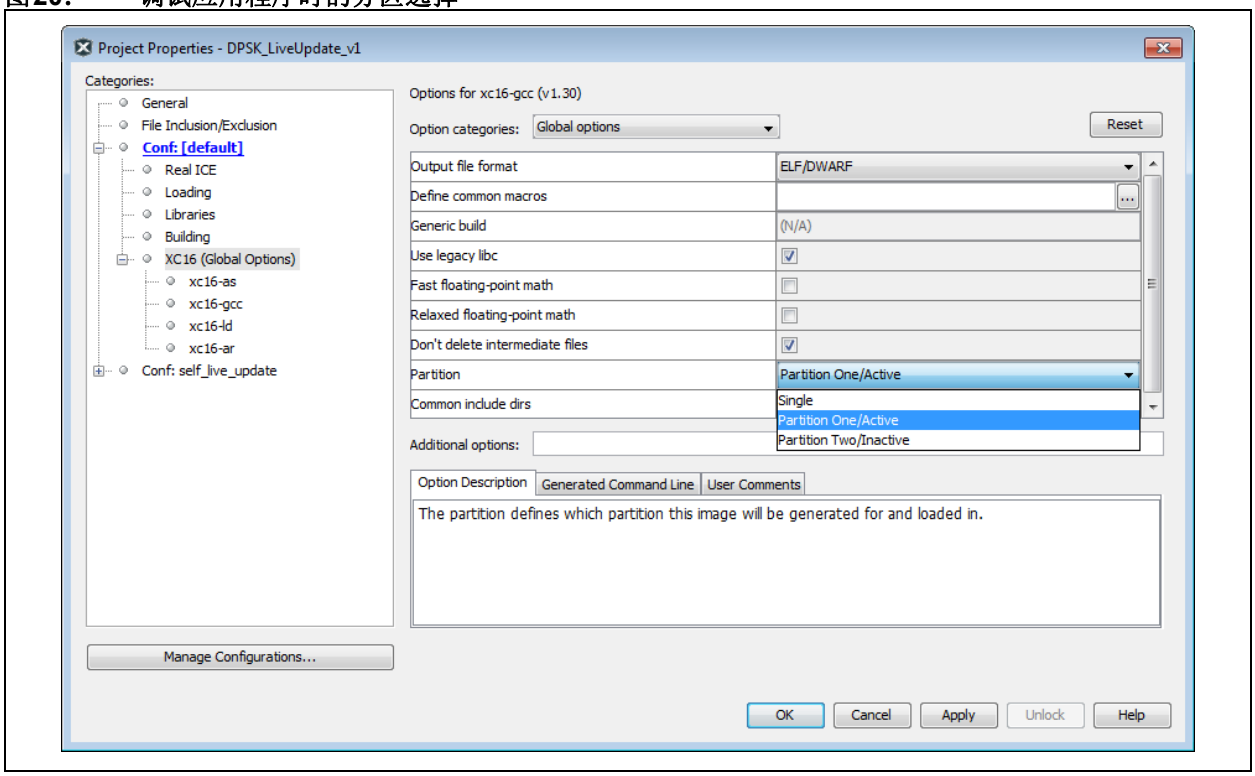
图19： 750W PFC切换时序同步



调试在线更新应用程序

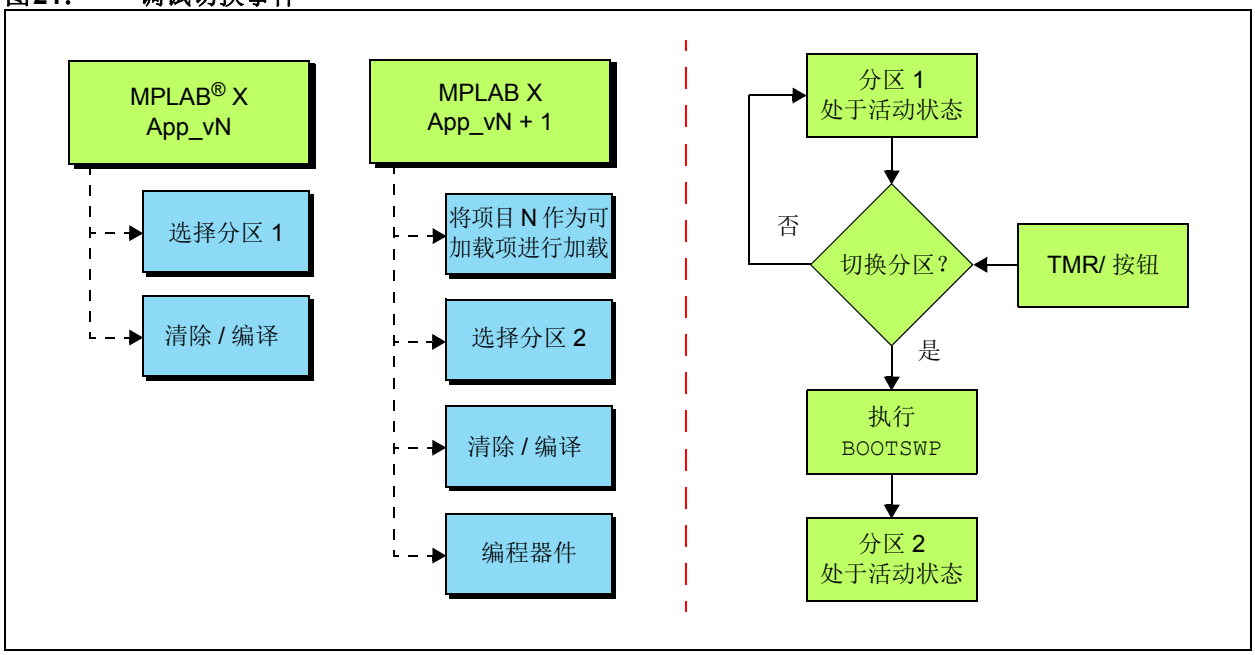
由于MPLAB X允许选择将代码物理编程到的目标分区，因此可以调试在线固件更新。在双分区器件中，活动分区是惟一一个可供CPU执行操作的活动分区，其起始地址始终位于0x000000。非活动分区始终驻留在0x400000，并且可用于非阻塞擦除和编程命令，但只有在交换位置并成为活动分区之后才能用于执行。对于当硬件比较FBTSEQ配置字值时、用来确定哪个分区在器件复位时处于活动状态的上下文之外的软件而言，分区1和分区2的物理概念没有意义。通过在**Project Properties**对话框的XC16全局选项中选择Partition One/Active（分区1/活动）或Partition Two/Inactive（分区2/非活动），代码将被编译和链接，以从地址0x000000开始执行，但最终.hex文件中的数据记录将应用0x000000或0x400000偏移量。有关分区位置选择的信息，请参见图20。

图20：　调试应用程序时的分区选择



为了调试切换事件，需要将前一个项目作为可加载项添加到新的活动项目中，并且需要选择适当的分区。作为本应用笔记中使用的示例，使用默认序列号并始终以分区1为目标；将为分区2配置用于调试的活动项目。这与执行在线更新事件的自然顺序一致。要将上一个项目作为可加载项添加，请右键单击MPLAB X中**Projects**选项卡下的**Loadables**（可加载项）文件夹，然后选择**Add Loadable Project**（添加可加载项目）。浏览到前一个项目并选择**Add**（添加）。确保前一个项目编译时选择了正确的分区（图21）。

