C bare metal programming on ARM with Xilinx Microcontrollers

Presented by

Matteo Facchinetti

Embedded Systems Engineer for

Sirius Electronic Systems

facmatteo@gmail.com

Topics

- HW / SW equipment
- XSDK
- Bare-Metal programming intro
  - Register access
  - HW interrupt handler
- Example: gpio
  - HW / SW Description
  - Organize the code in memory: linker script
  - Boot sequence and C startup code

HW equipment

- Board:

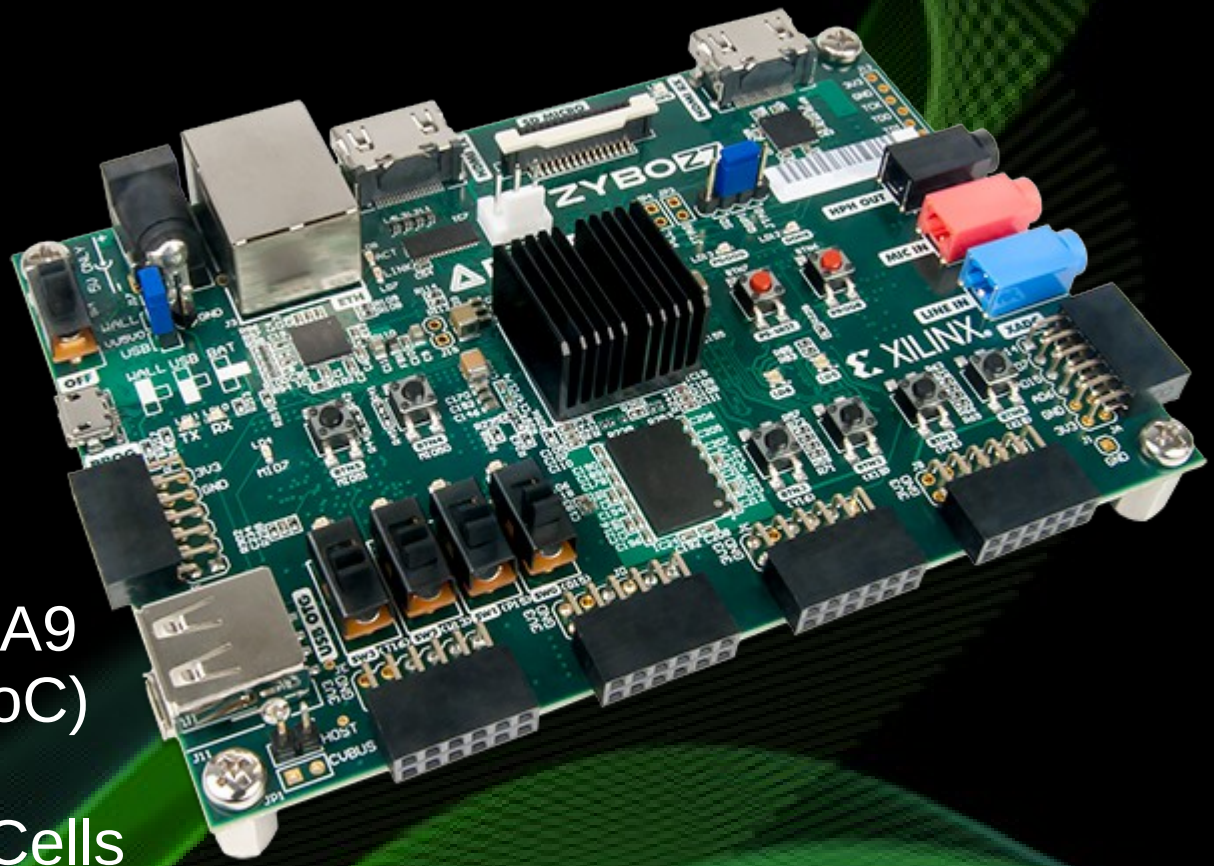
  Digilent ZYBO

- Microcontroller:

