International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I7P104

Volume 72 Issue 7, 26-31, July 2024

Original Article

Hybrid Hardware in Loop Architecture for Autonomous Vehicles

Shobhit Kukreti[1], Tanvi Hungund[2], Priyank Singh[3]

[1]*Independent Software Researcher, Carnegie Mellon University, PA, USA.*
[2]*Independent Software Researcher, Cal State Fullerton, CA, USA.*
[3]*Indepedendent Software Researcher, Rochester Institute of Technology, NY, USA.*

[1]*Corresponding Author : skukreti@linux.com*

Received: 10 May 2024 Revised: 21 June 2024 Accepted: 11 July 2024 Published: 30 July 2024

Abstract *- The automotive world is going through a paradigm shift. With sensor technologies like 4K cameras, LiDARs, Radars, etc., becoming more affordable and advances in Deep Learning algorithms development, research is trending towards developing Autonomous Vehicles and bringing them mainstream. However, a challenge remains in the commercialization of technologies. An automotive vehicle undergoes a rigorous verification and validation model. Perfected over decades, standards and guidelines are introduced to minimize failures. However, there exists a gap in testing when working with new sensors and algorithms, and our paper focuses on a hybrid setup for hardware in loop testing, which enables software to iterate over their development faster without having to wait to deploy on actual cars for testing.*

Keywords *- Autonomous Vehicles, Linux, Operating System, Emulation, QEMU.*

1. Introduction

Embedded Machine Learning, computing on the edge is crucial for fostering innovation in a multitude of intelligent devices. The evolution of ARM processors has created an ecosystem of smart devices where significant computing exists to run Machine Learning Models. The next innovation is touted to be in the automotive sector, where machine learning and hardware combine to improve the transportation sector. The edge computer means that the decision-making power is now with an ECU [1] (Engine Control Unit). A car can have multiple ECUs, with each having its own functionality, from controlling the wipers to controlling the steering accelerator based on the user input. In autonomous vehicles, a software agent shall drive these actuators. This makes the testing of the software, along with the underlying hardware, a challenge. Hardware-in-the-loop (HIL) testing involves connecting a real hardware system (such as an ECU) to a virtual environment that simulates real-world conditions.

This allows engineers to test the control systems' behavior under various conditions without needing the entire vehicle. HIL testing allows identify potential issues early in the development process, thus reducing the need for expensive and time-consuming physical prototypes. Figure 1 shows a V-Model, also called the verification and validation model. International industry standards prescribe the model for the development of safety-critical systems like ISO26262. While the left side of the V model focuses on requirements at both macro and micro levels, the right side focuses on software integration and testing. In the initial stages, the software is tested in a purely simulated environment to allow for unit and module testing, while the later stages test the overall software on the target hardware platform with a full load of sensors.

Often, the testing infrastructure between the initial stages in the simulated environment vs the actual target hardware brings in a huge variation. For instance, a software driver is implemented to capture 3d point cloud data from a LiDAR. The software driver in order to test, will fake add function stubs in code to mimic the sequence of operations to be performed on the LiDAR hardware only to see hardware-related issues when testing the software on the target hardware.

Our paper aims to bridge the gap in the testing process with a Hybrid Hardware Loop Testing process, which uses QEMU [2] virtualization and creating custom device models in QEMU to enable hardware systems to be patched through to the virtualized environment.

The rest of the paper is organized as follows. Introduction to hardware emulators in section II. Section III describes QEMU, Section IV describes the Process of Adding User Defined Devices in QEMU. Section V shall have the Device Driver Development in Linux and we shall conclude in section VI. Fig. 1 v- model of development

