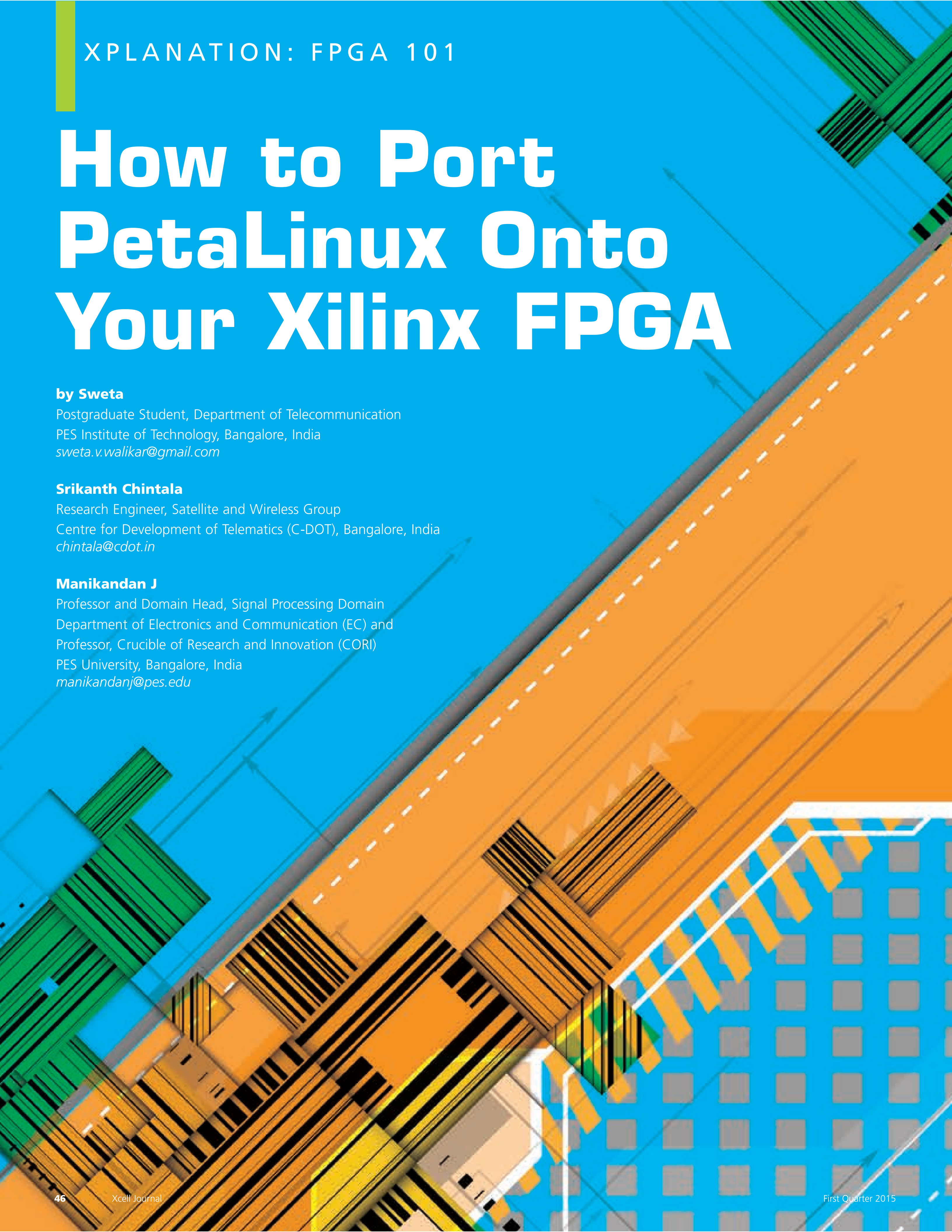
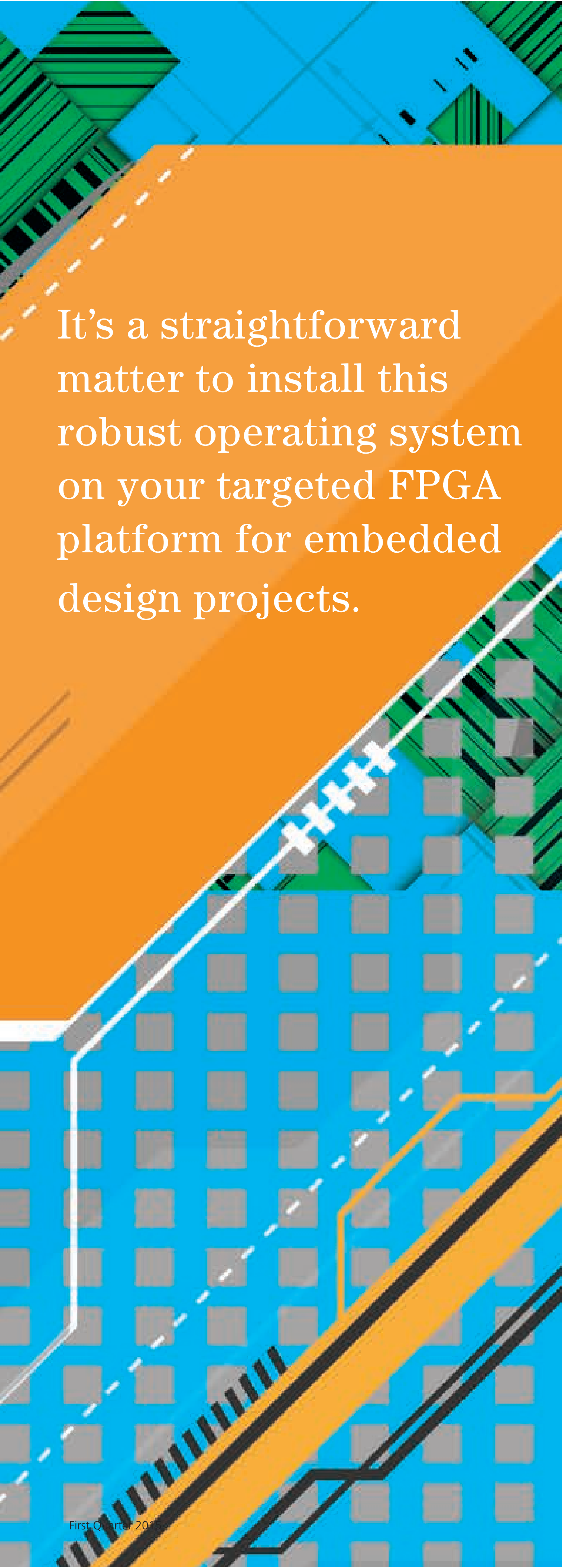
How to Port PetaLinux Onto Your Xilinx FPGA

by Sweta
Postgraduate Student, Department of Telecommunication
PES Institute of Technology, Bangalore, India
sweta.v.walikar@gmail.com