  Xilinx Z-7020

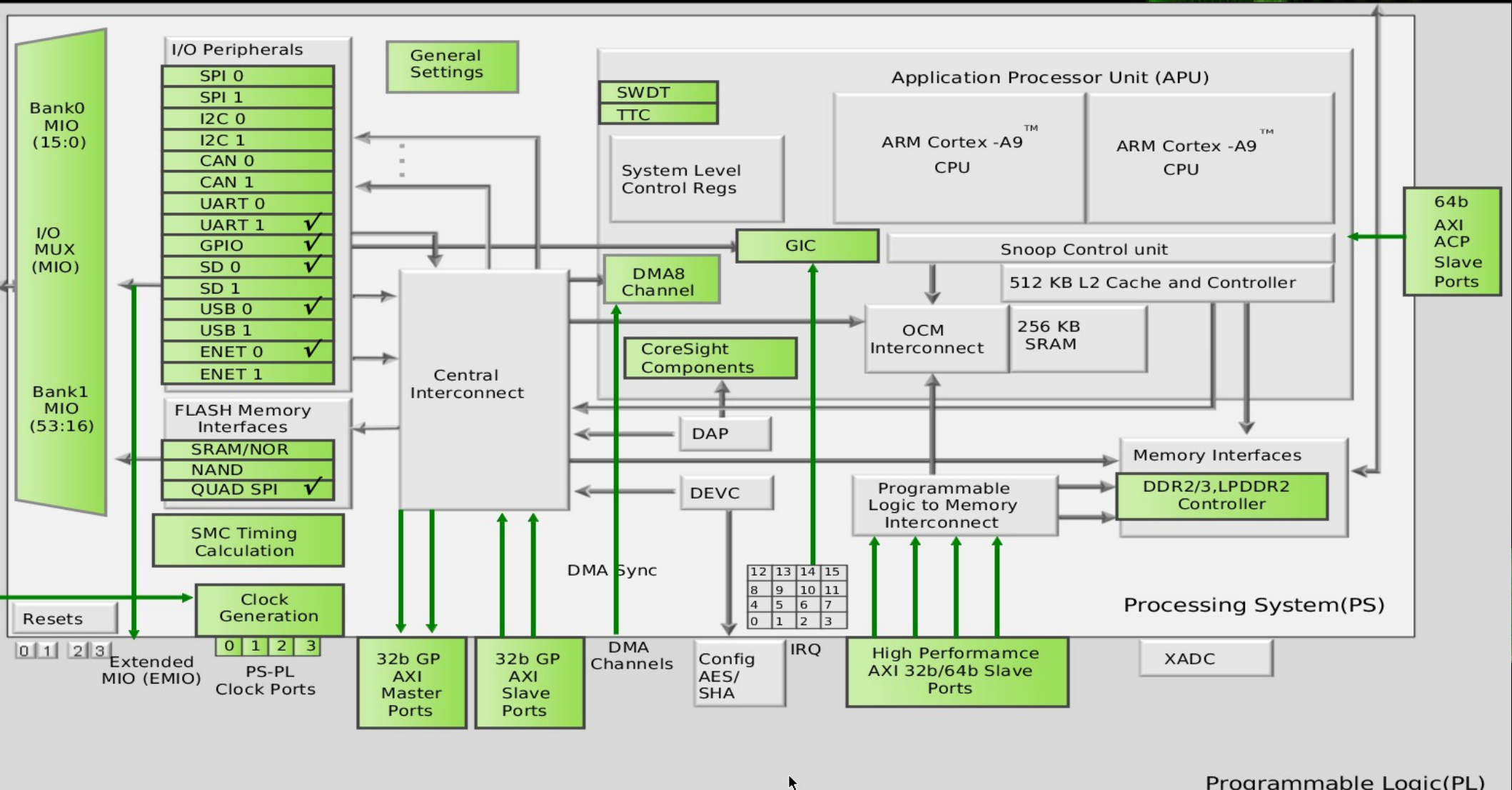
  Dual-Core ARMCortex-A9
  866 MHz (Zynq 7000 SoC)
  +
  FPGA Artix-7 85K Logic Cells