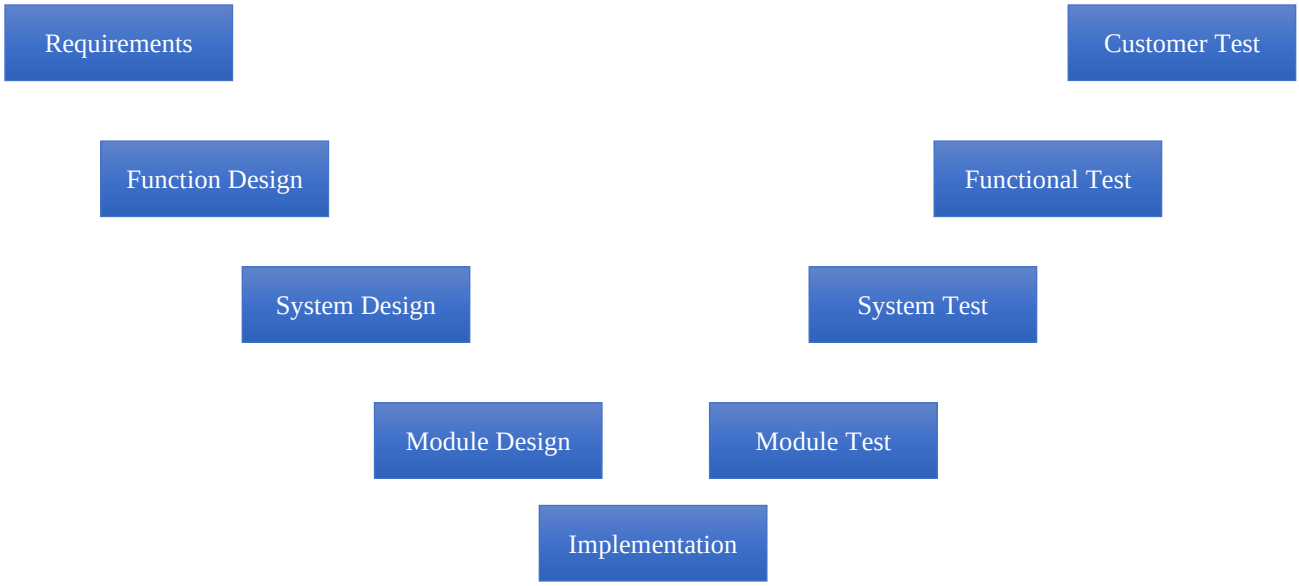

Fig. 1 v-model of development

```
cd qemu
mkdir build
cd build
../qemu/configure --target-list=aarch64-softmmu
make
```

Fig. 2 Qemu build instruction

```
qemu-system-aarch64 -M virt -cpu cortex-a57 \
-smp 4 -m 1G  -kernel Image \
-append "root=/dev/vda rw console=ttyAMA0" \
-semihosting -nographic -drive file=rootfs.ext4
```

Fig. 3 Invoking QEMU

2. Hardware Emulators

An embedded system product development depends on the availability of the hardware. In the absence of hardware and for the software development to progress, developers often use hardware emulators. For instance Android developers use the Android Emulator [3]for developing their application. In this context, emulation can be defined as using a host CPU's resources to mimic the functionality of another CPU architecture. Thus a software developed for say ARM CPU architecture can be run on a host system which has an x86 architecture. This is the typical use-case of an Android emulator for android application development. Some widely used emulation products are QEMU, BOCHS [4] RENODE [5] In automotive ECU HIL testing commercial setups like dSPACE [6] are often used for validation. Our paper focuses on QEMU and its application in the Hardware in Loop Testing. QEMU, an open-source emulator which employs dynamic binary translation to achieve high performance when emulating guest systems on different host architectures. The modular nature of QEMU allows it to emulate a wide range of hardware components, including CPUs, network interfaces, and storage controllers, making it highly adaptable for various use cases. QEMU's KVM (Kernel-based Virtual Machine) integration allows for near-native execution speeds by leveraging hardware virtualization extensions if supported by the host system. While QEMU supports multitude of standard hardware interfaces, the Autonomous Vehicle domain brings in new hardware which are either not modeled in QEMU or not yet available in public domain to be accessible to everyone.