Srikanth Chintala
Research Engineer, Satellite and Wireless Group
Centre for Development of Telematics (C-DOT), Bangalore, India
chintala@cdot.in

Manikandan J
Professor and Domain Head, Signal Processing Domain
Department of Electronics and Communication (EC) and
Professor, Crucible of Research and Innovation (CORI)
PES University, Bangalore, India
manikandanj@pes.edu





It's a straightforward matter to install this robust operating system on your targeted FPGA platform for embedded design projects.

FPGAs have come a long way from their humble beginnings as glue logic. The logic capacity and flexibility of today's FPGAs have catapulted them into a central position in embedded designs. Today, a complete system fits on a single programmable chip, an architecture that facilitates hardware/software co-design and integrates hardware with software applications.

These kinds of FPGA-based embedded designs need a robust operating system. PetaLinux has emerged as a favorite among embedded designers. It is available free of cost as open source and also supports various processor architectures, such as the Xilinx® MicroBlaze® CPU as well as ARM® processors. In order to port PetaLinux onto a particular FPGA, the kernel source code, boot loader, device tree and root file system must be customized, configured and built for the targeted platform.

For a design project here at PES University and C-DOT, our team set out to port PetaLinux and run several PetaLinux user applications on Xilinx's KC705 evaluation board, which features a Kintex®-7 XC7K325T FPGA. It turned out to be a fairly straightforward process.

WHY CHOOSE PETALINUX?

Before going into the details of how we did it, it's worth taking a moment to consider the various OS options available for FPGA-based embedded systems. PetaLinux is one of the most commonly used OSes on FPGAs, along with µClinux and Xilkernel. µClinux is a Linux distribution or ported Linux OS that includes a small Linux kernel and is designed for a processor that does not have a memory-management unit (MMU) [1]. µClinux comes with libraries, applications and tool chains. Xilkernel, for its part, is a small, robust and modular kernel that allows a higher degree of customization than µClinux, enabling users to tailor the kernel to optimize their design in terms of size and functionality [2].

PetaLinux, meanwhile, is a complete Linux distribution and development environment targeting FPGA-based system-on-chip (SoC) designs. PetaLinux consists of preconfigured binary bootable images; fully customizable Linux for Xilinx devices; and an accompanying PetaLinux software development kit (SDK) [3] that includes tools and utilities to automate complex tasks across configuration, build and deployment. The PetaLinux development package, available free of cost and downloadable from Xilinx, includes hardware reference projects designed for various Xilinx FPGA development kits. Also included are a kernel configuration utility for Xilinx FPGAs, software tools such as a cross-compiler, a hardware design creation tool and many more design aids.

It has been reported that Xilkernel performs better than µClinux [4] and that PetaLinux outperforms Xilkernel [5]. For that reason, we chose PetaLinux for our project, especially since the packages were readily available for our Xilinx target board. Another advantage of porting PetaLinux is that the user can have the facility of remote programming. That means you can load the FPGA target board with a new configuration file (or bitstream file) through Telnet using remote access.

There are two approaches to creating a software platform for building a PetaLinux system: PetaLinux commands on a Linux terminal or a GUI with a pulldown menu.

BEGINNING THE INSTALLATION

Let's take a detailed look at how our team installed PetaLinux. For the first step, we downloaded the PetaLinux package 12.12 and the board support package (BSP) for the Kintex-7 target board. We ran the PetaLinux SDK installer and installed the same into the */opt/Petalinux-v12.12-final* directory using the following commands in the console:

```
@ cd /opt
@ cd /opt/PetaLinux-v12.12-final-full.tar.gz
@ tar zxf  PetaLinux-v12.12-final-full.tar.gz
```

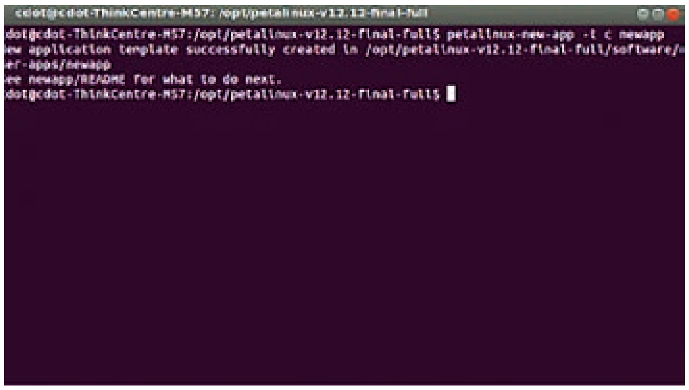
We then copied and pasted the PetaLinux SDK license obtained from the Xilinx website into the .xilinx and .Petalogix folders. Next, we set the SDK working environment by sourcing the appropriate settings using the following commands:

```
@ cd /opt/PetaLinux-v12.12-final
@ source settings.sh
```

In order to verify whether the working environment was set or not, we used the following command:

```
@ echo $PETALINUX
```

If the environment is set properly, the path where PetaLinux is installed will be displayed. In our case, the path where PetaLinux was installed was */opt/PetaLinux-v12.12-final*.



```
cdot@cdot-ThinkCentre-M57: /opt/petalinux-v12.12-final-full
dot@cdot-ThinkCentre-M57:/opt/petalinux-v12.12-final-full$ petalinux-new-app -t c newapp
ew application template successfully created in /opt/petalinux-v12.12-final-full/software/u
er-apps/newapp
ee newapp/README for what to do next.
dot@cdot-ThinkCentre-M57:/opt/petalinux-v12.12-final-full$
```

