kern

Sail River O-RU (WS-FAD-00111)

Software Architecture

Last updated: **2025-05-22**

Altera Confidential

Contents

List of Figures

No table of figures entries found.

List of Tables

No table of figures entries found.

1.0 Introduction

This document covers Sail River's software architecture which involves Kernel Driver interface and the User Application for configuring the HW resources or modules instantiated in FPGA. It also covers the MATLAB User Interface used for validating the data, captured at the in and out interfaces of specific modules. The document describes all the kernel driverset in Sail River, except from the F-Tile Ethernet driver support, which is documented separately.

2.0 System Software Architecture

The overall system software architecture is shown in the figure below. The below diagram shows the SW link between three different blocks available in the system. The link between the server and the Agilex SoC is enabled via a 25G Ethernet link. The 1G Ethernet interface of the Agilex SoC is used to interconnect with remote PCs for debugging purposes.

Based on the setup, the software is further subdivided into three main sections namely SOC application, MATLAB application and Server application.

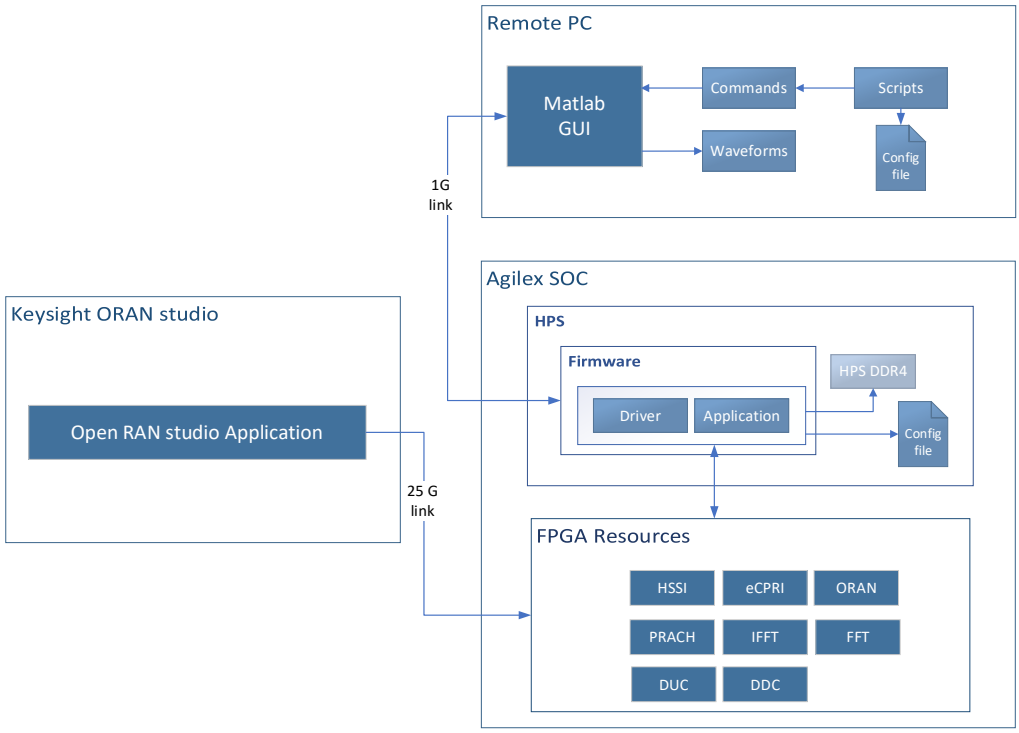


Figure 2-1 System Software setup

Note: Here the main scope is to cover the SoC and MATLAB SW architecture.

At a high level, the ORAN packets including the beamforming weights are transmitted by the server application to the Agilex SoC.

The SoC application is mainly used to configure the Phips Peak system based on the user input which will be triggered by the MATLAB application through a series of commands. The communication between MATLAB and the SoC application is enabled via 1G socket connection, where the SoC application acts as a client and the MATLAB application acts as a server. MATLAB application can also be used for analyzing data at different stages of the system where required data will be captured and plotted for further analysis and debugging.

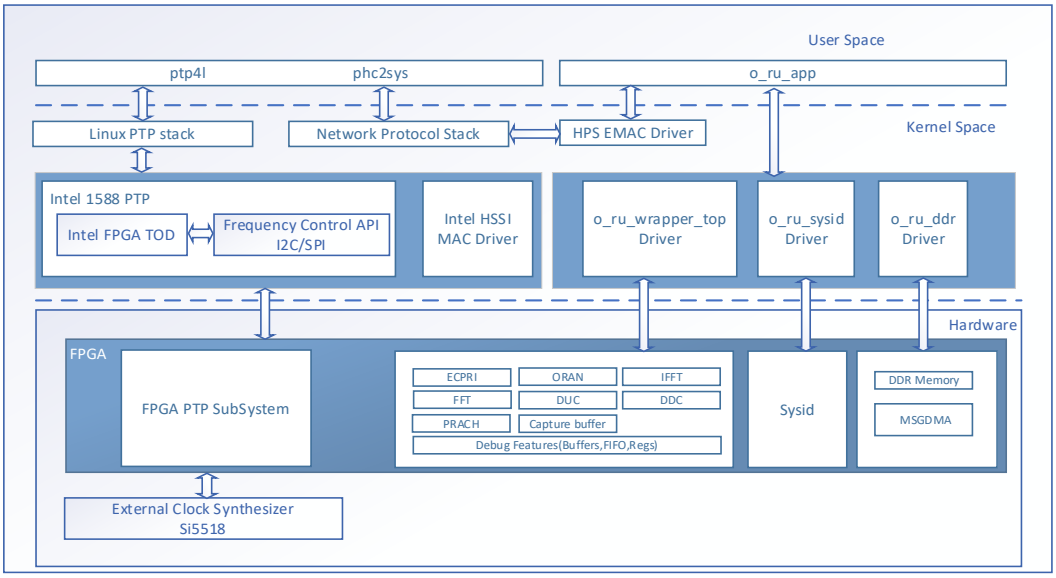


Figure 2-2 Software Architecture

The Software Architecture has User space, Kernel space, and interface to deal with Hardware part. The User Space Application has o_ru_app, ptp4l and phc_ctl application. o_ru_app uses Network protocol stack from Kernel space for TCP socket communication. Ptp4l uses the Altera 1588 PTP driver from kernel space for clock and frequency adjustment.

The O_RU_sysid_driver has resources for sysid from FPGA Hardware

The O_RU_wrapper_top_driver has resources for Radio config, eCPRI, ORAN, IFFT, FFT, DUC, DDC, PRACH, capture buffer etc. from FPGA Hardware.

The O_RU_cap_buf_driver has resources for Capture memory and MSGDMA from FPGA Hardware.

The above three drivers are used to read and write resources in sysid, wrapper top and cap_buf/On-Chip RAM modules.

3.0 SoC Firmware

Firmware architecture consists of two layers. User space and Kernel space to access the Hardware layer. The user space has the SoC application which will access the HW resources through kernel drivers using IO system calls.

Each configurable resource available in FPGA will have its own kernel driver and user space application. For example, to configure 25Ethernet MAC IP a separate kernel and user application will be created. The address maps of each resource will be registered in the kernel driver and a platform and character (file system) interface will be created for easy access from the user space where a virtual memory map will be created.

Further, the user application has a Socket layer to communicate with the remote PC. The command decoder will decode and check for the commands received from PC application with MATLAB User Interface and forward it to the appropriate resource handlers based on the resource ID (i.e., eCPRI/ORAN/PRACH/etc.) received as part of the command.