- FPGA IDE:
  - **Vivado**



HW equipment

- Zync SoC



SW equipment

- IDE:
  - **Xilinx SDK**

- Eclipse + plugin Xilinx
- Toolchain GCC for Armv7 Cortex-A:
  - Linux
  - Bare Metal

SW equipment

- IDE:
  - **Xilinx SDK**
  - **Gitlab docker build image**

https://gitlab.com/teox/petalinux-tools-docker

For legal distribution reasons, the Petalinux Tools files cannot be included with any public materials. To obtain a free legal copy of the PetaLinux 2018.2 Installer, please download it from the Xilinx website.

For the same reason you cannot push this Petalinux Docker image to the Docker Hub

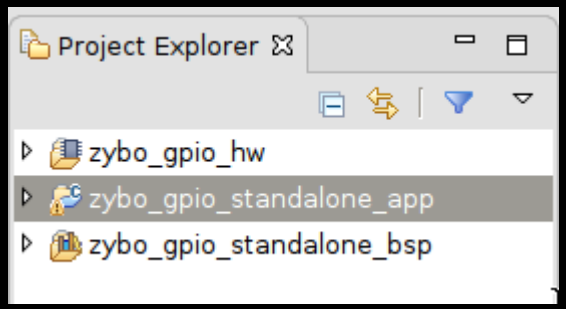
XSDK

Toolchain features

- arm-none-eabi-gcc
  - -mcpu=cortex-a9
  - -mfpu=vfpv3
  - -mfloat-abi=hard
- GCC version 7.2.1 20171011
  - Build Linaro GCC 7.2-2017.11-rc1
  - Thread model: single

XSDK

Workspace structure:



- zybo_gpio_hw:
  - hw init startup code
  - system desc: hdf file
- zybo_gpio_standalone_bsp:
  - Libraries configured for a specific HW
- zybo_gpio_standalone_app:
  - User application that use a specific BSP (Board Support Package)

XSDK

Base system library

- standard input/output
- access to processor HW features
- HW debug feature: I/O request from application to a host running a debugger
- Device drivers for all SoC (libxil.a)

This libraries are automatically included when create a “standalone” BSP

XSDK

Additional Libraries

- Libmetal
- Openamp
- Lwip: stack TCP/IP
- Xilffs: fat file system
- Xilflash: flash raw func
- Xilsf: Xilinx in System Flash
- Xilmfs: memory file system
- Xilpm: power menagement
- Xilrsa: RSA
- Xilskey: secure key

XSDK

At work:

- create a simple application with a related BSP.
  - (echo server)
- tour into workspace

Bare-Metal programming intro

- registers access
  - Direct Mapped Memory to access I/O peripheral registry
- HW interrupt handlers
  - No Operative System, so there's only a main process interrupted from HW

Direct Mapped Registry Access

- The easiest way:
  - Pointers to fixed address

```
#define REGBASE 0x40000000

unsigned int volatile * const reg = (unsigned int *) REGBASE;


*reg = value; /* write to port */

value = *reg; /* read from port */
```

  - “External factors” could change memory, so it must be labeled as volatile
  - Register address is const because can't change

Direct Mapped Registry Access

- Group of registers
  - Using define to simplify the code

```
#define TIMER_REG_BASE 0x40000000
#define TmLoad ((volatile unsigned int *) TIMER_REG_BASE) /* 32 bits */
#define TmValue ((volatile unsigned short *)(TIMER_REG_BASE + 0x04)) /* 16 bits */
#define TmClear ((volatile unsigned char *)(TIMER_REG_BASE + 0x08)) /* 8 bits */
```

```
unsigned short short_val;
```

```
*TmLoad = (unsigned int) 0xF00FF00F;
short_val = *PortValue;
*TmClear = (unsigned char) 0x1F;
```

Warning!