Figure 1 – Snapshot of a Linux terminal window for user settings

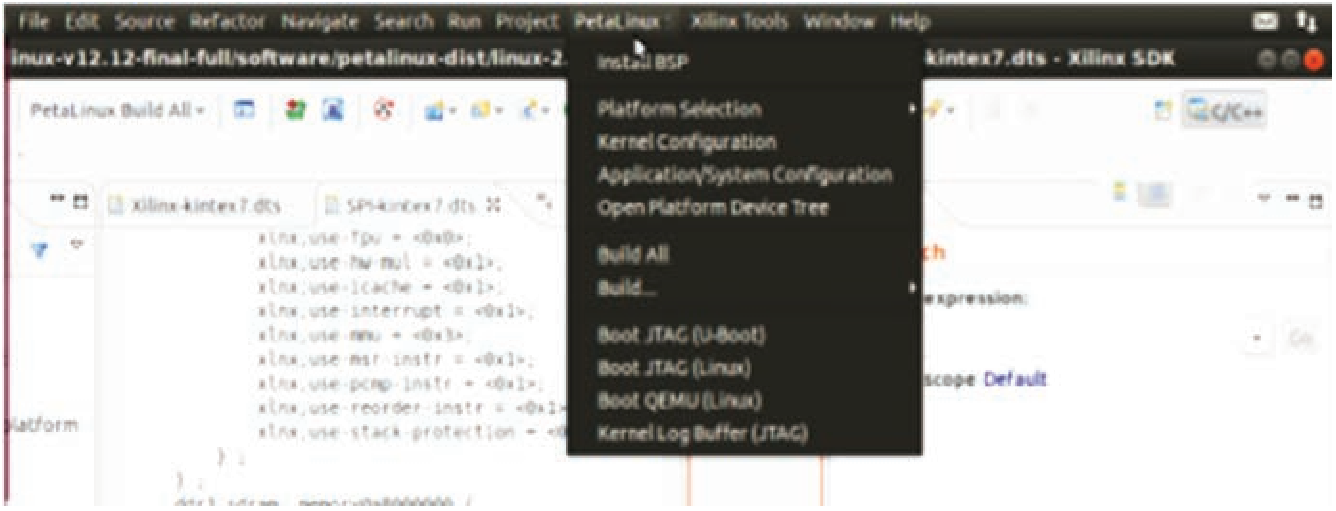
Our next task was to install the BSP, which includes the necessary design files, configuration files and prebuilt hardware and software packages that are already tested and readily available for downloading onto the target board. Packages are also available for booting in the Quick Emulator (QEMU) system simulation environment. In order to install the BSP, we created a folder named "bsp" in the path /opt and copied the ZIP file of the KC705 BSP using the following commands:

```
@ cd  /opt/PetaLinux-v12.12-final-full
@ source settings.sh
@ source /opt/Xilinx/14.4/ISE_DS/settings32.sh
@ PetaLinux-install-bsp /bsp/Xilinx-KC705
    -v12.12-final.bsp
```

There are two approaches to creating and configuring a software platform for building a PetaLinux system customized to a new hardware platform. One method is to use PetaLinux commands in their corresponding path locations using a Linux terminal, as shown in Figure 1. The second approach is to use a GUI with a pulldown menu, as shown in Figure 2. You can use either of these approaches to select the platform, configure the Linux kernel, configure the user application and build images. The PetaLinux console is available once the OS is installed, whereas the GUI is available after installing the PetaLinux SDK plug-in. Once you've installed the plug-in, you can set the configurations using the PetaLinux GUI found in the PetaLinux Eclipse SDK (Figure 2). The GUI has features such as user application and library development as well as debugging, building and configuring PetaLinux and the hardware platform.

BUILDING THE HARDWARE

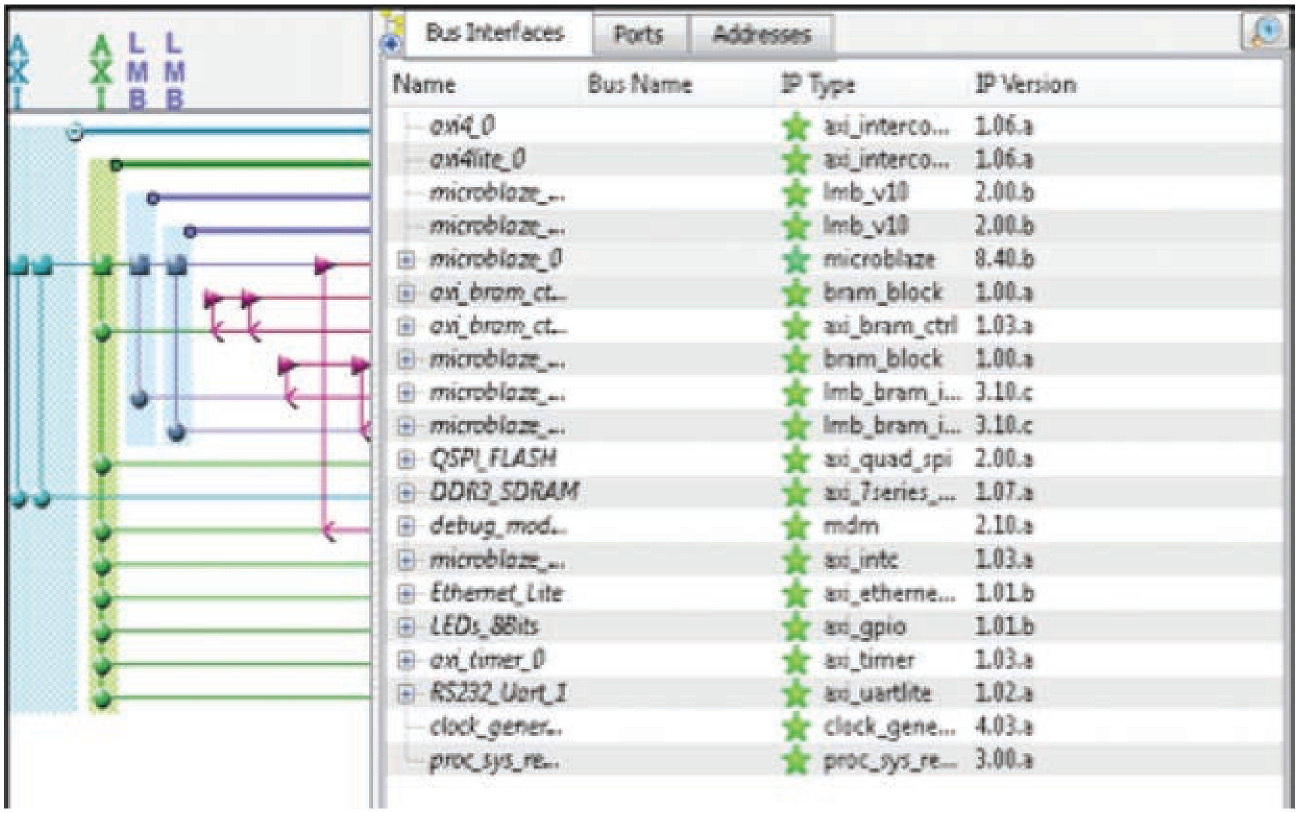
We used the Kintex-7 FPGA-based KC705 evaluation board for our project. The hardware interfaces required for the design included an RS232 interface to monitor the output, a JTAG interface to program the FPGA and an Ethernet interface for remote programming. Besides the PetaLinux SDK, other software required for the proposed design included Xilinx Platform Studio (XPS) [6,7] and the Xilinx Software Development Kit (SDK) [7].



```
xlnx,use-fpu = <0x0>;
xlnx,use-hw-mul = <0x1>;
xlnx,use-icache = <0x1>;
xlnx,use-interrupt = <0x1>;
xlnx,use-mmu = <0x3>;
xlnx,use-msr-instr = <0x1>;
xlnx,use-pcmp-instr = <0x1>;
xlnx,use-reorder-instr = <0x1>
xlnx,use-stack-protection = <0
} ;
} ;
```