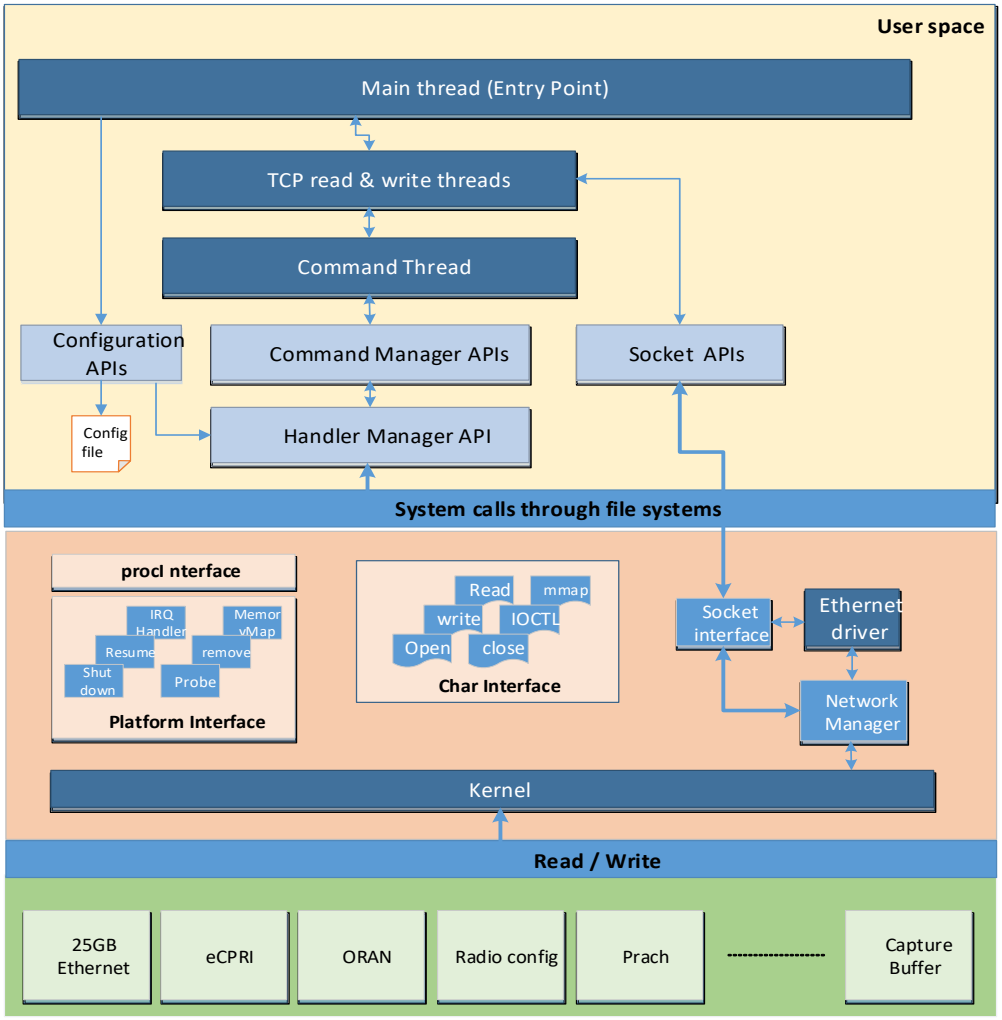


Figure 3-1 SoC Firmware Architecture

3.1 Kernel-Driver Interface

The driver is registered as a platform driver with the compatibility "altr,<module-version". The kernel relates this name with the compatibility parameter in the device tree and the driver is loaded for the device which creates a character interface and a /proc entry to access the resources from user space.

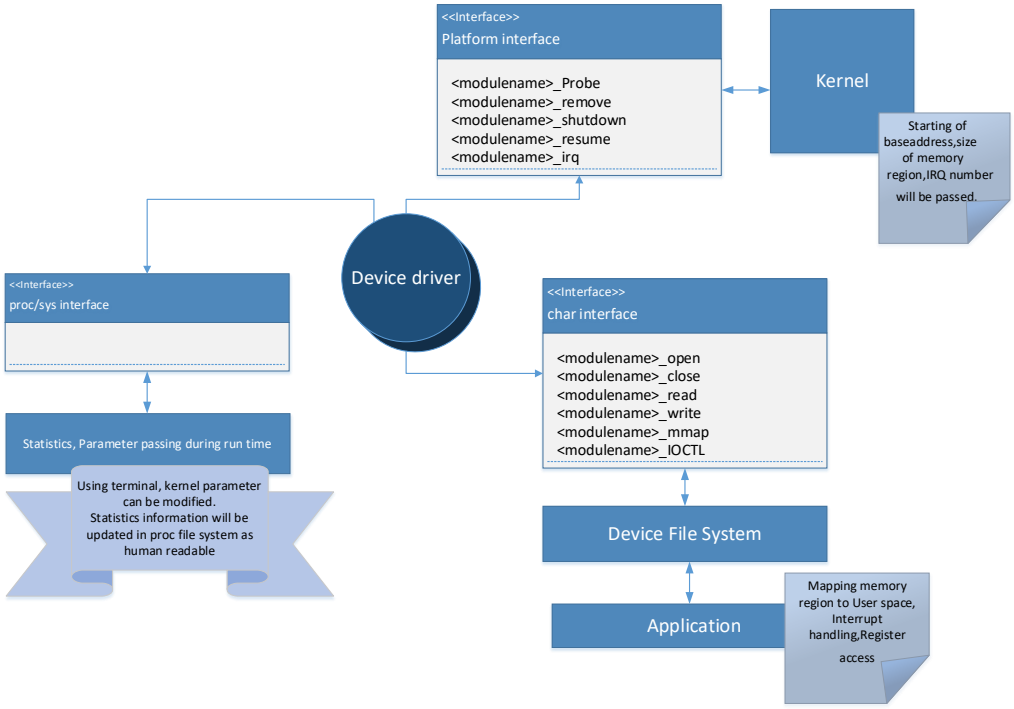


Figure 3-2 Interfaces of a Kernel Driver

The resource information is derived from the device-tree reg fields. An example of sysid module from a device tree is shown below.

```
sysid: sysid@0x240000 {

        compatible = "altr,sysid-19.1.0", "altr,sysid-1.0";

        reg = <0x00240000 0x00000008>;

        clocks = <&clk_100>;

/* embeddedsw.dts.params.id type STRING */
```

```
id = "-1395275009";

/* embeddedsw.dts.params.timestamp type NUMBER */

timestamp = <1604516279>;

};
```

Note: The device tree needs to be generated from the sopcinfo file which is an outcome of qsys block generation.

3.1.1 Platform Interface

The platform probe is responsible for probing the device tree to register character driver and allocate the memory space for the resources. The driver creates proc and character interfaces for each instance of the resources and adds it to the Kernel. Then it stores the memory pointers into the common device structure.

The remove function is called when the module is to be unloaded. The character device region allocated will be unregistered when the device driver is removed from the kernel. The resources are unmapped, and the character device structure is deleted. When the last instance of the device is removed, all the proc entries inside the /proc/<module> are removed.

3.1.2 Character Interface

Each device is registered with the kernel in the module name of device name along with the instance number of the device appended to it. It exposes certain APIs to kernel to access the hardware. Kernel APIs to access the device node files are added as part of file_operations structure.

```
static struct file_operations <module_fops> =
{
   .owner = THIS_MODULE,
   .open = <API_open>,
   .mmap = <API_mmap>,
   .unlocked_ioctl = <API_ioctl>,
   .release = <API_release>,
   .llseek = NULL,
};
```

Open:

This function is called when a process tries to open the device structure to access respective character driver.

Ioctl:

After opening the device node files, the application will interact with the driver using ioctl calls to get to know about the information like,

Number of resources available for each instance of the device

The physical address and the size (in bytes) of each resource

Mmap:

Once the user space application gets the physical addresses and size of each resource for the device, it requests a virtual memory and maps it to the user space address. So that application can access the hardware from user space.

Close:

The data that is initialized during the open call will be set to NULL in this function.

3.1.3 Proc Interface

The proc interface is used to read from and write into the resources and exposes the values in each memory of the register addresses from user space.

The directory "/proc/<module>" has num_of_instance that exposes the number of device instances probed. Based on the "num_of_instance" parameter, it will have many entries from 0,1,2 etc to represent individual instances. Under these entries, the address map of each resource will be mapped.

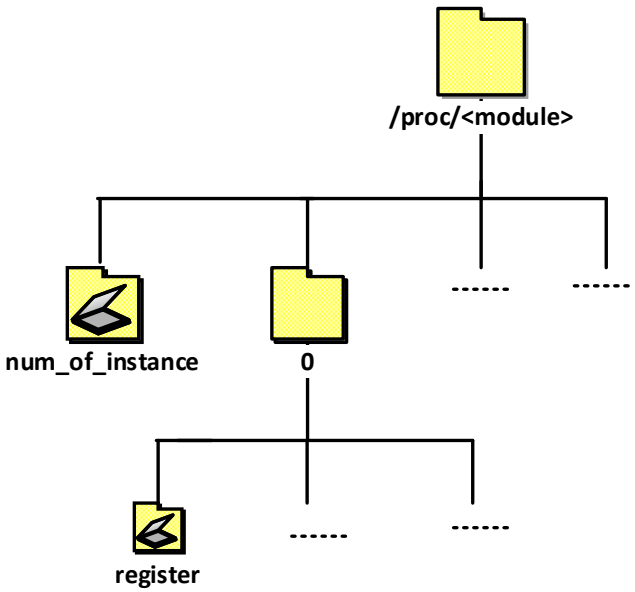


Figure 3-3 Proc Structure

3.2 Kernel Driver

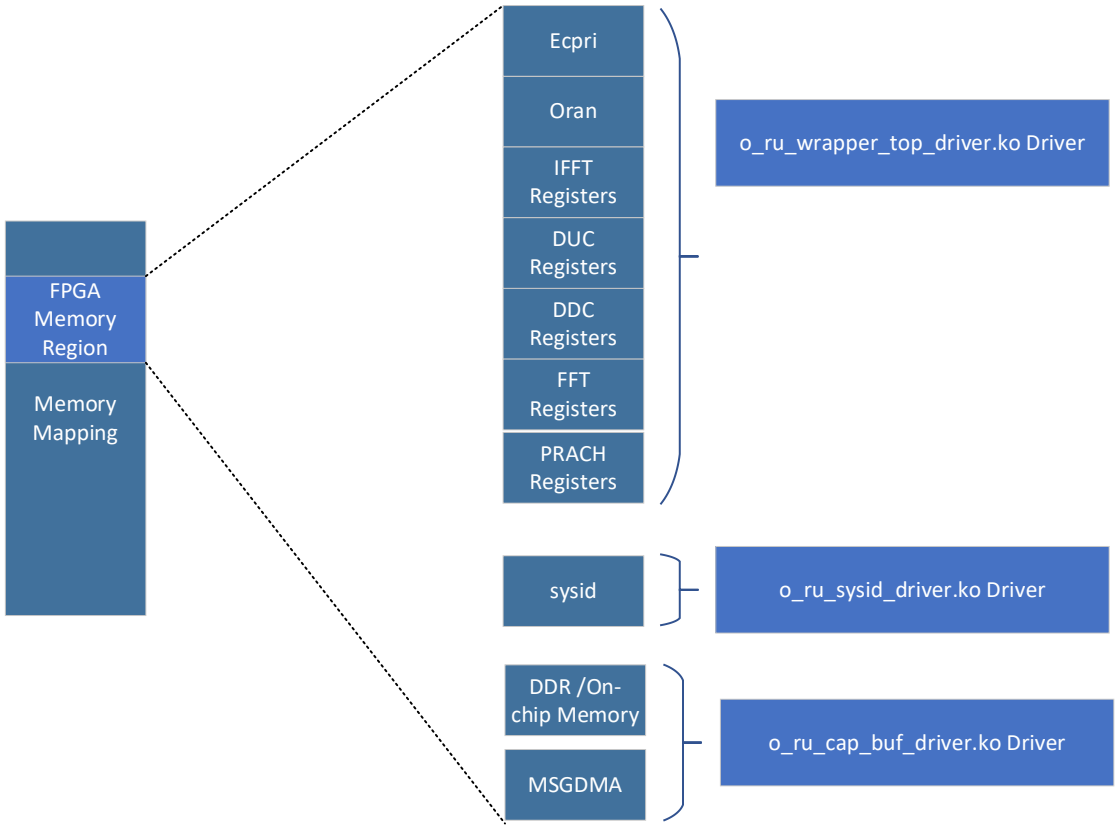


Figure 3-4 Kernel Drivers

The FPGA Memory Region consists of Radio config, eCPRI, ORAN, IFFT, FFT, and PRACH registers for configuration. o_ru_wrapper_top_driver.ko Driver is a single integrated Kernel Driver for all these mentioned resources.