- define directive depends from the compiler
- could cause incomprehensible compiler errors.

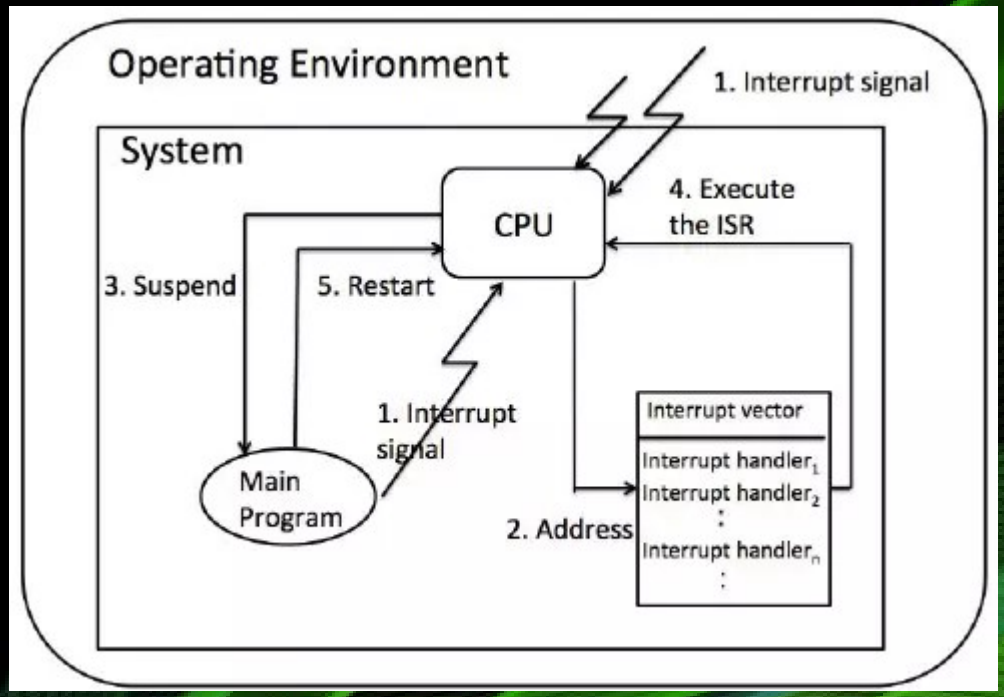
Direct Mapped Registry Access

- Group of registers
  - Using struct to improve portability

```
#define TIMER_REG_BASE 0x40000000
struct TimerRegs {
  unsigned int Load; /* offset 0 */
  unsigned short Value; /* offset 4 */
  unsigned short dummy1;
  unsigned char Clear; /* offset 8 */
  Unsigned char dummy2[3];
};
volatile struct TimerRegs *tm = (struct TimerRegs *) TIMER_REG_BASE
tm->Load = (unsigned int) 0xF00FF00F;
```

HW interrupt handlers

- Interrupt vector mechanism



HW interrupt handlers

- Interrupt vector mechanism:
  - stop CPU for a HW event
  - save the PC of the next instruction on the stack
  - jumps to the memory location of the interrupt vector table
  - gets the ISR (Interrupt Service Routine) address from the vector table and jumps to it
  - execute the ISR code and restore the previews PC.

