Operációs Rendszerek

AMP Laboratórium

VIMIAB00

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with you using PREEMPT_RT."

Linus Torvalds

Bajczi Levente

AMP on Consumer SoCs

A System on a Chip (SoC) is a tightly integrated hardware element that contains most of the components necessary for the computer to function. Nowadays these incorporate mostly multi-core processors, as we will see during this laboratory exercise, where we will uncover the reasons why a general-purpose operating system is not the best idea for tasks where either exact timing or low and stable latencies are necessities.
The tasks marked with an asterisk (*) are more advanced and therefore those should only be done if the other tasks are already finished.

Supported Hardware

Theoretically, any multicore Pi supports this version of setting up AMP. These are the following:

- Raspberry Pi 2
- Raspberry Pi 3 *****
- Raspberry Pi 3B+
- Raspberry Pi 3 Model A+

The other versions (The original Raspberry Pi, the RPi Zero and some compute models) have only **one** core, and therefore they are not suitable for this laboratory! An exception to this rule is the Raspberry Pi 4 – it won't be compatible with *all* the steps, due to the modifications the engineers made while developing it. You can follow along, but some steps won't be compatible – there will always be a note stating this incompatibility.
Furthermore, depending on the hardware you have access to, you have two possibilities:

1. If you have *dupont wires* (or any other ordinary wire that can connect two pins of the Pi together) you can go the **measuring route**
2. If you don't have access to the hardware above, you can go the **led-blinking route** (this is *not* available for the models with an asterisk (*) above)

Most of the steps are the same for both of these routes, but the actual goals are different – the former one aims at proving that response times (and system responsiveness) differ between various environments, and the latter provides a way to show how these enviroments are set up.
For all tasks it is expected for the user to have access to a micro SD card and an SD card reader.

Structure of the Laboratory

In the first half of this laboratory exercise, we will prepare the SD images we will use to test the different setups of the Pi. These environments will be the following:

1. "Normal" linux on all 4 cores
2. Real-time linux on all 4 cores
3. Real-time linux on 3 cores, and a custom bare metal program on the 4th core.

In the second half, we will do some experiments on these environments using *stress-tests*, and see for ourselves whether the difficulty of the setup makes a noticeable difference.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Necessary Knowledge

General purpose tools

- SSH - SimpleWiki
- SCP - Wikipedia
- Makefiles - Wikipedia
- Mount - Wikipedia

Specific to this lab exercise (read through these before doing the lab!):

- **AMP -** Wikipedia
- **PREEMPT_RT -** LinuxFoundationWiki
- **Linux Kernel Build System -** Linuxjournal
- **Linux Priorities -** IBM Tutorials

Useful commands

For some of these commands, *sudo* or being root is necessary.

- Make a file executable: `chmod +x <file>`
- Copy a file: `cp -r <source> <dest> # -r is for directories`
- Move a file: `mv <source> <dest>`
- Delete a file: `rm -rf <file(s)> # -r is for directories`
- Find absolute path: `echo $PWD/<relative path of file>`
- Copy a file to the pi: `scp <local-file> <user>@<ip>:<remote-file>`
  Log in to the pi using ssh: `ssh pi@<ip>`
- Launch a program: `./<program>`
  - Modify its 'nice' value: `nice -N ./<program> # for -20: nice -20`[1]
  - Modify its real-time priority: `chrt N ./<program>`[2]
- Extracting a *.tar.xz* file: `tar xvf <tar.xz file>`
- Check system information: `uname -a`
- List devices of a live system: `dtc -I fs /sys/firmware/devicetree/base`
- Print out the system log: `dmesg`
- Clear the system log: `dmesg -C`
- Flash an SD card with an image:
  `dd if=<in> of=<out> bs=4M status=progress conv=sync,noerror`
  - out: */dev/sdc* or something similar
- Stress the CPUs: `stress-ng --cpu N -t Xm # N threads for X minutes`
  - Add *--aggressive* to crank the stress up
  - Add *sudo* to the beginning to make it even more aggressive (might lock up!)