It has sysid register which can be accessed from o_ru_sysid_driver.ko driver. It also has DDR/On-Chip RAM memory and msgdma register which can be accessed from o_ru_cap_buf_driver.ko driver.

3.3 User Application Interface

The User Application has the TCP/IP communication protocol implemented for MATLAB communication and Resource handler for all modules and in between it has command decoder which decodes the commands based on operation type. The Scope of the application is for two purposes.

- The resource can be read or written using the stand-alone application.

- The resource can be read or written based on the commands from MATLAB like set, get through the socket communication.

TCP/IP: The client application follows the steps below to establish connection with server.

Socket, Connect, Send/Recv

The first steps of socket communication are creating the socket with socket () call. Three parameters are needed to be specified domain, type, protocol. (AF_UNSPEC, SOCK_STREAM and IPPROTO_TCP) and it returns the integer value socket descriptor.

The second step is connect () system call. Here we need to specify the ip address and port number of the server that we are going to connect with. It will return the value of successful connection.

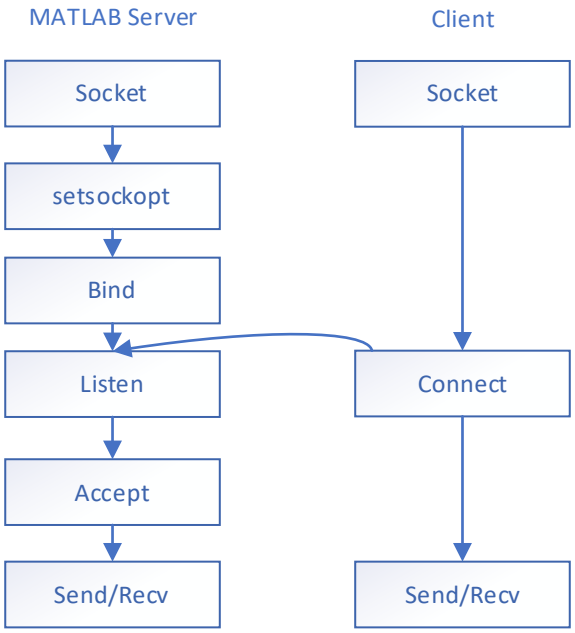


Figure 3-5 TCP Communication

The User Application runs as the client when "./O_RU _app --server "ipaddress" --port "port number"" is executed. The Application will continuously try to connect to server until the connection is established.

Two thread functions start running to transmit(transport_write_thread) and receive(transport_read_thread) TCP/IP packets from MATLAB.

Command decoder: The command decoder decodes the packets based on the command Header and do the read or write operation with the pData value on the respective module registers. The command structure can be found Command Structure

Headers	Field	Bits		Description
Packet Header	Header_String	24		This is the first field of Packet. Incoming packets will be checked for this string
	Id	8		Unique Id for each incoming command
	Length	32		This includes Command and data
	server_index	8		
	pData	8		Points to the memory that has command header
Command Header	Transfer_Type	1		This field specifies whether command is polling or service type.
	Module_Index	7		Index of module to configure.
	Operation_Type	4		Operation to be performed e.g Read, Write and Config.
	Instance	4		Instance of module to configure
	Resource	8		Index of the resource to be accessed
	Address	16		Offset address or index of configuration to be done
	Bit_Mask	32		To mask register data during write and read
	Length	22		Number of data offsets succeeding command header
	is_CPRI_CM	1		Decides if it is C&M packet
	server_index	1		
	reserved	24		
	pData	8		Points to the memory that has data

Table 3-1 Packet Headers

The o_ru_command_manager.c file has read and write function for all individual modules. Based on the Operation_Type and Modules_Index in Command Header these functions are called. Some modules have multiple resources, all resources are remapped and stored in their respective resource handler. The remapped resource in handler is chosen from the index of resource from command header and the offset is added to the resource from the Address field from command header. After the decoding the done, the data is written in the offset address from the resource. If the data is more than bytes, the offset is incremented and written.

Resource Handler: All Modules have resource handlers which have dev_name, device_file_handler, device_instance_number, reg_phy_addr, reg_remap_addr, uio_irq_device_number, ISR_thread_handler.

The dev_name is the name of the device/module created in dev with the device driver.

The device file handler has the number returned after opening the device.

The device_instance_number has the instance number for the instance in the module. Some modules have multiple instances.

The reg_phy_addr has the physical address of the resource which is read from kernel with ioctl command. The single instance in the module has many resources and each resource has a physical address.

The reg_remap_addr has the remapped address of the reg_phy_addr.

3.4 Resource Access Flow

A common flow involved in accessing any module's resources (registers/buffers) are shown below.

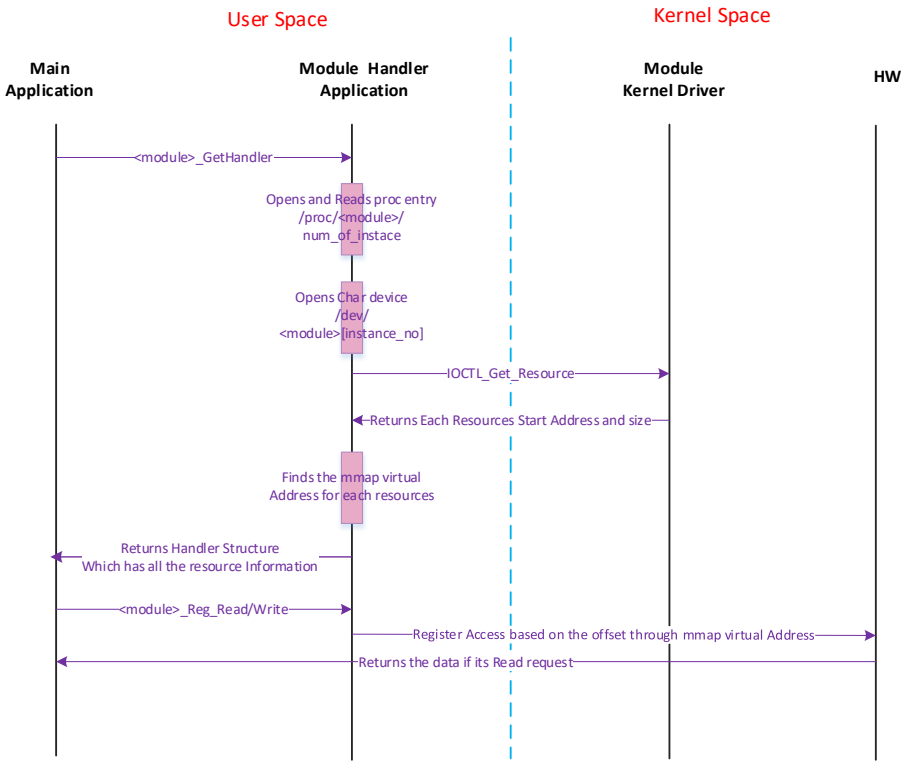


Figure 3-6 Resource access flow

3.5 DDR/On-Chip RAM Memory access through DMA

DDR /On-Chip RAM Memory is used for capture logic in design. The DDR /On-Chip RAM memory is accessed through DMA with the steps below.

- Get the Address offset of DMA CSR, DMA descriptor and DDR /On-Chip RAM memory.
- Form the descriptor with read address, write address, length, and control parameters in construct_standard_st_to_mm_descriptor () function. In this

function the read is streaming interface so read address is hard coded to 0. Write address is the pointer to the DDR /On-Chip RAM memory where the captured data is to be stored. Capture size is configured through length field denoted in bytes. In descriptor control register, "GO" bit is set to 1.

- Write the DMA descriptor using write_standard_descriptor() call. For this function call the parameters are DMA CSR base address, DMA descriptor base address and pointer to constructed descriptor.
- Wait for DMA completion status by polling read busy () call.
- After successful polling of DMA completion, descriptor is set to free,
- Read the captured data from DDR /On-Chip RAM memory to HPS memory (non-DMA operation).

