Universal Verification Methodology

Assignment Manual

10x Engineers (Pvt.) Ltd.

Table of Contents

Overview

In these assignments you will be integrating a verification environment for a YAPP router design. Save all the work as it will be used in the preceding parts of assignments and next assignments as well. You will be integrating multiple UVC components provided to you and will use multichannel sequences and TLM.

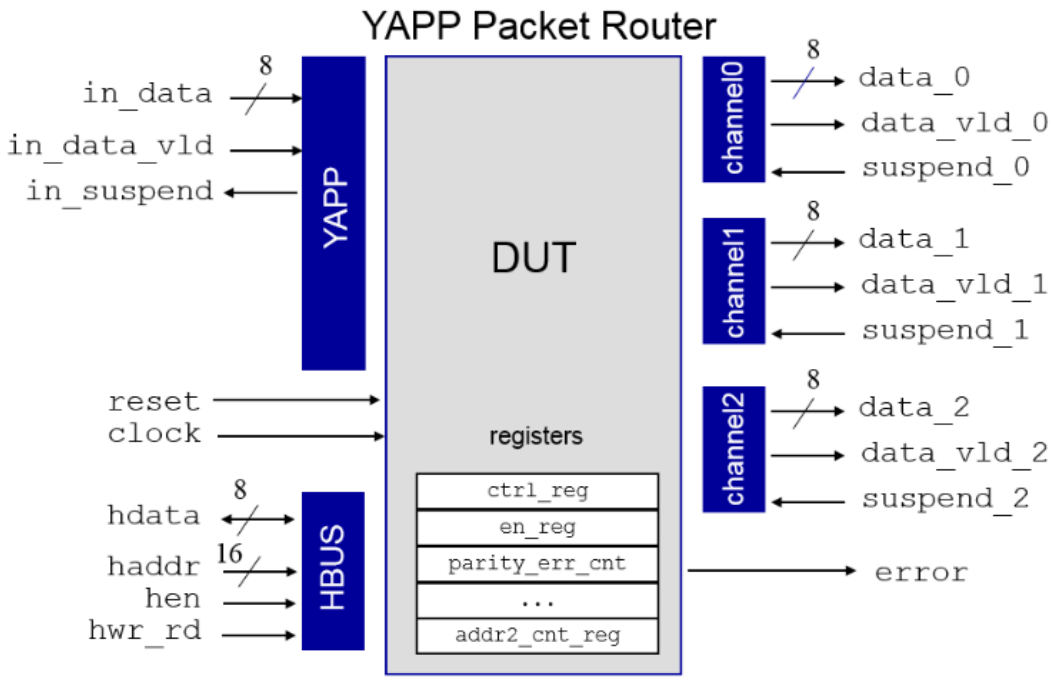
YAPP Router Specification

YAPP Router Description

The YAPP router accepts data packets on a single input port, `in_data`, and routes the packets to one of three output channels: `channel0`, `channel1` or `channel2`. The input and output ports have slightly different signal protocols.

The router also has an HBUS host interface for programming registers that are described in the next section.

High-Level Diagram – YAPP Router (Yet Another Packet Protocol)

