Introducing software breakpoints in MPLAB® IDE and comparing them to their hardware cousins

November 2011

Everyone is familiar with breakpoints, which can be used to halt execution of your program in a debugger, but are you familiar with software breakpoints and the differences between them and hardware breakpoints? And do you know the advantages and limitations of each type? If you're currently scratching your head trying to think, then maybe this webinar is for you. In this presentation we'll also look at how software breakpoints are enabled in MPLAB IDE and a few related IDE debug features.
Hardware Breakpoints

● When a hardware breakpoint (BP) is set:
  ● Address is sent to the target device
  ● Stored in an internal debug register

● When the device is executing code:
  ● The stored address is compared with this address
  ● Execution is then halted when equal

● Can be set on data values as well as code

First, let’s look at breakpoints that most people are familiar with: hardware breakpoints. We'll look at the details of hardware breakpoints so the difference between these and software breakpoints will become apparent. Note that many of the issues we will discuss in this webinar apply to execution on a debugger, such as a MPLAB REAL ICE or ICD3, not in the simulator.

When you set a hardware breakpoint, the breakpoint address is sent to the target device and stored in an internal debug register. When the device is running and the program counter compares equal to this stored address, control is transferred to the debug executive and the user’s application is effectively halted. (The debug executive is a small program that MPLAB IDE downloads into the target device when you are debugging code.) As its names implies, circuitry in the target device is detecting the breakpoint address and causing execution to halt.

Hardware breakpoints can also be set to trigger when a file register address is read or written, or trigger when a file register assumes a specified value.
The operation of hardware breakpoints is reliant on the silicon implementation, and this leads to a few limitations.

The first is purely a matter of resources. Since each hardware breakpoint needs an independent address stored in an internal register, there will be a limit on the number of breakpoints that can be set at any given time since there is a limited number of registers. For some PIC devices, this is a maximum of 10, others 6, but most implement 3 registers or less.

The other limitation with hardware breakpoints relates to exactly where it will halt execution. Once a hardware breakpoint has been triggered it may take time to actually halt the device due to the instruction pipeline which must be executed before halting. This process is called 'skidding'. Skidding means that the program may stop one or two instructions after the address where you set the breakpoint. The graphic shows what you might see in MPLAB IDE when a hardware breakpoint is encountered. Notice the program counter, represented by the green arrow, is one instruction ahead of where the breakpoint was set, represented by the red circle.

The exact halt location is dependent on the instruction located at the breakpoint address. If there are flow-control instructions, such as a call or jump, at this address, their execution may result in the device halting at a location no where near the breakpoint address. This can be overcome by inserting one or more NOP instructions into your code or setting the breakpoint prior to desired halt location, but you have to add and remove these instructions by hand and be mindful of this skidding effect at all times.

PIC32 devices do not exhibit skidding and this information does not apply to them. Due to the fact that breakpoints trigger an exception and the pipeline is flushed, the program counter will stop at the desired address when a breakpoint is encountered.
Software Breakpoints

- When set, the instruction at the breakpoint address is replaced by a special instruction.
- The IDE/debugger manages the breakpoint:
  - Replaces the specified instruction when set.
  - Ensures the original instruction is executed correctly when execution is resumed.
  - Replaces the original instruction when the breakpoint is removed.
  - Always shows the original instruction in memory views.

And so we come to introduce software breakpoints.

As their name suggests, the process of halting is performed by an actual assembly instruction. When you set a software breakpoint at an address, the instruction at that address is replaced with a special debugger instruction that will immediately halt the device once it is executed. MPLAB IDE manages software breakpoints for you: it swaps in the special debugger instruction when the breakpoint is set, and it ensures that the original instruction is executed correctly in its original context when you resume. The original instruction is also restored by the IDE when you remove the breakpoint. The IDE always shows the original instructions in any program memory view, even if a software breakpoint is set, so you can always see the instructions relevant to your code.
Software BP Benefits

- No internal registers or circuitry is required
  - An unlimited number of breakpoints can be used concurrently
- Execution of the special halt instruction causes an immediate halt
  - The device does not skid
  - Breakpoint locations can be set quickly and accurately

Software breakpoints are not affected by the limitations experienced by hardware breakpoints; however, as we shall see later, they do have their own caveats.

As no internal registers or circuitry is required to implement software breakpoints, you can have a huge number of these, all active in your project at the same time. And since a special debugger instruction is used to stop program execution, the device does not skid.

Skidding of hardware breakpoints implies that the instruction at the breakpoint address is executed before the device halts, which is not what we would prefer. When a software breakpoint is encountered, the device will still execute the instruction at the breakpoint address, but since this a special debugger instruction that will not affect the device's state -- we will essentially observe the device as it was prior to the breakpoint address being reached. The IDE and debugger take care of backing up the program counter to show the device stopped exactly where you want.

No skidding means that you don't need to think about the code where the breakpoint is to be set. It doesn't matter what instruction is located there, the breakpoint will always stop at the required location.
Let's take a look at device-dependent features of both hardware and software breakpoints.