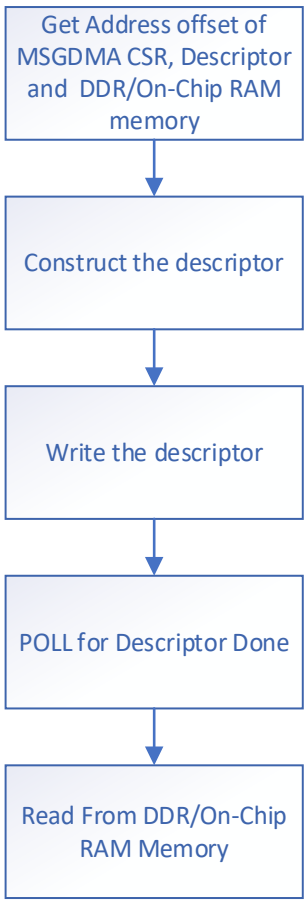


Figure 3-7 DDR /On-Chip RAM memory access through DMA flow

3.6 Interrupt Handling

The HW interrupt numbers which are mapped in FPGA for C plane timeout, U plane timeout and FIFO FULL status are 8, 9 and 7 respectively. These interrupt values indicate the position of these interrupts on the FPGA-to-HPS interrupt receiver which is a 64-bit vector.

The way Agilex FPGA interrupts translate into the HPS GIC is architecture specific. On Agilex SoC devices, the 0th bit of that vector represents interrupt 49 into the GIC of the ARM processor. In the device tree, we must subtract off the first 32 interrupt locations that represent inter-processor interrupts. We can define an equation like this to represent these FPGA driven interrupts in the device tree, FPGA-interrupt-vector + 49 - 32.

Hence, the interrupt entries for C plane timeout, U plane timeout and FIFO FULL status in the device tree are 25, 26 and 24 respectively. Since the interrupt control and status registers are implemented in Radio Config registers of wrapper_top module, the corresponding kernel ISRs are developed in the O_RU_wrapper_top_driver.c.

There are three main registers defined for interrupt handling.

1. Interrupt Mask -> control to enable (Mask:0) or disable (Mask:1) the interrupts.
2. Interrupt clear -> Clear the interrupt if that is serviced.
3. Interrupt Status -> Status of interrupt occurrence. It’s a sticky register which will be refreshed with a new status once cleared.

In driver, the interrupt numbers are retrieved through platform_get_irq and then the corresponding interrupts are registered with the service routines. Once the interrupts are triggered, the corresponding ISR will be called. In the ISR, the clear and mask bits are set until the action for that ISR is taken. Basically, this needs to be notified to the User application to send interrupt status to the monitoring thread. Once the thread captures the correct information, the mask bit for the IRQ will be cleared and hence the same routine continues.

4.0 MATLAB Firmware

This section covers how the Configuration and Register Read/Write commands issued from a Windows/Linux application, typically MATLAB, are handled in the underlying Library, transferred to SoC and response from firmware running in SoC is returned to MATLAB application. Commands are used for configuring, reading status, and receiving alarms from SoC Design.

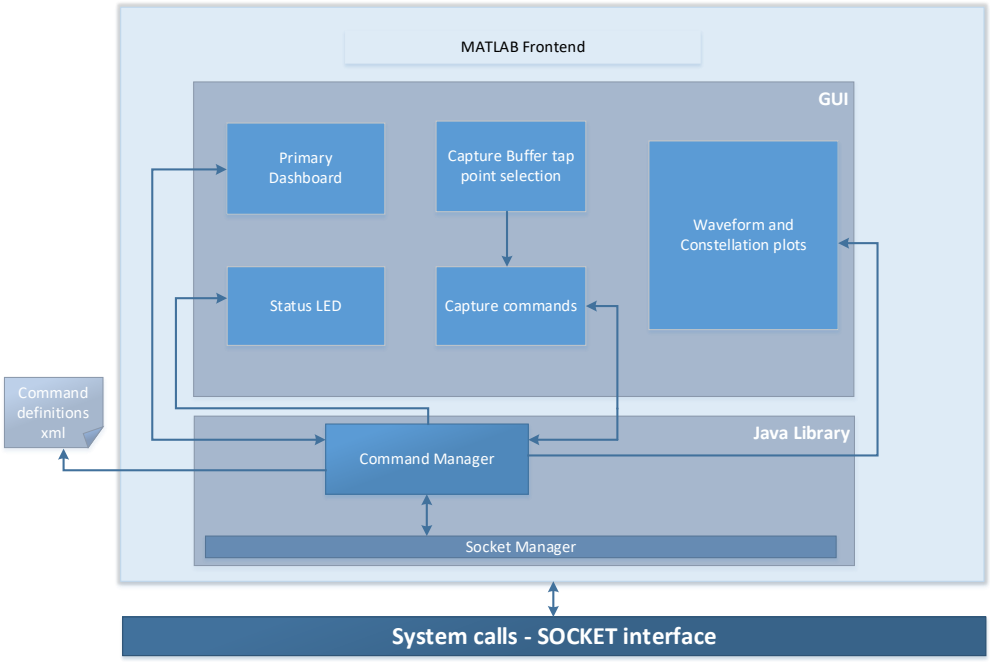


Figure 4-1 MATLAB Firmware Architecture

The GUI application interacts with the library to open a TCP/IP server socket for establishing connection with the client application running in SoC. The port number of the socket can be configured while creating a socket. Once the client application establishes connection with the socket created in the library, commands can be exchanged. The library is implemented in Java and compiled with JDK 1.8.0.45.

The Application imports the library and calls the APIs for creating TCP/IP Socket for communication with client, sends and receive commands to and from the client application running in SoC.

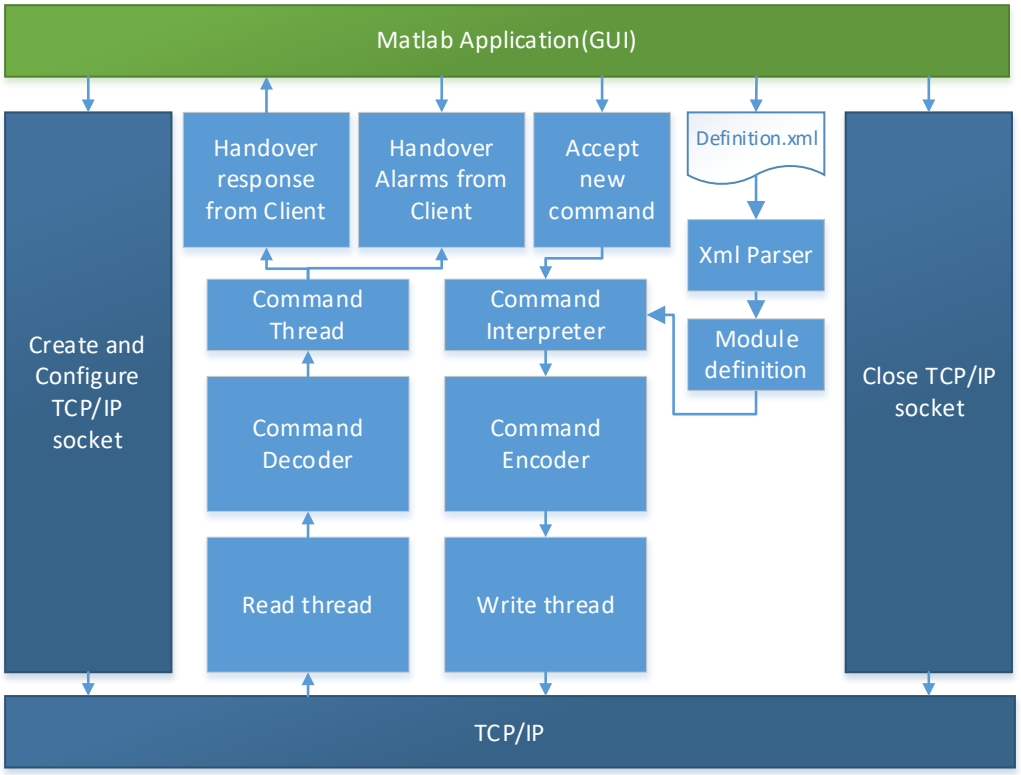


Figure 4-2 MATLAB library architecture

The SoC design and the module register definition is handed off to the library using Definition document in the xml format. XML parser in the library parses the definition document for getting SoC design knowledge like list of modules available in SoC, list of instances of each module, list of AVMM interfaces in each module, list of registers in each module, list of possible values in each bit group and its behavior. Command interpreter is generic across projects and the SoC design is extracted from definition document. Interpreted commands are encoded and sent to the SoC application through TCP/IP protocol. Read thread waits for response from SoC application and passes the response data to the top application.

5.0 Command Structure

Commands to be issued to the library from the top application need to follow the syntax which enables the Command interpreter to interpret the command correctly, encode command string to equivalent bytes and transfer to the Client application running in SoC over TCP/IP.

The client/Soc application decodes these commands and routes to the respective module for appropriate action.

Command string starts with Operation name which defines the type of action needs to be performed. The command types are defined in command.xml.

The list of possible operations/command names are,

Operation	Usage
set	Write one or series of registers/bits with value or array of values. Command should specify the module, instance, resource, register and bit name to be accessed. Response to the command will be sent immediately from the Client application.
get	Read one or series of registers/bits and return value or array of values. Command should specify the module, instance, resource, register and bit name to be accessed. Response to the command will be sent immediately from the Client application.
config	Custom commands to each module. This can be used to perform series of register access within same command. Command should specify the module and instance name. The resources and register to be accessed for each custom command within a module should be known to the command handlers in the application running in SoC. Response to the command will be sent immediately from the client application.
service	Custom commands to each module. This is like config command except that the response for the service command can be received from the Client application asynchronously until the service is stopped from the Server application.

Table 5-1 List of Commands

5.1 Mapping SoC Device Information to Commands

In SoC, the list of all modules integrated is listed in the device tree. One such example is shown below.

