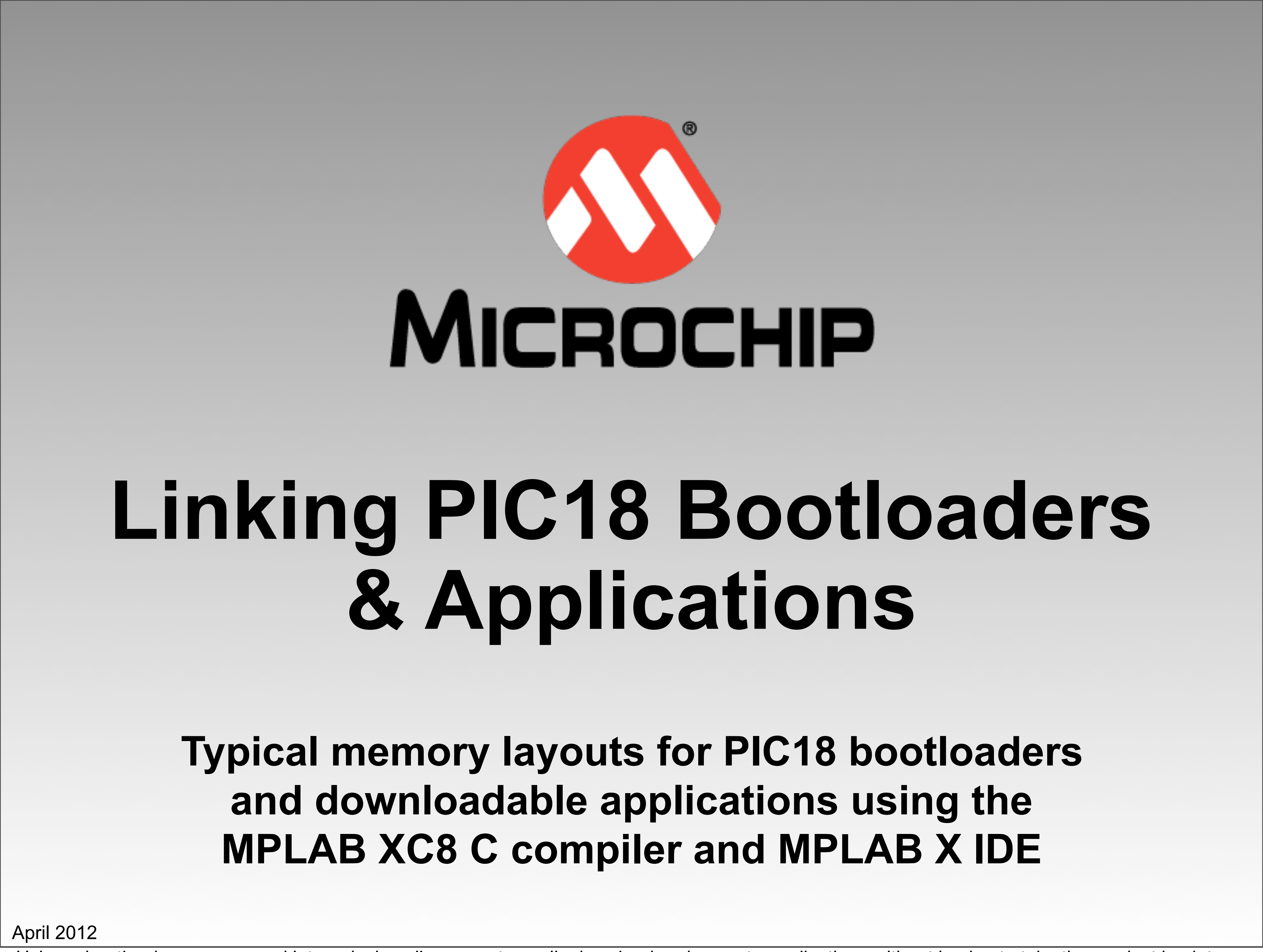


MICROCHIP

Linking PIC18 Bootloaders & Applications

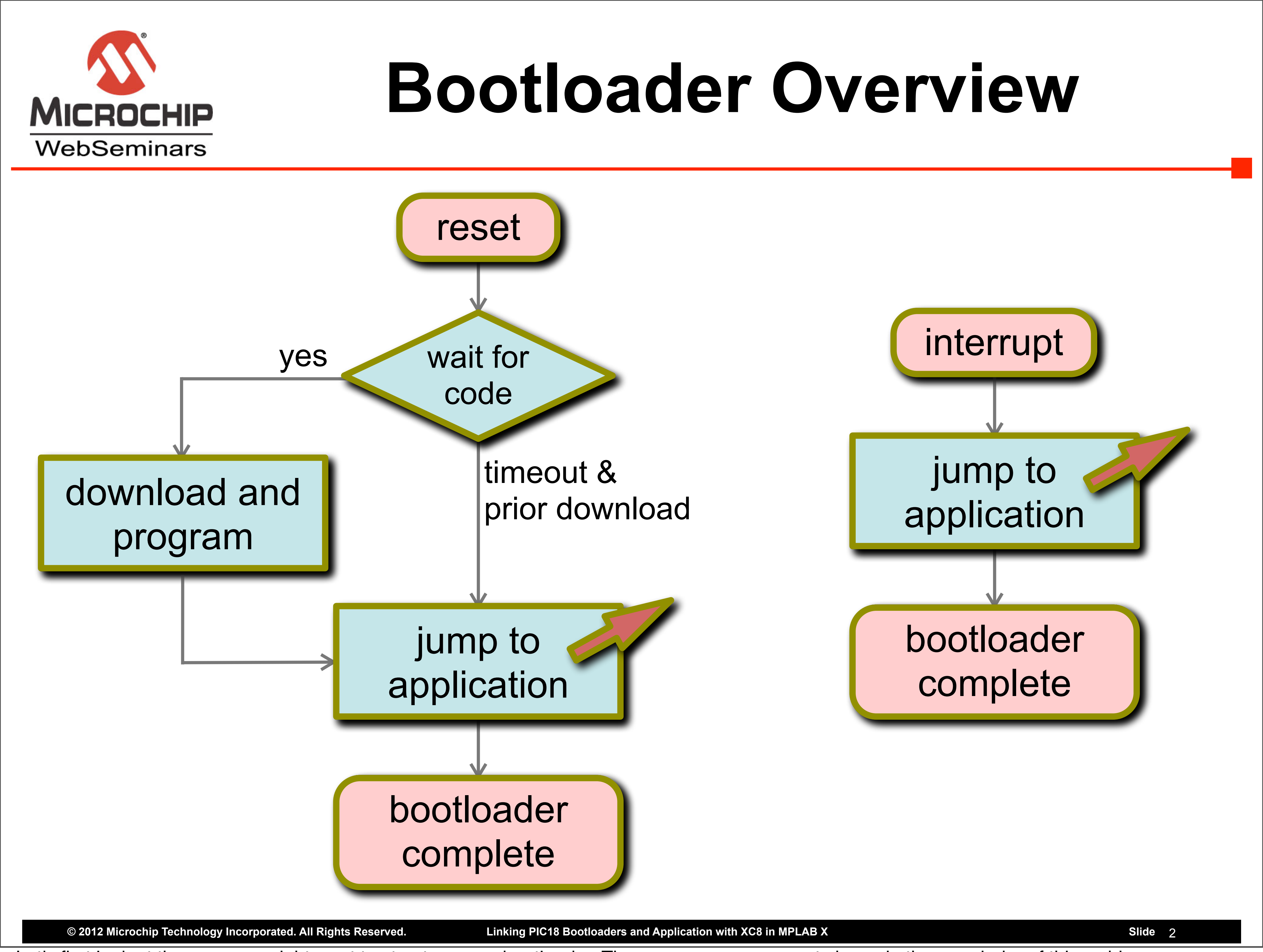
Typical memory layouts for PIC18 bootloaders and downloadable applications using the MPLAB XC8 C compiler and MPLAB X IDE

April 2012

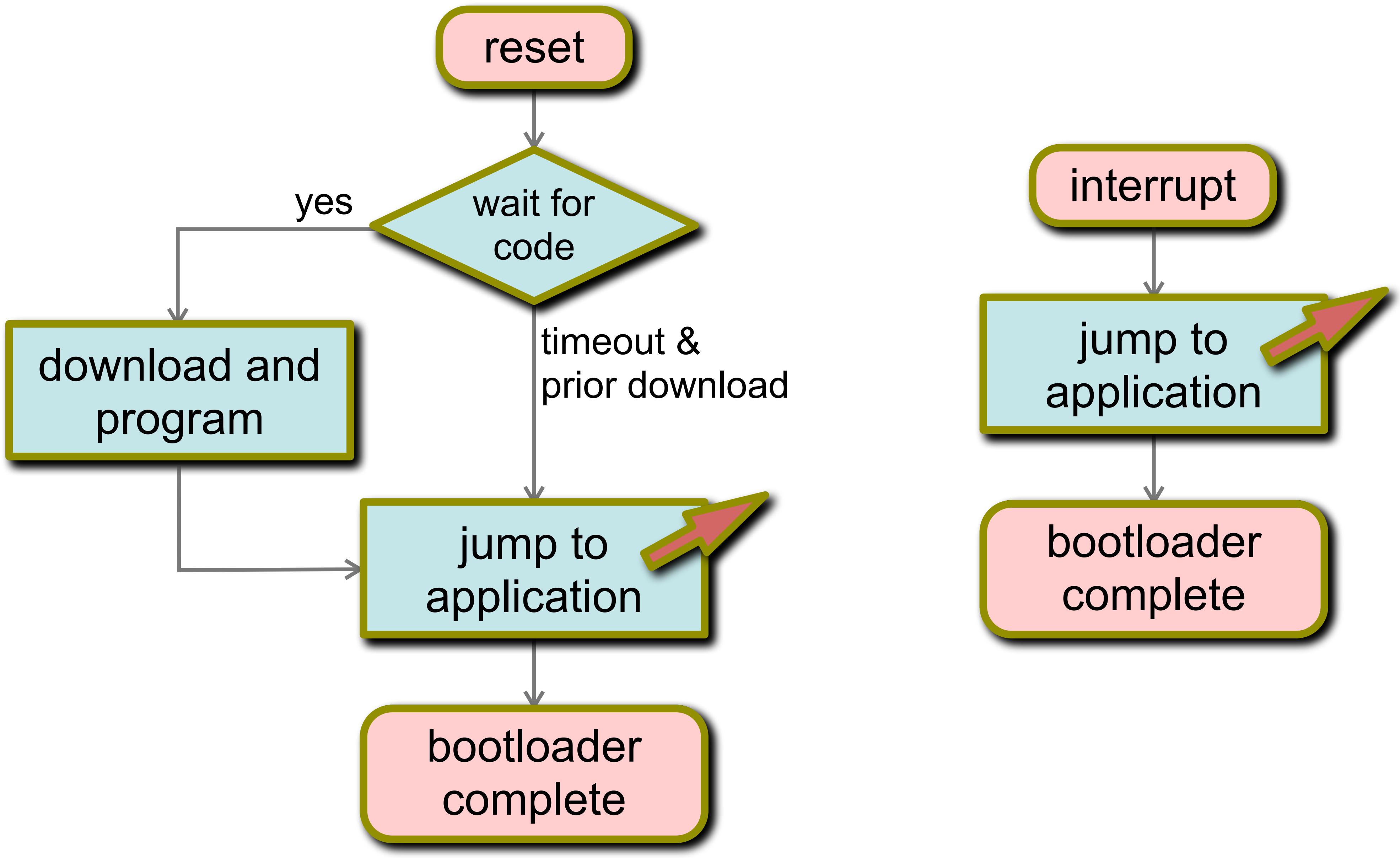
Using a bootloader programmed into a device allows you to easily download and execute applications without having to take the product back to the workshop. But since the bootloader and application are both resident in the one device, special care must be taken to ensure that they share the available memory.

