CS3500: Operating Systems Course Project

UART Device Driver for ZephyrRTOS

Anirudh Ajith (CS18B070) Abhinav Hampiholi (CS18B065)

1 Background

The goal of this project was to write a UART device driver for the ZephyrRTOS running on the Shakti Parashu SoC. We have defined a few of these terms below.

The RISC-V micro-controller that we are working on is the *Parashu Shakti E-class* SoC. This board, being E-class is meant to be used in embedded systems applications.

We will be implementing a device driver for the *UART* or Universal Asynchronous Reciever-Transmitter serial communication protocol/device. The Parashu board comes with 3 UART controllers. We will be writing a driver which is capable of handling communication over all 3 controllers.

A *device driver* is a program which provides a software interface to hardware devices. This allows Operating Systems and other computer programs to access hardware functions without needing to know precise details about the hardware being used.

Our goal is to write a device driver that creates an interface between the built-in UART controllers on the Parashu SoC and the Zephyr Operating System.

Doing so requires an understanding of three things: the UART controller and protocol itself, the API for a UART peripheral that ZephyrRTOS lays out, and the details of how the UART devices are memory-mapped in the Parashu SoC. The aim of this section is to provide some background on these topics.

1.1 The UART controller

A UART is a hardware device that facilitates serial communication. Each UART has two pins, a transmitting (Tx) pin and a receiving (Rx) pin. When two UART devices communicate, data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART.

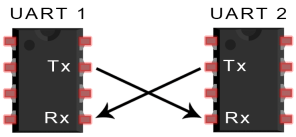


Figure 1: Typical UART data transmission [1]

UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving

UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

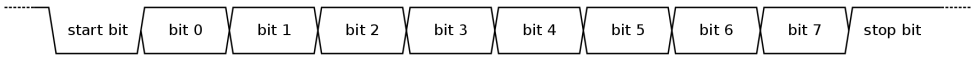


Figure 2: Start and Stop bits [7]

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. If their baud rates differ by more than 10%, the data received by the receiving device would be erroneous. However, the details of packing the data into packets before transmission and unpacking them on reception are handled by the hardware circuitry of the UART device and we need not concern ourselves with those details while writing a device driver.

1.2 The Zephyr Operating System

ZephyrRTOS is a real-time operating system that can be built to run on over two-hundred different SoCs out-of-the-box. Adding support for a new board can be done by following the documentation for adding a new board. The Zephyr project includes a versatile command line tool named `west`. To build Zephyr for a particular board, we can run the `west build -b <BOARD>` command.

Zephyr also provides APIs for several peripherals. According to the official API documentation, Zephyr communicates with a UART peripheral in 'polled mode' through a structure `uart_driver_api` given below

```
struct uart_driver_api = {
   .poll_in        = uart_poll_in,
   .poll_out       = uart_poll_out,
   .err_check      = NULL,
};
```

`uart_poll_out` can be called by user applications to write a single character to the output stream emanating from the UART Tx pin (in polled mode). This routine must check if the transmitter buffer is not empty by continually polling it, and when space for a character in the tranmitter buffer has been freed up, it must write a character to the data register.

`uart_poll_in`, when called by the user application, must continually poll the reciever pin of the UART device for input. When a packet is received, it must somehow return that character to the user program.

Writing a device driver consists mostly of filling in function definitions for the functions in the above structure while keeping in mind the hardware specifications of the Parashu SoC.

Here, we note that although there are additional requirements for an 'interrupt driven' implementation of UART, we have chosen to ignore them and support only a polling implementation. This choice was made in the interest of simplicity.

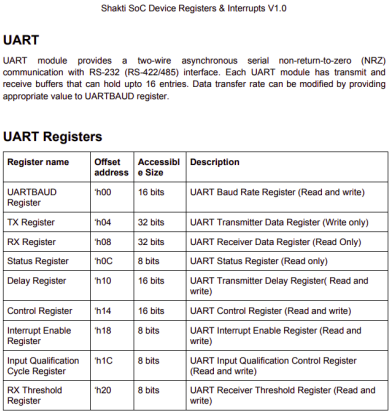
1.3 The Parashu memory map

The SHAKTI based SoCs consist of the processor, memory and various Input-Output devices (I/O). The devices in the SoC are memory mapped. This means that certain locations in the physical address space are associated with registers of the I/O devices. The Shakti SoC user manual tells us that the memory-mapped locations on the Parashu SoC for all the peripheral devices it supports. For instant, the registers associated with the UART0 controller are associated with the memory locations `0x11300 - 0x1133f`.

