Cora Z7-10 petalinux project

Detailed guide on petalinux project creation

Dmitrii Matafonov

OS

Contents

Introduction

This guide is intended to give a step-by-step guidelines on how to create a linux project based on the *.xsa file exported from Vivado. It would automatically configue base drivers and kernel setup, except for the custom IP module. The guidelines are going to give examples on how to write a kernel module to handle custom IP's interrupt and route it to a software.

Tools

Petalinux tools can be used only on linux machines. It works well with WSL2 on Windows, but the project must be in the EXT4 local virtual hard drive, or it would as slow as if you were running it on a machine from the 80s.

> ***Note***: Native installations support Ubuntu 20.04 maximum. You WILL experience a lot of issues if you run it on 22.04.

If you run the newest one, a pre-made docker image might be a good solution for you: petalinux-docker[1]. The installation is very clear and simple.

For native installations the links are down below.

Petalinux 2021.2: download link[2]

Required packages: download link[3]y

Minor suggestion: I would recommend installing mc (Midnight commander) for the convenience of editing and navigating in the console. It works great on your machine and on the embedded linux we would be building.

Listing 1: Optional: mc installation

```
sudo apt-get install mc
```

How to install it on the board will be covered in the scope of this document.

SD card

SD card has to be formatted to FAT32 filesystem. 512 Mb SD card is usually enough, but I use 4, 8 or 16 Gb ones. If one wants the linux image to have persistent storage (save files and changes across reboots),

[1] https://github.com/carlesfernandez/docker-petalinux2
[2] https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html
[3] https://support.xilinx.com/s/article/73296?language=en_US

they need to make 2 partitions: FAT32 for bootloader and EXT4 for rootfs. Detailed instructions are here[4].

External kernel sources

Since this project is aimed at creating a Real-time patched version of Xilinx kernel, it is necessary to download the external kernel sources package to be able to patch it and build easily without messing with the original petalinux installation.

Linux Xilinx kernel 2021.2[5].

For more specific applications, you may want to utilize different kernels from other providers, such as Analog devices, they provide additional kernel drivers for their own products, it may be useful for automotive applications.

The patching approach remains the same for those versions of kernel.

[4] https://docs.xilinx.com/r/2021.2-English/ug1144-petalinux-tools-reference-guide/Partitioning-and-Formatting-an-SD-Card

[5] https://github.com/Xilinx/linux-xlnx/releases/tag/xilinx-v2021.2

Petalinux project development

Before using petalinux tools, one needs to source the settings file located in petalinux installation dir.

Listing 2: Sourcing petalinux settings

```
source ~/petalinux/settings.sh # adjust to your petalinux installation folder
```

Petalinux tools quick reference

Listing 3: Quick reference for petalinux commands

```
# To create a new folder with initial petalinux project template use
petalinux-create --type project --template zynq --name *your_project_name*

# To make initial linux project autoconfig from the HDL design use
petalinux-config --get-hw-description ~/*your_project_folder*
# After that you can just use petalinux-config to edit the existing settings

# /*To change kernel config (add kernel drivers,FS system, kernel tweaks) use */
petalinux-config -c krenel

# To change the contents of the build's rootfs use
petalinux-config -c rootfs

# To change the build's u-boot settings use
petalinux-config -c u-boot

# To build the project use
petalinux-build

# To fully clean the build use
petalinux-build -x mrproper

# To build some component (kernel, for example) separately use
petalinux-build -c kernel
# or u-boot, rootfs, u-boot, device-tree

```

```
# To pack the build into writable files under ??/images/linux/
petalinux-package --boot --fsbl --fpga --u-boot --force

# In case of InitRAM pacakging type use
petalinux-package --boot --fsbl --fpga --u-boot --kernel --force
# to embed everything into a single BOOT.BIN

# To boot linux in qemu emulator
petalinux-package --prebuilt --force
petalinux-boot --qemu --prebuilt 3

# If necessary, you can try to boot the whole image over JTAG using
petalinux-package --prebuilt --force
petalinux-boot --jtag --prebuilt 3
# It's a VERY long process
```

All the changes to the default tamplate are stored in `project-spec`.