3. QEMU

We will need the QEMU source code to add custom hardware. You can download the source from the official QEMU repository https://gitlab.com/qemu-project/qemu.git. To build QEMU source code: We also require a root file system and a Linux Kernel Image to use on our custom QEMU emulator. You can use the debootstrap [7] tool to generate a Debian [8] base image and the Linux Kernel [9] source code from the link below. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git .

Another popular option is to use buildroot [10] (https://buildroot.org/) for generating a root file system along with the kernel image. Since we are generating an image for the QEMU Virtual Platform, we shall use the config *qemu_aarch64_virt_defconfig.* Once both qemu and buildroot finish the *making* process, Qemu can be invoked using the command below, shown in Figure 3. QEMU shall execute with the kernel image and the roots parameter to start a serial console emulating an ARM64 [11] virtual system, as shown in Figure 4.

```
Booting Linux on physical CPU 0x0000000000 [0x411fd070]
Linux version 6.6.18 (shwetashob@t-1000) (aarch64-buildroot-
linux-gnu-gcc.br_real (Buildroot 2024.02-479-gdb37b0e27d)
12.3.0, GNU ld (GNU Binutils) 2.41) #15 SMP Sun Apr 28
23:22:46 PDT 2024
random: crng init done
…….
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: udhcpc: started, v1.36.1
udhcpc: broadcasting discover
udhcpc: no lease, forking to background
OK

Welcome to Buildroot
buildroot login:
```

Fig. 4 Serial console

4. User Defined Custom Hardware Model

With the standard QEMU invocation completed, we shall now focus on creating a custom hardware model. To re-iterate, this user-defined model being developed can be used to replicate a hardware interface, say a sensor register layout, add function hooks into a world view simulator CARLA [12] or add software to create a pass-through of the camera data, which is connected to the host via a special interface like GMSL [13]. Our target system is an Embedded ARM64 SOC, as shown in Figure 5, with a user-defined co-processor added to the system bus. To show QEMU's scalable and modular architecture, we can even send an interrupt event from this co-processor to the main Application Processor. In the QEMU source code directory, we shall modify the virtual platform defined under *hw/arm/virt.c* and its associated header file. In the *include/hw/arm/virt.h* file we add a device type *VIRT_CUSTOM_COPROC*. ARM/ARM64 architecture has a flat memory model.