Terminal tips

- Normal crtl+c/v does not work in the terminal. Use ctrl+shift+c/v instead.
- Use *tab* once to complete the current command, twice to see the possibilities.

[1] https://askubuntu.com/a/656787/613336
[2] https://stackoverflow.com/a/52501811/4564311

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Discovering the Cross-Compilation Toolchains

The computer you will be working on most likely contains an X86 (Intel or AMD) processor. If you compile a program using the ordinary `gcc` compiler, the generated machine code will run on any other X86 processor (running the same operating system). For our purposes though, this will not suffice: the Pi contains an ARM CPU, and is therefore incompatible with code compiled for X86. Two solutions exist for this problem: one, you could write all code on the Pi itself, and compile *natively* – or you could use a so called *cross compiler*. This is a collection of binaries that are compiled for a host architecture (in our case, X86/Linux), and targets another (in our case, ARM).

Start the VM, log in (user: *meres*, pwd: *LaborImage*) and start a terminal. Check that the following commands are available:

- `arm-linux-gnueabihf-gcc`
- `arm-none-eabi-gcc`

If everything works correctly, both of these commands return an error message that no source files were provided. So provide them with one:

```
#include <stdio.h>
int main()
{
   printf("Hello World!\n");
   return 0;
}
```

Write this into a file (say *program1.c*) and try to compile it using both toolchains.

Use the 'workspace' directory (visible on the Desktop), as only that has enough space!

Question: Did both toolchains work? Was this expected? Why?

These toolchains' names are exported into the following two environment variables:

- CROSS_LINUX
- CROSS_BARE

Preparing the images

To complete this laboratory exercise, you will need to prepare 3 different images:

1. Normal off-the-shelf Raspbian Lite
2. Raspbian Lite with the PREEMPT_RT patch enabled
3. Raspbian Lite with a modified kernel without access to the 4th core

Download the most up-to-date image from the official website, and unzip it. This is a "virtual SD cards", meaning if you put files on it, and then burn it to an SD card and boot the Pi off of that, the Pi will see those files in its filesystem. However, it is not trivial how to modify these kinds of images. **Using the description on the next page, place an empty and extensionless file called "ssh" on the *boot* partition of the image file!**

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Mount an Image to Configure

To modify an image file, we will mount its partitions using an intermediary loop device (everything with *sudo*!):

1. `losetup -f #This will return with a usable loop device`
2. `losetup /dev/loopX <img> #This will make the image use the device`
3. `fdisk -l /dev/loopX #This will print out the partition info`
    a. You should see something similar:

```
Disk /dev/loop0: 2.21 GiB, 2369781760 bytes, 4628480 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x738a4d67

Device       Boot  Start     End Sectors  Size Id Type
/dev/loop0p1        8192  532479  524288  256M  c W95 FAT32 (LBA)
/dev/loop0p2      532480 4628479 4096000    2G 83 Linux
```

    Make note of the values in the *Start* column and the sector size (most likely 512)! (This is *ss* in the examples)
4. `mkdir /mnt/boot; mkdir /mnt/rootfs`
5. Do one (to modify either the *boot* partition or the *root* partition, but only one at a time):
    a. `mount /dev/loopX /mnt/boot -o offset=$((<ss>*<start1>))`
    b. `mount /dev/loopX /mnt/rootfs -o offset=$((<ss>*<start2>))`
6. `# do the necessary modifications, detailed in the next 2 sections`
7. Do one:
    a. `umount /mnt/boot`
    b. `umount /mnt/rootfs`
8. `losetup -d /dev/loopX # Un-register the device when done`

Always unmount the device before flashing the image to an SD card!

As we will need to have access to a compiled kernel source (and we want to modify a few things), let's replace the current kernel on the SD image with one we built – follow the instructions on the next page to get the sources, compile it and place the built files into the SD image.