3.	UART0	0x00011300	0x00011340
4.	UART1	0x00011400	0x00011440
5.	UART2	0x00011500	0x00011540

Figure 3: Parashu Memory Map

The Shakti SoC registers documentation gives us more information about which memory locations each UART register is mapped to, relative to the base address (eg. `0x11300` for UART0).



Shakti SoC Device Registers & Interrupts V1.0

UART

UART module provides a two-wire asynchronous serial non-return-to-zero (NRZ) communication with RS-232 (RS-422/485) interface. Each UART module has transmit and receive buffers that can hold upto 16 entries. Data transfer rate can be modified by providing appropriate value to UARTBAUD register.

UART Registers

Register name	Offset address	Accessible Size	Description
UARTBAUD Register	'h00	16 bits	UART Baud Rate Register (Read and write)
TX Register	'h04	32 bits	UART Transmitter Data Register (Write only)
RX Register	'h08	32 bits	UART Receiver Data Register (Read Only)
Status Register	'h0C	8 bits	UART Status Register (Read only)
Delay Register	'h10	16 bits	UART Transmitter Delay Register(Read and write)
Control Register	'h14	16 bits	UART Control Register (Read and write)
Interrupt Enable Register	'h18	8 bits	UART Interrupt Enable Register (Read and write)
Input Qualification Cycle Register	'h1C	8 bits	UART Input Qualification Control Register (Read and write)
RX Threshold Register	'h20	8 bits	UART Receiver Threshold Register (Read and write)

Figure 4: Detailed Parashu Memory Map

2 Workflow

We began by cloning the latest version of the ZephyrRTOS operating system codebase from the official repository onto our local machine. We then followed the instructions available on the Zephyr documentation website to correctly set up the Zephyr SDK on our machine. This involved installing and/or updating dependencies, setting a few environment variables used by Zephyr to specify the toolchain used, the installation directory of the SDK and so on. To test if the we had set up the SDK correctly, we tried to compile the Zephyr codebase for a RISC-V board called `hifive1` (supported by Zephyr out-of-the-box), which we knew was similar to our target architecture: the Parashu Shakti E-class board. To do this, we ran the command

```
west build -p auto -b hifive1 samples/hello_world/
```

from within the `zephyr` directory. After some debugging, we were able to get the cloned version of Zephyr to build without errors.

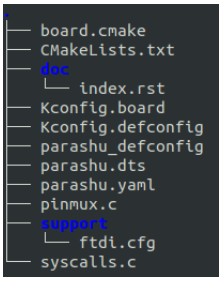


Figure 5: Contents of `zephyr/boards/riscv/parashu`

Next, we had to provide the specifications of the Parashu board to Zephyr in order to allow `west` to be able to compile the OS for its hardware. We found out that the specifications and various configuration files for supported SoCs were laid out in the `zephyr/boards` directory. We found the configuration files for `hifive1` under `zephyr/boards/riscv/hifive1` and explored them to figure out how add support for Parashu. To finally add support, we replicated the directory tree structure we found under `zephyr/boards/riscv/hifive1` under a new directory, `zephyr/boards/riscv/parashu` and analogously filled in the contents of the newly created files.

Importantly, we had to specify details about the memory-mapping scheme for the various devices (including the UART controllers) inside `parashu.dts`. We also had to provide a more elaborate list of memory-maps and their configurations in the file `zephyr/dts/riscv/riscv32-parashu.dtsi`. We did this in accordance with the information we found about the Parashu board in the Shakti SoC User Manual. Specifically for the UART controllers, we found that the UART controllers are memory-mapped between the addresses `0x11300-0x1133f`, `0x11400-0x1143f` and `0x11500-0x1153f` for the controllers `uart0`, `uart1` and `uart2` respectively.