图21： 调试切换事件



目前，使用当前的开发工具无法将硬件断点添加到非活动分区。硬件断点将始终映射到活动分区。要在切换事件后停止CPU执行，可以将软件断点添加到新固件中。在新的活动分区中暂停后，可使用硬件断点。在暂停单片机之后，确保某些外设（如PWM）继续执行可能很重要。

由于两个固件映像均正在编程，调试在线更新切换事件不需要自举程序固件。但是，需要开发额外的软件来启动分区交换（例如I/O端口更改（按钮））或使用定时器的超时周期。这可以极大地简化应用程序中测试分区交换事件以及测试关键初始化变量/函数的过程。

技巧与诀窍

交换时间优化

提供了CRT初始化函数和EZBL_PartitionSwap()函数仅供参考。这些函数可基于应用进行修改，以减少非应用特定切换时间。在分区交换函数中，通过清零所有IECx寄存器来禁止所有中断。尽管此程序通用于多个处理器，但仍进行了优化以确保所有中断都被禁止。不过，禁止所有中断的做法可能多余。在启动切换之前，所有低优先级中断均已提前禁止，以创建切换窗口。目前只需要禁止关键中断。将此程序简化为仅禁止关键中断可以将切换时间缩短约200 ns。全局中断允许位GIE（INTCON2<15>）可用于禁止关键ISR并进一步缩短切换时间。只需确保在允许中断之前执行与关键ISR相关的所有关键初始化代码。

与本应用笔记中讨论的代码示例类似，CRT函数也可以重写，以帮助缩短切换时间。其中有多个分支条件（可以用位测试替换）和跳过类型指令（可以在关键路径中节省几个指令周期）。

