

Tracking Down Microcontroller Buffer Overflows with 0xDEADBEEF

By Stuart Cording (Elektor)

Software development on microcontrollers can be a real challenge at times. Sometimes the code just doesn't function as planned, but you can't really figure out why. In such cases, it is time to approach the challenge like Columbo, Jessica Fletcher, or (insert your super sleuth here), do some investigative research, and find out who the culprit is! And for that, you may need to lay a trap using dead beef (0xDEADBEEF).

Code sometimes stops working as expected for seemingly no rhyme or reason. Perhaps some data changes unexpectedly, the program just seems to "stop," or the microcontroller resets unexpectedly. The first thing to do is to admit that there is most likely a programming problem. Microcontrollers only do what we tell them to do, so we must have, inadvertently, put it in a position to do the unexpected. Such issues may also arise because we do not fully understand how the compiler and linker functions.

Make the error reoccur

The first step in your investigation, if the error is sporadic, should be to determine how to make the error reoccur. Only by invoking the error repeatedly are you able to undertake further tests to see whether you can make it "go away." However, you need to make sure the cause is clear so that you can be sure that whatever you changed actually resolved the error and didn't just shift the problem elsewhere!

It is tempting to add extra "test code" and variables to help in this process. If such a change makes the error "disappear," it could be that a memory problem is the cause of the issue. These can be the result of buffer overflows or an overflow of the stack.

Buffers are typically implemented as arrays on microcontrollers in C and will be defined with a fixed length. Less used on microcontrollers, but also possible, is an array that is allocated at run-time using "alloc" (such as `malloc` or `calloc`). These are known as dynamically allocated arrays. Either way, the C language makes it very easy for programmers to write code that accesses memory beyond the defined bounds of an array. The result: writing to an array could accidentally overwrite neighboring variables.

Laying a trap with dead beef

One way of checking whether arrays are being written beyond their bounds is to fill them with a known pattern before they are used. Because hexadecimal includes the letters A to F, simple words can be formed that provide easy to recognize patterns. One of these is 0xDEADBEEF.

One way to do this is through the IDE and debugger being used to develop the code for your microcontroller (**Figure 1**). The first step is to determine the start and end address of the array. Then, using a view of the memory (RAM), it is usually possible to write such a pattern from the start address to the end address.

The best approach to this debugging method is to download the code to the microcontroller and allow the code to run until `main()` has been reached (set a breakpoint at `main()` or use a "run to cursor" feature). Determine the start/end address of your array, write the pattern, and then allow the code to execute for a while. Pause code execution and then check the memory location you filled with your 0xDEADBEEF pattern. If none of the pattern can be found, this may be because data has been written beyond the end of the array at some point. All that is left is to determine when and why.

Note: Your development environment (IDE) may not allow the full 0xDEADBEEF pattern to be written. In the examples included here using MPLAB X, only 16-bit patterns can be written into the memory of the PIC24F (as it is a 16-bit microcontroller). Here we chose to just use 0xBEEF, which is also easily recognizable. Alternatively, you may be able to write your pattern into an ASCII file, save it to your hard drive, and then load it into memory.