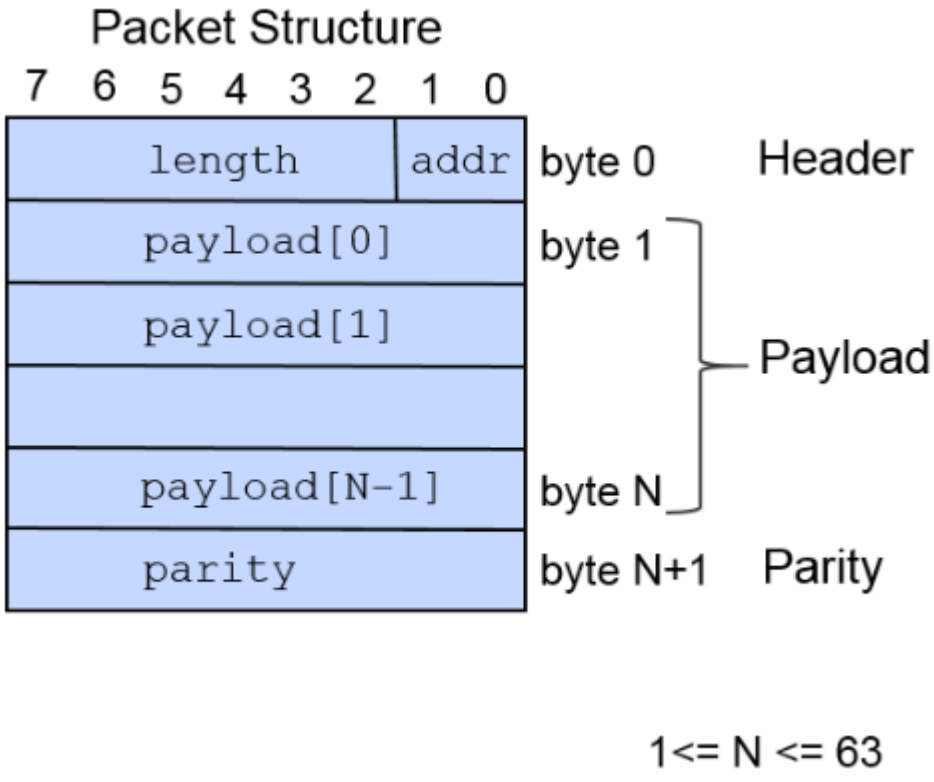


Packet Data Specification

A packet is a sequence of bytes with the first byte containing a header, the next variable set of bytes containing payload, and the last byte containing parity.

The header consists of a 2-bit address field and a 6-bit length field. The address field is used to determine which output channel the packet should be routed to, with the address 3 being illegal. The length field specifies the number of data bytes (payload).

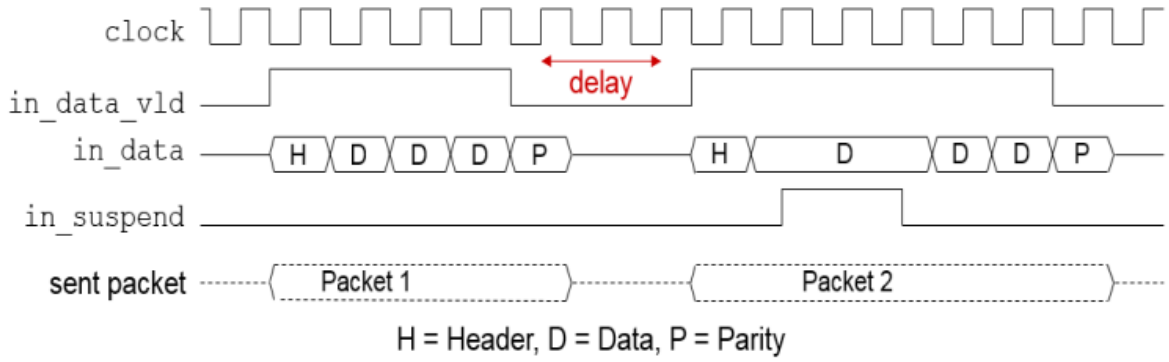
A packet can have a minimum payload size of 1 byte and a maximum size of 63 bytes.
The parity should be a byte of even, bitwise parity, calculated over the header and payload bytes of the packet.



