It's a Bug, not a Feature: Pitfalls of initialized BlockRAM use in soft processor systems

hochschule mannheim

Filip Brkic, Ruediger Willenberg
Department of Information Technology, Mannheim University of Applied Sciences, Germany

Motivation

It's a useful property for FPGA embedded systems that BlockRAM can come out of configuration pre-initialized with code and data. However, improper use of this property enables flawed software behaviour.

Example problem

This simple C program is compiled for a small *AMD Microblaze* processor system and BlockRAM is pre-initialized with its code and data:

```
#include <stdio.h>

int global = 42;

int main()
{
   xil_printf("global: %d\n", global);
   global++;
   return 0;
}
```

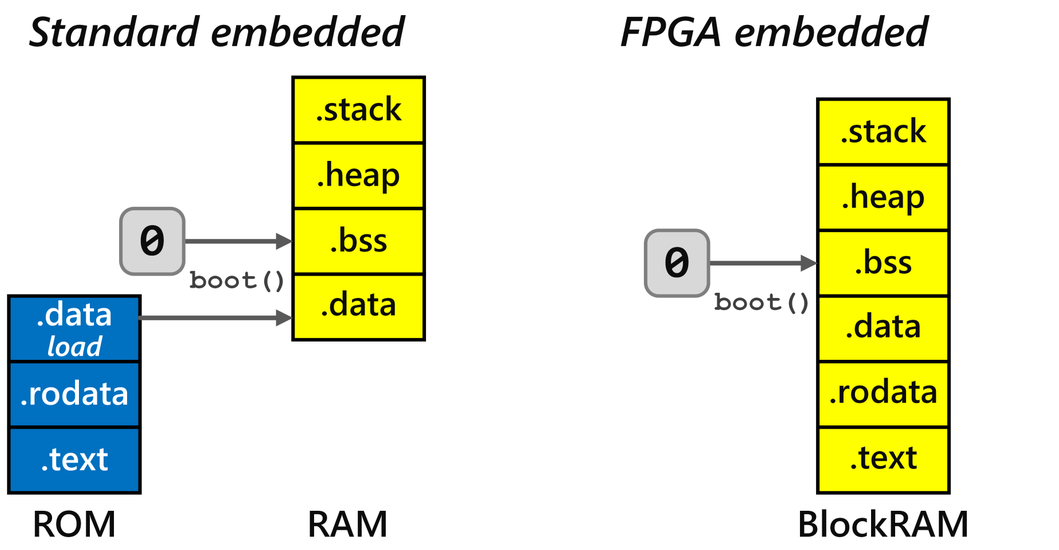
After FPGA configuration, the processor system's *stdout* is observed by UART. The program is then restarted twice through a *soft reset*:

```
global: 42
  -- first soft reset --
global: 43
  -- second soft reset --
global: 44
```

The variable `global` is apparently not re-initialized to its original state when the program restarts after the reset.
While this issue appears avoidable, it also affects standard library functions like `malloc()` and `printf()` that rely on initialized globals.

Explanation

In a common embedded system (left image), initial values for globals are stored in ROM together with constants and code. After each reset, the boot code copies the values into the actual *.data* segment in RAM.



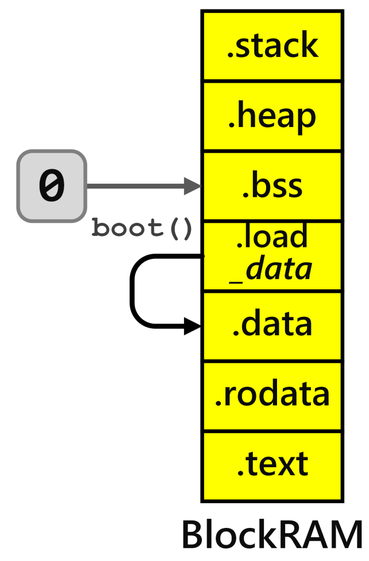
In the Microblaze system (right image), the *.data* segment in Block RAM is pre-initialized by the bitstream. The startup code provided by AMD/Xilinx does not re-initialize the segment after reset.

