CONTROLLING GPIO ON EMBEDDED LINUX

Oliwier Jaworski
PXL hogeschool
Elektronica ICT
oliwier.jaworski@student.pxl.be

ABSTRACT

This guide describes the steps required to go from a hardware design to an embedded application that interfaces with the onboard LEDs and buttons of the PYNQ-Z2. The implementation relies on the AXI interface in the programmable logic, a custom Linux distribution running on the ARM processing system, and the UIO (Userspace I/O) driver to enable communication between the user-space applications and hardware.

Keywords: PYNQ-Z2, Petalinux, Userspace I/O driver, AXI-gpio, embedded linux, User-space, Kernel-space

1. INTRODUCTION

Developing applications on the PYNQ-Z2 typically requires bridging custom hardware blocks in the programmable logic (PL) with software running on the ARM cores in the processing system (PS). To accomplish this, an interconnect protocol is required. For ARM-based Zynq devices, the AXI (Advanced eXtensible Interface) fabric is used for communication between the PS and PL.

Vivado provides prebuilt AXI GPIO IP blocks that can be instantiated to simplify the hardware design. After the design has been tested and the bitstream successfully generated, the next step is to build the system image using the PetaLinux tools. These tools streamline the process of configuring the kernel and root file system through user-friendly interfaces and command-line utilities. They also allow the user to access and modify the device tree, which is necessary to expose the custom hardware to the Linux system.

Once the image has been built and flashed to an SD card, the system can boot into the custom Linux distribution. At this stage, the hardware can be accessed from user-space applications through the UIO (Userspace I/O) driver. UIO exposes the hardware registers to user space by creating a virtual mapping of the device's memory. This mapping can be accessed using the mmap system call, allowing user applications to directly read from and write to the GPIO hardware register space.

2. HARDWARE REQUIREMENTS

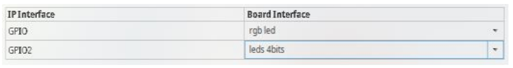
This is the minimal hardware list to be able to reproduce this tutorial, take note that this tutorial with a little bit of research, can be ported to other hardware too.

- Host machine (compatible linux distribution)
- Vivado, Petalinux, Vitis (version 2024.1)
- PYNQ-Z2
- Ethernet and micro-USB data cable
- SD-card(atleast 8GB)

3. HARDWARE DESIGN

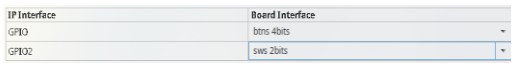
Start by creating a new empty Vivado project, then create a block design. This is where all IP blocks will be visually connected. For this project, only the shipped IP blocks are used.

1. **Add the ZYNQ7 Processing System** block to the block design.
   - This block provides the software interface around the Zynq-7000 Processing System and acts as the logic connection between the PS and the PL.
2. **Add two AXI GPIO blocks** to the design.
   - These blocks provide an AXI-based interface for the PS, allowing later access to these peripherals via memory mapping.
3. **Configure the first AXI GPIO** block:



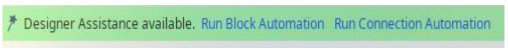
IP Interface	Board Interface
GPIO	rgb led
GPIO2	leds 4bits

4. **Configure the second AXI GPIO** block:

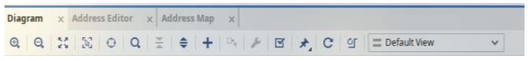


IP Interface	Board Interface
GPIO	btns 4bits
GPIO2	sws 2bits

5. **Run block automation** for the ZYNQ7 Processing System, leaving all default settings.
6. **Run block automation** for the ZYNQ7 Processing System, leaving all default settings.
7. **Run connection automation**, also leaving default settings.

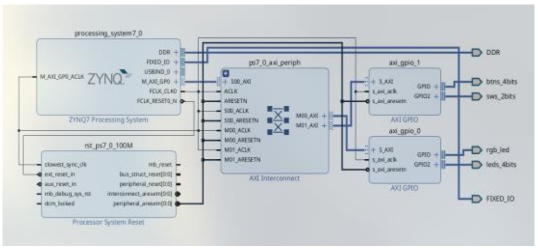

Designer Assistance available. Run Block Automation Run Connection Automation

8. **Edit the block design layout** to make it more readable. Use the "Regenerate Layout" option at the top of the block design window.



9. **Validate the design** using the option in the top tray of the block design window.

- This is necessary to detect connection, data width, or protocol errors early, without encountering them during packaging.