`Kconfig.defconfig` and `Kconfig.board` were used to set a few configuration properties

```
58          uart0: serial@11300 {
59                  compatible = "shakti,uart0";
60                  interrupt-parent = <&plic>;
61                  interrupts = <25>;
62                  reg= <0x11300 0x40>;
63                  reg-names = "control";
64                  label = "uart_0";
65                  status = "ok";
66          };
67          uart1: serial@11400 {
68                  compatible = "shakti,uart0";
69                  interrupt-parent = <&plic>;
70                  interrupts = <26>;
71                  reg= <0x11400 0x40>;
72                  reg-names = "control";
73                  label = "uart_1";
74                  status = "ok";
75          };
76          uart2: serial@11500 {
77                  compatible = "shakti,uart0";
78                  interrupt-parent = <&plic>;
79                  interrupts = <27>;
80                  reg= <0x11500 0x40>;
81                  reg-names = "control";
82                  label = "uart_2";
83                  status = "ok";
84          };
```

(a) Snippet from `riscv32-parashu.dtsi`

3.	UART0	0x00011300	0x00011340
4.	UART1	0x00011400	0x00011440
5.	UART2	0x00011500	0x00011540

(b) Parashu SoC documentation extract

Figure 6: Specifying memory locations of UART devices

of the board while `parashu_defconfig` was to enable/disable some global constants which served as flags to `west` to conditionally compile certain parts of the codebase.

```
1  # SPDX-License-Identifier: Apache-2.0
2
3  config BOARD_PARASHU
4          bool "PARASHU RISC-V cores"
5          depends on SOC_RISCV_SHAKTI_ECLASS
```

(a) `Kconfig.board`

```
1  # SPDX-License-Identifier: Apache-2.0
2
3  if BOARD_PARASHU
4
5  config BOARD
6          default "parashu"
7
8  config HAS_FLASH_LOAD_OFFSET
9          default n
10
11 config SYS_CLOCK_TICKS_PER_SEC
12         default 128
13
14 endif
```

(b) `Kconfig.defconfig`

```
1  # SPDX-License-Identifier: Apache-2.0
2
3
4  CONFIG_SOC_SERIES_RISCV_SHAKTI=y
5  CONFIG_SOC_RISCV_SHAKTI_ECLASS=y
6  CONFIG_BOARD_PARASHU=y
7  CONFIG_CONSOLE=y
8  CONFIG_PRINTK=y
9  CONFIG_SERIAL=y
10 CONFIG_UART_CONSOLE=y
11 CONFIG_UART_SHAKTI=y
12 CONFIG_UART_SHAKTI_PORT_0=y
13 CONFIG_UART_INTERRUPT_DRIVEN=n
14 CONFIG_RISCV_MACHINE_TIMER=y
15 CONFIG_PLIC=y
```

(c) `parashu_defconfig`

Figure 7: Adding the `parashu` configuration files

After making these changes, we were able to successfully build ZephyrRTOS for the `parashu` board using the command

```
west build -p auto -b parashu samples/hello_world/
```

.

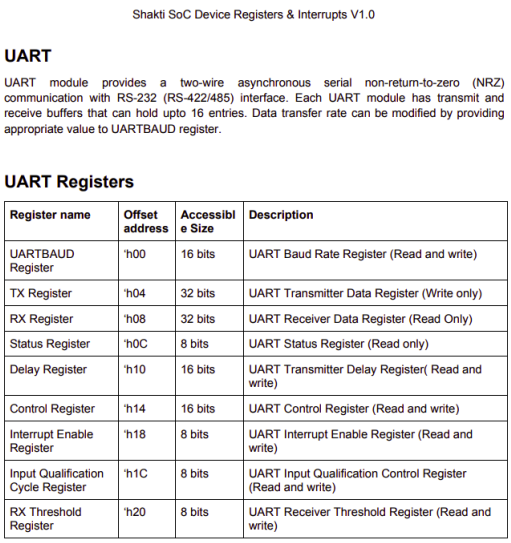
Next, we moved on to adding UART driver support for Parashu. We researched the UART protocol from various sources and read Zephyr's API documentation for UART. We also examined the existing UART driver implementation for the `hifive1` board which we found at `zephyr/drivers/serial/uart_sifive.c`. Writing a device driver for UART involved obeying Zephyr's expected UART API to provide subroutines with certain functionality. What we had to do was to write implementations for these functions. According to the UART protocol, these functions are supposed to read and write to certain registers on the UART device in a specified manner. We do this by making use of the fact that each UART controller's registers are memory mapped into a contiguous 64-byte segment starting at the `0x11300`, `0x11400` and `0x11500` respectively such that the relative positions of the registers in each memory mapping our consistent. Following work that our TA had done writing device drivers for ZephyrRTOS on Shakti devices, we decided to use a C `struct` whose base address would be set to either `0x11300`, `0x11400` or `0x11500` to conveniently refer to various register addresses within the right segment. We could do this since we knew that the member-variables of a `struct` in C are laid out (roughly) contiguously in memory by the C-compiler. Hence, a pointer to a `struct` is an ideal tool for accessing positions relative to a base address via human-readable labels.

