Linux Embedded Virtual Machine

by Claus Schroeter
(clausi@chemie.fu-berlin.de)

Abstract

This paper describes the suggestions for a multi purpose generic driver mechanism. The Embedded Virtual Machine (EVM) is a virtual processor embedded in a Linux-Driver to provide a flexible and realtime capable programming interface to the hardware.

Why EVM ?

The design of a hardware driver is strongly dependent on the specialized capabilities and the complexity of the hardware. In this matter designing UNIX-Drivers seems to add more overhead to the driver design since the driver writer must take care on all the memory and process protection stuff. The design of a realtime-capable driver requires even more deep insights in the architecture and the whole timing structure of the kernel. This are only few reasons why hardware developers still rely on DOS in this case.
Linux-EVM should be one solution for this problem, it should serve as a testing and development environment for developers and a powerful instrument to drive hardware for the normal user. The second important advantage is that companies could provide binary code for supporting their hardware boards without providing the full source code.

What is EVM?

Linux-EVM is an virtual machine embedded in a normal linux device-driver that consists of all parts that are necessary for a full control over the hardware boards in the bus. A microcode can be downloaded to the EVM's virtual RAM and it can be executed by the virtual processor (VPU) to perform operations on the virtual port interface or any component of the EVM. The calling process has the full control over the VPU state registers and the virtual memory system, Data can be written or read from the EVM, traps can be triggered and branches can be requested from the calling process.

The EVM architecture

The Virtual Machine in EVM consists of four basic functional units, the VPU the memory subsystem the virtual interrupt controller and the virtual dma-controller. Additionally a port interface and a access mechanism to external memory is provided. Because the clock cycles in the emulation is comparatively slow the command set has to be as much effective as possible. As usual microprocessors the VPU has 8 16-bit multi-purpose registers (R0-R7) and 8 Registers that can have predefined functions for example a Stack pointer , PC and one status flag register. Assigned to the VPU there is a 64kWord memory area to hold the microcode. Additionally there are one DMA count and one DMA base register for the virtual DMA controller and two registers assigned to the virtual interrupt

controller.
To keep the architecture as effective and simple as possible, hardware port-access and access to any hardware memory page is implemented by special commands.

Operating Modes

The microcode can be downloaded by the calling process by performing the VPUBOOT ioctl and writing the microcode to the virtual RAM device. The microcode can be downladed to be started immediately after boot or holded in the memory as virtual BIOS for later use.
In this operating mode the VPU goes into HALT state immediately after boot and after return from exception. The calling process now can push some parameters in some of the VPU-Registers and perform a VPUTRAP ioctl to cause one of 16 software exceptions to start the desired exception handler.
In BOOT and RUN mode the VPU starts to process all microcode commands beginning at address 0h in the virtual RAM and runs continously until a HALT command is performed.

Virtual Communication Interface

Data can be read or written to the virtual RAM device directly or by using the virtual communication interface (VCI). The VCI emulates a true terminal interface that can be used with a terminal emulator or whatever to communicate with the EVM. The difference between communicating over VCI and by poking data directly to the virtual RAM is that the VCI must be serviced by the VPU to send or receive data.

Exception Handling

Exceptions can be triggered by one of the following conditions:

- a true hardware interrupt occurs
- a software interrupt (trap) has been performed
- the calling process sends a trap signal (VPUTRAP)

In case of hardware interrupt a flag will be set in the IRQP register and a indirect branch to a service handler (irq_vec) will be executed. Hardware interrupts can be enabled or disabled by setting the desired flags in the IRQE register of the virtual interrupt controller and performing the update command.
In case of software interrupt the desired trap vector will be executed. The exception vectors are located in the first 16 words of virtual RAM.

VPU capabilities

Internally the VPU contains a 16-bit Arithmetic Logic Unit (ALU)[1]. The ALU functions are Add or Substract with or without carry, bitwise logical operations, shift operations and management functions as clearing and saving/loadin the ALU flags.

Addressing and Branching Modes

Four fundamental addressing modes are supported by the VPU:

- register direct - refers the operation to one of the register contents
- register indirect - takes one register content as pointer to an address
- indirect indexed - an extension word contains the displacement to one of the register contents.
- indiect with post increment - same as register indirect but the regiter is incremented after the operation.

Four different modes of specifying the target address for branches are supported:

- Register Direct - the control is transferred to the location pointed to by one register.