HW interrupt handlers

- Register a Interrupt Service Routine

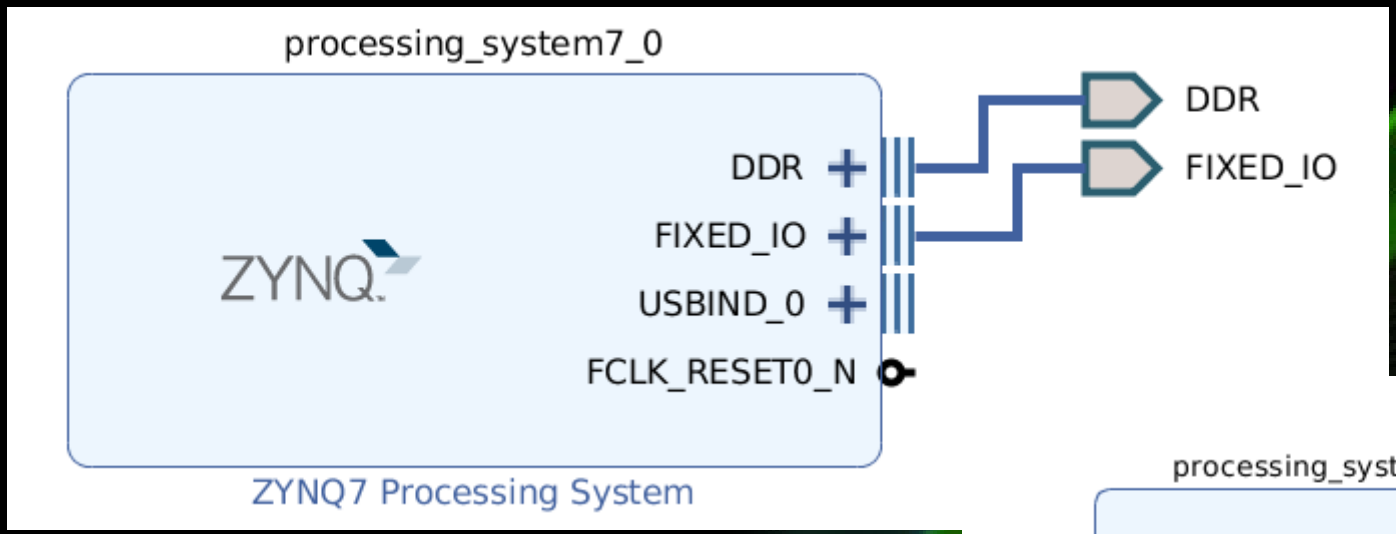
  Generic Interrupt Controller (GIC) driver

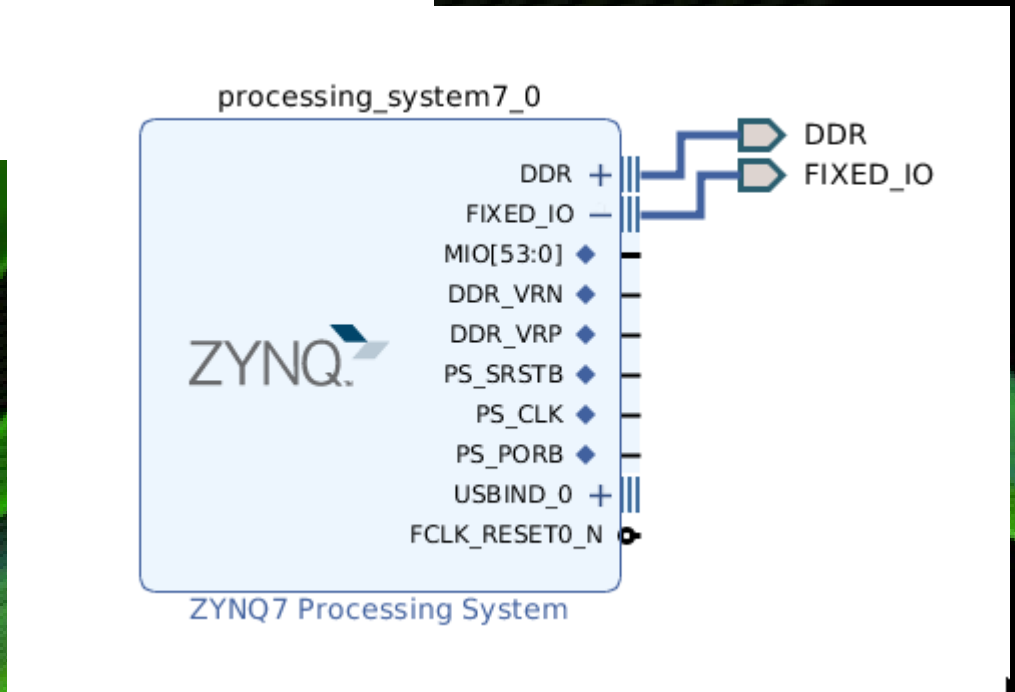
```
XScuGic_Connect(InterruptController, 61, IRQHandler, InterruptController);
XScuGic_Enable(InterruptController, 61);


void IRQHandler(void *CallbackRef) {
  print("RcfgModuleC IRQ Received!\n\r");
}
```