For simplicity's sake, we decided to implement a polling-driven UART driver and not an interrupt-driven one. We created a new file called `zephyr/drivers/serial/uart_shakti.c` and structured it similar to `zephyr/drivers/serial/uart_sifive.c`. We populated the fields of the `uart_shakti_regs_t struct` such that that there were fields aligned to all 9 of the important register-mappings which belonged to the UART controller device in accordance with Shakti SoC Registers documentation.

```
38  struct uart_shakti_regs_t {
39          uint16_t baud;   /*! Baud rate configuration Register -- 16 bits*/
40          uint16_t reserv0;        /*! reserved */
41          uint32_t  tx;    /*! Transmit register -- the value that needs to be tranmitted needs to be written here-32 bits*/
42          uint32_t  rx;    /*! Receive register -- the value that received from uart can be read from here --32 bits*/
43          char  status;    /*! Status register -- Reads various transmit and receive status - 8 bits*/
44          char  reserv1;   /*! reserved */
45          uint16_t  reserv2; /*! reserved */
46          uint16_t delay;    /*! Delays the transmit with specified clock - 16bits*/
47          uint16_t reserv3;  /*! reserved */
48          uint16_t control;   /*! Control Register -- Configures the no. of bits used, stop bits, parity enabled or not - 16bits*/
49          uint16_t reserv5;  /*! reserved */
50          char ie;             /*! Enables the required interrupts - 8 bits*/
51          char reserv6;   /*! reserved */
52          uint16_t reserv7;  /*! reserved */
53          char  iqcycles; /*! 8-bit register that indicates number of input qualification cycles - 8 bits*/
54  };
```

Figure 8: `uart_shakti_regs_t`

The two main functions we had to implement were `static int uart_shakti_poll_in(const struct device *dev, unsigned char *c)` and `static void uart_shakti_poll_out(const`



Shakti SoC Device Registers & Interrupts V1.0

UART

UART module provides a two-wire asynchronous serial non-return-to-zero (NRZ) communication with RS-232 (RS-422/485) interface. Each UART module has transmit and receive buffers that can hold upto 16 entries. Data transfer rate can be modified by providing appropriate value to UARTBAUD register.

UART Registers

Register name	Offset address	Accessible Size	Description
UARTBAUD Register	'h00	16 bits	UART Baud Rate Register (Read and write)
TX Register	'h04	32 bits	UART Transmitter Data Register (Write only)
RX Register	'h08	32 bits	UART Receiver Data Register (Read Only)
Status Register	'h0C	8 bits	UART Status Register (Read only)
Delay Register	'h10	16 bits	UART Transmitter Delay Register(Read and write)
Control Register	'h14	16 bits	UART Control Register (Read and write)
Interrupt Enable Register	'h18	8 bits	UART Interrupt Enable Register (Read and write)
Input Qualification Cycle Register	'h1C	8 bits	UART Input Qualification Control Register (Read and write)
RX Threshold Register	'h20	8 bits	UART Receiver Threshold Register (Read and write)

Figure 9: Detailed Parashu Memory Map

`struct device *dev, unsigned char c)`. `uart_shakti_poll_in` is a function which reads a single character incoming into `Rx`. It returns -1 if there is no incoming character and 0 otherwise, storing the character into `*c`. The `uart_shakti_poll_out` function is used to write a single character to `Tx` to transmit it to the other device. It does this by waiting/spinning until the `Tx` buffer is no longer full, and then writing a character to `uart->tx`.

```
99  static int uart_shakti_poll_in(const struct device *dev, unsigned char *c)
100 {
101         volatile struct uart_shakti_regs_t *uart = DEV_UART(dev);
102
103         if (uart->status & UART_STS_RX_NOT_EMPTY) {
104                 *c = (uart->rx);
105                 return 0;
106         }
107
108         return -1;
109 }
```