This webinar looks at one strategy you might choose to link bootloaders and the applications they download, the compiler options that make this happen, and how you can check that everything was positioned as required. Exactly what you put into your bootloader and how you write it is left for your discretion; however, we will look at a typical flow diagram for a PIC18 bootloader and also show specific code sequences that the bootloader can use to invoke the application and to handle interrupts.

Although the compiler options detailed in this presentation can be applied to other 8-bit devices, mid-range PIC bootloaders need to be structured differently due to the alternate instruction set, particularly when using interrupts.



Bootloader Overview



Let's first look at the way you might want to structure your bootloader. The memory arrangement shown in the remainder of this webinar assumes the bootloader is designed in this way.

Your bootloader is the only code that is programmed into the device at *all* times, and so *it* has to define the code that will be ★ executed on reset. This code will proceed to ★ wait for the application data to be send on the peripheral of your choice, for example a USART.

Once data starts to arrive, ★ the bootloader reads and programs it into flash memory starting at, what we will call, the *application start address*, which must be predefined and remain fixed. The easiest format to use for the application data is the Intel HEX file that is produced by default when you build with the MPLAB XC8 C compiler.

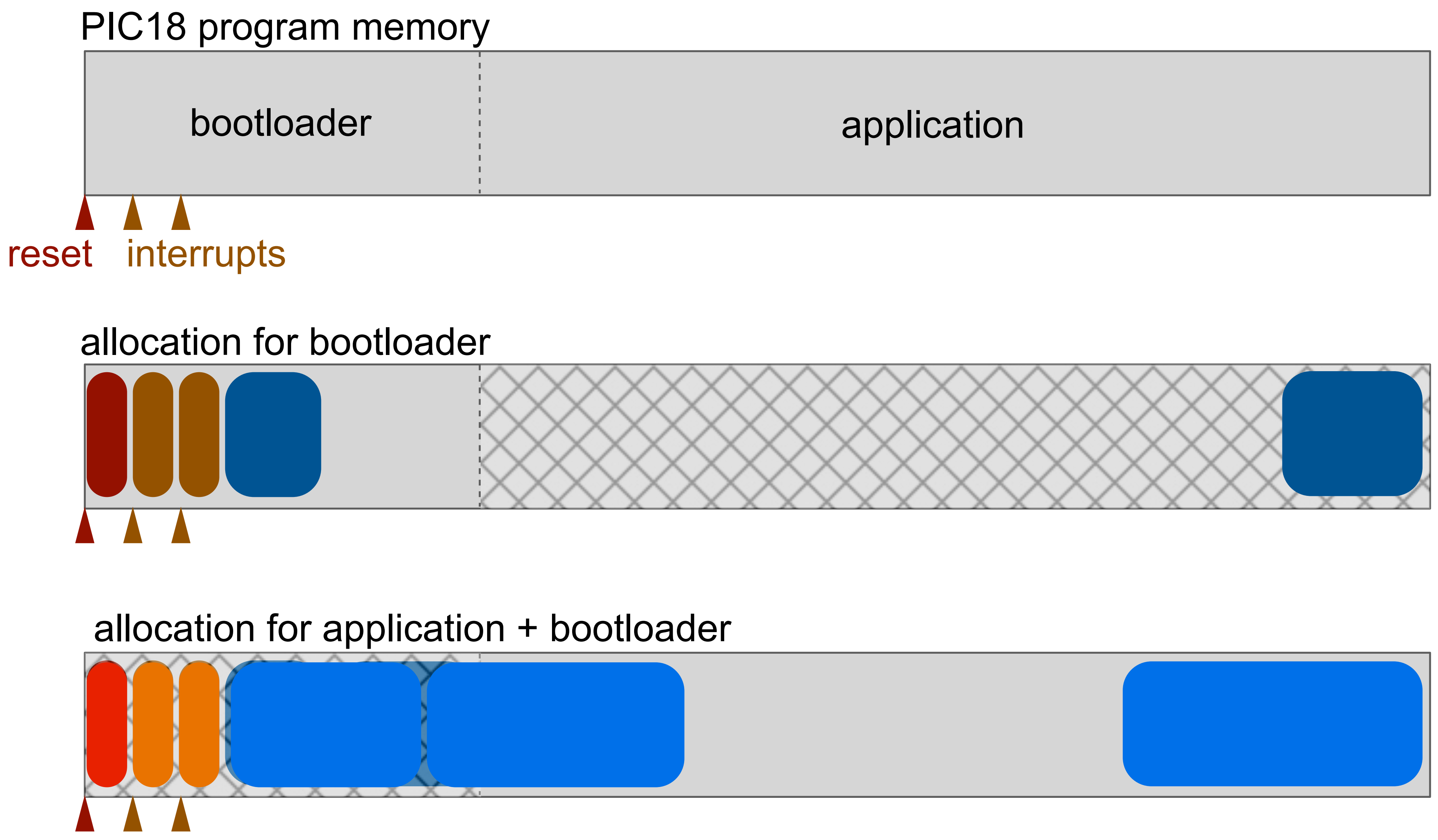
If no data is sent to the bootloader on the peripheral, the bootloader should time-out, and you can, if you want, simply assume that there has been code downloaded previously.

Once the application has been fully downloaded and programmed, or the code which waits for a download has timed out, the bootloader can then ★ jump to the application at the predefined application start address. Since this jump will never return, the bootloader is ★ now essentially complete and the application takes control of the device. ★

If your application needs to use interrupts, this needs to be handled in the bootloader as well. It is easiest to have the bootloader define the code that will reside at the interrupt vector location and ★ when an interrupt occurs, have this code simply ★ jump to a predefined interrupt address in the application. The application will define and execute the return-from-interrupt sequence, so after this jump, ★ the bootloader code is again complete.