Figure 2 – Snapshot of PetaLinux SDK menu for user settings

For the hardware portion of the embedded design, our first task was to design a MicroBlaze processor-based hardware platform using the Base System Builder (BSB) in XPS. The BSB allows you to select a set of peripherals available on the target board. You can add or remove the peripherals based on the demands of the application. The set of cores or peripherals employed for our proposed application included an external memory controller with 8 Mbytes of memory, a timer enabled with interrupts, an RS232 UART with a baud rate of 115,200 bps, Ethernet, nonvolatile memory and LEDs. Once we made our selections, we obtained the hardware peripherals along with their bus interfaces (Figure 3). For designs based on the MicroBlaze processor, PetaLinux requires an MMU-enabled CPU. Hence, we selected low-end



Name	Bus Name	IP Type	IP Version
axi4_0		axi_interco...	1.06.a
axi4lite_0		axi_interco...	1.06.a
microblaze_...		lmb_v10	2.00.b
microblaze_...		lmb_v10	2.00.b
microblaze_0		microblaze	8.40.b
axi_bram_ct...		bram_block	1.00.a
axi_bram_ct...		axi_bram_ctrl	1.03.a
microblaze_...		bram_block	1.00.a
microblaze_...		lmb_bram_i...	3.10.c
microblaze_...		lmb_bram_i...	3.10.c
QSPI_FLASH		axi_quad_spi	2.00.a
DDR3_SDRAM		axi_7series_...	1.07.a
debug_mod...		mdm	2.10.a
microblaze_...		axi_intc	1.03.a
Ethernet_Lite		axi_etherne...	1.01.b
LEDs_8Bits		axi_gpio	1.01.b
axi_timer_0		axi_timer	1.03.a
RS232_Uart_1		axi_uartlite	1.02.a
clock_gener...		clock_gene...	4.03.a
proc_sys_re...		proc_sys_re...	3.00.a

Figure 3 – Hardware configuration of the FPGA

At this point we had the hardware design completed. We could now use the first-stage boot loader to bring up the kernel.

Linux with an MMU by double-clicking on the microblaze_0 instance in the XPS window.

Next, we converted the hardware configuration into a bitstream using a three-step conversion process. First, we used XPS to generate a netlist that represented the embedded hardware platform. Second, we mapped the design into FPGA logic. Finally, we converted the implemented design into a bitstream that could be downloaded onto the FPGA. The final output of XPS was s*ystem.bit and system_bd.bmm* files.

Once we had generated the bitstream, we exported the hardware platform description to the SDK so as to observe the targeted hardware platform in the SDK. The exported system.xml file consisted of information the SDK required to write application software and debug it on the targeted hardware platform. Our next task was to add a PetaLinux repository in the SDK using *Xilinx Tools → Repository → New* and then select the path where PetaLinux was installed. In our case, the path was *$PetaLinux/Hardware/edk_user_repository*.

Next, we created a PetaLinux BSP using *File → Board support package → PetaLinux*. We configured the PetaLinux BSP by selecting necessary drivers based on the application required. Then we built the BSP and created and configured the first-stage boot loader application (fs-boot) to bring up the kernel. The BSP establishes interaction between the hardware and boot application. The output of the SDK is *fs-boot.elf*. A data-to-memory converter command *data2mem* is available that merges *system.bit*, *system_bd.bmm and fs-boot.elf* into a single bitstream file called *download.bit*, which serves as the final FPGA bitstream.

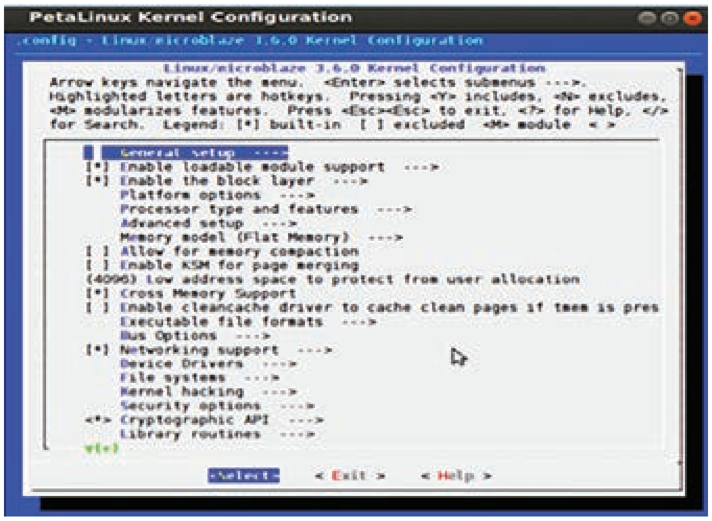
At this point we had the hardware design completed, which among other things included a MicroBlaze core with the PetaLinux OS running on it. We could now use the first-stage boot loader application to bring up the kernel.