编译器运行时启动函数可以在以下路径的编译器目录中找到：

<xc16 安装目录>\src\libpic30.zip

由于通过解码位于闪存中的打包记录表来实现CRT数据初始化，因此使用CRT初始化priority变量可能需要很多指令周期。因此，关键时序路径中的变量应具有Persistent属性，并使用放置在明确赋值的priority函数中的代码进行初始化，而不是由CRT初始化。尽管在指令流中编码的文字和派生常量会消耗更多的闪存空间，但这将在关键路径中节省大量时间，并允许这些变量针对冷启动和软交换情况进行适当的初始化。请参见例8中的优先级初始化。

保留地址；不保留正确性

值得注意的是，有几个变量声明实例中的工具链能够满足变量的保留地址要求，但这会导致运行时行为不正确且不发出编译警告。例如，当结构体的成员顺序发生变化，甚至结构体和/或数组大小缩小时，不向用户提供任何编译指示。尽管这是可以预料的，但在在线更新情形中，如果处理不当，可能会产生问题。假设这样一个示例：保留结构体的成员顺序发生变化。链接器将结构视为一个整体，Preserved属性将使结构体的基址保持不变。但是，由于结构体中包含成员变量，每个成员变量都具有相对于结构体基址按顺序增加的地址偏移量，因此对内部成员进行重新排序或对成员的类型宽度进行更改将导致使用错误的相对偏移量（进而使用错误的系统RAM地址）访问所有后续成员变量。为了保留具有这些内部变化的结构，初始化函数需要交换受影响成员的RAM值，并密切关注由于新成员排序而可能添加或删除的隐藏对齐填充字节的影响。另请注意，使用已存储指针或内部结构成员偏移量的任意函数也需要在新软件版本中进行更新。如果结构的更改很复杂，则一种较为可靠的方法是将所需结构体实现为新变量，同时保留旧变量并编写一个初始化程序以将保留的数据从原始结构体打包到新结构体中。

修改后的数组会产生的问题基本相同。链接器只能保留数组的基址，并会在数组增大到与另一些保留变量或绝对地址变量发生重叠时发出警告。基本数据类型、可索引几何数据及属性的变化或元素减少会对数据的存储或访问方式造成重大影响，因此可能需要特殊的切换程序。

对于保留的结构体或数组长度发生变化（缩短）的两种情况，在使能-fdata-sections后，链接器可以重用这些新开放的RAM存储单元。这意味着新添加（Update属性）的变量可能会被CRT初始化，并且会破坏保留变量中的现有尾部数据。如果保留的尾部数据要保存并在一个单独的专用变量中重用，则该尾部数据需要在被CRT覆盖前通过priority函数复制。此外，如果用指针间接访问，用户有责任确保新的固件版本正确更改了任何索引功能。如果不进行相应处理，可能会读取不正确的数据和/或产生地址错误陷阱。

附加调试技术

有时，尽管尽最大努力保持变量地址并定义切换程序以在交换分区后保持一致的工作状态，但在在线更新后不久可能会遇到陷阱异常或器件自发复位。几乎可以肯定，这些问题是由需要特殊处理的运行时数据造成的，但是会在使用属性修饰变量和设置初始化程序时消失。根据应用程序的不同，再现并调试问题可能会面临严峻的挑战。

为了在这些情况下提供帮助，为快速UART调试控制台提供硬件配置非常重要，电气隔离的硬件是首选，这样您便可以传输全面的RAM、SFR和闪存转储，以便在文本文件中轻松进行分析。通过在交换分区之前保存一个日志并在发生故障后尽早保存另一个相同结构的日志，可以比较这两个文件以及在意外位置查找意外数据。

可以使用几个EZBL API来帮助生成此类日志。值得关注的是位于ezbl_lib.a（位于ezbl_lib\weak_defaults\EZBL_TrapHandler.c的EZBL发布版中的源代码）中的EZBL_TrapHandler()函数。该函数实现了一个通用陷阱异常处理程序，该处理程序可输出调试通常需要的一些SFR、RAM内容和闪存内容，并且经简单修改后可输出所有器件状态的综合报告。该数据以人们可阅读的文本形式显示，因此PC上的任何串行控制台应用程序都可用于查看和/或保存该数据。

要尝试该陷阱处理程序：

1. 确保ezbl_lib.a已添加到项目的库中。
2. #include “ezbl.h”
3. 在任何.c文件的文件级范围中，插入：
   EZBL_KeepSYM(EZBL_TrapHandler);
4. 注释掉项目中的任何其他陷阱处理函数，例如 _DefaultInterrupt() 和 _AddressErrorTrap()。只有在未针对特定陷阱定义项目函数时，才会调用EZBL_TrapHandler()。
5. 确保将系统时钟、NOW时序API和UART TX引脚配置为在任何陷阱之前传输调试输出。配置UART所需的代码可能采取以下形式：