We'll talk more about the code which jumps to the application later in this presentation, but for now, let's look at how we can arrange our bootloader and application code in the memory of the PIC18 device.

Program Memory Layout


PIC18 program memory
bootloader
application
reset
interrupts
allocation for bootloader
allocation for application + bootloader

Here we see the program memory for a PIC18 device. Unlike mid-range devices, this memory does not appear to be paged, so this is not a concern in this procedure.

We typically do not need to consider the data memory of the device. Once the bootloader has finished executing, any RAM it was using is now free and can be used by the application, so no sharing of this memory takes place.

basic approach we will use is to conceptually ★ split the program memory, assigning one part to the bootloader and the other to the application. Provided neither uses memory assigned to the other, the code will work correctly once the two program images are merged in the device. Typically, as little memory as possible is allocated to the bootloader.

There are a couple of significant memory locations that you must consider. First, the device reset ★ is hardcode d to address 0. After reset, code will always vector to this address. Second are the interrupt ★ vectors which are located nearby at addresses 8 and 18h. The high and low priority interrupts, respectively, vector to these locations, and again, these addresses are fixed by the device.

So let's look now at how we might link the ★ bootloader application itself. We'll see later code that we can use to deal with interrupts, but let's assume we just write the bootloader in C. Left to its own devices, the compiler will ★ generate code that handles the device reset and have this allocated to the reset vector; our ★ interrupt code will be linked to the interrupt vector locations; but the ★ remainder of the code will be linked anywhere in program memory that is available. It is typically for the linker to start collating some sections up from the start of free memory, and others down from the top of memory. The exact link location of this code is irrelevant, but we need to ensure that it is all contained within an arbitrary memory range.

We can do that by ★ reserving memory using a compiler option. We can remove memory from the default on-chip ranges and ★ force the linker to locate our code into a small region of the program memory space. We'll see the relevant option and how you access it in the IDE shortly, but that's about all there is to linking the bootloader.