Example: gpio

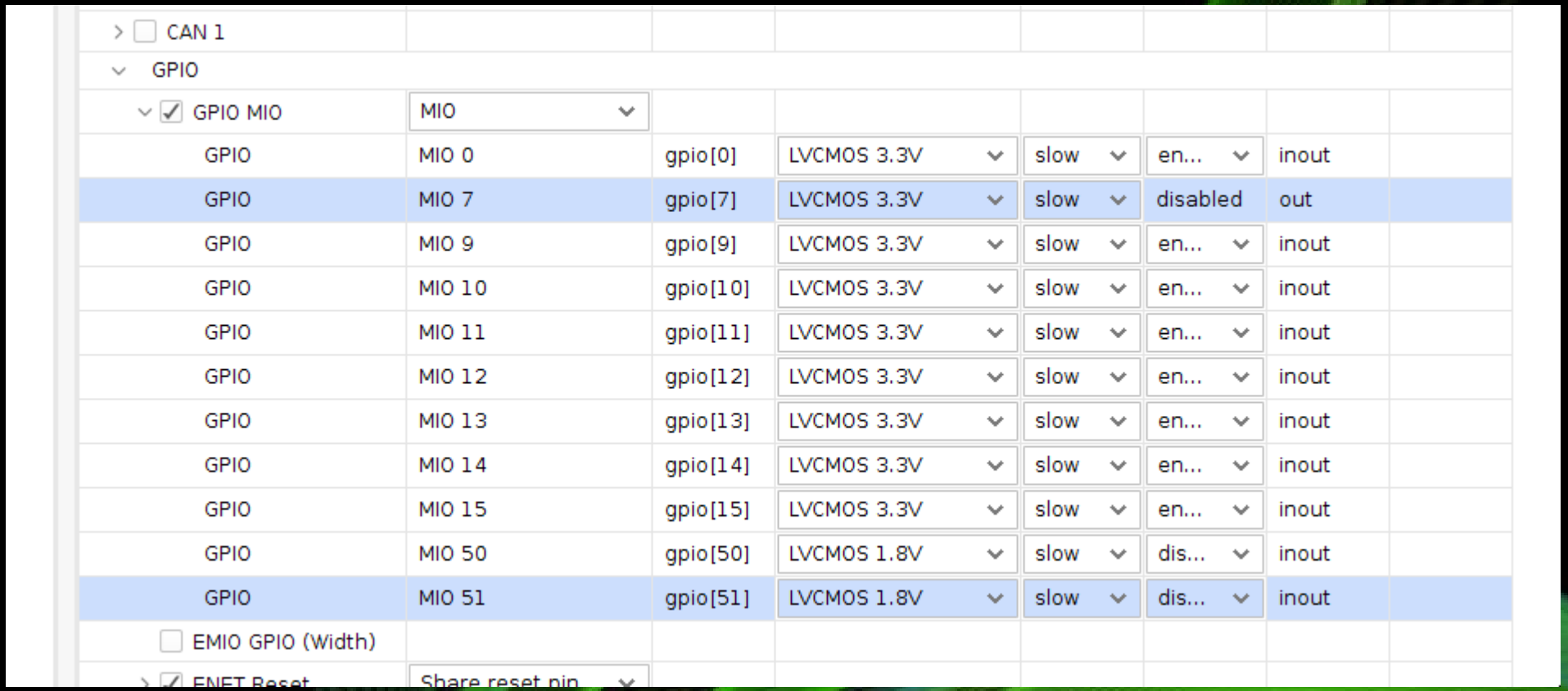
- HW description





Example: gpio

- HW description



CAN 1						
GPIO						
GPIO MIO	MIO					
GPIO	MIO 0	gpio[0]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 7	gpio[7]	LVCMOS 3.3V	slow	disabled	out
GPIO	MIO 9	gpio[9]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 10	gpio[10]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 11	gpio[11]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 12	gpio[12]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 13	gpio[13]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 14	gpio[14]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 15	gpio[15]	LVCMOS 3.3V	slow	en...	inout
GPIO	MIO 50	gpio[50]	LVCMOS 1.8V	slow	dis...	inout
GPIO	MIO 51	gpio[51]	LVCMOS 1.8V	slow	dis...	inout
EMIO GPIO (Width)						
ENET Reset	Share reset pin					

MIO7: LED

MIO51: SWITCH

Example gpio

- SW description
  - gpio initialization

```
#define OUTPUT_PIN 07   /* MIO7 */
#define INPUT_PIN  51   /* MIO51 */
/* init */
ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
XGpioPs_CfgInitialize(&GpioPs, ConfigPtr, ConfigPtr->BaseAddr);
/* config LED */
XGpioPs_SetDirectionPin(&GpioPs, OUTPUT_PIN, 1);
XGpioPs_SetOutputEnablePin(&GpioPs, OUTPUT_PIN, 1);
/* config SWITCH */
XGpioPs_SetDirectionPin(&GpioPs, INPUT_PIN, 0);
```