After you are done, make three copies of the SD card, and name them the following:

1. **normal.img**
2. **rt.img**
3. **amp.img**

Note: Yes, we will use the RT-patched kernel on the "normal" image as well. The difference between the normal (non-RT) operation of the ordinary and the RT-patched kernel is very small, and it is not worth it to compile two different versions of the kernel. We will just pretend during testing that the "normal" image is not capable of real-time scheduling.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Compile an RT-patched kernel

1. Get the sources: `git clone https://github.com/raspberrypi/linux.git --depth 1 --branch rpi-4.19.y-rt`
2. Disable the LED:
   1. Open one of the following files, depending on your pi's model:
      a. arch/arm/boot/dts/bcm2709-rpi-2-b.dts # RPi 2
      b. arch/arm/boot/dts/bcm2710-rpi-3-b.dts # RPi 3
      c. arch/arm/boot/dts/bcm2710-rpi-3-b-plus.dts # RPi 3A+/B+
      d. arch/arm/boot/dts/bcm2711-rpi-4-b.dts # RPi 4
   2. Delete the following lines:

```
act_led: act {
      label = "led0";
      linux,default-trigger = "mmc0";
      gpios = <&gpio xx 0>;
};
```

```
      act_led_gpio = <&act_led>,"gpios:4";
      act_led_activelow = <&act_led>,"gpios:8";
      act_led_trigger = <&act_led>,"linux,default-trigger";
```

3. Depending on the Pi's model, issue either:
   a. `make bcm2711_defconfig CROSS_COMPILE=$CROSS_LINUX #RPi4`
   b. `make bcm2709_defconfig CROSS_COMPILE=$CROSS_LINUX #other`
4. `make -j4 zImage modules dtbs CROSS_COMPILE=$CROSS_LINUX`

   This will take a long time. Take a short break while it compiles, maybe read up on AMP systems[3] or the PREEMPT_RT patch[4]!
5. Mount the *root* partition of the image file to */mnt/rootfs!*
6. `make modules_install INSTALL_MOD_PATH=/mnt/rootfs CROSS_COMPILE=$CROSS_LINUX`
7. Unmount */mnt/rootfs* and mount the *boot* partition of the image file to */mnt/boot!*
8. Delete the following files in */mnt/boot*: **.dtb, kernel*, overlays/**
9. `cp arch/arm/boot/zImage /mnt/boot/`
10. `cp arch/arm/boot/dts/<name>.dtb /mnt/boot/`

    The <name> possibilities are (choose your Pi's version):
    a. bcm2711-rpi-4-b
    b. bcm2710-rpi-3-b-plus
    c. bcm2709-rpi-2-b
11. `cp arch/arm/boot/dts/overlays/*.dtb* /mnt/boot/overlays/`
12. Put `kernel=zImage` at the end of /mnt/boot/config.txt

[3] https://en.wikipedia.org/wiki/Asymmetric_multiprocessing
[4] https://wiki.linuxfoundation.org/realtime/documentation/start

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Why did we need to disable the LED? Normally the activity LED (yellow LED on the board near the red power LED) displays the "heartbeat" of the system – meaning it is constantly switched on and off based on the load the system is under. However, if you wanted to use it for something else (in our case, the examples are blinking the LED, which is **only** visible if this heartbeat display is disabled), you need to take the control away from the kernel.

To finish up preparing the images, we will need to configure the most special one – *amp.img*. The default bootloader (which runs on the GPU of the Pi) does not respect all kernel parameters that are passed to it. Therefore, we need to insert another 2nd stage bootloader into the boot sequence, which is called *Das U-Boot*[5]. **This is not compatible with the RPi 4! Skip this step.**

1. Clone the repo: `git clone git://git.denx.de/u-boot.git`
2. Configure: `make CROSS_COMPILE=$CROSS_LINUX rpi_2_defconfig #`
3. Build: `make CROSS_COMPILE=$CROSS_LINUX u-boot.bin -j4`
4. Mount the *boot* partition of the image being modified to */mnt/boot*
5. Copy the created *u-boot.bin* file to the boot partition
6. Create a file called *boot.cmd* in the boot partition containing the following code:

```
setenv bootargs earlyprintk console=tty0 console=ttyAM0
root=/dev/mmcblk0p2 rootfstype=ext4 rootwait noinitrd
fatload mmc 0:1 ${fdt_addr_r} bcm2710-rpi-3-b.dtb #your version
fatload mmc 0:1 ${kernel_addr_r} zImage
bootz ${kernel_addr_r} - ${fdt_addr_r}
```

7. Issue `mkimage -C none -A arm -T script -d boot.cmd boot.scr` to compile *boot.cmd* to the u-boot file *boot.scr*.
8. Verify that a *boot.scr* has been created in /mnt/boot
9. Add the following line to *config.txt*: `kernel=u-boot.bin`

If done well, this process has inserted the U-Boot bootloader into the boot process seamlessly, meaning that the same kernel works as before but with an additional step when booting.