BUILDING THE SOFTWARE

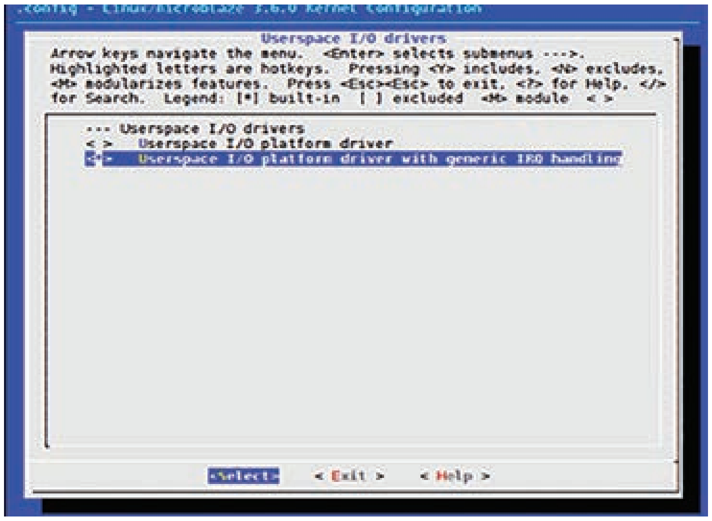
Once our hardware platform was built, we created a customized PetaLinux software platform targeted to the hardware using the following commands:

```
$ cd/opt/PetaLinuxv12.12
$  PetaLinux-new-platform -c <CPU-ARCH> -v
   <VENDOR> -p <PLATFORM>
```

where –c <cpu-arch> is the supported CPU type (here, the MicroBlaze processor), –v <vendor> is the vendor name (here, Xilinx) and –p <platform> is the product name (here, the KC705). The configuration files of the software platform are generated in the directory where PetaLinux is installed, namely */opt/PetaLinuxv12.12/software/ PetaLinux-dist/vendors/Xilinx/ KC705*.



```
PetaLinux Kernel Configuration
.config - Linux/microblaze 3.6.0 Kernel Configuration
                Linux/microblaze 3.6.0 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects 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  [ ] excluded  <M> module  < >

        General setup  --->
    [*] Enable loadable module support  --->
    [*] Enable the block layer  --->
        Platform options  --->
        Processor type and features  --->
        Advanced setup  --->
        Memory model (Flat Memory)  --->
    [ ] Allow for memory compaction
    [ ] Enable KSM for page merging
    (4096) Low address space to protect from user allocation
    [*] Cross Memory Support
    [ ] Enable cleancache driver to cache clean pages if tmem is pres
        Executable file formats  --->
        Bus Options  --->
    [*] Networking support  --->
        Device Drivers  --->
        File systems  --->
        Kernel hacking  --->
        Security options  --->
    <*> Cryptographic API  --->
        Library routines  --->
    v(+)

            <Select>    < Exit >    < Help >
```



```
.config - Linux/microblaze 3.6.0 Kernel Configuration
                    Userspace I/O drivers
  Arrow keys navigate the menu.  <Enter> selects 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  [ ] excluded  <M> module  < >

    --- Userspace I/O drivers
    < >   Userspace I/O platform driver
    <*>   Userspace I/O platform driver with generic IRQ handling

            <Select>    < Exit >    < Help >
```

Figure 4 – The kernel configuration menu

To customize the software platform template to match the hardware, we merged the existing platform configuration with the kernel configuration using the command *PetaLinux-copy-autoconfig*. This command generates the hardware configuration files *Xilinx-KC705.dts*, *xparameters.h* and *config.mk*.

We configured the Linux kernel by opening the kernel configuration menu using the GUI (PetaLinux SDK → Kernel Configuration). You can also do it using the following commands in the Linux terminal:

```
$ cd /opt/PetaLinux_v12.12
$  PetaLinux-config-kernel
```

We enabled the drivers for the application in the kernel configuration pop-up window (shown in Figure 4). In order to access devices through the user-space input/output (UIO) interface for the proposed work, we enabled the UIO driver in the kernel configuration menu.

After configuring the kernel, we designed some applications. PetaLinux provides user application templates for C and C++ programming [8]. These templates include application source code and Makefiles, so it was easy to configure and compile applications for the targeted chip and install them into the root file system. You can create a new PetaLinux user application either by using the GUI (File → PetaLinux New Application) or by typing the following commands into the Linux terminal:

```
$ cd /opt/PetaLinux_v12.12
$  PetaLinux-config-apps
```

We then provided a file name to the user application. In our case, we created *gpio-dev-mem-test* and *gpio-uio-test* user applications and modified the template source code based on the application requirements.

Next, we built the PetaLinux system image by using the GUI (as shown in Figure 2). You can also do it by using the *make* command in the Linux terminal, as follows:

```
$ cd $PETALINUX/software/ PetaLinux-dist
$ make
```

Now the software platform with OS and customized user application is ready to be used, along with the hardware design we've already discussed.

TESTING PETALINUX RUNNING ON THE DEVICE

Here's how PetaLinux boots up. The MicroBlaze processor executes the code residing in Block RAM. The first-stage boot loader (*fs-boot*) will initialize basic hardware, execute *fs-boot.elf* and search for the the Universal Bootloader, or U-Boot, address in a flash partition, as the address of U-Boot is specified while configuring *fs-boot*. The *fs-boot* will then fetch the U-boot image from the U-Boot partition in flash, send it to the device's DDR3 memory and run the kernel. Once you have built all the images required for booting, you can test them on hardware via JTAG, Ethernet or the Quick Emulator. QEMU is an emulator and a virtual machine that allows you to run the PetaLinux OS [9]. Let's look at booting methods for all three solutions.

JTAG is the traditional method for programming and testing FPGA designs. To program the FPGA using the JTAG, we used the pulldown menu "Xilinx Tool → Program the FPGA" and downloaded the *download.bit* file that we generated earlier. Then we downloaded the image onto the board using the GUI (PetaLinux SDK → BOOT JTAG [Linux]), as shown in Figure 2. You can also use the following commands in the Linux terminal:

```
$ cd/opt/PetaLinux_v12.12/software/
  PetaLinux-dist
$  PetaLinux-jtag-boot -i images/image.elf
```

Alternatively, you can perform an indirect kernel boot using U-Boot to boot PetaLinux. The system is first bootstrapped by downloading U-Boot via the JTAG interface using either the GUI (PetaLinux SDK → BOOT JTAG [U-Boot]) or the following commands:

```
$ cd $PETALINUX/software/ PetaLinux-dist
$  PetaLinux-jtag-boot -i images/u-boot.elf
```

Figure 6 shows a snapshot of the U-Boot console.