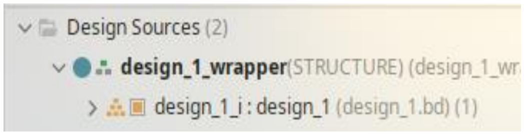


10. **Create the HDL wrapper** by selecting the block design (usually called design_1) under the Sources section and clicking "Create HDL Wrapper." Let Vivado manage and auto-update the wrapper.

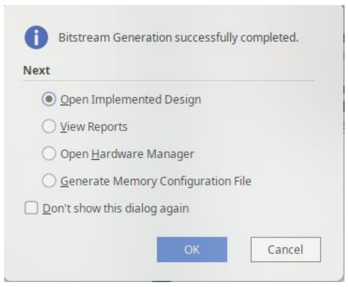
- Once successful, a wrapper will appear over your block design file.

11. **Generate the bitstream** under the PROGRAM AND DEBUG section, leaving default settings.

- *Note:* The bitstream is generated for the design set as the top design, which is distinguished by bold text in the Sources panel.


Design Sources (2)
design_1_wrapper(STRUCTURE) (design_1_wr
design_1_i : design_1 (design_1.bd) (1)

12. Verify bitstream generation


Bitstream Generation successfully completed.

Next

[✓ Open Implemented Design]
View Reports
Open Hardware Manager
Generate Memory Configuration File

Don't show this dialog again

OK Cancel

- Once the bitstream generation completes, Vivado will display a message indicating that the design has been successfully implemented and compiled to FPGA logic.
- This confirms that your hardware design is ready to be exported for use in the PetaLinux build process.

13. **Export the hardware design.** Go to File → export → export hardware

- Make sure to include bitstream
- Save the exported hardware platform (XSA file) in your workspace directory for later use in the PetaLinux build.

4. BUILD DEVICE TREE BLOB

Once the hardware design has succesfully been exported. The next step is to generate the **device tree** for the given design.

- A **device tree** is a data structure used by the Linux kernel to describe the hardware components of a system. It informs the kernel about available peripherals, memory regions, interrupts, and other hardware-specific configurations, allowing the operating system to interface correctly with the hardware without hard-coded assumptions.
- In this workflow, the device tree will include nodes for the AXI GPIO blocks and any other custom IP, enabling user-space access via the UIO driver.

The problem with the automatically generated device tree is that it does not assign **UIO (Userspace I/O)** as a compatible driver for the AXI GPIO peripherals. By default, the generated nodes typically use the **xlnx,axi-gpio** binding, which means no corresponding /dev/uio* device file will be created by the kernel. To fix this we will first extract the device tree from the design*.xsa and in the section to follow incorporate the device tree changes to our custom linux image.

1. Source the **Xilinx design tools**

- Before using PetaLinux or related build tools, the Xilinx environment must be sourced by running the setup script, which is typically located under

```
~/tools/Xilinx/S*/V*/settings64.sh
```

- where S* and V* correspond to the specific software (e.g., Vitis) and version you want to use.

3. **Obtain the Xilinx device tree sources** by cloning the official repository into your workspace:

```
git clone -b xilinx_v2024.1
https://github.com/Xilinx/device-tree-xlnx
```

4. **Change** into the newly cloned directory

```
cd device-tree-xlnx
```

5. Run **XSCT** command

- *Note*: This command must be executed in the same terminal where the Xilinx environment was sourced.

6. **Open XSA/HDF** file using the following command:

```
hsi open_hw_design <design_name>.<xsa|hdf>
```

7. Set repository path to the cloned device-tree-xlnx repository

```
hsi set_repo_path <path to device-tree-xlnx
repository>
```

8. **Extract the processor cell name** which will return a list of valid processors

```
set procs [hsi get_cells -hier -filter
{IP_TYPE==PROCESSOR}]
```

9. **generate a device tree source** tailored to the exported hardware.

```
hsi create_sw_design device-tree -os
device_tree -proc ps7_cortexa9_0
```

- folder named device-tree
- **-os device_tree** specifies that the operating system for this project is a device tree generator, not a full OS.
- **-proc psv_cortexa72_0** specifies the processor instance that the software design will target.