To achieve our goal of not having all the cores under linux's scheduler, we have to configure the kernel parameters the bootloader passes to the booting kernel. This is specified in the *bootargs* environment variable in the *boot.cmd* file. A (somewhat complete) list of parameters can be found in the kernel's documentation[6].

Question:

1. **Which parameter has to be set to only allow 3 CPUs in the system?**
   a. *Hint: it is SMP-related.*
2. **The *mem=nn[KMG]* parameter can be used to limit the RAM usage of the system. What is KMG, and how to set it to 512 Megabytes?**
3. **What do the existing parameters mean in the *boot.cmd* file?**

Modify the boot.cmd file (and of course run through steps 7-8 again) in the boot partition. Flash the SD card and boot the pi, then verify that it indeed only has 3 cores and 512M of RAM[7].

[5] https://www.denx.de/wiki/U-Boot
[6] https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html
[7] Use the *htop* CLI program to verify this.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Second part

In the Remote Peripherals Datasheet of the Pi, there is a picture and a short description of the memory mapping of the Raspberry Pi's SoC under section **1.2**.

Question:

1. **At which bus address does the I/O block start?**
2. **Where can we reach it from software running *without* an OS?**
3. **Where can we reach it from software running *on top of* an OS running in kernel mode?**

In the same document, we can find the registers used to configure the General Purpose Input-Output pins of the SoC (GPIO) under section **6**. We know that to use a GPIO we need to do the following tasks:

- Configure the pin as input/output (registers named **GPFSELn**)
- Write to a SET register when using it as an output to pull it **up** (**GPSETn**)
- Write to a CLEAR register to pull it **down** (**GPCLRn**)
- Read a LEVEL register when using it as an input to read the current state. (**GPLEVn**)

Question:

1. **Which register (name and *bus* address) is responsible for selecting the GPIO function of pins #14, #15, #23 and #24? Which bits?**
2. **Which registers are the corresponding SET, CLEAR and LEVEL registers (name and *bus* address)?**

Using *make*, compile the following programs based on the path you are following:

Measuring path	**Led-blinking path**
• userspace-gpio Mirrors pin #15 to #23 • kernel-gpio Mirrors pin #15 to #23 in a kernel module • bare-gpio Mirrors pin #15 to #23 on a separate core	*Don't forget to modify the source files to reflect the model of your Pi!* • userspace-led Blinks the on-board led • kernel-led Blinks the on-board led in a kernel module • bare-led Blinks the on-board led on a separate core
▪ loader Creates a special file /dev/loader which acts as the interface to the separate core	

Some explanations for the code follow.

Megjegyzés: Nem tudom, mi lenne a legjobb módja annak, hogy ezeket a feladatokat kicsit interaktívabban írják meg – teljesen nulláról esélytelen, de még kis lyukakkal is sok idő volt a próba óra alatt.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Mapping Physical Addresses to Virtual Addresses

Even though while running in kernel mode, the I/O is reachable as a simple I/O instruction, user mode is not so lucky. Instead of that, the use of the *mmap* function is necessary. The arguments to be passed are the following:

1. Addr - NULL, because we don't care where it will end up. The kernel knows better.
2. Prot - `PROT_WRITE`
3. Flags - `MAP_SHARED`
4. Fd - Returned file descriptor of the file "**/dev/gpiomem**" using the open system call.
5. Offset - 0, because the file above is aligned to the start of the GPIO block
6. Size - calculate it using the value of the GPIO register at the highest address

Running Bare-Metal Programs on the Pi

Bare-metal in this context means that there is no operating system we could rely on while developing the program we are trying to run. This means that anything we want to achieve has to be done completely from scratch, there is no abstraction of the hardware we could use. As already uncovered in Section 1., not even *printf* has a default implementation - the compiler does not know *how* it could give us an output.