It's worth noting that the FPGA board is connected to the Ethernet interface. You must select the Ethernet interface in the hardware resources part of the XPS. Once U-Boot boots, check whether the IP address of the server and host are the same. If they are not, set the IP of the host using the following commands in the U-Boot terminal:

```
u-boot>print serverip // prints 192.168.25.45(server ip)
u-boot>print ipaddr   // prints IP address
of the board as   // 192.168.25.68
u-boot>set serverip <HOST IP> // Host IP 192.168.25.68
u-boot>set serverip 192.168.25.68
```

Now the server (PC) as well as the host (KC705 board) have the same IP address. Run the netboot command from the server to download the PetaLinux image and boot:

```
u-boot> run netboot
```

After running netboot, you should see the PetaLinux console, as seen in Figure 5.

Last but not least, you can perform kernel boot by means of QEMU by using either the GUI (PetaLinux SDK → BOOT QEMU [Linux]) or the following commands:

```
$ cd $ PETALINUX/software/ PetaLinux-dist
$ PetaLinux-qemu-boot -i images/image.elf
```

Using this fast method produces the screen shown in Figure 7.

TESTING APPLICATIONS RUNNING ON THE DESIGN

Once the booting of PetaLinux is tested, the next task is to test the user application designed for PetaLinux. The MicroBlaze processor looks at the hardware peripherals on the Kintex-7 FPGA board as a set of memory registers. Each register has its own base address and end address. In order to access any peripheral, the user must know its base and end addresses. You will find details about the addresses in the device tree source (*.*dts*) file. For our design, we developed and tested four applications: Accessing DDR3; Accessing GPIO Using /dev/mem; Accessing GPIO Using UIO; and File Transfer.

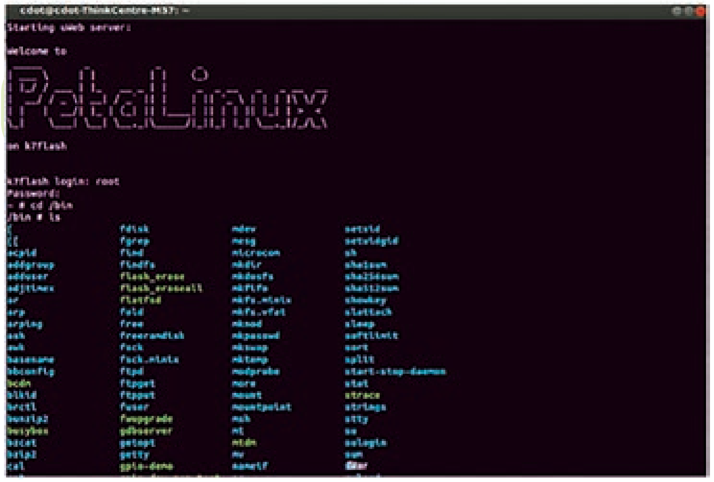

```
cdot@cdot-ThinkCentre-M57: ~
Starting uWeb server:

Welcome to
PetaLinux

on k7flash


k7flash login: root
Password:
~ # cd /bin
/bin # ls
[                  fdisk              mdev               setsid
[[                 fgrep              mesg               setuidgid
acpid              find               microcom           sh
addgroup           findfs             mkdir              sha1sum
adduser            flash_erase        mkdosfs            sha256sum
adjtimex           flash_eraseall     mkfifo             sha512sum
ar                 flatfsd            mkfs.minix         showkey
arp                fold               mkfs.vfat          slattach
arping             free               mknod              sleep
ash                freeramdisk        mkpasswd           softlimit
awk                fsck               mkswap             sort
basename           fsck.minix         mktemp             split
bbconfig           ftpd               modprobe           start-stop-daemon
bcdn               ftpget             more               stat
blkid              ftpput             mount              strace
brctl              fuser              mountpoint         strings
bunzip2            fwupgrade          msh                stty
busybox            gdbserver          mt                 su
bzcat              getopt             mtdm               sulogin
bzip2              getty              mv                 sum
cal                gpio-demo          nameif             tar
```

Figure 5 – Snapshot of PetaLinux console confirming that the OS has booted

1. Accessing DDR3

We used the PetaLinux application titled *DDR3-test.c* to access the DDR3 memory. The application is designed to write data to and read data from a DDR3 memory location. DDR3 is a dual-in-line memory module that provides SDRAM for storing user code and data. As mentioned earlier, the user should know the start and end addresses of DDR3 memory—*0xC0000000* and *0xC7FFFFFF* respectively. The memory size is 512 Mbytes. The Linux kernel resides in the initial memory locations of DDR3 memory. Hence, the writing location for DDR3 memory is selected in such a way that the Linux kernel is not corrupted. The command we used to write data to DDR3 memory was

```
#DDR3-test -g 0xc7000000 -o 15
```

where DDR3-test is the application name, –g is the DDR3 memory physical address, –o is output and 15 is the value expected to be written on the DDR3 memory at the location 0xc7000000. To test whether the value is written at the expected location, we used the following command to read data from DDR3 memory:

```
#DDR3-test -g 0xc7000000 -i
```

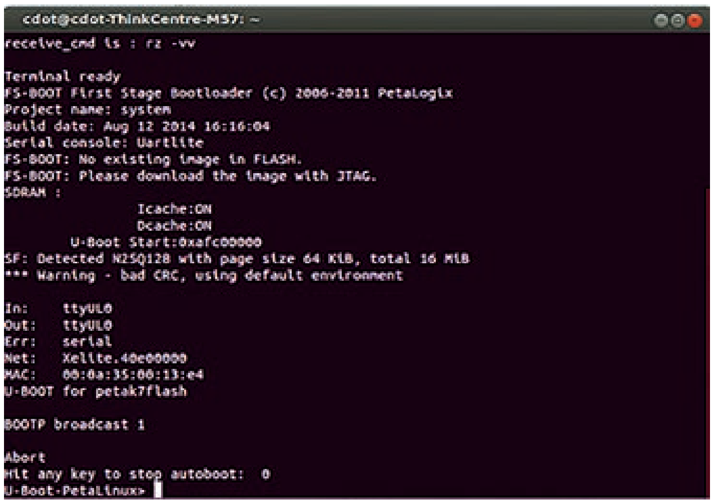

```
cdot@cdot-ThinkCentre-M57: ~
receive_cmd is : rz -vv

Terminal ready
FS-BOOT First Stage Bootloader (c) 2006-2011 PetaLogix
Project name: system
Build date: Aug 12 2014 16:16:04
Serial console: Uartlite
FS-BOOT: No existing image in FLASH.
FS-BOOT: Please download the image with JTAG.
SDRAM :
                Icache:ON
                Dcache:ON
        U-Boot Start:0xafc00000
SF: Detected N25Q128 with page size 64 KiB, total 16 MiB
*** Warning - bad CRC, using default environment

In:    ttyUL0
Out:   ttyUL0
Err:   serial
Net:   Xellite.40e00000
MAC:   00:0a:35:00:13:e4
U-BOOT for petak7flash

BOOTP broadcast 1

Abort
Hit any key to stop autoboot:  0
U-Boot-PetaLinux>
```