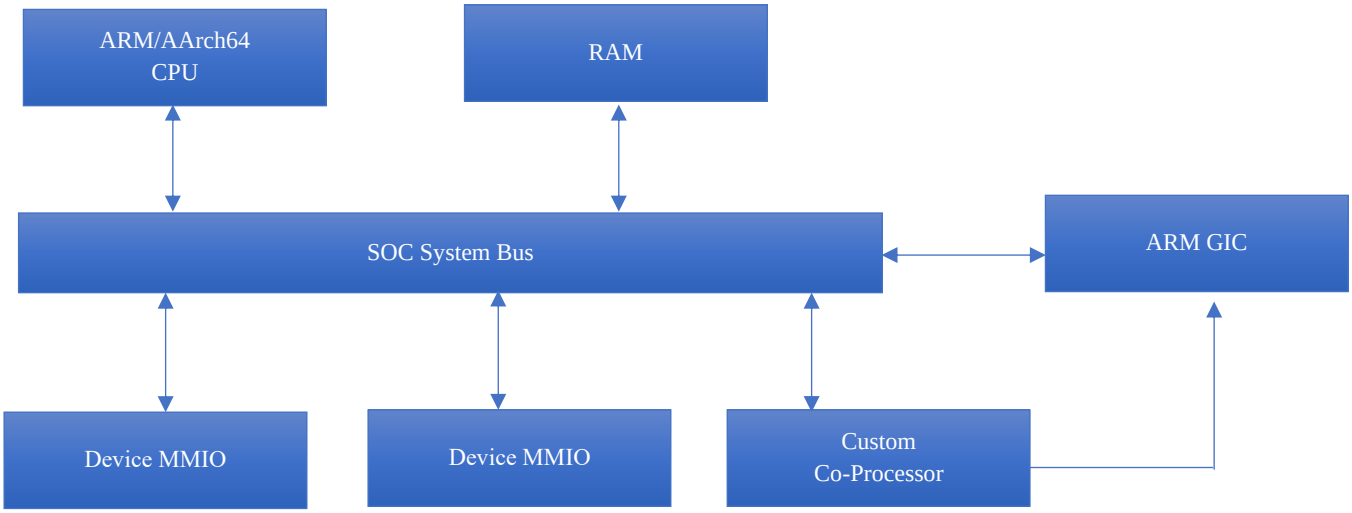

Fig. 5 ARM virtual platform

The data structure *MemMapEntry* holds the memory map of the entire SOC. We shall add an entry in a vacant space at address 0x0b000000 with a length of 0x200. Similarly, we add a virtual IRQ from our user-defined co-processor to the Application Processor. Next, we shall create the software model of our user-defined model, which is slotted at base address 0x0b000000 and has an associated interrupt. Under include/hw/arm and under hw/arm, create a new sub-folder, which will contain the header file and the source file of your model.Our user-defined model has a device id of *0x42024*. The id offset can be read at base_address + 0x0 (kPeripheralIDOffset). The simple model shown here shall take in two 32-bit input numbers in the input registers from the Application Processor and shall return the output of it in the output register. The register at offset 0x14 (kStartProcessing) shall be the trigger from the software driver to begin the compute. Additionally, the device driver can check if the user modeled coprocessor is busy with the register at 0x10 (kProcIdle) as well as reset the co-processor with a reset register at 0x30 (kResetCoProc). If the Device Driver configures the interrupt (register offset 0x8) to be enabled, the user model shall send a completion interrupt back to AP once the execution is completed.

```
static const MemMapEntry base_memmap[] = {
    /* Space up to 0x8000000 is reserved for a boot ROM */
    [VIRT_FLASH] =              {          0, 0x08000000 },
    [VIRT_CPUPERIPHS] =         { 0x08000000, 0x00020000 },
   .....
    [VIRT_MMIO] =               { 0x0a000000, 0x00000200 },
    [VIRT_CUSTOM_COPROC] =      { 0x0b000000, 0x00000200 },
   .....
    /* Actual RAM size depends on initial RAM and device memory settings */
    [VIRT_MEM] =                { GiB, LEGACY_RAMLIMIT_BYTES },
};
```

Fig. 6 Virtual ARM Platform Memory Map

```
static const int a15irqmap[] = {
    [VIRT_UART] = 1,
    [VIRT_RTC] = 2,
    .....
    [VIRT_SMMU] = 74,    /* ...to 74 + NUM_SMMU_IRQS - 1 */
    [VIRT_PLATFORM_BUS] = 112, /* ...to 112 + PLATFORM_BUS_NUM_IRQS -1 */
    [VIRT_CUSTOM_COPROC] = 112 + PLATFORM_BUS_NUM_IRQS,
};
```

Fig. 7 Virtual ARM Platform IRQs

```
#define TYPE_CUSTOM_COPROC "CoProcessor-Custom"
#define CoProcessor(obj) OBJECT_CHECK(CoProcessorState, (obj), TYPE_CUSTOM_COPROC)


static const uint32_t kPeripheralID = 0x042024;


/* Register Layout */
static const uint32_t kPeripheralIDOffset = 0x0;

static const uint32_t kCompletionIRQEnOffset = 0x8;
static const uint32_t kCompletionIRQStatus = 0xc;

static const uint32_t kCoProcIdle = 0x10;
static const uint32_t kStartProcessing = 0x14;

static const uint32_t kInpReg1 = 0x18;
static const uint32_t kInpReg2 = 0x1c;
static const uint32_t kOutputReg = 0x20;

static const uint32_t kResetCoProc = 0x30;

static const uint32_t kDebugEn = 0x40;


typedef struct
{
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    QEMUTimer *timer;   // QEMU Internal Time
    qemu_irq irq;       // To send IRQ to AP
    bool irqStatus;     // Single IRQ Status Line
    uint32_t id;        // Co-Processor ID
    uint32_t en;        // Enable IRQ flag
    uint32_t inp1;      // Input 1 Register
    uint32_t inp2;      // Input 2 Register
    uint32_t out;       // OutPut Register
    bool busy;          // Co-Processor Busy Signal
    bool reset;         // Reset CoProcessor
    bool debug;         // SW Debug Flag for Verbose Output

} CoProcessorState;
```