Another difference arises from the memory usage - unless we explicitly say otherwise, the program has direct access to the entirety of the *physical* addresses. This makes it easier to develop applications, but at the same time, makes it a security vulnerability to do so. In our use-case, the cores running Linux will see 512MBytes of the RAM and the rest (however large it is) will belong to the bare-metal program. It could, however, potentially see and even modify the RAM contents of the processes running on top of Linux.

Furthermore, the core is *uninitialized* the first time we make it jump to the program's address, and therefore we need to set up a few things before we could un anything else. To do so, we will need the following file structure:

- **Makefile** - to facilitate compilation
- **main.c** - contains our program's code
- **start.S** - contains necessary code to bootstrap the CPU and provide basic LCM[8]

Starting a Core on the Raspberry Pi

Normally, the boot process takes care of starting all cores a given system has, but in this case, we have specifically made it so that it only has access to 3 of the 4 available cores. We *could* start it from under Linux, but then it would be assigned tasks from the scheduler, and we want to refrain from that. The easiest way of figuring out how to wake up a core is to look at the kernel's source and see how it does it[9]. In this case, the cores are in a *Wait-for-Instruction* (WFI) loop, meaning they are monitoring a register which will contain a memory location to jump to. In the case of the Raspberry Pi, this register is the *Mailbox* of the core, more specifically the 3rd one.

[8] Life Control Management - Describing what should happen before/after the program's end/start
[9] https://github.com/raspberrypi/linux/blob/rpi-4.1.y/arch/arm/mach-bcm2709/bcm2709.c#L1248

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Question: Using the Pi's local peripheral documentation[10], figure out the following:

1. **What is the *SET* address of the 3rd core's 3rd mailbox?**
2. **What is the CLEAR address of the 3rd core's 3rd mailbox?**
3. **Which is readable?**

Linux Kernel Modules

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.
In the scope of this project we will create two kernel modules, as Tasks #2 and #8 outlined.
The Linux Kernel Build System[11] makes the development and compilation of a kernel module very convenient, but we must abide by a few rules:

- The running kernel's source (more specifically, the headers) must be used for compiling the module
- To build the module (from the same directory as the source.c):

```
make obj-m+=<source.c without .c >.o ARCH=arm \
CROSS_COMPILE=$CROSS_LINUX -C <linux-source-root> M=$PWD modules
```

- To clean the project (from the same directory):

```
make -C <linux-source-root> M=$PWD modules
```

This can easily be incorporated into a Makefile of its own:

```
obj-m += kernel-module.o
all:
   make ARCH=arm -C ../../linux M=$(PWD) modules
clean:
   make ARCH=arm -C ../../linux M=$(PWD) clean
```

The generic outline of any kernel module (my_module.c) is the following:

```
#include <linux/module.h> // Necessary include for kernel modules
MODULE_LICENSE("GPL");    // Could be any other license as well
MODULE_AUTHOR("Author Name");         // Your name
MODULE_DESCRIPTION("Short description"); // Concise summary
MODULE_VERSION("0.01");               // Version number
static int __init lkm_example_init(void) // This will run when loaded
{ printk(KERN_INFO "Hello World!"); return 0; }
static void __exit lkm_example_exit(void)// This will run when unloaded
{ printk(KERN_INFO "Goodbye World!"); }
module_init(lkm_example_init); // To make the correct function run
module_exit(lkm_example_exit); // To make the correct function run
```

[10] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf
[11] https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

After compiling the module, a *.ko* will appear in the directory of the source, along with a plethora of other (for us) uninteresting files. After copying it to the device, you can load and unload it with *insmod <file>.ko* and *rmmod <file>.ko*.