Figure 6 – Indirect kernel boot via Universal Bootloader (U-Boot)

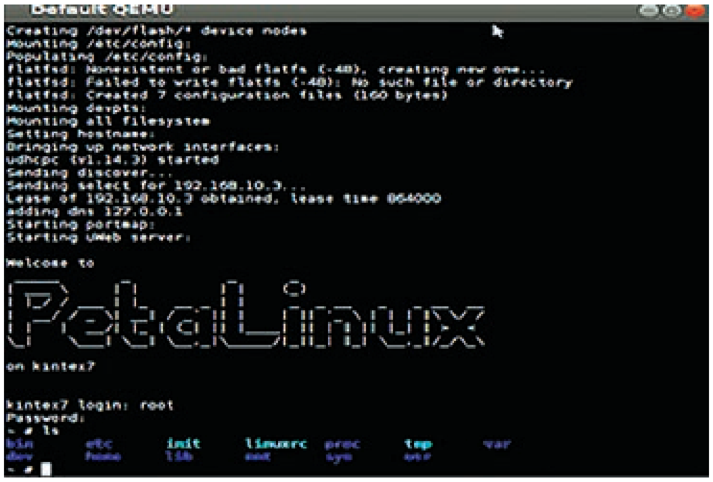

```
Default QEMU
Creating /dev/flash/* device nodes
Mounting /etc/config:
Populating /etc/config:
flatfsd: Nonexistent or bad flatfs (-48), creating new one...
flatfsd: Failed to write flatfs (-48): No such file or directory
flatfsd: Created 7 configuration files (160 bytes)
Mounting devpts:
Mounting all filesystem
Setting hostname:
Bringing up network interfaces:
udhcpc (v1.14.3) started
Sending discover...
Sending select for 192.168.10.3...
Lease of 192.168.10.3 obtained, lease time 864000
adding dns 127.0.0.1
Starting portmap:
Starting UWeb server:

Welcome to
PetaLinux

on kintex7


kintex7 login: root
Password:
~ # ls
bin       etc       init      linuxrc   proc      tmp       var
dev       home      lib       mnt       sys       usr
~ #
```

Figure 7 – Running PetaLinux through QEMU

The application is designed to control an 8-bit discrete output and is tested by connecting LEDs onboard to the GPIO.

The value 15 was observed in the terminal, which confirms the DDR3 memory read and write operations were operating perfectly.

2. Accessing GPIO Using /dev/mem

For our next application test, we used a PetaLinux application titled *gpio-dev-mem-test.c* to access general-purpose I/O (GPIO). The application is designed to control an 8-bit discrete output and test that output by connecting LEDs onboard to the GPIO. In order to access any device from the user space, open */dev/mem* and then use *mmap()* to map the device to memory. The start and end addresses of the LED GPIO we used are *0x40000000* and *0x4fffffff*, respectively.

The GPIO peripheral has two registers: a data register (GPIO_DATA) and a direction register (GPIO_TRI_OFFSET). In order to read the status of the GPIO, we set the direction bit to 1 (i.e., GPIO_TRI_OFFSET=1) and read the data from the data register. To write data to GPIO, set the bit to 0 and write the value to the data register. Data is written on GPIO using the following command on the PetaLinux terminal:

```
#gpio-dev-mem-test –g 0x40000000 –o 255
```

where *gpio-dev-mem-test* is the application name, *–g* is the GPIO physical address, *–o* is output and 255 is the value transmitted from GPIO, which is connected to LEDs. The results of the test were verified when the LEDs lit up as programmed.

3. Accessing GPIO Using UIO

An alternative way of accessing GPIO is via the user-space input/output. We used a PetaLinux application titled *gpio-uio-test.c* to access the GPIO using UIO. The application is designed to control an 8-bit discrete output and is tested by connecting LEDs onboard to the GPIO. A UIO device is represented as */dev/uioX* in the file system. In order to access GPIO through UIO, we opened */dev/uioX* or *sys/class/uio/ui0* and then used the *mmap()* call. We configured the kernel to support UIO and enabled the UIO framework in the kernel. Then, using a parameter called "Compatibility," we set the LEDs' GPIO to be controlled as the UIO device, instead of the normal GPIO device. We also changed the label of the device from *gpio@40000000* to *leds@40000000*.

We then rebuilt PetaLinux and tested the GPIO access using UIO. We obtained details about the information of UIO modules loaded using

```
# ls /sys/class/uio/
 uio0 uio1 uio2
```

The name of the UIO and its address are found in */sys/class/uio/uioX*. We used the following command to access GPIO LED through the UIO driver:

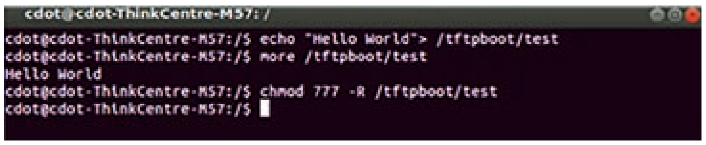
```
# cd "/sys/class/uio/uioX
# gpio-uio-test –d /dev/uio1 –o 255
```

Here, *gpio-uio-test* is the application name, *–d* is the device path, *–o* is the output and 255 is the value passed out to GPIO through UIO. The results were verified by the LEDs glowing based on the data written on GPIO lines using the above command.