Path	Description
project_folder/project-spec/meta-user/recipes-apps	**sources for user applications**
project_folder/project-spec/meta-user/recipes-bsp	**sources for device-tree, u-boot and FSBL**
project_folder/project-spec/meta-user/recipes-kernel	**sources for linux kernel changes**
project_folder/project-spec/meta-user/recipes-modules	**sources for linux kernel modules**

New project

To create a new project, make a folder where you'd want to keep the projects. If one wants to do this in this repository, they may use a command similar to this:

Listing 4: Creating a new project

```
cd ~/Cora_Z7/linux_build_cora # it assumes the repository is stored in
    your home dir
petalinux-create --type project --template zynq --name
    your_project_name # choose a new name for the project
```

> ***Note***: this step is the same for custom boards as well.

To make the default configuration based on the design created at the previous step in Vivado, issue:

Listing 5: First configuration based on Vivado design

```
petalinux-config --get-hw-description ~/Cora_Z7/linux_build_cora/hw # adjust to the used path for *.xsa file
```

This is the "main", overall configuration menu. It has the basic settings for with adjustments for the specific Vivado design already in place. At this staep no changes are needed, but it would be nice for the reader to make themselves familiar with the settings.

Click `ESC` 2 times to exit the settings menu and let the tool finish the configuration.

It is not necessary for many simple projects to have an external kernel, but for this course because of the Real-Time patch it is going to be needed.

Download Xilinx krenel for 2021.2: link[6].

Untar it somewhere (`tar -xvzf ./linux-xlnx-xilinx-v2021.2` to untar locally), I suggest using the `linux_cora_build` folder, one level above from petalinux project.

Issue `petalinux-config` and navigate to `Linux components selection`.

1. Choose `linux-kernel`.
2. Choose `ext-local-src`.
3. The new menu entry will appear called `External linux-kernel local source settings`. Choose it.
4. Edit `External linux-kernel local source path` to point to the untared linux sources.
5. *Optional*. If persistent storage is needed (i.e. files are preserved across reboots, changes are permanent), in the settings top menu choose `Image packaging` **`configuration`** -> `Rootfs filesystem` **`type`** -> choose `EXT4 (SD/eMMC/SATA/USB)`. The SD card must have 2 partitions for this (link[7]).
6. `ESC` until it exits. Save changes when prompted.

Now the first basic changes are made, the first test image may be built.

Building and packaging the project

Building is simple, just issue

[6] https://github.com/Xilinx/linux-xlnx/releases/tag/xilinx-v2021.2
[7] https://docs.xilinx.com/r/2021.2-English/ug1144-petalinux-tools-reference-guide/Partitioning-and-Formatting-an-SD-Card

Listing 6: Building the linux image

```
petalinux-build
```

and wait for the completion. There should be no errors since it is built with default settings from a template. The build products (built linux image) will be located in `project_folder/images/linux`.

Packaging options are different and are based on the `Image packaging configuration`.

1. If no changes were made at configuration stage, the default option is `InitRAM`. There are several options on how to package it.
    1. `petalinux-package --boot --fsbl --fpga --u-boot --force` It packages bootloaders and kernel+rootfs separately. If one used this option, they need to copy `BOOT.BIN`, `boot.scr` and `image.ub` to the FAT32 SD card partition.
    2. `petalinux-package --boot --fsbl --fpga --u-boot --kernel --force` It packages everything into a single BOOT.bin image, including kernel and rootfs. It would be the only file one needs to copy to the FAT32 partition.

> ***Note***: in this packaging mode the changes are not saved across reboots. It is useful if one needs a static image.

2. If the chosen packaging mode is SD card, the packaging command is `petalinux-package --boot --fsbl --fpga --u-boot --force`. Files to copy: `BOOT.BIN`, `boot.scr`, `image.ub` to FAT32 partition. With **sudo file manager** (`sudo mc`, for example) copy the **contents** of the rootfs.tar.gz into the EXT4 partition.