Fig. 8 User defined model attributes

Our new model registration with QEMU uses the macro type_init.

```
static const TypeInfo _coProcessorInfo = {
    .name = TYPE_CUSTOM_COPROC,
    .parent = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(CoProcessorState),
    .instance_init = coProcessorInit,
    .instance_finalize = coProcessorCleanUp,
};

static void registerCoProcessor(void)
{
    type_register_static(&_coProcessorInfo);
}

type_init(registerCoProcessor)
```

Fig. 9 Code to register the user-defined model

```
static const MemoryRegionOps CoProcOps = {
    .read = CoProcRead,
    .write = CoProcWrite,
    .endianness = DEVICE_NATIVE_ENDIAN,
};

static void CoProcessorWorkCompletion(void *opaque)
{
    CoProcessorState *s = (CoProcessorState *)opaque;
    s->out = s->inp1 + s->inp2;
    s->busy = false;
    s->irqStatus = true;
    CoProcSetIRQ(s);
}

static void coProcessorInit(Object *obj)
{
    CoProcessorState *state = CoProcessor(obj);
    SysBusDevice *sbd = SYS_BUS_DEVICE(obj);

    memory_region_init_io(&state->iomem, OBJECT(state), &CoProcOps, state,
                          TYPE_CUSTOM_COPROC, 0x200);
    sysbus_init_mmio(sbd, &state->iomem);
    sysbus_init_irq(sbd, &state->irq);

    // initialize internal state
    state->id = kPeripheralID;
    state->timer = timer_new_ns(QEMU_CLOCK_VIRTUAL, CoProcessorWorkCompletion, state);
}
```

Fig. 10 Init

```
static uint64_t CoProcRead(void *opaque, hwaddr offset, unsigned size)
{
    CoProcessorState *s = (CoProcessorState *)opaque;

    uint32_t ret = 0xDEADBEEF;

    switch (offset)
    {
    case kPeripheralIDOffset:
        ret = s->id;
        break;
    ....
    case kCoProcIdle:
        ret = !s->busy;
        break;
    ....
    case kOutputReg:
        ret = s->out;
        break;
    }

    return ret;
}
```

Fig. 11 MMIO read function

```
static void CoProcWrite(void *opaque, hwaddr offset, uint64_t value,
                        unsigned size)
{
    CoProcessorState *s = (CoProcessorState *)opaque;
    switch (offset)
    {

    case kCompletionIRQEnOffset:
        s->en = value & 0x1;
        break;
    case kInpReg1:
        s->inp1 = value;
    break;
    case kInpReg2:
        s->inp2 = value;
    break;
....
    break;
    case kStartProcessing:
        s->busy = value ? true : false;
        if (s->busy) {
            s->out = 0;
            setUpTimer(s);

        }
        break;
....
    }
}
```

Fig. 12 MMIO write function

```
static void CoProcessorWorkCompletion(void *opaque)
{
    CoProcessorState *s = (CoProcessorState *)opaque;
    s->out = s->inp1 + s->inp2;
    s->busy = false;
    s->irqStatus = true;
    CoProcSetIRQ(s);
}
```