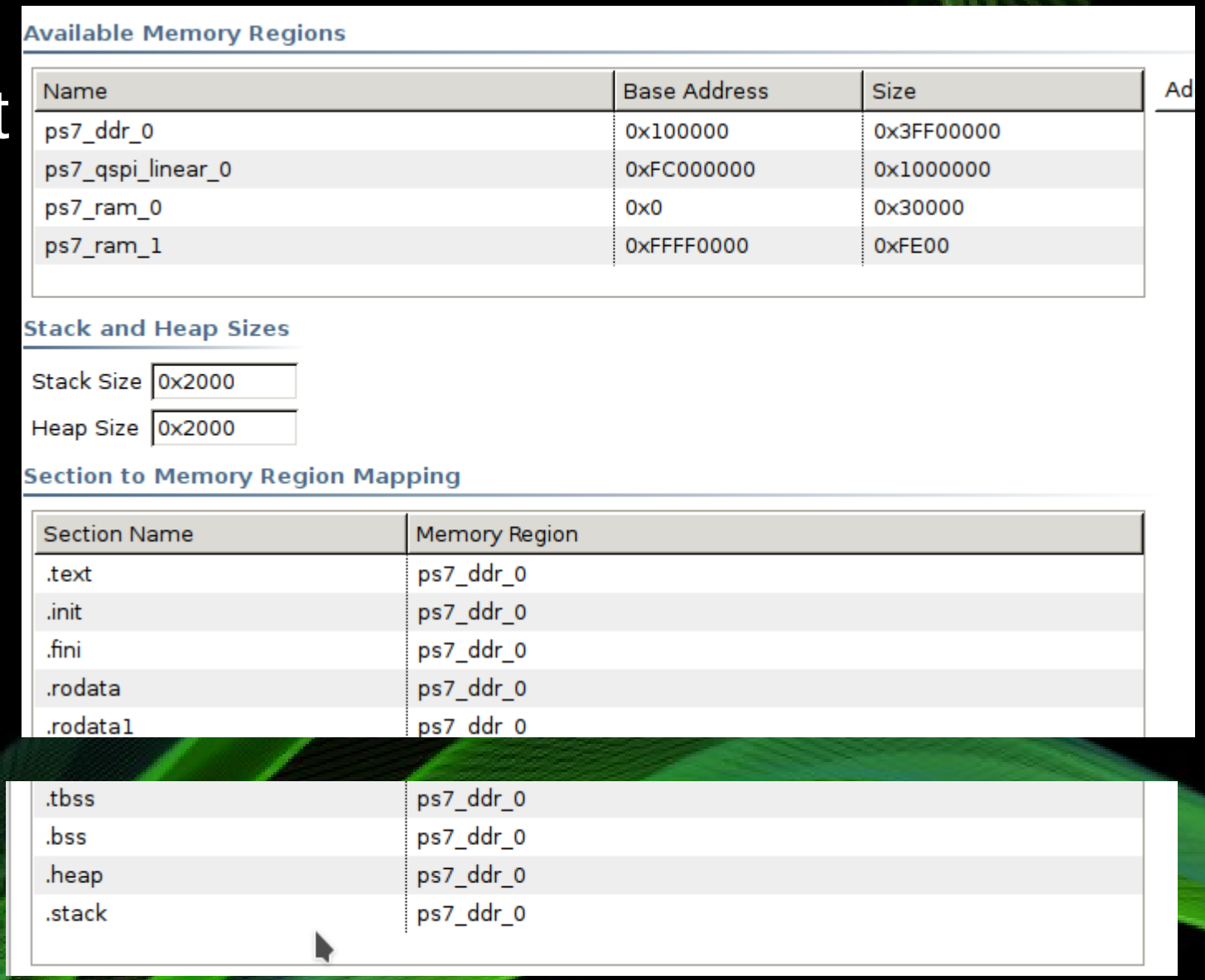
Example gpio

- SW description
  - main loop

```
printf("GPIO MMIO SWITCH LED start...\n");
while (1) {
  uint32_t data = XGpioPs_ReadPin(&GpioPs, INPUT_PIN);
  XGpioPs_WritePin(&GpioPs, OUTPUT_PIN, data);
  if (data)
    printf("Hello World %ld\n\r", cnt++);
}
printf("GPIO MMIO SWITCH LED stop...\n");
```

Example gpio

- Linker script



Available Memory Regions

Name	Base Address	Size	Ad
ps7_ddr_0	0x100000	0x3FF00000	
ps7_qspi_linear_0	0xFC000000	0x1000000	
ps7_ram_0	0x0	0x30000	
ps7_ram_1	0xFFFF0000	0xFE00	

Stack and Heap Sizes

Stack Size 0x2000

Heap Size 0x2000

Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.tbss	ps7_ddr_0
.bss	ps7_ddr_0
.heap	ps7_ddr_0
.stack	ps7_ddr_0

Example gpio

- Linker script
  - memory regions

```
/* Define Memories in the system */
MEMORY
{
   ps7_ddr_0 : ORIGIN = 0x100000, LENGTH = 0x3FF00000
   ps7_qspi_linear_0 : ORIGIN = 0xFC000000, LENGTH = 0x1000000
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
}
```

Example gpio

- Linker script
  - sections

```
...
.bss (NOLOAD) : {
   __bss_start = .;
   *(.bss)
   *(.bss.*)
   *(.gnu.linkonce.b.*)
   *(COMMON)
   __bss_end = .;
} > ps7_ddr_0
...
```

__bss_start and **__bss_end** are symbols defined here and is possible to use in your application

all parts ***(...)** give instruction to GCC where place these sections of the ELF binary code format

> ps7_ddr0 give instruction to GCC to append .bss section to the DDR at the first aligned free space next to the previous section.

Example gpio

- Boot sequence

boot.S → xil-crt0.S → main()

boot.S and xil-crt0.S are part of BSP standalone

Example gpio

- boot.S
  - init low level CPU features
    - disable MMU...cache...
  - jump to C startup code

Example gpio

- xil-crt0.S
  - init ELF sections required by C
    - .bss and others
  - init low level feature needed by libxil.a
    - AMP – PROFILING stuff
  - jump to C main function

XSDK

At work:

- tour into example gpio