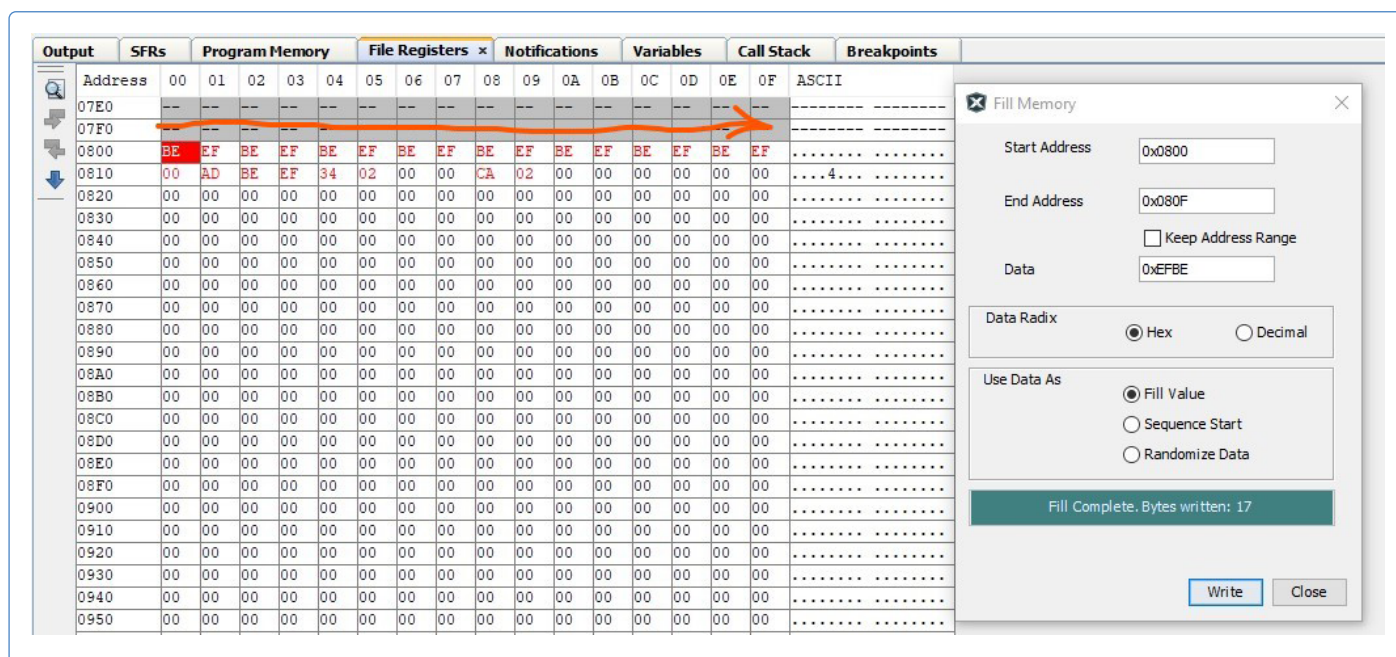


Figure 1: The development environment (here MPLAB X IDE) can be used to fill a section of RAM with a pattern.

Did my memory change?

Another helpful feature of most debuggers in such investigations is that they highlight memory locations that have changed in a specific color, such as red, when the microcontroller is halted. If you halt code execution with the memory region you are examining in view, any bytes that have changed in value in the meantime will be shown in this different color (Figure 2). This can be especially helpful if any third-party functions or libraries were called that may have overrun a buffer.

If your 0xDEADBEEF pattern is completely gone, it will be difficult to know how far the erroneous code has written beyond the bounds of the array. In such cases, try defining additional arrays as a "buffer" before and after the array being examined (Figure 3). This is a little tricky in C/C++ as, without some extra code, you cannot define the address to which variables are allocated. However, generally speaking, variables defined after one another in code appear after one another in memory too. This means that if you declare some "extra buffer"