Fig. 13 QEMU timer callback

The code snippets in Figures 9 to 14 show how the co-processor is registered with QEMU, its init sequence where the memory device memory region is intialized with the length 0x200. The read/write APIs get invoked when a driver makes a hardware register access in the co-processor assigned memory section. Once the driver sets a non-zero value in the kStartProcessing register, we trigger the QEMU internal timer. On completion of the time period, we receive a callback where we output the result of the execution as well as send an interrupt to the Application Processor. After adding the code and adding the source files to the QEMU Kconfig/Meson build system, we are ready to rebuild the QEMU binary.

```
static void create_custom_coproc(const VirtMachineState *vms)
{
    hwaddr base = vms->memmap[VIRT_CUSTOM_COPROC].base;
    hwaddr size = vms->memmap[VIRT_CUSTOM_COPROC].size;
    int irq = vms->irqmap[VIRT_CUSTOM_COPROC];
    char *nodename;

   sysbus_create_simple(TYPE_CUSTOM_COPROC, base, qdev_get_gpio_in(vms->gic, irq));

    MachineState *ms = MACHINE(vms);

    nodename = g_strdup_printf("/coproc@%" PRIx64, base);
    qemu_fdt_add_subnode(ms->fdt, nodename);
    qemu_fdt_setprop_string(ms->fdt, nodename, "compatible", "coproc");
    qemu_fdt_setprop_sized_cells(ms->fdt, nodename, "reg",
                                 2, base, 2, size);
    qemu_fdt_setprop_cells(ms->fdt, nodename, "interrupts",
                           GIC_FDT_IRQ_TYPE_SPI, irq,
                           GIC_FDT_IRQ_FLAGS_LEVEL_HI);

    g_free(nodename);
}
```

Fig. 14 Instantiating the CoProcessor in virt.c

5. Device Driver and Validation

With the new compiled qemu binary, if we start the emulator with the ARM virtual machine in the monitor mode and run the command info mtree, you shall see the user-defined coprocessor in the system address map.

Next, if we read the memory using devmem [14] [at the base-address of our new device, we should read the hard-coded device id, which confirms our user-model read operation is working as expected.

```
build ./qemu-system-aarch64 -machine virt -monitor stdio
QEMU 9.0.50 monitor - type 'help' for more information
(qemu) info mtree
address-space: cpu-memory-0
address-space: memory
0000000000000000-ffffffffffffffff (prio 0, i/o): system
 0000000009010000-0000000009010fff (prio 0, i/o):
pl031
....
....
 000000000b000000-000000000b0001ff (prio 0, i/o):
CoProcessor-Custom
```

Fig. 15 Address map with user model

```
# devmem 0xb000000
0x00042024
#
```

We shall write a tiny linux device driver which will match against the compatible string “coproc” as defined earlier in our hw/arm/virt.c file. We read the interrupt property in the device tree and register an interrupt handler for it. Recall that our co-processor fires an interrupt when the device driver initiates the processing of the inputs by writing to the register at offset 0x14, which in turn triggers an internal QEMU Timer. The timer call back is used to send an interrupt to the AP.

```
static int coproc_remove(struct platform_device *pdev)
{
   return 0;
}
static const struct of_device_id coproc_of_match[] = {
   { .compatible = "coproc", },
   { }
};
MODULE_DEVICE_TABLE(of, coproc_of_match);
static struct platform_driver coproc_driver = {
   .probe = coproc_probe,
   .remove = coproc_remove,
   .driver = {
      .name = "coproc-driver",
      .of_match_table = of_match_ptr(coproc_of_match),
   },
};
module_platform_driver(coproc_driver);
```

Fig. 16 Registering the linux device driver

```
static int coproc_probe(struct platform_device *pdev)
{
   struct device *dev = &pdev->dev;
  ...
   cd->base = devm_ioremap(dev, res->start,
resource_size(res));

   cd->virq = irq_of_parse_and_map(pdev->dev.of_node,
0);
   if (cd->virq == 0) {
   }
   else {
      ret = request_irq(cd->virq,
(irq_handler_t)coproc_irq_handler,
                  IRQF_TRIGGER_RISING,
"COPROC_IRQ_HANDLER", cd);
   }
   // Enables Interrupt
   writel(1, cd->base + 0x8);
}
```