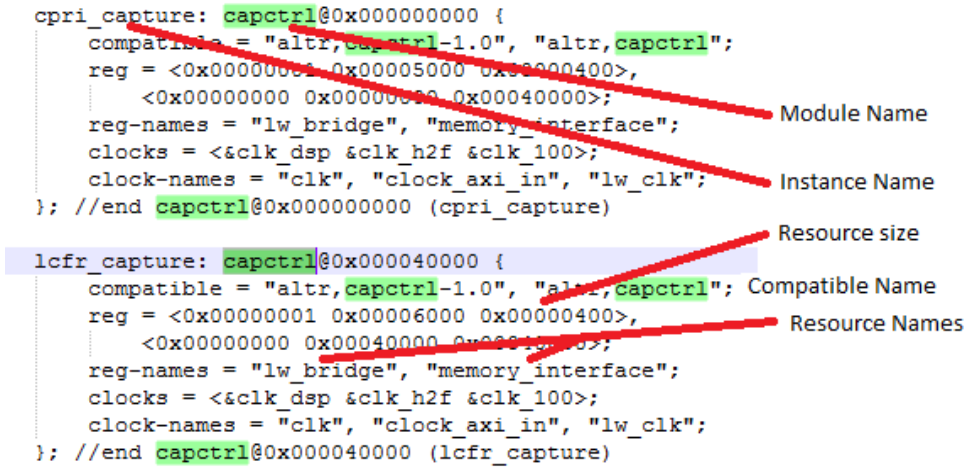


Figure 5-1 device tree structure

This structure needs to be understood by both soc and MATLAB application so that, it will decode or encode the same respectively in the form of commands.

capctrl is a module that is instantiated multiple times. Each instance is referenced by its name say cpri_capture, lcfr_capture, etc. Each instance of capctrl module has two Avalon Memory Mapped slave interfaces. lw_bridge for configuring capture logic and the other is the buffer that holds captured samples.

A write/read command needs to specify the module needs to be accessed. If the module is instantiated more than once in the Qsys, then the command needs to mention which instance needs to be written/read. If a module has more than one Avalon Memory mapped slave interface, i.e., resource, then the command needs to mention which resource needs to be accessed. Along with the above information, address, and the data to be written should also be embedded in the command.

Command supports writing/reading many consecutive registers. In that case command should specify the start address and the length of data to be written/read.

5.2 XML definitions

Modules instantiated in SoC and the register address details of each Avalon Memory Mapped interface in the modules is handed over to the library using Definition XML file. The structure of the Definition XML file is shown below.

MATLAB library has an XML parser that parses the Definition XML to get the details and the Command Interpreter reads the incoming command and encodes as a packet to be sent over TCP/IP. The SoC application receives these packets, decodes, and stores the resource information into predefined structure to write or read a configuration.

SoC Definition XML contains the details of all modules, number of instances of each module, number of Avalon MM slave interface in each module i.e., resources, details of registers in each resource, bit description of each register and valid values in the bit. Code Automation Tool makes use of Device tree generated from sopcinfo to get the modules name, instances of each module and resources in each module. Register description of each resource is extracted from FID document.

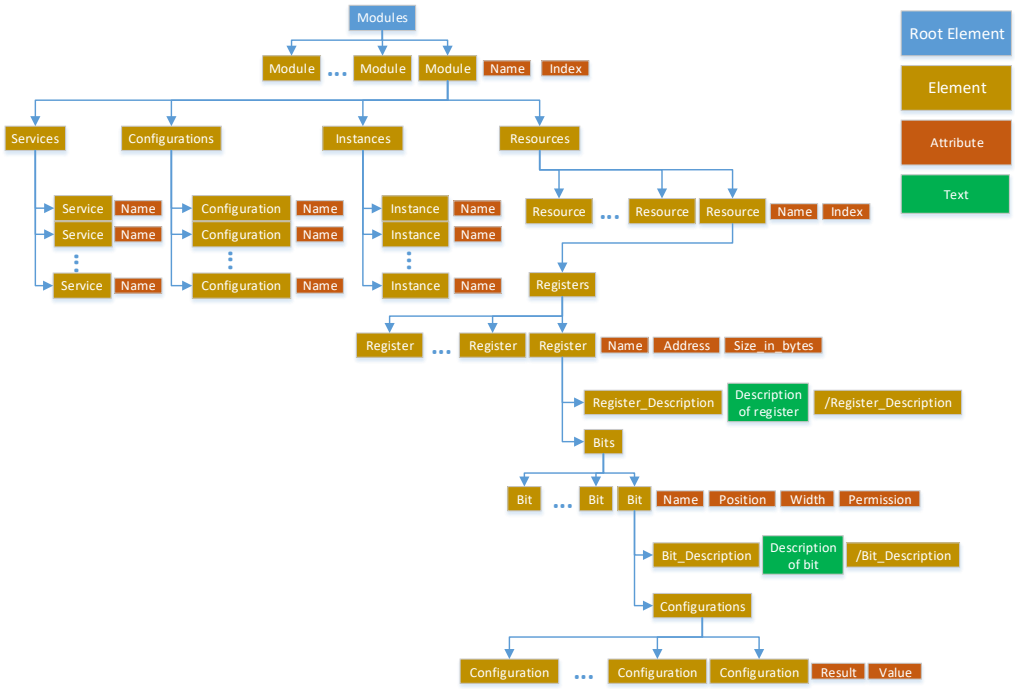


Figure 5-2 SoC Definition XML structure

Modules:

The root element of XML is Modules.

Modules element has one or many Modules element. Each Module element denotes a unique module instantiated in Qsys.

Each Module element has Name and Index attribute. Name attribute is the module name used in RTL. The index attribute mentions the numerical value that will be used as the module number to refer to.

Instances:

- Instances element for each Module element holds the name of each instance of that module in the Qsys. Instances element has one or many Instance element.
- Each Instance element denotes an instantiation of a module in Qsys.

Resources:

- Each Module has one Resources element which holds the information of Avalon Memory Mapped Slave interface. Resources element may have one or more Resource element.
- Each Resource denotes one Avalon Memory Mapped Slave interface.
- The resource element has Name attribute which is the name of AVMM interface mentioned in the component_hw.tcl file. The index attribute is the resource number that will be encoded in the commands corresponding to that resource.

Registers:

- Each Resource element will have Registers element that may have one or more Register elements.
- Each Register element denotes a 32bit register in the SoC.
- Each Register element has a Name attribute which resembles the register name used in the module. Address attribute is the read/write address in Avalon Memory Mapper Slave interface.
- Bits element has one or more Bit element. Each Bit element denotes one or a series of bits combined for same logic. For example, Mux selection may have one, two or more selection bits in a 32-bit register. This is combined as one Bit element. Bit element enables bit access commands from application. Bit element has Name, Position, Width and Permission attributes. Name attribute is the reasonable name for the bit or group of bits. Position element specifies the bit position in the 32bit register. Position starts from 0th indexing. Width attribute specifies the number of bits the Bit element is valid in the 32bit register.
- Bit element has Bit_Description element which describes the bit functionality. The configurations element of each Bit denotes the possible values and functionality corresponding to each value. Each Configuration element has a Result attribute which specifies the functionality and Value attribute specifies the value to be programmed in the Bit to achieve the functionality in the Result attribute.

Now the possible examples of command structures are explained below.

- Single Register Write
  - "set module_name instance_name resource_name register_name <bits> configuration_value"
  - It's possible to configure the whole 32-bit value if bits is not specified.
- Consecutive Writes
  - "set module_name instance_name resource_name register_name length_in_words array_of_values"
- Single Register Read
  - "get module_name instance_name resource_name register_name <bits>"
  - It's possible to read the whole 32-bit value if bits is not specified.

- Consecutive Reads
    - "get module_name instance_name resource_name register_name length_in_words"
    - Read data will be returned to the Application as a return argument.
- Config Command - send some values to the SoC or receive some values from SoC or both
    - "config module_name instance_name config_type length_in_words array_of_values"
    - The SoC application can send data back to the Library/Application by overwriting the received data. This data will be returned to the Application as return argument. The length_in_words should be set to the number of values to be sent/received from SoC. The config command handler in the SoC should make sure it doesn't write locations more than the length sent.
    - "config module_name instance_name config_type length_in_words"
    - The above command sends the length_in_words to zeroes. The application in SoC should receive configuration command and act accordingly.
- Service command structure is same as config command structure. This is mainly used for receiving results from the SoC to MATLAB application asynchronously.

6.0 List of Software

The table highlights the list of all software running in kernel, user space of SOC and PC.