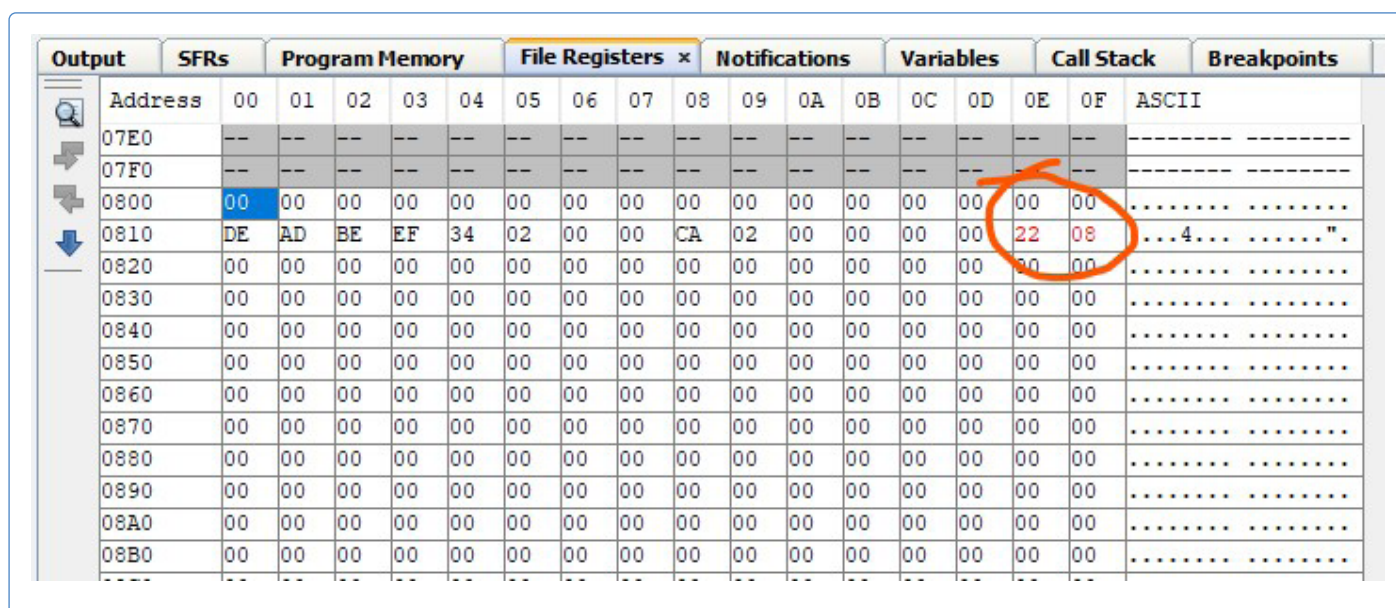


Figure 2: Since the MCU was last halted the memory location circled in orange has changed, as its contents are highlighted in red. Black values denote no change. If you weren't expecting these values to change, you may be on your way to finding your software bug.

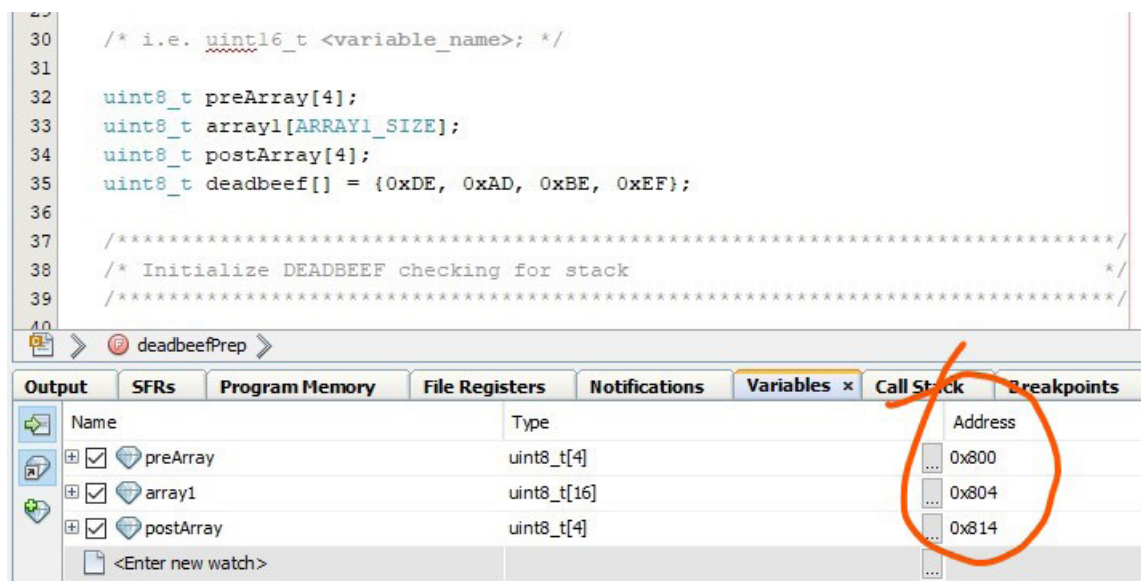


Figure 3: Two extra arrays defined to help debugging are, luckily, located at addresses before and after the array we are investigating. The linker can also be forced to use specific addresses if this is not the case.

arrays, one before and one after the array you are investigating, you'll probably find they are allocated directly before and after that array in memory. Simply fill all arrays with `0xDEADBEEF`, execute the code, then check to see if the pattern in all array is intact.

Debugging real-time code

Many real-time applications, such as those controlling motors, or implementing interfaces like USB or Ethernet, cannot be stopped and debugged line by line as this would stop them working completely. This means that we cannot stop the code to write the `0xDEADBEEF` pattern into memory using the IDE. Instead we need to do this in software. This is simple to implement using a short array that is pre-filled with `0xDEADBEEF` and a bit of code that copies it into the array(s) you are investigating (Figure 4).