Microblaze Workaround I

In lieu of AMD, we are providing workarounds for Microblaze systems in our *Github* repository. They illustrate how other vendors' flows can be fixed, too.

The easiest workaround is to modify each global data section of the linker script so that it allocates an extra loadable copy in Block RAM, as illustrated in the image. For the Microblaze, we provide automated modification by shell script.

The boot process before `main()` must then be extended to initialize the global variables from the load copy. We provide two low-effort solutions to add this to the binary.

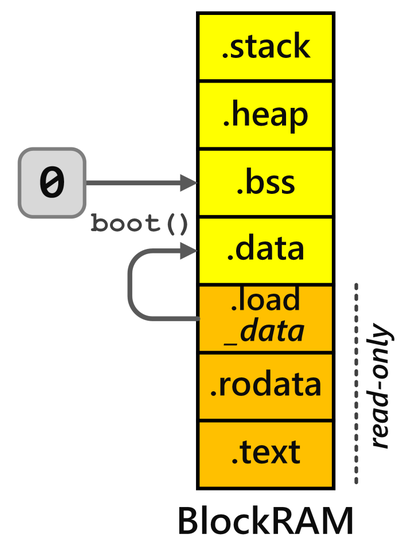


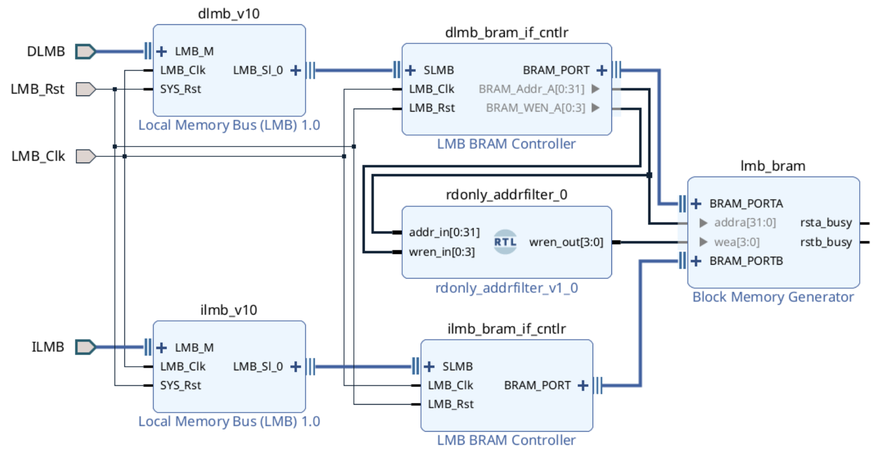
Microblaze Workaround II

Our second approach also allocates load copies of the global data. However, it logically splits the available Block RAM in half and groups segments into *read-only* and *writeable* segments. Again, a shell script is provided to modify the linker script.

The required extension to a binary's boot code is identical with the first approach.

However, the logical separation enables us to add a modification to the Microblaze system hardware that implements actual write protection for the read-only segment:





The advantage is that neither accidental nor malicious changes to code, constants or initialization values are possible anymore. The post-reset FPGA system acts like a common embedded system with true ROM.

Open source repository

In our *Github* repository, you can find the workaround code described above, tested for the classic (non-RISC-V) Microblaze with the 2024.1 tools (old and new Vitis IDE).
We also provide a paper with more extensive analysis (not part of FPL proceedings).

A survey of FPGA vendors' soft processor flows

Legacy core / RISC-V core	**Altera** Nios II/Nios V	**AMD/Xilinx** Microblaze/MB-V	**Efinix** Sapphire	**Lattice** RISC-V SM/MC/RX	**Microchip** Mi-V RV32
Behaviour examined in HW / only in SW tools	HW	HW	SW	SW	SW
Board Support Package tailored to HW	✓	✓	✓	✓	✗
Optional load segments in linker scripts	✓	✗	✗	✗	example
Boot code can re-load *.data* segment	✓	✗	✓	✗	✓
Read-only boot memory supported in HW design	✓	✗	✗	✓	✓
Load segments auto-assigned to read-only memory	✓	-	-	✗	example