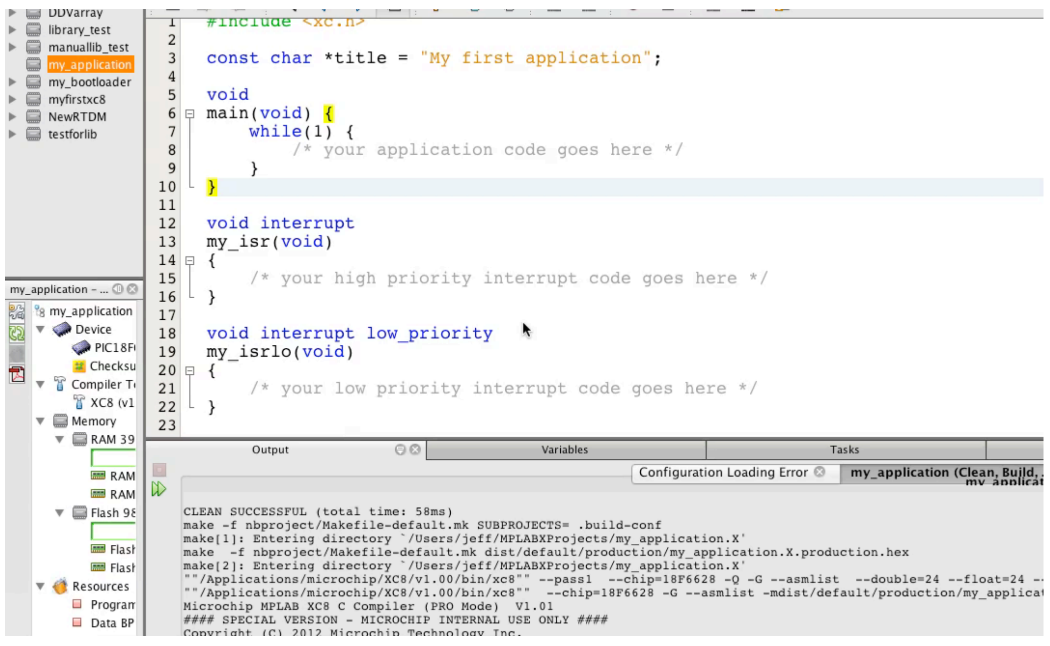
Now to our downloadable application. ★ Again, if we don't use options to indicate otherwise, the compiler will generate and link code at the ★ reset vector, our ★ interrupt functions will be linked to the interrupt vectors, and the ★ remainder of the code will be allocated some other free location. This isn't what we require, so we could try using that same option to reserve the lower memory. However, this would only ★ move those sections of our code that are linked in free space. Our reset and interrupt code will still be linked at the vector locations because the linker has *explicit* instructions for them which must be obeyed.

There is, however, a different compiler option that we can use, one that was specifically designed with bootloaders in mind. What this does is not only reserve memory below the address you specify, it also ★ shifts up both the reset and interrupt code. It's as if the entire program memory image is shifted up by an offset.

Use of these two options ensures that when the bootloader and application are ★ loaded into the device, there is no overwrite or interference.

Linking Application Code

- **Add in `--CODEOFFSET` option**
  - Project properties > XC8 Linker category > Additional Options tab > Codeoffset
- **Recompile**
- **Confirm changes**
  - Use the map file



```
#include <xc.h>

const char *title = "My first application";

void
main(void) {
    while(1) {
        /* your application code goes here */
    }
}

void interrupt
my_isr(void)
{
    /* your high priority interrupt code goes here */
}

void interrupt low_priority
my_isrlo(void)
{
    /* your low priority interrupt code goes here */
}
```

```
CLEAN SUCCESSFUL (total time: 58ms)
make -f nbproject/Makefile-default.mk SUBPROJECTS= .build-conf
make[1]: Entering directory `/Users/jeff/MPLABXProjects/my_application.X'
make  -f nbproject/Makefile-default.mk dist/default/production/my_application.X.production.hex
make[2]: Entering directory `/Users/jeff/MPLABXProjects/my_application.X'
""/Applications/microchip/XC8/v1.00/bin/xc8"" --pass1  --chip=18F6628 -Q -G --asmlist  --double=24 --float=24 -
""/Applications/microchip/XC8/v1.00/bin/xc8""  --chip=18F6628 -G --asmlist -mdist/default/production/my_applicat
Microchip MPLAB XC8 C Compiler (PRO Mode)  V1.01
#### SPECIAL VERSION - MICROCHIP INTERNAL USE ONLY ####
Copyright (C) 2012 Microchip Technology Inc.
```

Let's now look at how we make the compiler link our application code as we want. I've decided that the bootloader will occupy the address range 0 up to 2FF hex, and our application will start at address 300. So, we need to move the entire application program image up by 300 hex and we'll use the `--codeoffset` option to do this. We'll then confirm that everything worked using the map file. Let's swap into the IDE to see how this is done.

--

I already have my application project open in MPLAB X IDE. I have only empty functions entered, but you can see that I have a `main` function and both interrupt functions defined in the usual way, albeit empty. These functions will, of course, be populated with your application code.

Before we adjust the compiler options, let's take a look at the map file, which indicates where things have been linked. It will show us how things are arranged *before* applying the codeoffset option. First, let's build this code so the map file is up to date. Now, for a regular build, the map file will be located in your project directory, under the `dist` > `default` > `production` directory. It will have a `.map` extension.