(a) `poll_in`

```
81  static void uart_shakti_poll_out(const struct device *dev,
82                  unsigned char c)
83  {
84          volatile struct uart_shakti_regs_t *uart = DEV_UART(dev);
85
86          /* Wait while TX FIFO is full */
87          while (uart->status & UART_STS_TX_FULL);
88          uart->tx = (int)c;
89  }
```

(b) `poll_out`

Figure 10: Implementations of `poll_in` and `poll_out`

Finally, we expose these functions to the API by setting the appropriate function pointers in a `struct` of type `uart_driver_api`.

```
129 static const struct uart_driver_api uart_shakti_driver_api = {
130         .poll_in          = uart_shakti_poll_in,
131         .poll_out         = uart_shakti_poll_out,
132         .err_check        = NULL,
133 };
134
```

Figure 11: Exposing functions to API

We also had to make changes to a few other files to integrate the driver at `zephyr/drivers/riscv/uart_shakti.c` with the rest of the operating system. Here are some of the important changes we had to make:

- We added mentions of all 3 UART controllers along with their memory mapping ranges and other configuration details to `zephyr/boards/riscv/parashu/parashu.dts` and to `zephyr/dts/riscv/riscv32-parashu.dtsi`.
- We enabled the flags `CONFIG_UART_CONSOLE`, `CONFIG_UART_SHAKTI` and `CONFIG_UART_SHAKTI_PORT_0` in `zephyr/boards/riscv/parashu/parashu_defconfig`.
- We added `uart` to the list of supported peripherals in `zephyr/boards/riscv/parashu/parashu.yaml`.
- Added `uart_shakti.c` to the list of files to be compiled in `zephyr/drivers/serial/CMakeLists.txt` such that it is conditioned on the truth value of the `CONFIG_UART_SHAKTI` flag.
- We provided configuration metadata for the Shakti UART driver under `zephyr/drivers/serial/Kconfig.shakti`.
- We added a line to `drivers/serial/Kconfig` which points to `zephyr/drivers/serial/Kconfig.shakti`.

```
27  &uart0 {
28          status = "ok";
29          current-speed = <19200>;
30          clock-frequency = <50000000>;
31  };
32
33  &uart1 {
34          status = "ok";
35          current-speed = <19200>;
36          clock-frequency = <50000000>;
37  };
38
39  &uart2 {
40          status = "ok";
41          current-speed = <19200>;
42          clock-frequency = <50000000>;
43  };
```

(a) Snippet from `parashu.dts`

```
58          uart0: serial@11300 {
59                  compatible = "shakti,uart0";
60                  interrupt-parent = <&plic>;
61                  interrupts = <25>;
62                  reg= <0x11300 0x40>;
63                  reg-names = "control";
64                  label = "uart_0";
65                  status = "ok";
66          };
67          uart1: serial@11400 {
68                  compatible = "shakti,uart0";
69                  interrupt-parent = <&plic>;
70                  interrupts = <26>;
71                  reg= <0x11400 0x40>;
72                  reg-names = "control";
73                  label = "uart_1";
74                  status = "ok";
75          };
76          uart2: serial@11500 {
77                  compatible = "shakti,uart0";
78                  interrupt-parent = <&plic>;
79                  interrupts = <27>;
80                  reg= <0x11500 0x40>;
81                  reg-names = "control";
82                  label = "uart_2";
83                  status = "ok";
84          };
```

(b) Snippet from `riscv32-parashu.dtsi`

Figure 12: Adding UART configuration details

```yaml
identifier: parashu
name: Shakti parashu
type: mcu
arch: riscv32
toolchain:
  - zephyr
ram: 16
testing:
  default: true
  ignore_tags:
    - net
    - bluetooth
    - xip
  supported:
    - uart
```

Figure 13: Adding UART to the list of supported peripherals in `parashu.yaml`

At this point, we were able to use `west` to successfully build the OS (with integrated driver support for UART controllers) for Parashu.

```
west build -p auto -b parashu samples/hello_world
```

The Zephyr project comes with default test programs in the `tests` directory. We built the test program with the command

```
west build -p auto -b parashu tests/drivers/uart/uart_basic_api
```

and generated an ELF file, `zephyr.elf`. The final phase of testing involved flashing this executable onto the Parashu board and running it. We followed the the instructions to run an executable on the RISE Lab FPGA which already had the Parashu SoC programmed onto it. The executable ran successfully and we were able to verify that the device driver we wrote was working as expected.