/ Example code that writes 0xDEADBEEF pattern into an array on a Microchip PIC24F microcontroller */*

```
#define ARRAY1_SIZE 16
uint8_t deadbeef[] = {0xDE, 0xAD, 0xBE, 0xEF};
uint8_t array1[ARRAY1_SIZE];

int16_t main() {
    uint8_t x = 0;
    uint8_t y = 0;

    while(1)
    {
        y = 0;
        for (x = 0; x < ARRAY1_SIZE; x++) {
            array1[x] = deadbeef[y];
        }
    }
}
```

```
y++;
if (y >= 4) {
    y = 0;
}
}
```

The code can be allowed to execute in its entirety until it either stops due to the error, or the microcontroller is halted by the developer. When this occurs, the memory can be analysed using the IDE. Even if the error caused the microcontroller to reset this does not clear the memory (unless some start-up code clears it all to zero) as long as it remains powered on. This means the debugger can still access the RAM in the state it was in before the reset and you can search for any buffer overruns that have overwritten your `0xDEADBEEF` patterns.

Filling the stack with 0xDEADBEEF before main()

Another possible issue is that the stack is being written beyond its limits. Many microcontrollers have hardware detection for stack overruns. However, such a feature may not be available, or you may have defined the use of RAM differently. The stack can, of course, be filled using the IDE as already mentioned. However, you may need to fill this space with your `0xDEADBEEF` pattern before your code executes (i.e., before `main()` is reached).

Microcontrollers have quite a bit to do before `main()`, more than we have time to cover here, but they include some hooks allowing user code to be executed before the CPU jumps to `main()`. This capability can be used to fill the stack with `0xDEADBEEF`.


```

79  while(1)
80  {
81      y = 0;
82      for (x = 0; x < ARRAY1_SIZE; x++) {
83          array1[x] = deadbeef[y];
84          y++;
85          if (y >= 4) {
86              y = 0;
87          }
88      }
89  }
90  }

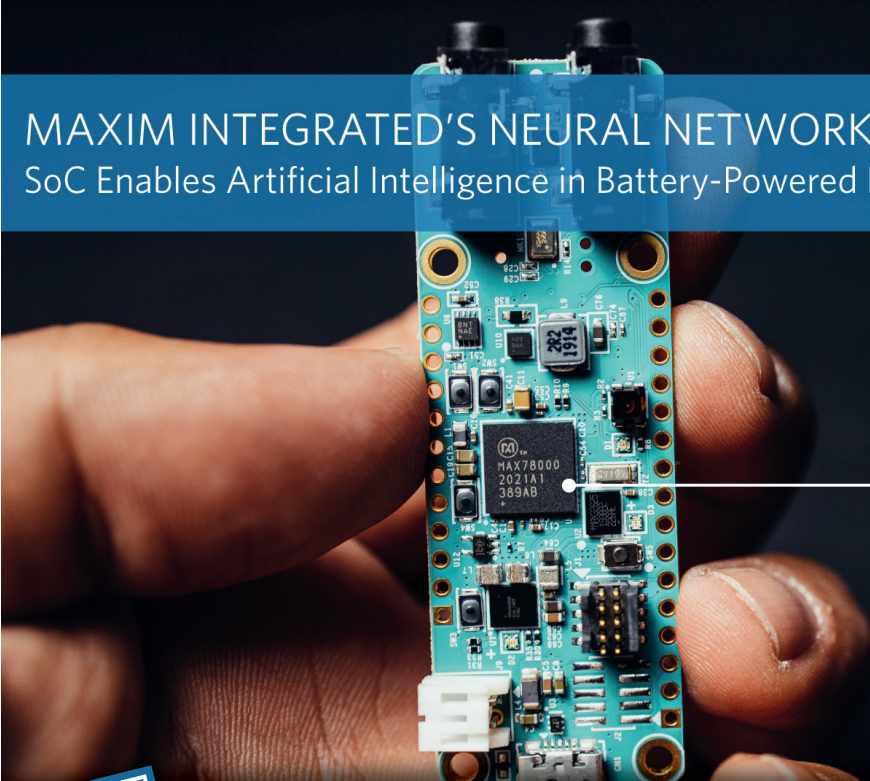
```