Input Port Protocol

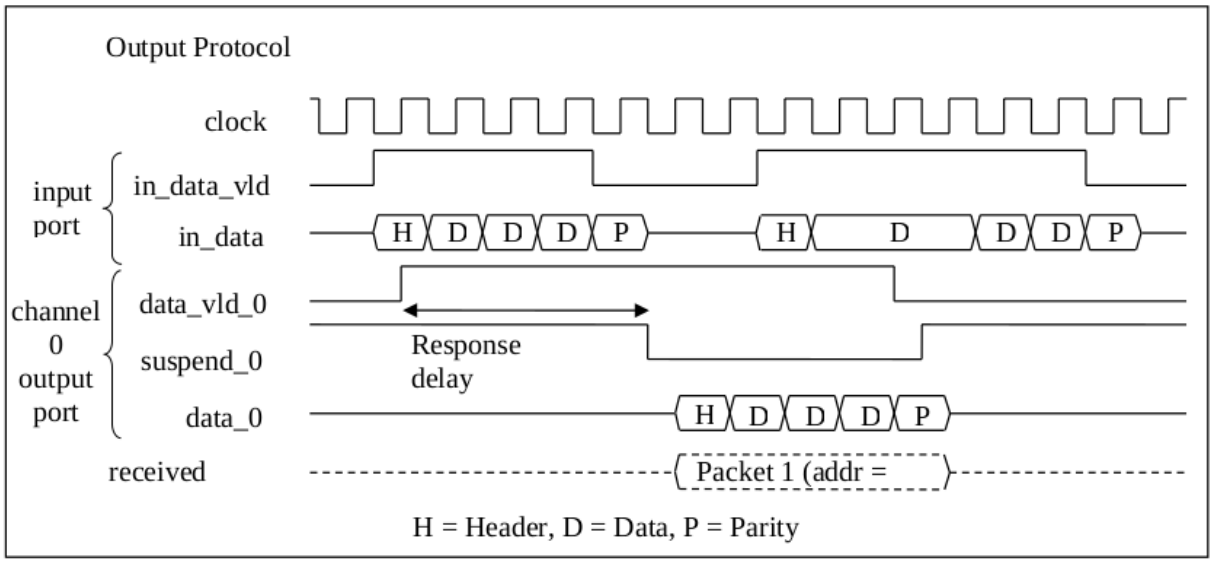
All input signals are active high and are to be driven on the *falling* edge of the clock. The `in_data_vld` signal must be asserted on the same clock when the first byte of a packet (the header byte) is driven onto the `in_data` bus. As the header byte contains the address, this tells the router to which output channel the packet needs to be routed. Each subsequent byte of data needs to be driven on the data bus with each new falling clock.

After the last payload byte has been driven, on the next falling clock, the `in_data_vld` signal must be de-asserted, and the packet parity byte needs to be driven. The input data cannot change while `in_suspend` signal is active (indicating FIFO full). The `error` signal asserts when a packet with bad parity is detected, within 1 to 10 cycles.



Output Port Protocol (Channel Ports)

All output signals are active high and are to be sampled on the *falling* edge of the clock. Each output port is internally buffered by a FIFO of depth 16 and a width of 1 byte. The router asserts the `data_vld_x` signal when valid data appears on the `data_x` output bus. The `suspend_x` input signal must then be de-asserted on the falling clock edge in which data is read from the `data_x` bus. As long as the `suspend_x` signal remains inactive, the `data_x` bus drives a new valid packet byte on each rising clock edge.



Packet Router DUT Registers

The packet router contains internal registers that hold configuration information. These registers are accessed through the host interface port. Register characteristics are as follows:

address	register	reset	field	field name	policy	description
0x1000	ctrl_reg	0x3f	5:0	maxpktsize	RW	Maximum packet length
			7:6		RW	Unused
0x1001	en_reg	0x01	0	router_en	RW	Router enable
			1	parity_err_cnt_en	RW	Parity error count enable
			2	oversized_pkt_cnt_en	RW	Length error count enable
			3	[reserved]	RW	Not implemented
			4	addr0_cnt_en	RW	Address 0 packet count enable
			5	addr1_cnt_en	RW	Address 1 packet count enable
			6	addr2_cnt_en	RW	Address 2 packet count enable
			7	addr3_cnt_en	RW	Address 3 packet count enable
0x1004	parity_err_cnt_reg	0x00	7:0		RO	Packet parity error count
0x1005	oversized_pkt_cnt_reg	0x00	7:0		RO	Packet length error count
0x1006	addr3_cnt_reg	0x00	7:0		RO	Address 3 packet count
0x1009	addr0_cnt_reg	0x00	7:0		RO	Address 0 packet count
0x100a	addr1_cnt_reg	0x00	7:0		RO	Address 1 packet count
0x100b	addr2_cnt_reg	0x00	7:0		RO	Address 2 packet count

If the input packet length is greater than the `maxpktsize` field of the `ctrl_reg` register, then the router drops the entire packet.

The `router_en` field of the `en_reg` register controls the enabling and disabling of the router. A disabled router drops all packets. Enabling or disabling the router during packet transmission will yield to unpredictable behavior.

The router counters are enabled by individual bits in `en_reg`. The router specification says that if these bits are changed while the router is processing a packet, then the router behavior is undefined.