At the top of the file is a copy of the linker options that were used to build the project. This might look very cryptic, but there are really only two types of options commonly used to link sections -- or psects, as we call them. The first of these options is the `-p` option. Remember how I indicated that the code associated with the reset and interrupt vectors are *explicitly* linked to these locations? Well, this is the option that does that. Here you can see the `-p` option that links the reset vector section at address 0 (`-preset_vec=0h`). This same option also links the interrupt code at address 8, and the low priority interrupt code at address 18 hex.

The other sort of option is the `-A` option, which just defines a range of addresses, called a linker class. Remember, also, how I indicated that most sections are linked anywhere in a memory range. Well, these `-A` options are how those ranges are defined. Here you can see the option that defines the `CODE` class to be a range of addresses from 0 to 17FFF hex (`-ACODE=0h-17fffh`) which is the program memory for the device I have selected. Remember, different devices have different memory sizes. Almost all executable code is placed in a class called `CODE`, so it is a good one to look for in the map file.

If you scroll down in the map file, you can see where the sections were actually placed. You can see the link address for all the sections allocated in the CODE class are all over the place. The reset and interrupt vectors are where we would expect, but other code has been placed up higher. There is `const` data linked into program memory, as well. We don't need to worry about the classes that follow as they are in RAM.

So currently our application is over the top of our bootloader, which we do not want, so let's open the project properties, select the Linker options category, and then select the Additional options tab. The option we want is called Codeoffset -- click on it for help. We simply give it a hex offset address which will be applied to our entire program. So in this example we need to offset by 300 hex, and then apply the change.

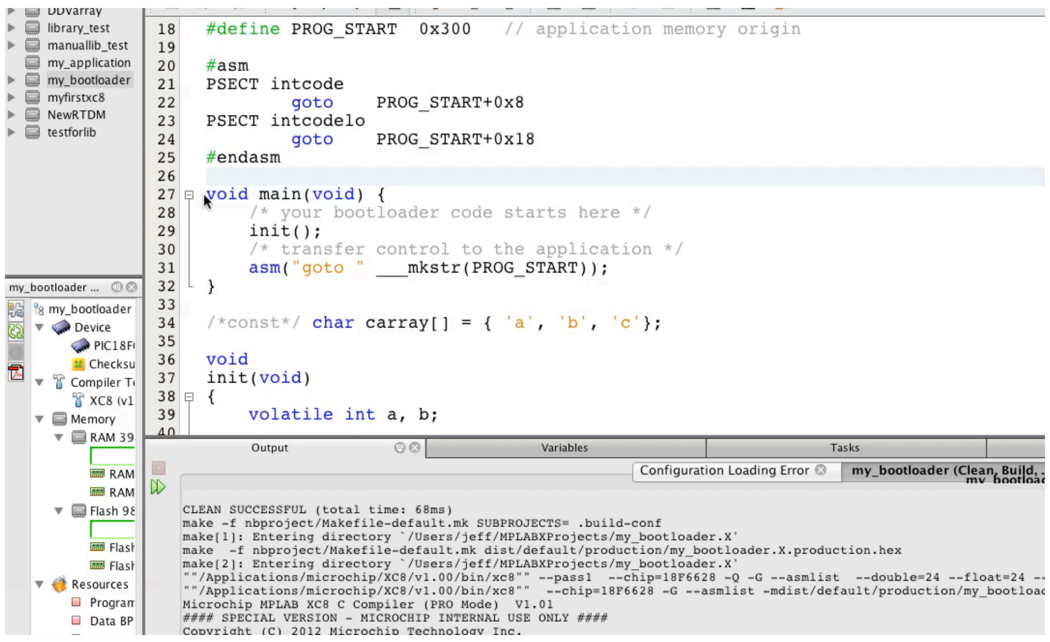
We need to recompile, and that's all we need to do. However, let's confirm that the changes took place -- back to the map file.

You can now see that the reset vector code has been linked at address 300 hex; the interrupt code at 308 and the low priority interrupt code at 318. If you check the linker classes, for example, the `CODE` class, you can see that it is now a memory range that starts at 300 rather than 0. You can also check the actual addresses allocated to sections down lower. You can see that none of the code is lower than address 300. The const data is also linked in the right range.

So there's our application done.

Linking Bootloader Code

- **Reserve memory using `--ROM` option**
  - Project properties > XC8 global options category > ROM ranges
- **Recompile**
- **Confirm changes**
  - Use the map file



```
#define PROG_START  0x300   // application memory origin

#asm
PSECT intcode
        goto    PROG_START+0x8
PSECT intcodelo
        goto    PROG_START+0x18
#endasm

void main(void) {
    /* your bootloader code starts here */
    init();
    /* transfer control to the application */
    asm("goto " ___mkstr(PROG_START));
}

/*const*/ char carray[] = { 'a', 'b', 'c'};

void
init(void)
{
    volatile int a, b;
```

```
CLEAN SUCCESSFUL (total time: 68ms)
make -f nbproject/Makefile-default.mk SUBPROJECTS= .build-conf
make[1]: Entering directory `/Users/jeff/MPLABXProjects/my_bootloader.X'
make  -f nbproject/Makefile-default.mk dist/default/production/my_bootloader.X.production.hex
make[2]: Entering directory `/Users/jeff/MPLABXProjects/my_bootloader.X'
""/Applications/microchip/XC8/v1.00/bin/xc8"" --pass1  --chip=18F6628 -Q -G --asmlist  --double=24 --float=24 --
""/Applications/microchip/XC8/v1.00/bin/xc8""  --chip=18F6628 -G --asmlist -mdist/default/production/my_bootload
Microchip MPLAB XC8 C Compiler (PRO Mode)  V1.01
#### SPECIAL VERSION - MICROCHIP INTERNAL USE ONLY ####
Copyright (C) 2012 Microchip Technology Inc.
```

Now, let's turn to the bootloader and see how we link this in the IDE. In this project, we just need to ensure that everything is linked below a certain address and we'll use the `--ROM` option to do this.