10. **Generate DTS/DTSI** files to folder

```
hsi generate_target -dir  DT_EXTRACT
```

11. **Clean up.**

```
→ hsi close_hw_design [hsi current_hw_design]
→ exit
```

Take note of the generated pl.dtsi file in the output folder. This file contains the device tree entries for the GPIO components instantiated in the hardware design, and these entries will later be modified to bind the peripherals to the UIO driver for user-space access.

```
...
/ {
        amba_pl: pl-bus {
        ...
        axi_gpio_0: gpio@41200000 {
            ...
                };
                axi_gpio_1: gpio@41210000 {
            ...
                };
        };
};
```

5. BUILDING CUSTOM PETALINUX IMAGE

Building the custom Linux image with PetaLinux follows a straightforward workflow. First, a new PetaLinux project is created and configured according to the hardware design and system requirements. During this configuration step, additional packages can be selected to be included in the root file system. Once the project is set up, the image is compiled and written to an SD card, either with dedicated tools or by manually copying the output files.

Although the process is documented in detail in Xilinx reference material, several difficulties may arise. In particular, some configurations provided by the board support package (BSP) are not automatically applied when using an exported hardware design (XSA). This requires manual adjustment to enable certain features.

This section will walk through the complete process, including the creation of the PetaLinux project, configuration of the kernel, root file system, and PetaLinux settings, as well as the necessary modifications to the device tree to add driver support for the GPIO hardware.

1. Creating the petalinux project

- the following command can be used to create a petalinux project from an xsa file.

```
petalinux-create project -n PYNQZ2_SDL2 --
template zynq --force
```

- **-n** can be replaced by the user desired name
- **--template zynq**, we select the option to use a template project based on our processing system architecture.
- **--force** makes petalinux create the project folder even if it already exists overwriting old contents.

2. Adding **hw-description** using .xsa

```
petalinux-config –get-hw-description=${full-
path-to-xsa}/*.xsa --silentconfig
```

- replace the placeholder with the full-path to your *.xsa file
- this step will incorporate your custom hardware design into the project.

3. Adding kernel **boot arguments**

```
petalinux-config
```

- This will open a window inside the command line, in which the petalinux system configuration can be done. We are interested in adding boot arguments so we select DTG Settings → kernel bootargs → Add extra boot argument

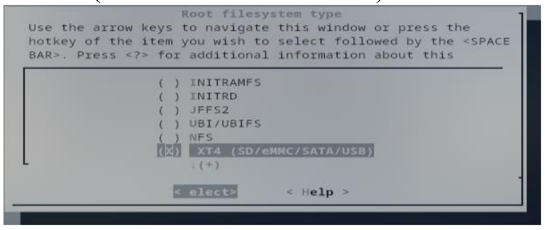
- A new window will pop up in which you will need to write the following line to enable binding the UIO driver to device.

```
uio_pdrv_genirq.of_id=generic-uio
```

- *Note:* Don't forget to save your configuration before exiting the configuration panel.

4. Setting the **root file system type** to **EXT4.**

- This is necessary so that the bootloader can correctly locate and mount the root file system during boot.
- To achieve this go back to the root of the petalinux system configuration and select Image Packaging Configuration→ Root filesystem type(press space to select) EXT4(SD/eMMC/SATA/USB)



```
                    Root filesystem type
 Use the arrow keys to navigate this window or press the
 hotkey of the item you wish to select followed by the <SPACE
 BAR>. Press <?> for additional information about this

              ( ) INITRAMFS
              ( ) INITRD
              ( ) JFFS2
              ( ) UBI/UBIFS
              ( ) NFS
              (X)  XT4 (SD/eMMC/SATA/USB)
                  ↓(+)

               < elect>      < Help >
```

- *Note:* Don't forget to save your configuration before exiting the configuration panel.
- Once you exit, the configuration changes will be applied automatically; at this stage, the only concern is whether the process completed successfully.

5. Enabling root login

- By default, root login is disabled, leaving no way to log in to the system. This can be corrected by adjusting the root file system settings. In this guide, only the simplest method will be shown. Note that alternative approaches exist for creating users or changing passwords within the PetaLinux project itself.
- To enable root login we will configure the rootfs with the following command:

```
petaliinux-config -c rootfs
```

- In **Image Features**, enable **debug-tweaks** to allow root login, and optionally enable **empty-root-password** to skip the login process entirely.