The router counters are defined as follows:

`parity_err_cnt_reg` – incremented when a bad parity packet is received

`oversized_pkt_cnt_reg` – incremented when packet with length greater than `maxpktsize` is received

`addr3_cnt_reg` – incremented when a packet with an illegal address (3) is received

`addr0_cnt_reg` – incremented when a packet with address 0 is received

`addr1_cnt_reg` – incremented when a packet with address 1 is received

`addr2_cnt_reg` – incremented when a packet with address 2 is received

Router Memories

The router contains two memory blocks as follows:

Start Address	Name	Size	Policy	Description
0x1010	yapp_pkt_mem	[0:63]	RO	Stores the bytes of the last packet received by the yapp router.
0x1100	yapp_mem	[0:255]	RW	“Scratch” memory

Host Interface Port Protocol (HBUS)

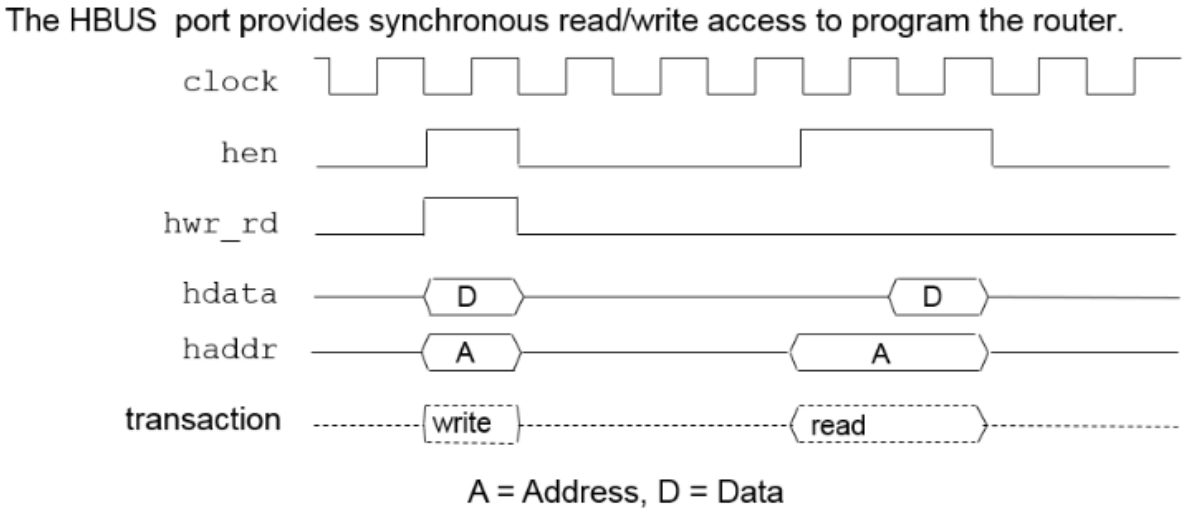
All input signals are active high and are to be driven on the *falling* edge of the clock. The host port provides synchronous read/write access to program the router.

A WRITE operation takes one clock cycle as follows:

- `hwr_rd` and `hen` must be 1. Data on `hdata` is then clocked on the next rising clock edge into the register based on `haddr` decode.
- `hen` is driven to 0 in the next cycle.

A READ operation takes two clock cycles as follows:

- `hwr_rd` must be 0 and `hen` must be 1. In the first clock cycle, `haddr` is sampled and `hdata` is driven by the design under test (DUT) in the second clock cycle.
- `hen` is then driven low after cycle 2 ends. This will cause the DUT to tri-state the `hdata` bus.



Note:

File names, component names, and instance names are suggested for many of the labs. You are not required to use these names but if you do not, you may need to edit code provided to match your path and instance names.

These labs do not include step-by-step instructions, and do not tell you exactly what you need to type.