First, hardware breakpoints are available on all flash-based devices so they are always available to you.

Software breakpoints are available on many devices, but are absent from the Baseline and Mid-range parts, that is the PIC10, 12 and 16 devices, although support is well under way for the enhanced Mid-range devices, that is those with a 16F1xxx part number. All PIC18 devices support software breakpoints, but the 18F K parts will get this support at a future date. All 16- and 32-bit devices have support, except the smaller PIC32MX1 and MX2 devices. Note that software breakpoints are not permitted if you are using the alternate MIPS16 instruction set available on PIC32 devices.

As to the number of breakpoints available, the lower table shows the maximum number of hardware breakpoints for each device family. Some families implement a varying number of hardware breakpoints. For example, PIC18F devices implement either 1, 3 or 5 breakpoints, dependent on the actual device you are using. Although some devices allow for up to 10 hardware breakpoints, the majority of devices support 3 or less.

The number of software breakpoints you can implement is unlimited.
Which should I Use?

- **Use Hardware BPs if:**
  - You are using a production part
  - You need to break on data values
  - Software BPs are not supported on your device

- **Use software BPs if:**
  - You need a large number of breakpoints
  - Skidding is an issue

<table>
<thead>
<tr>
<th>Feature</th>
<th>Hardware BP</th>
<th>Software BP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to set breakpoint</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Impacts device endurance</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can operate on data registers</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

There are benefits to both hardware and software breakpoints.

By far the biggest difference between these is the number of breakpoints that can be used concurrently, as we have just seen. We have also mentioned skidding past the breakpoint. But, other differences have been summarized in the table shown on this slide.

Since setting a software breakpoint requires reprogramming of the device, it can take a longer time to set than it does with hardware breakpoints, which only requires communication with the debugger. If you set and clear breakpoints often, aside from being slow, the frequent reprogramming may impact on the endurance of the device. It is for this reason that Microchip does not recommend using software breakpoints with production devices.

So, if you only need a very small number of breakpoints, and negating the effect of skidding can be easily accomplished, hardware breakpoints are probably well suited to your project. You *must* use hardware breakpoints if you wish to break on a data value, not a code address, and hardware breakpoints are your only option for some devices.

On the other hand, if you need a large number of breakpoints or cannot easily workaround skidding issues, software breakpoints will work best during code development.
If you wish to use software breakpoints, they need to be enabled in MPLAB IDE, as hardware breakpoints are used by default. Assuming your debugger is already selected, open the 'Settings' dialog under the 'Debugger' menu in MPLAB IDE v8. Click the 'Configuration' tab and select 'Use Software Breakpoints'. There is a shortcut to this menu item in the Debug toolbar. In MPLAB X IDE, open the Project Properties, click the Debugger category -- in the diagram a REAL ICE is being used -- and under the 'Debug Options', click 'Use Software Breakpoints'. You can also do this using a button on the far left of the project Dashboard.

Breakpoints are essentially an assembly code feature as they stop execution after completing an assembly instruction, but they can still be set in C code. The IDE can map a breakpoint set on a line of C code to the first assembly instruction that corresponds to that C statement. Setting a breakpoint on a C source line will ultimately set a breakpoint on a single assembly instruction and you will be able to see this breakpoint set at this instruction in the Program Memory view. Be aware that compiler optimizations, particularly assembly optimizations, can make this C-to-assembly mapping less reliable. And make sure that the line of C code at which you plan to break is something that actually generates assembly code.

If the assembler optimizers are affecting debugging or you want to set a breakpoint within the middle of code associated with a C statement, you can always set breakpoints in the Program Memory view even when working with C source code.
Who Stole My Breakpoint?

- The Debugger's step-over/out and run-to-cursor features require one breakpoint
  - If you assign all available breakpoints, these features are disabled
  - Step-into will still operate as normal

- If you are using hardware BPs, skidding will also affect step- and run-to-features
  - Skidding will not occur with step- and run-to-features when software BPs are used

If you still think hardware breakpoints may be the best choice for your project, just consider the following final points.

If you plan to use the step-over, step-out or run-to-cursor features, just be aware that these need a breakpoint to operate. Thus, if your device only supports a small number of hardware breakpoints and you plan to use these step- and run-to-cursor features, then you may have only a few or no breakpoints left to set at arbitrate program locations. The step-into feature, however, does not use a breakpoint, so this may be used at all times.

Skidding will affect the step- or run-to-cursor features, just as it does an ordinary breakpoint which you set yourself. If you do a lot of stepping, particularly stepping of C code, this could be the catalyst to swap to software breakpoints. If code skids with every step, it will make it more difficult to track how your code is behaving. And as these step features are commonly used, you should consider how skidding will impact debugging your code.

Although hardware breakpoints are enabled by default, the software breakpoints that many PIC devices and MPLAB IDE can utilise are actually a very viable alternative to debugging that solve a number of issues that hardware breakpoints present. The next time you run out of breakpoints, or skid past your mark, why not consider making the switch and break softly.