Insert the SD card into the slot and power up the board. Connect to the serial console using putty or other terminal. Make sure it boots, after that you may attach an ethernet cable to have SSH functionality. By default, the login/password is root/root.

Modifying the project

Generally, all the modifications that are necessary are stores in `project-spec` folder. `meta-user` folder contains recipes (Yocto) that need to be introduced into the final build. Here are the main folders in `meta-user` that are important.

Folder	Description
recipes-apps	Contains the recipes for applications. Create new app templates with `petalinux-create --type apps --name app_name --enable`. It would appear here for the user to edit.
recipes-bsp	Contains the recipes for device-tree (structure to connect the kernel drivers to the hardware), u-boot and FSBL
recipes-core	Contains the recipes for core system functionality. For example, if you want to change the default behaviour of the SSH server, it would be there to place the files
recipes-kernel	Usually it stores the changes that the user has made through nemuconfig (`petalinux-config -c kernel`)
recipes-modules	It stores the recipes for kernel modules

FSBL patch

First stage bootloader is a standalone baremetal application provided by Xilinx that set up the processor for running and that handles the contents of BOOT.BIN file. By default, there are no debug prints (or any prints) from it, but sometimes it may be necessary to have those. Unfortunately, there are bugs in the menu config in petalinux that accept changes from the user for the build flags for FSBL, but they don't make it into the final image.

To bypass that issue, there is an option to customize the build process for the proces.

Navigate to (create if it doesn't exist) to `./project-spec/meta-user/recipes-bsp/embeddedsw/`. Create an empty folder called `files` and a file called `fsbl-firmware_%.bbappend`, if they don't exist.

Add the following contents to the `fsbl-firmware_%.bbappend` file.

Listing 7: Adding debug functionality to FSBL

```
#Add debug for FSBL(optional)
XSCTH_BUILD_DEBUG = "1"

#Enable appropriate FSBL debug or compiler flags
YAML_COMPILER_FLAGS_append = " -DFSBL_DEBUG_INFO -DRSA_SUPPORT"
```

In this example, not only debug prints are added to the fsbl, but also RSA support, it is good to have

this option at the early stage just not to come back to this if one needs to enable secure boot in the future.

Kernel module

There is a simplier way to have access to FPGA resources with generic-uio drivers that support interrupts, but this example is intended to give more details and flexibility to the reader to be able to create their own custom modules and projects, handle the interrupts their own way. The users would be able to have custom interrupt handlers to make more of their FPGA designs than just RW access.

To create a template for kernel module that would handle the interrupt from the FPGA, use

Listing 8: New kernel module in petalinux project

```
petalinux-create -type modules --name module-name --enable
```

> ***Note***: Don't use '_' in naming.

In this repository it is named `fpgatimer`.

Petalinux would create a template for a kernel module. Replace the contents of `./project-spec/meta-user/recip` with the contents from **Appendix A** where the source code is presented.

This module registers itself based on the interrupt number provided through device tree and relays the hardware interrupt to the userspace using OS signal SIGUSR1.

It gives better performance compared to regular polling and frees the application from the wasteful constant polling. It may be very important in real-time (or performance sensitive) applications.

Device-tree

To connect the interrupt in the FPGA fabric to the kernel module, a change to the device tree is needed.

Changes to the default configuration are made in the `./project-spec/meta-user/recipes-bsp/device-tree/`

> This file overwrites the device-tree nodes if they are present in the default device tree. Otherwise, it adds the missing configurations.

```
/*Add this to the end of the file*/
/*Interrupt handler custom kernel module*/
&simple_hardware_timer_0 {
    compatible = "homeuser,fpgatimer";
    interrupt-parent = <&intc>;
```

```
6       interrupts = <0 44 1>;
7       };
```

The entry name is taken from Vivado design. It overwrites the default blob entry.