System	Type	Name/Example	Description	Owner
SoC	Kernel Driver	O_RU_sysid_driver.c	System Id of the FPGA design	Altera
SoC	Kernel Driver	O_RU _wrapper_ top_driver.c	A top driver which contains the modules including "ecpri_csr", "oran_csr", "ifft_csr", "fft_csr", "Prach", etc.	Altera
SoC	Kernel Driver	O_RU _ cap_buf _driver.c	A CAP_BUF/On-Chip RAM memory driver which contains the modules including "cap_buf_mem" and "msgdma"	Altera
SoC	Kernel Driver	Altera F-Tile TOD Driver	F tile Ethernet driver to handle ptp packets (Phase II testing with 1588 on FPGA)	Altera
SoC	OS	Linux Kernel	6.1.38-lts (Linux Kernel version upgrade to be finalized)	Open Source / Altera
SoC	Other SD image binaries	dtsi, u-boot, preloader etc.	To be referred from Golden software Reference Design (GSRD) binaries	Altera
SoC	User Application	o_ru_sysid_app.c o_ru _sysid.c o_ru _sysid.h	SysId application	Altera
SoC	User Application	o_ru _eth_1588_tod_app.c	An application to reset frame sync counters based on TOD.	Altera
SoC	User Application	o_ru _wrapper_ top_app.c o_ru _wrapper_ top_debug_app.c o_ru _wrapper_top.c o_ru _wrapper_top.h	Top level application to handle "radio_config_csr", "ecpri_csr", "oran_csr", "cb_st_csr", "cb_st_mem", "Prach", etc. Also, to handle debug features including fifo monitoring and packet profiling	Altera
SoC	User Application	o_ru _cap_buf_app.c o_ru _cap_buf.c sgdma_dispatcher.c o_ru_cap_buf.h alt_types.h csr_regs.h descriptor_regs.h io.h response_regs.h	Top level application to handle "cap_buf _mem" and "msgdma" resource and to do captures from capture points.	Altera

		sgdma_dispatcher.h		
SoC	User Application	o_ru _tcp_client.c o_ru _protocol.h	Files to enable 1G Socket Communication - client	Altera
SoC	User Application	o_ru _mem_access.c	File for common reg write and read of all modules	Altera
SoC	User Application	o_ru _commandmanager.c o_ru _commandmanager.h o_ru _command_format.h	Files to decode command structures	Altera
SoC	User Application	o_ru _main.c o_ru _firmware.h	main application and the structure definitions to store device information	Altera
SoC	User Application	spidev_test	spi based applications to configure clock chips (Si) etc. (Phase-II)	Altera
SoC	Other files	prod_fw.boot.bin user_config.boot.bin	configuration files	Altera
SoC	User Application	ptp4l, phc_ctl	PTP synchronization application	Open source
PC	User Application	JAR library	1G Socket Communication - Server (Released as a java library instead of source)	Altera
PC	User Application	GUI, m scripts, xml, mat files etc	MATLAB GUI with predefined command sets, config/xml files, waveforms etc.	Altera
SoC	User Application	o_ru_api.c o_ru_powermeter_api.h o_ru_powermeter_api.c	Power Meter API to configure various control and status registers, manage memory configurations, and facilitate starting and stopping the power meter in both single and continuous modes.	Altera

Table 6-1 List of SW Applications

7.0 List of APIs

7.1 Generic APIs for Read and Write

APIs	Description
short O_RU_reg_read32 (**unsigned int** module_index, **unsigned int** *OutPtr, **unsigned int** Length, **unsigned int** offset, **unsigned int** resource_index);	Generic API used for reading all the 32-bit registers/buffers. Returns 0 on success or 1 on failure. OutPtr -> should be a memory allocated pointer to hold the read data with specified length. Length -> count of 32-bit data to read. (Ex-1,2, etc.) offset -> address offset (0,1,2 etc.) from the base address. Base address will be identified by the application. module_index -> Index of the module to be accessed. resource_index-> Index of the resource to be accessed within a module. **Note:** It's user's responsibility to provide valid parameters considering R/W permissions
short O_RU _reg_write32 (**unsigned int** module_index, **unsigned int** *InPtr, **unsigned int** Length, **unsigned int** offset, **unsigned int** resource_index)	Generic API used for writing all 32-bit registers/buffers. Returns 0 on success or 1 on failure. InPtr -> should be a memory allocated data pointer with specified length. Length -> count of 32-bit data to write. Maximum of 10 count is supported. offset -> address offset (0,1,2 etc.) to read from the base address. Base address will be identified by the application. module_index -> Index of the module to be accessed. resource_index -> Index of the resource to be accessed within a module. **Note**: It's user's responsibility to provide valid parameters considering R/W permissions

Table 7-1 Generic API List

Notes:

The module_index and resource_index specifies the list of all modules in the design and the resources available within each module respectively.

The module numbers are listed in the o_ru_firmware.h as shown below,
#define O_RU _SYSID_MODULE 0
#define O_RU _WRAPPER_TOP_MODULE 1
#define O_RU _CAP_BUF_MODULE 2
#define O_RU _TOT_MODULES 3

The resource numbers are listed in the respective module's header files. For example, the wrapper_top module's resources are listed in the header file o_ru_wrapper_top.h as shown below.
#define O_RU_wrapper_top_WRAPPER_TOP_INSTANCE_INDEX 0

```
#define O_RU_wrapper_top_ORAN_SS_CSR_RESOURCE_INDEX 0
#define O_RU_wrapper_top_ECPRI_CSR_RESOURCE_INDEX 1
#define O_RU_wrapper_top_ORAN_CSR_RESOURCE_INDEX 2
#define O_RU_wrapper_top_FH_COMP_CSR_RESOURCE_INDEX 3
#define O_RU_wrapper_top_PROFILING_MEM_RESOURCE_INDEX 4
#define O_RU_wrapper_top_LOWPHY_SS_CSR_RESOURCE_INDEX 5
#define O_RU_wrapper_top_IFFT_L1_CSR_RESOURCE_INDEX 6
#define O_RU_wrapper_top_IFFT_L2_CSR_RESOURCE_INDEX 7
#define O_RU_wrapper_top_FFT_L1_CSR_RESOURCE_INDEX 8
#define O_RU_wrapper_top_FFT_L2_CSR_RESOURCE_INDEX 9
#define O_RU_wrapper_top_LONG_PRACH_L1_CSR_RESOURCE_INDEX 10
#define O_RU_wrapper_top_LONG_PRACH_L2_CSR_RESOURCE_INDEX 11
#define O_RU_wrapper_top_PB_MEM_RESOURCE_INDEX 12
#define O_RU_wrapper_top_DXC_SS_CSR_RESOURCE_INDEX 13
#define O_RU_wrapper_top_DUC_L1_CSR_RESOURCE_INDEX 14
#define O_RU_wrapper_top_DUC_L2_CSR_RESOURCE_INDEX 15
#define O_RU_wrapper_top_CA_INTERP_CSR_RESOURCE_INDEX 16
#define O_RU_wrapper_top_DEC_DLY_COMP_CSR_RESOURCE_INDEX 17
#define O_RU_wrapper_top_DDC_L1_CSR_RESOURCE_INDEX 18
#define O_RU_wrapper_top_DDC_L2_CSR_RESOURCE_INDEX 19
#define O_RU_wrapper_top_DFD_SS_CSR_RESOURCE_INDEX 20
#define O_RU_wrapper_top_TOD_BUF_CSR_RESOURCE_INDEX 21
#define O_RU_wrapper_top_TOT_RESOURCES 22
```

This information can also be referenced from the SW FID register map document. In addition to that, individual address offsets and the list of register details within a resource can also be referred.

Example configuration:

For example, if a user wants to configure the wrapper_top modules eCPRI vlan_tag_0 register, the configuration values should be as follows

- Module_index – 1(O_RU_WRAPPER_TOP_MODULE)
- Resource_index – 20(O_RU_wrapper_top_DFD_SS_CSR_RESOURCE_INDEX)
- Offset in word addressing – (0x10/4 = 0x4)
- Length - 1

Subsystem	Module Name	Register Name	Address Offset	POR	Field Name	Bit Offset	Bit Width	Access	Field Description
DFD SS Subsystem	DFD SS Registers	dsp_start_capture	0x0	0	dsp_capture_start	0	1	RW	Bit[0]- Capture Start - '0' - Disables capture '1'
				0	dsp_trigger_based_capture	1	1	RW	Bit[1]- enable/disable trigger based capture ('0'-disable , '1'-enable)
				0	dsp_frame_based_capture	2	1	RW	Bit[2]- enable/disable frame based capture ('0'-disable , '1'-enable).
		dsp_ss_sel	0x4	0	subsystem_select	0	32	RW	Subsystem Select
		dsp_channel_sel	0x8	0	channel_number_select	0	32	RW	Selects a particular channel of the selcted interface.
		dsp_subframe_config	0xC	0	sample_select	0	15	RW	Bit[14:0]- selects a particular sample
				0	symbol_select	15	5	RW	Bit[19:15]- selects a particular symbol.
				0	subframe_select	20	4	RW	Bit[23:20]-selects a particular subframe
		dsp_frame_config	0x10	0	radio_frame_select	0	10	RW	Bit[9:0] - selects a particular radio frame
		dsp_interface_sel	0x14	0	interface_select	0	32	RW	Selects a particular interface .