6. changing the **GPIO driver** inside the petalinux **device tree**

- navigate to the following directory inside your petalinux project

```
project-spec/meta-user/recipes-bsp/device-
tree/files/system-user.dtsi
```

- Open the file with your prefered text editor. It will look as following:

```
/include/ "system-conf.dtsi"
/ {
};
```

- The previously generated device tree will now be used to modify specific entries for the AXI GPIO hardware. By adding the following lines, the default driver for the GPIO blocks is replaced with the **UIO driver**, enabling user-space access.

```
/include/ "system-conf.dtsi"
/ {
};
&axi_gpio_0{
    compatible = "generic-uio";
};
&axi_gpio_1{
    compatible = "generic-uio";
};
```

- By doing this, both GPIO peripherals will appear as /dev/uioX devices, allowing direct access from user-space applications using **mmap**

7. Enabling **UIO kernel driver** on boot**.**

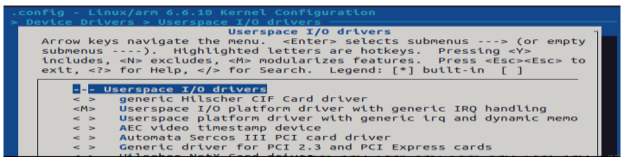
- To ensure the UIO driver is loaded automatically at boot the kernel configuration must have UIO support enabled. This can be achieved by changing a value inside the kernel configuration
- accessing the kernel configuration happens with the following command

```
petalinux-config -c kernel
```

- Inside the kernel configuration open the following sections

  Device Drivers → Userspace I/O drivers

- Select **Userspace I/O (UIO) platform drivers** with **generic IRQ handling** and set it to `*` (built-in) instead of `M` (modular). This ensures that the UIO driver is compiled directly into the kernel and available immediately at boot, without requiring manual module loading.



```
.config - Linux/arm 6.6.10 Kernel Configuration
> Device Drivers > Userspace I/O drivers
                         Userspace I/O drivers
   Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
   submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
   includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
   exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

       --- Userspace I/O drivers
       < >   generic Hilscher CIF Card driver
       <M>   Userspace I/O platform driver with generic IRQ handling
       < >   Userspace platform driver with generic irq and dynamic memo
       < >   AEC video timestamp device
       < >   Automata Sercos III PCI card driver
       < >   Generic driver for PCI 2.3 and PCI Express cards
```

!Note: No additional packages were added to this PetaLinux build, so it will include only the default packages provided by PetaLinux.Links to instructions for adding custom packages are provided in the *Noteworthy* section of this tutorial.

Once all configurations have been applied to meet the user's requirements and ensure the minimum settings for driver support, the final step is to build the custom Linux system using the following command:

```
petalinux-build
```

!Note: The build process can take a significant amount of time, may fail on fetching data. If that happens just rerun the command. and may consume large amounts of **RAM**. Based on experience, it is recommended to **enable swap memory** or increase it to at least **32 GB**. Without sufficient memory, the build can quickly max out system RAM, causing the system to become unresponsive and potentially stalling the build indefinitely.

6. PACKAGING CUSTOM LINUX

After the build completes, a new directory is created inside the PetaLinux project at **/images/linux/**. This directory contains all files required to create the boot image and package the Linux distribution into a **.wic** file. The resulting **.wic** file can then be written to the SD card using a tool such as **Balena Etcher**, or alternatively by manually copying the relevant files to the appropriate SD card partitions.

1. Generation of boot image(**BOOT.BIN**)

- The boot image (**BOOT.BIN**) typically contains the **First Stage Boot Loader (FSBL)**, which initializes the processing system (DDR, clocks, and I/O) and then loads the **Second Stage Boot Loader (U-Boot)**. U-Boot provides the environment to load and start the Linux kernel.
- Optionally, the boot image can also include the FPGA bitstream to configure the programmable logic. In this tutorial, the bitstream (**system.bit**) is included. However, Xilinx also provides the **fpgautil** package with Zynq processing systems, which allows users to load FPGA logic dynamically at runtime. This tutorial does not cover the use of fpgautil or the steps required to enable UIO on such a setup.
- The **BOOT.BIN** file is generated inside the **images/linux/** directory, as the command structure and available arguments make this the most convenient working location.

```
petalinux-package boot --fsbl zynq_fsbl.elf --
fpga system.bit --u-boot u-boot.elf
```

- The following command must be executed from the base directory of the **PetaLinux project**; otherwise, the **rootfs.tar.gz** file will not be found.

```
petalinux-package wic -b "BOOT.BIN, uImage,
boot.scr"
```

This command creates a .wic image by bundling the following list of components:

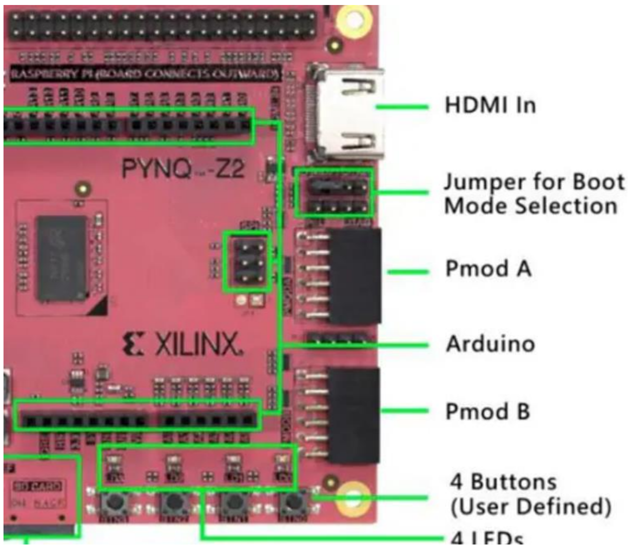
- **BOOT.BIN -** Contains the First Stage Boot Loader (FSBL), U-Boot, and optionally the FPGA bitstream. It is the first image loaded by the Zynq boot ROM.
- **Uimage** - The Linux kernel image in U-Boot–compatible format. It is the binary executed after U-Boot passes control to the kernel.
- **Boot.scr** - A compiled U-Boot script (boot.cmd → boot.scr) that contains the boot commands. It instructs U-Boot how to load the kernel, device tree, and root filesystem.

Together, these files form the minimal set required for booting the custom Linux system from the SD card.

7. IMAGE FLASHING

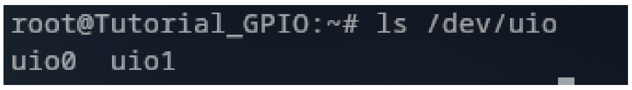
To flash the image onto the **SD** card, **Balena Etcher** was used. Simply select the generated **.wic** file, choose the target SD card, and click Flash. Wait for the process to complete before removing the card.

Note: Ensure the boot mode jumper is set to SD rather than **JTAG**; otherwise, the board will not boot from the SD card.



7. FUNCTIONALITY CHECK

Once the board has booted and the login screen appears, log in as **root**. To verify that the **UIO driver** has been bound correctly, check for the presence of device files under **/dev/uio***. These entries should exist if the configuration was done correctly.

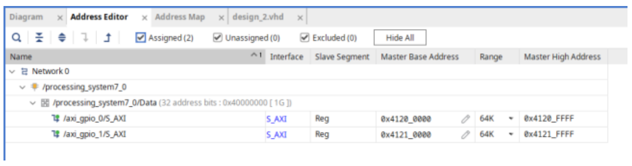


```
root@Tutorial_GPIO:~# ls /dev/uio
uio0  uio1
```

After verifying the presence of the **/dev/uio*** entries, you can inspect the details of a bound device (such as its name, base address, offset, and size) by reading from the sysfs interface:

```
cat /sys/class/uio/uio0/maps/
```

It is up to the user to determine which device corresponds to each UIO entry by comparing the reported base addresses with the address mapping defined in the Vivado project. These mappings can be found in the **Address Editor** section of **Vivado**.



Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
/axi_gpio_1/S_AXI	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF

There are two common methods to obtain the register mapping for each device:

1. Using **Vitis**:

- Create a new Vitis project and locate the header file corresponding to the device, e.g., **xgpio_l.h.**
- This file defines the offsets for each register:

```
#define XGPIO_DATA_OFFSET 0x0   /**< Data
register for 1st channel */

#define XGPIO_TRI_OFFSET  0x4   /**< I/O
direction reg for 1st channel */

#define XGPIO_DATA2_OFFSET       0x8   /**<
Data register for 2nd channel */

#define XGPIO_TRI2_OFFSET 0xC   /**< I/O
direction reg for 2nd channel */



#define XGPIO_GIE_OFFSET  0x11C /**< Glogal
interrupt enable register */

#define XGPIO_ISR_OFFSET  0x120 /**< Interrupt
status register */

#define XGPIO_IER_OFFSET  0x128 /**< Interrupt
enable register */
```

2. Using the **generated AXI implementation files** (for custom AXI IP only

- Open the generated VHDL implementation file for the custom IP
- There is usually a commented section that outlines each register offset and its function, which can be used to determine the mapping.

For standard IP blocks like AXI GPIO, the Vitis header file method is preferred as it is easier and more reliable. For custom AXI IP, the VHDL implementation file is often the only source of detailed register information.