> If unsure how the entry was names, after the first build the default generated device tree can be found at `./components/plnx_workspace/device-tree/device-tree/pl.dtsi`. This is an gutogenerated folder and no changes would be saved here.

Interrupt parent comes from the fact that the interrupt is connected to the main PS interrupt. Interrupts property reflects that this is not an SPI interrupt (0), interrupt number comes from Vivado design and 1 means rising edge triggering.

Custom application

In order to have your application embedded, there is an approach to compile sources as the part of the build process, but my personal preferrance is to embedd precompied and tested applications.

In order to do that, create a new application with

```
petalinux-create --type apps --name your_app_name --enable
```

Navigate to `./project-spec/meta-user/recipes-apps/your_app_name/files` and replace everything in that folder with the files from your precompiled application.

One level above, modify your `your_app_name.bb` in the following fashion.

```
#
# This file is the your_app_name recipe.
#

SUMMARY = "Short your_app_name description"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
    "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://file1 \
           file://file2 \
           file://file3 \
           file://file4 \
           file://file5 \
          "
```

```
16
17 S = "${WORKDIR}"
18
19 FILES_${PN} += "/home/root/your_app_name_folder/*"
20
21 do_install() {
22     install -d ${D}/home/root/your_app_name_folder
23     cp ${S}/file1 ${D}/home/root/your_app_name_folder/file1
24     cp ${S}/file2 ${D}/home/root/your_app_name_folder/file2
25     cp ${S}/file3 ${D}/home/root/your_app_name_folder/file3
26     cp ${S}/file4 ${D}/home/root/your_app_name_folder/file4
27     cp ${S}/file5 ${D}/home/root/your_app_name_folder/file5
28 }
```

Custom application auto-start

> Auto-start is not featured in the repository's recipes.

If boot-time auto-start is necessary, a startup script is a nice-to-have.

```
1 petalinux-create --type apps --name startup --enable
```

Navigate to the newly created application's folder and edit the *.bb file similar to the following:

```
1  #
2  # This file is the startup recipe.
3  #
4
5  SUMMARY = "Startup script which starts from init.d and can be edited
       at ~/startup.sh"
6  SECTION = "PETALINUX/apps"
7  LICENSE = "MIT"
8  LIC_FILES_CHKSUM =
       "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
9
10 SRC_URI = "file://startup.sh \
11             file://startup_init \
12             "
13
14 S = "${WORKDIR}"
15
16 FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
17
```

```
inherit update-rc.d

INITSCRIPT_NAME = "startup_init"
INITSCRIPT_PARAMS = "start 00 5 ."


do_install() {
        install -d ${D}${sysconfdir}/init.d/
        install -m 0755 ${S}/startup_init
           ${D}${sysconfdir}/init.d/startup_init
        install -d ${D}/home/root
        install -m 0755 startup.sh ${D}/home/root/startup.sh
}

FILES_${PN} += "${sysconfdir}/*"
FILES_${PN} += "/home/root/*"
```

The `startup_init` could be the following:

```
#!/bin/sh

if [ "$1" = "start" ]; then
    echo " "
    echo "Launching custom startup.sh"
    sh /home/root/startup.sh
fi

if [ "$1" = "stop" ]; then
    echo "Startup executing stop"
fi

exit 0
```

The `startup.sh` is the call for your application to launch or whatever is necessary to be done at the startup.

Real-time patch

To make the kernel give more throughput for computing and application's reactions, there is a real-time patch specific to the kernel version.

Guidelines for this approach were taken from Hackster.io[8].

[8] https://www.hackster.io/LogicTronix/real-time-optimization-in-petalinux-with-rt-patch-on-mpsoc-5f4832

Link to the patch specific to 2021.2 kernel version: 5.10-rt16[9]

Link to the patches storage: Linux real-time patches[10]

The necessary patch is also stored in this repository in `project-spec` folder.

To patch the kernel call patch from the kernel sources directory.