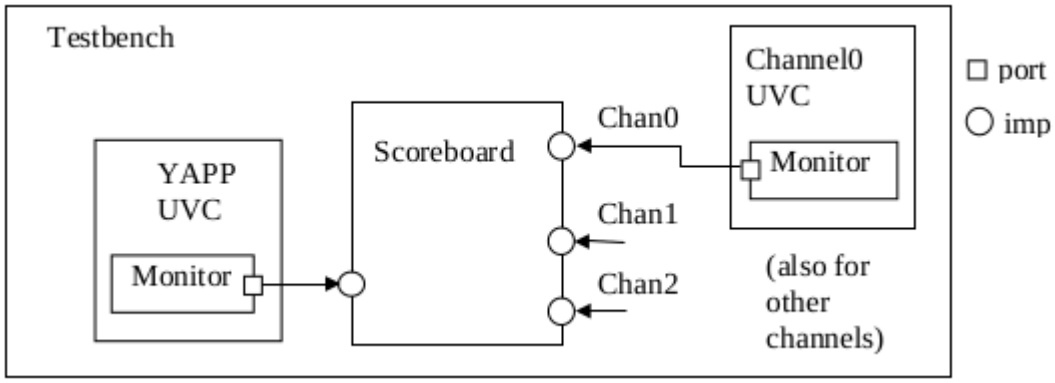
Assignment 2: Creating a Scoreboard Using TLM and creating Router Module UVC

Objective: To build a scoreboard using TLM imp connectors and to create a module UVC for the router using the scoreboard.

For this assignment, you will build and connect a scoreboard for the router, and create TLM analysis port connections to hook up the scoreboard to the UVCs.

The router Module UVC is a complex design, so this assignment has been deliberately broken down into separate steps to build the UVC progressively.

The first step is to implement the scoreboard component itself and connect it up to the YAPP and Channel UVCs. For this part of the assignment we assume all packets are sent to legal addresses with legal payload length, i.e. the router does **not** drop any packets.



1. Copy the files from `part2_mcseq/tb` into `assignment2_sba/tb`. Work in the `assignment2_sba/tb` directory.

2. A TLM analysis port has already been implemented in the Channel UVC monitor for collected YAPP packets. This port is named `item_collected_port`. Check the Channel monitor in the file `channel/sv/channel_rx_monitor.sv` to make sure you understand how this port is used.

3. Modify `yapp/sv/yapp_tx_monitor.sv` to create an analysis port instance.

   a. Declare an analysis port object, parameterized to the correct type.

   b. Construct the analysis port in the monitor constructor.

   c. Call the port `write()` at the appropriate point.

4. In the `assignment2_sba/sv` directory, create the scoreboard, `router_scoreboard.sv`.

   a. Extend from `uvm_scoreboard` and add a component utility macro and a

constructor.

b. As the YAPP and Channel UVCs use different packet types, you will need a custom comparison function to compare `yapp_packet` and `channel_packet` packets.

   You can use either simple Verilog comparison operators or the `uvm_comparer` class (see slides and reference material for details).

   A simple comparison operator is provided to you in the file `packet_compare.sv`, copy this into your scoreboard.

c. Define four analysis `imp` objects (for the YAPP and three Channels) using `` `uvm_analysis_imp_decl `` macros and `uvm_analysis_imp_*` objects.

d. Create analysis `imp` instances in the scoreboard constructor.

e. Use the YAPP `write()` implementation to clone the packet and then push the packet to a queue. *Hint:* Use a queue for each address.

f. Use Channel `write()` implementations to pop packets from the appropriate queue and compare them to the channel packets, using your custom comparer function.

g. Add counters for the number of packets received, wrong packets (compare failed) and matched packets (compare passed).

h. Add a `report_phase()` method to print the number of packets received, wrong packets, matched packets and number of packets left in the queues at the end of simulation.

5. In the `tb` directory, update the `tb_top` module to include the router scoreboard.

6. In the `tb` directory, modify the `router_tb.sv` as follows:

   a. Declare and build the scoreboard.

   b. Make the TLM connections between YAPP, Channel, and scoreboard.

   c. Use a test which generates a good number of legal YAPP packets, i.e. short packets with legal addresses, so that no packets are dropped by the router. We could use the multichannel sequence test or you could modify the exhaustive sequence test to add the Channel and Clock and Reset sequences.

   d. Check that the simulation results are correct, and debug as required.

7. Once you are happy that the scoreboard is working, check that it correctly reports mismatched packets. You can achieve this by commenting out the short YAPP packet type override in your test class which will allow the YAPP UVC to send oversized packets which will be dropped by the router and create mismatches in the scoreboard.

Make sure that your HBUS sequencer is executing `hbus_small_packet_seq` to set the `MAXPKTSIZE` register to 20.

Run a simulation with the type override removed and check the log file for the following:

a. Your YAPP UVC is generating packets of type `yapp_packet`.

b. The RTL router code generates `ROUTER DROPS PACKET` messages.

c. Your custom comparer function generates `uvm_error` messages when the comparison fails.

d. Packets are left in the scoreboard queues at the end of simulation.