例11：

```
NOW_Reset(TMR1, 7370000/2);            // 配置定时器1的NOW时序API，假定系统时钟
                                          为FRC/2 Hz
IOCON2bits.PENH = 0;                   // 禁止所需U2TX引脚上的PWM2H功能
_RP45R = _RPOUT_U2TX;                  // 将U2TX功能分配到RP45引脚
UART_Reset(2, NOW_Fcy, 230400, 1);     // 初始化UART2 @ 230400波特并设置为标准输出报文的目标
```

6. 可以通过调用处理程序或创建一个除以0的数学错误的方式来测试处理程序：

   ```
   EZBL_CallISR(EZBL_TrapHandler);
   ```

   或

   ```
   volatile int i = i/0;
   ```

注：

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了 《数字器件千年版权法案 （Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其他受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

Microchip 位于美国亚利桑那州 Chandler 和 Tempe 与位于俄勒冈州 Gresham的全球总部、设计和晶圆生产厂及位于美国加利福尼亚州和印度的设计中心均通过了 ISO/TS-16949:2009 认证。Microchip 的 PIC® MCU 与 dsPIC® DSC、KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品严格遵守公司的质量体系流程。此外，Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949═

ISBN：978-1-5224-3292-0

全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://www.microchip.com/support
网址：www.microchip.com

亚特兰大 Atlanta
Duluth, GA
Tel: 1-678-957-9614
Fax: 1-678-957-1455

奥斯汀 Austin, TX
Tel: 1-512-257-3370

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Novi, MI
Tel: 1-248-848-4000

休斯敦 Houston, TX
Tel: 1-281-894-5983

印第安纳波利斯 Indianapolis
Noblesville, IN
Tel: 1-317-773-8323
Fax: 1-317-773-5453
Tel: 1-317-536-2380

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608
Tel: 1-951-273-7800

罗利 Raleigh, NC
Tel: 1-919-844-7510

纽约 New York, NY
Tel: 1-631-435-6000

圣何塞 San Jose, CA
Tel: 1-408-735-9110
Tel: 1-408-436-4270

加拿大多伦多 Toronto
Tel: 1-905-695-1980
Fax: 1-905-695-2078

亚太地区

中国 - 北京
Tel: 86-10-8569-7000

中国 - 成都
Tel: 86-28-8665-5511

中国 - 重庆
Tel: 86-23-8980-9588

中国 - 东莞
Tel: 86-769-8702-9880

中国 - 广州
Tel: 86-20-8755-8029

中国 - 杭州
Tel: 86-571-8792-8115

中国 - 南京
Tel: 86-25-8473-2460

中国 - 青岛
Tel: 86-532-8502-7355

中国 - 上海
Tel: 86-21-3326-8000

中国 - 沈阳
Tel: 86-24-2334-2829

中国 - 深圳
Tel: 86-755-8864-2200

中国 - 苏州
Tel: 86-186-6233-1526

中国 - 武汉
Tel: 86-27-5980-5300

中国 - 西安
Tel: 86-29-8833-7252

中国 - 厦门
Tel: 86-592-238-8138

中国 - 香港特别行政区
Tel: 852-2943-5100

中国 - 珠海
Tel: 86-756-321-0040

台湾地区 - 高雄
Tel: 886-7-213-7830

台湾地区 - 台北
Tel: 886-2-2508-8600

台湾地区 - 新竹
Tel: 886-3-577-8366

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733

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

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

印度 India - Pune
Tel: 91-20-4121-0141

日本 Japan - Osaka
Tel: 81-6-6152-7160

日本 Japan - Tokyo
Tel: 81-3-6880-3770

韩国 Korea - Daegu
Tel: 82-53-744-4301

韩国 Korea - Seoul
Tel: 82-2-554-7200

马来西亚 Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

马来西亚 Malaysia - Penang
Tel: 60-4-227-8870

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

新加坡 Singapore
Tel: 65-6334-8870

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

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

欧洲

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

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

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

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

德国 Germany - Garching
Tel: 49-8931-9700

德国 Germany - Haan
Tel: 49-2129-3766400

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

德国 Germany - Karlsruhe
Tel: 49-721-625370

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

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

以色列 Israel - Ra'anana
Tel: 972-9-744-7705

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

意大利 Italy - Padova
Tel: 39-049-7625286

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

挪威 Norway - Trondheim
Tel: 47-7289-7561

波兰 Poland - Warsaw
Tel: 48-22-3325737

罗马尼亚 Romania - Bucharest
Tel: 40-21-407-87-50

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

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

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

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