Listing 9: Patching kernel for real-time

```
cd ~/Cora_Z7/linux-xlnx-xilinx-v2021.2 # adjust to your location
zcat ../linux_build_cora/project-spec/kernel-5.10-realtime_patch.gz |
    patch -p1 # adjust to your location
```

Kernel settings

After patching issue

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

to adjust the settings to take advantage of the patch.

Settings to change:

1. In the menuconfig choose `General setup`.
2. Navigate to `Preemption model` and choose `Fully preemptible kernel (Real-time)`
3. Make sure `High resolution timer` is ticked in `Timers subsystem`.
4. Navigate to `Kernel features` in the root of menuconfig, find `Timer frequency` and choose `1000 Hz`.

CPU isolation

> ***Note***: these instructions are for Dual-core ARM processors.

To isolate the application from kernel interruptions, some changes are necessary in default kernel boot arguments. To change those, issue

```
petalinux-config
```

1. Navigate to `DTG Settings` -> `Kernel bootargs` and copy autogenerated bootargs (Highlight + ctrl + shift + c).

[9] https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/5.10/older/patch-5.10-rc7-rt16.patch.gz
[10] https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/

2. Turn off **Generate** bootargs automatically
3. Paste the copied bootargs and add `isolcpus=1 NOHZ_FULL`

This isolated the CPU #1 from the kernel. NOHZ_FULL makes the application non-preemtible, but needs to utilize SCHED_FIFO for this option to work.

The software that this guide would cover as the next step utilizes these features.

Checking board's build (Launching)

Place files on the SD card and power up the board. Connect to the serial to see the bootlog.

After booting, issue

```
uname -a
```

The output should be similar to the following: `Linux cora 5.10.0-rt16-xilinx-v2021.2 #1 SMP PREEMPT_RT`

If that happened, the realtime patch was successfull.

Check the presense and the contents of isolated CPU entries:

```
cat /sys/devices/system/cpu/isolated
```

The output should be 1 (In case isolcpus was set to 1)

Check the dmesg output for kernel module loading messages:

```
dmesg | grep fpga
```

There should be prints from the kernel module.

Appendix A

Listing 10: Kernel module for hardware FPGA interrupts relay

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/platform_device.h>
#include <linux/of.h>
#include <linux/of_device.h>
#include <linux/of_irq.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/signal.h>
#include <linux/fs.h>
#include <linux/sched/signal.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

// The soft OS signal we're going to use to relay the FPGA interrupt
#define SIG_NUM SIGUSR1

#define DRIVER_NAME "fpgatimer"

static struct of_device_id fpgatimer_driver_of_match[] = {
    { .compatible = "homeuser,fpgatimer", },
    {}
};
MODULE_DEVICE_TABLE(of, fpgatimer_driver_of_match);

// Init PID file with 0
static int pid = 0;

// Interrupt handler fetches the contents of 'pid' attribute file
// for the PID where it needs to send the singal. If the PID is 0, it
   doesn't send anything.
// Debug prints are commented out not to spam dmesg.
// Application needs to write 0 into the PID file before exiting.
// If it was killed (PID != 0), don't spam dmesg.
static irqreturn_t fpgatimer_isr(int irq, void *dev_id)
```

```
38 {
39     struct task_struct *task;
40     int ret;
41 
42     // Check if the PID is 0
43     if (pid == 0) {
44         //printk(KERN_INFO "No app, PID == 0, skipping interrupt\n");
45         return IRQ_HANDLED;
46     }
47 
48     /* Find the task associated with the PID */
49     task = pid_task(find_vpid(pid), PIDTYPE_PID);
50     if (!task) {
51         //printk(KERN_ERR "Could not find the task with PID %d\n",
               pid);
52         return IRQ_NONE;
53     }
54 
55     /* Send the signal */
56     ret = send_sig(SIG_NUM, task, 0);
57     if (ret < 0) {
58         printk(KERN_ERR "Error sending signal to application with PID
               %d\n", pid);
59         return IRQ_NONE;
60     }
61 
62     return IRQ_HANDLED;
63 }
64 
65 
66 // Function to show the contents of the attribute file
67 static ssize_t pid_show(struct kobject *kobj, struct kobj_attribute
       *attr,
68                         char *buf)
69 {
70     return sprintf(buf, "%d\n", pid);
71 }
72 
73 // Function to enable write (store) into the attribute file
74 static ssize_t pid_store(struct kobject *kobj, struct kobj_attribute
       *attr,
75                          const char *buf, size_t count)
76 {
```