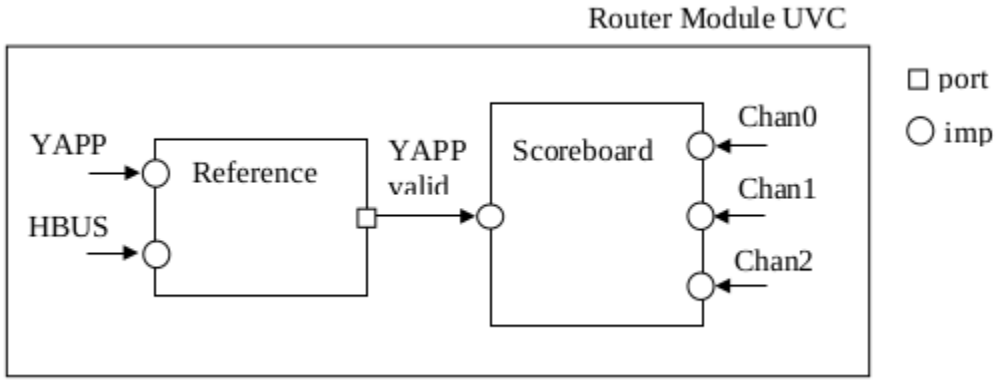
e. Your scoreboard reports the number of received, mismatched and matched packets, as well as the number of packets left in the queues. Check the number of packets add up!

8. (Optional) Write your own custom comparer function which uses `uvm_comparer` methods instead of Verilog comparisons. Test your new implementation in simulation.

In reality, the scoreboard will only be one part of a larger router module UVC. For example, the router UVC may also contain reference models and coverage. All these components will be enclosed in an env class.

In our example, we need to know the `maxpktsize` and `router_en` register field settings so we know which packets are dropped. We can implement this in a separate router reference model component.

The router reference model connects to the YAPP and HBUS UVC analysis ports and selectively passes on YAPP input packets to the scoreboard depending on the register settings. An `env` wrapper will instantiate both reference and scoreboard into a single router module UVC, as shown.



Copy the files from `assignment2_sba` into `assignment2_sbb`. Work in the `assignment2_sbb` directory.

1. A TLM analysis port has already been implemented in the HBUS UVC monitor. Note that the HBUS UVC has a common monitor, `hbus_monitor.sv`, for both the master and slave agents. The HBUS monitor analysis port for collected `hbus_transaction`'s is named `item_collected_port`.

   Check this to make sure you understand how it is written.

2. Create the router reference, `router_reference.sv`, in the sv directory.

   a. Extend from `uvm_component`.

   b. Define two analysis `imp` objects for the YAPP and HBUS monitor analysis ports, using `` `uvm_analysis_imp_decl `` macros and `uvm_analysis_imp_*` objects. (Copy declarations from your scoreboard.) These are for input data to the reference.

   c. Define one analysis port object for the valid YAPP packets. This is for output data to the scoreboard.

   d. Define variables to mirror the `maxpktsize` and `router_en` register fields of the router and update these in the HBUS `write()` implementation.

   e. In your YAPP `write()` implementation, forward the YAPP packets onto the scoreboard only if the packet is valid (router enabled; `maxpktsize` not exceeded; address valid). Keep a separate count of invalid packets dropped due to size, enable and address violations.

3. Create the router module environment, `router_module_env.sv`, in the sv directory.

   a. Declare and build the scoreboard and router reference components.

   b. Connect the "valid YAPP" analysis port of the reference model to the YAPP analysis imp of the scoreboard model.

4. In the `tb` directory, modify the `router_tb.sv`.

   a. Replace the scoreboard declaration and build with the router module.

   b. Modify the TLM connections for the YAPP and Channel analysis ports to allow for the `router_env` layer.

   c. Add a connection for the HBUS analysis port.

5. Create a `router_module.sv` package which includes the router module environment, reference and scoreboard files. Import this package into your UVM top module.

a. Use the same multichannel sequences to test your scoreboard and system monitor implementation.

b. Check the simulation results are correct and the scoreboard and monitor report the right number of packets.

6. The router module can now be used as a standalone UVC. Copy your router module UVC files to the `router` directory.