Fig. 17 The probe function

```
static irqreturn_t coproc_irq_handler(int irq, void *data)
{
   struct coproc_data *cd = (struct coproc_data*) data;

   pr_info("--------------------------------CoProc Irq Rx\n");
   readl(cd->base + 0xc);
   pr_info("--------------------------------CoProc Irq Cleared
When Status Reg is Read\n");
   return IRQ_HANDLED;
}
```

Fig. 18 The Co-Processor interrupt handler

We compile the device driver into the Linux Kernel Image and re-invoke QEMU with the new kernel image. On running cat */proc/interrupts*, we should see our IRQ handler registered.

```
# cat /proc/interrupts
           CPU0
 11:       8577 GIC-0  27 Level     arch_timer
 13:        144 GIC-0  33 Level     uart-pl011
 16:          0       MSI 32768 Edge      virtio1-config
 17:        188       MSI 32769 Edge      virtio1-req.0
 18:          0       MSI 16384 Edge      virtio0-config
 19:          0       MSI 16385 Edge      virtio0-input.0
 20:          0       MSI 16386 Edge      virtio0-output.0
 21:          0 GIC-0  34 Level     rtc-pl031
 22:          5 GIC-0 208 Edge      COPROC_IRQ_HANDLER
 23:          0 GIC-0  23 Level     arm-pmu
IPI0:         0       Rescheduling interrupts
```

Fig. 19 Linux interrupts

```
Welcome to Buildroot
buildroot login: root
# sh test.sh
CoProc ID:
0x00042024
CoProc Will Add 10 20
Trigger Start Processing
Sending IRQ to Main Processor
------------------------------CoProc Irq Rx
------------------------------CoProc Irq Cleared When Status Reg is Read
Check Completion Status Reg
0x00000000
Output Reg
0x0000001E
#
```

Fig. 20 Running the shell script

We shall use a shell script with devmem to write two input numbers and observe the output. An interrupt is generated on completion of the work and we see the print statements in the serial console.

6. Conclusion

In this paper, we present a new hybrid approach to performing Hardware Loop Testing for automotive applications. QEMU being a versatile emulator, we add a user-defined model using the 'C' programming language.

With the HIL setup as part of the test infrastructure, developers can focus on higher-level Computer Vision and Deep Learning problems with real input data such as camera images, radar data or the 3d point cloud data from a LiDAR.

For future work, we intend to provide create a sensor framework layer which will allow the user-defined model to interface with automotive sensors seamlessly.

References

[1] Jayshri Sudhir Potdar, and Yashwant B Mane, "Hardware Design and Development of Engine Control Unit for Four Cylinder Engine," *2018 Fourth International Conference on Computing Communication Control and Automation*, Pune, India, pp. 1-5, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[2] Run apps on the Android Emulato, Developers. [Online]. Available: https://developer.android.com/studio/run/emulator

[3] QEMU. [Online]. Available: qemu.org

[4] BOCHS. [Online]. Available: https://bochs.sourceforge.io/

[5] RENODE. [Online]. Available: https://renode.io/

[6] dSPACE HIL. [Online]. Available: https://www.dspace.com/

[7] Debootstrap. [Online]. Available: https://wiki.debian.org/Debootstrap

[8] Debian. [Online]. Available: https://www.debian.org/

[9] Linux Kernel. [Online]. Available: https://www.kernel.org/

[10] Buildroot. [Online]. Available: https://buildroot.org/

[11] ARM64. [Online]. Available: https://en.wikipedia.org/wiki/AArch64

[12] CARLA. [Online]. Available: https://carla.org/

[13] Kainan Wang, Gigabit Multimedia Serial Link (GMSL) Cameras as an Alternative to GigE Vision Cameras, Analog Devices, 2023. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/gigabit-multimedia-serial-link-gmsl-cameras.html

[14] Devmem. [Online]. Available: https://trac.gateworks.com/wiki/linux/devmem