7. APPLICATION CROSS COMPILATION

There are many ways to cross-compile an executable for the ARMv7 architecture, but this guide will focus on the method that requires the least effort. An alternative approach will be briefly mentioned for reference.

Cross compiling using the petalinux SDK

1. Navigate to your **PetaLinux project folder**.
2. Build the **SDK** by running:

```
petalinux-build -sdk
```

   !Note: The build process can take a significant amount of time and may consume substantial system resources.

3. After the build completes, locate and run the generated **sdk.sh** script in **images/linux/**. You will be prompted to provide a path where the root filesystem and cross-compiler should be installed.
4. Once the installation finishes, the script will provide instructions on how to set up the cross-compilation environment. Follow these instructions to source the environment.
5. To verify the cross-compiler, run:

```
echo $CC
```

- The output should show the name of the cross-compilation toolchain, typically:

```
arm-linux-gnueabihf-gcc
```

   when targeting an ARM processor.

An example application that uses a Makefile to build the executable can be found in the Github repository, inside the **gpio_app folder**.

8. RESULTS

Finally, the application can be tested. Using mmap, the device memory is mapped to virtual memory, allowing access from user-space. By configuring the registers as shown in **Example 1** and writing to the data registers, the **LED** states can be controlled.

Note: *As illustrated in the example and hardware design, no interrupt handling is implemented in this tutorial. Handling button interrupts and registering them will be covered in a future tutorial.*

```
#include <stdio.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <stdint.h>
#include <sys/mman.h>
#include <unistd.h> // for usleep

#define GPIO_DATA_OFFSET 0x8
#define GPIO_TRI_OFFSET  0xC

#define UIO_DEVICE "/dev/uio0"
#define GPIO_SIZE  0x10000

int main(void) {
    int fd = open(UIO_DEVICE, O_RDWR);
    if(fd < 0) {
        perror("Failed to open UIO device");
        return -1;
    }

    // Map GPIO memory
    uint32_t *gpio = (uint32_t*) mmap(NULL, GPIO_SIZE, PROT_READ | PROT_WRITE,
                                      MAP_SHARED, fd, 0);
    if(gpio == MAP_FAILED) {
        perror("mmap failed");
        close(fd);
        return -1;
    }

    // Set all pins to output
    gpio[GPIO_TRI_OFFSET/4] = 0;

    uint32_t led_value = 1;
    while(1) {
        gpio[GPIO_DATA_OFFSET/4] = led_value; // turn on one LED
        usleep(500000); // delay 0.5 seconds

        // move to next LED in circle
        led_value *= 2;
        if(led_value > 8) led_value = 1; // wrap back to first LED
    }

    munmap(gpio, GPIO_SIZE);
    close(fd);
    return 0;
}
```

Example 1: LED Blinking Pattern

REFERENCES

- Testing UIO with Interrupt on Zynq Ultrascale
- Refer to the Boot Mode Jumper Configuration image provided by Xilinx
- How could /dev/mem Linux directory be used in order to control the peripherals (MM/IO) ?
- Linux GPIO Driver by xilinx
- Build Device Tree Blob by xilinx

- GPIO and Petalinux -part 1, medium, 15 june 2020, Roy Messinger
- MicroZed Chronicles: UIO Part One - Introduction and Set Up,adiuvoengineering , 23 august 2023, AMD
- PetaLinux Tools Documentation: Reference Guide (UG1144),2025-05-29, AMD
- Include Custom C Application in Petalinux (embedded linux) kernel on AMD Zynq Z2 board, */*/2024, fpgabe
- Custom Application Creation in PetaLinux on the Zynqberry, hackset.io, */*/2019,Whitney Knitter
- How to register my device as UIO on PetaLinux,digilent forums, 03/09/2016, ,izumitomonori
- Koch, Hans J. "Userspace I/O Drivers in a Realtime Context." Linutronix GmbH, Bahnhofstr. 25, 88690 Uhldingen, Germany. Email: hjk@linutronix.de
- Linux Drivers list, 18/06/2025, Confluence wiki admin
- Solution ZynqMP PL Programming, 05/06/2025, Confluence Wiki Admin
- Linux Loadable Kernel Modules,05/09/2019 ,linn
- Using Linux UIO drivers for reserved vidememory?,reddit/embedded,*/*/2024,TimeDilution
- How to list the kernel Device Tree [duplicate], Unix&Linux,*/*2014
- BSP for petalinux/Pynq,AMD PYNQ, 03/2022,abdyoyo