```
77      int ret;
78
79      ret = kstrtoint(buf, 10, &pid);
80      if (ret < 0)
81          return ret;
82
83      return count;
84  }
85
86  static struct kobj_attribute pid_attribute =
87      __ATTR(pid, 0664, pid_show, pid_store);
88
89  // Find the actual IRQ number and register the interrupt
90  static int fpgatimer_driver_probe(struct platform_device* dev)
91  {
92      printk(KERN_INFO "fpgatimer_driver_probe()\n");
93
94      unsigned int irq;
95      irq = irq_of_parse_and_map(dev->dev.of_node, 0);
96      printk(KERN_INFO "found matching irq = %d\n", irq);
97      if (request_irq(irq, fpgatimer_isr, 0, DRIVER_NAME, &dev->dev))
98          return -1;
99      printk(KERN_INFO "registered irq\n");
100
101     return 0;
102 }
103
104 // Unregister the interrupt upon removal
105 static int fpgatimer_driver_remove(struct platform_device* dev)
106 {
107     printk(KERN_INFO "fpgatimer_driver_remove()\n");
108
109     free_irq(of_irq_get(dev->dev.of_node, 0), &dev->dev);
110
111     return 0;
112 }
113
114 // Driver's struct
115 static struct platform_driver fpgatimer_driver = {
116     .probe = fpgatimer_driver_probe,
117     .remove = fpgatimer_driver_remove,
118     .driver = {
119         .name = DRIVER_NAME,
```

```
120         .owner = THIS_MODULE,
121         .of_match_table = fpgatimer_driver_of_match,
122     },
123 };
124 
125 
126 // kobject
127 static struct kobject *fpgatimer_kobj;
128 
129 // Module init function
130 static int __init fpgatimer_init(void)
131 {
132     printk(KERN_INFO "fpgatimer_init()\n");
133 
134     int retval;
135 
136     // Create a kobject and add it to the sysfs
137     fpgatimer_kobj = kobject_create_and_add(DRIVER_NAME, kernel_kobj);
138     if (!fpgatimer_kobj) {
139         printk(KERN_WARNING "failed to create kobject\n");
140         return -ENOMEM;
141     }
142 
143     // Create the 'pid' attribute file
144     retval = sysfs_create_file(fpgatimer_kobj, &pid_attribute.attr);
145     if (retval) {
146         printk(KERN_WARNING "failed to create sysfs file\n");
147         kobject_put(fpgatimer_kobj);
148         return retval;
149     }
150 
151     // register platform driver
152     if (platform_driver_register(&fpgatimer_driver)) {
153         printk(KERN_WARNING "failed to register platform driver
                \"%s\"\n", DRIVER_NAME);
154         return -1;
155     }
156     printk(KERN_INFO "registered platform driver\n");
157 
158     return 0;
159 }
160 
161 // Stop routine
```

```
162 static void __exit fpgatimer_exit(void)
163 {
164     printk(KERN_INFO "fpgatimer stopped\n");
165     platform_driver_unregister(&fpgatimer_driver);
166     sysfs_remove_file(fpgatimer_kobj, &pid_attribute.attr);
167     kobject_put(fpgatimer_kobj);
168 }
169 
170 module_init(fpgatimer_init);
171 module_exit(fpgatimer_exit);
172 
173 MODULE_AUTHOR ("Dmitrii Matafonov");
174 MODULE_DESCRIPTION("FPGA hard interrupt to userspace relay");
175 MODULE_LICENSE("GPL v2");
176 MODULE_ALIAS("custom:fpga-timer");
```