Kthreads

In the previous section, the basic skeleton of a kernel module was introduced. It did something when it was loaded and something else when it was unloaded - but what if some long-running functionality was necessary? In our case, this is the situation, as we want to respond to GPIO signals right away *at any time* - meaning it should not be restricted to the time we load it. For this purpose, two solution exist:

1. Making the long-running functionality time-constrained, i.e. limiting how long it should be allowed to run. Then it can be invoked from the init function.
2. Creating a long-running background thread that is allowed to exist until we unload the module.

Option #2 is a lot more elegant, and therefore we will explore that one.
As we are running in kernel mode, there is no way to just simply invoke a *fork* syscall and use it to run a function asynchronously. Instead, since kernel version 2.6, there are new primitives to be used for this purpose (until then, this functionality was always implemented in a custom way), called kthreads, which are lightweight processes running in kernel mode.
Its use is fairly simple and well detailed in the source (https://lwn.net/Articles/65178/).

Character Devices[12]

So far, we have seen how kernel modules are built - but how can we actually use them, besides loading and unloading them? The UNIX philosophy (and by extension, that of Linux) says that *everything is a file*[13] - and this is how most device drivers are built. For example, there are files under */dev/* representing physical disks such as HDDs and SSDs (e.g. /dev/sda) and virtual terminals (e.g. /dev/tty1). In this example, the former is called a *block special* and the latter a *character special* file (confirm these by issuing *file /dev/{sda,tty1}*).
To achieve the goals of our loader, we are going to use a *character special* file which will provide the following services:

- When being read, it prints a message to the kernel log stating it is write only
- When being written to, it treats it as a program to be run - if the last program has terminated and is currently waiting for the next, the module will load the new program into memory and then make the core jump to it.

Linux provides us with a well-build toolset to deal with character devices[14].

[12] https://linux-kernel-labs.github.io/master/labs/device_drivers.html
[13] https://en.wikipedia.org/wiki/Everything_is_a_file
[14] http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Megjegyzés: Ez a rész még nincs kész. Ki kéne fejteni. Egyelőre még nem volt rá időm, de szívesen megcsinálom.

Measuring path: We connect pin 14-15, and 23-24 together. We have written the programs so that pin 15 is mirrorred to pin 23 (**not** using interrupts, using a busywait loop). We change the level of pin #14, and measure the level on pin #24 using interrupts (so that there is virtually no wait time), and we take note of the time in both instances. The difference is the *latency* of the system (of course a bit skewed,as there is an additional two jumps, but good enough).

Blinking-led path: We make the on-board led blink.

Structure of the Tasks (**measuring path**)

1. Using a general-purpose kernel
    a. Measure latency with minimal stress on the cpus
    b. Measure latency with aggressive stress on the cpus
    c. Measure latency with aggressive stress and priority adjustments
        i. Low: 19
        ii. High: -20
    d. Measure latency while running in kernel-space
2. Using a real-time kernel
    a. Measure latency with aggressive stress and real-time priority
        i. 1
        ii. 99
    b. Measure latency with aggressive *real-time* stress and real-time priority
        i. Lower (e.g. 1 for the stress and 99 for the task)
        ii. Higher (e.g. 1 for the task and 99 for the stress)
3. Using an AMP system (Asynchronous Multi-Processing)
    a. Measure latency with minimal stress on the linux-cpus
    b. Measure latency with aggressive stress on the linux-cpus

Structure of the Tasks (**blinking led path**)

1. Using a general-purpose kernel
    a. Blink the leds with minimal stress on the cpus
    b. Blink the leds with aggressive stress on the cpus
    c. Blink the leds with aggressive stress and priority adjustments
        i. Low: 19
        ii. High: -20
    d. Blink the leds while running in kernel-space
2. Using a real-time kernel
    a. Blink the leds with aggressive stress and real-time priority
        i. 1
        ii. 99
    b. Blink the leds with aggressive *real-time* stress and real-time priority
        i. Lower (e.g. 1 for the stress and 99 for the task)

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

ii. Higher (e.g. 1 for the task and 99 for the stress)

3. Using an AMP system (Asynchronous Multi-Processing)
    a. Measure latency with minimal stress on the linux-cpus
    b. Measure latency with aggressive stress on the linux-cpus

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.