4. File Transfer Application

For our last test, we transferred a file from a server to a client, where the server is the host PC and the client is the KC705 board. For this test, we connected the server and client through an Ethernet cable. We used the Trivial File Transfer Protocol (TFTP), which is well known for its simplicity and is generally used for automated transfer of configuration or boot files. In order to test the file transfer from server to client using TFTP, we created a file called *test* in the server PC at */tftpboot*. We used the following commands to write "Hello World" in the file and to view the contents in the same file (as shown in Figure 8):

```
@ echo "Hello World" > /tftpboot/test
@ more /tftpboot/test
```



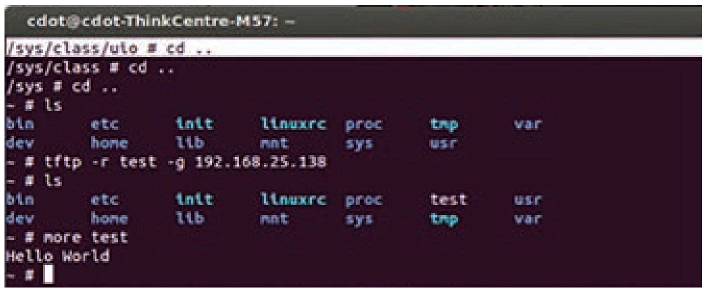
```
cdot@cdot-ThinkCentre-M57: /
cdot@cdot-ThinkCentre-M57:/$ echo "Hello World"> /tftpboot/test
cdot@cdot-ThinkCentre-M57:/$ more /tftpboot/test
Hello World
cdot@cdot-ThinkCentre-M57:/$ chmod 777 -R /tftpboot/test
cdot@cdot-ThinkCentre-M57:/$ █
```

Figure 8 – Snapshot of file creation in the server

To receive this file from the server, we typed the following get command (*-g*) in the PetaLinux terminal window that was running as the client on the KC705 board:

```
# tftp -r test -g 192.168.25.68
# ls -a
```

A new file was created with the filename “test” in the client (as shown in Figure 9). We can view the contents of this file using the *more* command, as seen in Figure 9.

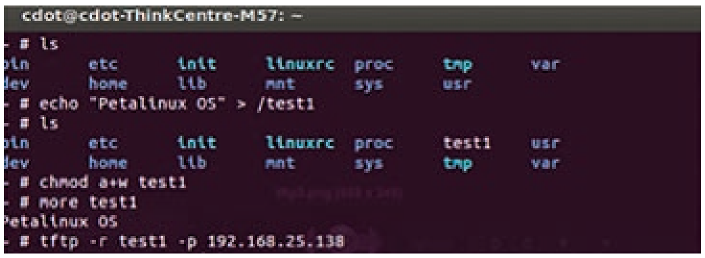


```
cdot@cdot-ThinkCentre-M57: ~
/sys/class/uio # cd ..
/sys/class # cd ..
/sys # cd ..
~ # ls
bin       etc       init      linuxrc  proc      tmp       var
dev       home      lib       mnt      sys       usr
~ # tftp -r test -g 192.168.25.138
~ # ls
bin       etc       init      linuxrc  proc      test      usr
dev       home      lib       mnt      sys       tmp       var
~ # more test
Hello World
~ # █
```

Figure 9 – Snapshot for file reception in the client

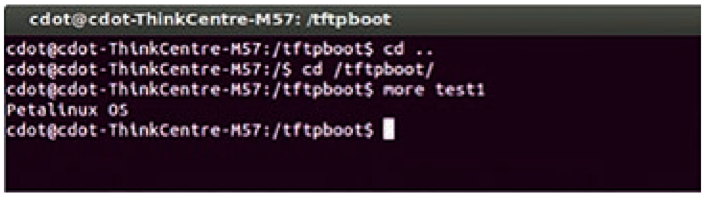
Similarly, transferring a file from the client to the server is done by creating a file called *test1* with the content “PetaLinux OS” in the client machine. To transmit the file from the client to the server, use the following “put” command (*-p*) in the PetaLinux terminal running from the client (as shown in Figure 10):

```
# tftp -r test1 -p 192.168.25.68
```



```
cdot@cdot-ThinkCentre-M57: ~
~ # ls
bin       etc       init      linuxrc  proc      tmp       var
dev       home      lib       mnt      sys       usr
~ # echo "Petalinux OS" > /test1
~ # ls
bin       etc       init      linuxrc  proc      test1     usr
dev       home      lib       mnt      sys       tmp       var
~ # chmod a+w test1
~ # more test1
Petalinux OS
~ # tftp -r test1 -p 192.168.25.138
```

Figure 10 – Snapshot for file transmission from client to server



```
cdot@cdot-ThinkCentre-M57: /tftpboot
cdot@cdot-ThinkCentre-M57:/tftpboot$ cd ..
cdot@cdot-ThinkCentre-M57:/$ cd /tftpboot/
cdot@cdot-ThinkCentre-M57:/tftpboot$ more test1
Petalinux OS
cdot@cdot-ThinkCentre-M57:/tftpboot$ █
```

Figure 11 – Snapshot of file reception in the server

A blank *test1* file is created in the server. Its contents are read after the file transfer operation, and the contents are verified as shown in Figure 11.

Implementing an embedded system and running PetaLinux on an FPGA were pretty straightforward operations. Next, we plan to implement a design using remote programming where the boot files are transferred via Ethernet and the client is capable of running a new application.

REFERENCES

1. Kynan Fraser, “MicroBlaze Running uClinux,” Advanced Computer Architecture from *http://www.cse.unsw.edu.au/~cs4211*

2. Xilkernel from Xilinx Inc., Version 3.0, December 2006

3. PetaLinux SDK User Guide from Xilinx Inc., UG976, April 2013

4. Gokhan Ugurel and Cuneyt F. Bazlamacci, “Context Switching Time and Memory Footprint Comparison of Xilkernel and µC/OS-II on MicroBlaze,” 7th International Conference on Electrical and Electronics Engineering, December 2011, Bursa, Turkey, pp.52-55

5. Chenxin Zhang, Kleves Lamaj, Monthadar Al Jaberi and Praveen Mayakar, “Damn Small Network Attached Storage (NAS),” Project Report, Advanced Embedded Systems Course, Lunds Tekniska Hogskola, November 2008

6. Platform Studio User Guide from Xilinx Inc., UG113, Version 1.0, March 2004

7. “EDK Concepts, Tools and Techniques: A Hands-On Guide to Effective Embedded System Design,” Xilinx Inc., UG683, Version 14.1, April 2012

8. PetaLinux Application Development Guide from Xilinx Inc., UG981, April 2013

9. PetaLinux QEMU Simulation Guide from Xilinx Inc., UG982, November 2013