main > while(1) > for (x = 0; x < ARRAY1_SIZE; x++) > if (y >= 4) > then >


| Output | SFRs | Program Memory | File Registers x | Notifications | Variables | Call Stack | Breakpoints | | | | | | | | | | |
|---------|------|----------------|------------------|---------------|-----------|------------|-------------|----|----|----|----|----|----|----|----|----|-----------|
| Address | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | ASCII |
| 07E0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | ----- |
| 07F0 | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- | ----- |
| 0800 | 00 | 00 | 00 | 00 | DE | AD | BE | EF | DE | AD | BE | EF | DE | AD | BE | EF | |
| 0810 | DE | AD | BE | EF | 00 | 00 | 00 | 00 | DE | AD | BE | EF | 38 | 02 | 00 | 00 |8... |
| 0820 | 00 | 00 | 00 | 00 | 03 | 00 | 00 | 22 | 08 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |" |
| 0830 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | |
| 0840 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | |
| 0850 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | |

Figure 4: The example code has written the 0xDEADBEEF pattern into the array being analyzed (marked in orange).


MAXIM INTEGRATED'S NEURAL NETWORK ACCELERATOR SoC Enables Artificial Intelligence in Battery-Powered Devices



AI inferences at less than 1/100th the energy of other embedded solutions



maxim integrated™



Join the MAX78000 AI Design Contest at www.elektormagazine.com/ai-contest-max78000

Listing 1: user_init function for a PIC24 microcontroller.

```
/* **** */
/* Initialize DEADBEEF checking for stack */
/* **** */
void __attribute__((user_init)) deadbeefPrep(void) {
    uint8_t *stackPointer = (uint8_t*)0;
    uint16_t y = 0;

    // Find out where stack is pointing now from SP (WREG15)
    stackPointer = (uint8_t*) WREG15;

    // Increment to next memory position after current stack pointer location
    // (we have a byte pointer into word-addressable memory)
    stackPointer += 2;

    // Loop through stack area until we reach stack limit address (in SPLIM register)
    for (y = 0; stackPointer < (uint8_t*) SPLIM; ++stackPointer) {
        // Write part of 'DE AD BE EF' pattern into stack
        *stackPointer = deadbeef[y];

        // Cycle through four bytes of DE AD BE EF array
        ++y;
        if (y >= 4) {
            y = 0;
        }
    }
}
```

The method for doing this varies from microcontroller to microcontroller, but can typically be broken down into the following steps:

- Determine the start of the stack.
- Determine the end of the stack.
- Fill this region with `0xDEADBEEF`.

The example in **Listing 1**, written in C, is for a Microchip PIC24F 16-bit microcontroller, implementing the `user_init` function (search the compiler documentation [2] for more details) that will be called once the device is initialized but prior to `main()`.

Knowing that the stack pointer is stored in a specific register, in this case `WREG15`, the address to which it is pointing is stored in a pointer variable. In this architecture, this is pointing to the current return address for the function currently being executed, so we need to avoid overwriting it. Therefore, we increment the pointer by two bytes (this is a 16-bit device) so we can start writing the `0xDEADBEEF` pattern from the next word-addressable location in memory. The remainder of the code writes the pattern, stopping when the end of the stack is reached. This limit is determined by using the content of the stack limit register `SPLIM` (another a register unique to this processor architecture).

This approach can be adapted for most other microcontrollers quite easily, but will probably require some in-depth reading of the documentation for the microcontroller and the compiler tool chain.

The optimal approach to discovering buffer and stack overflows

Probably the most important thing to remember is that, if the code on a microcontroller doesn't work as expected, it is probably because of a programming error. Try to determine what causes the problem so that it can easily be replicated. From this point it will then be easier to determine which code changes you have made are resolving the issue and what the ultimate cause may be. If buffer or stack overflows are suspected, try to `0xDEADBEEF` them using one of the approaches discussed here.

200722-01

WEB LINKS

- [1] Elektor articles about microcontrollers: www.elektormagazine.com/tags/microcontroller
- [2] Microchip Technology, "MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs User's Guide," 2008: <https://bit.ly/3btCHec>