--

So, here is my *bootloader* project open in MPLAB X IDE. Again, exactly what you put in your bootloader is up to you, but we will cover the assembly code snippets you can see in my file later in this presentation.

So recall, our bootloader will not be moved up in memory like our application. The reset and interrupt code will be linked to the usual vector locations. We do, however, have to ensure that nothing in our bootloader extends past the address 300h.

Let's first compile the project, then open the map file. This project is obviously for the same device, so the linker options will initially be the same as they were before. So the reset and interrupt sections are linked just the way we would like. But you can see that the CODE class, for example, still covers the entire range of on-chip program memory. If we scroll down you can see that in this particular case, all the code is below 300 hex, but we should not rely on this being always the case.

Let's open the project properties and make sure only the lower memory is available. Go to the XC8 global options and we want to adjust the ROM ranges option. Click on it for more information. By default, all the on-chip memory is available. With this option, we can either ask for all the default memory and subtract out that part we do not want, but it is easier just to specify the memory that we do want, which would be `0-2ff` in this example.

We need to recompile, and we'll again check the map file, just to make sure the option was accepted. You can now see that the `CODE` class option, for example, reflects the new memory range, which ensures that nothing allocated to this class will be linked over the top of our application. And all the sections are linked as we would expect, as shown lower down in the map file.

Invoke the Application

- **The bootloader lastly runs the application**
  - Jump to the application's start address (reset code)
  - This will be the code offset specified in the options
- **Place the goto in appropriate function**

```
#define <xc.h>
#define PROG_START  0x300   // application offset

void main(void)
{
    ...
    // ready to jump to application
    asm("GOTO " ___mkstr(PROG_START));
```

The bulk of the bootloader can be written in C code, but you will need assembly code to transfer control to the application. After the bootloader has downloaded the application code, it jumps to the application start address. The code located here will be that normally linked to the reset vector if we had not applied the code offset. By jumping to this location, we ensure that the application's runtime startup code is executed and the application will behave as expected. Do not be tempted to jump to the application's `main` function. The address of this function is not fixed, and doing so would mean that any program initialisation will be skipped.

The application start address will be exactly the same as the amount by which you offset the application program image. In the previous slides, we applied an offset of 300h, so this becomes the application start address. In the code example on this slide, I have created a macro, `PROG_START`, to represent the application offset. In the bootloader code, at the point at which you are ready to transfer control to the application, place the single in-line assembly command which inserts a `GOTO` instruction. This instruction must be inserted into a C function, but it does not need to go in the `main` function. Remember than this `GOTO` instruction will never return, so this will be the last code of your bootloader to execute.

You can encode the address, 300, directly into the `GOTO` instruction, but to make things easier to manage, I have used the macro I defined. The 'make string' macro (`___mkstr`) is predefined once you include `xc.h`, and it will turn the `PROG_START` address into a C string which will be concatenated to the `GOTO` instruction mnemonic. Note that there are *three* leading underscores associated with this macro's name.

MICROCHIP WebSeminars

Redirect Interrupts

- **Bootloader code must redirect control to the application interrupts**
  - Jump to the application's interrupt entry points
  - These will be 8 and 18h above the code offset
- **Code must be independent to any function**
  - Use the default psects avoid linker option changes

```
#asm
PSECT intcode
        GOTO    PROG_START+0x8
PSECT intcodelo
        GOTO    PROG_START+0x18
#endasm
```

Now let's look at how interrupts are handled.

The application code defines what the interrupt functions do, but the bootloader code resides over the interrupt vector locations. Thus, the bootloader must redirect control to the application if an interrupt occurs. This also has to be done using assembly code.