Figure 7-1 DFD subsystem registers from FID document

7.2 Other fundamental APIs

Module	APIs	Description
o_ru_sysid	pdevice_handler o_ru**_sysid_get_handler** (**unsigned int** instance_num);	Sysid handler API used to collect and hold all the resource and address information
	void O_RU**_sysid_reg_read** (pCommand_Header pCmdHeader,**unsigned int** packet_id);	sysid read API used to read the id based on MATLAB command request
	void O_RU**_sysid_read** ();	sysid read API used to read the id from the HPS application
o_ru_wrapper_top	pdevice_handler o_ru**_wrapper_top_get_handler** (**unsigned int** instance_num);	Wrapper_top handler API used to collect and hold all the resource and address information
	void O_RU**_wrapper_top_reg_read** (pCommand_Header pCmdHeader,**unsigned int** packet_id);	A 32-bit read API specific to o_ru_wrapper_top module which will be triggered based on MATLAB command request

	void O_RU**_wrapper_top_reg_write** (pCommand_Header pCmdHeader,**unsigned int** packet_id);	A 32-bit write API specific to o_ru _wrapper_top module which will be triggered based on MATLAB command request
	void O_RU**_wrapper_top_config(**pCommand_Header pCmdHeader,**unsigned int** packet_id);	Config API specific to o_ru_wrapper_top module which will be triggered based on MATLAB command request
	void O_RU**_wrapper_top_init** ();	API to initialise the wrapper top related functionalities
	unsigned int O_RU**_FIFO_status_read** ();	Reads the FIFO status register and returns the value
	void O_RU_tod_reset_counters (**int** alpha, **int** beta);	Resets the frame sync counters with respect to the radio frame boundary
	void o_ru**_pkt_profiler** (uint8_t dl_ul, uint32_t pkt_count);	Configures the dl/ul profiling mux and Triggers dumping of the packets. Reads the FIFO based on the rdusedw until the count specified. After reading, write the contents into a .txt file and .csv file.
	void cu_plane_connection_monitoring_timer (uint8_t new_msec)	By default, C/U plane connection is monitored every 160msec. This function allows the monitoring timer to be configured with new timer value
	void cu_plane_connection_disable(void**)**	Disable the operation of C/U plane monitoring
	int O_RU_wrapper_top_capture_config(**unsigned int** bw_sel, **unsigned int** select_lpbk, **unsigned int** capture, **unsigned int** axc, **unsigned int** sfn, **unsigned int** symbol)	Captures the data based on the selection and store it in text file.
	int check_regdefault ()	Checks for the register read (POR) value for all registers
o_ru_cap_buf	pdevice_handler **o_ru_cap_buf_get_handler** (**unsigned int** instance_num);	CAP_BUF/On-Chip RAM handler API used to collect and hold all the resource and address information
	int init_dma_toread_from_emif (**unsigned int** *pData, **unsigned int** capture_sel, **unsigned long** Length,**unsigned int** cap_buf_ocm_sel);	Wrapper Function for construct and write the descriptor
	int read_dma_status(**unsigned int** *pData);	Reads DMA done status
	int read_from_emif_without_dma (**unsigned int** *pData, **unsigned int** capture_sel, **unsigned long** Len**, unsigned int** cap_buf_ocm_sel);	Reads the data from CAP_BUF /On-Chip RAM

	int construct_standard_st_to_mm_descriptor (sgdma_standard_descriptor *descriptor, alt_u32 *write_address, alt_u32 length, alt_u32 control);	Function to form the descriptor.
	int write_standard_descriptor (alt_u32 csr_base, alt_u32 descriptor_base, sgdma_standard_descriptor *descriptor);	Function to write the descriptor into the DMA.

Table 7-2 Other APIs List

7.3 Capture API

int O_RU_wrapper_top_capture_config(unsigned int bw_sel, unsigned int select_lpbk, unsigned int capture, unsigned int axc, unsigned int sfn, unsigned int symbol, unsigned int cap_buf_ocm_sel)	Generic API used to capture data based on the selection. Returns 0 on success or 1 on failure. bw_sel -> should be either 100 or 60. select_lpbk -> 0 - ifft/fft lpbk , 1- DFE loopback. capture -> capture point to be selected. IFFT input - 0 IFFT output - 1 DUC output - 2 Summer output - 3 UL mixer output - 4 DDC output - 5 FFT output – 6 PRACH output -7 axc-> axc number to be captured. 0 - CC1, TRX1 1 - CC2, TRX1 2 - CC1, TRX2 3 - CC2, TRX2 4 - CC1, TRX3 5 - CC2, TRX3 6 - CC1, TRX4 7 - CC2, TRX4 sfn -> subframe to be captured. Varies from 0 to 9. symbol-> subframe to be captured. Varies from 0 to 27. cap_buf_ocm_sel ->2-cap_buf capture,1-ocm capture Note: It's user's responsibility to provide valid parameters considering R/W permissions

Table 7-3 Capture API

Example configuration:

For example, if a user wants to capture 100 MHz, DFE loopback, FFT output, CC1 and TRX2, 8th subframe and symbol 7 data the configuration values should be as follows.

- bw_sel - 100
- select_lpbk -1
- capture - 6
- axc - 2
- Sfn -8
- Symbol-7
- cap_buf_ocm_sel -1

```
login as: root
root@agilex:~# cd oru
root@agilex:~/oru# ./o_ru_app --capture 100 1 6 2 8 7 1
Bandwidth_select--> 100
Loop_back_select--> 1
Capture_select--> 6
AXC_select--> 2
Subframe_select--> 8
Symbol--> 7
cap_buf_ocm_select--> 7
Capturing TRX 1 of CC:1
FFT output capture started
sym_15= 9
 sample_15= 2291
 cap_subframe= 8ss_sel:0
 axc_sel:1
 sub_sel:8685811
 axc_sel:0
int_sel:4
capture_length:3276DMA init done
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 1
dma status in capture : 1
DMA status Wating... : 1
DMA status: 0
dma status in capture : 0
CAP_BUFF read done
Captured loopback
 main : 572 Running in standalone mode
```

Figure 7-2 Capture API

NOTE:

1. Sail River supports compile time number of cc selection . If number of cc_selection is 1 axc 1 3 5 and 7 should not be configured.

For example, if a user wants to capture 100 MHz, DFE loopback, IFFT input, CC2 and TRX2, 0th subframe and symbol 0 data the observation will be as mentioned in figure.

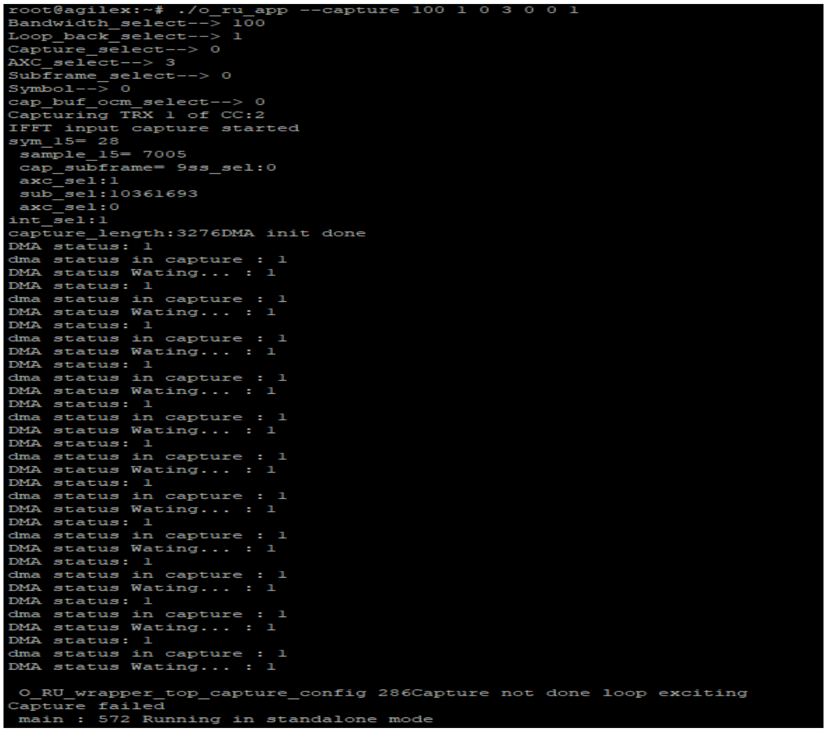


Figure 7-3 Capture API

7.4 Power Meter API

APIs	Description
oru_ErrAction_e oru_PwrMtrSsCtrlRegSetMode (const uint32_t pm_type, const uint32_t set_mode)	API to configure power meter mode(single/continuous). pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 set_mode[in]: Provide power meter mode type (1 – Single, 2 – Continuous). On successful, return ORU_SUCCESS (1)

oru_ErrAction_e oru_PwrMtrSsCtrlRegStartNStop (const uint32_t pm_type, const uint32_t start_n_stop)	API to start and stop power meter. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 start_n_stop[in]: Provide power meter start and stop. 1 -> Start 0 -> Stop On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrSsCtrlRegIrqEnableNDisable (const uint32_t pm_type, const uint32_t irq_enable)	API to configure power meter IRQ. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 irq_enable[in]: Provide power meter irq enable/disable 1 -> irq_enable, 0-> irq_disable On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrSsCtrlRegDLPwrEn (const uint32_t pm_type, const uint32_t dl_pwr_en)	API to configure power meter DL power enable. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 dl_pwr_en[in]: 1 -> dl power enable. On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrSsCtrlRegULPwrEn (const uint32_t pm_type, const uint32_t ul_pwr_en)	API to configure power meter UL power enable. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 ul_pwr_en[in]: 1 -> ul power enable. On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrSsCtrlRegRst (const uint32_t pm_type, bool reset)	API to clear the power meter status register. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 reset[in]: 1 -> For clearing the status register On successful, return ORU_SUCCESS (1)
uint32_t oru_PwrMtrSsStatusRegGetDataStatus (const uint32_t pm_type)	API to get memory bank of Histogram and Statistic memory where data is present.

	pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return: 0 -> memory bank 0 1 -> memory bank 1
oru_ErrAction_e oru_PwrMtrSsSetComputeTime (const uint32_t pm_type, const uint32_t compute_time_ms)	API to configure compute time in milliseconds. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 compute_time_ms[in]: Provide Power meter compute time in milliseconds. On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrThreashRam (const uint32_t pm_type**, const uint32_t** start_bin**, const uint32_t** stop_bin**)**	API to configure threshold memory with bin value. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 start_bin[in]: Provide start index of bin value stop_bin[in]: Provide stop index of bin value On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrHistSingleStart (const uint32_t pw_type**, const float** compute_time_ms**);**	API to start power meter in single mode. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 compute_time_ms[in]: Provide compute time in milliseconds On successful, return ORU_SUCCESS (1)
oru_ErrAction_e oru_PwrMtrHistContStart (const uint32_t pm_type**, const float** compute_time_ms**, const uint32_t** count**)**	API to start power meter in continuous mode. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 compute_time_ms[in]: Provide compute time in milliseconds count[in]: Provide the number of iterations for continuous mode. count > 0 On successful, return ORU_SUCCESS (1)

bool oru_GetHistDone (const uint32_t pm_type**)**	API to get hist done status before reading histogram memory. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return 1 -> On success 0 -> Failure
bool oru_IsOverFlow (const uint32_t pm_type**)**	API to check for overflow condition before reading histogram/statistic memory. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return 1 -> On success 0 -> Failure
bool oru_IsUnderFlow (const uint32_t pm_type**)**	API to check for underflow condition before reading histogram/statistic memory. pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return 1 -> On success 0 -> Failure
uint32_t *oru_GetHistRamBaseAddr(const uint32_t pw_type**);**	API to get histogram memory base address pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return: On successful, return base address of histogram memory
uint32_t *oru_GetStatRamBaseAddr(const uint32_t pw_type**);**	API to get statistic memory base address pm_type[in]: Provide Power meter type: 0 ->FFTL1, 1 -> FFTL2, 2 -> IFFTL1, 3 -> IFFTL2 Return: On successful, return base address of statistic memory
void oru_PwrMtrHistRamFileWrite(const uint32_t *temp_buffer**, const uint32_t** count**)**	API to write histogram data to file. temp_buffer[in]: Provide buffer containing histogram data count[in]: 1-> single, greater than 1 -> continuous Return: NULL

void oru_PwrMtrStatRamFileWrite(const uint32_t *temp_buffer**, const uint32_t** count**)**	API to write statistic data to file. temp_buffer[in]: Provide buffer containing statistic data count[in]: 1-> single, greater than 1 -> continuous Return: NULL

Table 7-4 Power Meter APIs

Example configuration:

For example, if a user wants to start the power meter in single mode, then he needs to provide the power meter type (FFTL1(0), FFTL2(1), IFFTL1(2), IFFTL2(3)), compute time (in milliseconds) and no. of count (especially for continuous mode).

Command to start power meter in single mode:

./o_ru_app –startpm_single 2 10

- 1st argument is power meter type (IFFTL1)
- 2nd argument is compute time (10 ms)

```
root@agilex:~# ./o_ru_app --startpm_single 2 10
Power meter in single mode:
Power meter register configuration
Entering into Power meter Single mode ------------------------------------------
Step 1: Configuring Compute time
 Step 2: Configuring Threshold memory
 Step 3: Clearing the status register
 Step 4: Clearing the control register
 Step 5: Configuring mode to single
 Step 6: Power meter start
No of symbol: 280
step size 31775.031250
Step 7:
Mem bank: 1
Power meter executed successfully
 main : 728 Running in standalone mode
root@agilex:~#
```

Figure 7-4 Power meter Single Mode

Command to start power meter in Continuous mode

./o_ru_app –startpm_cont 2 10 10

- 1st argument is power meter type (IFFTL1)
- 2nd argument is compute time (10 ms)
- 3rd argument is no. of count (required for continuous mode)

```
root@agilex:~# ./o_ru_app --startpm_cont 2 10 10
Power meter in continuous mode:
Power meter register configuration
Entering into Power meter Continuous mode --------------------------------------------
Step 1: Configuring Compute time
Step 2: Configuring Threshold memory
Step 3: Clearing the status register
Step 4: Clearing the control register
Step 5: Configuring mode to continuous mode
Step 6: Power meter start
No of symbol: 280
step size 31775.031250
Step 7:
mem bank: 1
mem bank: 0
mem bank: 1
mem bank: 0
mem bank: 1
mem bank: 0
mem bank: 1
mem bank: 0
mem bank: 1
mem bank: 0
Power meter executed successfully
 main : 728 Running in standalone mode
root@agilex:~#
```

Figure 7-5 Power meter Continuous Mode

8.0 Steps to add wrapper top resource in MATLAB GUI for control tab

The control tab in MATLAB GUI is mainly used for register read and write of all modules. The control tab information can also be referenced from the VTP and VTR document.

To add resources in wrapper top module we need to make changes in following files.

1. O_RU_start.m
2. O_RU_design.m
3. O_RU_control.m

In O_RU_start.m file, add the resource name to be added at the end of app.resource.Items variable.

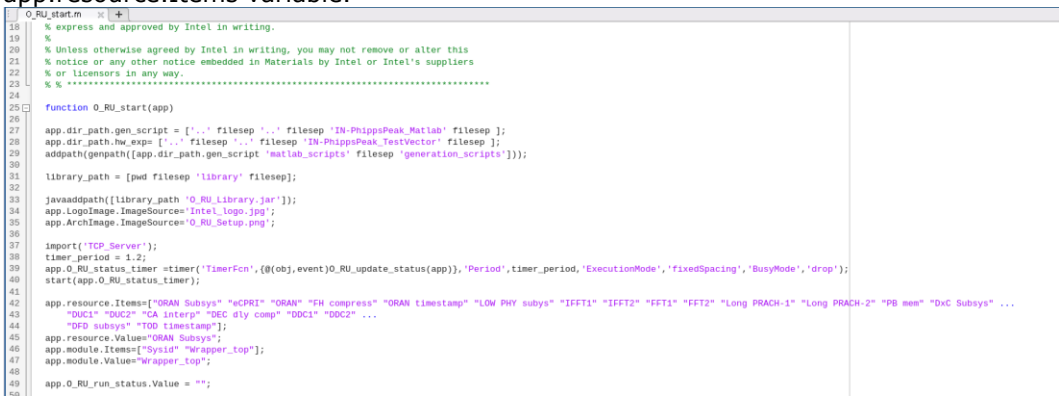
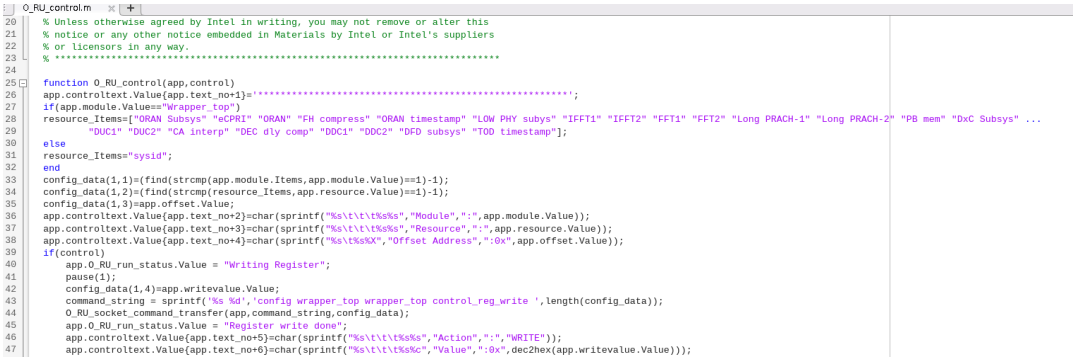


Figure 8-1 adding resource in O_RU_start.m

In O_RU_design.m file, add the same resource name at the end of app.resource.Items variable.

Figure 8-2 adding resource in O_RU_design.m

In O_RU_control.m file add the same resource name in resource_Items variable.



```
O_RU_control.m
20  % Unless otherwise agreed by Intel in writing, you may not remove or alter this
21  % notice or any other notice embedded in Materials by Intel or Intel's suppliers
22  % or licensors in any way.
23  % ***************************************************************************
24
25  function O_RU_control(app,control)
26  app.controltext.Value{app.text_no+1}='*********************************************************';
27  if(app.module.Value=="Wrapper_top")
28  resource_Items=["ORAN Subsys" "eCPRI" "ORAN" "FH compress" "ORAN timestamp" "LOW PHY subys" "IFFT1" "IFFT2" "FFT1" "FFT2" "Long PRACH-1" "Long PRACH-2" "PB mem" "DxC Subsys" ...
29          "DUC1" "DUC2" "CA interp" "DEC dly comp" "DDC1" "DDC2" "DFD subsys" "TOD timestamp"];
30  else
31  resource_Items="sysid";
32  end
33  config_data(1,1)=(find(strcmp(app.module.Items,app.module.Value)==1)-1);
34  config_data(1,2)=(find(strcmp(resource_Items,app.resource.Value)==1)-1);
35  config_data(1,3)=app.offset.Value;
36  app.controltext.Value{app.text_no+2}=char(sprintf("%s\t\t\t%s%s","Module",":",app.module.Value));
37  app.controltext.Value{app.text_no+3}=char(sprintf("%s\t\t\t%s%s","Resource",":",app.resource.Value));
38  app.controltext.Value{app.text_no+4}=char(sprintf("%s\t%s%X","Offset Address",":0x",app.offset.Value));
39  if(control)
40      app.O_RU_run_status.Value = "Writing Register";
41      pause(1);
42      config_data(1,4)=app.writevalue.Value;
43      command_string = sprintf('%s %d','config wrapper_top wrapper_top control_reg_write ',length(config_data));
44      O_RU_socket_command_transfer(app,command_string,config_data);
45      app.O_RU_run_status.Value = "Register write done";
46      app.controltext.Value{app.text_no+5}=char(sprintf("%s\t\t\t%s%s","Action",":","WRITE"));
47      app.controltext.Value{app.text_no+6}=char(sprintf("%s\t\t\t%s%c","Value",":0x",dec2hex(app.writevalue.Value)));
```

Figure 8-3 Adding resource in O_RU_control.m

9.0 Revision History

Date	Version	Changes
2025-05-22	0.0.1	Initial release.