[1]The register layout for the VPU and the operating modes has been taken over from HECTOR 1600 Programmers guide

- absolute - control is transferred to the location pointed to the content of the extension word
- relative - jump to the location calculated from the PC and one extension word
- indexed - jump to the location calculated from a memory cell and one register and one extension word

Opcode Format

The suggested 16-bit Opcode format is the same as for the HECTOR 1600[2]. For some opcodes as Branch or port accessing commands the bits 0-11 have different meanings. For normal adressed commands the Opcode word looks as follows:

15	14	13	12	11-10	9-8	7-4	3-0
OP3	OP2	OP1	OP0	Mode0	Mode1	Src	Dst

With

Bit 12-15 The 4 bit code is the Opcode

Bit 10-11 The source-adressing mode or fixed bit settings for special extended commands

Bit 9-8 The target-adressing mode or fixed bit settings for special extended commands

Bit 7-4 source descriptor

Bit 3-0 target descriptor

Acknowledgements

The idea to Linux-EVM has been inspired by discussions about a Laboratory Hardware driver API, the dosemu and the bsvc project.

[2] See the bsvc package

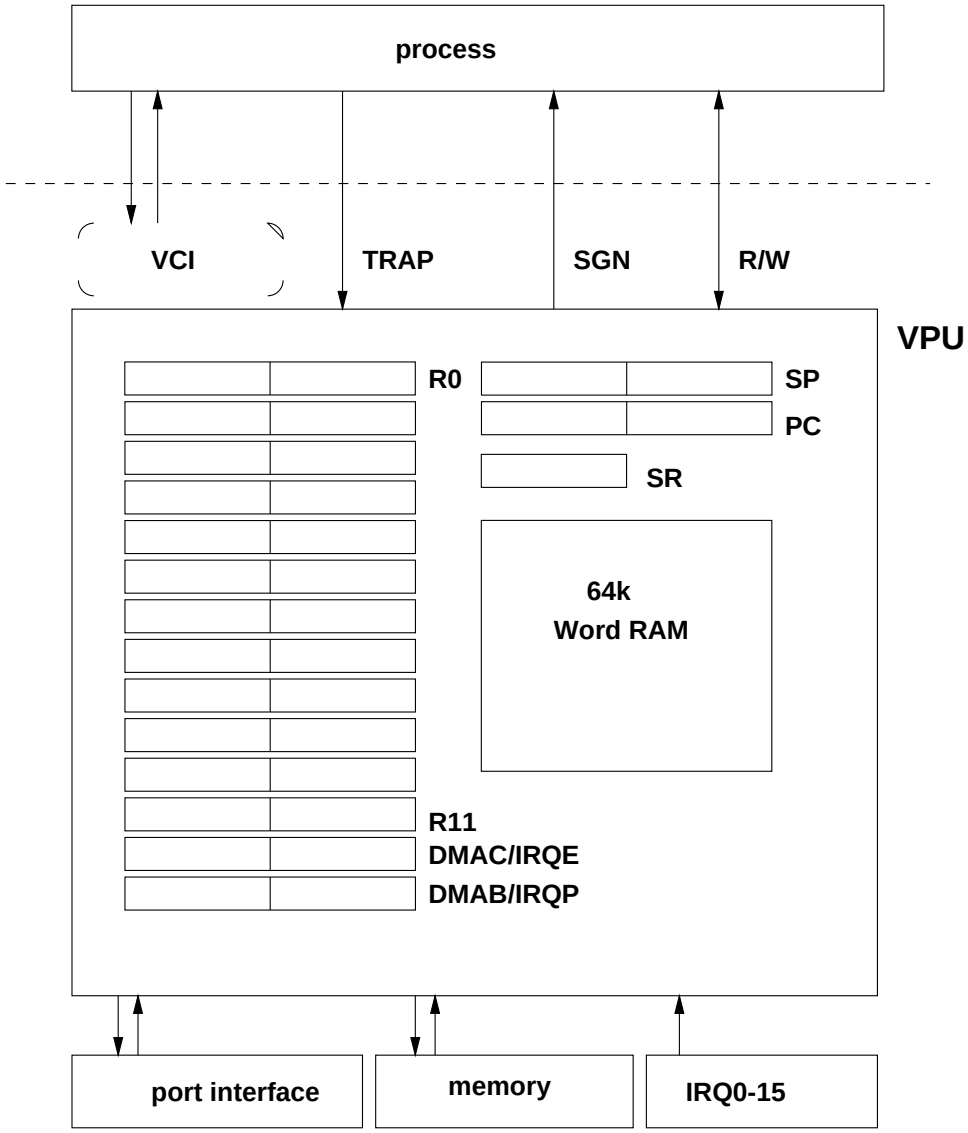

Figure 1: Architecture of Linux-EVM