3 Future Work

Our polling-driven UART driver, while being simple to implement, may not be the best choice for the embedded system environment that Shakti E-class is designed for. Several processor clock cycles may be wasted in polling. A possible improvement could be using Zephyr's interrupt driver UART API to implement the driver. Although an interrupt-driven driver seems like a better option, we cannot be sure whether the overheads involved in interrupting the processor are smaller than the cost of continually polling in practical scenarios. This can only be settled by comparing both methods on a physical board during real-life usage.

4 Our Code

All our code can be found in this repository.

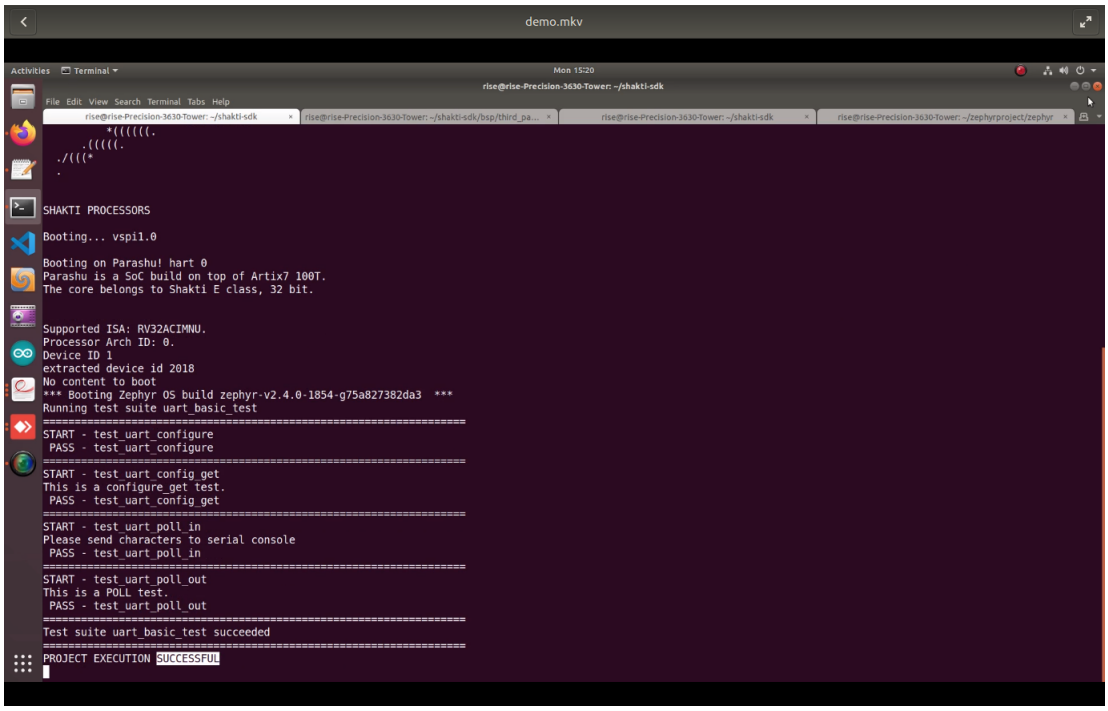


Figure 14: UART test running successfully on the Parashu Board

References

[1] Scott Campbell. *Basics of UART communication*. URL: https://www.circuitbasics.com/basics-uart-communication/.

[2] millsinghion. *UART driver for LPC driver*. URL: https://www.youtube.com/watch?v=RU_NUPprx2Y.

[3] Sathya Narayanan. *Zephyr for Shakti*. URL: https://gitlab.com/sathya281/zephyr.

[4] ShaktiDevelopmentTeam. *Shakti SoC Registers*. URL: https://shakti.org.in/docs/ShaktiSoCRegisters.pdf.

[5] ShaktiDevelopmentTeam. *Shakti User Manual*. URL: https://shakti.org.in/docs/Shakti_SoC_User_Manual.pdf.

[6] *The Zephyr Project*. URL: https://www.zephyrproject.org/.

[7] *Universal asynchronous receiver-transmitter*. URL: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.

[8] *Zephyr codebase*. URL: https://github.com/zephyrproject-rtos.