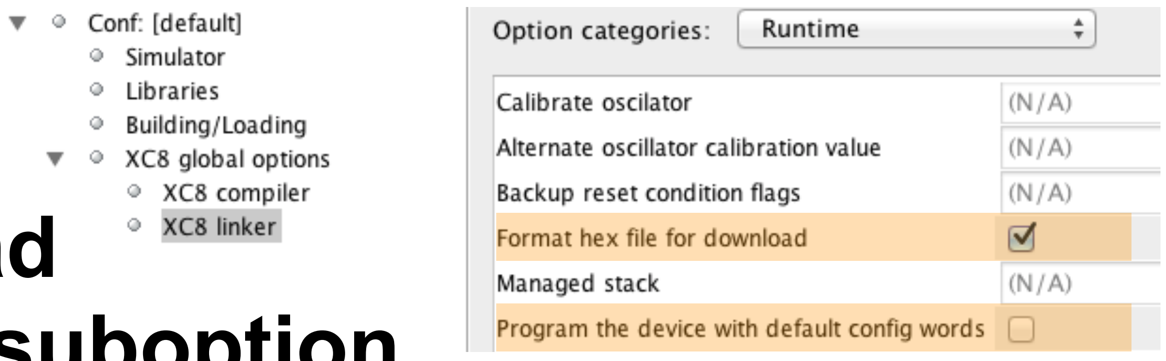
The application interrupt entry points will also be moved up by the same offset that was applied to the reset vector, thus in this example, the two application interrupts will be offset to 308h and 318h. These are the addresses you need to jump to.

Like the code that invoked the application, a `GOTO` instruction can be used here as well, but unlike that code, these `GOTO` instructions should be placed *outside* of the function definitions. In the example box on this slide, I have placed both instructions in the one block of assembly, using the `#asm` - `#endasm` construct, but notice that I have placed each instruction in its own section, or psect. The two psects called `intcode` and `intcodelo`, are the psects in which the interrupt entry points are normally located if you write interrupt functions in C. By using the existing psects I do not need to define new ones, add new linker options or adjust existing options.

When you check the map file for the bootloader, you should see these two psects defined. They should be linked to the normal interrupt vector locations: 8 and 18h.

Helpful Tips

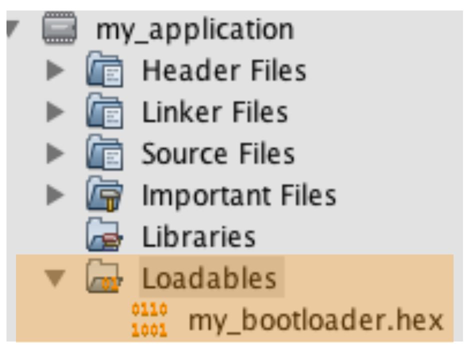
- **Config bits typically set in bootloader**
  - Disable default config bits setting
- **Simplify download using `download` suboption to `--RUNTIME`**
  - Ensures each HEX record is 16 bytes long
- **Bootloader HEX file can be merged with application**
  - Add bootloader HEX file or project to the MPLAB X Loadables folder



Conf: [default]
- Simulator
- Libraries
- Building/Loading
- XC8 global options
  - XC8 compiler
  - XC8 linker

Option categories: Runtime

Calibrate oscilator	(N/A)
Alternate oscillator calibration value	(N/A)
Backup reset condition flags	(N/A)
Format hex file for download	✓
Managed stack	(N/A)
Program the device with default config words	☐



my_application
- Header Files
- Linker Files
- Source Files
- Important Files
- Libraries
- Loadables
  - my_bootloader.hex

We've seen how to link your bootloader and application, but here are some other points you can consider.

As with any project, the device configuration bits need to be set for correct operation. Since the bootloader is resident in the device at all times, it makes sense to have this code define all the configuration bits. If required, your application can modify the configuration bits at runtime, but only if that is allowed by the target device.

The XC8 C compiler can supply default configuration bits if your code does not define them. You should disable this feature when you build the application so there is no chance default settings will conflict with those set in the bootloader. The `--RUNTIME` option controls this feature and in MPLAB X IDE, it is found in the Runtime tab of the XC8 linker category and is called 'Program the device with default config words', as shown in the upper diagram.

Your bootloader should be as small as possible to allow as much room for the applications. To help you achieve this, you may be able to simplify the download algorithm if you assume the application's HEX file has a fixed record length. The `download` suboption to the `--RUNTIME` option can do this. In the IDE, it is also located in the Runtime tab of the XC8 linker category and is called 'Format HEX file for download', again shown in the upper diagram. Use of this option forces each HEX record to be exactly 16 bytes long. The record is padded if it would otherwise be shorter than this length.

Once you have built your bootloader and application projects, you will end up with 2 independent HEX files. During testing, you may want to simply use a programmer to program both files rather than use the bootloader's download code. Your programmer may allow the two files be programmed separately. Make sure your programmer is configured so that it will not clear the program memory before any download. Alternately, you can merge the two files when you compile. When you build your application, include the bootloader HEX file or the bootloader project itself into the application's Loadables folder in MPLAB X IDE. This will build the application and merge in the bootloader code, resulting in a single HEX file output.

--

In this webinar we have looked at how you might structure and link a bootloader and downloadable applications. The prime consideration is that the bootloader and application are allocated their own discrete program memory ranges, which they can use without corrupting each other. Since the reset and interrupt vector locations are fixed by the device, you must consider whether it is the bootloader or application that will define these. In this presentation, the bootloader populated the vector locations, but this code simply redirected control to the application, which was offset higher in memory.

Memory allocation touches on some advanced compiler topics, such as linker options and psects. More assistance on these topics is available in the compiler's user's guide. You will find the map file invaluable when it comes checking memory allocation. The compiler's user's guide also has information describing the content of this file.